Arquitecturas de altas prestaciones

Ivan Rodero Castro
Francesc Guim Bernat

PID_00191919

UOC
Universitat Oberta de Catalunya

Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundación para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es

Índice

Introducción 5

Objetivos 8

1. Descomposición funcional y de datos 9

2. Taxonomía de Flynn 13

3. Arquitecturas de procesador *SIMD* 16
- 3.1. Propiedades de los procesadores vectoriales 18
- 3.2. Ejemplo de procesador vectorial 19

4. Arquitecturas de procesador multihilo o MIMD 21
- 4.1. Arquitecturas superthreading 23
  - 4.1.1. Compartición a nivel fino 23
  - 4.1.2. Compartición a nivel grueso 25
- 4.2. Arquitecturas simultaneous multithreading 26
- 4.3. Convirtiendo el paralelismo a nivel de hilo a paralelismo a nivel de instrucción 27
- 4.4. Diseño de un *SMT* 28
- 4.5. Complejidades y retos en las arquitecturas *SMT* 30
- 4.6. Implementaciones comerciales de *SMT* 31
  - 4.6.1. El HyperThreading de Intel 31
  - 4.6.2. El Alpha 21464 35
- 4.7. Arquitecturas multinúcleo 37
  - 4.7.1. Limitaciones de la *SMT* y arquitecturas superthreading 37
  - 4.7.2. Producción 39
  - 4.7.3. El concepto de multinúcleo 39
  - 4.7.4. Coherencia entre núcleos 41
  - 4.7.5. Protocolos de coherencia 44

5. Factores determinantes en el rendimiento en arquitecturas modernas 47
- 5.1. Factores importantes para la ley de Amdahal en arquitecturas multihilo 48
- 5.2. Factores vinculados al modelo de programación 49
  - 5.2.1. Definición y creación de las tareas paralelas 49
  - 5.2.2. Mapeo de tareas a hilos 51
  - 5.2.3. Definición e implementación de mecanismos de sincronización 53

5.2.4. Gestión de acceso concurrente a datos 57
5.2.5. Otros factores que hay que considerar 62
5.3. Factores ligados a la arquitectura 62
5.3.1. Compartición falsa 63
5.3.2. Penalizaciones para fallos en la L1 y técnicas de *prefetch*.. 66
5.3.3. Impacto del tipo de memoria caché 68
5.3.4. Arquitecturas multinúcleo y multiprocesador 71

Resumen.. 74

Actividades.. 77

Bibliografía.. 78

Introducción

Durante muchos años el rendimiento de los procesadores ha ido explotando el nivel de paralelismo inherente a los flujos de instrucciones de una aplicación. En 1971 se puso a la venta el primer microprocesador simple, el llamado Intel 4004. Este procesador, de 4 bits, estaba formado por una sola unidad aritmeticológica: un banco de registros con 16 entradas y un conjunto de 46 instrucciones. En 1974, Intel fabricó un microprocesador nuevo de propósito general de 8 bits, el 8080, que contenía 4.500 transistores y era capaz de ejecutar un total de 200.000 instrucciones por segundo.

Desde estos primeros procesadores, que solo permitían ejecutar 200.000 instrucciones por segundo, se ha llegado a arquitecturas tipo Sandy Bridge (Intel, Sandy Bride Intel) o AMDfusion (AMD, página AMD Fusion), con los que se pueden llegar a ejecutar unos 2.300 billones de instrucciones por segundo.

Para lograr este incremento en el rendimiento, a grandes rasgos se distinguen tres tipos de mejoras importantes aplicadas a los primeros modelos mencionados anteriormente:

- Técnicas asociadas a explotar el paralelismo de las instrucciones.
- Técnicas asociadas a explotar el paralelismo a nivel de datos.
- Técnicas asociadas a explotar el paralelismo de hilos de ejecución.

El primer conjunto de técnicas consistió en intentar explotar el nivel de paralelismo de las instrucciones (en inglés, *instruction level parallelism*, ILP de ahora en adelante), que componían una aplicación. Algunos de los procesadores que las usaron son VAX 78032, PowerPC 601, los Intel Pentium o bien los AMD K5 o AMDK6. Dentro de este primer grupo de mejoras, se encuentran, entre otros, arquitecturas súper escalares, ejecución fuera de orden de instrucciones, técnicas de predicción, *very long instruction word* (VLIW) o arquitecturas vectoriales. Estas últimas optimizaciones eran bastante interesantes desde el punto de vista de la ILP, puesto que se basaban en la posibilidad que tienen los compiladores de mejorar la planificación de las instrucciones. De este modo el procesador no necesita llevarlas a cabo.

Lecturas complementarias

Sobre *la very long instruction word*, podéis leer:

J. Fisher (1893). "Very Long Instruction Word Architectures and the ELI-512". En: *Proceedings of the 10th Annual International Symposium on Computer Architecture* (pág. 140-150).

Y sobre arquitecturas vectoriales:

R. Baniwal (2010). "Recent Trends in Vector Architecture: Survey". *International Journal of Computer Science & Communication* (vol. 1, núm. 2, pág. 395-339).

El tercer conjunto de técnicas consistió en intentar explotar el nivel de paralelismo a nivel de datos. En estos casos se aplicaban las mismas instrucciones sobre bloques de datos de manera paralela, por ejemplo, la suma o resta de dos vectores de enteros. Este tipo de técnicas permitió aumentar de manera radical la cantidad de operaciones de coma flotante (FLOPS) (en inglés, *floating point*

operations per second) que las arquitecturas podían proporcionar. Sin embargo, como se verá más adelante, este modelo de programación no se puede aplicar de manera genérica a todos los algoritmos de computación.

El tercer conjunto de técnicas intentan explotar el paralelismo asociado a los hilos que componen las aplicaciones. Durante las décadas de 1990-2010, la cantidad de hilos que componen las aplicaciones ha aumentado notoriamente. Se pasó del uso de pocos hilos de ejecución al uso de centenares de hilos por aplicación. Un ejemplo claro de este tipo de aplicación son las empleadas por los servidores, por ejemplo Apache o Tomcat. Sin embargo, también encontramos aplicaciones multihilo en los ordenadores domésticos. Algunos ejemplos de estas aplicaciones son máquinas virtuales como Parallels y VMWare o navegadores como Chrome o Firefox. Para apoyar a este tipo de aplicaciones, el hardware ha ido haciendo la evolución natural causada por esta demanda creciente de paralelismo.

Esta tendencia se hizo patente con los primeros multiprocesadores, procesadores con *simultaneous multithreading*, y se ha confirmado con la aparición de sistemas multinúcleos. En todos los casos, las aplicaciones tienen acceso a dos conjuntos o más de hilos de ejecución disponibles dentro del mismo hardware. En las primeras arquitecturas estos hilos se encontraban repartidos en diferentes procesadores. En estos casos, la aplicación podía ejecutar trabajos paralelos: uno en cada uno de los procesadores. En las segundas arquitecturas, un mismo procesador proporcionaba acceso a diferentes hilos de ejecución. Por lo tanto, en estos escenarios, diferentes hilos de ejecución se pueden ejecutar en un mismo procesador.

En relación con este segundo tipo de arquitecturas, durante la primera década del 2000 aparecieron arquitecturas de computación empleadas en entornos gráficos (GPU) con un nivel de paralelismo muy alto. A pesar de que estas arquitecturas nuevas se orientaban a la renderización gráfica, rápidamente se extendió el uso en el mundo de computación de altas prestaciones.

La evolución de arquitecturas de procesadores ha estado vinculada a la aparición de nuevas técnicas y paradigmas de programación.

En general, las generaciones emergentes han ido acompañadas de complejidades y restricciones nuevas que se han tenido que incorporar al diseño de aplicaciones pensadas para explotar estas nuevas presentaciones. En todos los casos, estas funcionalidades nuevas se han podido usar empleando lenguaje de bajo nivel o máquina. Por ejemplo, la aparición de unidades vectoriales fue acompañada de una

extensión del juego de instrucciones, con instrucciones para especificar cómputos con registros vectoriales.

Sin embargo, explotar las nuevas funcionalidades empleando lenguaje de bajo nivel no es una cosa factible, tanto por complejidad como por productividad. Por eso han ido apareciendo modelos de programación nuevos que han permitido explotarlas a la vez que mitigan sus complejidades. Open-MP, MPI, CUDA, etc. han sido ejemplos de estos modelos.

Por un lado, en este módulo se presentan los conceptos fundamentales de las diferentes arquitecturas de computadores más representativas, así como ejemplos de estas. Por otro lado, se introducen los conceptos más importantes en el estudio de computación de altas prestaciones, así como los paradigmas de programación asociados.

Objetivos

Los objetivos principales que tiene que alcanzar el estudiante al acabar este módulo son los siguientes:

1. Estudiar qué es la taxonomía de Flyint y saber cómo se relaciona esta con las diferentes arquitecturas actuales.

2. Entender qué arquitecturas de computadores hay y cómo se pueden emplear en la computación de altas prestaciones.

3. Estudiar qué es el paralelismo a nivel de datos y saber cómo las aplicaciones pueden incrementar su rendimiento usando este paradigma. Y dentro de este ámbito, entender qué son las arquitecturas vectoriales y cómo funcionan.

4. Entender cuáles son los beneficios de usar arquitecturas que explotan el paralelismo a nivel de hilo.

5. Estudiar qué tipo de arquitecturas multihilo hay y conocer las propiedades de cada una de ellas.

6. Entender cómo se puede mejorar el rendimiento de los procesadores explotando el paralelismo a nivel de hilo y a nivel de instrucción conjuntamente.

7. Estudiar cuáles son los factores relacionados con el modelo de programación que hay que tener en cuenta a la hora de desarrollar aplicaciones multihilo.

8. Estudiar cuáles son los factores relacionados con la arquitectura de un procesador multihilo que hay que considerar en el desarrollo de aplicaciones multihilo.

9. Estudiar cuáles son las características del modelo de programación de memoria compartida y memoria distribuida.

1. Descomposición funcional y de datos

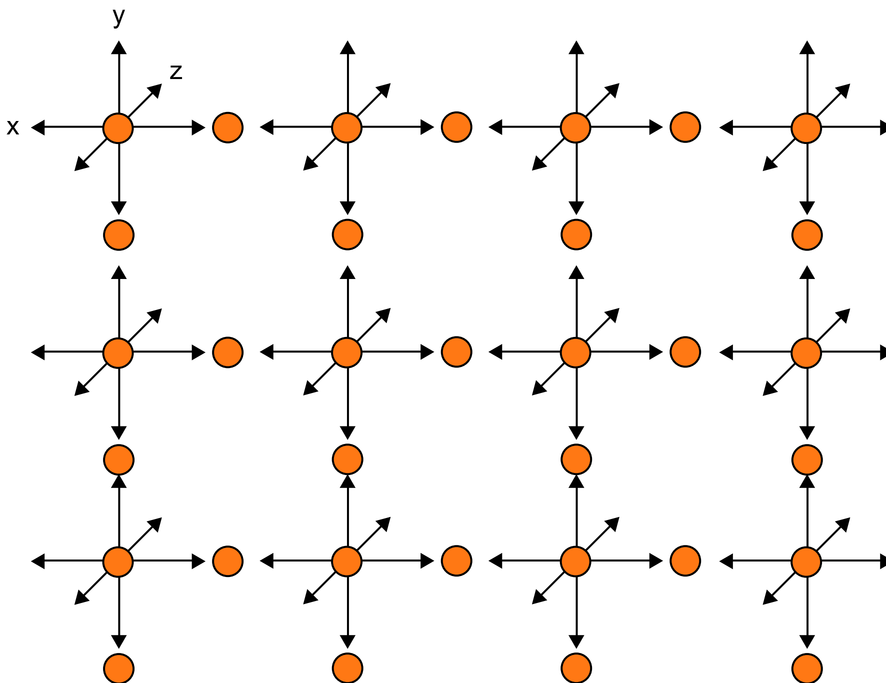
Como se muestra a lo largo de esta unidad didáctica, la mayoría de arquitecturas de altas prestaciones que hoy en día se pueden encontrar en el mercado permiten ejecutar más de un hilo de ejecución de manera paralela. Por otro lado, algunas también proporcionan acceso a unidades de proceso específicas, que son capaces de aplicar un mismo conjunto de operaciones a un conjunto de datos diferentes de manera simultánea.

Sacar rendimiento a todas las funcionalidades diferentes que facilitan estas arquitecturas implica que tanto las aplicaciones como los modelos de programación se tienen que adaptar a las características de estas. Por un lado, los modelos de programación tienen que facilitar a las aplicaciones maneras de explicitar fuentes de paralelismo. Por otro lado, hay que adaptar los algoritmos que implementan las aplicaciones a estas.

Para poder adaptar una aplicación a un modelo de programación orientado a arquitecturas paralelas hay que estudiar sus funcionalidades y saber cómo procesa los datos.

Figura 1. Descomposición de datos

$$(x,y) = F(x - 1, x + 1, z - 1, z + 1, y - 1, y + 1)$$



El ejemplo anterior muestra un tipo de aplicación que procesa los diferentes puntos de una malla según sus vecinos. En este ejemplo se puede considerar que cada uno de los puntos puede ser tratado independientemente. Por lo tanto, a la hora de diseñar una implementación paralela para esta, se podría descomponer cada procesamiento de estos puntos de manera independiente.

El proceso de decidir cómo se procesan los datos de un problema determinado también se puede denominar descomposición de datos. Esta descomposición acostumbra a ser la más compleja a la hora de paralelizar una aplicación, puesto que no solamente se tiene que decidir cómo procesan los datos los diferentes hilos de una aplicación, sino cómo estas se guardan en los diferentes niveles de memoria caché. Dependiendo de dónde se encuentren ubicadas en las diferentes memorias caché y de la manera como los hilos accedan a las mismas, el rendimiento de la aplicación se puede ver sustancialmente menguado.

Descomposición de datos

Algunos ejemplos muy extendidos de estos tipos de descomposición son lo que se denomina partición en bloques para la computación en matrices. El objetivo principal acostumbra a ser tratar de hacer particiones del procesamiento de la matriz de manera independiente por los diferentes hilos del procesador, y tratar de mantener localidad en las memorias caché donde se encuentran.

El segundo de los problemas principales a la hora de paralelizar una aplicación consiste en dividir el problema que se quiere resolver en las funcionalidades en las que el problema puede ser descompuesto. Hay que hacer notar que pueden ser ejecutadas de manera paralela o no.

Lectura recomendada

Se recomienda leer el artículo siguiente, en el que se explican diferentes algoritmos de partición teniendo en cuenta la medida de las memorias caché:

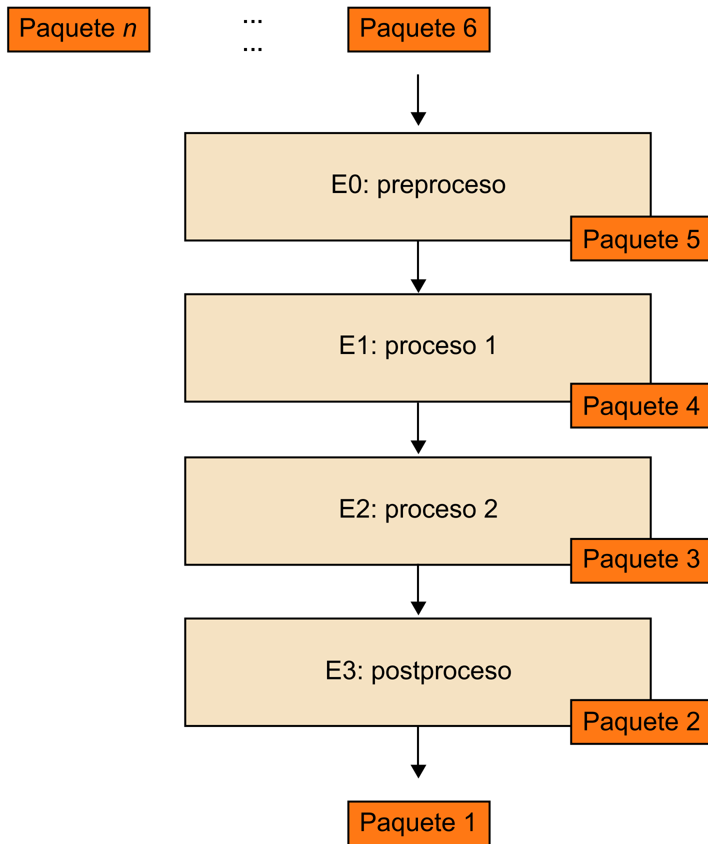
M. S. Lam; E. E. Rothberg; M. E. Wolf (1991).

La figura siguiente ilustra un ejemplo de este tipo de descomposición. Como se puede observar, el problema se ha dividido en cuatro etapas diferentes: una de preproceso, dos de proceso y una de postproceso. Cada elemento que es procesado por el algoritmo en cuestión pasa por cada una de ellas. Sin embargo, en un momento dado podemos tener en cada una de las etapas un paquete diferente. Por lo tanto, cada uno se estará procesando de manera concurrente.

Descomposición funcional del problema

La definición de qué funciones son las que definen el problema que se trata, cómo se encuentran relacionadas y cómo circulan los datos se denomina descomposición funcional del problema.

Figura 2. Descomposición funcional



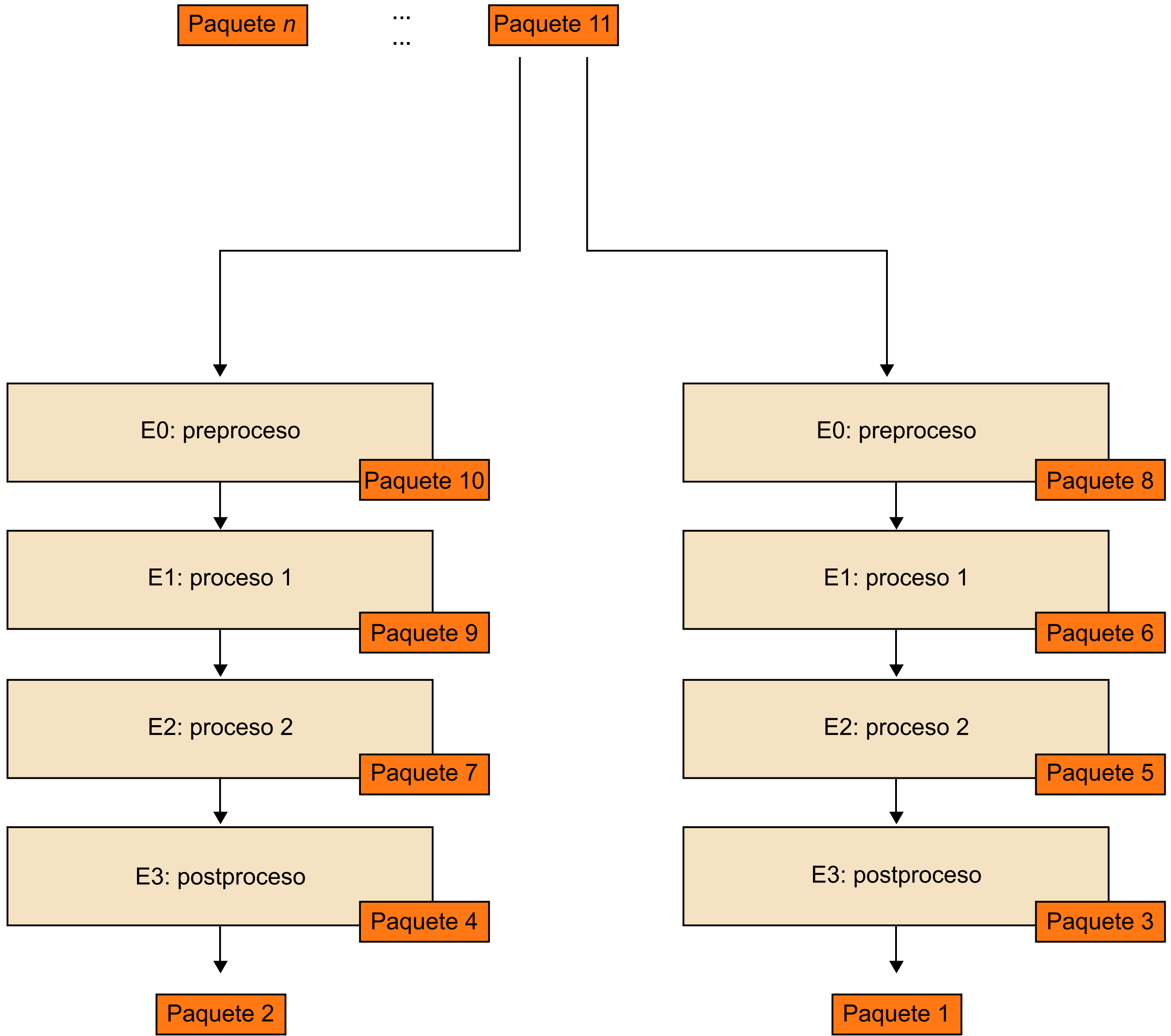
A pesar de que los dos ejemplos anteriores tratan la descomposición de datos y funcional separadamente, en la realidad estos dos problemas están directamente relacionados. A la hora de analizar la paralelización de una aplicación, se estudian tanto la descomposición funcional como la de datos. En general se busca una combinación óptima de las dos.

