Herramienta de monitorización del rendimiento aplicada al Pentium 4

LUIS MARÍA MARTÍNEZ GARCÍA
DAVID FERNÁNDEZ CASTRO
Proyecto Sistemas Informáticos
Ingeniería en Informática - UCM

Contenido

Índice de figuras

Introducción al hardware para monitorización del rendimiento

CONTENIDO

1. Introducción

Actualmente, la mayoría de los procesadores que incorporan los sistemas con los que trabajamos poseen hardware *on-chip* para poder analizar o monitorizar el rendimiento del propio procesador. Rendimiento es sinónimo de un uso efectivo de la jerarquía de memoria, saltos bien predichos, pocas paradas en el pipeline, etcétera. Y en particular para cada programa, por ejemplo podría ser poco deseable un elevado número de operaciones en punto flotante normales y esperar completar más instrucciones SSE.
En definitiva, los datos recolectados por este hardware específico nos proporcionan información sobre el comportamiento de una determinada aplicación, del sistema operativo o del propio procesador. Esta información, debidamente analizada y tratada, puede ser la clave para descubrir aplicaciones o secuencias de código con bajo rendimiento y puede ser una guía que dirija los esfuerzos en la dirección adecuada para optimizar los algoritmos.

Los análisis obtenidos al monitorizar la eficiencia de los programas ayudan no sólo a optimizar el código de las aplicaciones y sistemas operativos sino también a mejorar los compiladores y a pensar en futuros diseños de procesadores. Sin embargo, veremos que el soporte actual que dan muchos procesadores a la monitorización de rendimiento es limitado.
Como veremos, el procesador Intel® Pentium® 4 incorpora muchas mejoras en este aspecto y consigue eliminar muchas de estas limitaciones.

El hardware para la monitorización del rendimiento consta, entre otros recursos, de una serie de registros contadores que cuentan el número de veces que se producen determinados *eventos* en el procesador a lo largo de la ejecución de un código, a los que llamamos *eventos de rendimiento*.

El número de contadores disponibles en el procesador se ha incrementado notablemente en los procesadores modernos, pasando de procesadores que tan sólo incorporaban dos contadores a los 18 contadores que tiene el Intel® Pentium® 4, haciendo más flexible el trabajo de análisis del rendimiento.

Procesadores como el Pentium, Athlon, Alpha, Cray, UltraSparc, PowerPC, Itanium y muchos más poseen hardware para monitorización del rendimiento.

2. Eventos de rendimiento

Como hemos indicado, la monitorización del rendimiento de un determinado programa o aplicación se basa en la medición de eventos, de eventos de rendimiento.

2.1. Clasificación de eventos

Los eventos de rendimiento pueden clasificarse, por ejemplo, en cinco grupos:

Caracterización o naturaleza del programa

Ejemplos típicos de estos tipos de eventos son número de instrucciones o tipo de instrucciones (*loads, stores,* saltos, operaciones en punto flotante, …) completadas a lo largo del programa a monitorizar.

Accesos a memoria

Suele ser la categoría más amplia por tratar una multitud de eventos relacionados con toda la jerarquía de memoria del sistema. Algunos ejemplos son referencias a una determinada caché o fallos en una determinada caché.

Paradas del pipeline

Los eventos que pertenecen a esta categoría ayudan a analizar el comportamiento del flujo de instrucciones a través del pipeline del procesador.

Predicción de saltos

Estos eventos tienen como finalidad permitir un análisis del rendimiento que ofrece el hardware para la predicción de de saltos. Por ejemplo, un evento puede ser el número de saltos tomados que han sido predichos como no tomados por el hardware de predicción de saltos.

Utilización de recursos

Permite monitorizar el uso que el procesador hace de determinados recursos. Por ejemplo, podríamos tener un eventos que contara el número de ciclos que una determinada unidad funcional está siendo usada por el procesador.

2.2. Medidas *non-retirement* vs. medidas *at-retirement*

Clasificando los eventos según la instrucción que lo produce, podemos distinguir dos tipos de eventos:

Eventos generados por instrucciones que se retiran

Estos eventos los producen instrucciones que alcanzan la fase de *commit* y se retiran satisfactoriamente. Independientemente de

que la instrucción que produce la aparición del evento haya sido especulada o no, lo importante es que la instrucción está en la rama correcta de ejecución y que, por tanto, la aparición del evento sigue el flujo normal del programa.

Eventos generados por instrucciones que no se retiran nunca

Estos eventos los produce una instrucción que está en una rama especulativa y que, finalmente, no se retira. Sin embargo, el evento (que sí se produce y se puede contar) no debería haberse producido siguiendo el flujo normal del programa. La aparición de estos eventos se debe, normalmente, a errores en el cálculo de la predicción de saltos.

Las instrucciones que no se retiran pueden influir mucho en la medida de un evento, incluso pudiendo llegar a superar a las medidas producidas por instrucciones que se retiran. Usar esta medida global en vez de separar las medidas producidas por uno u otro tipo de instrucciones puede llevar a análisis erróneos del rendimiento del programa.

Por esto, en algunas circunstancias interesa medir solamente el primer grupo de eventos, evitando contar la aparición de eventos producidos por instrucciones que no se retiran. Algunos procesadores utilizan mecanismos para dar soporte a este tipo de medidas, las llamadas medidas *at-retirement*. Por otro lado, las medidas *non-retirement* se refieren a las medidas de eventos independientemente de si la instrucción que lo produce vaya a alcanzar la fase *commit* o se tenga que anular por tratarse de una instrucción especulada erróneamente.

3. Hardware para monitorización del rendimiento

El soporte hardware necesario para poder monitorizar y analizar el rendimiento del procesador está incorporado (*on-chip*) en el propio procesador.
Este hardware consta, principalmente, de contadores y detectores de eventos. De esta manera, una correcta configuración de ambos recursos permite obtener en los contadores el número de veces que ha sucedido un determinado evento bajo unas determinadas condiciones deseadas.

3.1. Detectores de eventos

Los detectores de eventos son el hardware encargado de detectar la aparición u ocurrencia de los eventos para los que está configurado que detecte. Cada detector suele incluir un registro donde, mediante la activación de determinados bits, indicamos que evento queremos detectar y bajo qué condiciones.

3.1.1. Clases de eventos y tipos de evento

En los detectores de eventos el usuario indica qué clase de eventos (*decisión de grano grueso*) se va a considerar, a medir, de entre todos los eventos definidos. A menudo, estos detectores poseen un campo de máscara (*mask*) donde se especifican los *tipos de evento* o sub-eventos (*decisión de grano fino*) que se van a medir.
Es decir, normalmente una clase de eventos no es un evento concreto sino que alberga una serie de tipos de evento que son los que finalmente se contabilizan.
A modo de ejemplo, indicamos un par de casos concretos de clase de eventos con sus tipos de evento asociados, tomados del procesador Intel® Pentium® 4:

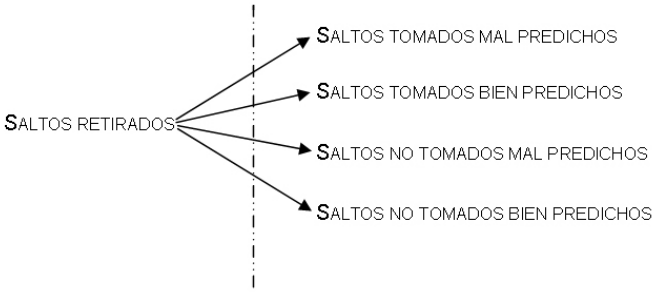

Figura 1. Ejemplo de sub-eventos para la clase de eventos *saltos retirados*.

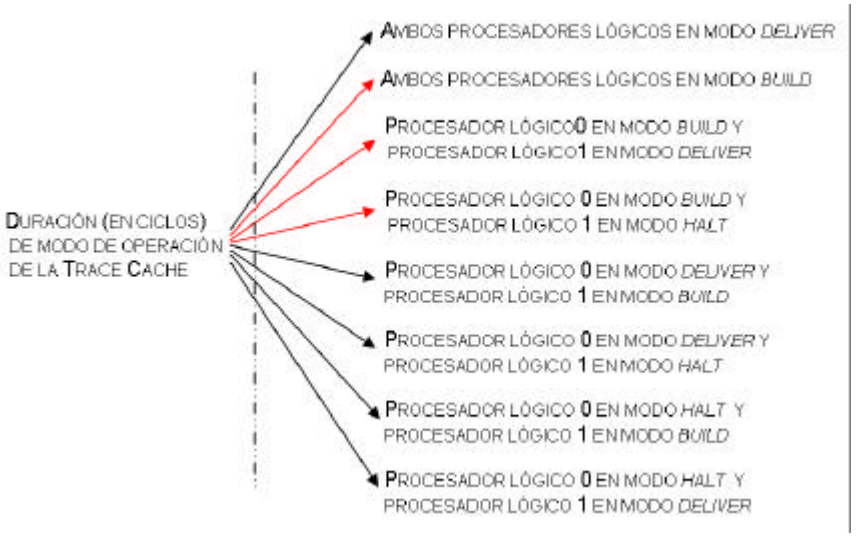

Figura 2. Ejemplo de sub-eventos para una clase de eventos de Intel® Pentium® 4.

Podemos seleccionar varios tipos de evento simultáneamente. Por ejemplo, para obtener el número de ciclos de reloj en los que el procesador lógico 0 de nuestro procesador Intel® Pentium® 4 está en modo *build* tendremos que configurar la máscara para medir tres sub-eventos o tipos de evento (los indicados mediante una flecha roja).

3.1.2. Filtrado de eventos en el detector

En los detectores de eventos también podemos filtrar los eventos detectados, considerando sólo las detecciones del evento que cumplan ciertas condiciones. Podemos discriminar entre eventos producidos en modo privilegiado (sistema operativo) o en modo usuario (aplicaciones de usuario). Esto nos permite tres opciones, medir eventos generados por código del sistema operativo, de una aplicación de usuario o contabilizar ambos sin hacer distinción. A esto le llamamos *filtrado por eventos de usuario/SO*.
En los procesadores con soporte SMT (***S**ymmetric **M**ulti-**T**hreading*) también se puede hacer un *filtrado por procesador lógico* que discrimina entre eventos producidos por instrucciones ejecutadas en un procesador lógico o en otro.

3.2. Contadores de eventos

Los contadores de eventos están conectados a un detector de eventos a través de una serie de líneas. En cada ciclo de reloj, el detector de eventos notifica a un contador, a través de estas líneas, el número de veces que se ha detectado el evento, habiendo pasado ya por el filtrado en el detector.
Dado que este número de líneas es finito, un contador de eventos sólo puede incrementarse en cada ciclo de reloj hasta 2^n-1 unidades, siendo n el número de bits de estas líneas.
Por norma general, el contador se incrementará en tantas unidades como el detector de eventos le indique, aunque mediante el *filtrado de eventos en el contador* podemos modificar este comportamiento.

El filtrado de eventos en el contador es una característica del contador de eventos que sirve para incrementar el contador solamente bajo ciertas condiciones.
Veamos qué tipos de filtrado se pueden hacer en el contador.

3.2.1. Filtrado de eventos en el contador

Al igual que en el detector, podemos realizar un filtrado de eventos en el contador, permitiendo un mayor control sobre qué es lo que realmente queremos contar. Veamos algunos ejemplos:

Comparación con valor umbral

Esta característica del filtrado de eventos permite, en cada ciclo, incrementar el contador en función de la comparación entre el valor recibido del detector de eventos y un valor umbral (*threshold value*) fijado por el usuario. El contador se incrementará en una unidad solamente si el valor recibido del detector es mayor que el valor umbral utilizado (o viceversa, es decir, menor o igual). La decisión de si se incrementa cuando el valor recibido es mayor que el umbral o cuando es menor o igual que el umbral se configura también en el contador.
La comparación con umbral sólo es útil para eventos que pueden reportarse más de una vez por ciclo.
Diremos que una comparación es satisfactoria si dicha comparación conlleva un incremento en el contador. En otro caso, diremos que la comparación no ha sido satisfactoria.

Por ejemplo, para saber cuántos ciclos ha habido en los que se han completado más de 2 instrucciones utilizamos este mecanismo con un valor umbral igual a 2 y utilizando comparación *medida tomada* > *umbral*.

Detección de flancos

La detección de flancos (*edge detection*) es otro mecanismo del contador para filtrar eventos. Podemos utilizar detección de flancos de subida o detección de flancos de bajada.
Cuando la detección de flancos de subida (bajada) está activada, el contador se incrementará en una unidad solamente en el caso en que el valor que reciba el contador del detector de eventos sea mayor (menor o igual) que en el ciclo anterior.

Vamos a explicar esta característica mediante un ejemplo. Supongamos que nos interesa saber cómo se está comportando el pipeline, concretamente queremos saber cuántas paradas en el pipeline se producen durante la ejecución de un determinado algoritmo. Sin embargo, lo que realmente queremos saber no es el número de ciclos en los que el pipeline está detenido sino el número de veces que se produce una parada en el pipeline (sean paradas largas de varios ciclos o de un sólo ciclo), es decir, nos

interesa el número de veces que el pipeline pasa de no estar detenido a estar detenido.
Por lo tanto, en este caso haríamos uso de un filtrado de eventos con detección de flancos de subida.

Filtrado por estado de los procesadores lógicos

Este tipo de filtrado sólo tiene importancia en procesadores que utilicen SMT, es decir, que implementen más de un procesador lógico.
Ese filtrado consiste en incrementar el valor del contador solamente bajo ciertas condiciones referentes a los estados de los procesadores lógicos.
Por ejemplo, podríamos incrementar el contador solamente si un procesador lógico está activo y el otro no lo está.

3.2.2. Configuración de contadores en cascada

Algunos procesadores permiten la configuración de contadores en cascada. Esto permite correlacionar eventos, permitiendo la contabilización de un cierto tipo de eventos después de haber contabilizado un número determinado de veces la ocurrencia de otro evento.
Cuando dos contadores se configuran en cascada, el primero empieza a contar hasta que se desborda, momento en el cual empieza a contar el segundo contador. Si por ejemplo inicializamos el primer contador para que se desborde tras la cuenta de N eventos e inicializamos el segundo para que se desborde tras la cuenta de M eventos y que genere una interrupción PMI al desbordarse, obtendremos información detallada sobre la correlación de ambos eventos.

4. Perfiles de rendimiento

Mediante el uso de detectores de eventos y contadores se puede detectar la presencia de problemas de rendimiento en un determinado código o algoritmo. Sin embargo, sería útil conocer que fragmento del código (aplicación o sistema operativo) o qué instrucciones están siendo las causantes de ese bajo rendimiento. Con este conocimiento, un programador podría centrar sus esfuerzos en modificar el código de alto nivel de los algoritmos que utiliza la aplicación o realizar otros cambios a bajo nivel a fin de reducir el impacto del problema, mejorando el rendimiento global de la aplicación.

Para obtener la información que nos permita identificar la causa del bajo rendimiento, los contadores crean *perfiles* que, una vez analizados, pueden indicarnos la localización más o menos exacta de las instrucciones responsables del bajo rendimiento de la aplicación.

4.1. Perfiles basados en tiempo

Una técnica común para encontrar las zonas en las que debemos dirigir los esfuerzos para mejorar el rendimiento es obteniendo un *perfil basado en tiempo* de la aplicación analizada.
Básicamente, un perfil basado en tiempo identifica las partes del código en las que la aplicación está pasando la mayoría del tiempo.
Para obtener un perfil basado en tiempo, lo que se hace es interrumpir la ejecución de la aplicación cada ciertos intervalos de tiempo regulares. En cada interrupción, el manejador de la interrupción guarda el valor del contador de programa. Así, al final de la ejecución se obtiene un perfil a modo histograma que contiene todas las muestras tomadas para cada valor de contador de programa. De este modo, se espera que el perfil indique las partes del código o instrucciones (valor del contador de programa) que más se han ejecutado.

4.2. Perfiles basados en eventos

Los *perfiles basados en eventos* tratan de construir un histograma que represente el número de ocurrencias de determinados eventos de rendimiento en función de la localización en el código. En vez de interrumpir la ejecución de la aplicación cada cierto tiempo, el hardware de monitorización de rendimiento interrumpe la aplicación cada *N* ocurrencias de un determinado evento. Por lo tanto en este caso, el perfil pretende encontrar las instrucciones más frecuentemente ejecutadas que causan la aparición de un determinado evento.
Para dar soporte al muestreo basado en eventos o EBS (***E**vent-**B**ased **S**ampling*), el hardware de monitorización de rendimiento genera una interrupción PMI (***P**erformance **M**onitor **I**nterrupt*) cada vez que un contador de rendimiento se desborda (*overflow*).
En estas condiciones, se ha de inicializar el contador de rendimiento que vaya a realizar EBS a N unidades de su valor máximo, para que después de contar N se desborde y genera una interrupción, produciendo una muestra más para

el perfil. Tras el desbordamiento, el contador debe inicializarse de nuevo a su valor máximo menos N unidades.
Las interrupciones se tratan a través de una ISR (***I****nterruption* ***S****ervice* ***R****outine*), que guarda datos del programa (entre otras cosas, el valor del contador de programa) en un fichero que, al final de la ejecución, constituye el perfil basado en eventos.

5. Las posibilidades de la monitorización en la actualidad

5.1. Herramientas software

Una técnica común para encontrar las zonas en las que debemos dirigir los
En la actualidad, hay una gran variedad de herramientas software que sirven como interfaces de alto nivel para explotar las posibilidades de análisis del rendimiento a partir de hardware para monitorización de rendimiento.

La herramienta PAPI (***P**erformance **A**pplication **P**rogramming **I**nterface*) proporciona un interfaz común para el uso de los contadores hardware, dando soporte a multitud de procesadores: Alpha, Athlon, Pentium, Itanium, MIPS, ...

Otra herramienta bien conocida es *VTune Performance Analyzer*, de Intel®, que da soporte para todos los procesadores Pentium e Itanium.

Rabbit Performance Counter Library es otro software para procesadores Pentium y AMD bajo sistemas Linux.

Por último, *Brink & Abyss* es el nombre de otra herramienta software, un interfaz de alto nivel para gestionar la monitorización del rendimiento en procesadores Intel® Pentium® 4 en sistemas Linux. Será la herramienta que analizaremos en el presente trabajo.

5.2. Soporte hardware

Los procesadores Intel® de la familia P5 y P6 sólo contienen un par de contadores de rendimiento. Soportan generación de interrupciones tras desbordamiento de contadores, comparación con umbral y distinción entre eventos de usuario y de sistema operativo.
Por ejemplo, el procesador Intel® Pentium® III define más de 80 eventos que se pueden medir.

Los procesadores Athlon de AMD incorporan cuatro contadores de rendimiento, con características similares a las del Intel® Pentium® III. Sin embargo, los Athlon sólo definen alrededor de 25 eventos.

El PowerPC 750 de IBM también incorpora cuatro contadores, con más de 40 eventos definidos. El modelo PowerPC 7450 de Motorola posee un total de seis contadores y define hasta 200 eventos de rendimiento.
Una característica interesante de los PowerPC es que permiten, porque dan soporte hardware, medir sólo los eventos que generan determinados procesos bajo un sistema multitarea.

El procesador Intel® Itanium® tiene cuatro contadores hardware y alrededor de 90 eventos definidos. Estos procesadores soportan multitud de funciones y características adicionales.

Los UltraSparc I y II de Sun poseen tan sólo dos contadores de rendimiento, y permiten medir eventos de usuario y de sistema operativo sobre unos 20 eventos definidos. Tienen posibilidades limitadas.

El Alpha 21264 de Compaq también posee dos contadores. Fue el primer procesador en dar soporte a la técnica de generación de perfiles *ProfileMe*.

El Intel® Pentium® 4, objetivo de nuestro trabajo, proporciona un gran soporte para la monitorización del rendimiento. Incorpora hasta 18 contadores hardware y una gran cantidad de eventos que se pueden medir. Además, soporta diversos mecanismos para conseguir medidas más útiles y precisas.

El procesador Intel® Pentium® Xeon fue el primero en dar soporte a la monitorización del rendimiento para SMT (***S**imultaneous **M**ulti-**T**hreading*), pudiendo realizar nuevos tipos de filtrado.

5.3. Limitaciones

En la actualidad, las principales limitaciones asociadas al uso de mecanismos de monitorización de rendimiento son las siguientes:

Escasos contadores hardware

Con alguna pequeña excepción, la mayoría de procesadores en la actualidad contienen de dos a seis contadores. Esto significa que se pueden medir muy pocos eventos concurrentemente, siendo necesario ejecutar el programa muchas veces para poder medir unos pocos eventos.

Medidas at-retirement

Las medidas *at-retirement*, analizadas anteriormente, conllevan una complejidad que hay que abordar para poder dar soporte a esta característica.
El Intel® Pentium® 4, como veremos, utiliza *mecanismos de etiquetado* para poder hacer medidas *at-retirement*.

Imprecisión de mecanismos EBS

Tal vez la mayor limitación sea la poca precisión de algunos mecanismos de EBS usados por los procesadores para generar perfiles de rendimiento. Por esto, a algunos de estos mecanismos se les llama mecanismos de IEBS (***I**mprecise **E**vent-**B**ased **S**ampling*).
El problema se presenta cuando se genera la interrupción PMI tras el desbordamiento del contador. En muchos casos, la rutina ISR no recoge la dirección de la instrucción que ha generado el evento y el consiguiente desbordamiento del contador, sino que se produce un retraso o desplazamiento y lo que recoge es la dirección de una instrucción distinta, en algunos (sólo algunos) casos cercana a la deseada. Este notable error de precisión se debe al intervalo de tiempo que transcurre entre el desbordamiento del contador y la señalización de la interrupción.

En este "pequeño" intervalo de tiempo pueden retirarse instrucciones, lo que hace que el contador de programa ya no apunte exactamente a la instrucción que ha generado el evento. Hay sistemas que implementan mecanismos más precisos de generación de perfiles basados en eventos. A estos mecanismos de EBS se les suele llamar PEBS (***P****recise* ***E****vent-****B****ased* ***S****ampling*).

Falta de perfiles de direcciones de datos

Hay una limitación adicional referente a la generación de archivos de perfiles de rendimiento que padecen muchos procesadores. Así como un perfil que contiene direcciones de instrucciones es útil para identificar problemas de rendimiento en el código de un programa, también puede ser interesante recoger direcciones de datos que han causado problemas de rendimiento. Por ejemplo, puede ocurrir que a lo largo de la ejecución de un programa los accesos a una determinada dirección de memoria estén ocasionando muchos fallos de caché. Podría ser interesante conocer el valor de dicha dirección de memoria tan problemática. La limitación surge del hecho de que, en muchas ocasiones, el hardware no permite tomar muestras de las direcciones de datos que están produciendo estos problemas de rendimiento.
Para dar soporte a esta característica, el procesador debe capturar muestras de estas direcciones o ser capaz de almacenar suficiente información de estado (registros, ...) y computar estas direcciones.

6. Motivación del proyecto

El motivo de nuestro proyecto ha sido el proporcionar una herramienta útil en la medición de eventos para el procesador Intel® Pentium® 4 bajo Linux. La razón se debe a que no existía ninguna herramienta con las características que requeríamos. Había dos aplicaciones que servían para medir eventos del Intel® Pentium® 4 pero ambas adolecían de carencias que exponemos a continuación.

Brink & Abyss es una aplicación muy flexible y que se diseñó pensando en hacer una interfaz de alto nivel en XML para abstraer la complejidad inherente a la lógica del procesador. También proporciona una manera de modificar el funcionamiento del programa (añadir nuevos eventos) sin tener que recompilarlo. Esto es útil dado, con el tiempo, Intel® documenta nuevas métricas para su procesador. En definitiva, se trata de un software muy versátil, pero tiene sus problemas. El primero es que sólo permite medir ejecuciones de programas completos y no partes aisladas de un código que queramos optimizar. Otro problema surge en la falta de documentación.

Perfctr resuelve a través de sus librerías el problema de medir una sección de nuestro código pero una vez mas nos surgen problemas ya que hay que conocer a fondo la lógica del Intel® Pentium® 4 para poder utilizarlo. Y es que una manera de utilizar esta librería es introduciendo directamente por parámetros el valor de los registros para monitorización en hexadecimal. Otra dificultad radica en la absoluta falta de documentación disponible, prácticamente ninguna.

Por lo tanto nuestro objetivo se ha centrado en conseguir una aplicación que aunase lo mejor de ambas aplicaciones antes descritas. En la última parte de este trabajo explicaremos el modo de funcionamiento de nuestra aplicación.

Arquitectura del Intel® Pentium® 4

Contenido

1. Introducción

En Noviembre del año 2000 y tras una mala época para la compañía Intel® Corporation (INTC), que estaba perdiendo la lucha contra su archienemigo AMD (*Advanced Micro Devices*), aparece el primer procesador Intel® Pentium® 4, el *Willamette*. Y es que tras varios años sin aportar nada nuevo al mundo de la micro-arquitectura, este procesador ha inaugurado una nueva arquitectura que todavía hoy en día sigue funcionando, la micro-arquitectura NetBurst®.
La nueva micro-arquitectura NetBurst® refleja el final de la era de la familia P6 empezada en 1995 con el Intel® Pentium® Pro y utilizada hasta el Intel® Pentium® III.
En este documento nos centraremos principalmente en esta arquitectura, común a todos los procesadores Intel® Pentium® 4 (aunque los últimos modelos, los *Prescott*, presentan importantes variaciones). También analizaremos los distintos aspectos tecnológicos existentes en los distintos modelos disponibles.

Arquitectura IA-32
NetBurst® utiliza el repertorio de instrucciones IA-32 (*32-bit Intel® Arquitecture*), básicamente el mismo que el que se usaba ya en 1978 en el procesador 80x86 de primera generación: el 8086. Aunque el 8086 era una arquitectura de 16 bits, IA-32 se refiere tanto a 32 bits como a 16 bits ya que son compatibles. Esto quiere decir que cualquier aplicación escrita hace unos 25 años para el procesador 8086 podrá ser ejecutada sin problemas en el Intel® Pentium® 4 (y claro está, en cualquier otro procesador Intel® entre medias de estos dos).

La arquitectura IA-32 es una compleja arquitectura CISC con más de 300 instrucciones y con registros de propósito general (GPRs). Con el inicio de la arquitectura P6 (el Intel® Pentium® Pro), Intel® dotó a los procesadores de un núcleo RISC, añadiendo un decodificador CISC/RISC en el *front-end* de la arquitectura. Este decodificador traduce cada instrucción IA-32 (instrucciones CISC, complejas y de longitud variable) en una o varias micro-operaciones o µops (instrucciones RISC, sencillas y de longitud fija) ejecutadas en el núcleo del procesador.

IA-32 es capaz de direccionar, a nivel de byte, hasta un espacio de 64 Gbytes y es de tipo *little-endian*.

IA-32 tiene seis modos de direccionamiento: inmediato, directo por registro, indirecto por registro, indirecto base + desplazamiento, indirecto índice + desplazamiento e indirecto base + índice + desplazamiento.
Existen tres modos de ejecución en IA-32. Son el *modo protegido*, *modo de direccionamiento real* y *modo system management*.

Mantener la compatibilidad con las versiones anteriores, para no perder el privilegiado puesto que Intel® consiguió en el mercado doméstico, supone afrontar una serie de desventajas que lleva a Intel® a partir en desventaja frente a otras alternativas.

Registros IA-32 y extensiones MMX, SSE, SSE2 y SSE3

Se siguen manteniendo sólo 6 registros de propósito general accesibles para el programador (8 registros si contamos el puntero de pila y el puntero base), 6 registros segmentados, un puntero a la instrucción y un registro de flags de estado.

Estos registros eran de 16 bits en el 8086. Con el 80386, la mayoría pasaron a tener 32 bits.

Sin embargo, desde la arquitectura Intel386™ la primera gran modificación al repertorio de instrucciones lo produjo la tecnología MMX™ (***M****ulti****M****edia e****X****tensions*). El procesador Intel® Pentium® MMX fue el primero en introducir estas extensiones que añadían 8 registros de 64 bits (*mm0 – mm7*) a la arquitectura y 57 nuevas instrucciones para operaciones con enteros.

La continuación de MMX™ tuvo lugar en el procesador Intel® Pentium® III con la inclusión de las nuevas instrucciones SSE (***S****treaming* ***S****IMD* ***E****xtensions*), creadas específicamente para 3D. Estas extensiones añadieron 8 nuevos registros de 128 bits (*xmm0 – xmm7*) y 70 nuevas instrucciones para operar sobre números en coma flotante de 32 bits.

