Indice

1 Introducción ........................................................................................................... 5
  1.1 Prólogo ........................................................................................................... 5
  1.2 Motivación .................................................................................................... 6
  1.3 Objetivos generales ...................................................................................... 7

2 Características de nuestra máquina ................................................................. 11
  2.1 Hardware ..................................................................................................... 11
    2.1.1 Procesador ............................................................................................ 11
    2.1.2 Jerarquía de memoria ............................................................................. 12
        L2 cache ..................................................................................................... 13
        Memoria principal o RAM ........................................................................ 13
        TLB ............................................................................................................ 14
  2.2 Entorno de trabajo ....................................................................................... 15
    2.2.1 Sistema Operativo ................................................................................ 15
    2.2.2 Compilador .......................................................................................... 15
        Opciones del compilador .......................................................................... 15
    2.2.3 Oprofile 0.9.1 ..................................................................................... 16
        Opcontrol .................................................................................................. 17
        Opreport .................................................................................................. 18
        Opannotate .............................................................................................. 19
    2.2.4 Gprof .................................................................................................... 19
    2.2.5 Time ...................................................................................................... 20
    2.2.6 Pin ......................................................................................................... 20

3 El Intel del Pentium 4 ..................................................................................... 23
  3.1 Resumen del Procesador .............................................................................. 23
  3.2 Visión General del procesador .................................................................... 25
  3.3 Pipeline del Pentium 4 ................................................................................. 27
    3.3.1 Fase de fetch ....................................................................................... 28
        In-order front-end, Prefetching, ITLB y BTB ........................................... 28
        Decodificador y ROM de microinstrucciones ........................................ 29
        Trace cache ............................................................................................. 30
    3.3.2 Lógica de ejecución Out-of-Order ......................................................... 30
        Allocator y el Register Renaming ............................................................ 31
        Planificación de microinstrucciones ......................................................... 32
    3.3.3 Instrucciones vectoriales ..................................................................... 32
    3.3.4 Latencia y Throughput .......................................................................... 34
        Latencia y throughput de las principales instrucciones .......................... 35
    3.3.5 Unidades de ejecución de enteros y de coma flotante ......................... 35
        Puertos de envío (issue ports) .................................................................. 35
        Unidad de ejecución de enteros .............................................................. 37
        Cache de datos de primer nivel .............................................................. 38
        Store-to-load forwarding ....................................................................... 38
        Unidad de ejecución de floats y SIMD ................................................... 39
4 Aplicaciones Bioinformáticas ................................................................. 41

4.1 Conocimientos previos ............................................................................. 41
  4.1.1 Introducción a la Bioinformática .......................................................... 41
  4.1.2 Qué es una proteína? ............................................................................ 42
  4.1.3 ¿Para qué nos sirve comparar proteínas? ................................................. 46

4.2 Conceptos básicos en bioinformática ......................................................... 47
  4.2.1 Protein structural alignment ................................................................. 47
  4.2.2 Protein folding ..................................................................................... 48
  4.2.3 Protein docking ................................................................................... 49

4.3 Aplicación Combinatorial Extensión (CE) .................................................. 52
  4.3.1 Introducción ......................................................................................... 52
  4.3.2 Características del CE ................................................................. 53
     Glosario del CE ..................................................................................... 53
     Ejecución del CE .................................................................................. 55
     Salida del CE ....................................................................................... 56
     Principales funciones del CE ............................................................... 57
     ➢ Función scratch_align_ent() ................................................................. 57
     ➢ Función ce_1() .................................................................................... 58
     ➢ Función dpAlign() .............................................................................. 59

4.4 Aplicación FTDock ..................................................................................... 61
  4.4.1 Introducción ....................................................................................... 61
  4.4.2 Características del FTDock ............................................................... 63
     Transformadas de Fourier ................................................................... 63
     FFTW ..................................................................................................... 64
     Glosario del FTDock ......................................................................... 67
     Ejecución de FTDock ........................................................................... 68
     Salida de FTDock ............................................................................... 69
     Estructura y descripción del algoritmo de FTDock. .............................. 70

5 Análisis de las aplicaciones ........................................................................... 75
  5.1 Metodología empleada en general ......................................................... 75
  5.1.1 Caracterización de la carga de trabajo ................................................ 75
  5.1.2 Metodología de optimización ............................................................. 75
  5.1.3 Metodología con el PIN ................................................................. 76

5.2 Glosario de optimizaciones ..................................................................... 78

5.3 Análisis de la aplicación CE ................................................................. 82
  5.3.1 Caracterización de la carga de trabajo de CE ...................................... 82

5.4 Optimización de CE ................................................................................. 84
  5.4.1 Especialización de la función dpAlign ............................................... 84
  5.4.2 Mejora de la explotación de la localidad de datos en dpAlign .......... 87
     Análisis de la mejora propuesta .......................................................... 93
  5.4.3 Especialización de ce_1 ..................................................................... 95
  5.4.4 Optimización de ce_1 ....................................................................... 98
  5.4.5 Comparativa de CE con reales de doble y simple precisión ............. 105

5.5 Instrucciones y accesos a Memoria del CE .............................................. 109

5.6 Análisis del FTDock ................................................................................. 114
  5.6.1 Caracterización de la carga de trabajo de FTDock ............................ 114

5.7 Optimización de FTDock .......................................................................... 118
  5.7.1 Comparativa del FTDock usando con FFTW 2.1.3 y FFTW 3.1.2 .... 121
  5.7.2 Comparativa de FTDock usando las librerías FFTW con floats ...... 125
5.7.3 Comparativa de FTDock usando FFTW3 con SIMD y *doubles* ........................................... 133

5.8 Análisis del tipo de Instrucciones y accesos a Memoria ............................................. 136
  5.8.1 *Grid_size* por defecto con funciones genéricas ...................................................... 137
      Instrucciones ........................................................................................................... 137
      Memoria ................................................................................................................. 138
  5.8.2 *Grid_size* por defecto con codelets ......................................................................... 140
      Instrucciones ........................................................................................................... 140
      Memoria ................................................................................................................. 141
  5.8.3 *Grid_size* de 128 ................................................................................................. 143
      Instrucciones ........................................................................................................... 143
      Memoria ................................................................................................................. 145

**Bibliografía** .................................................................................................................. 147

**Apéndice** ....................................................................................................................... 151

**CE** ................................................................................................................................ 151
  Función *dpAlign Original* .............................................................................................. 151
  Función *dpAlign Optimizada* ....................................................................................... 155

**Funciones genéricas de FTDock** .................................................................................... 158
  *fftw_hc2hc_backward_generic (versión original)* ......................................................... 158
  *fftw_hc2hc_backward_generic (versión vectorial)* ......................................................... 159

**Codelets en FTDOCK** .................................................................................................... 162
  Codelet de 2 ................................................................................................................... 162
  Codelet de 2 con SIMD .................................................................................................. 163
  Codelet de 8 ................................................................................................................... 163
  Codelet de 8 con SIMD .................................................................................................. 165
1 Introducción

1.1 Prólogo

Durante las últimas décadas, los avances en investigación biológica han revolucionado nuestra comprensión sobre las bases de la vida. Conforme la genética avanza, disponer de métodos computacionales con los que extraer, ver y analizar genomas, resulta esencial [12]. De ahí, que el reciente crecimiento exponencial en la cantidad de información sobre el genoma ha disparado el interés en el análisis sobre información genética, y de ese interés han aparecido nuevas (y numerosas) aplicaciones. La bioinformática permite a los investigadores identificar la información relevante de los masivos datos biológicos disponibles. A su vez, este campo se está volviendo más y más importante debido al gran interés que han mostrado la industria farmacéutica y las compañías biotecnológicas. Recientes estudios de mercado estiman que el volumen de negocio del mercado farmacológico crecerá a $243000 millones en 2010.