La figura siguiente muestra un ejemplo en el que se han aplicado los dos tipos de descomposición. Por un lado, la descomposición funcional ha definido las diferentes etapas de nuestro algoritmo y cómo cada una puede ser ejecutada independientemente de la anterior. Por otro lado, se ha definido una descomposición de datos en la que cada uno de los paquetes puede ser procesado de manera independiente del resto.

Ved también

En los apartados siguientes se explican arquitecturas de altas prestaciones, en las cuales se pueden ejecutar aplicaciones que tienen paralelismo a nivel de hilo o de datos. En el supuesto de que se quiera sacar rendimiento de una aplicación determinada en una arquitectura determinada, habrá que estudiar cómo adaptar el tratamiento funcional y de datos al sistema donde se ejecutará.

Figura 3. Paralelismo a nivel de datos y funcionalidades



2. Taxonomía de Flynn

Desde los primeros procesadores hasta las arquitecturas actuales se han propuesto muchos tipos de arquitecturas de computadores diferentes. Desde procesadores con un solo hilo de ejecución con un *pipeline* en orden, hasta procesadores con centenares de núcleos y unidades de computación que explotan el paralelismo a nivel de datos.

En 1972, Michael J. Flynn propuso una clasificación de las diferentes arquitecturas de computadores en cuatro categorías. Esta categorización clasifica los computadores según cómo gestionan los flujos de datos e instrucciones. Es muy interesante considerando la época en la que se hizo y cómo se puede aplicar en las arquitecturas de hoy en día. A pesar de que fue definida hace treinta años, la mayoría de los procesadores actuales se pueden incluir en alguna de estas cuatro categorías. Además, muchos de los trabajos académicos han seguido utilizando esta nomenclatura hasta el día de hoy. Los cuatro tipos de arquitecturas que Flynn describió son las siguientes:

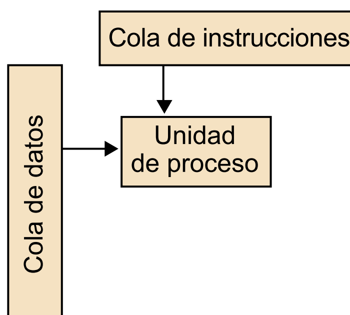
1) Una instrucción, un dato[1]**:** son las arquitecturas tradicionales formadas por una sola unidad de proceso. Como se puede observar a continuación, no explotan ni el paralelismo a nivel de hilo, ni el paralelismo a nivel de datos. En este caso, tenemos un solo flujo de datos y de instrucciones. La figura siguiente ilustra el modelo abstracto de un computador *SISD*.

(1)En inglés, *single instruction, single data stream* (*SISD*).

Computador *SISD*

Ejemplos de este tipo de procesador son IBM 370 (IBM, System/370 Modelo 145), Intel 8085 (Intel, 2011) o el Motorola 6809 (Hennessy y Patterson, 2011).

Figura 4. Una instrucción, un dato



2) Una instrucción, múltiples datos[2]**:** son las arquitecturas en las que una misma instrucción se ejecuta en múltiples unidades de proceso de manera paralela, usando diferentes flujos (*streams*) de datos. Cada procesador tiene su propia memoria con datos; por lo tanto, tenemos diferentes datos en global, pero la memoria de instrucciones y unidad de control son compartidas. Las instancias de este tipo de arquitecturas más famosas son las unidades de proceso vectoriales. Este tipo de arquitecturas son empleadas hoy en día en la mayoría de procesadores de altas prestaciones, puesto que permiten hacer cálculos sobre grandes volúmenes de datos de manera paralela. Así, es posible in-

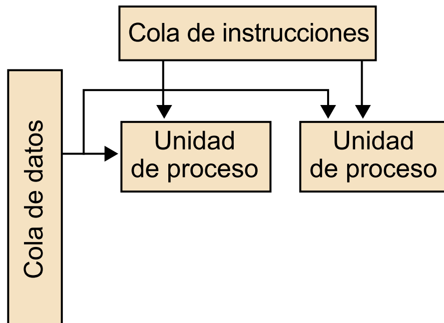
(2)En inglés, *single instruction, multiple data stream* (*SIMD*).

Unidades de proceso vectoriales

Los procesadores de la empresa Cray (insideHPC, *InsideHPC: A Visual History of Cray*) son ejemplos de este tipo de arquitecturas.

crementar notoriamente los flops de un procesador. Por otro lado, como veremos más adelante, este paradigma de computación también se emplea dentro de las arquitecturas dedicadas a procesamiento de imagen o de gráficos.

Figura 5. Una instrucción, múltiples datos



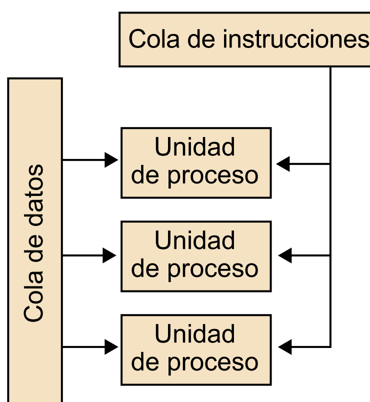
3) Múltiples instrucciones, un dato[3]**:** permiten ejecutar diferentes flujos de instrucciones sobre un mismo bloque de datos. En este caso, tenemos diferentes unidades de proceso que operan sobre el mismo dato. Este tipo de arquitecturas es muy específico y no se encuentra ninguno dentro del mundo de la arquitectura de procesadores de altas prestaciones o de propósito general. Sin embargo, se han hecho implementaciones de multiprocesadores de propósito específico que siguen esta definición.

(3)En inglés, *multiple instruction, single data stream* (*MISD*).

Multiprocesadores de propósito específico

Un ejemplo de esto es un procesador en el que las unidades de proceso se encuentran replicadas para tener redundancia. Este tipo de arquitecturas son especialmente útiles en entornos en los que la fiabilidad es el factor más importante, como por ejemplo, la aviación. Otro ejemplo sería el que se denomina *pipeline image processing*: cada punto de la imagen es procesado por diferentes unidades de proceso, donde cada una aplica un tipo de transformación diferente sobre este.

Figura 6. Múltiples instrucciones, un dato

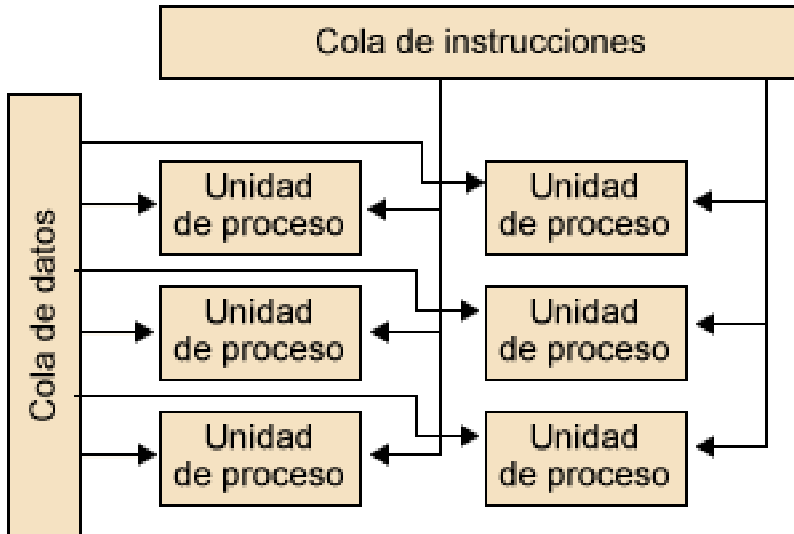


4) Múltiples instrucciones, múltiples datos[4]**:** están formadas por múltiples unidades de proceso que trabajan sobre diferentes flujos de datos de manera paralela. Como muestra la figura siguiente, la arquitectura está formada por diferentes unidades de proceso que tienen diferentes flujos de instrucciones que

(4)En inglés, *multiple instruction, multiple data stream* (*MIMD*).

trabajan con diferentes flujos de datos. Este es el claro paradigma de computación actual en el que las arquitecturas permiten procesar diferentes flujos de datos con diferentes flujos de computación.

Figura 7. Múltiples instrucciones, múltiples datos



Las arquitecturas *MIMD* facilitan el acceso a múltiples hilos de computación y múltiples flujos de datos. Los flujos de computación se pueden encontrar completamente aislados entre sí o bien compartir recursos y contextos. En el primero de los casos estaremos hablando de diferentes procesos, en los que cada uno de los procesos contiene un contexto de ejecución completamente aislado del resto. En el segundo caso, estaremos hablando de hilos de ejecución o *threads*, en los que diferentes hilos de ejecución comparten un mismo contexto. Es decir, comparten los datos y las instrucciones.

Es importante hacer notar que algunas de las arquitecturas que se pueden encontrar en la actualidad siguen más de uno de los paradigmas anteriores. Por ejemplo, se pueden encontrar arquitecturas multinúcleo que contienen unidades vectoriales para hacer cómputos con vector de manera paralela. Por lo tanto, en un mismo procesador encontraremos partes *SIMD* y *MIMD*. Las arquitecturas que se pueden encontrar dentro del mundo de computación de altas prestaciones son básicamente arquitecturas *SIMD* y *MIMD*.

Ved también

Los próximos apartados introducen los conceptos más importantes de las arquitecturas más frecuentes en las arquitecturas de altas prestaciones: las *SIMD* y las *MIMD*. Para cada una, se discute un ejemplo de procesador real para poder estudiar arquitecturas reales.

3. Arquitecturas de procesador *SIMD*

Las arquitecturas *SIMD*, como ya hemos dicho, se caracterizan porque procesan diferentes flujos de datos con un solo flujo de instrucciones. La clase de arquitecturas más conocidas de esta familia son las arquitecturas vectoriales.

La mayoría de procesadores de altas prestaciones tienen unidades específicas de tipo vectorial. Como explica esta sección, estas clases de arquitecturas tienen ciertas propiedades que las hacen muy interesantes para llevar a cabo cálculos científicos y procesar grandes volúmenes de datos.

Ejemplo de suma vectorial

```
Parámetros:
VectorEnteros[1024] A;
VectorEnteros[1024] B;
VectorEnteros[1024] C;

Código:
por(i = 0; i < 128; i++)
    C[i] = A[i]+B[i];

Parámetros:
VectorEnteros[1024] A;
VectorEnteros[1024] B;
VectorEnteros[1024] C;

Código:
RegistroVectorial512 a,b,c;

Por(i=0; i<512; i+=16)
{
    carga(a,A[i]); carga(a,B[i]);
    c = suma_vectorial(a,b);
    guarda(C[i],a);
}
```

Como indica el mismo nombre, a diferencia de las *SISD*, este tipo de arquitecturas operan a nivel de vectores de datos. Por lo tanto, con una sola instrucción podemos hacer la suma de dos registros que son capaces de guardar *k* elementos diferentes.

Por ejemplo, una unidad vectorial podría trabajar con registros de 512 bytes. En este caso, un vector podría guardar 16 elementos de tipo *float* (asumiendo que un *float* ocupa 32 bytes) y, por lo tanto, con una sola instrucción podría sumar dos bloques de 16 elementos.

El ejemplo de código anterior muestra cómo se podría vectorizar la suma de dos vectores A y B que se guarda en un tercer vector C (cada vector es de 1.024 elementos enteros). Este ejemplo es sencillo, pero suficiente para dar una idea del incremento de rendimiento que las aplicaciones pueden obtener si pueden usar correctamente las unidades vectoriales que el procesador contiene.

El primero de los códigos muestra la clásica suma de vectores tal como la haría un procesador escalar normal. En este caso, la aplicación hará 1.024 iteraciones. En cada iteración, el procesador leerá el valor entero de la posición que se está procesando de los vectores a, b y c, sumará a y b, y finalmente lo escribirá en C. El segundo de los códigos muestra cómo se puede calcular la misma suma de vectores si disponemos de una unidad vectorial con registros de 512 bytes. En este caso, como trabajamos con elementos enteros de 32 bytes, dentro de cada registro vectorial podemos guardar 16 elementos. De este modo, en cada iteración se leen 16 elementos del vector A y B, se guardan en los dos registros vectoriales a y b, se suman al registro vectorial c, y finalmente se escriben los 512 bytes del vector en la posición correspondiente del vector C.

Como se puede observar, en el código vectorial el incremento del índice es de 16 en 16. Por lo tanto, por cada iteración del código vectorial se tienen que hacer 16 del código escalar. A pesar de que el código es sencillo, da una idea del potencial de este tipo de arquitecturas para procesar estructuras de tipos vectores o matrices y de la mejora que pueden ofrecer respecto de unidades escalares tradicionales.

Hay que hacer notar que esto no es tan solo un mecanismo software. Es decir, el sistema y las bibliotecas proporcionan un conjunto de interfaces que permiten operar con los bloques de 512 bytes, pero estas llamadas se acaban transformando al lenguaje nativo del procesador, que es capaz de operar con estos bloques. Cada procesador vectorial facilita un conjunto de instrucciones específicas[5] a este para operar con los datos de tipo vectorial. Por lo tanto, el tipo de operaciones vectoriales que se podrá emplear en una arquitectura concreta dependerá de la *ISA* vectorial que esta proporcione.

(5) En inglés, *instruction set architecture* (*ISA*).

En el ejemplo anterior, si la *ISA* del procesador vectorial no facilita una suma de dos registros vectoriales, el compilador probablemente traducirá la suma de dos registros a 16 sumas escalares consecutivas. Sin embargo, en caso afirmativo, el compilador traducirá la suma de los dos registros en una sola instrucción ensamblador, que calculará la suma de los dos elementos.

El ejemplo anterior muestra la vectorización de un código extremadamente sencillo. Pero el proceso de adaptar el código de la aplicación para que esta trabaje con bloques de 512 bytes puede no ser factible en todas las aplicaciones. Por ejemplo, esto puede ser difícil o imposible en algoritmos que trabajan en estructuras de datos representados en grafos. Por otro lado, en aplicaciones científicas que trabajan con datos matriciales, este tipo de procesamiento es muy habitual. Sin embargo, en muchos casos vectorizar el código es altamente complejo o imposible.

Lectura recomendada

Este subapartado no explica en detalle cómo vectorizar aplicaciones ni las arquitecturas vectoriales. Si queréis profundizar en este ámbito, os recomendamos leer el libro siguiente:

G. Sabot (1995). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, Massachusetts: Addison-Wesley.

3.1. Propiedades de los procesadores vectoriales

Como acabamos de ver, los procesadores o unidades vectoriales pueden mejorar sustancialmente el rendimiento de las aplicaciones cuando pueden ser vectorizadas. Sin embargo, el beneficio de hacer operaciones vectoriales no solo radica en poder hacer *m* sumas o restas paralelas. El hecho de poder operar un conjunto de datos en bloque otorga a los procesadores vectoriales ciertas características interesantes:

a) Trabajar con bloques de datos hace que el compilador o el programador especifique al procesador que no hay dependencias entre los diferentes elementos que componen los vectores. Por lo tanto, el procesador no tiene que hacer validaciones sobre riesgos estructurales o datos entre los diferentes elementos de los vectores. Si es un procesador escalar normal, el procesador tendría que verificar que no hay riesgos entre las diferentes operaciones de los elementos de los vectores. Sin embargo, el procesador tiene que computar y validar dependencias solo entre las operaciones de los registros vectoriales.

b) Para llevar a cabo el cálculo paralelo de los diferentes elementos de los registros vectoriales, el procesador puede emplear unidades funcionales replicadas de manera paralela para cada uno de los elementos de los registros.

Suma de registros

Para sumar dos registros de 512 bytes que contienen 16 elementos enteros, por ejemplo, el procesador puede tener 16 unidades de suma que pueden computar en paralelo la suma de cada una de las posiciones.

c) En general, los procesadores facilitan un conjunto de instrucciones vectoriales bastante extenso. En este subapartado solo se han mencionado las operaciones de memoria y aritméticas típicas (*add*, *sub*, *load*, *store*, etc.). Pero el juego de instrucciones que acostumbran a proporcionar es bastante más completo y complejo en algunos casos.

Juegos de instrucciones

Un ejemplo de ello son los *broadcasts* o los *shuffles*. Algunos de los juegos de instrucciones más conocidos son las *advanced vector extensions* (AVX; I. Corp, 2011) o las *streaming SIMD extensions* (SSE; I. Corp, 2007).

Todos los puntos discutidos hacen que las arquitecturas vectoriales sean muy atractivas para aplicaciones científicas o aplicaciones de ingeniería. Algunas de las aplicaciones que hacen un buen uso de este tipo de prestaciones son las simulaciones de choques de coches o las simulaciones de tiempos. Ambos tipos de aplicaciones emplean grandes volúmenes de datos y pueden ejecutarse en supercomputadores durante periodos de tiempo largos. Otro tipo de aplicaciones que se benefician mucho de esta clase de arquitecturas son las aplicaciones multimedia, que se caracterizan por un nivel abundante de paralelismo a nivel de datos.