El procesador Intel® Pentium® 4 añade ahora el repertorio SSE2 (***S****treaming* ***S****IMD* ***E****xtensions* ***2***), que son 144 nuevas instrucciones para operaciones de enteros de 128 bits y para coma flotante de 64 bits.

Micro-arquitectura NetBurst®

Los procesadores Intel® Pentium® 4 e Intel® Xeon® fueron los primeros en utilizar la micro-arquitectura NetBurst®. Las principales misiones de esta nueva arquitectura eran dos: mantener la compatibilidad con aplicaciones anteriores IA-32 (usaran operaciones SIMD o no) y aumentar eficazmente la frecuencia de reloj. Más adelante detallaremos las características de esta arquitectura.

2. Procesadores Intel® Pentium® 4

Los procesadores Intel® Pentium® 4, de 32 bits, han evolucionado mucho desde los primeros modelos (*Willamette*) hasta los últimos (los recientes *Prescott*). Por ejemplo, casi se ha triplicado la frecuencia del procesador, se ha doblado la velocidad del bus, se ha cuadriplicado el tamaño de la caché de segundo nivel (L2), se han añadido nuevas tecnologías (como la tecnología HT) y algunos modelos hasta incorporan una caché de tercer nivel (L3). Además, los últimos procesadores Intel® Pentium® 4 tienen casi cuatro veces más transistores que los primeros procesadores. Por si fuera poco, la tecnología de fabricación ha pasado de 0.18 micras a 0.09 micras. Mostramos a continuación una tabla con todos los modelos Intel® Pentium® 4 disponibles hasta la fecha.

Frecuencia	Encapsulado	Tecnología	Bus	Core Stepping	Hyper-Threading
1.3 GHz	Socket 423	Willamette	400 MHz	B2, C1	No
1.4 GHz	Socket 423, 478	Willamette	400 MHz	B2, C1, D0	No
1.5 GHz	Socket 423, 478	Willamette	400 MHz	B2, C1, D0	No
1.6 GHz	Socket 423, 478	Willamette	400 MHz	E0, C1, D0	No
1.6A GHz	Socket 478	Northwood	400 MHz	B0	No
1.7 GHz	Socket 423, 478	Willamette	400 MHz	E0, C1, D0	No
1.8 GHz	Socket 423, 478	Willamette	400 MHz	E0, C1, D0	No
1.8A GHz	Socket 478	Northwood	400 MHz	B0, C1, D1??	No
1.9 GHz	Socket 423, 478	Willamette	400 MHz	E0, D0	No
2.0 GHz	Socket 423, 478	Willamette	400 MHz	D0	No
2.0A GHz	Socket 478	Northwood	400 MHz	B0, C1, D1	No
2.2 GHz	Socket 478	Northwood	400 MHz	B0, C1, D1	No
2.26 GHz	Socket 478	Northwood	533 MHz	B0, C1, D1	No
2.4 GHz	Socket 478	Northwood	400 MHz	B0, C1, D1	No
2.4B GHz	Socket 478	Northwood	533 MHz	B0, C1, D1, M0	No
2.4C GHz	Socket 478	Northwood	800 MHz	D1, M0	Sí
2.4A GHz	Socket 478	Prescott	533 MHz	C0	No
2.5 GHz	Socket 478	Northwood	400 MHz	C1, D1	No
2.53 GHz	Socket 478	Northwood	533 MHz	B0, C1, D1	No
2.6 GHz	Socket 478	Northwood	400 MHz	C1, D1	No
2.6C GHz	Socket 478	Northwood	800 MHz	D1	Sí
2.66 GHz	Socket 478	Northwood	533 MHz	C1, D1	No
2.8 GHz	Socket 478	Northwood	533 MHz	C1, D1	No
2.8C GHz	Socket 478	Northwood	800 MHz	D1, M0	Sí
2.8A GHz	Socket 478	Prescott	533 MHz	C0	No
2.8E GHz	Scoket 478	Prescott	800 MHz	C0	Sí
3 GHz	Socket 478	Northwood	800 MHz	D1	Sí
3E GHz	Socket 478	Prescott	800 MHz	C0	Sí
3.06 GHz	Socket 478	Northwood	533 MHz	C1, D1	Sí
3.2 GHz	Socket 478	Northwood	800 MHz	D1	Sí
3.2E GHz	Socket 478	Prescott	800 MHz	C0	Sí
3.2^{EE} GHz	Socket 478	Northwood	800 MHz		Sí
3.4 GHz	Socket 478	Northwood	800 MHz	D1	Sí
3.4E GHz	Socket 478	Prescott	800 MHz	C0	Sí
3.4^{EE} GHz	Socket 478	Northwood	800 MHz		Sí

EE – Intel Pentium 4 Extreme Edition

2.1. Willamette

Los procesadores Intel® Pentium® 4 de primera generación son los *Willamette*. Estos procesadores de 32 bits aparecen en noviembre del año 2000 (modelo a 1.4 GHz), y son los primeros en utilizar la arquitectura NetBurst®, distanciándose de la arquitectura P6 que Intel® había mantenido con sus procesadores Pentium® Pro, Pentium® II, Pentium® III, Celeron® y Xeon®.

Los procesadores *Willamette* se fabrican con una tecnología de proceso de 0.18 micras y contienen alrededor de 44 millones de transistores. Utilizan interconexiones de aluminio.
Son los únicos procesadores Intel® Pentium® 4 con modelos que utilizan el encapsulado con zócalo de 423 pines, es decir, Socket 423. Sin embargo, todos los modelos del *Willamette* (excepto el modelo a 1.3 GHz) tienen una versión para Socket 478, el que se sigue manteniendo hoy en día.

Los procesadores *Willamette* poseen todas las características de la micro-arquitectura NetBurst®, de la cual hablaremos más abajo.
En cuanto a frecuencia de reloj, hay modelos disponibles desde 1.3 GHz hasta 2.0 GHz, pasando por todos los múltiplos de 100 MHz.
La caché de segundo nivel (L2) que incorporan se trata de una caché unificada de 256 Kbytes.
Willamette utiliza un de 400 MHz (4×100 MHz).
Los últimos *Willamette* aparecieron en agosto del año 2001.

2.2. Northwood

La evolución de los procesadores Intel® Pentium® 4 *Willamette* queda patente a primeros de enero del año 2002 con la aparición de los procesadores *Northwood.*
Todos estos procesadores se fabrican con una tecnología de proceso de 0.13 micras, lo que les permite albergar alrededor de 55 millones de transistores, unos 11 millones más que su antecesor. Este avance permite a los *Northwood* duplicar el tamaño de la caché unificada de segundo nivel (L2).

Además, los *Northwood* disipan menos calor que los *Willamette* debido al menor tamaño de sus componentes.

Evidentemente, los procesadores *Northwood* siguen utilizando la micro-arquitectura NetBurst®.
La gama de estos procesadores se extiende desde 1.6 GHz hasta 3.4 GHz.
Como hemos mencionado arriba, la caché de segundo nivel (L2) que utilizan es una caché unificada de 512 Kbytes.
Respecto al bus del sistema, hay tres tipos de versiones: 400 MHz (4×100 MHz), 533 MHz (4×133 MHz) y 800 MHz (4×200 MHz).

Tecnología Hyper-Threading

Además, los *Northwood* son los primeros procesadores en incluir procesamiento simultáneo multi-hilo (SMT), o lo que Intel® llama tecnología *Hyper-Threading* (HT). Esta tecnología, que ya se ha introducido anteriormente, está disponible sólo en algunos modelos *Northwood* y en los *Prescott*.
Más adelante hablaremos de esta tecnología.

Intel® Pentium® 4 Extreme Edition

A finales del año 2003, Intel® lanza el procesador Intel® Pentium® 4 Extreme Edition (EE). Con este nombre tan sugerente se esconde un procesador *Northwood* con algunas modificaciones estructurales.
En primer lugar, los procesadores Intel® Pentium® 4 Extreme Edition incorporan 178 millones de transistores, bastantes más que los 55 millones de los *Northwood* normales (aunque no nos alarmemos, las nuevas tarjetas gráficas de nVidia, la serie GeForce 6800, posee unos 222 millones).
Este elevado número de transistores se debe a que estos recientes procesadores incorporan una caché de tercer nivel (L3) de 2 Mbytes, convirtiéndolos en algo parecido a un procesador Xeon®.
Los procesadores Intel® Pentium® 4 Extreme Edition sólo están disponibles actualmente en las frecuencias 3.2 EE GHz y 3.4 EE GHz, ambos con bus de 800 MHz e Hyper-Threading activado.

Los procesadores *Northwood* más nuevos, por el momento, han aparecido en febrero del año 2004 y corresponden a los modelos de 3.4 GHz y 3.4 EE GHz.

2.3. Prescott

Con las 20 etapas que tiene el pipeline del *Northwood*, Intel® tiene complicaciones para superar la frecuencia de reloj por encima de los 3.4 GHz. En febrero del año 2004, aparecen los procesadores Intel® Pentium® 4 de tercera generación: los *Prescott*. Estos procesadores están fabricados con la nueva tecnología, de la que tanto alarde ha hecho Intel®, de 90 nanometros (0.09 micras).
Estos procesadores presentan algunas modificaciones en la micro-arquitectura NetBurst® de sus antecesores. Incorporan un nuevo pipeline, ahora con 31 etapas en lugar de las 20 etapas de los anteriores procesadores Intel® Pentium® 4. Así, Intel® seguramente no tendría problemas en ofrecer un *Prescott* a 5 GHz en un futuro próximo.
Otra vez, como supuso al pasar de *Willamette* a *Northwood*, se reduce la tecnología de fabricación y se duplica el tamaño de la caché unificada de segundo nivel (L2), ahora de 1 Mbyte.
Otro cambio importante es que con los *Prescott* aparece el nuevo repertorio de 13 instrucciones SSE3 (Streaming SIMD Extensions 3).
Además, parece ser que con los *Prescott* se ha mejorado la tecnología Hyper-Threading.

A pesar de fabricarse con tecnología de 90 nanometros, los *Prescott* alcanzan una temperatura mucho más alta que los *Northwood.*
Todos los *Prescott* utilizan un bus a 800 MHz.
Ningún *Prescott* tiene caché de tercer nivel (L3) como sucede con los modelos *Northwood* Intel® Pentium® 4 Extreme Edition.

En resumen, con la llegada del *Prescott* no es sólo la tecnología de fabricación y el consiguiente aumento del tamaño de la caché L2 lo único que ha cambiado. Los procesadores *Prescott* incluyen otros cambios estructurales, algo que ha provocado que algunos se aventuraran ha llamarlo Pentium 5. Resumimos estas modificaciones ha continuación:

- Pipeline de 31 etapas.
- Repertorio de instrucciones SSE3 (Streaming SIMD Extensions 3).
- Hyper-Threading mejorado.
- Aumento importante en la disipación de calor.

2.4. Futuros procesadores Intel®

Intel® había dado a conocer que el sucesor del *Prescott* sería el procesador Intel® Pentium® 4 *Tejas*. Este procesador iba a suponer la cuarta generación de procesadores con micro-arquitectura NetBurst®.
Las primeras noticias indicaban que Intel® lanzaría el procesador *Tejas* a finales del año 2004. Después se confirmó que se retrasaría medio año.
Intel® quería introducir varias mejoras en sus *Tejas*:

- Frecuencias a partir de 4.0 GHz o 4.2 GHz.

- Tecnología de 65 nanometros (0.065 micras).

- Nuevo encapsulado de 775 pines (Socket LGA-775).

- Bus mejorado, tal vez a 1200 MHz (4×300 MHz).

- Caché de primer nivel (L1) de 24 Kbytes, y Trace caché de 16 Kμops.

- Caché de segundo nivel de 1 Mbyte (2 Mbytes en 65 nanometros).

- Nuevo repertorio de instrucciones *Tejas New Instructions*.

- Hyper-Threading mejorado.

- Mecanismo de predicción de saltos mejorado.

- Posible aparición de procesadores de 64 bits, o aumento de unidades de ejecución.

Sin embargo, no hace mucho que Intel® ha confirmado que el proyecto *Tejas* ha sido cancelado y que no llegará a ver la luz (ni tampoco el *Jayhawk* para servidores, basado en el propio *Tejas* y evolución del Xeon®). La razón de esta decisión parece que se debe a un problema con el elevado consumo de potencia que requerían dichos procesadores.

Ahora, parece que Intel® tiene como objetivo acelerar la transición a multiprocesadores *on-chip* (*dual-core*) y también a procesadores de 64 bits.
Hay indicaciones de que Intel® partirá de su procesador Pentium® M para diseñar sus nuevos procesadores.
La evolución de los Pentium® M ya tiene un nombre, los procesadores *Jonah*, y servirá tanto para portátiles como para sobremesa. Ya se han dado fechas de aparición entre los años 2005 y 2006.
Jonah tiene previsto incluir dos núcleos *Dotham* (la inmediata evolución del *Banias*, ya disponible para portátiles) usando una tecnología de proceso de 65 nanometros.
Las previsibles evoluciones de *Jonah* serían *Merom*, *Conroe* (2006) y *Gilo* (2007), con una nueva arquitectura extensible a 64 bits.

3. Micro-arquitectura NetBurst®

La micro-arquitectura NetBurst® es el nombre que Intel® ha dado a la arquitectura que poseen sus procesadores Intel® Pentium® 4.

Esta arquitectura continúa respetando los principios más básicos de los antecesores procesadores de la familia Intel®. Este hecho, que ha debido suponer más de un dolor de cabeza a los diseñadores de Intel®, presenta como punto de partida la herencia de los antiguos 8086.

3.1. Novedades de la micro-arquitectura NetBurst®

Primero, veamos a grandes rasgos las novedades que introduce la micro-arquitectura NetBurst®.

Tecnología hiper-canalizada

El pipeline del Intel® Pentium® 4 tiene 20 etapas. Así, se duplica el número de etapas con respecto a los procesadores P6.
El aumento de etapas del pipeline permite aumentar la frecuencia de reloj, superando ya los 3 GHz en lo últimos modelos. Con esto se consiguen ciclos más cortos y, suponiendo el pipeline lleno, retirar instrucciones cada muy poco tiempo. Sin embargo, las paradas del pipeline (*pipeline stalls*) ocasionadas por distintos motivos provocan la anulación de un numero mayor de instrucciones, ya que en un pipeline de 20 etapas hay muchas instrucciones ejecutándose (*in-flight*).
Hay que tener en cuenta que el reciente Prescott tiene 31 etapas.

Bus del sistema rápido

El bus del sistema o FSB (***F**ront **S**ide **B**us*) aumenta su velocidad de forma notable con la micro-arquitectura NetBurst®. Hay tres velocidades de bus disponibles en la gama de procesadores Intel® Pentium® 4: 400 MHz, 533 MHz y 800 MHz.

Velocidad FSB	Chipsets
400 MHz	845 · 845GL · 845G · 845E · 845GV · 845GE · 845PE · 850 · 850E · 865P
533 MHz	845G · 845E · 845GV · 845GE · 845PE · 850 · 850E · 865P
800 MHz	865PE · 865G · 875P

Ejecución dinámica avanzada

Intel® defiende la posibilidad, aparentemente imposible, de mantener 126 micro-operaciones al mismo tiempo en el pipeline. Además, puede llegar a haber hasta 48 micro-operaciones *load* a la vez y hasta 24 micro-operaciones *store* a la vez. Esto se consigue mediante un núcleo de ejecución superescalar y OOO (***O**ut-**O**f-**O**rder*) o fuera de orden. Por otra

parte, el núcleo es capaz de despachar hasta 6 micro-operaciones en cada ciclo de reloj.

Sistema de ejecución rápida
Esta característica, a la que Intel® llama *Rapid Execution Engine*, indica que ciertas unidades funcionales operan a una frecuencia mayor que la del procesador. Concretamente, son dos unidades aritmético-lógicas (*double speed ALUs*) para operar sobre enteros las que funcionan al doble de frecuencia del procesador. Por ejemplo, las dos *double speed* ALUs de un Intel® Pentium® 4 a 2.4 GHz operarían a 4.8 GHz.

Execution Trace Cache
Quizás la introducción más destacable en la micro-arquitectura NetBurst® sea su nueva caché de instrucciones de primer nivel, la llamada Trace Cache. Esta caché almacena micro-operaciones (y no instrucciones IA-32), concretamente 12 Kμops. Examinaremos con más detalle la Trace Cache más adelante.

Cache avanzada de transferencia L2 (Advanced Transfer Cache L2)
La caché unificada de segundo nivel del Intel® Pentium® 4 ha reforzado el ancho de banda con la caché de datos L1 (ancho de 256 bits), para encarar las fuertes necesidades de bits de datos que requieren las aplicaciones multimedia. El tamaño de línea es inusualmente grande, 128 bytes. Pero esta elección tampoco es casual ya que, como veremos junto con el mecanismo de prebúsqueda (*prefetching*), ayuda a ocultar la gran latencia que suponen los accesos a memoria.
Esta caché esta segmentada, admitiendo una nueva operación cada dos ciclos de reloj. Así, en un Intel® Pentium® 4 *Willamette* a 1.5 GHz se consigue un ancho de banda de 48Gbytes por segundo.

Operaciones matemáticas con SSE2
Como ya hemos comentado anteriormente, la micro-arquitectura NetBurst® ha aumentado aún más la funcionalidad de las operaciones SIMD con el nuevo repertorio de instrucciones SSE2. Estas operaciones, hasta 144 instrucciones, son capaces de trabajar sobre dos datos en punto flotante empaquetados de doble precisión, o sobre enteros de 16 bytes, 8 palabras, 4 dobles palabras y 2 *quadwords*.
Con la introducción de las SSE2, los procesadores Intel® Pentium® 4 pasan a tener:

- La clásica unidad en punto flotante o FPU (***F****loating* ***P****oint* ***U****nit*).
- Las antiguas extensiones MMX del Intel® Pentium® MMX.
- Las instrucciones SSE introducidas en el Intel® Pentium® III.
- Las instrucciones SSE2, novedad del Intel® Pentium® 4.

Tecnología Hyper-Threading
Algunos modelos del procesador Intel® Pentium® 4 incorporan esta tecnología, introducida para lograr un mayor paralelismo a nivel de hilo ó TLP (***T****hread* ***L****evel* ***P****arallelism*). Con esto, un procesador Intel® Pentium® 4 que soporta HT (***H****yper-****T****hreading*) posee la habilidad de ejecutar hasta dos threads *a la vez*, mostrándose al sistema operativo como dos procesadores lógicos.
La tecnología HT requiere soporte del sistema operativo (como Windows XP y sistemas Linux con kernel 2.4.x o mayor).

3.2. Características generales de la micro-arquitectura NetBurst®

La micro-arquitectura NetBurst® se basa en un repertorio de instrucciones CISC, es decir, instrucciones complejas de longitud variable. A pesar de esto, los procesadores Intel® Pentium® 4 (y también sus antecesores) poseen un núcleo que ejecuta instrucciones sencillas, llamadas micro-operaciones *(*µops) o instrucciones RISC. Esto obliga a realizar una traducción CISC/RISC, es decir, una decodificación de instrucciones IA-32 (RISC) en micro-operaciones que serán las que realmente ejecute el núcleo. Este proceso de decodificación supone un proceso costoso y es un punto importante a considerar en el rendimiento de los procesadores Intel®. La micro-arquitectura NetBurst® tiene en cuenta este hecho y presenta una solución para poder suministrar un flujo de instrucciones superior al conseguido con las tradicionales cachés de instrucciones: la *Trace Cache*. La *Trace Cache* no es ni más ni menos que la caché de primer nivel (L1) de instrucciones (en este caso, de micro-operaciones) y su principal misión como veremos en detalle es evitar, al menos en la mayor medida posible, las costosas decodificaciones CISC/RISC.

Y como Intel® sigue apostando por mantener la compatibilidad con los antiguos 8086, deja solamente 8 registros visibles al programador, con la imperiosa necesidad de un mecanismo de renombramiento de registros a gran escala. Mantener la compatibilidad con las versiones anteriores de procesadores supuso afrontar estos problemas, partiendo en desventaja frente a otras alternativas.

La nueva arquitectura presenta un profundo pipeline de 20 etapas (excepción, 31 etapas para los modelos Prescott). Este aumento en el número de etapas frente a la arquitectura P6 ha permitido a Intel® aumentar vertiginosamente la frecuencia de los procesadores. Sin embargo, un pipeline largo también tiene unos riesgos potenciales en el rendimiento; y es que una parada en un pipeline de muchas etapas supone anular muchas instrucciones (en este caso, micro-operaciones) y, por consiguiente, una fuerte penalización en el rendimiento. Estas paradas en el pipeline (*pipeline stalls*) pueden producirse por fallos en la caché o por algún error en las predicciones de salto. Esto conlleva a redirigir los esfuerzos en pos de conseguir una política de saltos efectiva y una caché de primer nivel con baja tasa de fallos.

Como ya se sabe, la velocidad del procesador siempre ha ido un escalón por encima de la velocidad de acceso a la memoria. Teniendo esto en cuenta, el gran aumento en la frecuencia de procesador que supuso la aparición de los

procesadores Intel® Pentium® 4, ha provocado la necesidad de incluir una caché de primer nivel más rápida. Por esto, la micro-arquitectura NetBurst® incluye una caché de datos L1 más rápida pero más pequeña: sólo 8 Kbytes frente a los 16 Kbytes del Pentium® Pro y a los 64 Kbytes de los Athlon. Claro que esta pequeña caché permitió conseguir una extremadamente baja latencia de sólo 2 ciclos de reloj. El resultado son lecturas en L1 mucho más rápidas (menos de la mitad de tiempo) que en el Intel® Pentium® III. Sin embargo, el reducido tamaño de la caché puede ocasionar penalizaciones en el rendimiento. La aparición del modelo Prescott conlleva un incremento en el tamaño de la caché, duplicado su tamaño hasta 16 Kbytes.

3.3. Etapas del pipeline

Las 20 etapas del pipeline del Intel® Pentium® 4 (ignorando el Prescott que tiene 31) son las siguientes:

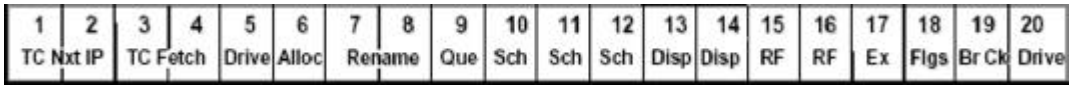

Figura 3. Etapas del pipeline de los procesadores Willamette y Northwood

TC Nxt IP

Se accede a la siguiente instrucción a partir del puntero de la BTB (Branch Table Buffer).

TC Fetch

Leer las instrucciones decodificadas o µops de la Trace Cache. En caso de fallo en caché habría que acceder a la caché L2.

Drive

Conduce las µops hacia la fase de ejecución.

Alloc

Se asignan los recursos necesarios a las µops para su ejecución. Estos recursos son el store buffer, el load buffer, ROB, …

Rename

Se renombran los registros. Los 8 registros lógicos virtuales (EAX, EBX, …) se traducen a un registro perteneciente a un espacio de 128 registros físicos (128 para enteros y 128 para punto flotante).

Que

Las µops se introducen en unas colas. Allí estarán hasta que el planificador las requiera.

Sch (Schedule)

Acceso a los planificadores y cálculo de dependencias.

Disp (Dispatch)

Despachar cada μops a su unidad funcional adecuada.

RF

Leer de los registros los operandos para poder realizar la operación.

Ex

Ejecutar la μop en su unidad funcional correspondiente.

Flgs

Realizar el cálculo de los flags de control tras haber efectuado la operación. Estos flags suelen servir como entrada a una instrucción de salto.

Br Ck (Branch Check)

Las μops de salto comparan la dirección predicha con la dirección actual del salto para ver si ha habido error o no en la predicción.

Drive

Conduce el resultado de la comparación del salto al front-end.

4. Arquitectura del Front-end

Podemos considerar que los elementos fundamentales que componen el front-end del Intel® Pentium® 4 son:

- El decodificador de instrucciones CISC/RISC.
- La ITLB (***I**nstruction **T**ranslation-**L**ookaside **B**uffer*)
- La Trace Cache
- El predictor de saltos

La figura 1 muestra la interconexión entre los elementos que componen el front-end.

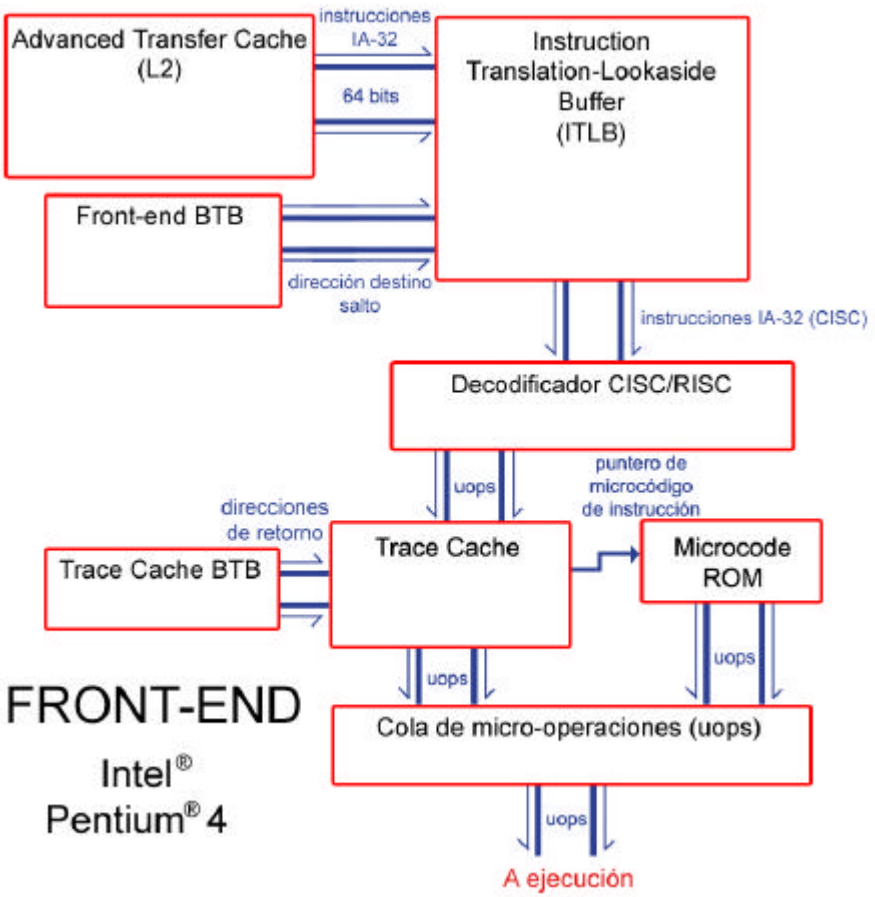

Figura 4. Arquitectura del front-end del Intel® Pentium® 4

4.1. Decodificador de instrucciones CISC/RISC

Como ya se ha mencionado, Intel® arrastra su complejo repertorio de instrucciones CISC para mantener la compatibilidad con las versiones anteriores de sus procesadores. Sin embargo, Intel® optó por dotar de un núcleo RISC a sus procesadores. Esto conllevó a realizar una obligada

traducción/decodificación de instrucciones complejas IA-32 a micro-operaciones RISC o µops.
Esta decodificación en muchos casos resulta ser 1:1, pero en algunos casos una instrucción IA-32 puede traducirse en más de 4 micro-operaciones.
Todos los procesadores Intel® más modernos disponen de un decodificador capaz de decodificar una instrucción IA-32 por ciclo. Y es que , aunque Intel® no lo indica claramente, suponemos que sólo existe un decodificador en contraposición a los tres decodificadores que tenía el Intel® Pentium® III. Esta simplificación podría deberse a la inclusión de la Trace Cache, que examinaremos más tarde.
Este decodificador recibe 64 bits de la caché unificada L2, que va almacenando en un buffer hasta que reconoce una instrucción completa. En caso de que la decodificación de dicha instrucción no supongas más de 4 micro-operaciones, se realiza la traducción y se siguen recibiendo bits de la caché L2.
En caso de que la traducción suponga más de 4 micro-operaciones se utiliza la ROM de micro-operaciones. Esta ROM guarda las traducciones, las micro-operaciones, relativas a las instrucciones CISC más complejas (como ciertas operaciones sobre movimientos de cadenas que podrían traducirse en miles de µops). Las µops que se obtienen directamente de la ROM no pasan por la Trace Cache sino que se pasan directamente a la fase de ejecución.

4.2. ITLB

Las instrucciones complejas IA-32 se almacenan en la caché unificada L2. Si no se encuentran ahí, estarán en memoria principal. Para acceder a estas instrucciones hay que traducir de la dirección la dirección virtual utilizada por el procesador a la dirección física reales que ocupan las instrucciones en memoria.
Para realizar esta traducción de direcciones, se utiliza la ITLB que almacena las traducciones más recientes. Además realiza comprobaciones de seguridad a nivel de página.
La ITLB del Intel® Pentium® 4 posee 128 entradas, es asociativa de 4 vías y usa páginas de 4 KBytes.