Para la realización de este proyecto hemos decidido analizar dos aplicaciones bioinformáticas. Concretamente, vamos a estudiar el Combinatorial Extension (CE) y Fourier Transform Docking (FTDock), que son algoritmos típicos de protein structural alignment y protein docking, respectivamente.

Protein structural alignment es una forma de alineamiento de secuencias basado en la comparación de la estructura. Este alineamiento, intenta establecer equivalencias entre dos o más estructuras poliméricas (ya veremos más adelante qué son), basándose en la forma tridimensional de las proteínas. Estas equivalencias estructurales suponen un origen biológico idéntico, y las diferencias en el comportamiento de estas, responde a la evolución de alguna de ellas.

La importancia de esta aplicación, radica en el hecho de que la gran mayoría de proteínas que la ciencia conoce tienen un comportamiento biológico (caracterizado por su interactividad con otras proteínas) incomprensible e impredecible. Incluso aparecen casos de proteínas que, a pesar de tener un proceso biológico determinado, interactúan con proteínas, cuyos procesos no tienen nada que ver. Gracias al alineamiento de secuencias se puede predecir el comportamiento biológico (desconocido hasta el momento) de una proteína.
Protein docking explora el modo en que interactúan dos moléculas, dependiendo de sus características topográficas y de su estructura energética, e intenta predecir su unión teniendo en cuenta las conformaciones que favorecen la interacción.

Esta técnica sirve para determinar la estructura molecular de posibles nuevos fármacos o drogas,... Formados mediante la unión de dos o más proteínas, sin la necesidad de realizar pruebas experimentales.

1.2 Motivación

Durante los primeros años de carrera, exceptuando algunas agradables sorpresas, la mayoría de las asignaturas del departamento de AC las fui sacando con más problemas que la mayoría (quizá por no poner el mismo interés que en otras, o tal vez por no dedicar el tiempo que requerían). Pero la cosa cambió radicalmente en mi cuarto año.

Tras matricularme en la asignatura Arquitectura de Computadores, de la que (para ser honestos) me habían hablado fatal (“es muy difícil”, “no la expician bien”, “los profes van muy sobrados, incluso te humillan”, …); decidí cambiar mi enfoque por una vez y centré la mayor parte de mis esfuerzos de ese cuatrimestre en dicha asignatura, lo que provocó no solo unos buenos resultados en cuanto a comprensión de la materia, sino que además me quedé un poco decepcionado por no haber podido aprovechar de la misma forma otras asignaturas.

Con lo que, para el que debía ser mi último año matriculando asignaturas, me obligué a elegir una asignatura de la rama de AC. Buscando con algunos compañeros, encontramos una, que bautizamos (con poco acierto) como “continuación de AC pero sin examen”. Su nombre era PCA (“programación consciente de la arquitectura”), y resultó ser (desde mi modesto punto de vista) una de las asignaturas más útiles de la carrera. Pues no solo me sirvió para aprender diversas técnicas para optimizar códigos, sino que además pudimos ver “experimentalmente” el funcionamiento y comportamiento de algunos componentes de la arquitectura, como la cache, el procesador segmentado, el alineamiento en memoria,...
Al final de dicho cuatrimestre ya tenía medio apalabrado mi PFC con un profesor del departamento de estadística, cuando Daniel Jiménez me habló por primera vez de unos algoritmos de docking que analizaban proteínas en tres dimensiones, y me comentó que estaba buscando gente para hacer PFCs sobre este tema.

A partir de ese momento comencé a documentarme sobre unos temas completamente nuevos para mí, las aplicaciones bioinformáticas.

Daniel me comentó que el objetivo inicial era analizar alguna de estas aplicaciones para poder caracterizar el comportamiento de estas en un procesador vectorial (G5). Esto nos permitiría saber qué aspectos del procesador ayudan a las aplicaciones a ejecutarse más rápido, y quizás, averiguar también qué aspectos arquitectónicos del procesador se podrían modificar.

Aunque lo que más me motivó para realizar un proyecto de este tipo, fue el poder profundizar un poco más en este campo, del que había pasado de puntillas por la carrera, y que finalmente me había llamado bastante la atención. Por otro lado, la mayoría de ofertas existentes en el momento de la elección del proyecto era de temas relacionados con los sistemas de información, sobre los cuales se realizan diversas prácticas a lo largo de la carrera y en los que no estaba del todo interesado.

Ahora pasaremos a comentar los objetivos del proyecto, que por diversos motivos han variado significativamente.

1.3 Objetivos generales

Debemos indicar que el objetivo inicial del proyecto era realizar sólo el análisis y optimización de la aplicación Combinatorial Extension (CE) en un procesador G5. Sin embargo el análisis y optimización de la aplicación CE no supuso el volumen de trabajo esperado; por consiguiente decidimos ampliar el análisis con otra aplicación (si bien, parecida en la temática) diferente.
Debido a problemas de disponibilidad del hardware, hemos decidido realizar los análisis y optimizaciones de las aplicaciones sobre un procesador Pentium 4, que también es un procesador vectorial.

Hemos de indicar, antes de explicar los objetivos generales, que dado el completo desconocimiento sobre proteínas, bioinformática, docking, folding, e incluso, transformadas de Fourier; ha habido una considerable carga de trabajo en lo que respecta a documentación sobre cada uno de estos temas. Aunque, ni mucho menos podemos considerarnos unos expertos conocedores de las proteínas y su funcionamiento, por el mero hecho de haber leído algunos documentos sobre cada tema, al menos hemos conseguido comprender de una forma básica todo lo que nos proponemos explicar (de una forma sencilla y lo más comprensible posible) en esta memoria.

La memoria está estructurada en dos partes principales (sin contar con introducción, bibliografía, conclusiones, etc.) que serían la parte informática y la parte bioinformática. En la primera, explicamos todo lo referente sobre software y hardware utilizado, incluidos un análisis y una descripción detalladas del Pentium 4.

Ya en la segunda parte, mucho más extensa, explicamos una introducción a la bioinformática y las principales características de las dos aplicaciones analizadas, para finalmente dar paso al análisis y optimización de las dos aplicaciones.

La parte donde analizamos y optimizamos las aplicaciones, la hemos dividido en tres. En la primera intentamos describir la metodología general aplicada (todos aquellos pasos comunes en el análisis y optimización para las dos aplicaciones) que es bastante breve, debido a que las optimizaciones realizadas entre las dos aplicaciones son bastante diferentes.

Las otras dos partes constan de los análisis y optimizaciones de cada una de las aplicaciones. Donde hemos diferenciado entre el análisis inicial (caracterización de la carga de trabajo) y las optimizaciones aplicadas.
Dicho esto y tras los cambios descritos anteriormente, podemos concluir que los objetivos finales de este proyecto son:

1. Descripción de las herramientas utilizadas para realizar los estudios, ubicadas en el segundo capítulo, dentro del entorno del trabajo; así como una extensa descripción de la arquitectura del procesador vectorial con el que trabajamos (Pentium 4) a lo largo de todo el tercer capítulo

2. Breves introducciones sobre (todas ellas en el capítulo 4):
   a. Bioinformática
   b. Tipos de algoritmos de las aplicaciones tratadas
   c. Aplicaciones utilizadas,
   d. Características y principales funciones

3. Caracterización de la carga de trabajo de las aplicaciones bioinformáticas:
   a. Combinatorial Extension
   b. FTDock

4. Desarrollo y análisis de:
   a. Optimizaciones de los códigos de las aplicaciones, siempre que sea posible, sin utilizar instrucciones vectoriales
   b. Modificación de las aplicaciones, siempre que sea posible, con el fin de utilizar las instrucciones vectoriales del Pentium 4

5. Comparación de los rendimientos obtenidos entre los diferentes códigos generados, así como un análisis de las instrucciones y de los accesos a memoria de estas. Para:
   a. Código original
   b. Códigos con cambios significativos respecto al original, o respecto a una optimización previa, ya analizada
   c. Códigos con decimales de simple precisión
   d. Códigos con instrucciones vectoriales

Los puntos 3, 4 y 5, aparecen todos ellos a lo largo del capítulo 5 (análisis de las aplicaciones), donde se caracterizan y optimizan las dos aplicaciones analizadas.
2 Características de nuestra máquina

2.1 Hardware

2.1.1 Procesador

Durante la realización de este proyecto, hemos trabajado con un procesador Pentium 4. Desde que fue lanzado en noviembre del 2000, diferentes tipos de chip, de este mismo procesador, han salido al mercado.