3.2. Ejemplo de procesador vectorial

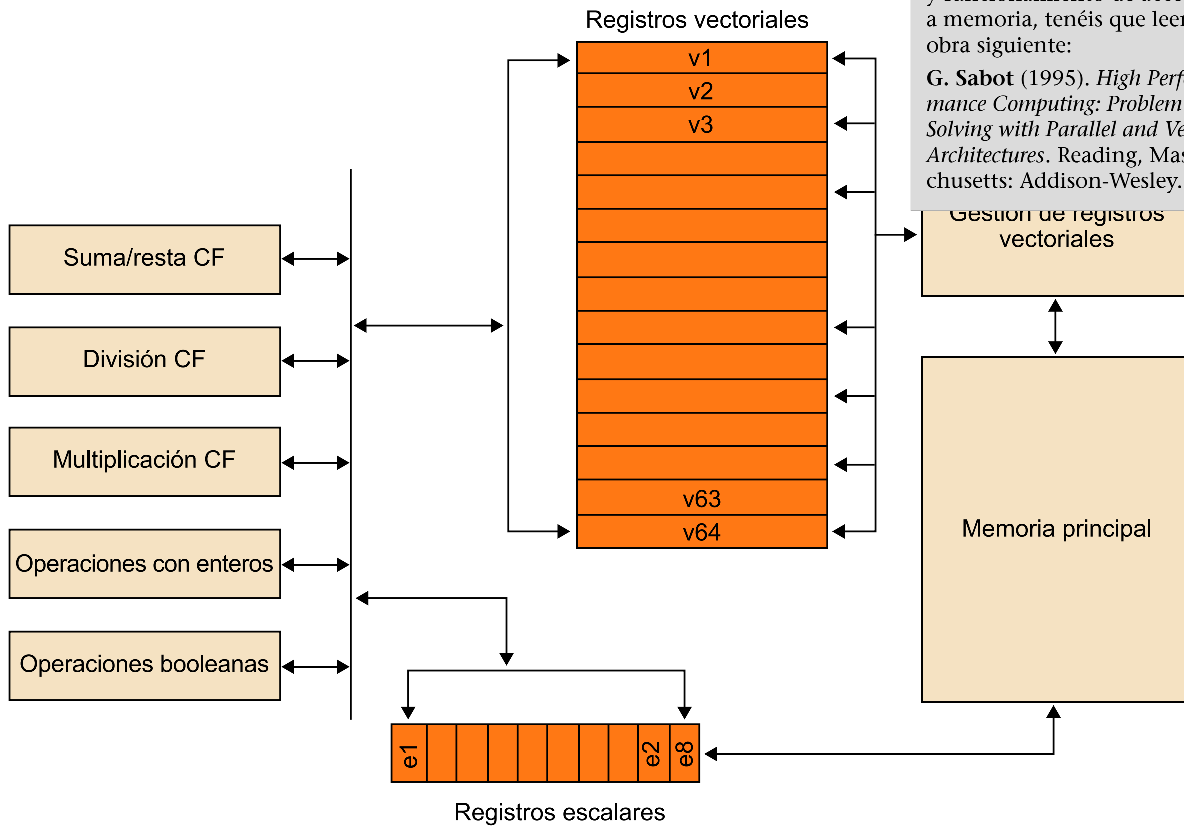
La figura siguiente muestra un ejemplo de posible procesador vectorial. Este subapartado presenta a alto nivel los diferentes elementos que podrían componer un procesador de este tipo.

Lectura recomendada

Recordad que si queréis profundizar más en el diseño y la implementación de las diferentes etapas que lo componen, tipo de operaciones y funcionamiento de acceso a memoria, tenéis que leer la obra siguiente:

G. Sabot (1995). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, Massachusetts: Addison-Wesley.

Figura 8. Ejemplo de procesador vectorial



Como se puede observar, está compuesto por dos tipos de banco de registros diferentes. El primer bloque son los registros. Por un lado se encuentran los vectoriales, que pueden guardar 512 bytes de información. Cada uno guarda lo que hemos denominado como vector, es decir, un conjunto de elementos de coma flotante o enteros. En el supuesto de que se quieran guardar enteros, un registro puede guardar un total de 16 elementos (32 bytes por elemento). Si se quieren guardar elementos de coma flotante, se pueden guardar un total de 8 elementos (64 bytes por elemento). Por otro lado, el procesador también dispone de un conjunto de registros escalares. Algunas de las instrucciones vectoriales pueden usar escalares como parte de la operación o pueden generar resultados escalares.

El segundo bloque lo forman las unidades funcionales, que se dividen en tres grandes bloques. En primer lugar están las unidades funcionales dedicadas a llevar a cabo el cómputo sobre vectores que contienen elementos de tipo coma flotante o que se quieren operar como coma flotante. El segundo conjunto in-

cluye la unidad funcional dedicada a hacer el cálculo sobre registros vectoriales que guardan datos enteros. Y finalmente, está la unidad funcional, que se encarga de llevar a cabo operaciones booleanas sobre los registros vectoriales.

El tercero y último bloque incluye los elementos orientados a gestionar el acceso a memoria (ya sea L1/L2 o memoria principal) y la transferencia de datos de esta a los registros vectoriales. En el caso de los registros vectoriales, como se trabajan en bloques de 512 bytes, se dispone de una unidad específica que se encarga de gestionar el acceso a memoria. Los registros escalares funcionan tal como funcionarían en un procesador escalar normal. Por lo tanto, no disponemos de una máquina específica para llevar a cabo esta carga.

El procesador vectorial introducido en este subapartado es un modelo básico de arquitectura vectorial. En cuanto a procesadores vectoriales, se pueden encontrar muchos durante las últimas décadas.

Procesadores Cray y Sandy Bridge

Un ejemplo importante de procesadores vectoriales son los Cray (insideHPC, *InsideHPC: A Visual History of Cray*), que sobre todo fueron importantes durante la década de los ochenta y noventa y se diseñaron hasta seis modelos diferentes. Durante la primera década del 2000, Cray anunció diferentes sistemas, que a pesar de que no eran exclusivamente vectoriales tenían una parte importante del hardware orientada a este tipo de procesamiento.

El primero de todos fue el Cray-1, que se presentó el año 1976. Se ejecutaba a una frecuencia de reloj de 80 MHz y disponía de 8 registros de tipo vectorial. Cada uno de estos registros estaba compuesto por 64 elementos de 64 bits (8 bytes). Para hacer operaciones vectoriales, disponía de seis unidades de computación vectoriales diferentes: coma flotante de suma, coma flotante de multiplicación, unidad para desplazar elementos del vector, una unidad entera de suma, una unidad para llevar a cabo operaciones lógicas y finalmente, una unidad dedicada a calcular unas operaciones específicas llamadas recíprocas.

Como ya se ha mencionado, muchos de los procesadores actuales, a pesar de no ser procesadores totalmente orientados al procesamiento vectorial, llevan unidades vectoriales. El procesador de la empresa Intel, Sandy Bridge (Intel, 2012), es un ejemplo de ello. Dentro de su juego de instrucciones, el Sandy Bridge incorpora el juego de instrucciones ya mencionado AVX, aparte de las SSE anteriores. Sin embargo, los procesadores Intel no son los únicos que incorporan instrucciones vectoriales dentro del juego de instrucciones. Los procesadores de la empresa AMD también lo hacen.

4. Arquitecturas de procesador multihilo o MIMD

Durante las primeras décadas de desarrollo de microprocesadores (1970 y 1980), el objetivo principal se centró en explotar al máximo el paralelismo de las aplicaciones a nivel de instrucción (*SISD*), denominado también *instruction level paralellism* (ILP). Durante estas décadas se intentó sacar rendimiento a las aplicaciones con técnicas que permitían ejecutar las instrucciones fuera de orden, técnicas de predicción más precisas o jerarquías de memoria de mayor capacidad.

Sin embargo, el rendimiento de las aplicaciones no escala proporcionalmente con la cantidad de recursos añadidos.

> Por ejemplo, pasar de una arquitectura que permite empezar dos instrucciones por ciclo a cuatro por ciclo no implica necesariamente doblar el rendimiento del procesador. Incluso, en muchos casos, aunque se añadan muchos más recursos, el rendimiento de la aplicación solo se incrementa ligeramente.

Este factor es el que motivó la aparición de arquitecturas MIMD, que llevan a cabo la ejecución simultánea de diferentes hilos de ejecución en un mismo procesador: paralelismo a nivel de hilo[6] (Lo, 1997). En estas arquitecturas:

(6)En inglés, *thread level paralelism* (*TLP*).

- Diferentes hilos de ejecución comparten las unidades funcionales del procesador (por ejemplo, unidades funcionales).
- El procesador tiene que tener estructuras independientes para cada uno de los hilos que ejecuta: registro de *renaming*, contador de programa, etc.
- Si los hilos pertenecen a diferentes procesos, el procesador tiene que facilitar mecanismos para que puedan trabajar con diferentes tablas de páginas.

Como se muestra a continuación, las arquitecturas actuales explotan este paradigma de muchas maneras diferentes. Sin embargo, la motivación es la misma: la mayoría de aplicaciones actuales tienen un alto nivel de paralelismo y su rendimiento aumenta, añadiendo más paralelismo a nivel de procesador. El objetivo es mejorar la productividad de los sistemas que pueden ejecutar aplicaciones que son inherentemente paralelas.

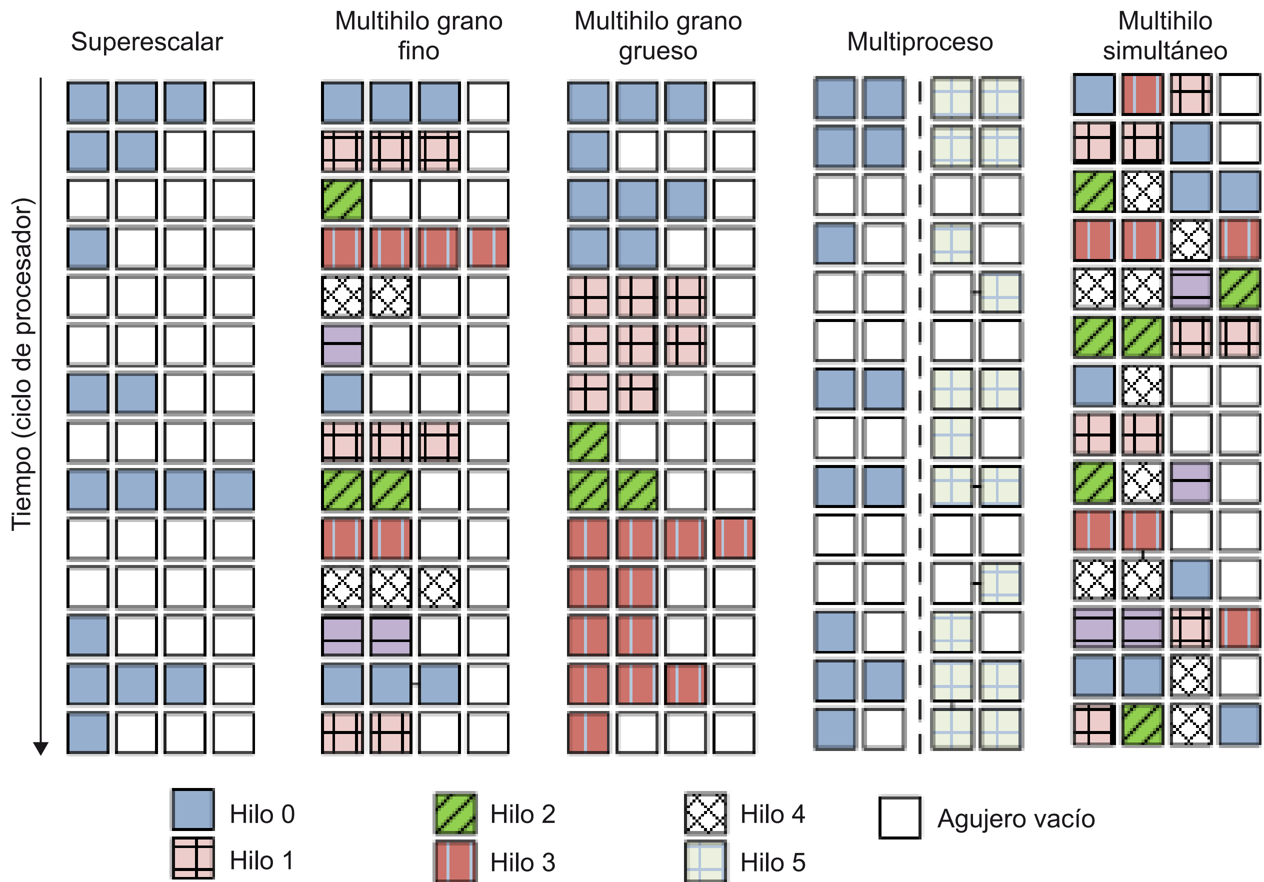
Aplicaciones paralelas

Los navegadores, las bases de datos, los servidores web, etc. son ejemplos de este tipo de aplicaciones.

En estos entornos actuales, sacar rendimiento explotando el TLP puede ser mucho más efectivo en términos de coste/rendimiento que explotar el ILP. En la mayoría de casos, cuanto más paralelismo se facilite, más rendimiento se puede sacar. En cualquier caso, la mayoría de técnicas que se habían empleado para sistemas ILP también se usan en sistemas TLP.

A continuación se presentan los diferentes tipos de arquitecturas TLP que se han propuesto durante las últimas décadas. La figura 9 muestra cómo se ejecutarían diferentes hilos de ejecución en cada una de las arquitecturas introducidas a continuación.

Figura 9. Modelo ILP frente a modelos TLP



1) Arquitecturas multiprocesador (MP). Esta es la extensión más sencilla a un modelo ILP. En este caso replicamos una arquitectura ILP *n* veces. El modelo más básico de estas arquitecturas es el multiprocesador simétrico. La desventaja principal es que, a pesar de que tiene muchos hilos de ejecución, cada uno sigue teniendo las limitaciones de un ILP. Sin embargo, como se muestra más adelante, hay variantes en las que cada procesador es a la vez multihilo.

2) Arquitecturas multihilo, también conocidas como *superthreading*. Esta fue la siguiente de las variantes TLP que apareció. En el modelo de *pipeline* del procesador se extiende considerando también el concepto de *hilo de ejecución*. En este caso, el planificador (que escoge cuál de las instrucciones empieza en este ciclo) tiene la posibilidad de elegir cuál de los hilos de ejecución empieza la instrucción siguiente en el ciclo siguiente.

TERA Systems

Por ejemplo, TERA Systems (Alverson, Callahan, Cummings y Koblenz, 1990) podía trabajar con 128 hilos a la vez.

3) Arquitecturas con ejecución de hilo simultánea[7]**.** Son una variación de las arquitecturas multihilo, que permiten a la lógica de planificación escoger instrucciones de cualquier hilo en cada ciclo de reloj. Esta condición hace que

(7)En inglés, *simultaneous multithreading* (*SMT*).

la utilización de los recursos sea mucho más elevada y eficiente. La desventaja más grande de este tipo de arquitecturas es la complejidad de la lógica necesaria para llevar a cabo esta gestión. El hecho de poder empezar varias instrucciones de diferentes hilos es muy costoso. Por eso el número de hilos que estas arquitecturas acostumbran a usar es relativamente bajo.

Hyperthreading

Por ejemplo, las arquitecturas Intel con *hyperthreading* (Marr, 2002) implementan dos o más hilos de ejecución, o bien la Alpha 21464 (Seznec, Felix, Krishnan y Sazeide, 2002) implementa cuatro hilos de ejecución.

4) ***Arquitecturas multicore***[8]. Conceptualmente, son similares a las primeras arquitecturas mencionadas, pero a escala más pequeña. En el caso de sistemas MP hay *n* procesadores independientes que pueden compartir la memoria, pero no comparten recursos entre sí, como por ejemplo una memoria caché de último nivel. Los SOC son una evolución conceptual de los MP, pero trasladada a nivel de procesador. En un SOC, un mismo procesador se compone por *m* núcleos en los que se pueden ejecutar hilos que pueden estar relacionados o incluso pueden compartir recursos.

(8) Llamadas *chip-multiprocessor* (*CMP*) o *system-on-chip* (*SOC*).

Los próximos subapartados estudian cada una de estas arquitecturas.

4.1. Arquitecturas superthreading

El TLP saca rendimiento porque comparte los recursos entre diferentes hilos de ejecución. Sin embargo, hay dos maneras de llevar a cabo esta compartición. La primera consiste a compartir los recursos en el espacio y el tiempo, es decir, en un ciclo concreto y en una etapa concreta del procesador, instrucciones de hilos diferentes pueden estar compartiendo las mismas etapas del procesador. La segunda consiste a compartir los recursos en el tiempo; es decir, en un ciclo concreto y en una etapa concreta del procesador, solo se pueden encontrar instrucciones de un mismo hilo. Este segundo tipo de compartición es conocida como *superthreading*.

Dentro de las arquitecturas *superthreading* hay dos maneras de compartir los recursos en el tiempo entre los diferentes hilos. Las más habituales son compartición a nivel fino o compartición a nivel grueso.

4.1.1. Compartición a nivel fino

En la compartición de recursos a nivel fino, el procesador cambia de hilo en cada instrucción que este ejecuta. Así, la ejecución de los hilos se hace de manera intercalada. Habitualmente, el cambio de hilo se hace siguiendo una distribución *round robin*, es decir, se ejecuta una instrucción del hilo *n*, después del $n + 1$, del $n + 2$, etc. En los casos en los que un hilo está bloqueado, por ejemplo, esperando datos de memoria, se salta y se pasa al siguiente. Para poder llevar a cabo este tipo de cambios, el procesador tiene que ser capaz de hacer un cambio de hilo por ciclo.

La ventaja de esta opción es que puede amortiguar bloqueos cortos y largos de los hilos que se están ejecutando, ya que en cada ciclo se cambia de hilo. La desventaja principal es que se reduce el rendimiento de los hilos de manera

individual, puesto que hilos preparados para ejecutar instrucciones quedan retrasados por las instrucciones de los otros hilos. Esto tiene implicaciones de complejidad de diseño y de consumo de energía, ya que el procesador tiene que poder cambiar de hilo.

UltraSPARCT1

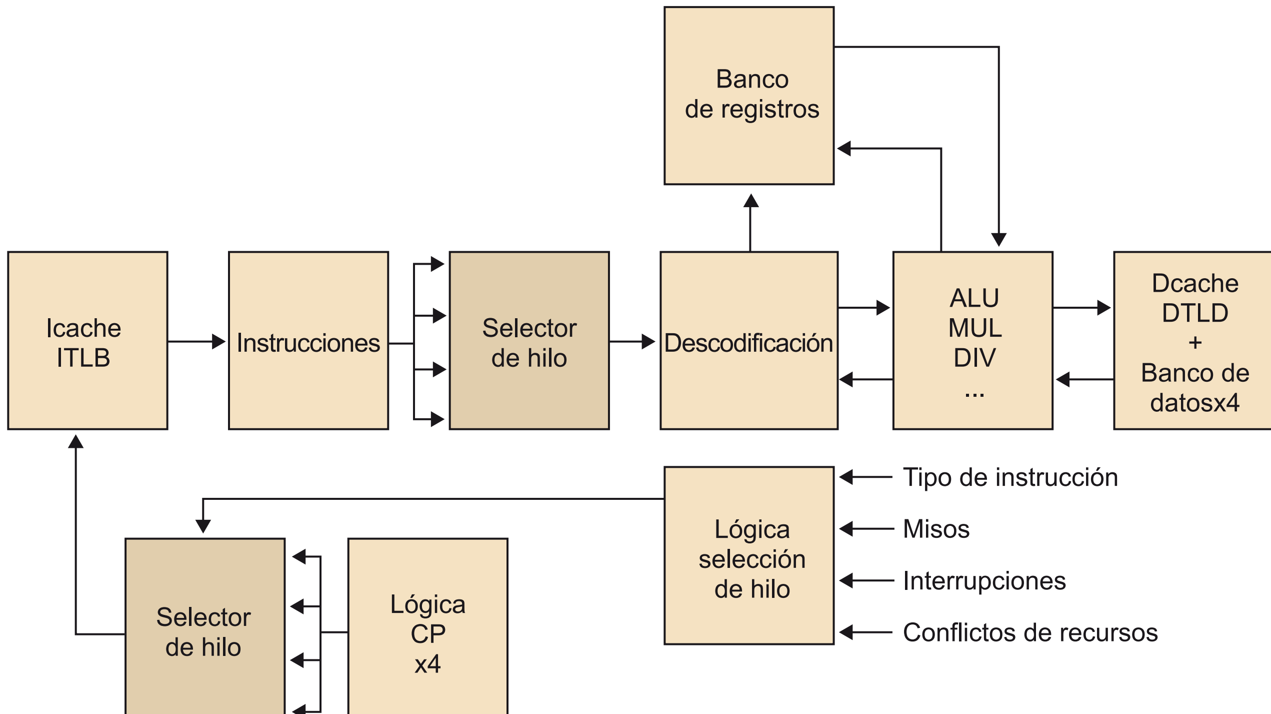
El UltraSPARCT1 (Kongetira, Aingaran y Olukotun, 2005) es un ejemplo de este tipo de procesador.