4.3. Trace Cache

La Trace Cache es la caché de instrucciones L1 del Intel® Pentium® 4. No almacena instrucciones CISC sino micro-operaciones, es decir, las traducciones de las instrucciones complejas. Este elemento, que es uno de los puntos estrella de la micro-arquitectura NetBurst®, tiene como objetivo reducir la penalización que supone la decodificación CISC/RISC, ya que los aciertos en esta caché suponen ahorrar costosas decodificaciones.
Intel® adoptó este elemento del ámbito académico, ya que el concepto de la Trace Cache fue presentado por primera vez en 1996. La idea básica es capturar el comportamiento dinámico de las secuencias de instrucciones (incluyendo saltos). Por tanto, la Trace Cache almacena trazas de ejecuciones y no bloques contiguos. La tradicional caché de instrucciones era incapaz de

predecir más de un salto por ciclo, lo cual supone una limitación del flujo de instrucciones si consideramos que los bloques básicos (con una instrucción de salto) son de unas 5 instrucciones. Así, solo un bloque básico como mucho podría ser suministrado al pipeline por ciclo, lo cual supone un cuello de botella potencial o real.
La Trace Cache almacena, en cada línea, trazas de varias micro-operaciones, donde puede haber varios saltos. Así, el problema de acceso en un mismo ciclo a posiciones de caché lejanas entre sí se ve superado por la Trace Cache.
Además de almacenar trazas, esta estructura contiene cierta información de control, como las predicciones realizadas para elaborar la traza. También indica cómo continuar una traza al finalizar una línea.

4.4. Predictor de saltos

El predictor de saltos es una de las fases críticas de cualquier procesador superescalar. Y es que debido al gran número de etapas del pipeline, la correcta predicción de saltos en el Intel® Pentium® 4 es importantísima. Y es que, la dirección efectiva del salto no se conoce hasta las etapas finales del pipeline, por lo que una mala predicción conlleva rehacer una gran cantidad de trabajo que ha sido inútil.
Consta de un mecanismo de prebúsqueda (*prefetching*) que accede a las instrucciones de la caché unificada L2 que han sido predichas como las próximas en ejecutarse y las lleva al decodificador.
En el Intel® Pentium® 4, esta prebúsqueda esta guiada por un complejo sistema de predictores. Se usan mecanismos de predicción dinámica y de predicción estática. Este procesador ha introducido mejoras que han permitido eliminar el 33% de los fallos de predicciones obtenidos en el Intel® Pentium® III.
El mecanismo de predicción de saltos de este procesador predice todos los saltos cercanos, pero no saltos como los *irets* o las interrupciones software.

Todos los saltos tomados, aunque sean predichos correctamente, reducen en cierta medida el paralelismo del código en los procesadores tradicionales. Esto se debe al ancho de banda desperdiciado en la decodificación que sigue al salto y que precede al destino si éstos no se encuentran al final o al principio de sus respectivas líneas de caché. La Trace Cache, al guardar trazas de ejecución, traería solamente una vez las instrucciones desde la caché en el caso ideal.

El Intel® Pentium® 4 permite anotar, a nivel software, las instrucciones de salto. Esta información se añade como prefijo de las instrucciones, la cual se ignora para los procesadores anteriores. Este prefijo ayuda al predictor de saltos en su trabajo, facilitando la generación de trazas. Esta información software tiene mayor prioridad que el predictor estático.

4.4.1. Predicción dinámica

La predicción dinámica de saltos se basa en una tabla de historia de saltos con su correspondiente tabla de direcciones de los saltos.

Parece que el predictor de saltos del Intel® Pentium® 4 está basado en dos niveles de historia. Se accede a la tabla de historia de saltos por compartición de índice o método *g-share*. El tamaño del BTB pasa de tener 512 entradas en los procesadores P6 a tener 4K entradas en el Intel® Pentium® 4. Estas entradas se organizan como una caché asociativa de 4 vías.

4.4.2. Predicción estática

En caso de que no se pueda hacer predicción dinámica porque no se encuentre ninguna entrada en el BTB que coincida con el contador de programa de la instrucción actual, se realiza una predicción estática basada en la dirección del salto, que se conoce tras la decodificación.
Si el salto es hacia atrás (desplazamiento negativo) se considerará un salto tomado. En caso contrario, se considerará como un salto no tomado.
Es importante darse cuenta que, sabiendo este comportamiento, se puede adaptar el código para tener en cuenta esta política y acertar más predicciones.

4.4.3. Pila de direcciones de retorno

Todos los retornos de una rutina o función siempre son saltos tomados. Sin embargo, dado que estos saltos pueden provenir desde distintas direcciones, no podemos utilizar eficazmente la predicción estática.
Para esto, el Intel® Pentium® 4 incorpora una pila de direcciones de retorno (*return address stack*) que predice las direcciones de vuelta para una serie de llamadas a funciones. Esta pila de direcciones tiene un tamaño mucho menor que el *Front-end BTB*, ya que tan sólo posee 16 entradas. El nombre que da Intel® a esta pila de direcciones de retorno es *Trace Cache BTB*.

5. Arquitectura de la fase de ejecución

Como ya se mencionó anteriormente, el Intel® Pentium® 4 posee un mecanismo de ejecución dinámica avanzada, lo que le permite mantener hasta 126 µops en el pipeline y pudiendo haber hasta 48 *loads* y 24 *stores* (aunque Intel® afirma que los últimos procesadores con tecnología de 90 nm permiten más de 24 *stores* a la vez).
Las principales funciones del mecanismo de ejecución fuera de orden del Intel® Pentium® 4 son:

- La asignación de recursos a las µops.
- El renombramiento de registros
- La planificación.

5.1. Asignación de recursos (*allocator*)

La lógica de asignación de recursos (*allocator*) asigna los *buffers* necesarios a las micro-operaciones para que éstas puedan ejecutarse. Mientras no haya recursos disponibles, la micro-operación quedará paralizada hasta que se liberen los recursos necesarios. Una vez que los recursos están disponibles, la lógica de asignación de recursos se los asigna a la micro-operación y ésta ya puede pasar a ejecutarse.
Los recursos que hay que asignar son el ROB (***R**e-**O**rdering **B**uffer*), el RF (***R**egister **F**ile*) con 128 registros para enteros, el RF con 128 registros para punto flotante, los 48 *buffers de load*, los 24 *buffers de store*.
Cada entrada del ROB alberga una µop que está en el pipeline ejecutándose.

El ROB se encarga de hacer el seguimiento de cada instrucción (hasta 126 a la vez) hasta su finalización, garantizando que ésta finaliza en el orden correcto. Es decir, las micro-operaciones pueden ejecutarse en desorden, pero una vez que finalizan permanecen en el ROB hasta que todas las instrucciones previas han finalizado.
La asignación y liberación de entradas de la ROB a micro-operaciones se realiza de modo secuencial. El ROB dispone de dos punteros: uno a la entrada ocupada por micro-operación más antigua y otro a la siguiente entrada libre. De este modo, el ROB funciona como una cola circular. En cada entrada del ROB, hay información sobre el estado (*status*) de la micro-operación que la ocupa. El estado puede ser: *en espera de operandos*, *en ejecución*, *finalizada*, ...
También hay información sobre el registro en el que se debe escribir, tanto del renombrado como del registro lógico, y sobre el contador de programa de la instrucción, para utilizarlo en las comparaciones pertinentes en las predicciones de saltos.

La búsqueda de los operandos se realiza en el momento de *issue*, momento en el que las µops salen de una de las dos colas para dirigirse a uno de los cuatro puertos de ejecución que dan acceso a las 7 unidades funcionales disponibles.

Así, los valores efectivos de los operandos no se cogen hasta que se realiza la operación.
Las dos colas de µops se corresponden al concepto de *estaciones de reserva*.

Las micro-operaciones con enteros tendrán un registro de enteros (de los 128 posibles) asignado en el que escribirán el resultado. Como ya se ha indicado, hay otros 128 registros para operaciones en punto flotante. Además, si la micro-operación es un load o un store, se reservaría uno de los 48 *buffers de load* o uno de los 24 *buffers de store* respectivamente.

5.2. Renombramiento de registros

Dado que el Intel® Pentium® 4 posee un reducido número de registros lógicos, se requiere el uso de un agresivo renombramiento de registros para evitar conflictos por dependencias de datos.
El proceso de renombrado de registros es muy distinto al de los procesadores con arquitectura P6. En éstos procesadores antiguos sólo se usaba una tabla RAT y no existía un conjunto de registros independiente. Además, existían explícitamente registros arquitectónicos para almacenar los operandos y resultados de las operaciones.
En el Intel® Pentium® 4 hay dos tablas RAT (***R**egister **A**lias **T**able*): la *Front-end RAT* y la *Retirement RAT*. Existe un conjunto de registros independiente y, además, no existen registros arquitectónicos explícitos para almacenar los resultados, de modo que cualquier registro disponible es válido.
Se utiliza un RF, que es una tabla de 128 entradas o registros para almacenar valores enteros. Hay otro RF para almacenar valores en punto flotante, pero trataremos del RF como si fuera uno solo por simplificar las cosas. Los RF son independientes del ROB, el cual no almacena ningún valor.

Los RAT son dos pequeñas tablas (la *Front-end* y la *Retirement*) de 8 entradas cada una. Cada entrada corresponde a un registro lógico: EAX, EBX, …
Cada entrada *X* del RAT apunta a una entrada *Y* del RF, donde está el valor más reciente del registro lógico asociado a la entrada *X* del RAT.

Como se ha indicado ya, el RF es direccionado por los RAT. Cada entrada del RF indica si el valor contenido es válido o no. Si es válido, la micro-operación que acceda a esta entrada para leer un operando podrá hacerlo sin problemas. Sin embargo, no se toma el operando sino el índice de la entrada del RF donde está el operando. Si la entrada contiene un valor inválido, quiere decir que hay una micro-operación a la espera de escribir en esta entrada un nuevo valor. Es decir, el valor que hay en el registro no es el que debemos utilizar sino un valor que está pendiente de escribirse. En este caso, también nos llevamos el índice de la entrada del RF para poder saber cuándo estará preparado el operando.
Al guardar el resultado de una micro-operación, ésta ocupa una nueva posición del RF donde escribirá el resultado. Las entradas del RF se asignan desde una lista de entradas libres, no secuencialmente como en el ROB. El índice de esta entrada donde se escribirá el resultado también se pasa junto con el resto de la micro-operación a la cola de micro-operaciones.

Por tanto, la lógica de renombrado de registros renombra registros lógicos IA-32, como el registro EAX, en una de las 128 entradas del RF. En cualquier momento, puede haber múltiples instancias en el RF del mismo registro lógico. Para saber cuál de estas instancias es la más reciente, se utiliza la *Front-end RAT*. Así, una nueva micro-operación que quiera leer, por ejemplo, de EAX sólo tiene que consultar el puntero asociado a esa entrada del *Front-end RAT* para saber de dónde debe recoger el operando. Ese puntero le llevará a una entrada de un RF donde se encuentra el operando (caso de entrada válida) o donde se va a escribir el operando (caso de entrada inválida).

5.3. Planificación y lanzamiento a ejecución

Los planificadores de μops determinan cuándo una μop está lista para ejecutarse, es decir, cuándo tiene disponibles o válidos los operandos que requiere. Esta parte es el núcleo o corazón del motor de ejecución.
Estos planificadores permite la reordenación de las μops para que se ejecuten según estén listas, permitiendo una *ejecución fuera de orden*. Eso sí, siempre manteniendo las dependencias de datos del programa original.
Para la planificación de micro-operaciones, el Intel® Pentium® 4 se utilizan colas de micro-operaciones y planificadores de μops.

Hay dos colas de planificación, una para operaciones de acceso a memoria (*loads* o *stores*) y otra para operaciones que no acceden a memoria. Ambas almacenan μops en orden FIFO, es decir, la primera que entra es la primera que sale de la cola. Sin embargo, el orden no tiene por qué preservarse entre μops pertenecientes a distintas colas.

Hay varios planificadores de μops para tomar decisiones sobre qué tipo de micro-operaciones han de ejecutarse primero en determinadas unidades funcionales. Así, los planificadores determinan cuándo las μops están listas para ejecutarse basándose en la disponibilidad de sus operandos fuente y en la disponibilidad de la unidad funcional requerida por la μop.

Hay cuatro puertos de ejecución. Hay planificadores ligados a cada puerto, y se usa uno u otro dependiendo del tipo de micro-operación (*load, store, aritmética,...*).

Los puertos 0 y 1 pueden despachar micro-operaciones en medio ciclo de procesador si suponemos que están usando las *Fast ALUs* (se explican un poco más abajo). Por esto, también hay planificadores para las *Fast ALUs* que planifican en medio ciclo. Los demás planificadores lo hacen en un ciclo de reloj.
Como máximo, pueden despacharse 6 micro-operaciones por ciclo: 2 en el puerto 0, 2 en el puerto 1, una micro-operación de *load* en el puerto 2 y una micro-operación de *store* en el puerto 3.
Este ancho de banda sobrepasa el del *front-end* y el de *retirement*. Ambos tienen un máximo de 3 micro-operaciones por ciclo.

Debido a la alta frecuencia de diseño, los resultados de las micro-operaciones pueden *puentear* los RF y acceder rápidamente a las micro-operaciones que necesitan de los datos recientemente calculados. Este *bypass* se hace antes de que el dato se escriba en el RF.

La memoria caché de datos L1 se utiliza en las operaciones de acceso a memoria: *loads* y *stores*. Es una caché *write-through*, es decir, todas las escrituras en ella se copian en la caché unificada L2. Puede hacer hasta un *load* y un *store* por ciclo. Posee líneas de 64 bytes y es asociativa de 4 vías. Basándose en esta caché de datos L1, el Intel® Pentium® 4 realiza una agresiva especulación de datos. Esto quiere decir que los planificadores de μops despachan las operaciones antes de que las μops anteriores hayan terminado de hacer un *load*. Esto es, el planificador asume que va a haber un acierto en caché y que el dato estará disponible para la μop recién lanzada. Si se produjera un fallo en caché, estas μops especuladas tendían datos incorrectos y tendrían que ser re-lanzadas, *replayed*. Este mecanismo se conoce como *replay* y, como hemos visto, sucede cuando tenemos que volver a lanzar a ejecución una micro-operación debido a que necesita recursos que aún no están disponibles.

La figura 2 muestra un diagrama resumido del núcleo de ejecución fuera de orden del Intel® Pentium® 4.

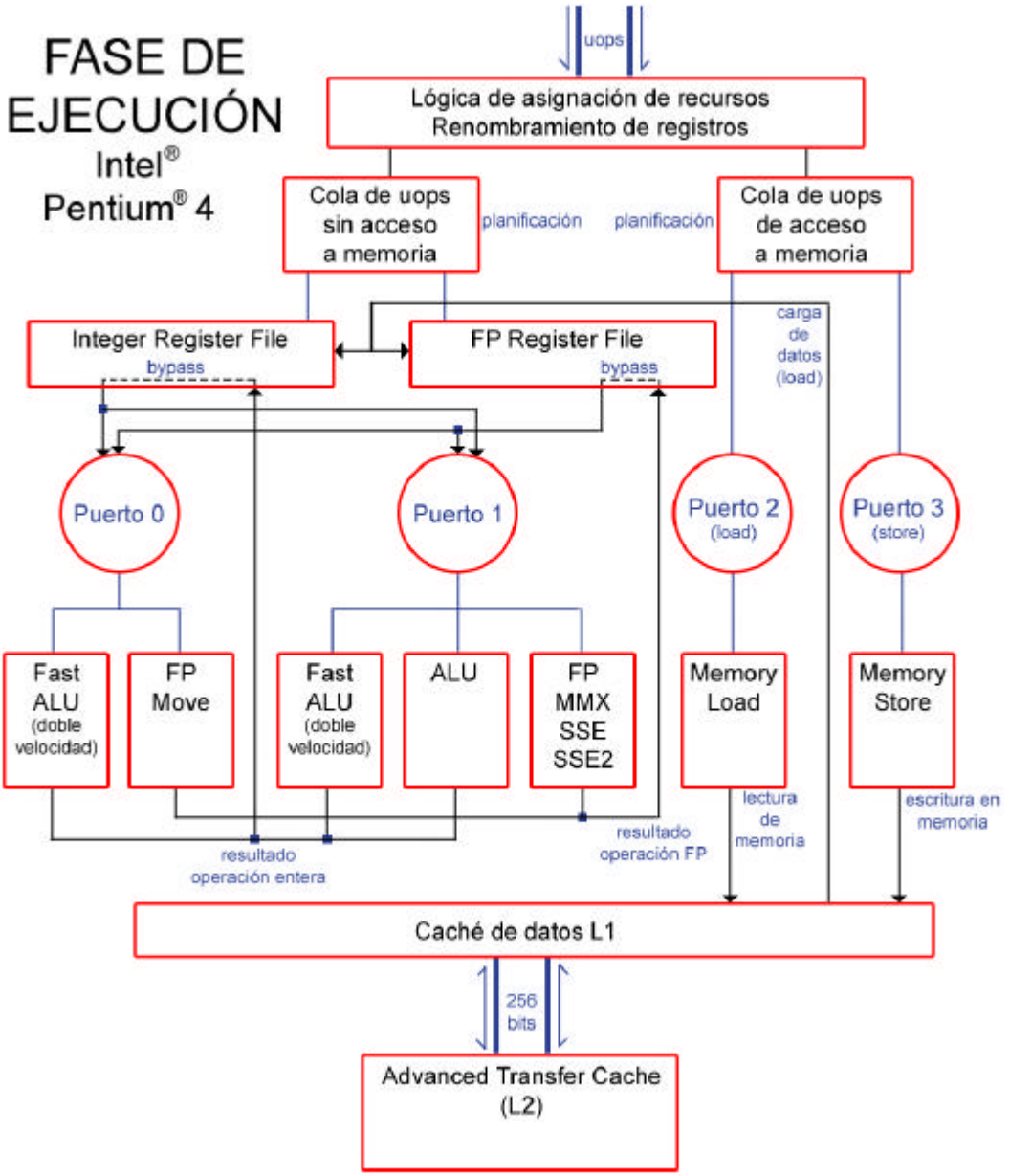

Figura 5. Arquitectura del núcleo de ejecución del Intel® Pentium® 4

5.3.1. Operaciones con enteros

Dos de los cuatro puertos de ejecución conducen a unidades funcionales para operaciones sobre enteros.

Intel® ha querido acelerar al máximo las operaciones más frecuentemente utilizadas, para lo cual ha implementado dos unidades funcionales para enteros muy rápidas, denominadas *Fast ALUs*. Estas unidades contienen solamente el hardware esencial para poder realizar un subconjunto de las operaciones con enteros. Las *Fast ALUs* operan al doble de velocidad que el procesador, por lo que también se las llama *Double Speed ALUs*.

El puerto 0 contiene una *Fast ALU* y el puerto 1 contiene una *Fast ALU* y una unidad normal para operaciones enteras.
Las dos *Fast ALU* tardan 3 ciclos en completar la operación, pero aquí se trata de ciclos que duran la mitad de tiempo que los ciclos de procesador. Así, estas unidades tardan ciclo y medio en completar una operación, consiguiendo una latencia de medio ciclo en caso de utilización continua de la unidad.
Para el resto de operaciones complejas como multiplicaciones, desplazamientos, cálculo de saltos y generación de *flags* se usa una unidad funcional normal para operaciones con enteros. Las latencias de estas operaciones varían dependiendo de la operación concreta: 4 ciclos para desplazamientos, 14 ciclos para multiplicaciones, 60 ciclos para divisiones, …

5.3.2. Operaciones en punto flotante

Existen dos unidades funcionales para operaciones en punto flotante en los puertos 0 y 1 respectivamente. Estas unidades realizan las operaciones de punto flotante y las operaciones SIMD basadas en las extensiones MMX, SSE y SSE2. En las primeras fases del diseño del Intel® Pentium® 4 se disponía de dos unidades funcionales completas pero, dado que la ganancia en rendimiento no era significativa, al final se prefirió reducir el coste especializando cada una de las unidades funcionales. Por tanto, la unidad funcional en punto flotante del puerto 0 se utiliza para operaciones de movimiento entre registros de 128 bits y escrituras a memoria. El puerto 1 sí incluye una unidad funcional completa en punto flotante, permitiendo hacer sumas, multiplicaciones, divisiones, operaciones MMX. Para reducir coste, la unidad funcional completa no está segmentada, pero esta unidad consigue una buena velocidad de ejecución.
El Intel® Pentium® 4 puede efectuar sumas en punto flotante en un ciclo de reloj, o en medio ciclo si se trata de operandos de precisión simple. Una suma SSE de operandos de 18 bits requiere dos ciclos, como una multiplicación.
En resumen, si suponemos que la velocidad de reloj del procesador es de 1.5 GHz, se alcanzan 6 GFLOPS en operaciones de precisión simple y 3 GFLOPS en operaciones de precisión doble.

5.3.3. Store-to-load forwarding

En una ejecución fuera de orden, los *stores* no pueden modificar el estado del sistema (caché L1, …) hasta que finalizan la etapa de *commit* y la micro-operación de *store* se retira.
Esperar a que se retire implica esperar a que todas las micro-operaciones que le preceden se retiren. Con el profundo pipeline del Intel® Pentium® 4, un store puede tardar mucho en retirarse. En un momento dado, puede haber hasta 24 *stores* en el pipeline, algunos de ellos se habrán retirado y otros no. A menudo, micro-operaciones *load* requieren un dato de memoria que es el resultado de un *store* pendiente. Para mejorar la eficiencia en estos casos, se dispone de un store buffer que permite que los *loads* utilicen los resultados de los *stores* previos antes de que éstos hayan escrito en la caché L1 de datos. Este proceso se llama *store-to-load-forwarding*.

Hay restricciones que en algunos casos impiden que se pueda efectuar este mecanismo. Por ejemplo, el tamaño del dato que carga el *load* debe ser igual o menor que el tamaño del dato que escribe el *store* previo, y ambos tienen que comenzar en la misma dirección física.
Cada una de las 24 entradas del *store buffer* puede albergar la dirección de escritura y el dato a escribir.

5.4. Subsistema de memoria

El procesador Intel® Pentium® 4 posee un subsistema de memoria capaz de afrontar los requerimientos de aplicaciones 3D, vídeo, …
El reducido número de registros disponibles para el programador provoca que los programas IA-32 contengan muchas referencias a memoria. Por esto, la efectividad de las cachés es fundamental si no queremos mermar la eficiencia del sistema.
La caché L1 de datos del Intel® Pentium® 4 es muy rápida y muy pequeña (aunque ampliada en los Prescott). La caché unificada L2 es mucho mayor y posee un gran ancho de banda. Ambas cachés son no bloqueantes, permitiendo acceder a ellas tras un fallo de caché. Esto es importante porque la demanda de accesos a memoria puede ser muy grande. El límite que este procesador pone a la caché L1 es de 4 lecturas simultáneas que fallen su acceso a la caché.

La traducción de la dirección virtual del dato requerido a la dirección física necesaria para direccionar la caché de datos, la realiza la DTLB (***D**ata **T**ranslation-**L**ookaside **B**uffer*). Esta estructura tiene 64 entradas y es totalmente asociativa. El acceso a la DTLB se realiza, normalmente, en paralelo a la caché.

La caché unificada L2, con política *write-back*, posee líneas de 128 bytes, lo cual es inusualmente grande. Sin embargo, estas líneas se dividen en dos sectores de 64 bytes, accesibles por separado. La independencia no es total dado que un fallo en caché obliga a traer la línea completa desde memoria principal.
La segmentación de esta caché permite el acceso a una nueva operación cada dos ciclos.
El ancho de banda con la caché de datos L1 esta bien reforzado con 256 bits.

El Intel® Pentium® 4 introduce una nueva estructura: el MOB (***M**emory-**O**rdering **B**uffer*). Esta estructura estaría entre la caché L1 y las unidades funcionales *Memory Load* y *Memory Store*. Un *store* o un *load* deben reservar una entrada en el MOB. La documentación que Intel® aporta sobre esta estructura es mínima.

6. Finalización de instrucciones

Recordemos que las micro-operaciones reservaban una entrada en el ROB. Cuando el ROB recibe la noticia de que la ejecución, por ejemplo, de una suma está completada entonces actualiza su estado (*status*). En este momento, la micro-operación ya está preparada para finalizar. Lo que falta es esperar a que todas las micro-operaciones previas finalicen sus cálculos, momento en el que podrá escribir sus resultados en los registros (siguiendo el ejemplo de la suma).

Cuando la entrada que ocupa la micro-operación en el ROB está a la cabeza, se retirará. Hasta 3 μops pueden retirarse por ciclo, de modo que la micro-operación será escogida junto con otras dos operaciones más, en caso de ser posible.
Como en el Intel® Pentium® 4 ya no hay registros separados para almacenar los valores (sólo tenemos los RF), necesitamos algún método para saber en cada momento qué entradas del RF hacen las veces de estos registros omitidos. La estructura que se encarga de esto es la *Retirement RAT* que, para cada uno de los registros lógicos visibles (EAX, EBX, …), apuntará a las entradas RF que guardan sus valores actuales.

Otra función del ROB consiste en comprobar si las predicciones de los saltos fueron correctas. Cuando una μop de salto finaliza su ejecución, vuelve a actualizar su entrada en el ROB. Entonces, se comprueba si la verdadera dirección de destino que se acaba de calcular coincide con la dirección predicha. Esta comparación se puede hacer utilizando el PC guardado en la siguiente entrada de la ROB, que contendrá el valor de la dirección predicha de salto. Si la comparación es exitosa, se prosigue sin incidentes. Si no, se vacía todo el pipeline, se borra el ROB, las colas de micro-operaciones y los registros de los RFs no apuntados por la *Retirement* RAT (esto es, los que no estarían en los registros EAX, EBX, … en caso de que existieran).

7. Tecnología Hyper-Threading

Aunque los primeros rumores indicaban que serían los *Prescott* los primeros procesadores en incorporar la nueva tecnología de Intel®, la presión de AMD y la estabilización de el sistema operativo Windows XP SP1 propiciaron que a finales del año 2002 Intel® incorporara la tecnología Hyper-Threading (HT) al *Northwood* de 3.06 GHz con FSB a 533 MHz.
Aunque los procesadores Intel® Pentium® 4 más antiguos ya poseen el hardware para usar Hyper-Threading, no lo tienen habilitado.
Más tarde, Intel® construye las versiones *Northwood* con Hyper-Threading activado y FSB a 800 MHz de todos los modelos disponibles con 533 MHz de bus (excepto el modelo a 2.26 GHz). Estos tres nuevos modelos son 2.4C GHz, 2.6C GHz y 2.8C GHz con FSB a 800 MHz e Hyper-Threading activado.

Resulta difícil aumentar el paralelismo a nivel de instrucción (ILP) en código x86, aun usando técnicas más avanzadas de microarquitectura que las utilizadas hoy en día. Además, el aumento de transistores, consumo y espacio puede no merecer la pena la mejora en el rendimiento obtenido.
La tecnología Hyper-Threading (HT) es el nombre que da Intel® a la implementación del procesamiento simultáneo multi-hilo (SMT). Esta tecnología, inicialmente llamada “tecnología Jackson”, trata de conseguir paralelismo a nivel de *thread* (TLP) en un solo procesador, y se beneficia de aplicaciones multi-hilo y de aplicaciones multitarea de un solo hilo.

Un procesador Intel® Pentium® 4 con HT activado puede procesar dos hilos (*threads*) simultáneamente mediante particionamiento, compartición y replicación de recursos (unidades funcionales, ...), mostrando al sistema operativo la existencia de dos procesadores lógicos dentro de un procesador físico. Así, un *thread* de ejecución puede utilizar los recursos de procesador que no esté usando el otro, ejecutándose ambos *threads* en paralelo.