Concretamente, nosotros, trabajamos con un procesador Intel Pentium a 2,6GHz (con chip Northwood). Este chip fue lanzado al mercado por Intel en enero de 2002 a una velocidad de 2,0 y 2,2 GHz. Esta nueva versión combinaba un incremento de 256 a 512 KB en la memoria cache de segundo nivel mediante la reducción a la tecnología de 130 nanómetros. Al estar el microprocesador compuesto por transistores más pequeños, podía alcanzar mayores velocidades y a la vez consumir menos energía. El nuevo procesador utilizaba el Socket 478, que ya se había visto en los últimos modelos de la serie Willamette.

Posteriormente, un Pentium 4 de 2,4 GHz fue introducido en abril de 2002, uno de 2,53 GHz en mayo (que incluyó un aumento del FSB de 400 a 533 MHz). En agosto vieron la luz los modelos de 2,6 y 2,8 GHz, y en noviembre la versión de 3,06 GHz.

Para conocer las propiedades concretas de nuestro procesador, mediante una captura al programa CPU-Z, obtenemos las características que aparecen en la figura 2.2. Las principales características serían la velocidad (de 2,6GHz como ya habíamos comentado), los tamaños de las memorias caches y finalmente el set de instrucciones MMX, SSE y SSE2.
2.1.2 Jerarquía de memoria

Los principales elementos que forman la jerarquía de memoria son:

- Memoria *trace cache* (Ver apartado 3.3.1)
- L1 *cache* de datos (Ver apartado 3.3.5)
- L2 *cache*
- Memoria principal o RAM
- TLB

**Fig. 2.2 Características del procesador de la máquina utilizada**

**Fig. 2.3 Características de las memorias cache y RAM de nuestra máquina**
**L2 cache**

Se trata de una memoria *cache* que reside en el chip. A diferencia de la *trace cache* o la L1 cache, cuyos tamaños no superaban los 12KBytes u 8KBytes, respectivamente; nos encontramos ante una memoria de tamaño de 512KBytes, que permite guardar mucha de la información que no cabe en las memorias anteriormente comentadas, y evita que se tenga que acceder a la memoria principal (accesos de mayor latencia).

Se trata de una memoria con 8-way associative, con un tamaño de línea de 128Bytes (cada vez que se produce un fallo en la L2 cache, se realizan dos accesos de 64Bytes en la memoria principal).

Cada acceso de lectura en esta memoria supone 7 ciclos de reloj del procesador; pudiendo empezar una nueva instrucción cada 2. Dado este *throughput* y el tamaño de acceso de 64Bytes, resulta un ancho de banda (con nuestro procesador a 2.6GHz) superior a 80GB/s.

**Memoria principal o RAM**

La memoria RAM es una memoria volátil (o lo que es lo mismo, sus datos desaparecen si se desconecta la energía eléctrica) en la que se pueden realizar tanto lecturas como escrituras.

En las máquinas actuales, se utiliza normalmente como memoria temporal para almacenar la información intermedia que no cabe en las memorias cache (de tamaño considerablemente inferior).

Nuestra máquina, dispone de dos módulos de 256MBytes; lo que supone 512MBytes de memoria RAM.

![Fig. 2.4 Módulo de memoria RAM (concretamente de 128MBytes)](image-url)
TLB

El *transational lookaside buffer* (TLB) es un buffer que se encarga de la traducción de las direcciones lógicas a físicas. Este buffer es el encargado de mejorar el rendimiento cuando se trata de acceder a la tabla de páginas.

El TLB puede residir tanto entre la CPU y la memoria *cache*, como entre la memoria *cache* y la RAM.

Si las direcciones de la memoria *cache* son lógicas, las peticiones que realiza la CPU se envían directamente a la memoria *cache*, que a continuación accederá al TLB. En cambio si la memoria *cache* utiliza direcciones físicas; es la CPU la que accede al TLB en cada operación de memoria, y la dirección física resultante es enviada a la *cache*.

Cuando se produce un fallo en el TLB, se produce un descenso en el rendimiento del sistema, pues se debe realizar un acceso a memoria para actualizar el TLB. Este puede cargarse previamente con la dirección de la siguiente página deseada, realizando un simple acceso a la mencionada página. Este es un proceso similar al que realiza el procesador cuando se trae una línea de *cache*.

Las estadísticas típicas para estos tipos de buffers son:

- Tamaño: entre 8 y 4096 entradas
- Tiempo en un *Hit*: 0.5 – 1 ciclo de reloj del procesador
- Penalización en un *Miss*: 10 – 30 ciclos de reloj del procesador
- Tasa de fallos: 0.01% - 1%

Con estos datos estadísticos, se puede calcular de forma sencilla el número de ciclos, por acceso a memoria. La tasa efectiva sería:

\[1 \times 0.99 + 30 \times 0.01 = 1.29 \text{ ciclos de reloj del procesador de promedio, por cada acceso a memoria.}\]
2.2 Entorno de trabajo

2.2.1 Sistema Operativo

Para realizar el proyecto, hemos escogido un sistema operativo Linux, en concreto la distribución SUSE Linux 10.0. El *kernel* de Linux que acompaña a esta distribución es la versión 2.6.13.

2.2.2 Compilador

Hemos utilizado el *GNU Compiler Collection 4.0.2* (abreviado gcc) que son un conjunto de compiladores de diversos lenguajes de programación producido por el Proyecto GNU. Es un software libre distribuido por la *Free Software Foundation*.

Es el compilador estándar para los diversos sistemas operativos libres, basados en UNIX y otros sistemas operativos derivados de este, como el Mac OS X.

Originalmente llamado GNU C Compiler, debido a que tan solo compilaba el lenguaje C, cambió posteriormente su nombre, al ser extendido a diversos lenguajes como C++, Java, Fortran, Ada, y otros.

Opciones del compilador

El compilador nos ofrece una serie de *flags* de optimización. De cara a probarlos y ver cuales funcionan mejor, realizaremos una serie de ejecuciones del CE; donde compararemos una serie de proteínas de diferentes tamaños con una proteína fija (de 1000 aminoácidos).

En la tabla de 2.1, mostramos los resultados, en segundos, obtenidos después de realizar las pruebas comentadas anteriormente.

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>-O0</th>
<th>-O1</th>
<th>-O2</th>
<th>-Os</th>
<th>-O3</th>
<th>-march</th>
</tr>
</thead>
<tbody>
<tr>
<td>1AFP (50)</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>1BUD (200)</td>
<td>21</td>
<td>13</td>
<td>17</td>
<td>15</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>1AMA (400)</td>
<td>84</td>
<td>48</td>
<td>61</td>
<td>59</td>
<td>43</td>
<td>72</td>
</tr>
<tr>
<td>1CY7 (600)</td>
<td>156</td>
<td>113</td>
<td>113</td>
<td>109</td>
<td>109</td>
<td>136</td>
</tr>
<tr>
<td>1M6N (800)</td>
<td>240</td>
<td>177</td>
<td>169</td>
<td>165</td>
<td>168</td>
<td>206</td>
</tr>
<tr>
<td>1WPE (1000)</td>
<td>105</td>
<td>83</td>
<td>75</td>
<td>72</td>
<td>80</td>
<td>88</td>
</tr>
</tbody>
</table>

Tabla 2.1 Segundos de ejecución obtenidos al comparar la proteína ITST de 1000 aminoácidos con cada una de las proteínas de la tabla, para los diferentes *flags* del compilador
Como vemos, los tiempos obtenidos ejecutando con el flag -O0 son mejorados en todas las comparaciones y por todas las opciones. Con lo que queda bastante claro, que la primera optimización va a resultar tan sencilla como elegir el flag del compilador que más nos convenga. Estudiando un poco más a fondo los demás resultados, (mirando tanto el cuadro de tiempos, como el gráfico de la figura 2.5) veremos que utilizar el flag –march=Pentium4, incomprensiblemente, para la aplicación CE es contraproducente, pues empeora los tiempos de ejecución con respecto a los demás códigos.