El procesador UltraSPARCT1, también conocido como *Niagara*, fue anunciado por la empresa SunMicrosystems en noviembre del 2005. Este nuevo procesador UltraSPARC era multihilo y multinúcleo, diseñado por arquitecturas de tipo servidor con un consumo energético bajo. Por ejemplo, a 1,4 GHz consume aproximadamente unos 72 W. Anteriormente, la empresa Sun ya había introducido dos arquitecturas multinúcleo: el UltraSPARC IV y el IV+. Sin embargo, el T1 fue el primer microprocesador que era multinúcleo y multihilo. Se pueden encontrar versiones con cuatro, seis u ocho núcleos, y cada uno puede encargarse de cuatro hilos concurrentemente. Esto implica que se pueden ejecutar hasta un máximo de 32 hilos concurrentemente.

La figura 10 muestra el *pipeline* que tiene cada uno de estos núcleos. Como se puede observar, contiene un selector que decide qué hilo se ejecuta en cada ciclo. Este selector va cambiando de hilo en cada ciclo. En cualquier caso, solo escoge sobre el conjunto de hilos disponibles en cada momento. Cuando hay un acontecimiento de larga latencia (como por ejemplo, un fallo de memoria caché que desencadena una petición a memoria), el hilo que lo ha causado se saca de la cadena de rotación[9]. Una vez este acontecimiento de larga duración acaba, el hilo se vuelve a añadir a la rotación para acceder a las unidades funcionales. El hecho de que el *pipeline* se comparta con diferentes hilos en cada ciclo hace que cada hilo vaya más lento, pero la utilización del procesador es más elevada. Otro efecto muy interesante es que el impacto de fallos de las memorias caché es mucho más reducido: si uno de los hilos causa un fallo, los otros pueden seguir usando los recursos y progresar.

(9) En inglés, *round robin*.

Figura 10. *Pipeline* UltraSPART T1



Como muestra la figura anterior, dado un ciclo, todos los elementos del *pipeline* son usados por solo un hilo. Sin embargo, algunas de las estructuras están replicadas por el número de hilos que tiene el núcleo, como por ejemplo, la lógica de gestión del contador de programa (CP). En este caso, el núcleo tiene que ser capaz de distinguir en qué parte del flujo se encuentra cada hilo. Por lo tanto, necesita tener esta estructura para cada uno de los hilos. Contrariamente, el resto de lógica, como por ejemplo la lógica de descodificación, es única y se usa solo por un hilo dado un ciclo.

4.1.2. Compartición a nivel grueso

En la compartición a nivel grueso los cambios de hilos se hacen cuando en el hilo que se ejecuta hay un bloqueo de larga duración.

Bloqueo de larga duración

Un ejemplo de este tipo de bloqueo es un fallo en la memoria caché de último nivel. En este caso habría que ir a memoria, hecho que implicaría muchos ciclos de bloqueo.

La ventaja de esta opción es que no se reduce el rendimiento del hilo individual, porque solo cambia de hilo cuando este se bloquea por un acontecimiento que requiere una latencia larga para ser procesado. La desventaja es que no es capaz de sacar rendimiento cuando los bloqueos son más cortos. Como el núcleo solo genera instrucciones de un solo hilo en cada ciclo, cuando este se bloquea en un ciclo concreto (por ejemplo, por una dependencia entre instrucciones), todas las etapas siguientes del procesador se quedan bloqueadas o congeladas y se vuelven a emplear después de que el hilo pueda volver a procesar instrucciones.

Otra desventaja importante es que los hilos nuevos que empiezan en el procesador tienen que pasar por todas las etapas antes de que el procesador empiece a retirar instrucciones por este hilo. En el modelo anterior, como en cada ciclo se cambia de hilo, el rendimiento de la productividad no se ve tan afectado. Por ejemplo, supongamos un procesador con 12 ciclos de etapas y 4 hilos de ejecución. En el mejor de los casos, un hilo nuevo tardará 12 ciclos en retirar la primera instrucción. Pero durante estos 12 ciclos, en el mejor de los casos, los otros 3 hilos han podido retirar 12 instrucciones. En cambio, en el grano grueso habríamos estado 12 ciclos sin retirar ninguno.

IBM AS/400

El IBM AS/400 (IBM, 2011) es un ejemplo de este tipo de procesador.

4.2. Arquitecturas simultaneous multithreading

Las arquitecturas con multihilo simultáneo[10] (Tullsen, Eggers y Levy, 1995) son una variación de las arquitecturas tradicionales multihilo. Estas arquitecturas nuevas permiten escoger instrucciones de cualquiera de los hilos que el procesador está ejecutando en cada ciclo de reloj. Esto quiere decir que diferentes hilos están compartiendo en un mismo instante de tiempo diferentes recursos del procesador. Esta compartición es tanto horizontal (por ejemplo, etapas de sucesivas del *pipeline*) como vertical (por ejemplo, recursos de una misma etapa del procesador).

(10)En inglés, *simultaneous multithreading* (*SMT*).

El resultado de estas técnicas es una alta utilización de los recursos del procesador y mucha más eficiencia. Hay que recordar que, a diferencia de las arquitecturas paralelas anteriores, en un mismo instante de tiempo dos hilos pueden estar compartiendo los mismos recursos de una de las etapas del procesador. Sin embargo, esta eficiencia no es gratuita, es decir, para apoyar esta compartición, el procesador contiene una lógica extra y muy compleja. Hay que hacer notar que, por defecto, los hilos de diferentes procesos no se tienen que poder ver entre sí, tanto por temas de seguridad como de funcionalidad, la ejecución de una instrucción A de un hilo X no puede modificar el comportamiento funcional de un hilo B. Hay que recordar que, en un sistema operativo, cada proceso tiene su propio contexto y que este es independiente de los contextos de los otros procesos, siempre que no se usen técnicas explícitas para compartir recursos.

Las arquitecturas que son totalmente *SMT* y capaces de trabajar con muchos hilos son extremadamente costosas en términos de complejidad. Es importante tener en cuenta que las estructuras necesarias para soportar este tipo de compartición crecen proporcionalmente con el número de hilos disponibles. Por lo tanto, si bien es cierto que con más paralelismo se obtiene más rendimiento, cuanto más paralelismo, más área y más consumo energético. En estos casos hay que conseguir un balance de rendimiento frente a un consumo energético, complejidad y área.

Ejemplos

El HyperThreading de Intel implementa solo contextos y dos hilos. Otro ejemplo es el Alpha 21464, que dispone de hasta 4 hilos paralelos.

El resto del apartado trata del diseño de este tipo de arquitecturas, como también de algunas de las arquitecturas *SMT* más relevantes de la literatura.

4.3. Convirtiendo el paralelismo a nivel de hilo a paralelismo a nivel de instrucción

Como ya se ha comentado, las arquitecturas *SMT* permiten que diferentes hilos de ejecución compartan diferentes unidades funcionales. Para permitir este tipo de compartición, se tiene que guardar de manera independiente el estado de cada uno de los hilos.

Por ejemplo, hay que tener por duplicado los registros que usan los hilos, su contador de programa y también una tabla de páginas separada por hilos. Si no se tienen duplicadas estas estructuras, la ejecución funcional de cada uno de los hilos interferirá con la de otro hilo. Por otro lado, podría haber problemas de seguridad graves. Por ejemplo, compartir la tabla de páginas implicaría que un hilo compartiría el mapeo de direcciones virtuales a física. Esto implicaría que un hilo podría acceder a la memoria del otro hilo sin ningún tipo de restricción. Sin embargo, hay otros recursos que no hay que replicar, como por ejemplo, el acceso a las unidades funcionales para acceder a memoria (puesto que los mecanismos de memoria virtual ya apoyan por multiprogramación).

La cantidad de instrucciones que un procesador *SMT* puede generar por ciclo está limitada por los desbalanceos en los recursos necesarios para ejecutar los hilos y la disponibilidad de estos recursos. Sin embargo, también hay otros factores que limitan esta cantidad, como el número de hilos que hay activos, posibles limitaciones en la medida de las colas disponibles, la capacidad de generar suficientes instrucciones de los hilos disponibles o limitaciones del tipo de instrucciones que se pueden generar desde cada hilo y para todos los hilos.

Las técnicas *SMT* asumen que el procesador facilita un conjunto de mecanismos que permiten explotar el paralelismo a nivel de hilo. En particular, estas arquitecturas tienen un conjunto grande de registros virtuales que pueden ser usados para guardar los registros de cada uno de los hilos de manera independiente (asumiendo, evidentemente, diferentes tablas *de renaming* para cada hilo).

Renaming

El *renaming* de registros facilita identificadores únicos de registro. Sin este tipo de *renaming* dos hilos podrían tener interferencias entre sus ejecuciones.

Ejemplo de flujo de instrucciones multihilo

El flujo de ejecución de los dos hilos que se presentan a continuación no funcionaría correctamente. Si los registros r2, r3 y r4 no fueran renombrados por cada uno de los hilos, la ejecución funcional de las diferentes instrucciones sería errónea. En los instantes 1 y 4 los valores que los dos hilos leerían serían incorrectos. En el primer caso el hilo 2 estaría empleando el valor del registro r3 modificado por el hilo 1 en el ciclo anterior. De manera similar, lo mismo sucedería en el ciclo 3, en el que el hilo 1 estaría sumando un valor r3 modificado por otro hilo.

```
ciclo(n)hilo 1-->"LOAD  #43,r3"
ciclo(n+1)hilo 2-->"ADD   r2,r3,r4"
ciclo(n+2)hilo 1-->"ADD   r4,r3,r2"
ciclo(n+3)hilo 1-->"LOAD  (r2),r4"
```

Gracias al *renaming* de registros, las instrucciones de diferentes hilos pueden ser mezcladas durante las diferentes etapas del procesador sin confundir fuentes y destinos entre los diferentes hilos disponibles. Es importante hacer notar que este tipo de técnica es la que se emplearía en los procesadores fuera de orden *SISD*. En este último caso el procesador tendría una sola tabla *de renaming*.

Así pues, el proceso de *renaming* de registros es exactamente el mismo proceso que hace un procesador fuera de orden. Por eso un *SMT* se puede considerar como una extensión de este tipo de procesadores (añadiendo, está claro, toda la lógica necesaria para soportar los contextos de los diferentes hilos). Como se puede deducir, se puede construir una arquitectura fuera de orden y *SMT* a la vez.

El proceso de finalización de una instrucción[11] no es tan sencillo como en un procesador no *SMT* (que solo tiene en cuenta un hilo). En este caso se quiere que la finalización de una instrucción sea independiente para cada hilo. De este modo, cada uno puede avanzar independientemente de los otros. Esto se puede llevar a cabo con las estructuras que permiten este proceso para cada uno de los hilos, por ejemplo, teniendo un *reoderbuffer* por hilo.

(11)En inglés, *commit*.

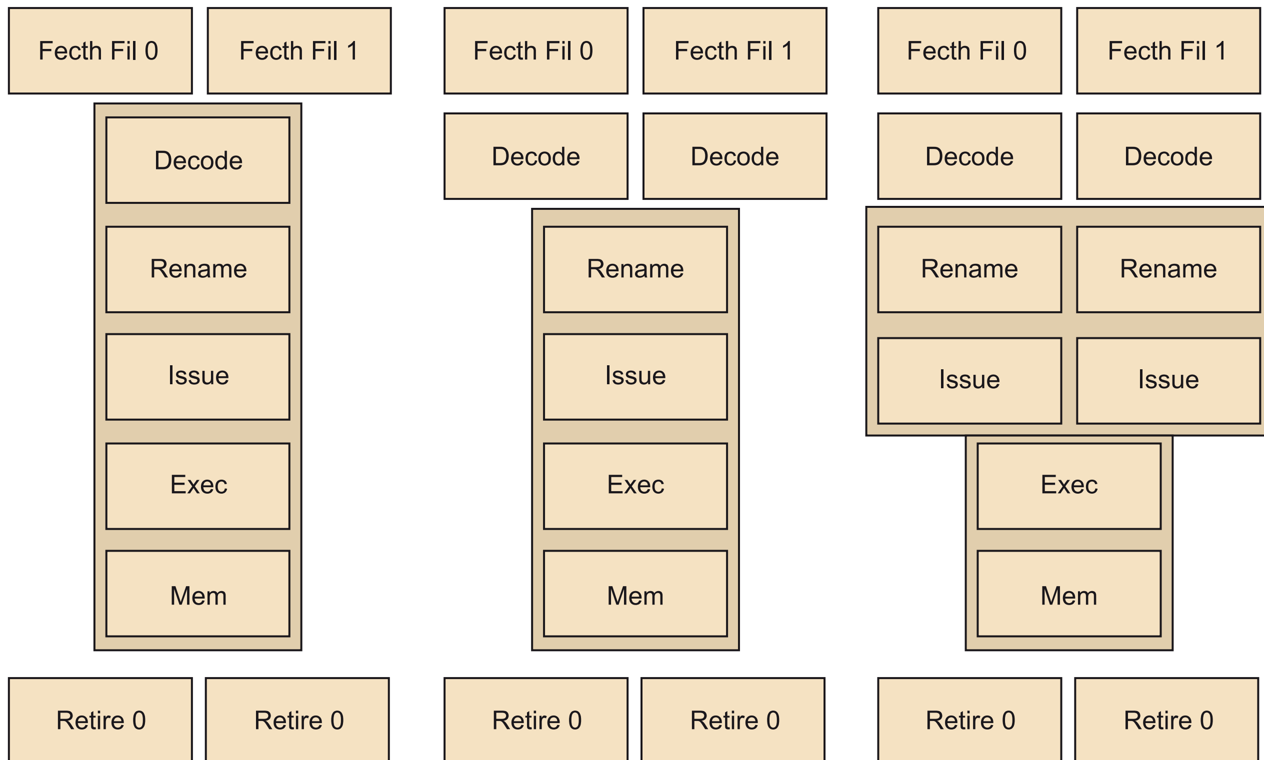
4.4. Diseño de un *SMT*

En términos generales, los *SMT* siguen la misma arquitectura que los procesadores superescalares. Esto incluye tanto los diseños generales de las etapas que los componen (etapa de búsqueda de instrucción, etapa de descodificación y lectura de registros, etc.), como las técnicas o los algoritmos empleados (por ejemplo, “Tomasulo”).

Sin embargo, muchas de las estructuras tienen que ser replicadas para apoyar a los diferentes contextos que el procesador tiene que gestionar. Algunas son las mínimas necesarias para evitar interferencias entre hilos y asegurar la corrección de las aplicaciones, como tener contadores de programas separados o tablas de páginas separadas. De todos modos, se pueden encontrar variantes arquitectónicas que no son estrictamente necesarias, pero que pueden dar más rendimiento en ciertas situaciones.

La figura siguiente muestra algunas de las opciones que se pueden tener en cuenta cuando se considera la arquitectura global de un procesador *SMT*. Esta figura muestra diferentes posibilidades de la manera como los diferentes hilos comparten o no las diferentes etapas del procesador (se han asumido las etapas típicas de un procesador superescalar fuera de orden). Por ejemplo, la primera de todas asume que solo la búsqueda de instrucción está dividida por hilos, el resto de etapas son compartidas.

Figura 11. Posible diseño de una arquitectura *SMT*



Cuanto más a la derecha nos movemos en la figura, las etapas se encuentran más divididas por hilos. Hay que remarcar que el hecho de que una etapa se encuentre separada por hilos ejemplifica que el procesador realiza la etapa dividida por hilos y que cada hilo tiene estructuras separadas para llevarla a cabo. Sin embargo, es lógico pensar que todos los hilos tienen una lógica compartida, puesto que el mecanismo es común entre todos.

En todos los casos, las etapas de ejecución y acceso a memoria se encuentran compartidas por todos los hilos. En un estudio académico se podría considerar que estas se encuentran replicadas por hilos, pero por ahora, en un entorno real, esto es muy costoso en términos de espacio y de consumo energético. Por otro lado, como es lógico, la utilización de estos recursos será mucho más elevada en los casos en que todos los hilos los compartan. Así pues, cuando unos estén bloqueados, los otros los usarán, y viceversa, o bien cuando unos estén haciendo accesos a memoria los otros pueden usar las unidades aritmeticológicas. Por lo tanto, para maximizar la eficiencia energética del procesador hace falta que se encuentren compartidas.

En estos ejemplos la etapa de búsqueda de instrucción se separa por hilos. De todos modos, esto no es común en todos los diseños posibles. Como se verá más adelante, el HyperThreading de Intel contiene una etapa de búsqueda de instrucción compartida por todos los hilos. En la búsqueda se incluye la lógica de selección de hilo, como también el predictor de saltos. También hay que

considerar que el acceso a la memoria caché de instrucciones es independiente por hilos. Esta memoria tiene que tener puertos de lectura suficientes para satisfacer la necesidad de instrucciones de los hilos.

Las etapas de descodificación y *renaming* identifican las fuentes y destinos de las operaciones, como también calculan las dependencias entre las instrucciones en vuelo. En este caso, por el hecho de que las instrucciones de los diferentes hilos son independientes, podrían tener la lógica correspondiente de manera separada. Sin embargo, tener las estructuras de *renaming* compartidas hace que el procesador pueda ser más eficiente en la mayoría de casos.

> Por ejemplo, si se asume que se dispone de un total de 50 registros de *renaming* para los dos hilos y las estructuras están separadas, cada hilo podrá tener acceso a 25 registros como máximo. En los casos en que uno de los hilos solo pueda emplear 10, pero el otro necesite 40, el sistema estará infrautilizado, de manera que el rendimiento del segundo hilo se reducirá sustancialmente.

4.5. Complejidades y retos en las arquitecturas *SMT*

Las arquitecturas *SMT* incrementan notoriamente el rendimiento del sistema aumentando la ventana de instrucciones que este puede gestionar. Así, en un mismo ciclo, el procesador puede escoger un abanico amplio de instrucciones de los diferentes hilos disponibles en el sistema. El resto de etapas se encuentran también más utilizadas por el mismo motivo. Sin embargo, hay que tener en cuenta que estas mejoras se dan en contra del rendimiento individual del hilo. En este caso, el rendimiento que un solo hilo puede conseguir puede ser menor del que habría obtenido en un procesador superescalar fuera de orden sin *SMT*.