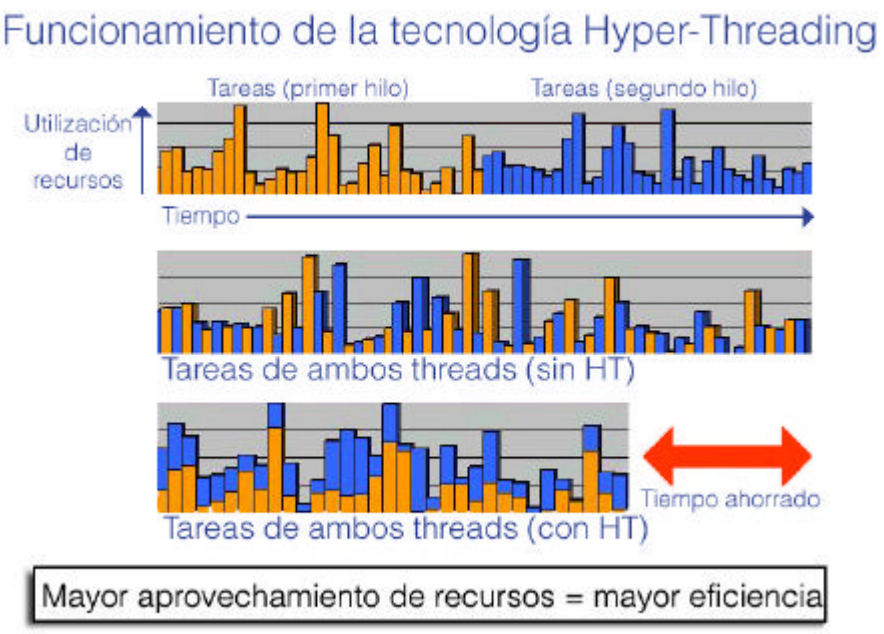

Figura 6. Ahorro de tiempo con el uso de la tecnología HT.

En la figura 4 observamos estas características.
La idea es aprovechar al máximo los recursos de CPU, como se muestra en la figura 5.

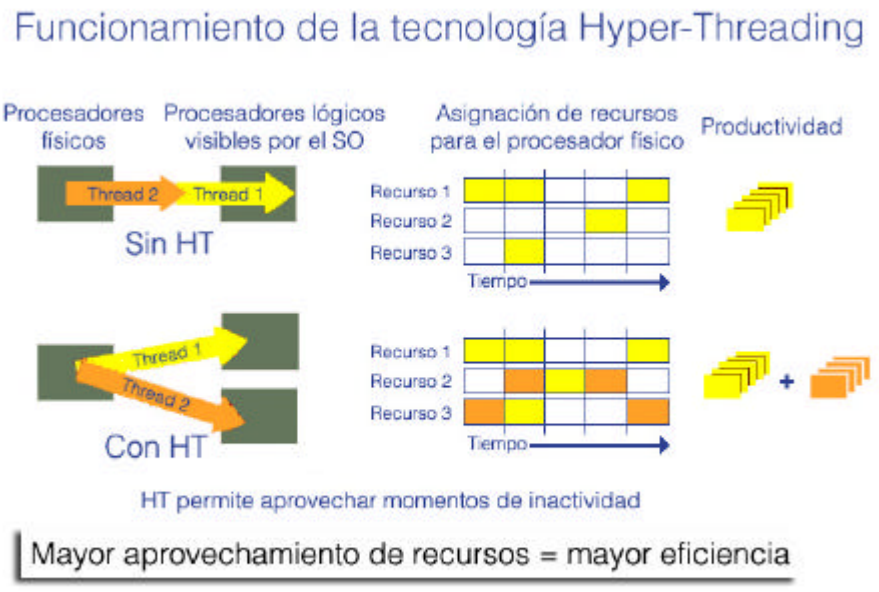

Figura 7. Principios básicos del funcionamiento de la tecnología HT.

La tecnología Hyper-Threading, mediante duplicado de partes del hardware, logra disponer de dos procesadores lógicos. Esta aproximación no es la misma que la de los sistemas con procesador dual.

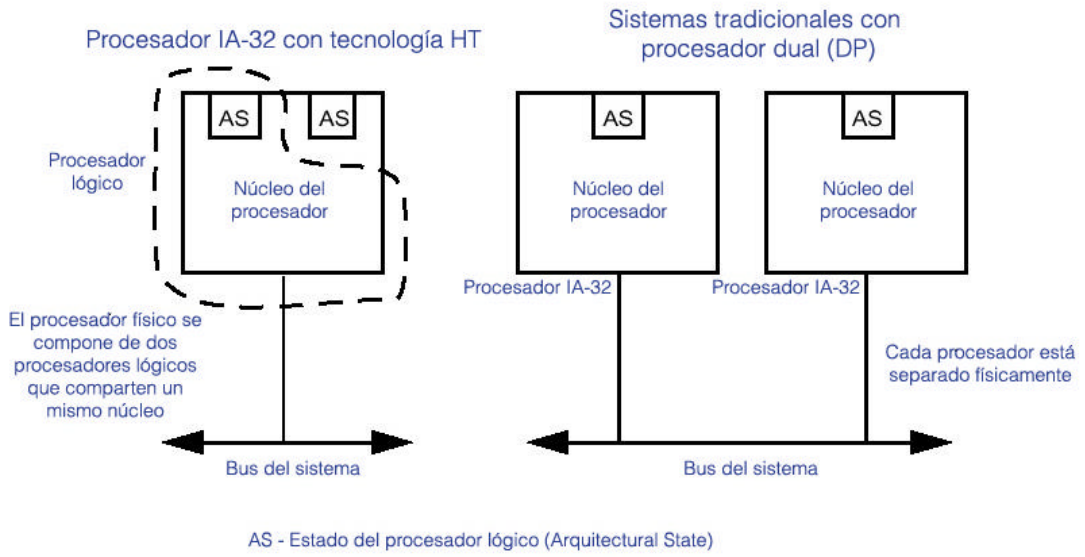

Figura 8. Diferencia entre un uniprocesador con HT y un sistema con procesador dual.

Con un sistema con procesador dual se obtiene un mejor rendimiento que con un procesador con tecnología HT de Intel®. Esto es obvio, hay que recordar que los dos procesadores lógicos tiene que compartir recursos tales como la caché, el acceso al bus y el motor de ejecución (*execution engine*).

La tecnología HT puede mejorar la eficiencia, sobre todo en ciertas situaciones donde el código está optimizado para ello. Además, esta tecnología es la transición al desarrollo de las aplicaciones multi-hilo que se ejecutarán en los futuros procesadores *multi-core*.

Monitorización del rendimiento en el Intel® Pentium® 4

Contenido

1. Introducción

Con la introducción del procesador Intel® Pentium® 4, Intel® da un gran salto hacia la versatilidad y flexibilidad de la monitorización del rendimiento mediante el uso de contadores hardware.
Respecto a la monitorización del rendimiento, las características de este procesador son:

- 18 contadores hardware de rendimiento *on-chip*.
- 18 registros de control asociados a los contadores.
- Entre 43 y 45 (depende del modelo) detectores de eventos.
- Otro hardware específico para monitorización.

La gran cantidad de contadores permite tomar muchas medidas de forma simultánea, en una misma ejecución, más que en cualquier otro procesador. Aunque potencialmente se pueden estar detectando hasta 43 o 45 eventos, sólo se pueden monitorizar hasta 18, que es el número de contadores disponible.
Este procesador incorpora multitud de características de filtrado de eventos, la posibilidad de configurar contadores en cascada y soporta mecanismos EBS para generación de perfiles de rendimiento.
Los manuales de documentación de Intel® incluyen tablas con indicaciones y explicaciones de cómo configurar los registros hardware para medir eventos predefinidos. Sin embargo, dependiendo de la versión de estos manuales la nomenclatura de las tablas puede variar. Además, resulta curioso observar que Intel® documenta nuevos eventos según actualiza dichos manuales.
Este trabajo hace referencia a estos manuales. Las versiones que se han utilizado de estos manuales son:

IA-32 Intel® Architecture Software Developer's Manual: Basic Architecture
Order Number: **253665**
IA-32 Intel® Architecture Software Developer's Manual: Instruction Set Reference, A-M
Order Number: **253666**
IA-32 Intel® Architecture Software Developer's Manual: Instruction Set Reference, N-Z
Order Number: **253667**
IA-32 Intel® Architecture Software Developer's Manual: System Programming Guide
Order Number: **253668**
IA-32 Intel® Architecture Optimization: Reference Manual
Order Number: **248966-011**

2. Soporte hardware del Intel® Pentium® 4 para monitorización

En esta sección trata sobre los recursos hardware disponibles en el Intel® Pentium® 4 para la monitorización del rendimiento. Se introducen los tipos de registros MSR (***M**odel **S**pecific **R**egisters*) y su organización en el procesador por zonas de monitorización. A continuación se comenta el funcionamiento de contado de eventos y algunas características de filtrado de eventos.

2.1. Organización de registros MSR

Los contadores de rendimiento del Intel® Pentium® 4 son registros MSR organizados por grupos y físicamente dispersos a lo largo del chip. Cada grupo de contadores es compartido por los detectores de eventos cercanos.
Los contadores hardware del Intel® Pentium® 4 son registros MSR llamados PMC (***P**erformance **M**onitoring **C**ounters*).
No todos los eventos se pueden contar a la vez ya que hay muchos más detectores de eventos (43 o 45) que contadores (18).
Los contadores se agrupan en *bloques de contadores* y los detectores de eventos se agrupan en *grupos de detección*.

Existen cuatro grandes grupos o *zonas de monitorización*:

- **BPU** (***B**ranch **P**redictor **U**nit*)
- **MS** (***M**icro-instruction **S**equencer*)
- **FLAME**
- **IQ** (***I**nstruction **Q**ueue*)

Cada zona de monitorización alberga varios grupos de detección de eventos y un bloque contadores que pueden utilizar los detectores de eventos de dichos grupos de detección.

Cada grupo de detección de eventos se compone de uno o más detectores de eventos, aunque en la mayoría de los casos se componen de dos detectores.
En el Intel® Pentium® 4, cada detector de eventos tiene asociado un registro MSR para configurar, entre otras características, la clase de eventos y los sub-eventos que va a detectar el detector. Estos registros se llaman ESCR (***E**vent **S**elect **C**ontrol **R**egisters*).
Como hemos indicado, la arquitectura del procesador Intel® Pentium® 4 limita cada zona de monitorización a poder utilizar solamente algunos grupos de detección de eventos. En el Intel® Pentium® 4, hay 21 grupos de detección de eventos, que suman un total de 45 registros ESCR (43 en los modelos nuevos).

Por otro lado, cada bloque de contadores está constituido por parejas de contadores. Igual que cada detector de eventos tiene asociado un registro ESCR, cada contador también tiene asociado registro MSR de control para habilitar o no el contador y realizar filtrado entre otras características. Estos registros asociados a cada contador son los CCCR (***C**ounter **C**onfiguration*

***C**ontrol **R**egister*) y, desde ahora, nos referiremos a la estructura contador/CCCR al conjunto formado por un contador y su CCCR asociado.
A continuación, la tabla 1 indica los recursos disponibles para cada zona de monitorización.

Zona de monitorización	Bloque de contadores	Grupos de detección asignados
BPU	2 pares (4 estructuras contador/CCCR)	BPU (***B**ranch **P**redictor **U**nit*) x2 ISTEER x2 IXLAT x2 ITLB (***I**nstruction **T**ransaction **L**ookaside **B**uffer*) x2 PMH (***P**age **M**iss **H**andler*) x2 MOB (***M**emory **O**rdering **B**uffer*) x2 FSB (***F**ront **S**ide **B**us*) x2 BSU x2
MS	2 pares (4 estructuras contador/CCCR)	MS (***M**icro-instruction **S**equencer*) x2 TC (***T**race **C**ache*) x2 TBPU x2
FLAME	2 pares (4 estructuras contador/CCCR)	FLAME x2 FIRM x2 SAAT x2 U2L x2 DAC x2
IQ	3 pares (6 estructuras contador/CCCR)	IQ* (***I**nstruction **Q**ueue*) x2 ALF x2 RAT (***R**egister **A**lias **T**able*) x2 CRU (***C**hecker/**R**etire **U**nit*) x6 SSU (***S**cheduler/**S**coreboard **U**nit*) x1

Tabla 1. Zonas de monitorización y sus registros MSR asociados

**Sólo disponible en los primeros procesadores (familia 0FH, modelos 01H - 02H)*

Además, dentro de cada zona de monitorización hay limitaciones adicionales sobre qué contadores puede usar cada grupo detector de eventos. Es decir, no cualquier ESCR de un grupo de detección de eventos puede modificar el valor de cualquier contador del bloque de contadores de su misma zona de monitorización.
Para aclarar esto, en la figura 1 se indica mediante un diagrama las interconexiones existentes entre los ESCRs y los contadores/CCCRs para cada zona de monitorización.

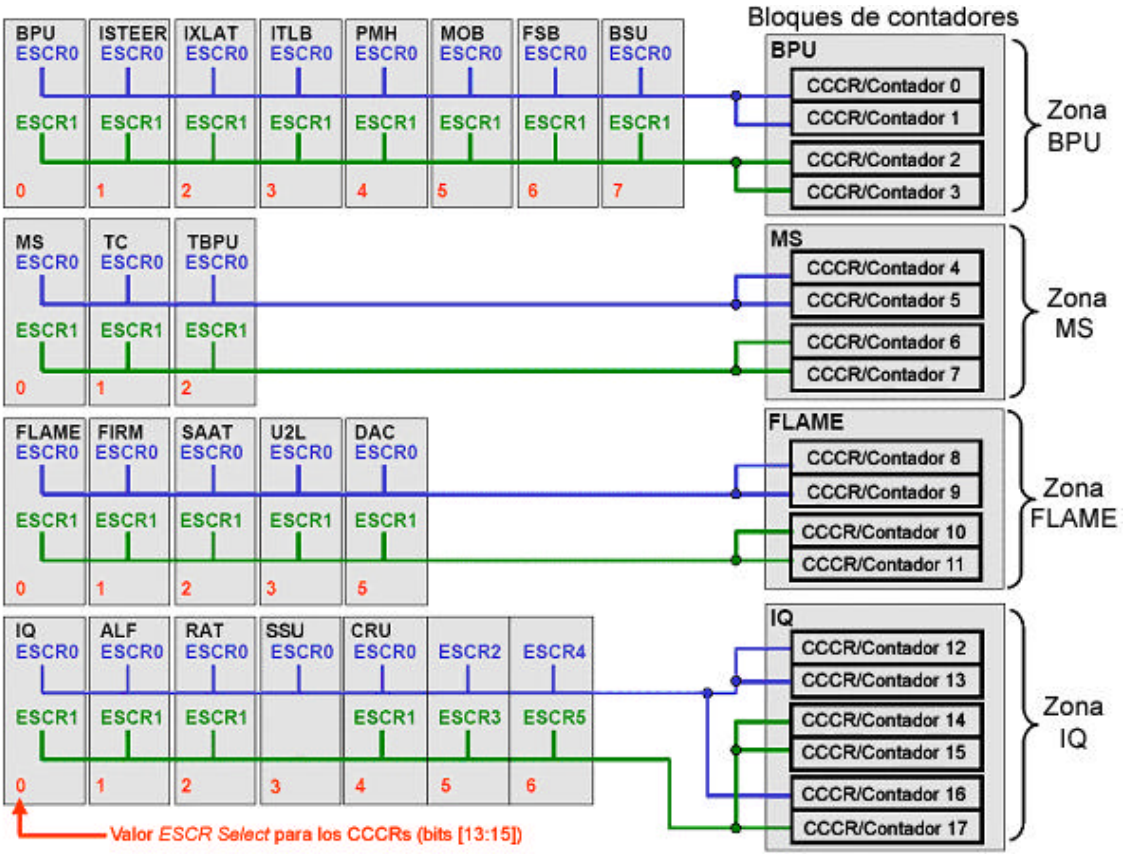

Figura 9. Interconexiones entre ESCRs y contadores/CCCRs

2.2. Mecanismo de métrica de eventos

Ahora veamos cómo se produce la detección de eventos y el consiguiente incremento en los contadores de rendimiento.

En cada ciclo cada ESCR se encarga de detectar el evento para el que está configurado que detecte. La aparición del evento, en cada ciclo, puede darse una vez, varias veces o ninguna.

El detector, utilizando la configuración del ESCR, realiza un filtrado cada vez que detecta un evento. Si el evento no supera el test de filtrado, el detector de eventos lo descartará. Más adelante veremos qué tipos de filtrado se pueden hacer.

En cada ciclo, el ESCR notifica a su contador/CCCR asociado el número de veces que ha detectado el evento, habiendo ya descartado las detecciones filtradas por los mecanismos de filtrado en el detector.

La asociación entre un detector ESCR y una estructura contador/CCCR se realiza desde el CCCR. En este registro de control hay un campo donde se configura qué grupo de detectores va a estar asociado a él y a su contador. Ya que cada contador sólo tiene acceso, a lo sumo, a un detector de eventos de cada grupo de detección de su grupo de monitorización (ver figura 1), con esta información el procesador ya puede asociar el detector/ESCR con el contador/CCCR.

En el Intel® Pentium® 4, la comunicación entre el detector/ESCR y el contador/CCCR se realiza a través de una línea de 4 bits. A través de esta línea, en cada ciclo, el detector indica al contador el número de veces que se ha detectado el evento de interés.
Dado el ancho de banda de la línea, un detector puede notificar un máximo de 15 detecciones por ciclo, aunque hubiera detectado más de 15 veces la aparición del evento en un mismo ciclo (situación más que complicada).

Una vez recibida la información del detector, se realiza un filtrado de eventos en el contador, configurable mediante el CCCR. Más adelante veremos qué tipos de filtrado se pueden hacer en el contador.
Es importante saber que si el ESCR notifica, por ejemplo, 3 detecciones del evento en un ciclo, esto no quiere decir que el contador asociado vaya a incrementarse en tres unidades. El contador podría estar deshabilitado y, si no lo estuviera, hay mecanismos de filtrado que pueden hacer que el contador se incremente desde ninguna a 3 veces (más de 3 veces en principio no parece viable).
Además, podemos configurar los contadores para que generen interrupciones al desbordarse (*overflow*), permitiendo enlaces en cascada y el uso de mecanismos EBS.
La figura 2 resume estos conceptos.

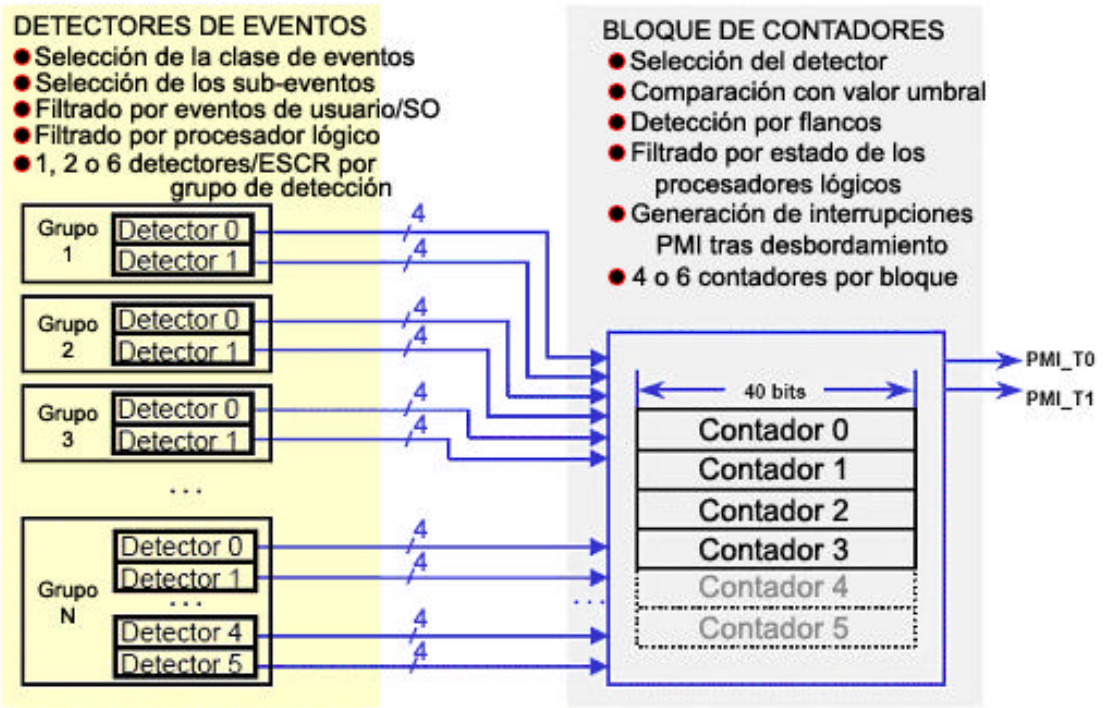

Figura 10. Asociación entre detectores/ESCR y contadores/CCCR

2.3 Filtrado de eventos

Como se ha indicado en el apartado anterior, el Intel® Pentium® 4 soporta varios mecanismos de filtrado, en los detectores de eventos y en los contadores.

Los mecanismos de filtrado en el detector se configuran en el ESCR asociado al detector. Mediante la configuración de ciertos bits del registro, podemos controlar el filtrado por eventos de usuario/SO y, para los procesadores con Hyper-Threading activado, filtrado por procesador lógico.
En el Intel® Pentium® 4, la distinción entre eventos de usuario y de sistema operativo se realiza mediante el nivel de privilegio o CPL (***C****urrent* ***P****rivilege* ***L****evel*) en el que se ejecuta la instrucción que produce la aparición del evento. Así, los eventos de usuario son aquellos producidos en los niveles CPL 1, 2 ó 3. Los eventos SO o de sistema operativo son aquellos producidos en nivel CPL 0.
Los campos del ESCR que se utilizan para el filtrado de eventos se explican con detalle en un apartado posterior.

Por otro lado, los mecanismos de filtrado que se realizan en el contador se configuran en el registro de control CCCR asociado. Muchos de los campos de este registro sirven para éste propósito. En concreto, podemos hacer un filtrado de eventos por comparación con valor umbral y por detección de flancos. El valor umbral usado para las comparaciones se configura en el propio registro CCCR. Además, los procesadores con Hyper-Threading activado permiten también hacer un filtrado basado en el estado de los procesadores lógicos en el momento de la aparición del evento.
Los campos del CCCR útiles para el filtrado de eventos se explican con detalle en un apartado posterior.

Respecto al filtrado por procesador lógico, hay que indicar que no todos los tipos de eventos soportan este filtrado. Por esto, surge la necesidad de clasificar los sub-eventos en:

> ***Sub-eventos TS*** (***T****hread* ***S****pecific*)
> Son los que sí pueden beneficiarse del uso del filtrado por procesador lógico.
>
> ***Sub-eventos TI*** (***T****hread* ***I****ndependent*)
> Estos eventos son independientes del procesador lógico que los genera, por lo que no soportan el filtrado por procesador lógico.

Si tratásemos de medir un sub-evento TI para un solo procesador lógico, obtendremos los resultados globales para ambos.
La documentación de Intel® facilita información para saber qué sub-eventos son TS y cuáles son TI en la tabla A-7 del apéndice A del manual *IA-32 Intel® Architecture Software Development Manual: System Programing Guide*.

3. El registro Time-Stamp Counter

La arquitectura IA-32, desde el primer Pentium, posee un registro que cuenta los ciclos de procesador. Este registro MSR de 64 bits se llama IA32_TIME_STAMP_COUNTER en el Intel® Pentium® 4, aunque lo abreviaremos como TSC (***T****ime-****S****tamp* ***C****ounter*).
Cuando se hace un reset al procesador se pone a 0. Después del reset, este registro/contador se incrementa en una unidad cada ciclo de procesador, incluso cuando el procesador esta en modo *halt*.

La instrucción RDTSC se utiliza para leer el contenido del TSC. Normalmente, esta instrucción puede ejecutarse en programas a cualquier CPL o nivel de privilegio y en modo virtual-8086.
El *flag* TSD del registro de control CR4 permite que RDTSC sólo pueda hacerse bajo el modo privilegiado (CPL=0). De este modo, el usuario no tendría acceso al TSC. Un sistema operativo que impide el acceso del usuario al TSC debe emular la instrucción RDTSC a través de un interfaz de programa accesible por el usuario.

También se puede leer o escribir el TSC con las instrucciones RDMSR y WRMSR respectivamente. Si usamos estas instrucciones tenemos que indicar la dirección del TSC, que es 10H. Con RDMSR podemos leer los 64 bits del contador pero, sin embargo, el uso de WRMSR sobrescribe solo los 32 bits de menor peso y pone el resto de bits (del 32 al 63) a cero.

4. Estructura y configuración de los PMC

Como ya se ha indicado, los contadores de rendimiento en el Intel® Pentium® 4 son registros MSR llamados PMCs (***P****erformance* ***M****onitoring* ***C****ounters*). Estos contadores, junto con los CCCRs se usan para filtrar y contar los eventos detectados por los ESCRs. El Intel® Pentium® 4 posee 18 contadores de rendimiento agrupados en 9 pares (ver figura 1). Como ya se ha indicado, cada par de contadores tienen asociada la capacidad de contabilizar un subconjunto de eventos, debido a que sólo pueden asociarse con los grupos de detección de eventos de su zona de monitorización.

Cada contador PMC es un registro de 64 bits. Sin embargo, sólo utiliza los 40 bits de menor peso, reservando los otros 24. El rango de valores que puede albergar un PMC es desde 0 hasta 2^{40}-1.
Para leer el valor de un contador, se utiliza la instrucción RDPMC. Con esta instrucción se pueden leer los 32 bits de menor peso o, con un coste mayor, los 40 bits.

Figura 11. Estructura de un contador PMC del Intel® Pentium® 4

La figura 3 muestra la sencilla estructura de cada contador.
Siguiendo la figura 1, los contadores PMC siguen la siguiente nomenclatura:

MSR_BPU_COUNTER0 (0)
MSR_BPU_COUNTER1 (1)
MSR_BPU_COUNTER2 (2)
MSR_BPU_COUNTER3 (3)

MSR_MS_COUNTER0 (4)
MSR_MS_COUNTER1 (5)
MSR_MS_COUNTER2 (6)
MSR_MS_COUNTER3 (7)

MSR_FLAME_COUNTER0 (8)
MSR_ FLAME_COUNTER1 (9)
MSR_ FLAME_COUNTER2 (10)
MSR_ FLAME_COUNTER3 (11)

MSR_IQ_COUNTER0 (12)
MSR_ IQ_COUNTER1 (13)
MSR_ IQ_COUNTER2 (14)
MSR_ IQ_COUNTER3 (15)
MSR_ IQ_COUNTER4 (16)
MSR_ IQ_COUNTER5 (17)

El único contador de los 18 disponibles que soporta el mecanismo PEBS es el contador 16, el MSR_IQ_COUNTER4.

El Intel® Pentium® 4 soporta la configuración de contadores en cascada. Los dos contadores de cada par pueden configurarse en cascada en cualquier

orden. Por ejemplo, se pueden configurar en cascada los contadores MSR_FLAME_COUNTER2 y MSR_FLAME_COUNTER3 en cualquier orden. Además, también se pueden configurar dos contadores en cascada pertenecientes a distinto par, siempre que coincidan sus situaciones relativas en su propio par. Un ejemplo de este caso sería configurar en cascada los contadores MSR_MS_COUNTER0 y el MSR_MS_COUNTER2 en cualquier orden.

5. Estructura y configuración de los ESCR

Los 45 registros ESCR son registros MSR de 64 bits que pueden ser configurados por software para contar eventos. Cada ESCR está asociado a dos o tres contadores, dependiendo de la zona de monitorización. Sin embargo, cada estructura contador/CCCR está asociada a varios registros ESCRs (ver figura 1).

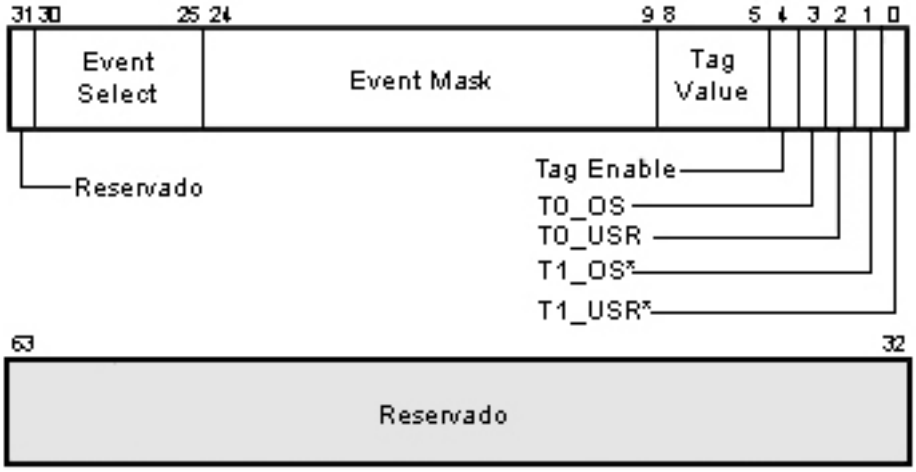

Figura 12. Estructura de un ESCR del Intel® Pentium® 4

**Sólo para los procesadores Intel® Pentium® 4 con la tecnología Hyper-Threading activada. Los demás micros sólo disponen del procesador lógico 0 y tienen reservados los bits 0 y 1 de cada ESCR.*

La figura 4 muestra la estructura de un ESCR. Cada ESCR configura un detector de eventos, permitiendo seleccionar qué eventos se quieren detectar y contar.
Explicamos la función de cada campo del ESCR:

***T1_USR*, bit 0**

Cuando se activa, se cuentan los eventos que son producidos por el procesador lógico 1 en modo usuario, es decir, para código que se ejecuta en CPL (***C**urrent **P**rivilege **L**evel*) 1, 2 ó 3.