Mirando las opciones que quedan (-O1 -O2 -Os y -O3), parece que va a ser difícil la elección. Como hemos comentado el flag -O3 es el que más se “arriesga” a la hora de realizar optimizaciones, y fijándonos bien en los tiempos, en general supera a sus competidores. Por este motivo (aunque no obtendríamos resultados muy diferentes de trabajar con –O2 o –Os) hemos decidido elegir esta opción de compilación.

Con lo que, aunque nadie puede asegurar que obtengamos en general los mejores resultados con esta decisión, tampoco hay ningún indicio para suponer que esta elección sea errónea.

2.2.3 Oprofile 0.9.1

Es un potente profiler para sistemas de Linux, capaz de realizar un análisis de todos los programas que se están ejecutando en el procesador con una pequeña sobrecarga. Oprofile [10], al igual que gcc, está distribuido bajo la licencia GPL de GNU.
Consta de un controlador del núcleo (*kernel driver*) y un servidor que va guardando los datos del muestreo; además dispone de varias herramientas que una vez finalizada la captación de datos, transforma estos en información útil para el usuario.

*Oprofile* utiliza los contadores de la CPU para poder realizar el *profile* de una gran variedad de interesantes estadísticas. Si no indicamos lo contrario, todo código es analizado, ya se trate de interrupciones hardware o software, *kernel* del sistema, librerías, o nuestras aplicaciones.

**Opcontrol**

Se utiliza para preparar tanto el *Oprofile* como el entorno de trabajo. Las opciones más importantes (en general las que hemos utilizado):

- **--start-daemon**: inicializa el servidor
- **--init**: carga el módulo para poder ejecutar *Oprofile*. En principio solo hay que ejecutarlo en la primera ejecución
- **--reset**: como su nombre indica, resetea los datos de todos los ficheros (una vez más, lo he utilizado junto con **--init**, antes de cada ejecución)
- **--stop**: para la recaptación de datos
- **--shutdown**: igual que **--stop**, pero además mata el *daemon*
- **--deinitialize**: sirve para descargar el módulo (para el *FTDock*, después de varios problemas con la actualización de los ficheros de *Oprofile*, decidimos ejecutar esta opción, siempre antes, de la cargar el módulo para asegurar la eliminación de los datos antiguos)
- **--no-vmlinux**: se utiliza cuando no se tiene un *kernel* vmlinux, y no se quiere tomar muestras de este *kernel*
- **--events**: sirve para elegir el evento (o eventos) que queremos analizar. Hay que especificar el número de eventos que se deben producir para que el programa cuente un *sample*. Además para cada evento, suele haber una serie de subeventos, que también hay que especificar. En la tabla 2.2, aparecen los eventos más utilizados
<table>
<thead>
<tr>
<th>Nombre</th>
<th>Descripción</th>
<th>Contadores usables</th>
<th>Opciones de bit mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBAL_POWER_EVENTS</td>
<td>Time during which processor is not stopped</td>
<td>0, 4</td>
<td>0x01: mandatory</td>
</tr>
<tr>
<td>BRANCH RETIRED</td>
<td>Retired branches</td>
<td>3, 7</td>
<td>0x01: branch not-taken predicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x02: branch not-taken mispredicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x04: branch taken predicted</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x08: branch taken mispredicted</td>
</tr>
<tr>
<td>BUS CACHE REFERENCE</td>
<td>Cache references seen by the bus unit</td>
<td>0, 4</td>
<td>0x01: read 2nd level cache hit shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x02: read 2nd level cache hit exclusive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x04: read 2nd level cache hit modified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x08: read 3rd level cache hit shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x10: read 3rd level cache hit exclusive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x20: read 3rd level cache hit modified</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x100: read 2nd level cache miss</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x200: read 3rd level cache miss</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x400: writeback lookup from DAC misses 2nd level cache</td>
</tr>
<tr>
<td>INSTR RETIRED</td>
<td>Retired instructions</td>
<td>3, 7</td>
<td>0x01: count non-bogus instructions which are not tagged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x02: count non-bogus instructions which are tagged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x04: count bogus instructions which are not tagged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x08: count bogus instructions which are tagged</td>
</tr>
</tbody>
</table>

Tabla 2.2 Principales eventos que hemos utilizado con Oprofile en un Pentium 4

Oprepor

Esta es una de las dos utilidades mediante las cuales mostraremos los datos que se han recogido con Oprofile.

Oprepor puede sacar un listado de las funciones ejecutadas, ordenadas por la cantidad de samples (del evento especificado. También es posible obtener los samples obtenidos a lo largo de toda la ejecución, sin tener en cuenta funciones, librerías,... En la mayoría de nuestras ejecuciones, un sample equivaldrá a 100000 eventos del sistema (aunque podemos especificar el valor que se desee, a partir de un mínimo de 3000).

Principales opciones que utilizamos con oprepor:

- `-l`: aparece la información simbólica (listado de funciones)
- `-t / --threshold X`: solo aparecen las funciones que contabilicen un X% del total de los eventos
- `Image:/path/executeable`: solo aparecerán los datos correspondientes al ejecutable especificado
**Opannotate**

Parecido al opreport, pero en este caso los samples se muestran por línea de código. Se puede obtener para el código fuente (--source) independientemente del código máquina o assembler (--assembly), o una mezcla combinada de código fuente y assembler (--source --assembler).

Para Opannotate, aparte de estas dos opciones que acabamos de comentar, la única opción remarcable es:

- Image:/path/executable: solo aparecerán los datos correspondientes al executable especificado

### 2.2.4 Gprof

Es un profiler estándar para los sistemas Linux. A diferencia del ya explicado Oprofile, Gprof no accede a los contadores hardware del procesador, con lo que no es capaz de mostrar información sobre eventos.

Pero por otro lado, (esta es el motivo por el que en alguna ocasión lo hemos utilizado) Gprof es capaz de informarnos del número exacto de invocaciones de cada rutina.

El funcionamiento de Gprof es bastante más sencillo que Oprofile. Tan solo hay que compilar el programa que se quiere analizar con los flags -g y -pg. Una vez compilado el programa, lo ejecutamos; y una vez finalizada esta, aparecerá un fichero gmon.out. Es ahí donde reside la información de la ejecución de nuestro programa.

Una vez tenemos el fichero, solo nos falta ejecutar Gprof. Por defecto, muestra:

- lista de las rutinas del programa, ordenadas por tiempo consumido en la CPU
- representación del grafo de invocaciones de las funciones del programa
- líneas de texto explicando el significado de la información mostrada
Las principales opciones de *Gprof* utilizadas, han sido:

- `-l`: muestra la misma información pero por líneas de código
- `-q`: únicamente muestra el *call graph*
- `-b`: no muestra el texto con el significado de la información mostrada

Esta herramienta, apenas ha sido utilizada. Principalmente, por que casi todas las prestaciones que ofrece, están disponibles (y posiblemente muy mejoradas) con *Oprofile*. Quitando unas primeras ejecuciones (pues como hemos comentado es bastante más sencillo de utilizar que *Oprofile*), tan solo lo hemos requerido en momentos en los que era importante saber el número de invocaciones de alguna función.