Para evitar este detrimento individual en los hilos, algunos de estos procesadores introducen el concepto de *hilo preferido*. La unidad encargada de generar o empezar instrucciones dará preferencia a los hilos preferidos[12]. *A priori* puede parecer que este aspecto puede favorecer el hecho de que algunos de los hilos tengan un rendimiento más alto y que no se sacrifique el rendimiento global del sistema.

(12)En inglés, *preferred threads*.

Ahora bien, esto no es del todo cierto, porque dando preferencias a un subconjunto de hilos se provoca una disminución en la ILP del flujo de instrucciones que circulan por el *pipeline* del procesador. El rendimiento de las arquitecturas *SMT* se maximiza cuando hay suficientes hilos independientes que permitan amortiguar los bloqueos que cada uno experimenta cuando se ejecuta.

Hay que comentar que también hay algunas arquitecturas en las que solo se consideran los hilos preferidos, siempre que no se bloqueen. Si uno no puede seguir adelante, el procesador considera los otros hilos. En estos casos lo que se está haciendo es causar un desbalanceo de rendimiento en los diferentes

hilos que el procesador ejecuta. Este factor se tiene que tener en cuenta a la hora de planificar la ejecución de los diferentes hilos que se ejecutan sobre el sistema operativo.

Además del reto que las arquitecturas *SMT* muestran hacia la mejora del rendimiento individual de los hilos, hay una variedad de otros retos que hay que afrontar en el diseño de un procesador de estas características, como son los siguientes:

- Mantener una lógica simple en las etapas que son fundamentales y que hay que ejecutar en un solo ciclo. Por ejemplo, en la elección de instrucción simple hay que tener presente que cuantos más hilos hay, la bolsa de instrucciones que se pueden escoger es más grande. Pasa lo mismo en la etapa de fin de instrucción, en la que el procesador tiene que escoger cuál de las instrucciones acabadas finalizarán en el próximo ciclo.

- Tener diferentes hilos de ejecución implica que hay que tener un banco de registros suficientemente grande como para guardar cada uno de los contextos. Esto tiene implicaciones tanto de espacio como de consumo energético.

- Uno de los problemas de tener diferentes hilos de ejecución compartiendo los recursos de un mismo procesador puede ser el acceso compartido a la memoria caché. Puede pasar que los mismos hilos hagan lo que se denomina falsa compartición, que es cuando hilos diferentes están compartiendo los mismos sets de la memoria caché, a pesar de que están accediendo a direcciones físicas diferentes. En los casos con mucha falsa compartición, el rendimiento del sistema se degrada de manera sustancial.

Los subapartados siguientes presentan dos tecnologías comerciales que incorporaron el concepto de multihilo compartido en sus diseños: el HyperThreading de Intel y el procesador 21464 de Alpha.

Ejemplos de procesadores

El Memory Logix MLX1 (Song, 2002), el Clearwater Netwroks CNP810SP (Melvin, 2000) o el Flow Strom Porthos (Melvin, 2003) fueron otros ejemplos de procesadores.

4.6. Implementaciones comerciales de *SMT*

4.6.1. El HyperThreading de Intel

HyperThreading fue el nombre comercial con el cual la empresa Intel introdujo en el mercado las tecnologías *SMT* en sus productos. En noviembre de este año Intel lanzó el procesador IntelPentium 4. Este fue el primero que incluyó *SMT* en su diseño.

En el 2010, Intel lanzó el procesador Atom, que también incluye la tecnología *SMT*. Sin embargo, este producto está orientado hacia el mercado de los dispositivos de bajo consumo. Esto incluye portátiles de bajo consumo, tabletas,

teléfonos móviles, etc. Por este motivo, a pesar de ser una tecnología *SMT*, este no es fuera de orden. Por lo tanto, no incluye técnicas de reordenamiento de instrucciones, ejecución especulativa o *renaming* de registros.

Dentro de la gama de los productos Xeon (Intel) orientada a servidores, también se puede encontrar esta tecnología. En este caso, como el segmento de mercado destinado es el de los servidores, los procesadores ofrecen un nivel de paralelismo mucho más elevado y todas las funcionalidades de un procesador fuera de orden. En este caso, nos movemos del procesador Bloomfield, con 4 núcleos y un total de 8 hilos de ejecución, hasta el procesador Beckton, con 8 núcleos y un total de 16 hilos de ejecución. Aquí hay que hacer notar que la tecnología *SMT* se incluye dentro de cada núcleo.

Como se puede observar, la tecnología *SMT* es aplicable a muchos tipos de segmentos: segmento de dispositivos móviles o segmentos de computación de altas prestaciones. El concepto de base en todos los casos es el mismo, potenciar el rendimiento de los sistemas aumentando el nivel de paralelismo de las aplicaciones. En unos casos se necesita un rendimiento muy elevado (por ejemplo, por servidor de bases de datos) y en otros solo poder tener diferentes hilos ejecutándose en paralelo (por ejemplo, el navegador y el gestor de correo). El consumo y complejidad en los dos casos también es bastante diferente.

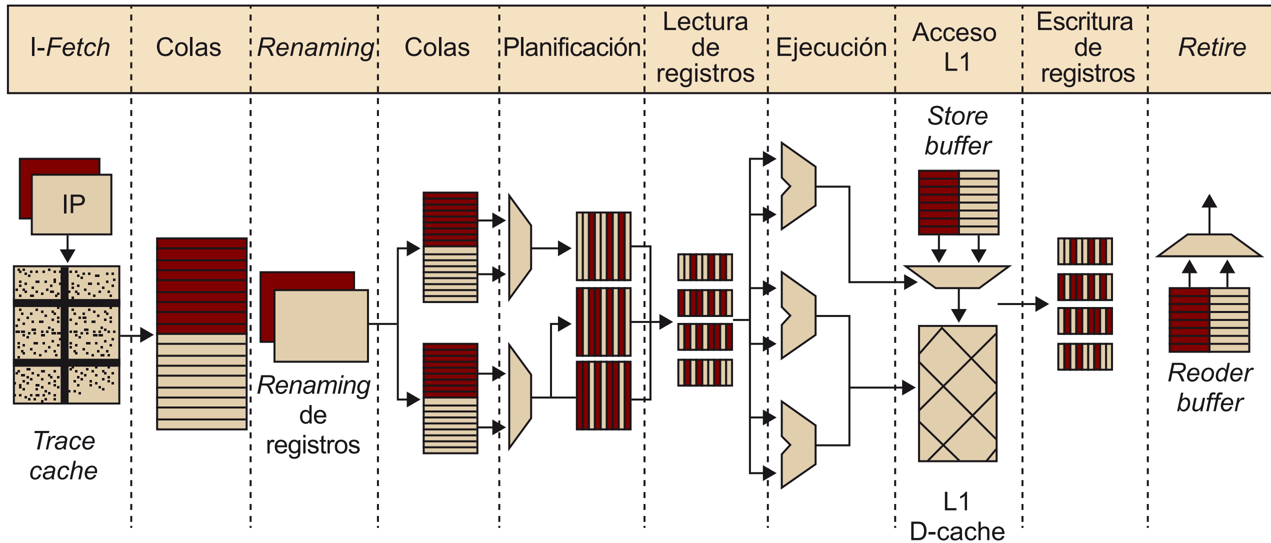
Consumo

Un procesador Atom puede consumir unos 10 vatios, mientras que un procesador Beckton puede consumir hasta 130 vatios.

La arquitectura

La figura siguiente presenta las diferentes etapas de la microarquitectura del Pentium IV, conocida también como *Netburst Micro-Architecture* (Intel, Sandy Bride Intel). El *pipeline* consta de diez etapas diferentes. Las cuatro primeras son en orden y son las relacionadas con la búsqueda y preparación de las instrucciones (búsqueda, *renaming* y traducción a microoperaciones). Las cinco siguientes pueden ser fuera de orden y son las encargadas de llevar a cabo la ejecución funcional de las operaciones. Finalmente, la última se ejecuta en orden y es donde las instrucciones finalizan.

Figura 12. *Pipeline* del Pentium IV



Durante las diferentes etapas del procesador, hay recursos que se encuentran replicados por procesador lógico, los hay que se encuentran compartidos pero divididos por procesador lógico y, finalmente, los hay que se encuentran totalmente compartidos.

En primera instancia, cada procesador lógico mantiene una copia separada de su estado arquitectónico necesario para una ejecución funcional correcta. Los recursos encargados de guardar este tipo de información son los que se encuentran replicados para cada contexto:

- El puntero en la instrucción siguiente.
- El *buffer* de instrucciones del hilo, es decir, el flujo de instrucciones que el hilo potencialmente ejecutará.
- El *translation look-aside buffer* (*TLB*), usado para hacer la traducción de direcciones virtuales a físicas. Cada hilo tiene que tener un mapeo de dirección virtual a una dirección física diferente. De lo contrario habría problemas de seguridad potenciales.

- El predictor de dirección de regreso[13], que, como el nombre indica, predice las direcciones de regreso de las funciones.

(13)En inglés, *return stack predictor*.

- El *advanced programmable interrupt controller* (*APIC*) orientado a la gestión de interrupciones.
- La mesa *de renaming*, necesaria para poder llevar a cabo el fuera de orden y poder hacer el remapeo de registros virtuales a registros físicos.

A pesar de que hay otros recursos que se encuentran replicados, los más importantes son los que acabamos de mencionar. También hay que hacer notar que esta descripción equivale a una arquitectura *hyperthreading* general. Cada

implementación concreta, por ejemplo, la del procesador Atom, puede tener variaciones según el tipo de requisitos que tiene el procesador y del diseño que han llevado a cabo los arquitectos.

En segunda instancia, encontramos ciertos recursos que se encuentran divididos entre los diferentes hilos. En general, la mayoría son *buffers*.

Buffers

Como por ejemplo, el *reorder buffer*, los *load* y *storebuffers*, colas entre las diferentes etapas, etc.

Finalmente, encontramos otros recursos que se encuentran totalmente compartidos entre todos los hilos del procesador:

- La *trace cache* o memoria caché de instrucciones (y los *buffers* correspondientes). Este es un mecanismo que se diseñó para aumentar el ancho de banda de la etapa de búsqueda de instrucciones y reducir el consumo del procesador. Consiste en guardar trazas de instrucciones que ya se han seleccionado previamente y de las cuales ya se tiene una descodificación.

- Las memorias caché.

- Los mecanismos de ejecución fuera de orden.

- Los predictores de salto.

- Lógica de control y buses de comunicación.

Uno de los puntos clave en un diseño *SMT* es la complejidad y área que se añade al diseño de este por el hecho de considerar los diferentes hilos de ejecución. Una de las ventajas de la HyperThreading es la capacidad de proporcionar una mejora de rendimiento importante respecto de un aumento de área reducido. Uno de los motivos principales es que los procesadores lógicos comparten casi todos los recursos del procesador físico. El aumento de área se produce básicamente por los estados arquitectónicos extras, la lógica de control adicional y la replicación de algunos de los recursos.

La perspectiva del sistema

El HyperThreading, tal como hemos dicho, hacía que un solo procesador se viera desde el punto de vista del sistema como si fueran diferentes procesadores lógicos (o hilos de ejecución). El procesador tiene una copia del estado arquitectónico para cada uno de los procesadores lógicos, y todos los procesadores lógicos comparten un conjunto de recursos físicos que permiten ejecutar los diferentes flujos de instrucciones.

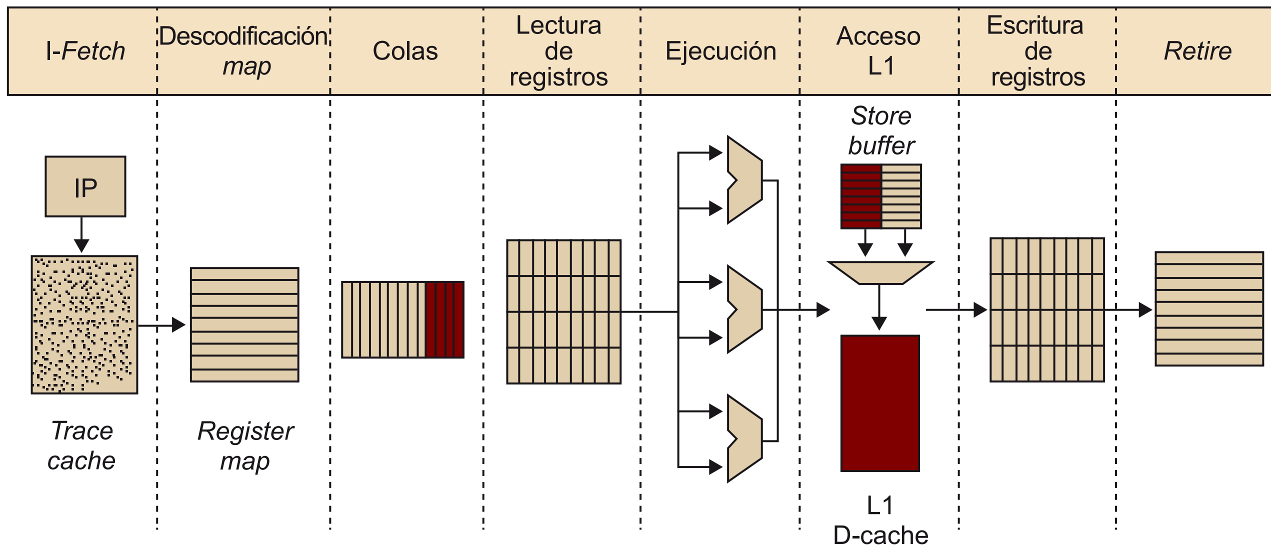
Desde la perspectiva del sistema operativo y de las aplicaciones, el hardware les ofrece *N* procesadores independientes. El software puede gestionar los diferentes hilos de ejecución sobre los procesadores lógicos según sus políticas de administración. Es importante hacer notar que, desde su punto de vista, es equivalente a tener a su disposición *N* procesadores físicos independientes.

Desde la perspectiva del modelo de programación, la HyperThreading se puede ver como una arquitectura multiprocesador con tiempo de acceso uniforme a memoria (conocido como *NUMA*). Todos los hilos tienen, de media, un tiempo de acceso a memoria similar.

4.6.2. El Alpha 21464

El Alpha 21464 (Rusu, Tam, Muljono, Ayers y Chang, 2006), también conocido como EV8, fue la evolución del Alpha 21364 (figura 13). La mejora más importante del EV8 respecto de este último fue que incorporaba técnicas *SMT*. A pesar de que tenía un rendimiento estimado bastante más elevado que su predecesor, la línea de procesador Alpha se canceló en el 2001. Esto incluyó el EV8.

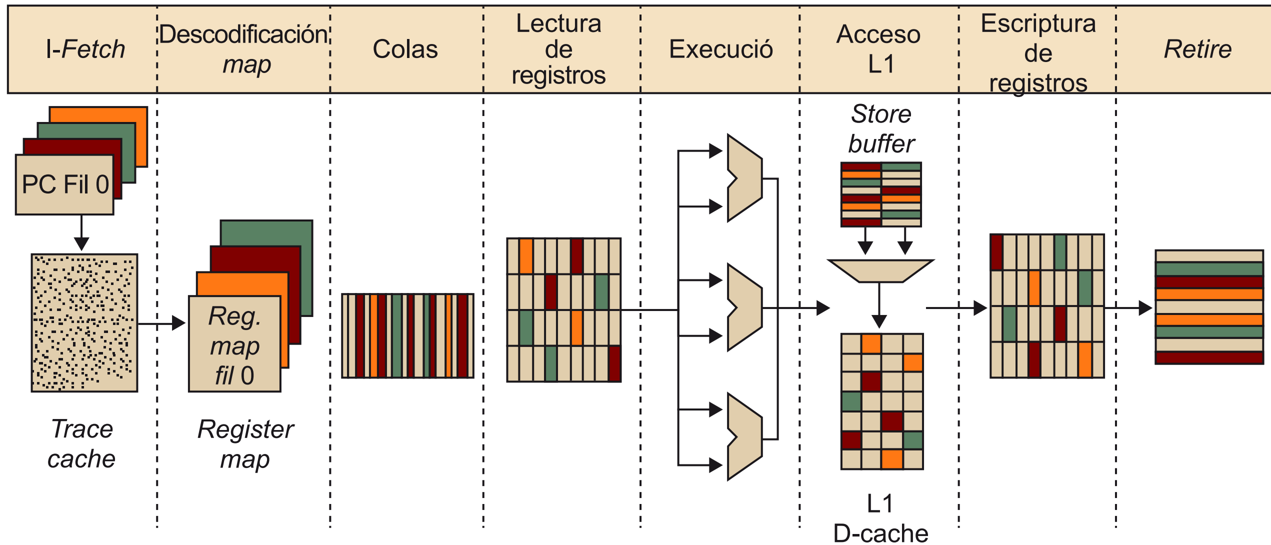
Figura 13. *Pipeline* del Alpha 21364



La arquitectura

La figura 14 muestra las diferentes etapas de que se componía. Para añadir *SMT* a la arquitectura Alpha, la EV8 incluye una lógica nueva para mantener cuatro contadores de programas independientes (uno por hilo). A cada ciclo, la etapa de *fetch* selecciona uno solo de los hilos disponibles de la memoria caché de instrucciones. Una vez la instrucción es seleccionada y cargada, se marca con el identificador del hilo al que pertenece. Este identificador va emparejado con la instrucción durante todas las etapas del procesador (etapa de descodificación, unidades funcionales, etc.).

Figura 14. *Pipeline* del Alpha 21464



La lógica que gestiona la selección fuera de orden de las instrucciones que pueden ser ejecutadas puede escoger una instrucción de cualquiera de los cuatro hilos. De manera similar a la que ya hemos visto con el HyperThreading, la complejidad añadida a la lógica fuera de orden y la lógica de *renaming* es bastante reducida. En el caso del EV8, esto implicaba solo un incremento del 6% del área.

El cambio más importante introducido en las arquitecturas Alpha para apoyar a la *SMT* fue que incorporaban 32 registros enteros y 32 de coma flotante para cada uno de los cuatro hilos de ejecución. Por lo tanto, para guardar todo el estado arquitectónico de todo el procesador hacen falta 256 registros. Por otro lado, para poder llevar a cabo el *renaming* y poder soportar un total de 256 instrucciones en vuelo, el EV8 dispone de 256 registros extra. En total incluye 512 registros.

A diferencia de la microarquitectura NetBurst, la Alpha 21364 solo replica el contador de programa y los *registermap*. El resto de recursos se encuentran compartidos:

- La cola de instrucciones.
- El *register file*, que como ya se ha mencionado, se encuentra incrementado notoriamente respecto del Alpha 21364.
- El primero y segundo nivel de memorias caché.
- El *translation look-aside buffer* (TLB).
- El predictor de saltos.

Desde el punto de vista del sistema o de la aplicación, la HyperThreading ofrece cuatro hilos de ejecución totalmente independientes. Como ya se ha dicho, cada hilo se ve como un procesador lógico diferente (cuatro hilos se ven como cuatro procesadores independientes).

La perspectiva del sistema

En el caso del EV8, el sistema también tiene acceso a un solo procesador físico. Sin embargo, en este caso también hay un solo procesador lógico. Cada hilo es visto como una unidad de proceso[14], que comparte recursos, como por ejemplo la memoria caché de instrucciones, memoria caché de datos, la TLB o la memoria caché de segundo nivel con los otros hilos.

(14)En inglés, *thread processing unit* (*TPU*).