***T1_OS*, bit 1**

Cuando se activa, se cuentan los eventos que son producidos por el procesador lógico 1 en modo protegido o de sistema operativo, es decir, para código que se ejecuta en CPL 0.

***T0_USR*, bit 2**

Cuando se activa, se cuentan los eventos que son producidos por el procesador lógico 0 en modo usuario, es decir, para código que se ejecuta en CPL 1, 2 ó 3.

***T0_OS*, bit 3**

Cuando se activa, se cuentan los eventos que son producidos por el procesador lógico 1 en modo protegido o de sistema operativo, es decir, para código que se ejecuta en CPL 0.

***Tag Enable*, bit 4**

Cuando se activa, el detector de eventos realizará el etiquetado (*tagging*) a las μ-operaciones. *Este bit sólo es útil y considerado en el uso del mecanismo de etiquetado en ejecución*.

***Tag Value*, bits [5:8]**

Valor de la etiqueta que se le da a la μ-operación que es detectada y etiquetada por este ESCR. *Este campo, como el anterior, sólo es aplicable al uso del mecanismo de etiquetado en ejecución.*

***Event Mask*, bits [9:24]**

Permite seleccionar qué sub-eventos o tipos de eventos vamos a contar, dentro de la clase de eventos seleccionada en el campo *Event Select* (a continuación). Estos sub-eventos son complementarios, pudiendo medir varios a la vez.

***Event Select*, bits [25:30]**

Permite seleccionar la clase de eventos que vamos a contar.

Los bits [31:63] están reservados.

6. Estructura y configuración de los CCCR

El Intel® Pentium® 4 contiene 18 registros CCCR, uno asociado a cada contador de rendimiento. Estos registros MSR son registros de 64 bits que se utilizan para contar y filtrar eventos, configurar contadores en cascada o lanzar interrupciones al producirse el desbordamiento de su contador asociado.

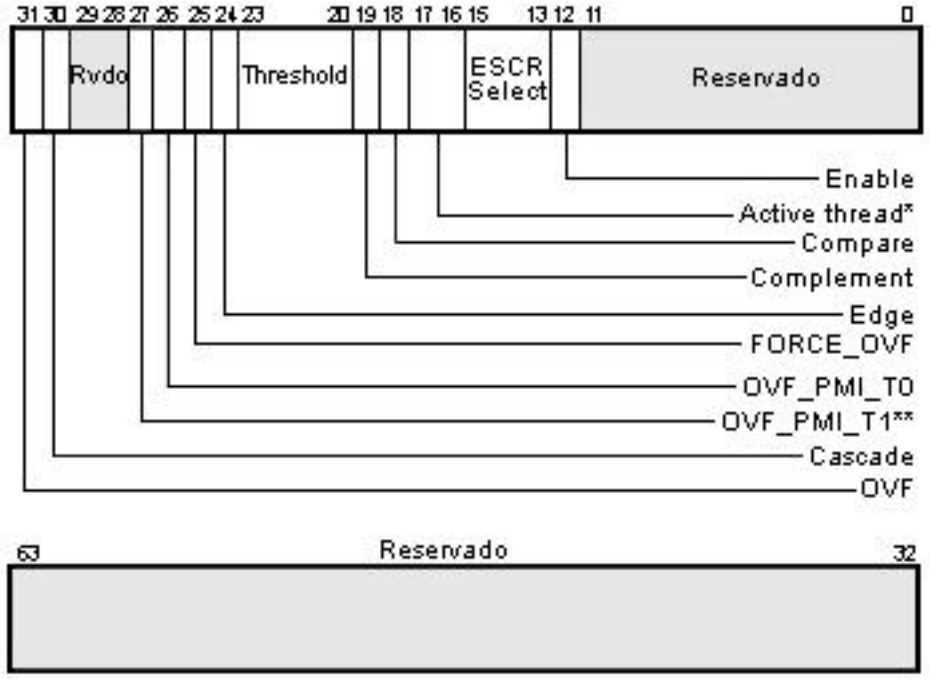

Figura 13. Estructura de un CCCR del Intel® Pentium® 4

Sólo para los procesadores Intel® Pentium® 4 con la tecnología Hyper-Threading activada. Para los demás Intel® Pentium® 4, estos bits no se usan y* ***deben fijarse a 1*

***Sólo para los procesadores Intel® Pentium® 4 con la tecnología Hyper-Threading activada. Para los demás Intel® Pentium® 4, este bit no se usa.*

La figura 5 muestra la estructura y campos de un CCCR. Cada CCCR contiene los siguientes campos:

***Enable*, bit 12**

Cuando este bit está activado, se activa la cuenta. Si está a cero, el contador no pude incrementarse. Al hacer un reset, este bit se pone a cero.

***ESCR Select*, bits [13:15]**

Indica qué registro ESCR está asociado al propio CCCR y su contador. Una vez establecido este vínculo, los eventos que detecte el ESCR asociado serán notificados al par contador/CCCR para que se incremente la cuenta.

***Active thread*, bits [16:17]**

Este campo activa un filtrado de eventos basado en el estado (activo o inactivo) de los procesadores lógicos. Este campo sólo

es configurable en procesadores Intel® Pentium® 4 con tecnología Hyper-Threading.
Las posibles opciones para este campo son:

00 - ninguno, cuenta solamente cuando ninguno de los dos procesadores lógicos está activo. Este caso, aparentemente extraño puede darse en ciertas circunstancias. Por ejemplo, imaginemos un sistema dual (dos procesadores físicos) donde el procesador físico 0 inicia una transacción de memoria que provoca que la caché del procesador físico 1 tenga que servir algún dato. Si en este procesador físico 1 se mide un evento relacionado con los accesos a la memoria caché usando un CCCR con el *Active thread* en modo ninguno, se contabilizarían los accesos a la caché cuando ambos procesadores lógico estén inactivos.

01 - único, cuenta solamente cuando uno y sólo uno de los dos procesadores lógicos está activo (el que sea), consumiendo todos los recursos del procesador físico.

10 – ambos, cuenta cuando ambos procesadores lógicos están activos, compartiendo los recursos del procesador.

11 – cualquiera, cuenta cuando al menos uno de los dos procesadores está activo. Este es el modo a utilizar si el procesador no soporta Hyper-Threading, de ahí que estos procesadores deben fijar este campo a 11B (ver figura 3).

***Compare*, bit 18**

Cuando se activa este bit, se activa el filtrado de eventos.

***Complement*, bit 19**

Determina cómo se efectúa la comparación entre el incremento y el valor umbral (*threshold*). Si el bit *complement* está activado, se incrementa el contador en caso de que el incremento detectado sea menor o igual que el valor umbral (especificado en el campo *threshold*, explicado a continuación). En otro caso, el contador se incrementará en una unidad si el incremento detectado es mayor que el valor umbral.
Este bit sólo tiene importancia si está activado el filtrado de eventos (es decir, si el bit *compare* está activado).

***Threshold*, bits [20:23]**

En este campo se indica el valor umbral que se utilizará en las comparaciones a la hora de realizar el filtrado de eventos. Este campo ocupa cuatro bits dado que el mayor incremento del contador que se puede producir en un ciclo es de 15 (ver figura 2). Este campo sólo tiene importancia si está activado el filtrado de eventos (es decir, si el bit *compare* está activado).

***Edge*, bit 24**

Cuando está activado se activa el modo de detección de flancos de subida. En este modo, el contador se incrementa en una unidad sólo si la comparación efectuada entre el incremento y el valor umbral es satisfactoria siempre que no lo haya sido en el ciclo inmediatamente anterior.
Este campo sólo tiene importancia si está activado el filtrado de eventos (es decir, si el bit *compare* está activado).

***FORCE_OVF*, bit 25**

Al activar este bit, se fuerza el desbordamiento del contador con cada incremento en el mismo. Si este bit está desactivado, el contador se comportará de forma normal, es decir, solamente se desbordará cuando el contador tenga su valor máximo (2^{40}-1) y se produzca un incremento en el mismo.

***OVF_PMI_T0*, bit 26**

Cuando esta activo, se genera una interrupción PMI para notificar al procesador lógico 0 que el contador asociado al CCCR se ha desbordado.
Si desactivamos este bit, no se lanza ninguna interrupción en caso de desbordamiento del contador.
Hay que indicar que la PMI se genera en la siguiente ocurrencia del evento una vez el contador se ha desbordado.

***OVF_PMI_T1*, bit 27**

Cuando esta activo, se genera una interrupción PMI para notificar al procesador lógico 1 que el contador asociado al CCCR se ha desbordado.
Si desactivamos este bit, no se lanza ninguna interrupción en caso de desbordamiento del contador.
Hay que indicar que la PMI se genera en la siguiente ocurrencia del evento una vez el contador se ha desbordado.

***Cascade*, bit 30**

Este bit se utiliza para poder configurar en cascada dos contadores de rendimiento. Este bit debe activarse en el contador que empieza a contar cuando el primero se ha desbordado.
Además, hay que desactivar el bit *enable* del CCCR del contador que empieza a contar tras el desbordamiento del primero.

***OVF*, bit 31**

Cuando está activo significa que el contador se ha desbordado.
Una vez se ha activado no se vuelve a poner a cero a no ser que lo desactivemos explícitamente mediante software.

Cuando se hace un *reset* al procesador, todos los bits de cada CCCR se inicializan a cero.

7. Medidas *at-retirement* en el Intel® Pentium® 4: mecanismos de etiquetado

La arquitectura NetBurst® del Intel® Pentium® 4 realiza muchas actividades de forma especulativa con la finalidad de incrementar la velocidad de proceso. Por ejemplo, esta arquitectura realiza una agresiva predicción de saltos y provoca que se ejecuten instrucciones especulativamente.
El Intel® Pentium® permite realizar medidas *at-retirement* para contar solamente los eventos producidos por instrucciones retiradas.
La tabla A-2 del apéndice A del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* lista los eventos sobre los que se puede hacer este tipo de medidas. Todos los eventos de esta tabla producen métricas *at-retirement*.
Además, un subconjunto de estos eventos soportan mecanismo de etiquetado, en concreto son los eventos *front_end_event*, *execution_event* y *replay_event*. Con el etiquetado se consigue medir el número de μ-operaciones que han producido un determinado evento. A lo largo de la ejecución, ese evento puede producirse varias veces para la misma μ-operación, por lo que medir directamente el número de veces que aparece el evento no es buena idea para saber cuántas μ-operaciones han producido el evento. Con el mecanismo de etiquetado, la μ-operación se etiqueta una vez si encuentra el evento al menos una vez, y la medida se realiza al final del pipeline al retirarse la μ-operación. Esta medida la realiza uno de los tres eventos indicados anteriormente, dependiendo del tipo de etiquetado que se haya realizado.
Los mecanismos de etiquetado se basan en marcar o etiquetar las μ-operaciones causantes de la aparición de un evento. Una vez etiquetada, una μ-operación conserva su etiqueta hasta que o bien es retirada (llega a la fase *commit*) o bien es cancelada por tratarse de una μ-operación mal especulada. Para conseguir medidas *at-retirement* utilizando este mecanismo, la idea es contabilizar las μ-operaciones etiquetadas que se retiran satisfactoriamente. Y, como se ha indicado, los eventos que cuentan estas μ-operaciones retiradas y etiquetadas son los eventos *front_end_event, execution_*event y *replay_event*. Hay algunas μ-operaciones que no pueden ser etiquetadas, como son operaciones de E/S, no cacheables, *locked accesses*, *returns* y *far transfers*.

Así, el Intel® Pentium® 4 implementa tres tipos de mecanismos de etiquetado. Cada uno de estos mecanismos es independiente, es decir, una μ-operación etiquetada por un mecanismo no será detectada por el detector de μ-operaciones etiquetadas que se están retirando de otro mecanismo. Para entenderlo, es como si cada mecanismo de etiquetado pusiera una etiqueta de un color distinto, de forma que cada mecanismo sólo contabiliza las instrucciones que se retiran con una etiqueta del mismo color que el usado para etiquetar.
Esta independencia entre mecanismos de etiquetado no es extensible al uso de PEBS. Si se usa PEBS, sólo se puede utilizar uno de los tres mecanismos de etiquetado.

Los mecanismos de etiquetado en el *front-ent* y de etiquetado por *replay* sólo son capaces de etiquetar o no etiquetar cada µ-operación. Es decir, no pueden establecer distintas etiquetas (distintos colores) según el evento que etiqueta la µ-operación. Esto implica que, en cada ejecución, no podemos separar las medidas producidas por dos eventos que utilicen el mismo mecanismo de etiquetado (*front-end* o *replay*).

Sin embargo, el etiquetado en ejecución no sólo es capaz de etiquetar o no las µ-operaciones sino que las puede etiquetar con distintas etiquetas. Y es que este mecanismo es el único que utiliza los cuatro bits *tag_value* del CCCR para dar valor a las etiquetas que pone a las µ-operaciones (ver figura 4). De este modo, en una misma ejecución, podemos utilizar el mecanismo de etiquetado en ejecución para que cuente por separado las medidas de hasta cuatro eventos simultáneamente.

7.1. Etiquetado en el front-end

El etiquetado en el *front-end* etiqueta las µ-operaciones que producen eventos en las primeras etapas del pipeline. Estos eventos estan relacionados con:

Decodificación de instrucciones en µ-operaciones

En este caso se utiliza el evento *uop_type* para etiquetar las µ-operaciones en función de su tipo (*load* o *store*).

Trace Cache

El etiquetado de µ-operaciones que causan la aparición de algún evento asociado a la Trace Cache se realiza en un registro MSR llamado MSR_TC_PRECISE_EVENT.
Actualmente, Intel® no define ningún evento relacionado con la Trace Cache, siendo nula la utilidad del MSR_TC_PRECISE_EVENT

El evento *front_end_event* contabiliza las µ-operaciones retiradas que han sido previamente etiquetadas por haber encontrado alguno de estos eventos.

La tabla A-2 del apéndice A del *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* describe el evento *front_end_event.* La tabla A-4 lista los eventos que se pueden usar para etiquetar µ-operaciones en el *front-end*.

7.2. Etiquetado en ejecución

Etiqueta instrucciones cuando éstas escriben sus resultados. El evento que contabiliza las instrucciones retiradas etiquetadas con este mecanismo es el evento *execution_event*.
Para medir un evento que utilice un mecanismo de etiquetado en ejecución, se utiliza un mecanismo de dos vías.

Primero, tenemos que utilizar un ESCR para especificar un evento a detectar y un valor de etiqueta (*tag value*) asociado con dicho evento. Esta es la parte de etiquetado o *upstream*. En este ESCR se debe activar el bit *tag enable*, así como dar un valor al campo *tag value*.
El valor del campo *tag value* ejerce una función de máscara en el etiquetado de las μ-operaciones, activando los bits de la etiqueta de las mismas según estén activados los bits del *tag value* en el ESCR, es decir, donde haya un 1 en el campo *tag value* se pondrá un 1 al etiquetar la μ-operación.

Por otro lado, se cuenta solamente un subconjunto de las μ-operaciones que han sido etiquetadas en el *upstream*, concretamente las que se retiran satisfactoriamente y además tienen un determinado valor en el campo *tag value*. De esto se encarga un segundo ESCR, que constituye la parte de contado o *downstream*. Este ESCR se configura con el evento *execution_event* y detecta solamente las μ-operaciones que se retiran y que están etiquetadas con determinados valores. La elección de qué valores de etiqueta vamos a contar lo indicamos en el campo *Event Mask* de este ESCR.

Para detectar las μ-operaciones etiquetadas por un determinado ESCR *upstream*, tendré que poner en el *Event Mask* del ESCR *downstream* el mismo valor que el *tag value* del ESCR *upstream*.
Por esto, se pueden medir hasta cuatro eventos de este tipo, es decir, podría configurar cuatro ESCR *upstream* y en ESCR con *execution_event* como *downstream*.
Las cuatro posibilidades son:

ESCR upstream tag value	***ESCR downstream Event Mask***
0001	0000000000000001
0010	0000000000000010
0100	0000000000000100
1000	0000000000001000

El valor de los campos *tag enable* y *tag value* en el ESCR *downstream* son irrelevantes.
En la tabla A-5 del apéndice A del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* aparecen los eventos documentados por Intel® que utilizan mecanismo de etiquetado en ejecución. El evento *execution_event* se documenta enla tabla A-2 del mismo manual.

7.3. Etiquetado por replay

Etiqueta las μ-operaciones que son re-lanzadas (*replayed*) debido, por ejemplo, a fallos de caché, a saltos mal predichos, a violaciones de dependencia y otros conflictos con los recursos del procesador. El evento que se utiliza para contabilizar las instrucciones retiradas que han sido etiquetadas con este mecanismo es el evento *replay_event*.
El evento *replay_event* contabiliza las μ-operaciones retiradas que han sido re-lanzadas (*replay*). También son etiquetadas mediante este mecanismo los fallos en las predicciones de saltos.

Se pueden etiquetar un subconjunto de todas las μ-operaciones que han realizado *replay*.

La tabla A-2 del apéndice A del *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* describe el evento *replay_event.* La tabla A-6 lista los eventos que se pueden usar para etiquetar μ-operaciones por el método de *replay*.

El mecanismo de etiquetado por *replay* hace uso de dos registros MSR:

MSR_PEBS_MATRIX_VERT
IA32_MSR_PEBS_ENABLE

Para usar este mecanismo de etiquetado, es necesario activar el bit 24 del registro IA32_PEBS_ENABLE.
En el registro MSR_PEBS_MATRIX_VERT se indica, mediante una configuración adecuada de bits activados, qué tipo de μ-operaciones de las que hacen *replay* se van a etiquetar.
Por otra parte, en IA32_PEBS_ENABLE indicamos qué tipo de evento se va a considerar.
La tabla A-6 indica como configurar ambos registros para cada clase de evento definida por Intel®.

8. Guía para la métrica de eventos usando la documentación de Intel®

A la hora de medir un evento, lo primero es pensar en qué es lo se quiere medir. Evidentemente no se puede medir absolutamente todo, pero el procesador Intel® Pentium® 4, como ya hemos indicado, es capaz de medir una gran variedad de eventos que se pueden contabilizar con propósitos de monitorización del rendimiento.
Una vez pensado en lo que se quiere medir, podemos consultar la tabla B-1 del apéndice B del manual *IA-32 Intel® Architecture Optimization: Reference Manual*.
Este apéndice consta de varias tablas que nos pueden ayudar a elegir qué vamos a medir. En estas tablas se indican sugerencias de cosas que se pueden medir, con una breve descripción de qué es lo que se mide y, sobre todo, nos indica qué evento hay que configurar y qué máscara se requiere para poder medirlo.
Las versiones de los manuales de Intel® utilizadas para el trabajo se indican al comienzo de este documento, en el apartado 1.
La tabla B-1, como hemos indicada, muestra sugerencias sobre métricas comunes que se puede realizar. Aquí puede haber tanto métricas *non-retirement* como *at-retirement* (que usan algún mecanismo de etiquetado).
Las tablas B-2, B-3 y B-4 indican todos los eventos que se pueden medir usando los mecanismos de etiquetado por *replay*, *front-end* y *ejecución* respectivamente. Sin embargo, estas tres tablas se limitan a indicar el nombre del evento y la configuración requerida. Las tablas B-2, B-3 y B-4 son las mismas que las tablas A-6, A-4 y A-5 respectivamente del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide.*

Suponiendo que vamos a medir un evento. La tabla B-1 indica en la primera columna, *Metrics*, métricas comunes que se pueden realizar.
En la siguiente columna, *Description*, aparece una breve descripción de la métrica para tener más claro lo que se mide.
Viendo estas dos columnas es posible que lo que queramos medir tenga relación con alguna métrica indicada en las dos primeras columnas de alguna fila de la tabla. En este caso, nos quedamos con dicha fila de la tabla y miramos la tercera columna, *Event Name or Metric Expression*, en la que encontraré el nombre real del evento que tenemos que configurar. En algunos casos, si la métrica es más complicada esta columna indicará una expresión (división, suma, …) cuyos operandos son el resultado de otras métricas. En cualquier caso, esta columna nos indica qué tenemos que configurar y qué operaciones tenemos que realizar sobre los resultados, si lo precisan.
Con los nombres definitivos de los eventos ya se puede recurrir a otras tablas, esta vez las tablas A-1 y A-2 del apéndice A del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide.*
La cuarta columna, *Event Mask Value Required*, nos indica qué valores debemos indicar en el campo *Event Mask* del ESCR para que se realice la métrica deseada.

Evidentemente en esta tabla no está todo lo que se puede medir, tan sólo algunas sugerencias de las cosas más comunes que se suelen medir. Además, hay que tener en cuenta que con los mecanismos de filtrado (uso de un valor umbral, detección de flancos, …) podemos conseguir variantes notables a partir de una métrica base.
De cualquier modo, si no encontramos nada que nos sirva en esta tabla siempre queda consultar directamente las tablas A-1, A-2, A-4, A-5 y A-6 del apéndice A del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide.* Estas tablas contienen todos los eventos documentados por Intel®.

Si queremos medir un evento algo complicado, es posible que la métrica pueda obtenerse a partir de la composición de varias métricas. Por ejemplo, la mencionada tabla B-1 nos indica que para medir la proporción (*ratio*) de error la predicción de saltos debemos medir los saltos retirados que se han predicho mal y dividirlo entre otra métrica, los saltos retirados bien predichos.

Para medir eventos, es necesario configurar correctamente los registros MSR. En el apéndice A del manual de Intel® *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* hay varias tablas con eventos definidos, indicando para cada uno de ellos los registros MSR que puede utilizar y los valores que se deben fijar.

ESCR restrictions

Esta fila indica qué registros ESCR, de todos los disponibles, se pueden utilizar para medir el evento en cuestión.
Todos los registros ESCR que se pueden usar han de estar en la misma zona gris (ver figura 1). Aquí se determina la zona gris asociada al evento.

Counters Number per ESCR

Por cada ESCR que se puede utilizar (los indicados en *ESCR restrictions*), esta fila informa qué contadores, de los 18 disponibles, se pueden emplear para contar el evento.

ESCR Event Select

El valor indicado en esta fila es el código de la clase de eventos que se va a medir. Este valor se introduce directamente en el campo *Event Select* del ESCR que se va a utilizar para realizar la métrica del evento (que debe ser un ESCR que aparece en *ESCR restrictions*).

ESCR Event Mask

Aquí aparecen los distintos tipos de eventos o sub-eventos que componen la clase de evento que se va a medir (definida en *ESCR Event Select*). Cada sub-evento tiene un bit asociado del campo *Event Mask* del ESCR. En esta fila aparecen los bits asociados a cada sub-evento. Para contabilizar un sub-evento hay que activar su bit correspondiente. Cuidado, estos bits son relativos al intervalo [9:24] del ESCR de modo que, por ejemplo, el bit 0 del campo *Event Mask* correspondería al bit 9 ESCR.

CCCR Select

Esta fila indica el valor que debemos darle al campo *ESCR Select* del registro CCCR que se va a utilizar para realizar la métrica del evento. Mirando la figura 1, este valor debe ser el número rojo que aparece debajo de la zona gris que está asociada a este evento.

Event Specific Notes

En esta fila se proporciona información relativa al evento, como pueden ser limitaciones o consideraciones a tener en cuenta a la hora de medirlo.

Can Support PEBS*

Indica si el evento soporta o no el uso de PEBS. Actualmente, sólo soportan PEBS los eventos *front_end_event*, *execution_event* y *replay_event*. Los demás eventos para realizar medidas *at-retirement* deben utilizar IEBS.

Require Additional MSRs for tagging**

Este campo indica si es necesario configurar algún registro MSR (ESCR u otros) adicional para utilizar el mecanismo de etiquetado.

** Sólo aplicable a los eventos de la tabla A-2.*
*** Sólo aplicable a los eventos front_end_event, execution_event y replay_event.*

La elección de los sub-eventos a considerar, el filtrado de eventos por procesador lógico, comparación con umbral, detección de flancos, medición de eventos de sistema operativo o de usuario, uso de interrupciones en desbordamiento o contadores en cascada los se dejan, evidentemente, a elección del usuario. Por esto, las tablas de eventos definidos de la documentación de Intel® no hace referencia a los bits de los MSRs para controlar estas funciones.

Respecto al filtrado de eventos por procesador lógico, en la tabla A-7 del apéndice A del mismo manual podemos consultar qué sub-eventos son TI y qué sub-eventos son TS.

Los eventos que son contados por un contador activo son seleccionados y filtrados en el siguiente orden por los campos de los registros ESCR y CCCR que se describen a continuación:

1. Los campos *Event Select* y *Event Mask* del ESCR seleccionan la clase de eventos y los tipos de eventos o sub-eventos dentro de dicha clase, respectivamente.

2. Los bits de OS y USR en el ESCR se consideran para realizar el filtrado por nivel de privilegio o CPL.

3. El campo *ESCR Select* del CCCR selecciona el ESCR que va a estar vinculado al par contador/CCCR.

4. Se considera el filtrado de eventos por comparación con umbral. Aquí entran en juego los bits *compare*, *complement* y el campo *threshold* del CCCR.

5. Se considera el filtrado de eventos por detección de flancos. El bit *edge* del CCCR habilita el modo de cuenta sólo en transiciones de falso a positivo (de 0 a 1).

9. Ejemplos de configuración de métricas de eventos

En esta sección se analiza con detalle la configuración necesaria para realizar métricas de eventos. Se explica cómo configurar los registros MSR para medir eventos y se analizan los tres tipos de mecanismos de etiquetado.

9.1. Configuración de un evento sin etiquetado

Supongamos que queremos medir los *saltos tomados* que se han ejecutado. Vamos a suponer además que el procesador Intel® Pentium® 4 que utilizamos soporta Hyper-Threading.
Consultando la tabla B-1 del apéndice B del manual *IA-32 Intel® Architecture Optimization: Reference Manual*. En esta tabla aparece una métrica para medir todos los saltos retirados. Nosotros sólo queremos medir los saltos retirados que han sido tomados. Podemos basarnos en esta métrica haciendo los cambios oportunos.
Buscamos en la tabla A-2 del apéndice A del *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* el evento *branch_retired*. Aquí veremos el valor *Event Select* es 06H, por lo que necesitamos introducir el valor 000110 en el campo *Event Select* del ESCR.
En esta misma tabla, nos informan que mediante el campo *Event Mask* podemos seleccionar entre cuatro tipos de sub-eventos:

- *Saltos no tomados bien predichos (mmnp)*
- *Saltos no tomados mal predichos (mmnm)*
- *Saltos tomados bien predichos (mmtp)*
- *Saltos tomados mal predichos (mmtm)*

En el apéndice B nos indicaban que activáramos los cuatro bits *mmtm*, *mmnm*, *mmtp* y *mmnp*. Parece obvio que nosotros sólo debemos activar los bits *mmtm* y *mmtp* ya que sólo nos interesan los saltos tomados.
Como ya se ha indicado, estos sub-eventos son complementarios, pudiendo medirlos todos, algunos o solamente uno. Dado que queremos medir los saltos tomados (acertada o no la predicción realizada) tenemos que activar, siguiendo la tabla A-2, los bits 2 y 3 para medir dos sub-eventos.
Dado que estas posiciones son relativas al intervalo de bits [9:24], tenemos que introducir el valor 0000000000001100 en el campo *Event Mask* del ESCR.

Como hemos supuesto que tenemos un procesador con Hyper-Threading activado, tenemos dos procesadores lógicos. Suponemos que nuestra intención es medir solamente los eventos de usuario en el procesador lógico 0. Esto lo podemos hacer dado que si consultamos la tabla A-7 veremos que todos los sub-eventos o tipos de evento de la clase de eventos *branch_retired* son TS. Por lo tanto, fijamos *T1_USR*=0, *T1_OS*=0, *T0_USR*=1, *T1_OS*=0.
Si el evento fuera de tipo TI, tendríamos que realizar la métrica para ambos procesadores lógicos sin poder hacer distinción. En este caso obtendríamos el mismo efecto activando sólo *T0_USR* que activando sólo *T1_USR* que activando ambos.