### 2.2.5 Time

Muestra información básica respecto al tiempo de ejecución de un proceso. Concretamente muestra:

- Tiempo de ejecución en modo usuario
- Tiempo de ejecución en modo sistema
- *Elapsed time* (el tiempo total)

Hemos utilizado esta herramienta, básicamente para tomar los tiempos de ejecución tanto de códigos originales como de los optimizados, y así calcular los *speedup* (referentes al tiempo de ejecución) obtenidos para las aplicaciones.

### 2.2.6 Pin

Es una herramienta que sirve para realizar la instrumentalización de programas.

A diferencia de otros programas de este estilo, Pin [11] no reescribe estáticamente el código de un ejecutable, sino que instrumentaliza mediante la inclusión de eventos y/o instrucciones en tiempo de ejecución (generalmente se incluyen
instrucciones tipo incremento de contadores) a lo largo del código que queremos analizar.

De cara a no interferir en la ejecución del programa, y que esta sea lo más parecida a una ejecución real, Pin se encarga de guardar y cargar los registros que son sobrescritos por el código introducido por la aplicación.

Pin viene con una serie de programas de ejemplo (con los conocimientos suficientes podríamos crear nuestro propio programa que sería capaz de simular el funcionamiento de gran parte de nuestra arquitectura para cualquier programa). Nosotros hemos utilizado Opcodemix, que genera un histograma basado en el número de instrucciones dinámicas ejecutadas. Este programa crea además otro histograma basado en los accesos a memoria y pila, estudiando para los primeros si el acceso fue de 1, 2, 4, 8 o 16 bytes.

Hemos decidido agrupar las instrucciones, ya que la cantidad de IA-32 que se ejecutan nuestras aplicaciones son bastante elevadas. Estos grupos son:

- *Load, store y Aritmética* (tanto de enteros como de *floats*)
- Instrucciones de tipo MOV
- Instrucciones de salto
- Vectoriales (separadas en MOV y aritmética)
- Resto de instrucciones
3 El Intel del Pentium 4

3.1 Resumen del Procesador

El Pentium 4 o también llamado y escrito Pentium IV, es un microprocesador de séptima generación basado en la arquitectura x86 y manufacturado por Intel. Es el primer microprocesador con un diseño completamente nuevo desde el Pentium Pro de 1995 (este nuevo tipo de microarquitectura Intel recibe el nombre de Intel NetBurst [1][4][9]).

Las principales características y funciones de la microarquitectura NetBurst son:

• Nuevo front-end de la máquina
• Nueva cache de instrucciones (Execution Trace Cache)
• Jerarquía de memoria, con una nueva memoria cache de datos de nivel 1 (L1) a una muy baja latencia (con tan solo 2 ciclos de reloj del procesador de tiempo de acceso)
• Nuevo motor de ejecución out-of-order para mejorar el paralelismo
• Nueva ALU de baja latencia, capaz de realizar dos operaciones por ciclo
• Mayor rendimiento con operaciones de coma flotante
• Hyper-pipeline (híper-segmentación) que permite una velocidad de reloj elevada
• Rápido motor de ejecución que reduce la latencia de las instrucciones básicas con enteros

Líneas de cache de 64 bytes Los principales objetivos de esta nueva tecnología de Intel son:

• Ejecutar las antiguas aplicaciones IA-32, y que a la vez se permita la ejecución de aplicaciones basadas en la tecnología single-instruction, multiple-data (SIMD) para conseguir un elevado throughput
• Trabajar con altas frecuencias de reloj y obtener un mayor rendimiento y una mayor velocidad del reloj en el futuro
Ventajas del diseño de la microarquitectura Intel NetBurst:

- Diseño más segmentado que permite una mayor frecuencia del reloj (con diferentes partes del chip funcionando a diferentes velocidades)
- Una segmentación que optimiza las instrucciones ejecutadas frecuentemente en circunstancias normales (como un acierto de cache); estas serán decodificadas eficientemente y ejecutadas con latencias mínimas
- Uso de técnicas para evitar penalizaciones; entre estas: ejecución en paralelo de instrucciones, buffering y especulación. La microarquitectura ejecuta instrucciones dinámicamente y out-of-order, con lo que el tiempo que se toma para ejecutar cada instrucción individualmente no es siempre determinista

El Pentium 4 original, denominado Willamette, trabajaba a 1,4 y 1,5 GHz; y fue lanzado en noviembre de 2000 y para sorpresa de la industria informática, no mejoró el viejo diseño P6 según las dos tradicionales formas para medir el rendimiento:

- velocidad en el proceso de enteros
- operaciones de coma flotante

La estrategia de Intel fue sacrificar el rendimiento de cada ciclo para obtener a cambio mayor cantidad de ciclos por segundo y una mejora en las instrucciones SSE (Streaming SIMD Extensión 2). Para ello, se añadieron 144 nuevas instrucciones de 128 bits Single Instruction Multiple Data (SIMD).

Además sus 42 millones de transistores de 180 nanómetros incluidos en 217mm² que consumían 55W a 1.5GHz y su velocidad del bus a 3.2GB/s, proporcionaban un ancho de banda de datos muy elevado, que satisfacían con creces las necesidades de las aplicaciones de la época, pero sobre todo estaba previsto para satisfacer la demanda de datos de aplicaciones venideras.

Las distintas versiones del procesador Pentium 4 son: Willamette, Northwood, Extreme Edition y Prescott. Nosotros haremos una pequeña descripción del Northwood, que es el procesador con el que hemos estado trabajando.
3.2 Visión General del procesador

En la figura 3.2, podemos ver un esquema de la microarquitectura Intel NetBurst del Pentium 4, con sus respectivas secciones.

Fig. 3.2 Esquema de la microarquitectura Intel NetBurst, donde se diferencian 4 partes, claramente delimitadas: front-end, motor fuera de orden, unidades de ejecución de enteros y floats y subsistema de memoria.

El front-end envía a ejecutar las instrucciones del programa al núcleo de la unidad out-of-order. También se encarga de buscar y decodificar las instrucciones IA-32 en microinstrucciones, que serán enviadas de forma continua al núcleo de ejecución, en el orden original del programa.

El out-of-order core (o núcleo de ejecución fuera de orden) realiza una planificación mediante la reordenación de las microinstrucciones, de forma que cuando una microinstrucción tiene sus “fuentes” preparadas y quedan unidades funcionales libres, se puede enviar a la unidad de ejecución tan pronto como sea posible (fuera de orden). Este núcleo permite la ejecución de múltiples microinstrucciones por ciclo.

La unidad Retirement, reordena las instrucciones ejecutadas asegurando que el resultado de la ejecución se procese de acuerdo al orden original del programa, y que los diferentes estados de la arquitectura sean actualizados de forma correcta.
Como acabamos de ver, la arquitectura se divide en 4 grandes etapas. Pero en cambio, en la explicación que acabamos de dar, hemos mencionado tan solo tres unidades (las más importantes del procesador out of order). Estas diferencias aparecen por el hecho que las unidades de ejecución, no pertenecen claramente a una etapa en concreto:

- Unidades de ejecución pertenecen, junto con la L1 cache, a la etapa “Integer and FP Execution Units” (Figura 3.2)
- Unidades de ejecución se unen con la lógica fuera de orden, para formar el núcleo de ejecución, la unidad de retirement trabaja aislada de este núcleo, y finalmente el primer nivel de cache pasa a formar parte de la jerarquía de memoria (Figura 3.3)

Fig. 3.3 Otra forma de esquematizar la microarquitectura NetBurst. En esta figura, se ve toda la jerarquía de memoria junta, mientras que en la figura 3.2, el primer nivel de cache aparecía en otra parte
3.3 Pipeline del Pentium 4

El pipeline (segmentación) de los primeros Pentium 4 estaba formado por 20 estados (a partir del Pentium Prescott, pasaron a ser 31). Como ya hemos comentado, nuestro procesador es previo al Prescott con lo que explicaremos los principales estados del pipeline original del Pentium 4.