4.7. Arquitecturas multinúcleo

4.7.1. Limitaciones de la *SMT* y arquitecturas superthreading

En todos los casos anteriores, la explotación del paralelismo se lleva a cabo añadiendo el concepto de hilo a la arquitectura del procesador. En este caso, el aumento de paralelismo se consigue incrementando la lógica del procesador, analizando el diferente número de hilos que se quiere considerar. Por ejemplo, si se quiere tener ocho hilos, habrá que replicar ocho veces las estructuras necesarias (más o menos veces según el diseño *SMT* que se está considerando). Este modelo, a pesar de ser adecuado y empleado en la actualidad por muchos procesadores, tiene ciertas limitaciones. A continuación se discuten las más representativas.

Escalabilidad y complejidad

Los modelos *SMT* son adecuados para un número relativamente pequeño de hilos (2, 4 u 8 hilos). Sin embargo, un número mayor de hilos puede significar ciertos problemas de escalabilidad. Como ya hemos visto, para cada uno de los hilos de que dispone un procesador hay mucha lógica que se encuentra replicada o compartida.

Este hecho podría implicar solo un problema de espacio. Es decir, duplicar el número de hilos implica duplicar el número de entradas de la *store buffer*, o duplicar el número de tablas de *renaming*. Aun así, el incremento en la cantidad y medida del número de recursos también lleva asociado un incremento exponencial en la lógica de gestión de estos recursos.

A modo de ejemplo, se estudia el caso del *reorder buffer*. Esta estructura es la encargada de finalizar todas las instrucciones que ya han pasado por todas las etapas del procesador y que quedan pendientes de ser finalizadas (etapa de *commit*). En el caso de tener dos hilos de ejecución y asumiendo que se genera una instrucción por ciclo por hilo, hay que poder finalizar o “commitear” dos instrucciones por ciclo. De lo contrario, el sistema no es sostenible. Poder finalizar dos instrucciones por ciclo es realista.

Sin embargo, si se incrementa el número de hilos de manera lineal, no es realista esperar que se pueda construir un sistema real que sea capaz de finalizar el número proporcional de instrucciones necesarias para mantener el rendimiento.

La lógica y los recursos necesarios para gestionar la finalización de un número tan elevado de instrucciones en vuelo serían extremadamente costosos y probablemente no alcanzables (si se tuvieran 128 hilos disponibles y 32 instrucciones en vuelo por hilo, harían falta 4.096). Cuando menos, considerando el estado actual de la tecnología de procesadores.

Consumo energético y área

Para apoyar un número elevado de hilos, hay que incrementar las estructuras proporcionalmente. En algunos casos, estos incrementos no son costosos en términos de complejidad y área, pero algunas de las estructuras son altamente costosas de escalar. Otra vez podemos poner como ejemplo el acceso a las memorias caché.

El incremento en el número de puertos de lectura o escritura de las diferentes memorias caché es altamente costoso (tanto en cuanto a complejidad como al área). Añadir un puerto de lectura nuevo en una memoria caché puede equivaler a un incremento de un 50% de área (Handy, 1998).

Como ya se ha dicho, si se quiere incrementar el rendimiento de manera más o menos proporcional al número de hilos, también hay que tener en cuenta cómo se accede a la memoria caché. Por lo tanto, si se quisiera aumentar el número de hilos, por ejemplo a 256, haría falta redimensionar (tanto en área como en lógica) toda la jerarquía de memoria coherentemente. Como ya se puede ver, y con la tecnología actual, esto es impracticable.

El consumo energético de un procesador *SMT* apoyando a un número muy elevado de hilos es alto. En las situaciones en las que no se usarán todos los hilos disponibles o el uso de los recursos correspondientes fuera ineficiente, el consumo en vatios del procesador sería muy elevado comparado con el rendimiento que se estaría obteniendo.

Como se analiza a continuación, en otras arquitecturas y en estas situaciones, se puede aplicar *dynamic voltage scaling* (Yao, Demers y Shenker, 1995), es decir, reducir la frecuencia y voltaje de algunas partes del procesador, puesto que esto permite reducir sustancialmente el consumo del procesador en situaciones como la planteada.

4.7.2. Producción

Durante las últimas décadas, dada una gama de procesadores que siguen un diseño arquitectónico similar (por ejemplo, los SandyBridge de Intel), se sacan diferentes versiones de un mismo procesador. En una misma familia se pueden encontrar versiones orientadas a los clientes (ordenadores de uso doméstico), versiones orientadas a dispositivos móviles y versiones orientadas a servidores.

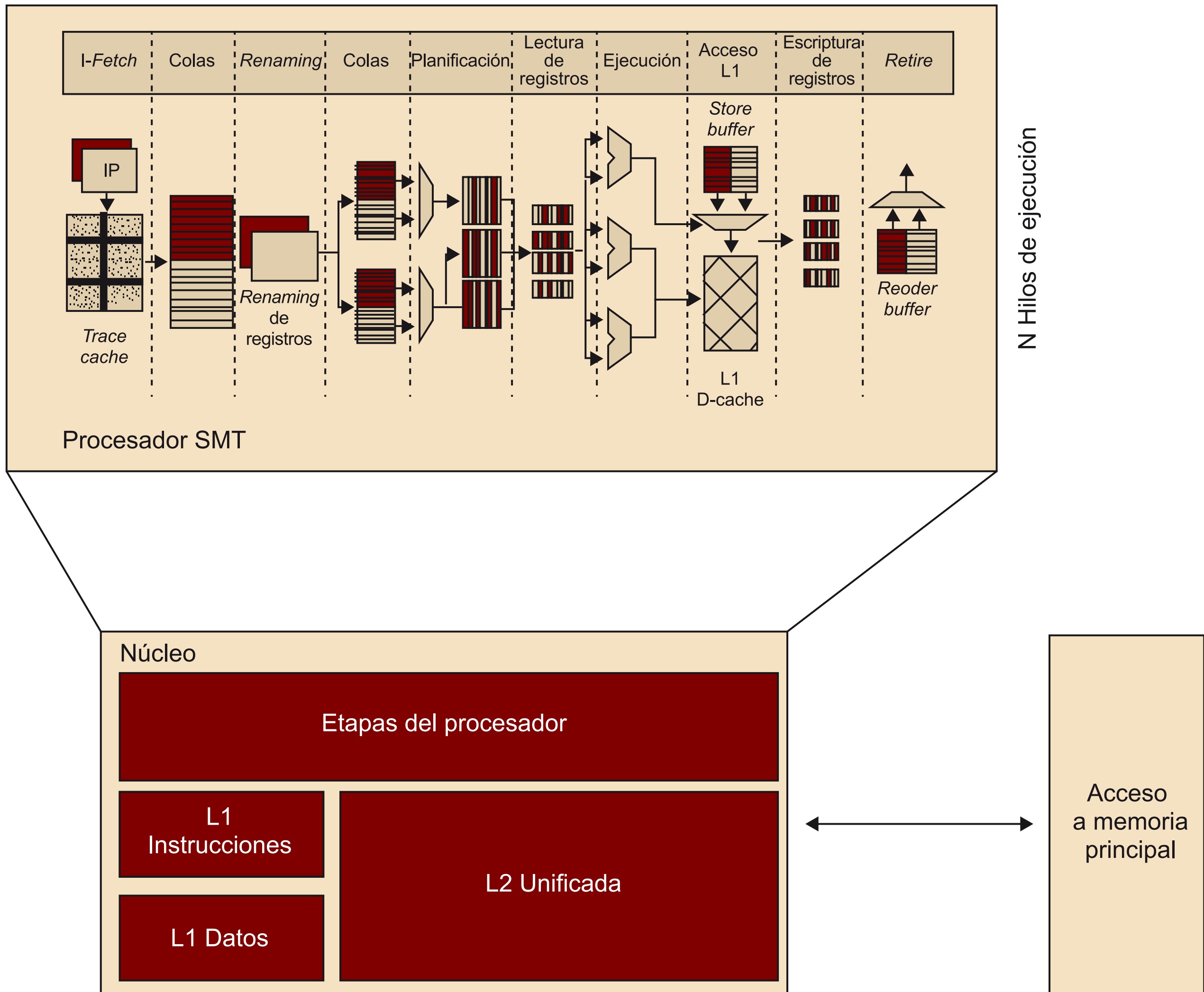
Por ejemplo, en el caso de la familia SandyBridge, se pueden encontrar los procesadores domésticos con 12 hilos y 130 vatios de consumo, procesadores para dispositivos móviles de 8 hilos y 55 vatios de consumo y procesadores para servidores con 16 hilos y 150 vatios de consumo. Justo es decir que no solo varía el número de hilos entre los diferentes tipos de procesadores, sino que también lo hacen las medidas de memorias caché y prestaciones específicas (por ejemplo, la capacidad de conectividad con otros procesadores). Dentro de un mismo tipo de procesadores (por ejemplo, los clientes) hay muchas variantes (por ejemplo, dentro de la familia SandyBridge de tipo cliente hay más de treinta variantes).

Intentando dar apoyo a toda esta variedad de número de hilos, limitándose a escalar la cantidad de hilos que la arquitectura *SMT* soporta, sería extremadamente costoso desde el punto de vista de producción. Es decir, la complejidad de tener tantos hilos diferentes encarecería mucho más el proceso de diseño, producción y validación de los procesadores. Como veremos a continuación, usando tecnologías multinúcleo este proceso deviene menos costoso y más factible.

4.7.3. El concepto de multinúcleo

En los últimos subapartados, hemos hablado de diferentes estructuras multihilos. Cada una implementaba un procesador superescalar fuera de orden, añadiendo el concepto de hilo. Independientemente del tipo de arquitectura multihilo (*superthreading* o *SMT*), estos procesadores se podrían ver como un elemento de computación, con *n* hilos y una jerarquía de memorias caché. Esta abstracción (como muestra la figura siguiente) se puede denominar núcleo. Hay que remarcar que en este caso no incluye otros elementos que un procesador superescalar sí que incluiría: sistema de memoria, acceso a la entrada y salida, etc.

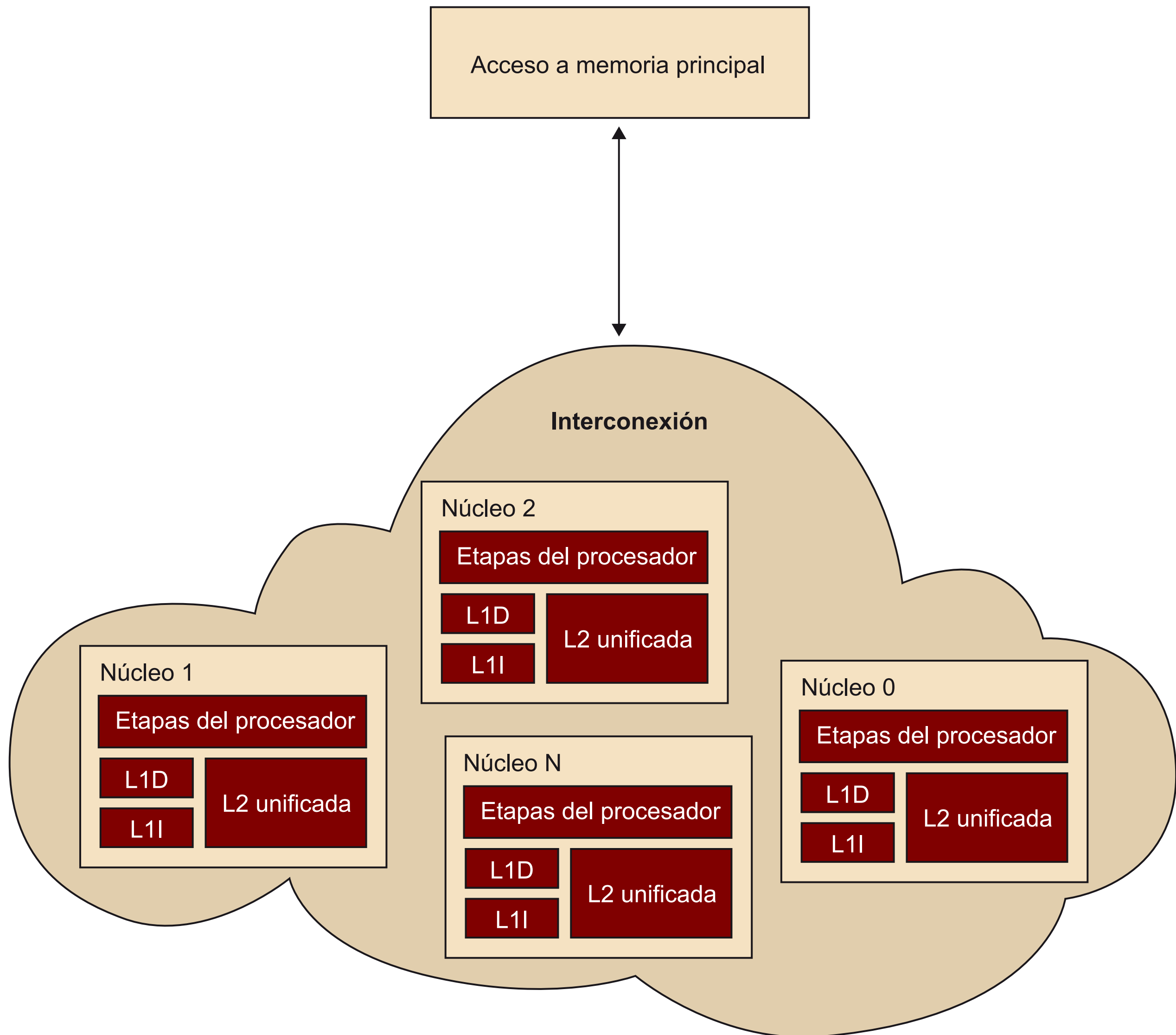
Figura 15. Abstracción de procesador multihilo



De hecho, el concepto de multinúcleo, como indica su nombre, consiste en replicar *m* núcleos diferentes dentro de un procesador (figura 16). Cada núcleo acostumbra a tener una memoria caché de primer nivel (de datos e instrucción), y puede tener una memoria de segundo nivel (acostumbra a ser unificada: datos más instrucciones). Aparte de los núcleos, el procesador acostumbra a tener otros componentes especializados y ubicados fuera de estos. Habitualmente, están los siguientes: una memoria de tercer nivel, un controlador de memoria, componentes para hacer procesamiento de gráficos, etc.

Todos estos componentes (incluidos los núcleos) están conectados por una red de interconexión, que es el medio físico y lógico que permite enviar peticiones de un componente a otro (por ejemplo, una petición de lectura de un núcleo en la L3).

Figura 16. Abstracción de multinúcleo



Como ya se ha dicho, uno de los componentes de un multinúcleo fundamental es el controlador de memoria. Este gestiona las peticiones de acceso al subsistema de memoria que hacen el resto de componentes (tanto lecturas como escrituras).

Por sí misma, una arquitectura multihilo puede parecer sencilla. Sin embargo, detrás de este tipo de arquitecturas hay mucha complejidad escondida que no se ve directamente: protocolos de coherencia, escalabilidad en la interconexión, sincronización, desbalanceos entre hilos, etc.

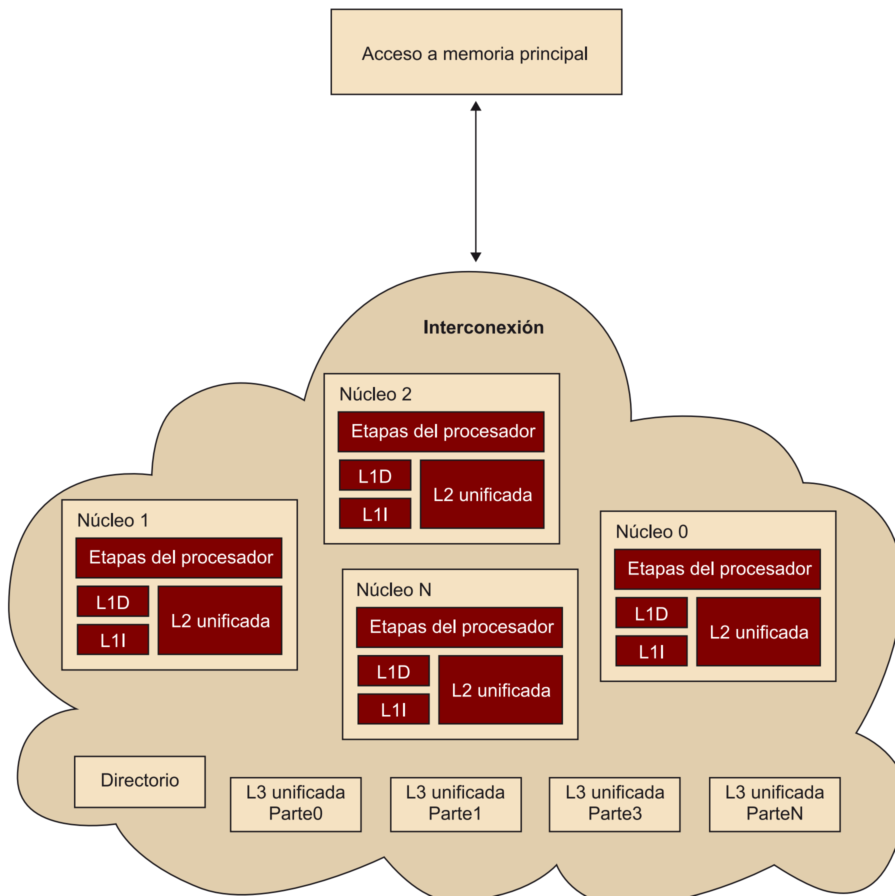
4.7.4. Coherencia entre núcleos

El modelo que acabamos de ver describe la estructura más sencilla de un multinúcleo: núcleos, interconexión y acceso a memoria principal. Sin embargo, como ya se puede deducir, hay muchas variantes de este modelo.

La figura 17 muestra una arquitectura a menudo empleada en el mundo de la búsqueda académica y también presente en modelos comerciales. Del mismo modo que el modelo anterior, se encuentran un conjunto de núcleos que proporcionan acceso a recursos computacionales (con un hilo o más) y a dos memorias caché (una de primero y una de segundo nivel). Aparte de estos componentes, hay dos nuevos: una memoria de último nivel (o memoria caché de tercer nivel) y un directorio.

A diferencia de la L1 o L2, la L3 está dividida en bloques de la misma medida y ubicada en componentes separados. Como veremos a continuación, el directorio es el encargado de saber qué líneas tiene cada bloque de la L3 y en qué estado se encuentran. Por otro lado, también se encarga de coordinar y de gestionar todas las peticiones que los diferentes núcleos hacen a este último nivel de memoria caché.

Figura 17. Arquitectura multihilo con L3 y directorio



En este modelo, cuando una petición de lectura o escritura no acierta ninguna línea ni de la L1 ni de la L2, la petición se reenvía a la memoria de tercer nivel. Conceptualmente, esto es equivalente al flujo de fallo de la L1 que pide la misma petición a la L2. Sin embargo, como ya se ha comentado, esta petición se reenvía hacia el directorio.

El directorio, dada la dirección que se está pidiendo, tiene información para saber cuál de los componentes de la L3 tiene esta línea y en qué estado se encuentra. En caso de que ningún componente la tenga, se encarga de pedirla a la memoria, guardarla en el bloque correspondiente de L3 y enviarla al núcleo. Esto se hace por medio de protocolos concretos que aseguran el orden y la finalización en el tratamiento de estas peticiones.

En el caso anterior, se ha asumido el escenario en el que ni la L1 ni la L2 del núcleo contienen la dirección que el hilo del mismo núcleo está pidiendo. También se ha asumido que el núcleo genera una petición de lectura en el directorio y que este gestiona la petición en la L3. Aun así, ¿qué pasaría si algún otro núcleo tuviera la misma dirección en su L2/L1?