Como este evento no utiliza mecanismos de etiquetado, pongo *tag_enable*=0 y *tag_value*=0000.

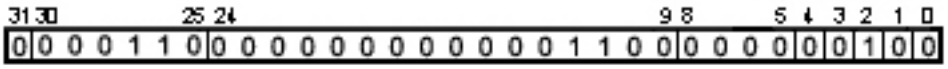

Figura 14. Valores del ESCR para medir el evento *branch_retired*

La figura 6 muestra cómo quedaría configurado el ESCR.

Mirando en la tabla A-2 del apéndice A del *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* consultamos qué registros MSR_CCCR se pueden utilizar para nuestro evento, *branch_retired*.
Los registros ESCR que podemos utilizar para medir el evento son, según la tabla, el MSR_CRU_ESCR2 y el MSR_CRU_ESCR3. Con esto, es obvio que el valor *ESCR Select* que indica la tabla sea 05H, como no podía ser de otra manera consultando la figura 1 (ambos ESCRs están en la misma zona gris, la 5, de la zona de monitorización IQ). Por tanto, los bits [13:15] del CCCR contendrán el valor 101.
La tabla nos indica que para el MSR_CRU_ESCR2 podemos utilizar los contadores 12, 13 y 16. Para MSR_CRU_ESCR3 podemos utilizar los contadores 14, 15 y 17. Esto también concuerda con la figura 1, donde podemos ver que son las únicas posibilidades para esos ESCRs.

Supongamos que se nos ocurre que en vez de contar todos los saltos tomados durante la ejecución, nos viene mejor contar el número de ciclos en que se da el evento más de dos veces, y en cualquiera de los dos procesadores lógicos. En este caso, tendría que activar el bit *compare* y fijar el umbral (*threshold*) a 2. El bit *complement* debe valer cero, ya que queremos incrementar el contador si el número de veces que se detecta el evento es *mayor* que el valor umbral. Al no utilizar filtrado por detección de flancos, el bit *edge* se queda a cero.

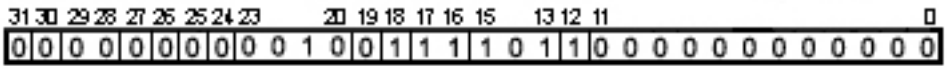

Figura 15. Valores del CCCR para medir el evento *branch_retired*

La figura 7 muestra la configuración del CCCR.
Si hubiéramos preferido medir los saltos tomados totales, tendríamos que poner *compare* a cero. Los campos *complement*, *threshold* y *edge* podrían tener cualquier valor dado que al estar el bit *compare* desactivado, estos campos no se tienen en cuenta.

9.2. Configuración de un evento con etiquetado en el front-end

Veamos un ejemplo de cómo medir un evento que utilice el mecanismo de etiquetado en el *front-end*. Vamos a medir μ-operaciones retiradas que cargan datos de memoria. Se trata de una medida *at-retirement* porque sólo nos quedamos con las μ-operaciones que retiran. Consultando la tabla B-1 del apéndice B del manual *IA-32 Intel® Architecture Optimization: Reference Manual* encontramos una métrica, *Loads retired*, para medir el número de operaciones load retiradas etiquetadas en el front-end. Como es lo que queremos medir, consultamos la tercera columna de esta fila. Aquí se indica que tenemos que configurar el evento *front_end_event* junto con la métrica de etiquetado *Memory_loads*.
Nos vamos a la tabla A-4 del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* para buscar *Memory_loads*. En esta tabla nos indican que debemos configurar el evento *uop_type*, que está en la tabla A-2, con el bit TAGLOADS activado. Además, hay que configurar el evento *front_end_event* con la máscara NBOGUS.
Primero configuramos el evento *uop_type*:

Evento uop_type

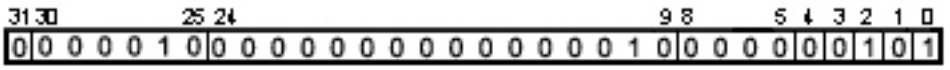

Figura 16. Valores del ESCR para medir el evento *uop_type*

En la figura 8 se muestra el ESCR que detectará el evento *uop_type*.
Hemos supuesto que nuestro procesador soporta Hyper-Threading y se ha decidido medir solamente los eventos de usuario para ambos procesadores lógicos.
Mirando la tabla A-2 para el evento *uop_type*, vemos que el código del evento es 02H, lo introducimos en el campo *Event Select*.
Como teníamos que activar el bit TAGLOADS, vemos en la tabla que es el bit 1 del campo *Event Mask*, que corresponde al bit 10 del ESCR. Activamos este bit.

Respecto al CCCR, tenemos que poner el valor *ESCR Select* al valor que nos indican en la tabla. En este caso es el valor 02H. Activamos el bit *Enable* para habilitar el contador y fijamos el campo *Active Thread*=11 para medir siempre que al menos uno de los dos procesadores lógicos esté activo. Como no realizamos ningún tipo de filtrado, los demás bits se dejan a cero. La figura 9 muestra el CCCR para el evento *uop_type*.

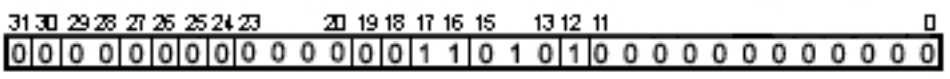

Figura 17. Valores del CCCR para medir el evento *uop_type*

Por otra parte, se necesita configurar el evento *front_end_event* para contar las μ-operaciones que se retiran y que han producido el evento anterior. Este evento también se explica en la tabla A-2.

Evento front_end_event

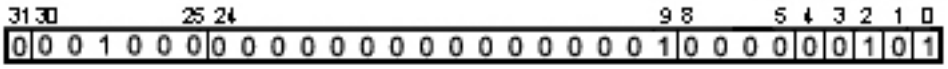

Figura 18. Valores del ESCR para medir el evento *front_end_event*

En la tabla, vemos que el código del evento (campo *Event Select*) es el 08H. Activamos la máscara NBOGUS como se nos había indicado. Este bit es el bit 0 del campo *Event Mask*, es decir, el bit 9 del ESCR.
La figura 10 muestra el ESCR configurado para el evento *front_end_event*.

Para configurar el CCCR, activamos el bit *Enable*, fijamos el campo *Active Thread*=11 e introducimos el valor *ESCR Select* indicado en la tabla A-2 para el *front_end_event*, que en este caso es 05H.
La figura 11 muestra el CCCR configurado.

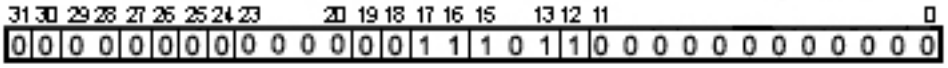

Figura 19. Valores del CCCR para medir el evento *front_end_event*

9.3. Configuración de un evento con etiquetado en ejecución

Vamos a medir el número de operaciones en punto flotante retiradas del tipo x87. Consultando el la tabla B-1 del apéndice B del manual *IA-32 Intel® Architecture Optimization: Reference Manual* encontramos esta métrica con el nombre *x87 Retired*. La columna tercera nos dice cómo tenemos que proceder para medir este evento. Tenemos que configurar el evento *execution_event* (*downstream*) y hacer el etiquetado con el evento *x87_FP_retired* (*upstream*).
Nos vamos a la tabla A-5 del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide* para buscar *x87_FP_retired*. En esta tabla nos indican que debemos configurar el evento *x87_FP_uop*, que está en la tabla A-1. Además, tenemos que activar el bit ALL del Event Mask. La clase de eventos *x87_FP_uop* actualmente no soporta otros sub-eventos, de modo que la única opción es activar este bit para detectar todas las operaciones en punto flotante ejecutadas en el x87.
Por último, hay que activar el bit *Tag Enable* y fijar un valor para el *Tag Value*. Además, hay que configurar el evento *execution_event* con la máscara adecuada. En la tabla nos indican que la máscara es NBOGUS0, sin embargo esto no es correcto ya que está máscara dependerá de lo que fijemos en el *Tag Value* del ESCR para el evento *x87_FP_uop*.

Primero configuramos el evento *x87_FP_uop*:

Evento x87_FP_uop

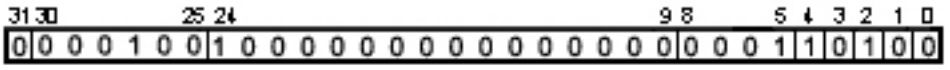

Figura 20. Valores del ESCR para medir el evento x87_FP_uop

En este caso vamos a suponer que nuestro procesador no soporta Hyper-Threading y que queremos medir solamente los eventos de usuario. Para esto, activamos el bit T0_USR.
En la tabla A-1 nos indican que el evento *x87_FP_uop* debe configurarse con el valor 04H en el campo *Event Select*. Además, tenemos que activar el bit ALL como se nos había indicado. Este bit es el bit 15 del *Event Mask*, que corresponde al bit 24 del ESCR.
Activamos el bit *Tag Enable* porque vamos a utilizar el mecanismo de etiquetado en ejecución. Para el campo *Tag Value*, elegimos el valor 0001 (aunque podríamos haber elegido 0010, 0100 o 1000).
En la figura 12 se muestra el ESCR que va a detectar el evento *x87_FP_uop*.

Para el campo *ESCR Select* del CCCR, introducimos el valor que aparece en la tabla A-1. En este caso, es el valor 01H. Activamos el bit *Enable* del CCCR y, al no tener Hyper-Threading, tenemos que fijar por obligación el campo *Active Thread* del CCCR al valor 11.
El CCCR queda como se muestra en la figura 13.

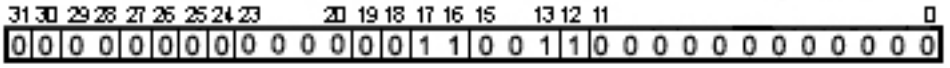

Figura 21. Valores del CCCR para medir el evento *x87_FP_uop*

Evento execution_event

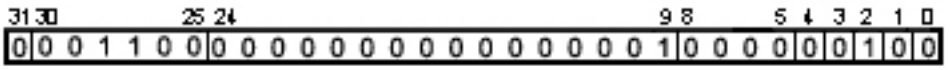

Figura 22. Valores del ESCR para medir el evento *execution_event*

Para detectar el evento *execution_event*, tenemos que introducir en el campo *Event Select* del ESCR el valor 0CH, según la tabla A-2 de la documentación de Intel®.

Como queremos detectar las μ-operaciones etiquetadas en el *upstream*, tenemos que indicar que hay que contar las que tengan el *tag value* a 0001 (al valor que hemos usado en el *upstream*). Para esto, activamos el

bit 0 (NBOGUS0) del campo *Event Mask,* que corresponde al bit 9 del ESCR.
La figura 14 muestra la configuración final del ESCR *downstream*.

Para el CCCR del *execution_event*, activamos el bit *Enable*, fijamos otra vez el campo *Active Thread*=11 e introducimos el valor *ESCR Select* indicado en la tabla A-2 para el *execution_event*, que en este caso es 05H. Al no realizar filtrado, los demás bits los dejamos en cero.
La figura 15 muestra el CCCR para este evento.

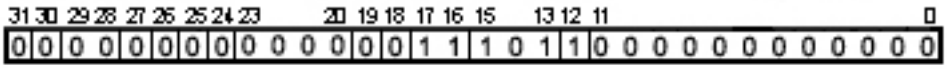

Figura 23. Valores del CCCR para medir el evento execution_event

9.4. Configuración de un evento con etiquetado por replay

Ahora vamos a ver un ejemplo de una métrica que utiliza el mecanismo de etiquetado por *replay*. Vamos a medir las μ-operaciones retiradas que han provocado un fallo, al menos uno, en caché L1. Es evidente la necesidad de etiquetado ya que no queremos calcular el número de fallos totales en caché L1. Una misma μ-operación podría causar más de un fallo en caché pero como vamos a hacer una medida *at-retirement* de las μ-operaciones que están etiquetadas, no importa el número de veces que hayan producido un fallo en caché las μ-operaciones ya que todas se contarán como una.

Miramos, como siempre, en la tabla B-1 del apéndice B del manual *IA-32 Intel® Architecture Optimization: Reference Manual* para ver si aparece la métrica que estamos buscando o alguna similar. Encontramos esta métrica con el nombre *1st-Level Cache Load Misses Retired*. También como siempre, la columna tercera nos dice cómo tenemos que proceder para medir este evento. En este caso hay que configurar el evento *replay_event* con la opción *1stL_cache_load_miss_retired*.
Para buscar este evento, consultamos la tabla A-6 del manual *IA32-Intel Architecture Software Developer's Manual: System Programing Guide*. En esta tabla encontramos el evento *1stL_cache_load_miss_retired,* donde nos explican cómo tenemos que configurar los registros MSRs. Como no se requiere configurar un MSR adicional, nos limitamos a configurar los registros IA32_PEBS_ENABLE y MSR_PEBS_MATRIX_VERT activando los bits que indica esta tabla. Nótese que en este caso no hay que configurar ningún ESCR ni CCCR para el etiquetado.
No hay que olvidarse de configurar el evento *replay_event,* tal y como se indicaba en la tabla B-1. Tenemos que poner la máscara NBOGUS como nos dicen tanto la tabla B-1 del primer manual y la tabla A-6 del segundo.

Primero vamos a configurar el evento *1stL_cache_load_miss_retired*:

Evento 1stL_cache_load_miss_retired

Configuramos los dos registros MSR como se indica en la tabla A-6.
Nota: según esta tabla, habría que activar el bit 25 del MSR IA32_PEBS_ENABLE. Bien, habiendo descubierto que herramientas como Brink & Abyss no activan este bit y observando que activando este bit se obtienen resultados sospechosos, no vamos a activar este bit. Además, parece ser (por la escasa documentación de Intel® al respecto) que este bit sirve para activar el mecanismo de PEBS, que no vamos a utilizar.

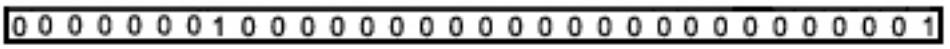

Figura 24. IA32_PEBS_ENABLE para el evento *1stL_cache_load_miss_retired*

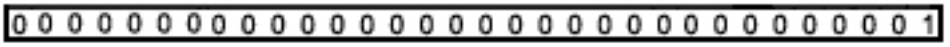

Figura 25. MSR_PEBS_MATRIX_VERT para medir *1stL_cache_load_miss_retired*

Evento replay_event

Consultamos la tabla A-2 para configurar correctamente el ESCR que va a detectar el evento *replay_event*. En *Event Select* debemos fijar el valor 09H. Asociamos, por tanto, el valor 001001 al intervalo [30:25] del ESCR. Como se indicó ates, activamos el bit NBOGUS del *Event Mask*, que corresponde al bit 9 del ESCR.
Medimos para eventos de usuario y sistema operativo en el procesador lógico 0.

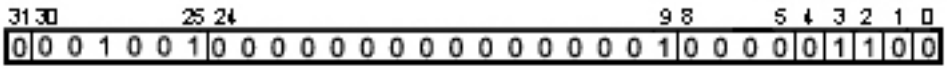

Figura 26. Valores del ESCR para medir el evento *replay_event*

La figura 18 muestra la configuración del ESCR para el evento *replay_event*.

Respecto al CCCR, activamos el bit *Enable* y fijamos *Active Thread* a 11. Para el campo *ESCR Select*, la tabla A-2 nos dice que replay_evemt requiere el valor 05H (al igual que *front_end_event* y *execution_event*). La figura 19 muestra el CCCR para el evento *replay_event*.

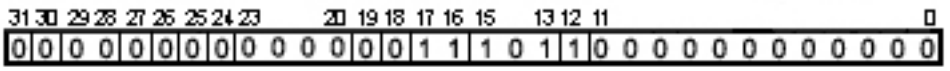

Figura 27. Valores del CCCR para medir el evento *replay_event*

10. Introducción a la herramienta Perfctr

Las librerías *Perfctr* (***Perf****ormance-Monitoring* ***C****oun****t****e****r****s Driver*) dan soporte al kernel de Linux para poder utilizar los contadores de rendimiento del sistema. Esta librería, creada por Mikael Pettersson, da soporte a las siguientes arquitecturas hardware:

- Familias P5 y P6 de procesadores Intel® e Intel® Pentium® 4.
- AMD K7 Athlon
- Cyrix 6x86MX, MII y MIII
- WinChip C6, 2, 2A, 2B y 3

Perfctr admite un modo de uso virtual. Así, cada proceso en el sistema tiene su propio conjunto de contadores virtuales. Esto es, cada proceso ve los contadores como un conjunto privado de contadores propios independientes de las actividades de los demás procesos del sistema. Estos contadores virtuales son de 64 bits (aunque los PMCs del Intel® Pentium® 4 sólo utilizan 40 bits). Además, cada proceso también posee su propio TSC virtual.
Estos contadores virtuales pueden ser modificados evitando la sobrecarga de realizar costosas llamadas al sistema (esto es así para la mayoría de plataformas).

Perfctr utiliza la siguiente estructura para almacenar el valor de todos los registros necesarios para realizar las métricas:

```
struct perfctr_cpu_control {
        unsigned int tsc_on;
        unsigned int nractrs;           // contadores a-mode, número de contadores utilizados
        unsigned int nrictrs;           // contadores i-mode
        unsigned int pmc_map[18];
        unsigned int evntsel[18];               // uno por contador, los CCCRs del Pentium 4

        struct {
                unsigned int escr[18];          // son los ESCRs
                unsigned int pebs_enable;       // para etiquetado por replay
                unsigned int pebs_matrix_vert;  // para etiquetado por replay
        } p4;

        int ireset[18];                         // < 0, para los contadores i-mode
        unsigned int _reserved1;
        unsigned int _reserved2;
        unsigned int _reserved3;
        unsigned int _reserved4;
};
```

11. Utilización de Perfex

Perfex es un programa, que hace llamadas a la librería perfctr. Se utiliza desde la línea de comandos y sirve para medir el valor de los contadores para la arquitectura x86 de un programa completo. Imprime por la salida de error estándar el valor de los contadores. Su sintaxis es la siguiente:

```
perfex [-e event] .. [--p4pe=value] [--p4pmv=value] [-o file]
      command

perfex { -i | -l | -L }
```

Las opciones que ofrece son:

-e event | --event=evento

Especifica un evento para ser contado. Se pueden poner varios eventos. La sintaxis completa es "evntsel/escr@pmc". “eventsel”, “escr” y “pmc” son números de 32 bits que se pueden escribir tanto en notación decimal como en hexadecimal.
evntsel corresponde al registro CCCR. Este campo es obligatorio.
escr corresponde al registro ESCR.
pmc nos dice a qué contador vamos a asignar este evento (PMC). Este valor lo sacamos de la figura 1. No podemos poner el mismo número de contador en dos eventos distintos cuando se cuenta más de un evento.

--p4pe=value | --p4_pebs_enable=value
--p4pmv=value | --p4_pebs_matrix_vert=value

Especifica los valores que se pondrán en los registros PEBS_ENABLE o PEBS_MATRIX_VERT, estos son usados en el etiquetado por replan. La documentación de Intel dice que hay que activar el bit 25 del registro PEBS_ENABLE, pero esto no es así por lo que el driver lo evitará.

-i | --info

Genera una salida identificando el procesador y sus características.

-l | --list

Genera una salida identificando el procesador y sus características. Sdemás nos muestra una lista con los eventos que se pueden contar.

-L | --long-list

Igual que –l pero la lista de eventos es más detallada.

-o fichero | --output=fichero

Escribir los resultado en fichero en vez de la salida estándar de error (*stderr*).

11.1. Ejemplos

Siguiendo los ejemplos utilizados anteriormente, veamos cómo realizar las métricas usando *perfex*:

Con el evento ***1stL_cache_load_miss_retired*** (etiquetado por replay) obteníamos:

ESCR
0001 0010 0000 0000 0000 0010 0000 1100
(1200020CH)
CCCR
0000 0000 0000 0011 1011 0000 0000 0000
(0003B000H)
IA32_PEBS_ENABLE
0000 0001 0000 0000 0000 0000 0000 0001
(01000001H)
MSR_PEBS_MATRIX_VERT
0000 0000 0000 0000 0000 0000 0000 0001
(00000001H)

Para averiguar los contadores que podemos utilizar miramos la fila *Counter numbers per ESCR* del evento en la tabla A-2 y vemos que puede utilizar los contadores 12, 13, 14, 15, 16 y 17. Utilizamos el 12, es decir, C en notación hexadecimal.
Por lo tanto el comando de *perfex* será:

```
perfex -e 0x0003B000/0x1200020C@0x8000000C --p4pe=0x01000001
       --p4pmv=0x1 algún_programa
```

Con el evento ***branch_retired*** obteníamos:

ESCR
0000 1100 0000 0000 0001 1000 0000 0100
(0C001804H)
CCCR
0000 0000 0010 0111 1011 0000 0000 0000
(0027B000H)

Podemos utilizar los contadores 12, 13, 14, 15, 16, 17.
Escogemos, por ejemplo, el 14 (E en hexadecimal)

```
perfex -e 0x00039000/0x0200020A@0x8000000E
```

Con el evento ***memory_loads*** (etiquetado en el *front-end*) obteníamos:

ESCR (uop_type)
0000 0100 0000 0000 0000 0100 0000 0101
(04000405H)
CCCR (uop_type)

0000 0000 0000 0011 0101 0000 0000 0000
(00035000H)

Podemos utilizar los contadores 12, 13, 14, 15, 16, 17. Escogemos el 14 (E en hexadecimal)

ESCR (front_end_event)
0001 0000 0000 0000 0000 0010 0000 0101
(10000205H)

CCCR (front_end_event)
0000 0000 0000 0011 1011 0000 0000 0000
(0003B000H)

Podemos utilizar los contadores 12, 13, 14, 15, 16, 17. El número 14 ya está ocupado, escogemos el 15 (F en hexadecimal)

```
perfex -e 0x00035000/0x04000205@0x8000000E
       –e 0x0003B000/0x10000205@0x8000000F
```

Con el evento ***X87_FP_retired*** (etiquetado en ejecución) obteníamos:

ESCR upstream (x87_FP_uop)
0000 1001 0000 0000 0000 0000 0011 0100
(09000034H)

CCCR (x87_FP_uop)
0000 0000 0000 0011 0011 0000 0000 0000
(00033000H)

Podemos utilizar los contadores 8, 9, 10, 11. Escogemos el 8.

ESCR downstream (execution_event)
0001 1000 0000 0000 0000 0010 0000 0100
(18000205H)

CCCR (execution_event)
0000 0000 0000 0011 0011 0000 0000 0000
(00033000H)

Podemos utilizar los contadores 12, 13, 14, 15, 16, 17. Escogemos el 14 (C en hexadecimal)

```
perfex -e 0x00033000/0x18008035@0x80000008
       -e 0x0003B000/0x18000205@0x8000000C
```

Apéndice A. Instalación de Perfctr

La instalación de *Perfctr* consta de tres partes. Primero tenemos que compilar un módulo para el kernel, para lo cual tendremos primero que parchear el kernel y después compilarlo. Después tendremos que crear un archivo de dispositivo. Por último, tendremos que compilar las librerías de usuario para la utilización de *Perfctr*.

A.1. Parcheado y compilación del kernel

Perfctr consta de dos partes: el controlador propiamente dicho y los parches necesarios para integrarlo en distintas versiones del kernel.

Los pasos a seguir son los siguientes:

0. Obtener el código fuente del kernel que esté soportado, esto se puede saber mirando los parches existentes en el directorio *patches* del código fuente de *Perfctr*.

Vamos a suponer que PDIR es el directorio raíz donde he descomprimido el paquete *Perfctr.*

Una versión del kernel, digamos KVER, está soportada si y sólo si se cumple una de las siguentes condiciones:

- $PDIR/patches/ contiene un fichero "patch-kernel-$KVER".
- $PDIR/patches/aliases contiene una línea que comienza por $KVER.

Entre los kernels soportados se incluyen muchos kernels estándar (www.kernel.org) y algunos kernels de RedHat. Los kernels de RedHat están identificados por el sufijo "-redhat" añadido a la versión del kernel.

Cualquier kernel que no cumpla lo anterior no está soportado.

1. Desempaqueta las fuentes de kernel. Sea KDIR su directorio raíz.

En el terminal, hacer un 'cd' a $KDIR.

Guarda una copia del fichero *.config* si habías compilado el kernel con anterioridad, esto es porque tal vez lo necesites en otro momento. Después ejecutar:

```
make mrproper
```

Esto es muy importante porque si no se hace seguramente falle.

2. Aplica todos los parches al kernel que necesites.

Siendo $KDIR el directorio actual, ejecutar:

```
$PDIR/update-kernel
```

Esto parcheará el kernel con el parche apropiado instalando los nuevos ficheros del controlador de *Perfctr*.

Si *update-kernel* no encuentra el parche apropiado fallará con un mensaje de error. Si de todas maneras quieres utilizar una versión V1 del kernel no soportada pero crees que el parche para una versión V2 funcionará, puedes probarlo de la siguiente manera:

```
$PDIR/update-kernel --test --patch=$V2
```

Esto aplica el parche en modo de prueba sin modificar ningún fichero. Si el parche se aplica limpiamente sin ningún fallo puedes forzar el uso de este parche quitando la opción --test en *update-kernel*. Este modo de funcionamiento está indicado sólo para gente experimentada en la compilación del kernel.

Finalmente, edita $KDIR/Makefile y cambia EXTRAVERSION para incluir al único para este kernel, como puede ser añadiendo al final "-perfctr" (sin las comillas). Este paso es recomendable (aunque no necesario) si estas instalando el controlador de *Perfctr* en una versión del kernel que ya está instalada en tu máquina.

3. Si has guardado el fichero .config, cópialo a $KDIR. A continuación configura el kernel con tu herramienta favorita, por ejemplo 'make menuconfig', 'make config' o 'make oldconfig'.

Deberías configurarlo con soporte para módulos (CONFIG_MODULES=y) y las versiones de éstos (CONFIG_MODVERSIONS=y), o completamente sin módulos (CONFIG_MODULES=n). Configurar el kernel con soporte para los módulos pero no soportar las versiones es una configuración insegura y por lo tanto no recomendable.

También deberías habilitar al menos CONFIG_PERFCTR, CONFIG_PERFCTR_VIRTUAL, y CONFIG_PERFCTR_GLOBAL.

Puedes seleccionar CONFIG_PERFCTR=m para compilar el controlador como un módulo del kernel. El módulo se llamará 'perfctr'.

Nota: El parche del kernel de perfctr añade una palabra al tipo `thread_struct', esto hace que el binario del kernel parcheado sea incompatible con el binario de uno no parcheado. Esto es el porqué es importante distinguir entre el kernel parcheado con EXTRAVERSION y CONFIG_MODVERSIONS.

4. Ejecutar:

```
make dep vmlinux modules
```

Esto compila el kernel y sus módulos.

5. Como root, ejecutar:

```
make modules_install
```

Esto es para instalar los módulos dentro de /lib/modules/.

6. Como root, edita /etc/lilo.conf para incluir una nueva entrada para el nuevo kernel. Copia una configuración ya existente y edita las líneas que tienen *image=* y *label=* para reflejar el nombre del nuevo kernel, incluyendo lo que pusiste en EXTRAVERSION.

A continuación, teclear:

```
make install
```

Esto instala la nueva imagen del kernel y actualizarel gestor de arranque.
Si no usas LILO, tendrás que adaptar este paso a tu gestor de arranque.

7. Reinicia la máquina.

A.2. Fichero de dispositivo

La parte que corresponde al kernel del paquete está implementada como un controlador de dispositivo de tipo carácter, el cual ha sido asignado como número mayor 10 y número menor 182. La primera vez que instalas el paquete y fichero especial que representa este dispositivo tiene que ser creado. Para ello ejecuta como root lo siguiente:

```
mknod /dev/perfctr c 10 182
chmod 644 /dev/perfctr
```

Si el controlador ha sido compilado como módulo habrá que cargarlo en el kernel antes de ser utilizado. Esto ocurrirá de manera automática si el kernel se compiló con soporte para ello (CONFIG_KMOD=y). Para kernels 2.6 no se necesita hacer ninguna otra cosa. Para kernels 2.4, lo siguiente debería ser añadido al fichero /etc/modules.conf:

```
alias char-major-10-182 perfctr
```

A.3. Librerías

Para compilar la librería, los ficheros de *include* y los programas de ejemplo, ejecutar:

```
make
```

Para instalar la librería la aplicación 'perfex', ejecutar:

```
make PREFIJO=$PREFIJO install
```

Esto instalará los binarios en $PREFIJO/bin, las librerías en $PREFIJO/lib y los ficheros de include en $PREFIJO/include. Cada uno de estos destinos puede ser sobrescrito individualmente:

```
make BINDIR=$BINDIR LIBDIR=$LIBDIR INCLDIR=$INCLDIR install
```

Brink & Abyss

Contenido

1. Introducción

Uno de los ingenieros que intervino en el diseño de los mecanismos de monitorización de eventos del procesador Intel® Pentium® 4, Brinkley Sprunt, decidió escribir una herramienta que permitiese un fácil acceso a estos mecanismos con una interfaz de alto nivel.
Brink & abyss es una herramienta para Linux que, como hemos indicado, proporciona un interfaz de alto nivel para utilizar los contadores hardware del procesador Intel® Pentium® 4.
La última actualización disponible, la versión 2.0, es de Septiembre de 2002.