Para hacernos una pequeña idea, en la figura 3.4 aparecen las etapas del pipeline.

---

**Fig. 3.4** En el esquema, podemos ver las 20 etapas del pipeline ordenadas secuencialmente.

En la figura 3.5, podemos ver un diagrama de la microarquitectura del Pentium 4. Este, será de gran ayuda para comprender (y poder visualizar) las partes más importantes del hardware que van a ser descritas a continuación.

---

**Fig. 3.5** Microarquitectura detallada del Pentium 4.
3.3.1 Fase de fetch

**In-order front-end, Prefetching, ITLB y BTB**

Como ya habíamos visto, el *front-end* consta principalmente de dos partes:

- Unidades de *fetch y decode* (búsqueda y decodificación)
- *Trace cache*

Las funciones que tiene que realizar son:

- Realizar una búsqueda previa (*prefetch*) de instrucciones IA-32, que es posible que se ejecuten
- Buscar las instrucciones que se necesitan y no habían sido buscadas con anterioridad
- Decodificar las instrucciones IA-32 en microinstrucciones
- Generar micro-código para instrucciones complejas y de "propósito especial"
- Enviar las microinstrucciones decodificadas de la *trace cache* al núcleo de ejecución
- Predecir saltos usando algoritmos avanzados

El *front-end* es la parte de la máquina que busca las siguientes instrucciones a ejecutar del programa y las prepara para ser utilizadas posteriormente en el *pipeline*. Su trabajo consiste en proporcionar una elevada secuencia de instrucciones decodificadas al núcleo de ejecución *out-of-order*, que se encargará de completar la instrucción.

Para lograr estas tasas de decodificación de instrucciones, se requiere una precisa lógica de predicción de saltos, pues una elevada cantidad de fallos en la predicción supondría una pérdida importante del rendimiento. En este contexto, aparece el *branch target buffer* (BTB).

El BTB es un buffer (de 4KBytes) que contiene en una lista, la historia de los saltos y los posibles destinos, para poder especular e intentar predecir, con mayor éxito, la siguiente parte del código a ejecutar. Aun así, cuando un salto no es encontrado, se predice estáticamente. Saltos hacia delante, "no saltar" y saltos hacia atrás, "saltar". Las
direcciones de las instrucciones predichas por esta unidad lógica, se usan para buscar las instrucciones en la cache de segundo nivel.

El BTB junto con el instruction fetcher, buscan las instrucciones IA-32 que predicen se van a ejecutar a continuación. El predictor de saltos permite al procesador buscar e incluso ejecutar instrucciones, bastante antes, que el salto previo sea confirmado.

El instruction fetcher simplemente se encarga de leer (en bloques de 32bytes, empezando por la dirección de destino) las instrucciones que va prediciendo el BTB.

En esta fase inicial, las instrucciones son buscadas y decodificadas en secuencias de microinstrucciones llamadas Traces, que se almacenan en la Execution Trace Cache (una avanzada forma de cache de instrucciones de L1, que explicaremos más adelante).

Para aquellos casos en que se falla en el acceso a la trace cache. El procesador utiliza el ITLB (Instruction Translation Lookaside Buffer) que es el encargado de dirigir junto con el BTB la fase inicial. El ITLB simplemente se encarga de traducir las direcciones lógicas en direcciones físicas, para poder acceder al L2 cache.

Decodificador y ROM de microinstrucciones

El procesador tan solo dispone de un decodificador que trabaja a una tasa máxima de una decodificación por ciclo (solamente se utiliza cuando la máquina falla en su acceso a la trace cache y debe acceder a la L2 cache para traer y decodificar la nueva instrucción). Recibe 64-bits del 2º nivel de cache, que traduce en microinstrucciones. La mayoría de estas instrucciones IA-32 se traducen en una sola microinstrucción, pero cuando las instrucciones son algo complejas, estás se pueden transformar en más de 4 microinstrucciones. Para estos casos, el procesador se ayudan de la ROM de microinstrucciones (esta ROM funciona como una pequeña librería para instrucciones complejas, cuando una de ellas es encontrada, la trace cache salta a la ROM, y cuando finaliza de leer las microinstrucciones correspondientes, el front-end se encarga de continuar lanzando las pertinentes instrucciones).
**Trace cache**

Es la memoria cache de instrucciones primaria. A diferencia de otras cache de instrucciones, esta se sitúa entre la lógica de decodificación de instrucciones y el núcleo de ejecución. Ahí, la *trace cache* es capaz de almacenar las instrucciones decodificadas. Puede llegar a lanzar las microinstrucciones al motor de ejecución fuera de orden a una tasa de 3 por ciclo.

La mayor parte de las veces, las instrucciones se ejecutan provenientes de la *trace cache*; pero cuando se produce un fallo en esta memoria primaria, (como habíamos comentado anteriormente en la decodificación) el procesador debe ir a buscar la instrucción a la L2 cache y decodificarla.

Esta memoria toma las microinstrucciones decodificadas y las agrupa ordenadamente (conforme al código original) en grupos de 6, formando una *trace line*.

La *trace cache* tiene su propio BTB (*trace cache BTB*) que indica al *instruction fetcher* a qué lugar de debe ir a continuación en la *trace cache*. Este BTB es bastante más pequeño que el del *front-end* pues solo debe indicar a que parte debe saltar, de una porción del código del programa.

### 3.3.2 Lógica de ejecución Out-of-Order

La habilidad del procesador para ejecutar instrucciones fuera de orden, es un factor importante a la hora de permitir el paralelismo. Esta característica permite al procesador, por medio de diversos *buffers*, reordenar las instrucciones, de forma que mientras algunas microinstrucciones permanecen esperando datos o recursos; otras, que aparecen posteriormente en la secuencia del programa original, las pueden preceder. Esto significa que cuando una parte del *pipeline* sufre un retraso, este puede ser cubierto por otras operaciones que se ejecutan en paralelo o por cualquier microinstrucción que esté en la cola del buffer.
**Allocator y el Register Renaming**

Para que el procesador pueda ejecutar tantas instrucciones por ciclo como le sea posible, debe enviar todas las instrucciones disponibles que pueda por cada ciclo de reloj del procesador, incluso si estas no están en el orden original del programa. Resulta obvio, que cuanto más amplio sea el número de instrucciones que el procesador tenga en cuenta, mayor será el número de instrucciones disponibles e independientes a ejecutar.

La arquitectura NetBurst permite realizar un profundo buffering. Permitiendo en condiciones óptimas tener hasta un máximo de 126 instrucciones por ejecutar, entre las cuales puede haber 48 loads y 24 stores.

El allocator se encarga de asignar las microinstrucciones a los diferentes buffers procesador. Si los recursos de una de las tres instrucciones que le llegan cada ciclo no está disponible, el allocator para la ejecución para esta instrucción hasta que sea posible su ejecución. En ese momento, asigna el resultado en el Reorder buffer (ROB), y se completan los correspondientes registros de las 126 posibles instrucciones.

El motor de ejecución fuera de orden, como se ve (Figura 3.6), tiene varios buffers donde rastrear, reordenar y secuenciar instrucciones.

*Register renaming* se trata de una lógica que da diferentes nombres a los registros lógicos IA-32, convirtiendo los 8 registros en una tabla de 128 alias (RAT, *register alias table*). Mediante esta tabla podríamos tener decenas de instancias para los registros EAX, EBX,... a la vez en el pipeline de la máquina.
Planificación de microinstrucciones

La planificación de microinstrucciones determina cuando una microinstrucción está lista para ser ejecutada. La base de la planificación fuera de orden no es otra que saber en qué momento están disponibles:

- los registros de entrada de los operandos
- las unidades funcionales a utilizar
- los registros de escritura

Hay dos colas físicas en el sistema:

- para instrucciones de memoria
- para las instrucciones que no acceden a memoria.