En este caso, podría pasar lo que muestra el ejemplo siguiente. El primer núcleo lee la dirección @X, la modifica y la escribe de vuelta en la memoria caché de último nivel con el valor nuevo. Si durante todo el proceso de esta transacción otro núcleo lee el valor de @X de la L3, recibe un valor incorrecto. En el caso del ejemplo, el núcleo 2 recibiría un valor erróneo.

Ejemplo de flujo de lecturas y escrituras incoherente

```
instante(n)     núcleo 1-->lectura       @X =11    (L3-->L1/L2)
instante(n+1)   núcleo 1-->escritura     @X=0      (L2)
instante(n+2)   núcleo 2-->lectura       @X=11     (L3-->L1/L2)
instante(n+3)   núcleo 1-->escritura     @X=0      (L2-->L3)
```

Para evitar estos problemas se usan protocolos de coherencia, que están diseñados para evitar situaciones como la presentada y mantener la coherencia entre todos los núcleos del procesador.

Cada procesador implementa modelos y mecanismos de coherencia específicos. Por lo tanto, cuando se desarrollan aplicaciones por procesadores multinúcleo hay que conocer bien sus características. El rendimiento de la aplicación depende en gran medida de la manera como se adapta a las características del procesador.

Interconexiones

Dentro del ámbito académico, se han propuesto muchas maneras diferentes de conectar los elementos o componentes de un procesador multinúcleo. La mayoría de estas propuestas vienen de una búsqueda ya realizada en el entorno de computación de altas prestaciones[15].

(15) En inglés, *high performance computing* (*HPC*).

En este ámbito, la problemática de conectar diferentes componentes también aparece, pero a una escala mayor. Es decir, en lugar de conectar núcleos, se conectan procesadores entre sí, o clústeres. El entorno *HPC* tiene mucho más

rodaje en la búsqueda de este tipo de problemas, puesto que es una ciencia que trabaja en estos problemas desde hace muy entrados los años ochenta, cuando se diseñaron los primeros computadores *HPC*.

En cuanto a las redes de interconexión se han propuesto muchos tipos (Agrawal, Feng y Wu, 1978): desde simples buses, hasta estructuras tridimensionales muy complejas como las topologías Torus o el *crossbar*. Sin embargo, por temas de complejidad o coste, no todas son aplicables al mundo de los multinúcleos, donde la escala y las restricciones de consumo y área son más grandes. Justo es decir que, cuanto más avanza la tecnología, más complejas son las redes y más cerca de estos modelos complejos se pueden aproximar los multinúcleos.

Redes en los multinúcleos

Algunos ejemplos de redes que se pueden encontrar en el mundo de los multinúcleos son los buses (Ceder y Wilson, 1986), multibuses (Reed y Grunwald, 1987), anillos de comunicación (Hong y Payne, 1989) o malla (Bell y otros, 2008).

4.7.5. Protocolos de coherencia

Los protocolos de coherencia tienen que garantizar que en todo momento la visión global del espacio de memoria es coherente entre todos los núcleos. Es importante saber qué tipo de protocolo implementa una arquitectura cuando se quiere desarrollar una aplicación. Dada una línea de memoria @X:

- Si uno de los núcleos quiere hacer una lectura de ello, recibe la última copia. No puede darse el caso de que otro núcleo lo esté modificando mientras este tiene una copia.

- Cuando un núcleo pide una línea de memoria, la puede pedir en exclusiva o en estado compartido. El núcleo solo puede modificar la línea cuando esta la tenga en estado exclusivo. Entonces la línea se encuentra en estado modificado.

- Dependiendo del modelo de coherencia que se use, dos núcleos pueden tener @X en las memorias caché respectivas en estado compartido. Ahora bien, no pueden modificar esta línea (porque entonces tendríamos dos valores diferentes de @X en dos núcleos).

- Si uno de los núcleos quiere modificar el dato, antes hay que avisar al resto de núcleos que tienen que invalidar esta línea. Si otro núcleo la vuelve a querer, primero se tiene que escribir en la L3 o en memoria y este lo tiene que leer de L3. Dependiendo del protocolo, podemos encontrar ciertas variantes.

Algunos de los puntos anteriores dependen del tipo de modelo de coherencia que estemos considerando, especialmente el tercero, puesto que, por ejemplo, si el procesador no soporta el estado de línea compartido, cada vez que un núcleo pide una línea, forzosamente tiene que invalidar esta línea de todos los núcleos que la tengan.

En este subapartado se considera un modelo MESI. En este caso, se considera que las líneas de memoria que contiene un núcleo pueden estar en uno de estos cuatro estados: *modified*, *exclusive*, *shared* e *invalid*. Sin embargo, hay otros tipos de modelos (Stenström, 1990): MESIF, MOSI, MEOSI, etc.

Los modelos de coherencia se pueden implementar sobre diferentes tipos de protocolos de coherencia. En términos generales, se pueden diferenciar en dos categorías diferentes:

1) Los protocolos de tipos Snoop. En estos protocolos (Katz, Eggers, Wood, Perkins y Sheldon, 1985) cuando un núcleo quiere hacer alguna acción sobre una dirección de memoria, de lectura o de escritura, tiene que notificarlo de manera pertinente al resto de núcleos para obtener el estado deseado.

> Por ejemplo, si quiere tener una dirección de memoria en exclusiva para modificarla, primero el núcleo tiene que invalidar todas las copias que tengan los otros núcleos. Cuando los otros núcleos hayan respondido notificando que han procesado la petición, el núcleo ya puede usar la línea. Antes, sin embargo, lo tiene que leer de la memoria o de la última memoria caché (L3).

2) Los protocolos basados en directorio. A diferencia de los protocolos de tipo Snoop, aquí los núcleos no se encargan de gestionar la coherencia cuando quieren usar una dirección de memoria (Chaiken, Fields, Kurihara y Agarwal, 1990). En este caso, aparece un componente nuevo que se encarga de gestionarla: el directorio. Cuando un núcleo quiere una línea en un estado concreto, este la pide al directorio. El directorio acostumbra a tener una lista concreta de los núcleos que tienen la dirección en cuestión y en qué estado la tienen. Por lo tanto, cuando procesa una petición ya sabe a qué núcleos tiene que avisar y a cuáles no.

El primero de los dos protocolos es menos escalable que el segundo y necesita muchos más mensajes para gestionar la coherencia entre núcleos. En el segundo caso, el directorio contiene la información de cada línea que tienen los diferentes núcleos y en qué estado la tienen. Por lo tanto, si un núcleo lee una línea que no tiene ningún otro núcleo, el directorio no envía ningún mensaje al resto, sino que directamente le da la línea en estado exclusivo. En cambio, en el caso de protocolos de tipo Snoop, el núcleo envía mensajes para invalidar la línea en concreto a los diferentes núcleos y recibe la notificación. En este segundo caso, el número de mensajes que circulan por la red es mucho más elevado que en caso del directorio.

En cualquier caso, el rendimiento de los protocolos basados en directorio no es gratuito. Las estructuras que guardan el estado y los propietarios de las diferentes líneas son costosas tanto en cuanto al área como al consumo energético. Por lo tanto, para sistemas relativamente pequeños es más eficiente un sistema basado en Snoop.

Hasta ahora hemos presentado los diferentes tipos de arquitecturas de computadores más representativos dentro del ámbito de computación de altas prestaciones. Como se ha visto, cada uno ofrece cierto tipo de prestaciones específicas que pueden ser más adecuadas dependiendo del tipo de aplicaciones que se quieran ejecutar.

Por ejemplo, en el caso de querer hacer cómputos con matrices, en los que las diferentes operaciones se pueden aplicar sobre bloques grandes (por ejemplo, 512 bytes), sería adecuado utilizar procesadores o unidades vectoriales. Por otro lado, si el tipo de problema que se tiene que resolver se caracteriza por un subconjunto de problemas independientes, es interesante emplear una arquitectura de procesador que facilite acceso a un conjunto elevado de hilos de ejecución.

Sin embargo, a pesar de que las arquitecturas mencionadas dan acceso a una capacidad de computación elevada, pueden ser extremadamente complejas de programar. Por este motivo, durante las últimas décadas han aparecido modelos de programación que facilitan el acceso a sus funcionalidades.

Por ejemplo, en el caso de las arquitecturas vectoriales, han aparecido bibliotecas como las *intel advanced vector extensions instrinsics* (también conocidas como *AVX*). Otro ejemplo muy claro son los diferentes modelos de programación que se han propuesto para acceder a arquitecturas de computadores, que dan acceso además de un hilo de ejecución, como son Pthreads, OpenMP, MPI, Intel TBB o Intel Cilck Plus.

A pesar de que estos modelos de programación son extremadamente potentes, hay que considerar ciertas restricciones o situaciones que pueden disminuir sustancialmente su rendimiento. Este tipo de situaciones aparecen más habitualmente en arquitecturas multihilo. El hecho de tener diferentes hilos de ejecución accediendo de manera simultánea y en algunos casos compartida a recursos de computación hace que en algunas situaciones deriven en una lucha que perjudica de manera importante el rendimiento global de las aplicaciones. Los siguientes dos apartados presentan algunos de los factores más importantes que hay que tener en cuenta a la hora de diseñar e implementar una aplicación multihilo.

5. Factores determinantes en el rendimiento en arquitecturas modernas

En el último apartado se han presentado diferentes arquitecturas de procesadores que implementan más de un hilo de ejecución, como también las arquitecturas vectoriales. Así pues, hay arquitecturas que facilitan dos hilos de ejecución, como las *shared multithreading*, y las hay que dan acceso a decenas de hilos de ejecución, como las arquitecturas multinúcleo.

En general, una arquitectura es más compleja cuantos más hilos facilita. Las arquitecturas multinúcleo son bastante escalables y pueden dar acceso a un número elevado de hilos. Ahora bien, como ya se ha visto, para llevarlo a cabo hay que diseñar sistemas que son más complejos. Por otro lado, la complejidad de esta también se ve incrementada si la forman componentes *MIMD* y *SIMD*. En este caso, el modelo de programación tiene que tener mecanismos para poder explotar, por ejemplo, el juego de instrucciones vectoriales.

Cuanto más compleja es la arquitectura, más factores hay que tener en cuenta a la hora de programarla. Si tenemos una aplicación que tiene una zona paralela que en un procesador secuencial se puede ejecutar en tiempo x, es de esperar que en una máquina multihilo con m hilos disponibles, esta aplicación potencialmente lo haga en un tiempo de x/m. Sin embargo, hay ciertos factores inherentes al tipo de arquitectura sobre la cual se ejecuta y al modelo de programación empleado que pueden limitar este incremento de rendimiento; por ejemplo, el acceso a los datos compartidos por hilos que se están ejecutando en diferentes núcleos.

Para obtener el máximo rendimiento de las arquitecturas sobre las cuales se ejecutan las aplicaciones paralelas, se tienen que considerar las características de estas arquitecturas. Por lo tanto, hay que considerar la jerarquía de memoria del sistema, el tipo de interconexión sobre el cual se envían datos, el ancho de banda de memoria, etc. Es decir, si se quiere extraer el máximo rendimiento hay que rediseñar o adaptar los algoritmos a las características del hardware que hay por debajo.

Si bien es cierto que hay que adaptar las aplicaciones según las arquitecturas en las que se quieren ejecutar, también es cierto que hay utilidades que permiten no tener en cuenta algunas de las complejidades de estas arquitecturas. La mayoría aparecen en forma de modelos de programación y bibliotecas que pueden ser empleadas por las aplicaciones.

Este apartado presenta los factores más importantes que pueden limitar el acceso al paralelismo que da una arquitectura ligados al modelo de programación.

5.1. Factores importantes para la ley de Amdahal en arquitecturas multihilo

La ley de Amdahal establece que una aplicación dividida en una parte inherentemente secuencial (es decir, solo puede ser ejecutada por un hilo) y una parte paralela *P*, potencialmente puede tener una mejora de rendimiento de *S* (en inglés *speedup*) aumentando el paralelismo de la aplicación a *P*.

$$T' = T * \frac{1}{\left(1 - P + \frac{P}{S}\right)}$$

Ahora bien, para llegar al máximo teórico hay que considerar las restricciones inherentes al modelo de programación y restricciones inherentes a la arquitectura sobre la cual se está ejecutando la aplicación.

El primer conjunto de restricciones hace referencia a los límites vinculados al algoritmo paralelo considerado, como también a las técnicas de programación empleadas. Un caso en el que aparecen estas restricciones es cuando se ordena un vector, puesto que hay limitaciones causadas por la eficiencia del algoritmo (por ejemplo, Radix Sort) y para implementarlo (por ejemplo, la manera de acceder a las variables compartidas, etc.).

El segundo conjunto hace referencia a límites ligados a las características del procesador sobre el cual se está ejecutando la aplicación. Factores como por ejemplo la jerarquía de memoria caché, el tipo de memoria caché o el tipo de red de interconexión de los núcleos pueden limitar este rendimiento.

Ejemplo

Puede causar muchas ineficiencias que una variable sea compartida entre dos hilos que no están dentro del mismo núcleo.

Los cercanos dos subapartados presentan algunos de los factores más importantes de estos dos bloques que acabamos de mencionar. Del primer conjunto no se estudian algoritmos paralelos (Gibbons y Rytte, 1988), puesto que no es el objetivo de la unidad, sino los mecanismos que usan estos algoritmos para implementar tareas paralelas y todos los factores que hay que considerar. Del segundo bloque se discuten las características más relevantes de la arquitectura que hay que considerar en el desarrollo de este tipo de aplicaciones.

Es importante remarcar que los factores que se estudian a continuación son una parte de los muchos que se han identificado durante las últimas décadas. Debido a la importancia de este ámbito, se ha hecho mucha búsqueda centrada a mejorar el rendimiento de estas arquitecturas y el diseño de las aplicaciones que se ejecutan (como las aplicaciones de cálculo numérico o las de cálculo del genoma humano).

Lecturas recomendadas

Para profundizar más en las problemáticas y estudios hechos tanto en el ámbito académico como empresarial, es muy recomendable extender la lectura de esta unidad didáctica con las referencias bibliográficas facilitadas.

5.2. Factores vinculados al modelo de programación

5.2.1. Definición y creación de las tareas paralelas

En el diseño de algoritmos paralelos se consideran dos tipos de paralelismo: a nivel de datos y a nivel de función. El primero define qué partes del algoritmo se ejecutan de manera concurrente, y el segundo, cómo se procesan los datos de manera paralela.

1) En la creación del **paralelismo a nivel de función** es importante considerar que las tareas que trabajen con las mismas funciones y datos tengan localidad en el núcleo en el que se ejecutarán. De este modo los flujos del mismo núcleo comparten las entradas correspondientes a la memoria caché de datos, como también sus instrucciones.

2) El **paralelismo a nivel de datos** también tiene que considerar la localidad mencionada, pero además tiene que tener en cuenta la medida de las memorias caché de que dispone. Es decir, el flujo de instrucciones tiene que trabajar con los datos de manera local en la L1 tanto como sea posible y el resto intentar mantenerla a niveles de memoria caché tan cercanos como sea posible (L2 o L3). Por otro lado, también hay que evitar efectos ping-pong entre los diferentes núcleos del procesador. Esto puede suceder cuando hilos de diferentes núcleos accedan a los mismos bloques de direcciones de manera paralela.

La figura siguiente muestra un ejemplo de escenario que se tiene que evitar en la creación de hilos, tanto en la asignación de tareas como en los datos. En este escenario, los dos hilos número 0 de ambos núcleos están ejecutando la misma función $F(x)$. Esta función contiene un bucle interno que hace algunos cálculos sobre el vector k y el resultado lo añade a la variable global k. Durante la ejecución de estos hilos se observa lo siguiente:

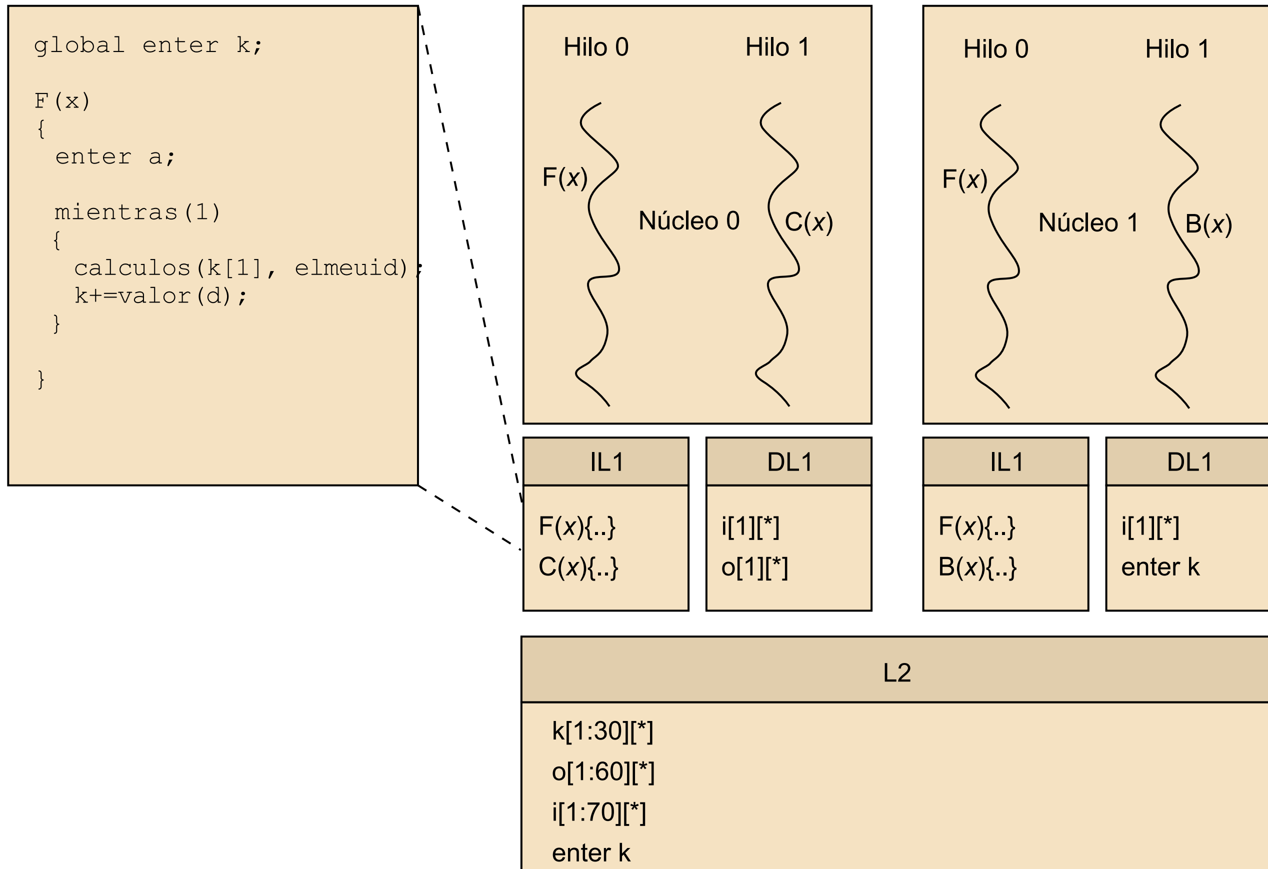
- Los dos núcleos cada vez que quieran acceder a la variable global k tienen que invalidar la L1 de datos del otro núcleo. Como ya se ha visto, dependiendo del protocolo de coherencia, puede ser extremadamente costoso. Por lo tanto, este aspecto puede hacer bajar el rendimiento de la aplicación.