Brink & abyss permite medir eventos normales y eventos que requieren etiquetado (*tagged events*). También, con la versión 2.0, soporta los mecanismos EBS (***E****vent* ***B****ased* ***S****ampling*) y PEBS (***P****recise* ***E****vent* ***B****ased* ***S****ampling*).
Además, permite utilizar contadores en cascada y el control del incremento de los contadores basado en comparaciones (*edge, threshold, complement, ...*).

2. Limitaciones

Uno de los mayores inconvenientes de *brink & abyss* es que no permite analizar fragmentos aislados de código, sino que nos obliga a analizar todo el programa sin poder centrarnos en alguna función, bucle o módulo específico del programa.

Como se ha indicado, la herramienta sólo funciona para procesadores Intel® Pentium® 4 bajo Linux (a partir de 2.4.x). El hecho de que sólo pueda usarse en esta plataforma se debe a que se usan llamadas al sistema nativas de Linux. Además, hay que “parchear” el kernel de Linux para poder utilizar la herramienta.

Brink & abyss sólo soporta sistemas uniprocesador, y tampoco soporta la tecnología *Hyper-Threading* (HT) existente en algunos modelos Intel® Pentium® 4.

3. Funcionamiento y programas

Brink & abyss se compone de cuatro programas distintos:

- *brink*, el programa principal que ejecutaremos.
- *abyss*, para gestionar los contadores hardware.
- *abyss_dev*, un driver o controlador utilizado por *abyss*.
- *"parche" EBS* para el kernel, necesario para utilizar EBS y PEBS.

Además, el script *brink* requiere la instalación de algunos módulos Perl, como veremos.

Brink es un programa escrito en perl (*perl script*) que actúa de *front-end* para configurar los MSR del Intel® Pentium® 4, los cuales controlan los contadores de rendimiento del procesador.

Abyss es el programa que realmente inicializa los registros MSR. Este programa, escrito en C, es ejecutado por *brink*, de forma que no se debe ejecutar aisladamente. *Abyss* es el interfaz para *abyss_dev*.

Abyss_dev es un controlador para el kernel 2.4 de Linux que permite controlar a bajo nivel las características de los contadores hardware para monitorización. Este controlador se simboliza en Linux como */dev/abyss*.
A su vez, este controlador requiere parchear al kernel con los parches EBS.

Los parches EBS son necesarios para permitir los mecanismos de EBS y PEBS. Permiten la instalación de un manejador de interrupciones producidas por el desbordamiento (*overflow*) de los contadores. Estas interrupciones son generadas por el APIC local de los procesadores Intel® Pentium® 4 para dar soporte a los mecanismos EBS y PEBS.

4. Árbol de directorios

A continuación mostramos los directorios que se crean al descomprimir la herramienta *brink & abyss* (versión 2.0). Por simplicidad, sólo aparecen los ficheros o directorios más significativos, indicando con puntos suspensivos el resto de ficheros.

```
abyss_dev
      | Makefile
      | abyss_dev.c
      | abyss_dev.c
      | abyss_dev_2_4.c
      | abyss_asm.h
      | abyss_dev.h
      | abyss_lib.h
      | abyss_p4_emon.h

brink
      | brink
      | pentium4_emon.txt
      | Readme.txt

doc
      | pentium4_emon.pdf
      | Readme.txt
      ...

examples
      | progs
            ...
      | cascade_exp.txt
      | craft_exp.txt
      | cross_exp.txt
      | ebs_exp.txt
      | exec_tag_exp.txt
      | fields_exp.txt
      | front_end_pebs_exp.txt
      | front_end_tag_exp.txt
      | pebs_vs_iebs_exp.txt
      | replay_pebs_exp.txt
      | replay_tag_exp.txt
      | simple_exp.txt
      ...
```

```
abyss_front_end
      | Makefile
            | abyss.c
            | Readme.txt

ebs_kernel_patches
      | ebs-kernel-patch-2.4.14
      | ebs-kernel-patch-2.4.17
      | ebs-kernel-patch-2.4.18

perl_modules
      | Storable-1.0.13.tar.gz
      | XML-Parser.2.30.tar.gz
      | XML-Simple-1.05.tar.gz
      ...
```

5. Instalación de Brink & Abyss

A continuación indicamos y explicamos los pasos necesarios para instalar la herramienta *brink & abyss* en Linux.

5.1. Compilar el kernel con el parche EBS

Obtener un kernel 2.4.x y parchearlo
Lo primero es disponer de las fuentes del kernel de Linux 2.4.x (en principio, los parches son para la versión 2.4. del kernel, para versiones más modernas del kernel habría que probar la compatibilidad). Las fuentes del kernel se pueden obtener en www.kernel.org.
De los tres parches EBS que incluye *brink & abyss*, sólo necesitamos utilizar uno, el que corresponde a nuestra versión de kernel.
Una vez que tenemos dichas fuentes descomprimidas es el momento de aplicar el parche EBS que incluye la herramienta *brink & abyss* (en el directorio *ebs_kernel_patches*).
Cambiar el directorio actual al directorio creado al descomprimir las fuentes del kernel 2.4.x. Desde este directorio vamos a aplicar el parche EBS.

```
patch –p5 <ruta al parche EBS adecuado>
```

Configurar el kernel 2.4. parcheado
Debemos estar en el directorio “linux” donde se encuentra nuestro kernel 2.4. ya parcheado.
Para restaurar los valores por defecto de configuración, ejecutamos:

```
make mrproper
```

Ahora pasamos a configurar las opciones del kernel. Aunque hay varias formas de hacerlo, una es ejecutando:

```
make xconfig
```

Aquí debes configurar el kernel con las opciones que desees. Sin embargo, para dar soporte a *brink & abyss* debes activar ciertas opciones (disponibles debido al parche EBS aplicado anteriormente).

Processor type and features | *Processor family*
Pentium 4

(esto activa la opción CONFIG_MPENTIUM4)

Processor type and features | *Symmetric multi-processing support*
N (no)

(esto es necesario porque *abyss_dev* no soporta multiprocesadores ni Hyper-Threading)

Processor type and features | *Local APIC support on uniprocessors*
Y (yes)

(esto activa la opción CONFIG_X86_LOCAL_APIC)

Guarda la configuración y sal de *xconfig*.

Compilar el kernel 2.4. parcheado y configurado
Para ver si hay problemas de dependencias, ejecutamos:

```
make dep
```

Si hubiera errores de dependencias tendríamos que volver a configurar las opciones del kernel hasta que no haya conflictos. Una vez hecho esto, compilamos el kernel con:

```
make bzImage
```

Esto puede tardar un rato, dependiendo del tamaño del kernel que estemos compilando. Una vez terminado, y sin errores, copiamos la imagen creada en el directorio de arranque. Ejecutamos, como usuario *root*, lo siguiente:

```
cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.17-ebs
cp System.map /boot/System.map-2.4.17-ebs
```

Ahora falta compilar los módulos del kernel:

```
make modules
make modules_install
```

Configurar el gestor de arranque
Para que al arrancar el sistema podamos elegir el nuevo kernel tenemos que editar nuestro gestor de arranque. Si, por ejemplo, usamos LILO tenemos que editar */etc/lilo.conf*. En un sistema particular, por ejemplo, habría que añadir algo así:

```
image=/boot/vmlinuz-2.4.17-ebs
label=2.4.17-ebs
read-only
root=/dev/hda5
```

La primera línea indica la ruta del kernel en el sistema, aquí mantenemos la nomenclatura que usamos más arriba. Los demás valores pueden (o deben) ser distintos en cada sistema.
Una vez editado */etc/lilo.conf*, tenemos que actualizar los cambios con el comando:

```
lilo –v
```

En caso de utilizar otro gestor de arranque (como pudiera ser GRUB), los pasos a seguir serán distintos. Una vez configurado el gestor para que reconozca el nuevo kernel reiniciamos el sistema:

```
reboot
```

Al arrancar debe aparecer nuestro kernel (si usamos LILO, el kernel tendrá el nombre que pusimos en la propiedad *label* de *lilo.conf*). Lo seleccionamos y esperamos a que arranque Linux.

Para comprobar que estamos usando el kernel que hemos compilado, podemos ejecutar el siguiente comando:

```
uname –r
```

Si hemos seguido los pasos anteriores, debería mostrarnos *2.4.17-ebs.*

5.2. Compilar e instalar el controlador abyss_dev

Modificar el archivo Makefile

Suponiendo que estamos en el directorio de instalación de *brink & abyss*, accedemos al directorio *./abyss_dev/*. Aquí encontramos varios archivos de sigo fuente (*.c* y *.h*) y un archivo *Makefile* para generar el binario.

Sólo hay que modificar *Makefile* para indicarle la ruta del directorio *include* del kernel EBS que construimos en el paso 1. Esta ruta depende de cada sistema específico. Tenemos que sustituir la siguiente línea del *Makefile* por nuestra ruta concreta:

```
INC_FLAGS = -I /home/kernel/linux-2.4.17-ebs/linux/include/
```

Construir abyss_dev.o y configurarlo como controlador /dev/abyss

Una vez modificado el archivo Makefile ya podemos compilar los archivos fuente y construir abyss_dev.o. Desde ./abyss_dev/ ejecutamos:

```
make
```

Ahora tenemos que crear un nodo para este controlador en */dev* con el siguiente comando, con el usuario *root*:

```
mknod /dev/abyss c 10 243
chmod 666 /dev/abyss
```

Cargar el controlador /dev/abyss

Para poder usar este controlador tenemos que cargarlo en el kernel, ya que se trata de un módulo. Para ello ejecutamos el comando:

```
insmod abyss_dev.o
```

Cada vez que reiniciemos el sistema, debemos ejecutar esta orden antes de poder utilizar *brink & abyss* u otro programa que utilice

/dev/abyss. Evidentemente, podemos incluir este comando en un *script* de inicio.

5.3. Compilar abyss (abyss_front_end)

Abyss es, como dijimos, un programa en C que utiliza *abyss_dev* para poder configurar los registros MSR del Intel® Pentium® 4.
Para compilarlo sólo tenemos que ejecutar, desde *./abyss/*, el siguiente comando:

```
make all
```

Una vez que tenemos el archivo binario, debemos colocarlo en un directorio de nuestro sistema que esté incluido en la variable de entorno PATH.

5.4. Instalar los módulos Perl

Brink, que como ya hemos indicado se trata de un script escrito en Perl, requiere algunos módulos para poder utilizarse. Se trata concretamente de tres módulos: *Storable, XML-Simple, XML-Parser*.

Si no disponemos de estos módulos en nuestro sistema, podemos descargarlos de http://www.cpan.org. De todos modos, la distribución de *brink & abyss* incluye en el directorio *./perl_modules* los tres módulos. La instalación de estos módulos es muy sencilla y se explica en la documentación asociada a cada uno de ellos.

5.5. Instalar brink

Una vez instalados los tres módulos de Perl, ya procedemos a instalar *brink*. Lo único que tenemos que hacer es copiar el archivo *./brink/brink* en algún lugar de nuestro sistema que esté incluido en la variable PATH. Por defecto, cuando ejecutemos *brink* éste buscará *abyss* y el archivo de configuración en el directorio en el que está el propio *brink*. Este comportamiento podemos modificarlo cambiando las siguientes propiedades en *brink*:

```
$default_path_to_config_file = "./pentium4_emon.txt";
$default_path_to_abyss = "./abyss";
```

Si no disponemos de estos módulos en nuestro sistema, podemos descargarlos de http://www.cpan.org. De todos modos, la distribución de *brink & abyss* incluye en el directorio *./perl_modules* los tres módulos. La instalación de estos módulos es muy sencilla y se explica en la documentación asociada a cada uno de ellos.

6. Fichero de configuración

Brink necesita recibir dos ficheros XML de entrada: un fichero de experimentos y un fichero de configuración. Los resultados de las medidas efectuadas por *brink & abyss* se escriben en varios ficheros de salida (también XML).
Brink & abyss incluye un fichero de configuración (*pentium4_emon.txt*) y varios ficheros de experimentos a modo de ejemplo.

Para ejecutar *brink*, se necesita tener un fichero de configuración.
El fichero de configuración es un fichero XML que define los eventos que vamos a poder medir con *brink & abyss*, cada uno con sus posibles variantes. Además, este fichero define algunas características de los MSR del Pentium 4, como sus direcciones físicas o el significado de sus bits.
Normalmente siempre usaremos el fichero **pentium4_emon.txt** (el que se incluye en la herramienta *brink & abyss*), aunque para introducir nuevos eventos o modificar algunos existentes podríamos crear otro fichero de configuración a partir del original.

Según Intel® vaya documentando o implementando nuevas características relativas a la monitorización de eventos, basta con modificar el archivo de configuración que estemos usando para dar soporte a estas nuevas características. Además, nuevas descripciones para *eventos etiquetados* pueden añadirse al fichero de configuración para que sirvan como eventos base a la hora de utilizarse en los ficheros de experimentos.

Al ejecutar *brink*, le indicamos el fichero de configuración que estamos usando mediante la siguiente sintaxis:

```
–config <fichero_de_configuración>
```

Si no incluimos el parámetro *–config*, *brink* utiliza por defecto el fichero de configuración indicado en la propiedad *$default_path_to_config_file* en el archivo *brink.*

Para poder “parsear” el fichero de configuración, *brink* requiere el uso del módulo *XML::Simple*.

7. Fichero de experimentos

Es necesario definir exactamente los eventos que queremos medir y el programa o programas sobre los que vamos a realizar las mediciones. Esta información se la proporcionaremos a *brink* en un fichero de experimentos.
El fichero de experimentos es otro fichero XML donde se indican precisamente los experimentos que queremos considerar y los programas que queremos analizar. En el fichero de experimentos se pueden especificar uno o más programas a ejecutar y uno o más experimentos a considerar.
La ejecución de *brink & abyss* requiere uno (y sólo uno) fichero de experimentos.

Un *experimento* es un conjunto de eventos. Todos los eventos de los que consta cada experimento deben ser *compatibles* entre sí, es decir, dichos eventos han de poder medirse simultáneamente (en la misma traza de ejecución). Para que esto pueda realizarse, no puede haber conflictos entre los registros MSR que necesita cada evento.
Por esto, la necesidad de definir varios experimentos en el mismo fichero de experimentos surge por esta razón. No es posible medir ciertas combinaciones de eventos simultáneamente debido a que la arquitectura del Intel® Pentium® 4 dispone, como es lógico, de un número limitado de registros MSR.

Además, en el fichero de experimentos también hay que indicar sobre qué programa o programas realizamos las medidas.
Un *job* es una combinación de un programa y un experimento. *Brink* genera un directorio por cada *job*, donde almacena varios ficheros de salida con los resultados de las medidas y el fichero de entrada que recibe *abyss* desde *brink*.

Brink & abyss ejecuta cada programa que queremos analizar una vez por cada experimento que hayamos definido en el fichero de experimentos. En cada ejecución, se toman las medidas de los eventos pertenecientes al experimento y programa que se están considerando. (Ejemplo: para 2 programas y 3 experimentos habría 6 *jobs* y *brink & abyss* ejecutaría cada programa 3 veces, una vez por cada experimento).

Para cada *job*, *brink & abyss* contará el número de ocurrencias de los eventos que constituyen el experimento a considerar, desde el comienzo de la ejecución del programa hasta el final del mismo. *No podemos medir fragmentos de código aislados (un bucle, una función, ...).*

El fichero de experimentos lo indicamos con el siguiente parámetro:

```
–exp <fichero_de_experimentos>
```

El uso del parámetro *–exp* es obligatorio (ya que, ¿qué fichero de experimentos debería tomar *brink* por defecto?).

Brink & abyss incorpora, a modo de ejemplo, varios ficheros de experimentos en *./examples*.

8. Ficheros de resultados

Brink & abyss, al ejecutarse satisfactoriamente, genera varios ficheros de resultados con los valores de los contadores. Todos estos ficheros se escriben en un directorio de salida que se indica con el siguiente parámetro:

```
-outdir <directorio_con_los_resultados>
```

Por defecto, brink & abyss escribe los ficheros en el directorio ./emon_data si no utilizamos el parámetro –outdir.

En el directorio de salida, brink & abyss escribe los resultados con todas las medidas para todos los jobs considerados. Se crean cuatro ficheros:

```
emon_config.txt
exp_config.txt
events_counts.<nombre_fichero_experimentos>.txt
summary.<nombre_fichero_experimentos>.txt
```

Los ficheros *emon_config.txt* y *exp_config.txt* son copias de los ficheros de configuración y experimentos utilizados, respectivamente. El fichero event_counts muestra los valores de las medidas para todos los jobs en un formato sencillo de tabla. Por último, el fichero *summary* contiene información más detallada de los resultados para todos los jobs (fecha, información sobre los valores de los registros MSRs, …)

Además de estos ficheros, se creará un subdirectorio por cada job. Es decir, si en el fichero de experimentos usado en la ejecución hay x programas y z experimentos, se generarán x·z subdirectorios (jobs). Los nombres de estos subdirectorios serán de la forma:

```
job.<nombre_programa>.<nombre_experimento>
```

Dentro de cada subdirectorio se crean cinco ficheros con datos sobre los resultados de las medidas para el job al que corresponde el subdirectorio. Estos ficheros son:

```
job.<nombre_programa>.<nombre_experimento>.abyss_input.txt
job.<nombre_programa>.<nombre_experimento>.raw.txt
job.<nombre_programa>.<nombre_experimento>.delta.txt
job.<nombre_programa>.<nombre_experimento>.raw.txt
summary.<nombre_fichero_experimentos>.txt
```

El fichero abyss_input es el que brink le pasa a abyss para que éste pueda realizar las medidas de los eventos determinados. Este fichero especifica la inicialización de los registros MSR para el job y le indica a abyss los contadores que debe leer para realizar cada muestra (sample).

El fichero raw contiene una fila para cada muestra realizada por *abyss*. En cada muestra, se indica el valor de los contadores y el TSC (***T****ime* ***S****tamp* ***C****ounter*).

El fichero delta informa del incremento o diferencia entre cada par de muestras consecutivas. Además, contiene una columna con el valor raw del TSC para cada incremento (delta).

Para cada job, brink & abyss ejecuta la orden /usr/bin/time para que el sistema operativo acumule información sobre el job según se está ejecutando. La salida del comando se introduce en una línea en el fichero time.

Por último, se crea un fichero con información detallada sobre los resultados de las medidas para el job concreto. Este fichero summary es como el fichero summary principal que se crea en el directorio de resultados, pero este caso solamente relativo a un job.

Excepcionalmente y sólo en los casos en que se haya utilizado el mecanismo de EBS, se creará un sexto fichero:

job.<nombre_programa>.<nombre_experimento>.ebs_samples.txt

Es en este fichero se recogen las muestras tomadas.

9. Definición y configuración de registros MSR

Como hemos indicado, es en el fichero de experimentos donde hay que indicar los eventos cuyo número de ocurrencias queremos contar. Sólo podemos medir los eventos que estén definidos en el fichero de configuración que estamos usando.
En el fichero de configuración se define la estructura de los registros ESCR, CCCR y contadores (por supuesto, fiel a la documentación de Intel®). El fichero *pentium4_emon.txt* incluido con la distribución de *brink & abyss* define así estos registros:

```
<escr_config size="64">
      <reserved1    bits="0-1"      default="00"/>
      <usr          bits="2"        default="0"/>
      <os           bits="3"        default="0"/>
      <tag_enable   bits="4"        default="0"/>
      <tag0         bits="5"        default="0"/>
      <tag1         bits="6"        default="0"/>
      <tag2         bits="7"        default="0"/>
      <tag3         bits="8"        default="0"/>
      <event_mask   bits="9-24"     default="0000000000000000"/>
      <event_select bits="25-30"    default="000000"/>
      <reserved2    bits="31"       default="0"/>
      <reserved3    bits="32-63"
                   default="00000000000000000000000000000000"/>
</escr_config>
```

```
<cccr_config size="64">
      <reserved1   bits="0-11"      default="000000000000"/>
      <enable      bits="12"        default="1"/>
      <escr_select bits="13-15"     default="000"/>
      <reserved2   bits="16-17"     default="11"/>
      <compare     bits="18"        default="0"/>
      <complement  bits="19"        default="0"/>
      <threshold   bits="20-23"     default="0000"/>
      <edge        bits="24"        default="0"/>
      <force_ovf   bits="25"        default="0"/>
      <ovf_pmi     bits="26"        default="0"/>
      <reserved3   bits="27-29"     default="000"/>
      <cascade     bits="30"        default="0"/>
      <ovf         bits="31"        default="0"/>
      <reserved4   bits="32-63"
                   default="00000000000000000000000000000000"/>
</cccr_config>
```

```
<counter_config size="64">
      <counter    bits="0-39"
      default="0000000000000000000000000000000000000000"/>
      <reserved   bits="40-63"
      default="000000000000000000000000"/>
</counter_config>
```

Esta definición de los registros es la que se debería utilizar siempre, dado que se basa en la documentación de Intel® y no tendría ningún sentido modificar estos valores (aunque se podría hacer).
Como vemos, lo que se hace básicamente es asignar un nombre a un bit o a un conjunto de bits. Después, en la definición de eventos (en el fichero de configuración) o en la utilización de eventos (en el fichero de experimentos) podremos referirnos a estos bits y darles valor utilizando estos nombres. Además todos los bits tienen un valor por defecto, que se indica mediante el atributo *default*.

Además, se definen otros dos registros que se utilizan para medir eventos etiquetados mediante el mecanismo de *replay*,

```
<pebs_enable size="64" address="0x3f1">
      <l0miss           bits="0"    default="0"/>
      <l1miss           bits="1"    default="0"/>
      <dtlbmiss         bits="2"    default="0"/>
      <reserved1        bits="3-8"  default="000000"/>
      <mob              bits="9"    default="0"/>
      <split            bits="10"   default="0"/>
      <reserved2        bits="11-23" default="0000000000000"/>
      <tag              bits="24"   default="0"/>
      <enable_pebs      bits="25"   default="0"/>
      <reserved3  bits="26-63" default="0000000000000000000
                                        0000000000000000000"/>
</pebs_enable>
```

```
<pebs_matrix_vert size="64" address="0x3f2">
      <ld         bits="0"    default="0"/>
      <st         bits="1"    default="0"/>
      <reserved1  bits="2-63" default="0000000000000000000000000
                                       0000000000000000000000000
                                       0000000000000000"/>
</pebs_matrix_vert>
```

Además, en el fichero de configuración se listan todo los eventos que se pueden medir, divididos en dos grupos: normales y etiquetados (*tagged*). Los eventos normales han de estar incluidos entre las etiquetas `<normal_events>` y `</normal_events>`. Los eventos etiquetados aparecerán entre las etiquetas `<tag_events>` y `</tag_events>`.
Como para cada evento concreto los bits *event mask* del ESCR adquieren un significado específico, daremos nombres específicos a estos bits (a los que tenga sentido utilizar) en cada definición de evento. A estos bits los llamaremos bits de configuración del evento.

Aunque no es muy normal hacerlo en la definición de eventos normales, sí que resulta necesario en los eventos etiquetados fijar ciertos bits de los registros CCCR o ESCR que utilizan. Podemos fijar el valor de los bits empleando las siguientes etiquetas:

- `<set></set>`, pone a “1” los bits que aparezcan entre ambas etiquetas.
- `<clr></clr>`, pone a “0” los bits que aparezcan entre ambas etiquetas.

Por ejemplo, para activar los bits 18 y 19 del ESCR y desactivar el bit 2 del CCCR haríamos:

```
<set>
      <compare/>
      <complement/>
</set>
<clr>
      <usr/>
</clr>
```

Además, cuando lo que queremos es dar un valor a un conjunto de bits utilizamos el atributo `val`. Por ejemplo,

```
<threshold val="0110"/>
```

fijaría el valor los bits 20, 21, 22 y 23 del CCCR con los valores indicados. Los bits que no se ponen explícitamente a “0” ni a “1” tendrán el valor por defecto definido en la definición de los registros en el fichero de configuración, como se indicó más arriba.

10. Ejemplos de definición de eventos

A continuación veremos varios ejemplos de definición de eventos. Estos ejemplos están tomados de eventos definidos en el fichero de configuración *pentium4_emon.txt*.
Primero analizamos la definición de un evento normal. A continuación, veremos tres ejemplos de eventos etiquetados, uno con cada mecanismo (*front end*, *ejecución* y *replay*).

Finalmente veremos cómo añadir un nuevo evento a nuestro fichero de configuración.

Es evidente que para definir cada evento es imprescindible consultar los manuales de Intel®, donde se indica cómo configurar los bits de los registros MSR de la forma adecuada.

10.1. Eventos normales

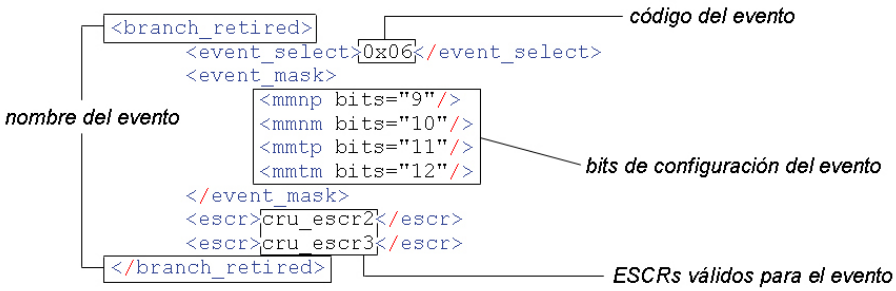

Figura 28. Ejemplo de definición de métrica *non-retirement*

Como vemos, cada evento normal consta de un nombre con el que abrimos y cerramos el cuerpo del evento (en este caso, *branch_retired*). Este nombre lo elegiremos con total libertad, pero conviene que bien el evento que representa. En el campo `<event_select>` indicamos el código del evento que queremos medir (proporcionado por Intel®). En `<event_mask>` es donde incluimos todas las opciones de configuración que permite el evento. En este caso hay cuatro bits de opciones y según los que activemos en el fichero de experimentos mediremos una cosa u otra. En este caso concreto, si en un fichero de experimentos activáramos el bit 10 (*mmnm*), contaríamos el número de saltos no tomados que se han predicho erróneamente. Los nombres *mmnp*, *mmnm*, *mmtp* y *mmtm* podrían ser otros, la única condición es usar los mismos nombres en los fichero de experimentos que utilicen este evento.
Para medir un evento normal se necesita un registro ESCR libre, pero cada evento sólo puede elegir un ESCR de entre un pequeño subconjunto de estos registros. Con las etiquetas `<escr>` indicamos los registros ESCR candidatos ser utilizados para este evento. En este caso, indicamos *cru_escr2* y *cru_escr3.* Estos nombres están definidos en el propio fichero de configuración y podríamos cambiarlos por otros (por ejemplo, *cru _1* y *cru_2*).

10.2. Eventos etiquetados

Los eventos etiquetados son algo más complicados. Estos eventos requieren dos componentes, el que hace el etiquetado (*upstream*) y el que cuenta número de instrucciones etiquetadas que son retiradas (*downstream*). La definición de estos eventos se divide en tres partes:

- `<type></type>`, donde indicamos que tipo de etiquetado se realiza. Hay tres tipos, *front_end*, *execution* y *replay*.
- `<tag_setup></tag_setup>`, donde indicamos el componente que hace el etiquetado, es decir, el que etiqueta las instrucciones que experimentan un evento. Por tanto, aquí debemos poner el nombre de un evento normal definido en el fichero de configuración. Podemos configurar el evento usando las marcas `<set>, <clr> y <val>`.
- `<count_setup></count_setup>`, donde indicamos el componente que realiza la cuenta de las instrucciones etiquetadas. Por tanto, aquí debemos poner el nombre de uno de los tres eventos normales que sirven para este fin. Para *pentium4_emon.txt*, estos son *front_end_event, execution_event* y *replay_event*.