Estas colas utilizan el algoritmo FIFO (first-in, first-out), pero permiten una lectura out-of-order para poder realizar la planificación dinámica.

La planificación de las microinstrucciones, se realiza en función de la disponibilidad de los 3 elementos mencionados anteriormente, pero los planificadores deben tener en cuenta la forma en que las unidades funcionales están conectadas a los 4 puertos de envío de instrucciones. Con lo que según que par de instrucciones no se pueden ejecutar a la vez (esto quedará más claro en el apartado donde veremos las unidades de ejecución).

3.3.3 Instrucciones vectoriales

Se trata de un tipo de instrucciones que permiten efectuar varias operaciones sobre un vector de datos en una sola instrucción [6][8], mientras que las instrucciones escalares solo pueden operar sobre un dato en cada instrucción.

En muchas aplicaciones, lo que se hace es aplicar el mismo conjunto de operaciones sobre un conjunto de datos. Esto significa, aplicar un número reducido de instrucciones sobre un conjunto amplio de datos, de forma repetitiva.
El principal motivo para utilizar este tipo de instrucciones, no es otro que el de acelerar el rendimiento de las aplicaciones que las usan, ya que estas permiten especificar para una instrucción vectorial cual es la operación que hay que realizar, y al mismo tiempo especificar el conjunto de datos sobre el que realizar la operación (de ahí el nombre Single Instruction, Multiple Data o SIMD).

Otra ventaja importante acerca de estas instrucciones, es que se puede lograr paralelismo a nivel de datos.

En nuestro procesador disponemos de los siguientes conjuntos de instrucciones:

- MMX: conjunto de 57 nuevas instrucciones para datos de 64 bits. Se puede operar con vectores, de 8 bytes, 4 words o 2 doubleword
- SSE: también Streaming SIMD Extensión, dispone de un banco de registros XMM de 128-bits, capaz de operar con vectores de 16 bytes, 8 words, 4 doublewords o 2 quadwords.
- SSE2: se añadieron más de 100 nuevas funciones al conjunto de instrucciones SSE. Todas ellas operaban con vectores del mismo tipo que los utilizados por el conjunto anterior.

Para poder trabajar con este tipo de instrucciones, debemos asegurarnos que no hay ningún tipo de dependencia de datos, ya que las operaciones se realizan en paralelo. Con lo que, de los ejemplos de la figura 3.7, el primero se podrá transformar en una instrucción vectorial, y el segundo no.

\[
\begin{align*}
A[i] &= B[i] + C[i]; \\
A[i+1] &= B[i+1] + C[i+1]; \\
A[i+2] &= B[i+2] + C[i+2]; \\
A[i+3] &= B[i+3] + C[i+3];
\end{align*}
\]

\[
\begin{align*}
A[i+1] &= A[i] + B[i+1]; \\
\end{align*}
\]

Fig. 3.7 A la izquierda podemos ver un conjunto de instrucciones vectorizables, mientras que el conjunto de la derecha no se puede vectorizar debido a una dependencia de datos

Posteriormente, Intel ha lanzado un nuevo conjunto de instrucciones vectoriales, a las que ha bautizado con el nombre SSE3. Pero estas, al no estar disponibles en nuestro procesador, nos limitamos comentar su existencia pero no explicaremos más detalles sobre las longitudes con que opera, las novedades que incorpora, etc.
3.3.4 Latencia y Throughput

El comportamiento de las instrucciones IA-32 viene definido por sus dos características más importantes: latencia y throughput [7]. El conjunto de instrucciones más importantes (que podemos encontrar en el apartado 3.3.4.1), contienen generalmente diversa información, de la cual, a nosotros solo nos interesaran estas dos, y para entender su significado, explicaremos ahora su definición.

Latencia: es el número de ciclos de reloj del procesador que son necesarios para que el núcleo de ejecución complete la ejecución de todas las microinstrucciones que forman una instrucción IA-32.

Throughput: Número de ciclos del reloj del procesador, que se deben esperar antes de que los puertos de envío (explicados en el apartado 3.3.5.1) estén libres para aceptar una nueva instrucción del mismo tipo.

Como veremos en el conjunto de instrucciones, para la mayoría de instrucciones IA-32, el throughput suele ser bastante menor que la latencia.

Para entenderlo, pondremos un ejemplo sencillo, en el que supondremos que solo disponemos de una unidad funcional para realizar sumas. Si tenemos 5 sumas consecutivas con una latencia de 3 ciclos y un throughput también de 3 ciclos, nos encontraremos que cada 3 ciclos acaba una instrucción y puede empezar otra; bajo estas condiciones, las 5 instrucciones tardarán 15 ciclos de reloj en completarse.

En cambio, en el caso que las sumas tuviesen una latencia de 7 ciclos y un throughput de 1 ciclo, si tenemos las mismo 5 sumas consecutivas de antes, comenzaría una en cada ciclo (la quinta suma en el quinto ciclo) y finalizarían todas 7 ciclos después de haber empezado. Esto quiere decir que la última finalizaría en el ciclo 5 + 7.

 Esto demuestra que no es tan importante el número de ciclos que tarda en completarse una instrucción (latencia), como el tiempo que tenga que esperar una instrucción del mismo tipo, para ejecutarse (throughput). Pues en el segundo caso a pesar de tener una latencia de más del doble, las cinco sumas acaban finalizándose 3 ciclos antes, gracias a un throughput menor.
Latencia y throughput de las principales instrucciones

En la tabla 3.1, mostramos las principales instrucciones que utilizan nuestras dos aplicaciones (hemos escogido las que más tiempo consumen y las que en mayor número aparecen). En ellas, mostramos el nombre de la instrucción, la latencia y el throughput.

<table>
<thead>
<tr>
<th>Instr</th>
<th>Lat</th>
<th>Thr</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>FSUB</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>FMUL</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>FDIV (sp)</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>FDIV (dp)</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>FDIV (sp)</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>FABS</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instr</th>
<th>Lat</th>
<th>Thr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD/SUB</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>AND/OR/XOR</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>CMP/TEST</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>DEC/INC</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>IMUL</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>MOV</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>PUSH</td>
<td>1.5</td>
<td>1</td>
</tr>
</tbody>
</table>

Tabla 3.1 Tabla de instrucciones con las latencias y throughputs

Hay que tener presente que:

- Las latencias para instrucciones complejas son una estimación conservadora, suponiendo siempre el peor caso. Con lo que la latencia normal suele ser algo más rápida (en algunos casos la diferencia puede ser considerable).

- Generalmente, cuando uno de los dos operandos es una dirección de memoria, se añade una microinstrucción (load o store): afectando al comportamiento de la instrucción, pues aumenta la latencia. A pesar de esto, el throughput se mantiene, ya que estas instrucciones utilizan los puertos de envío 2 o 3, y estos no interfieren en las operaciones.

- Para cada acceso a memoria, se asume que se producirá un L1 cache hit.

3.3.5 Unidades de ejecución de enteros y de coma flotante

Puertos de envío (issue ports)

La unidad de ejecución puede lanzar hasta 6 microinstrucciones por ciclo (observemos que esta tasa es muy superior a las 3 microinstrucciones que puede enviar la trace cache). Este mayor ancho de banda, permite en momentos de sobrecarga de instrucciones, por medio de elevadas tasas de envío, una mayor flexibilidad.
Pero hay que tener en cuenta que tan solo se dispone de 4 puertos de envío (*issue ports*), y que las unidades funcionales están conectadas a estos 4 puertos de una forma concreta; con lo que no siempre será posible alcanzar la tasa de 6 microinstrucciones por ciclo de reloj. En la figura 3.8 se muestra un esquema de la conexión de los puertos a las unidades funcionales.