10.3. Eventos etiquetados en el front end (*front end tagging*)

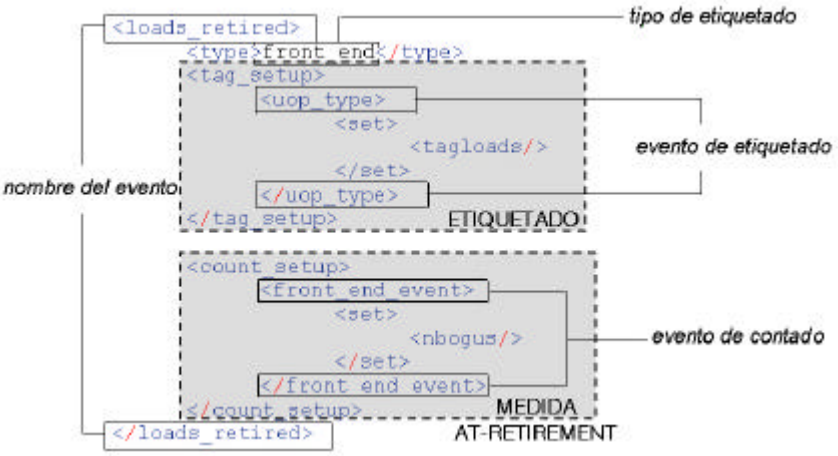

Figura 29. Ejemplo de definición de métrica con etiquetado en el *front-end*

Vamos a definir un evento que utilice el mecanismo de etiquetado en el *front end*. Decidimos dar el nombre *loads_retired* a este evento.
Como el evento que deseamos medir se trata de un evento etiquetado en el *front end*, introducimos *front_end* en la etiqueta `<type>`.
El evento de etiquetado es *uop_type*, cuya definición como evento normal es:

```
<uop_type>
      <event_select>0x02</event_select>
      <event_mask>
            <tagloads bits="10"/>
            <tagstores bits="11"/>
      </event_mask>
      <escr>rat_escr0</escr>
      <escr>rat_escr1</escr>
```

```
</uop_type>
```

Activamos el bit *tagloads* para medir solamente las lecturas.
En <*count_setup*>, introducimos el nombre del evento encargado de realizar el mecanismo de etiquetado para los eventos de ejecución, que en este caso sólo puede ser *front_end_event*. Activamos el bit *nbogus* de dicho evento para medir solamente las instrucciones que se retiran con éxito. En el fichero *pentium4_emon.txt*, este evento se define:

```
<front_end_event>
      <event_select>0x08</event_select>
      <event_mask>
            <nbogus bits="9"/>
            <bogus bits="10"/>
      </event_mask>
      <escr>cru_escr2</escr>
      <escr>cru_escr3</escr>
      <available_for_pebs/>
</front_end_event>
```

10.4. Eventos etiquetados en ejecución (*execution tagging*)

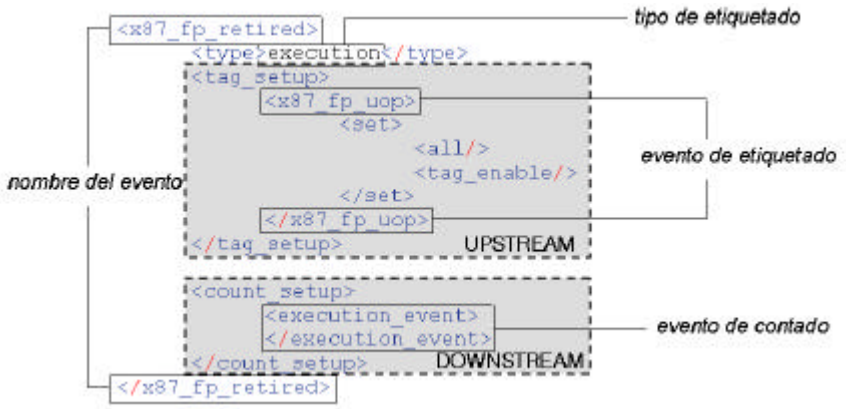

Figura 30. Ejemplo de definición de métrica con etiquetado en ejecución

Lo primero es especificar el nombre del evento, en este caso *x87_fp_retired*. Tenemos que indicar que tipo de etiquetado se precisa. Ya que el evento que deseamos medir es un evento de etiquetado en ejecución, introducimos *execution.*
Aquí hemos utilizado el evento de etiquetado *x87_fp_uop*, cuya definición es:

```
<x87_fp_uop>
      <event_select>0x04</event_select>
      <event_mask>
            <all bits="24"/>
      </event_mask>
      <escr>firm_escr0</escr>
      <escr>firm_escr1</escr>
</x87_fp_uop>
```

Además, activamos los bits *all* (definido en el evento *x87_fp_uop*, bit 24 del ESCR) y *tag_enable* (definido en *escr_config* como el bit nº 4 del ESCR).
En <*count_setup*>, introducimos el nombre del evento encargado de realizar el mecanismo de etiquetado para los eventos de ejecución, que sólo puede ser *execution_event*. Como no modificamos ningún bit, se dejan los valores por defecto para este evento. Este evento se define así en *pentium4_emon.txt*:

```
<execution_event>
      <event_select>0x0c</event_select>
      <event_mask>
            <nonbogus0 bits="9"/>
            <nonbogus1 bits="10"/>
            <nonbogus2 bits="11"/>
            <nonbogus3 bits="12"/>
            <bogus0 bits="13"/>
            <bogus1 bits="14"/>
            <bogus2 bits="15"/>
            <bogus3 bits="16"/>
      </event_mask>
      <escr>cru_escr2</escr>
      <escr>cru_escr3</escr>
      <available_for_pebs/>
```

```
</execution_event>
```

10.5. Eventos etiquetados por replay (*replay tagging*)

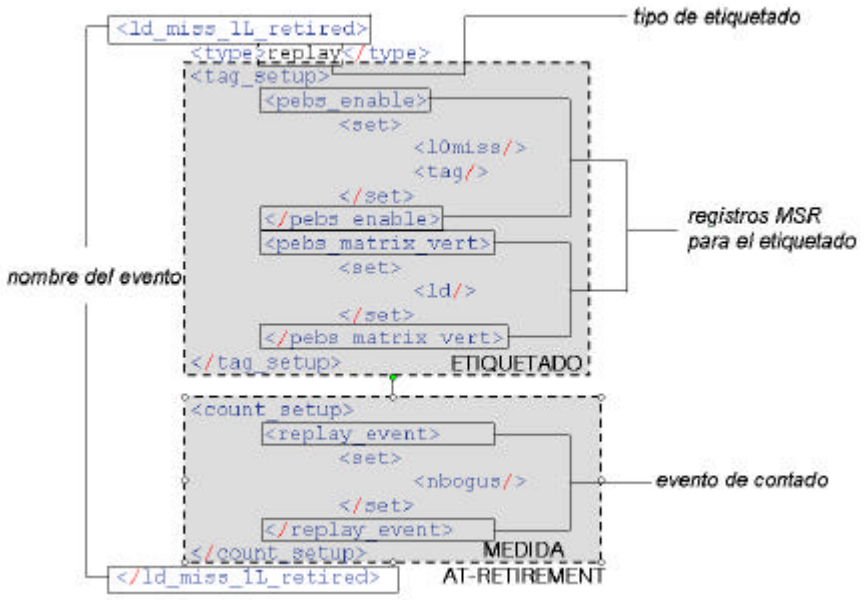

Figura 31. Ejemplo de definición de métrica con etiquetado por *replay*

Ahora vamos a definir un evento que utiliza el mecanismo de etiquetado en *replay*. La definición de estos eventos es un poco distinta a los dos casos anteriores de etiquetado (*front end* y *execution*).
Como queremos medir el número de fallos en la caché de primer nivel, vamos a llamar a nuestro evento *ld_miss_1L_retired*.
En la etiqueta `<type>` introducimos *replay*.
Todos los eventos que utilizan el etiquetado replay utilizan dos registros MSR especiales: *pebs_enable* y *pebs_matrix_vert*.
Para poder utilizar el mecanismo de etiquetado por replay, es necesario activar el bit *tag* del registro *pebs_enable* (es decir, el bit 24 del registro al que Intel® llama *ia32_pebs_enable*). También activamos el bit *l0miss* para medir concretamente los fallos de caché de primer nivel. En cuanto al *pebs_matrix_vert*, activamos el bit *ld* (bit 0), tal y como se indica en la documentación de Intel®.
Algunos eventos que utilizan el mecanismo de etiquetado en replay requieren configurar registros adicionales, pero éste no es el caso.
Finalmente, hacemos uso del evento *replay_event* para hacer el recuento de las instrucciones etiquetadas que llegan al final del pipeline. Se define así:

```
<replay_event>
      <event_select>0x09</event_select>
      <event_mask>
            <nbogus bits="9"/>
            <bogus bits="10"/>
      </event_mask>
      <escr>cru_escr2</escr>
      <escr>cru_escr3</escr>
      <available_for_pebs/>
</replay_event>
```

11. Configuración de EBS

Como es sabido, el procesador Intel® Pentium® 4 soporta los mecanismos IEBS (***I****mprecise* ***E****vent-****B****ased* ***S****ampling*) y PEBS (***P****recise* ***E****vent-****B****ased* ***S****ampling*). El objetivo de estos mecanismos de EBS es tomar muestras para saber qué es lo que está ejecutando el procesador y que contenidos tienen sus registros en el momento en que sucede un determinado evento. Mediante una correcta configuración del procesador para que genere una interrupción (para IEBS) o una *micro-assist* (para PEBS) a partir del desbordamiento de un contador, podemos obtener esta información de los registros. Tras un desbordamiento del contador podemos tomar una muestra, momento en el que volvemos a reinicializar el contador para que vuelva a desbordarse tras un número determinado de apariciones del evento determinado. Así, obtenemos varias muestras y podemos construir un perfil (*profile*) de rendimiento que nos informe de qué es lo que estaba ejecutando el procesador y qué contenían sus registros cuando sucedían los eventos. La idea es utilizar este perfil como guía para localizar las instrucciones o secuencias de instrucciones que producen un bajo rendimiento (debido a que provocan la aparición excesiva, no deseada, de un determinado evento).
La diferencia entre IEBS y PEBS radica en la precisión de las muestras tomadas. En definitiva, para IEBS la muestra tomada puede indicar una instrucción cercana a la instrucción que provocó realmente la aparición del evento, pero probablemente no sea la misma. Con PEBS, la muestra nos indicará con seguridad la instrucción que provocó la aparición del evento. Con esto, resulta evidente que las muestras obtenidas mediante PEBS son más útiles que las tomadas con IEBS. Sin embargo, mientras que todos los eventos del procesador Intel® Pentium® 4 soportan IEBS, solamente unos pocos soportan PEBS.

Brink proporciona una sencilla directiva para configurar los mecanismos EBS. Esta directiva es del tipo:

```
<ebs type="precise" interval="500000" buffer="500"
     sample="E" max="20000">
```

El significado de los atributos es el siguiente:

type puede indicar “imprecise” o “precise”, según estemos utilizando IEBS o PEBS respectivamente.

interval indica el número de veces que tiene que darse el evento determinado para que se tome una muestra. En general, a menor valor obtendremos un perfil de rendimiento más preciso, pero también podemos sobrecargar el programa y degradar su rendimiento (falseando las medidas).

buffer indica el número de muestras que se almacenarán en un buffer antes copiar los datos de las muestras al espacio de usuario. Por esto, un buffer mayor (que almacena muchas muestras) disminuirá la sobrecarga de obtener las muestras. Sin embargo, debido a que *abyss*

no detiene la ejecución del programa en el momento de pasar los datos de las muestras desde el espacio del kernel al espacio de usuario, el traspaso de la información de un buffer grande puede falsear las medidas provocando una importante aparición de eventos.

sample indica qué tipo de muestras se van a tomar. Este atributo puede valer “A” (ALL) o “E” (muestra EIP). En el primer caso en cada muestra almacenamos el valor de todos los registros.

max indica el número máximo de muestras que vamos a tomar. En caso de sobrepasarlo, abyss dejará de tomar muestras y finalizará con error.

La directiva <ebs> debe insertarse en el interior de la definición de un evento contenido en un experimento (en el fichero de experimentos).

A la hora de usar PEBS, es importante saber que sólo puede usarse con tres eventos:

```
front_end_event
execution_event
replay_event
```

Aunque parezcan muy pocos eventos, no son tan pocos si tenemos en cuenta que estos eventos sirven para medir una variedad de eventos usando mecanismos de etiquetado.
Además, si se usa PEBS en un experimento solamente uno de los tres mecanismos de etiquetado puede ser utilizado (esto es debido al soporte hardware).

Como hemos comentado anteriormente, al utilizarse un mecanismo EBS, los resultados de las muestras se escriben en un fichero de salida con el sufijo *ebs_samples.txt*.

12. Ejemplo de definición de un fichero de experimentos

Vamos a definir, a modo de ejemplo, un fichero de experimentos. El fichero constará de dos experimentos, por lo que se realizarán dos ejecuciones de cada programa.
Mediremos una serie de eventos en tres programas: el sencillo comando *ls* de UNIX y dos hipotéticos archivos binarios.

```
<?xml version='1.0'?>
<!--
        Esto es un fichero de experimentos de ejemplo. Contiene tres programas a ejecutar y
        dos experimentos. Por tanto, se generarán seis jobs (y seis directorios de resultados).
-->
<exp_config>
        <!-- Aquí indicamos, siempre, que utilizamos un Pentium 4 -->
        <processor type="pentium4"/>

        <!-- Tres programas a ejecutar, con su nombre y el ejecutable en cuestión -->
        <programs>
                <listado_dirs command="./bin/ls"/>
                <program_parallel command="./home/david/wavelet_two_threads"/>
                <program_SSE2 command="./home/david/SIMD_SSE2"/>
        </programs>

        <!-- Lista de experimentos, en este caso dos, a considerar (para cada programa) -->
        <experiments>

                <!-- Estos parámetros se aplican, por defecto, a todos los eventos -->
                <default>
                        <!-- Contamos sólo eventos de usuario (no de sistema operativo) -->
                        <set>
                           <usr/>
                        </set>
                </default>

                <!-- Primer experimento -->
                <exp1>
                        <uops_retiradas_usuario base="uops_retired">
                           <set>
                                <nbogusntag/>
                           </set>
                        </uops_retiradas_usuario>
                        <instr_x87_retiradas_usuario base="x87_FP_retired">
                           <set>
                                <nbogus/>
                           </set>
                        </instr_x87_retiradas_usuario >
                </exp1>

                <!-- Segundo experimento -->
                <exp2>
                        <uops_retiradas base="uops_retired">
                           <set>
                                <nbogusntag/>
                           </set>
                        </uops_retiradas>
```

```
                        <!-- Este es un evento que utiliza el mecanismo de etiquetado en el
                        front-end. Utiliza PEBS para obtener un perfil de rendimiento.
                        La configuración para PEBS se hace de forma que se produzcan
                        500.000 eventos entre cada toma de muestra. El buffer almacena 500
                        muestras antes de escribirlas en memoria de usuario.
                        Se toman muestras EIP y se restringe el máximo número de muestras
                        tomadas a 20.000 -->

                        <operaciones_load_retiradas_usuario_ebs base="loads_retired">
                                <ebs type="precise" interval="500000" buffer="500" sample="E"
                                    max="20000"/>
                        </operaciones_load_retiradas_usuario_ebs>

                        <!-- Este evento se usa para contar el número de operaciones de load
                        retiradas, usando el evento front_end_event. El etiquetado de las
                        operaciones de load se realiza mediante el evento de arriba -->
                        <operaciones_load_retiradas_usuario base="front_end_event">
                           <set>
                                <nbogus/>
                           </set>
                        </operaciones_load_retiradas_usuario>
                </exp1>
        </experiments>
</exp_config>
```

Documentación del software del proyecto

Contenido

1. La librería p4pm.a

Para el desarrollo de la aplicación, nos hemos basado en las librerías proporcionadas por *Perfctr* para implementar la primera parte del proyecto. *Perfctr* nos proporciona las librerías necesarias para interactuar con los contadores del procesador. Nos es útil el modo virtual que utiliza *Perfctr* para realizar las métricas de los eventos. Sin embargo, no nos proporciona las funciones necesarias para utilizarlo en los programas que queramos medir. Por lo tanto hemos generado nuestra propia librería, *p4pm.a*, que sea capaz de hacer esto.

Utilizamos una estructura propia, *tsc_cont*, para devolver el valor de los contadores y el TSC, la cual está declarada de la siguiente manera:

```
typedef struct
{
  unsigned long long tsc;
  unsigned long long contadores[18];
  int num_contadores;
  char nombres_eventos[18][50];
} tsc_cont;
```

Los campos de la estructura son los siguientes:

- *tsc*: time-stamp counter, que son el número de ciclos invertidos por el procesador.
- *contadores*: Es un array con el resultado de los contadores. Sólo son válidas las posiciones desde la *0* a *num_contadores-1*.
- *num_contadores*: número de contadores utilizados en la medida.
- *nombre_eventos*: Tiene guardado el nombre del evento correspondiente a cada contador.

Hay una constante definida en el fichero *p4pm.h* que se llama *MAX_MED* y está inicializada a un valor de 200. Esta constante indica el número máximo de medidas que se podrán hacer en el programa que estemos midiendo. Este valor lo podemos modificar si van a hacerse un número mayor de medidas. Podemos también utilizar la función *initialize_med* para pasarle por parámetro el número máximo de medidas. Si no pasamos explícitamente este valor, se tomará el valor por defecto indicado por MAX_MED.
Es necesario indicar el número máximo de medidas, ya que éstas se guardan en una zona de memoria estática que se reserva al inicializar los contadores. Se ha tomado la decisión de guardar las medidas de forma estática para que no influya en el resultado de las medidas. Reservar memoria mientras se ejecuta el programa que está siendo monitorizado puede suponer una carga (*overhead*) que distorsione los valores obtenidos en las métricas.
Cada medida completa (con todos los contadores y el TSC) ocupa *1056 bytes*, por lo tanto queda para el programador ver cómo puede influir en sus medidas si pone un valor muy alto en el número máximo de medidas.

Las funciones que proporciona nuestro programa, cuyas declaraciones se encuentran en el fichero de cabeceras *p4pm.h*, son las siguientes:

- *void initialize(char* ficheroConfiguracion)*
  Este procedimiento lee el fichero de configuración, que se le pasa como parámetro, e inicializa el valor de los CCCR y ESCR con los valores correspondientes. Inicializa el array de medidas con el tamaño MAX_MED (Constante definida en *p4pm.h*). Por lo tanto, aquí ya se ha indicado que eventos se van a medir.

- *void initialize_med(char* ficheroConfiguracion, int max_medidas)*
  Igual que *initialize* pero además le pasamos el número máximo de medidas que podemos medir con el parámetro *max_medidas* (en este caso, se ignora el valor de la constante MAX_MED).

- *void begin_count()*
  Guarda el estado actual de los contadores y TSC en la pila. Se llama a este procedimiento cada vez que se quiere empezar una nueva medida. No importa que los contadores tengan valores mayores que cero, lo que haremos será restar estos valores a los valores de los contadores obtenidos al final del trozo de código que se mide, obteniendo la diferencia que expresa el número de incrementos de los contadores entre el inicio y el fin de la medida.

- *tsc_cont *end_count_file(char* ficheroResultados)*
  Lee los valores de los contadores y calcula las medidas obtenidas relativas al último *begin_count* (haciendo la resta). Los resultados se guardarán en memoria. Además, devuelve una variable que contiene la estructura con los resultados de los contadores y el TSC. Se le pasa por parámetro el nombre del fichero donde queremos que guarde los resultados, pero no en este momento para no sobrecargar el programa con escrituras a disco. Esta escritura a disco se realizará cuando llamemos a la función *write_files()*.

- *tsc_cont *end_count()*
  Lo mismo que la función anterior, pero esta vez no se escribe a fichero sino que tan sólo se devuelve una variable con el resultado de las medidas relativas al último *begin_count*. Por lo tanto, cuando se invoque *write_files(),* no se realizará ninguna acción para esta medida.

- *void write_files()*
  Escribe en los ficheros correspondientes los resultados de las medias realizadas a lo largo del programa. Esta función debe llamarse al final de todo, después de haber realizado todas las medidas deseadas. No es necesario invocarla si no queremos obtener ficheros de salida con los resultados.

Veamos la utilización de estas funciones en un programa de ejemplo, *EjemploFact.c.* Al programa se le pasa por parámetro el fichero de experimentos. Este programa calcula, ITERACIONES veces, el factorial de NUMFACT.

```
#include "p4pm.h"

unsigned fac(unsigned n)
{
      return (n < 2) ? 1 : n * fac(n-1);
}

void do_fac(unsigned n)
{
      printf("\nfac(%u) == %u\n", n, fac(n));
}


int main(int argc, char *argv[])
{
      const int NUMFACT = 20000;
      const int ITERACIONES = 99;
      int i;
      tsc_cont *results;
      char  fichero[200];

      /* Inicio del valor de los contadores con el fichero pasado
      como parámetro del ejecutable */
      initialize_med(argv[1], 300);

      /* Se guarda el valor de los contadores */
      begin_count();
      for (i=0; i<ITERACIONES; i++)
      {
            begin_count();
            fac(NUMFACT);
            sprintf( fichero,
                     "resultados/resultado1Factorial%d.txt", i);
            end_count_file(fichero);
      }

      /* Guardamos en results el avance de los contadores desde
      el begin_count() */
      results = end_count_file("resultados/resultadoBucle.txt");

      /* Imprimimos por terminal el resultado de los contadores,
      haciendo uso de la variable devuelta */
      printf("TSC= %lld\n", results->tsc);
      for (i=0; i<results->num_contadores; i++)
      {
            printf ("%s = %lld\n", results->nombres_eventos[i],
                    results->contadores[i]);
      }

      /* Se guardan el resultado de los contadores en los
      ficheros que se han pasado a los end_count */
      write_files();
      return 0;
}
```

La línea para compilar este programa podría ser:

```
$gcc EjemploFact.c p4pm.a -o EjemploFact
```

Y para ejecutarlo:

```
$/camino_a_ejecutable/EjemploFact
/camino_a_ficheros_experimentos/predictor_saltos.txt
```

Los ficheros de experimentos que utiliza nuestra librería no son ficheros como los ficheros xml de *brink & abyss*. Estos ficheros son *más engorrosos*, y constan de ristras de bits que serán las que se escriban en los MSRs (ESCR, CCCR y PMC) para cada evento que se desea medir.
Es necesario que los eventos sean compatibles entre sí, esto es, que puedan medirse en una sola ejecución sin conflictos de recursos. Esta comprobación no la realiza la librería *p4pm*, que supone que estos ficheros son correctos y están libres de este tipo de errores.
Proporcionamos algunos ficheros de experimentos de este tipo junto con el software.

2. La herramienta brinkMod

Para evitar tener que tratar con estos ficheros y que el programador tenga total libertad sin tener que pelearse con la documentación de Intel® para ver uno a uno los bits necesarios para configurar los eventos que se desean medir, proporcionamos una aplicación que genera estos ficheros de manera automática a partir de ficheros de experimentos mucho más amenos y tratables.

Para generar los ficheros de experimentos de una forma más cómoda utilizamos una versión modificada de la aplicación *brink & abyss*. Aunque más concretamente, partimos de *brink* porque *abyss* no lo utilizamos para nada (esta parte ya la sustituye nuestra librería *p4pm*). Además, al partir de esta herramienta también aprovechamos la comprobación de conflictos entre eventos no compatibles. La mayor ventaja tal vez sea el no necesitar saber nada, o casi nada, de la configuración de los ESCR, CCCR y PMCs y el significado de cada bit.
Simplemente escribiremos un fichero en XML y nuestro programa derivado de *brink* generará varios ficheros de experimentos que acepta *p4pm*. La estructura de este fichero XML es exactamente la misma que la utilizada por *brink*, por lo que la documentación acerca de los ficheros de experimentos de *brink & abyss* es muy útil también para nuestra aplicación.
Hemos llamado a nuestro *brink* modificado, *brinkMod.* Al igual que *brink*, *brinkMod* está escrito en Perl. Básicamente, su diferencia con *brink* radica en la salida ya que lo que hace es generar nuestros ficheros de experimentos y no ceder el control a *abyss*. Por esto, *brinkMod* no realiza ninguna medida, tan sólo configura los registros MSRs para realizar las medidas y comprueba la existencia de conflictos de recursos.
Otra ventaja es que nos valen para *brinkMod* los ficheros de experimentos incluidos en la distribución de *brink & abyss*.
Proporcionamos algunos ficheros de ejemplo que se pueden utilizar bien para conseguir ficheros de experimentos, o simplemente para ver el formato utilizado. Al igual que *brink*, *brinkMod* requiere un fichero de configuración donde estén definidos los eventos que se pueden medir y las estructuras de los registros MSR. Los ficheros de experimentos que genera *brinkMod* se guardan en un directorio que se indica mediante el parámetro *–outdir*.

Un ejemplo de línea de ejecución sería:

```
$/ruta_a_ejecutable/brinkMod
      -exp fichero_de_experimentos
      -config fichero_de_configuración
      -outdir directorio_de_ficheros_experimentos
```

Con las dos herramientas explicadas hasta ahora conseguimos una gran funcionalidad. Con la librería *p4pm.a* podemos obtener medidas bastante ajustadas y tener libertad de medir sólo parte de nuestro código, como pudieran ser bucles o funciones aisladas. Por otro lado, tenemos la flexibilidad y sencillez de uso para generar ficheros de experimentos de la aplicación *brinkMod*.

Hay que notar que la librería *p4pm.a* por sí sola puede realizar métricas de eventos. Sin embargo, *brinkMod* es un programa auxiliar para facilitar el trabajo, pero no realiza ningún tipo de métrica por sí solo.

3. La herramienta CountIt

Después de esto, hemos dado un paso más para centralizar en un solo programa la métrica de eventos. Se trata del programa *CountIt,* que es un sencillo script en Perl. Este programa automatiza el proceso, de forma que permite realizar varias ejecuciones del programa para contrastar resultados o para realizar una gran cantidad de medidas de eventos distintos.
La línea de ejecución de este programa es la siguiente:

```
./CountIt
      -expfile fichero_de_experimentos
      [-iters iteraciones]
      [-configfile fichero_de_configuración]
      ejecutable
```

El fichero de experimentos y el de configuración son los mismos ficheros XML que pasaríamos a *brink & abyss*. Ciertamente, *CountIt* utiliza *brinkMod* para traducir el fichero XML a los ficheros de experimentos para p4pm.a. Después, *CountIt* ejecuta el binario tantas veces como se haya indicado en *iteraciones* (por defecto una vez). Así, podemos ajustar el número de veces que queremos que se ejecute nuestro programa con el parámetro *iteraciones*. El ejecutable es el programa del usuario que éste ha compilado previamente enlazando con la librería *p4pm.a*.

La idea inicial era que *CountIt* recibiera el código fuente del programa que se va a monitorizar pero la compilación entonces la haría nuestra herramienta, lo cual no suele ser muy indicado debido a que hay muchas formas de compilar los programas, con muchas opciones de compilación que debería decidir el usuario y no nuestra herramienta. Se podría haber pensado en algún mecanismo para que la compilación, a pesar de realizarla *CountIt*, fuera personalizada de alguna manera por el usuario de la aplicación.
Pero al final hemos optado por que *CountIt* reciba el binario del código a analizar, con la compilación ya realizada a medida del propio usuario.
Las condiciones que debe cumplir el usuario que vaya a utilizar *CountIt* son compilar el código enlazando con la librería *p4pm.a,* y tener en cuenta las siguientes consideraciones:

- El programa del usuario va a recibir dos parámetros extra, además de los que de por sí ya reciba. Estos dos parámetros serán los dos últimos parámetros que reciba el programa (es decir, *argv[argc-2]* y *argv[argc-1]*). El primero va a ser el nombre del fichero de experimentos de *p4pm.a* que se vaya a utilizar en el código del usuario. De esta forma, el usuario debe pensar que en *argv[argc-2]* hay un valor de tipo *char** con el nombre del fichero de experimentos.

- Dado que *CountIt* puede generar múltiples ficheros de experimentos a partir del fichero XML que recibe, es importante que el programa de usuario tenga alguna manera de saber cuál de estos ficheros generados por *CountIt* (por *brinkMod*) se está usando en cada ejecución.

Así, el argumento *argv[argc-1]* contiene siempre el nombre del fichero de experimentos (sin extensión) que se está usando en la actual ejecución. Así, el programador puede utilizar esta variable para distinguir entre unos ficheros de experimentos u otros.
Esto puede ser útil para dar nombre a los ficheros donde vamos a guardar los resultados de las medidas realizadas.