**Fig. 3.8 Esquema con los 4 puertos de envío, las unidades de ejecución y sus conexiones**

1. **Puerto 0**: (en la figura 3.8, “*Exec Port 0*”) disponemos de una unidad aritmético-lógica (ALU) de doble velocidad, con lo que se pueden realizar dos operaciones en un ciclo. En la primera mitad de ciclo puede realizar: o bien una operación de tipo *FP Move* o bien una operación aritmética. En cambio en la segunda mitad de ciclo tan solo se puede realizar una operación aritmética en la ALU.

2. **Puerto 1**: (en la figura 3.8, “*Exec Port 1*”) en la primera mitad de ciclo se puede o bien ejecutar una operación con decimales (incluidas todas las SIMD, a excepción de un *move*), o una instrucción con enteros (a velocidad normal) o bien una operación aritmética. En este caso también disponemos de una ALU a doble velocidad, con lo que en la segunda mitad del ciclo, podremos ejecutar otra operación aritmética.

3. **Puerto 2**: (en la figura 3.8, “*Load port*”) este puerto solo puede enviar una operación de carga (*load*) por ciclo.

4. **Puerto 3**: (en la figura 3.8, “*Store port*”) este puerto solo puede enviar una operación de almacenamiento (*store*) por ciclo.

Con esto se puede deducir, que solo se pueden lanzar 6 microinstrucciones por ciclo, cuando al inicio de este se envíen 4 instrucciones (una por puerto), y se envían 2 instrucciones más (mediante las ALU de doble velocidad) a mitad de ciclo.
Unidad de ejecución de enteros

Esta unidad se compone (figura 3.9) de 2 ALUs de doble velocidad (es decir, que se pueden llegar a ejecutar 2 microinstrucciones por unidad en el mismo ciclo de reloj del procesador) y además otra unidad llamada *Integer operation* encargada de realizar las instrucciones más complejas como multiplicaciones, divisiones, rotaciones o desplazamientos.

Las ALUs de doble velocidad (también reciben el nombre de “ALU de baja latencia”), logran completar hasta dos instrucciones por ciclo. En parte, estos es gracias al cortocircuito (*bypass* en inglés) ALU-ALU (se trata de una conexión que une la salida del la ALU con la entrada, permitiendo que justo cuando se acaba de obtener el resultado de una operación, este esté disponible en la entrada, sin tener que esperar a que sea almacenado en ningún registro o posición de memoria).

Esta comprobado que aproximadamente entre el 60% y el 70% de las microinstrucciones de la mayoría de los programas que trabajan con enteros utilizan este *bypass*. Obviamente, si este elevado porcentaje de instrucciones, se realizan en la mitad de tiempo, no se puede esperar un *speed-up* de 2, pero sí una mejora importante en el rendimiento de la mayoría de aplicaciones que trabajan con enteros.

Como ya hemos dicho las instrucciones más complejas (las de una latencia elevada) pasan a ejecutarse a una unidad con más hardware para ser completadas. El tipo de operaciones que requieren esta unidad, son aquellas cuya latencia es mayor. Este es el caso de operaciones de desplazamiento o rotación, con latencias alrededor de los 4 ciclos; pero sobretodo multiplicaciones y divisiones, con latencias próximas a los 14 y 60 ciclos de reloj del procesador, respectivamente.

Hay que tener en cuenta lo que comentamos en el apartado 3.3.4. El hecho que una instrucción tenga una latencia elevada, no supone mayor inconveniente; siempre y cuando su throughput sea de pocos ciclos.
Cache de datos de primer nivel

La L1 cache de datos de baja latencia, es una memoria de tamaño relativamente pequeño (en comparación con otras memorias como la L2 cache o la RAM), apenas 8KBytes, con un tamaño de línea de 64 Bytes. Forma parte de los pipelines de ejecución tanto de enteros y floats como del pipeline de las instrucciones SIMD. Se trata de una cache write-through o lo que es lo mismo, cada vez que se escribe en la L1 cache, se escribe en la L2 cache. Puede realizar una lectura y una escritura a la vez cada ciclo de reloj del procesador.

Su baja latencia (2 ciclos para lectura de enteros y 6 ciclos para floats) se hace especialmente importante en aquellos programas en que abundan los accesos a memoria, ya que es en estos casos cuando se hace patente la escasez de registros para el conjunto de instrucciones.

Se da el caso que la distancia en ciclos entre la planificación de instrucciones y la correspondiente ejecución es mayor que la latencia de la ejecución de lectura (tarda más una instrucción planificada en ejecutarse, que realizar una lectura de la L1 cache). Por consiguiente el procesador envía instrucciones dependientes de una lectura, con la esperanza de que se produzca un hit en la L1 cache. De no ser así, se produce un error en los datos (pues el fallo supone acceder a la L2 cache, con una latencia mayor mientras la instrucción dependiente ya ha leído la inconsistencia). Esto no supone mayor problema que la pérdida de unos ciclos (siempre y cuando no se produzca en demasiadas ocasiones tal hecho), pues un mecanismo llamado replay se encarga de volver a ejecutar todas las instrucciones que trabajan con datos incorrectos.

Store-to-load forwarding

Dado que un almacenamiento no se considera finalizado hasta que ha actualizado en la memoria sus datos, se sobreentiende que las demás instrucciones deben esperar a que esta instrucción sea retirada. Dada la profundidad de la segmentación del procesador, a un store le lleva demasiado tiempo completar su ejecución.
Si tenemos en cuenta que:

- Un *store* pude tener que esperar a otro.
- Que en el *pipeline* se pueden acumular hasta 24 instrucciones de este tipo.
- A menudo los *loads* utilizan los resultados de cualquiera de estos *stores* pendientes.

Nos encontramos ante un problema importante, pues el rendimiento del procesador puede caer drásticamente en programas que requieran muchos registros (de los que apenas disponemos 8). Para solucionarlo, es necesario permitir de alguna forma el uso de estos *stores* pendientes.

... Esto es posible gracias a un **buffer** de almacenamientos pendientes, que permite a las cargas utilizar los datos obtenidos tras la ejecución, a pesar de que no hayan sido escritos en la L1 *cache* de datos.

**Unidad de ejecución de floats y SIMD**

Las unidades de ejecución de instrucciones en coma flotante y las instrucciones SIMD, comparten *pipeline* y por consiguiente sus unidades funcionales. En este caso, para estos tipos de instrucciones tenemos dos unidades funcionales, cada una conectada a un puerto de envío diferente y capaces de empezar (cada una de ellas) una nueva instrucción a cada ciclo de reloj.

Conectada al puerto 0, nos encontramos con FP Move. Esta unidad, que trabaja con dos operandos desde 64 y 128 bits, se encarga de las instrucciones de *move* de registro a registro o para aquellas instrucciones que requieran escritura en memoria. Por otro lado, conectada al puerto 1, nos encontramos con la FP *Execute*, capaz de realizar una suma de 128 bits SSE/SSE2, cada dos ciclos. Entre las dos, pueden llegar a alcanzar picos de más de 10GFLOPS en nuestro procesador a 2.6GHz.
La cache de datos, es la misma L1 cache que utiliza el pipeline de enteros (véase apartado 3.3.5.3).

En esta unidad de ejecución se pueden encontrar tres tipos diferentes de instrucciones:

- Coma flotante
- MMX
- XMM

Para cada una de ellas el procesador pone a nuestra disposición 8 registros de longitud variable (figura 3.11), en función de la longitud máxima del tipo operandos con los que vamos a trabajar.

Fig. 3.11 Resumen de los registros que componen la unidad de ejecución de floats e instrucciones SIMD y del correspondiente tamaño de estos