- Los dos hilos acceden potencialmente a los datos del vector l. Así pues, las L1 de datos de ambos núcleos tienen almacenados los mismos datos (asumiendo que la línea se encuentra en estado compartido). En este caso sería más eficiente que los dos hilos se ejecutaran en el mismo núcleo para compartir los datos de la L1 de datos (es decir, más localidad). Esto permitiría usar más eficientemente el espacio total del procesador.

- De manera similar, la L1 de instrucciones de los dos casos de núcleos tienen una copia de las instrucciones del mismo código que ejecutan los dos

hilos (*F*). Del mismo modo, en su punto anterior este aspecto provoca una utilización ineficiente de los recursos, puesto que los mismos datos se encuentran replicados en las dos memorias caché. Hay que remarcar que, en este caso, los dos núcleos no invalidan las líneas compartidas, puesto que al generar las entradas de memoria de instrucciones, se encuentran en estado compartido y estas no se acostumbran a modificar.

Figura 18. Definición y creación de hilos



El ejemplo anterior muestra una situación bastante sencilla de solucionar. Ahora bien, la definición, la creación de tareas y la asignación de datos no es una tarea fácil. Como ya se ha mencionado, hay que considerar la jerarquía de memoria, la manera como los procesos se asignan a los núcleos, el tipo de coherencia que el procesador facilita, etc.

A pesar de la complejidad de esta tarea hay muchos recursos que ayudan a definir este paralelismo, como los siguientes:

- Aplicaciones que permiten la paralelización automática de aplicaciones secuenciales. Muchos compiladores incluyen opciones para generar código paralelo automáticamente. Ahora bien, es fácil encontrarse en situaciones en las que el código generado no es óptimo o no tiene en cuenta algunos de los aspectos introducidos.

Lecturas recomendadas

Para profundizar en este ámbito es recomendable leer:

X. Martorell; J. Corbalán; M. González; J. Labarta; N. Navarro; E. Ayguadé (1999). "Thread Fork/Join Techniques for Multe-level Parallelism Exploitation in NUMA Multiprocessors". *13th International Conference on Supercomputing*.

B. Chapman; L. Huang; E. Biscondi; E. Stotzer; A. G. Shrivastava (2008). "Implementing OpenMP on a High Performance Embedded Multicore MPSoC". *IPDPS*.

- Aplicaciones que dan asistencia en la paralelización de los códigos secuenciales. Por ejemplo, ParaWise (ParaWise, 2011) es un entorno que guía al usuario en la paralelización de código Fortran. En cualquier caso el resultado puede ser similar a la paralelización automática.

- Finalmente, también hay bibliotecas que proporcionan interfaces para implementar algoritmos paralelos. Estas interfaces acostumbran a ser la solución más eficaz para sacar el máximo rendimiento de las aplicaciones: facilitan el acceso a funcionalidades que permiten definir el paralelismo a nivel de datos y funciones, afinidades de hilos a núcleos, afinidades de datos a la jerarquía de memoria, etc. Algunas de estas bibliotecas son OpenMP, Cilck (Intel, Intel Cilk Plus, 2011), TBB (Reinders, 2007), etc.

5.2.2. Mapeo de tareas a hilos

Una tarea es una unidad de concurrencia de una aplicación, es decir, incluye un conjunto de instrucciones que pueden ser ejecutadas de manera concurrente (paralela o no) a las instrucciones de otras tareas. Un hilo es una abstracción del sistema operativo que permite ejecutar paralelamente diferentes flujos de ejecución. Una aplicación puede estar formada desde un número relativamente pequeño de tareas hasta miles.

Tareas

Ejemplos claros son los servidores de videojuegos o los servidores de páginas web: pueden tener miles de tareas atendiendo a las peticiones de los usuarios.

Sin embargo, el sistema operativo no puede dar acceso a un número tan elevado de hilos de ejecución por la limitación de los recursos. Por un lado, la gestión de un número tan elevado es altamente costoso (muchos cambios de contextos, gestión de muchas interrupciones, etc.). Por otro lado, a pesar de que potencialmente el sistema operativo pueda dar acceso a un millar de hilos, el procesador sobre el cual se ejecutan las aplicaciones da acceso a pocos hilos físicos[16]. Por lo tanto, hay que ir haciendo cambios de contexto entre todos los hilos disponibles a nivel de sistema y los hilos que el hardware facilite. Este es el motivo por el cual el número de hilos del sistema tiene que ser configurado coherentemente con el número de hilos del procesador.

(16)En inglés llamados *hardware threads*.

La aplicación, para sacar el máximo rendimiento del procesador, tiene que definir un mapeo eficiente y adecuado de las tareas que quiere ejecutar en los diferentes hilos de que dispone. Algunos de los algoritmos de mapeo de tareas en los hilos más empleados son los siguientes:

- Patrón *master/slave*: un hilo se encarga de distribuir las tareas que quedan para ejecutar a los hilos esclavos que no estén ejecutando nada. El número de hilos esclavos es variable.

- Patrón *pipeline* o cadena: donde cada uno de los hilos hace una operación específica en los datos que se están procesando, a la vez que facilita el resultado al hilo siguiente de la cadena.

- Patrón *task pool*: donde hay una cola de tareas pendientes de ser ejecutadas y cada uno de los hilos disponibles coge una de estas tareas pendientes cuando acaba de procesar el actual.

Este mapeo se puede hacer basándose en muchos criterios diferentes. Sin embargo, los aspectos introducidos a lo largo de esta unidad didáctica se tendrían que considerar en arquitecturas multihilo heterogéneas. Dependiendo de cómo se asignen las tareas a los hilos y cómo estén asignados los hilos a los núcleos, el rendimiento de las aplicaciones varía mucho.

Lectura recomendada

Un trabajo de investigación muy interesante sobre las técnicas de mapeo es el presentado por Philbin y otros. Los autores presentan técnicas de mapeo y gestión de hilos para mantener la localidad en las diferentes memorias caché:

J. Philbin; J. Edler; O. J. Anshus; C. Douglas; K. Li (1996). "Thread scheduling for cache locality". *Seventh international conference on Architectural support for programming languages and operating systems*.

Ejemplo de mapeo de tareas

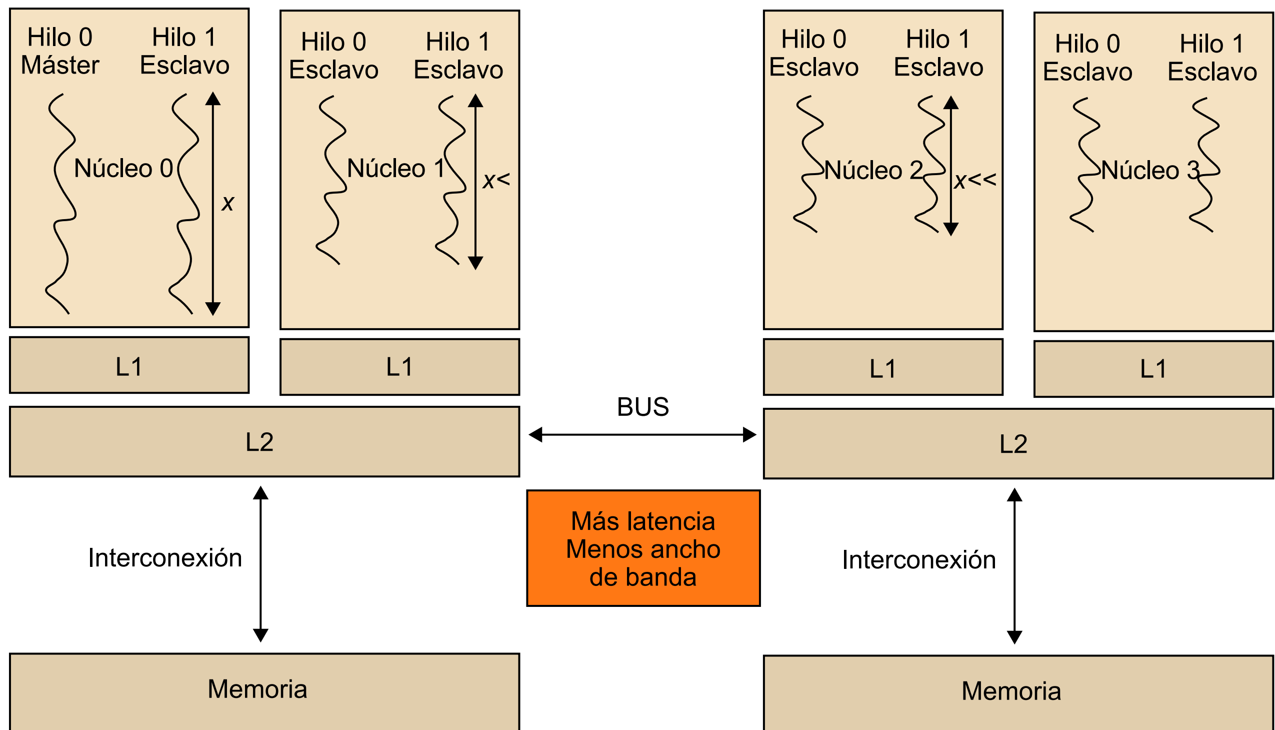
La figura 19 presenta un ejemplo de esta situación. Una aplicación multihilo se ejecuta sobre una arquitectura compuesta por dos procesadores. Cada uno con acceso a memoria y los dos conectados por un bus. Los hilos que se están ejecutando en el núcleo 2 y en el núcleo 3 acceden a los datos y a la información que el máster les facilita.

Respecto de una red de interconexión local, el bus acostumbra a tener una latencia más elevada y menos ancho de banda. Este es el motivo por el cual los hilos que se ejecutan en el núcleo 2 y en el núcleo 3 tienen cierto desbalanceo respecto de los que lo hacen en el núcleo 0 y en el núcleo 1. Estos factores hay que considerarlos a la hora de decidir cómo asignar las tareas y el trabajo en cada uno de los hilos.

En el ejemplo considerado, el rendimiento del procesador y de la aplicación es mucho menor del que potencialmente se podría lograr. En el instante de tiempo *T*, el hilo esclavo del núcleo 1 ha acabado de hacer el trabajo *X*. Los hilos 0 y 1 del núcleo 1 tardan un cierto tiempo (*T'*) más en finalizar su tarea. Y los hilos de los núcleos 2 y 3 tardan un tiempo (*T''*) bastante mayor que *T* hasta acabar. Por lo tanto, los núcleos 0 y 1 no se usan durante *T* – *T''* y *T '* – *T''* respectivamente.

No solo la utilización del procesador es más baja, sino que este desbalanceo hace finalizar la aplicación más tarde. En este caso, a la hora de diseñar el reparto de tareas hay que tener en cuenta en qué arquitectura se ejecuta la aplicación, como también cuáles son los elementos que potencialmente pueden causar desbalanceos y qué hilos pueden hacer más trabajo por unidad de tiempo.

Figura 19. Patrón *master/slave* en un multinúcleo con dos procesadores



5.2.3. Definición e implementación de mecanismos de sincronización

A menudo las aplicaciones multihilo usan mecanismos para sincronizar las tareas que los diferentes hilos están llevando a cabo. Un ejemplo lo encontramos en la figura del ejemplo anterior (figura 19), en el que el hilo máster se espera que los esclavos acaben mediante funciones de espera.

En general se acostumbran a usar tres tipos de mecanismos diferentes de sincronización:

- El uso de variables para controlar el acceso a determinadas partes de la aplicación o a determinados datos de la aplicación (como un contador global). Ejemplos de este tipo de variables lo son las de los semáforos o las de exclusión mutua.

- El uso de barreras para controlar la sincronización entre los diferentes hilos de ejecución. Estas nos permiten asegurar que todos los hilos no pasan de un determinado punto hasta que todos han llegado.

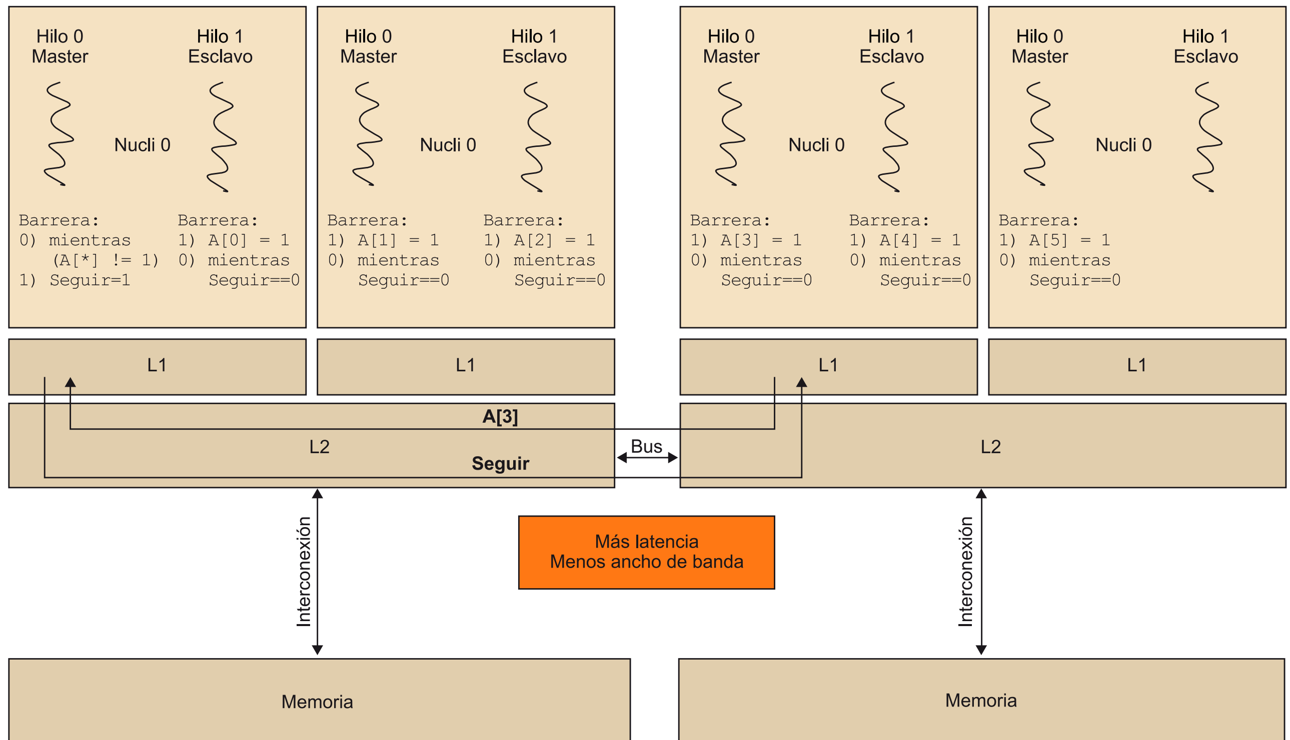
- El uso de mecanismos de creación y espera de hilos. Como el ejemplo anterior, el hilo máster espera la finalización de los diferentes hilos esclavos mediante llamamientos a funciones de espera.

Desde el punto de vista de arquitecturas multihilo/núcleo, el segundo y el tercer punto son menos intrusivos al rendimiento de la aplicación (Villa, Palermo y Silvano, 2008). Como se verá a continuación, las barreras son mecanismos que se emplean solo en ciertas partes de la aplicación y que se pueden implementar de manera eficiente dentro de un multihilo/núcleo. En cambio, un uso excesivo de variables de control puede provocar un descenso significativo en el rendimiento de las aplicaciones.

Barreras en arquitecturas multinúcleo

La figura 20 presenta un ejemplo de posible implementación de barrera y de cómo se comportaría en un multinúcleo. En este caso, un hilo se encarga de controlar el número de núcleos que han llegado a la barrera. Hay que hacer notar que el núcleo 0 para acceder al vector A tiene todos estos datos en la L1, en estado compartido o en estado exclusivo.

Figura 20. Funcionamiento de una barrera en un multinúcleo



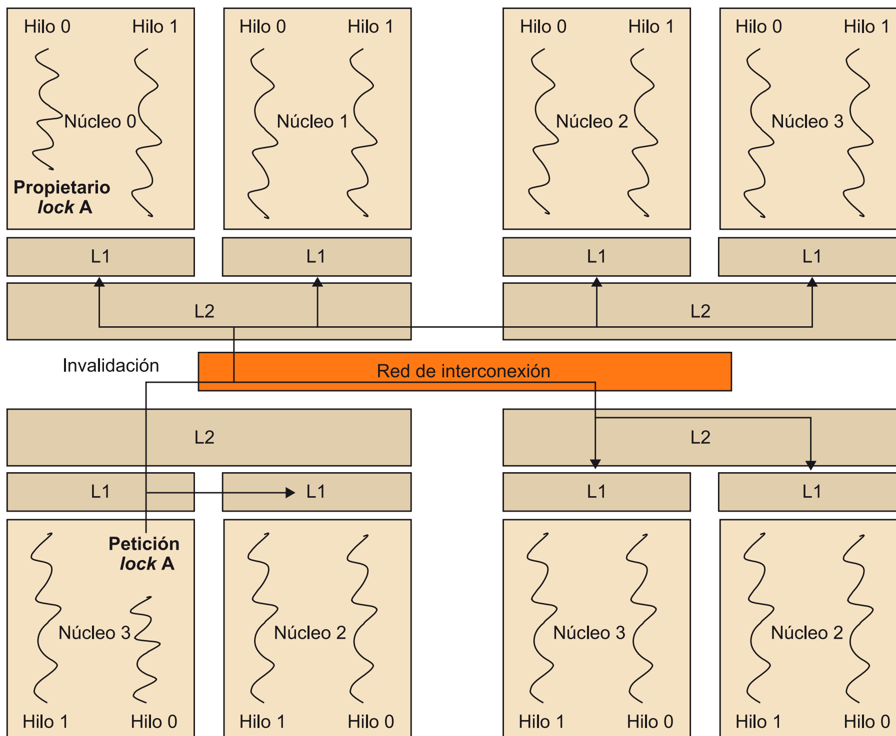
Cada vez que un hilo llega a la barrera, quiere modificar la posición correspondiente del vector. Por lo tanto, invalida la línea correspondiente (la que contiene A[i]) de todos los núcleos y la modifica. Cuando el núcleo 0 vuelve a leer la dirección de A[i], tiene que pedir el valor al núcleo i. Según el tipo de protocolo de coherencia que implemente el procesador, el núcleo 0 invalida la línea del núcleo i o bien la mueve a estado compartido.

Como se puede deducir de este ejemplo, el rendimiento de una barrera puede ser más o menos eficiente según su implementación y la arquitectura sobre la cual se está ejecutando. Así, una implementación en la que en lugar de un vector hay una variable global que cuenta los que han acabado sería más ineficiente, puesto que los diferentes núcleos tendrían que competir para coger la línea, invalidar los otros núcleos y escribir el valor nuevo.

Mecanismos de exclusión mutua en arquitecturas multinúcleo

Estos mecanismos se emplean para poder acceder de manera exclusiva a ciertas partes del código o a ciertas variables de la aplicación. Son necesarios para mantener el acceso coordinado a los recursos compartidos y evitar condiciones de carreras, *deadlocks* o situaciones similares. Algunos de estos tipos de recursos son semáforos, *mutex-locks* o *read-writerlocks*.

Figura 21. Adquisición de una variable de exclusión mutua



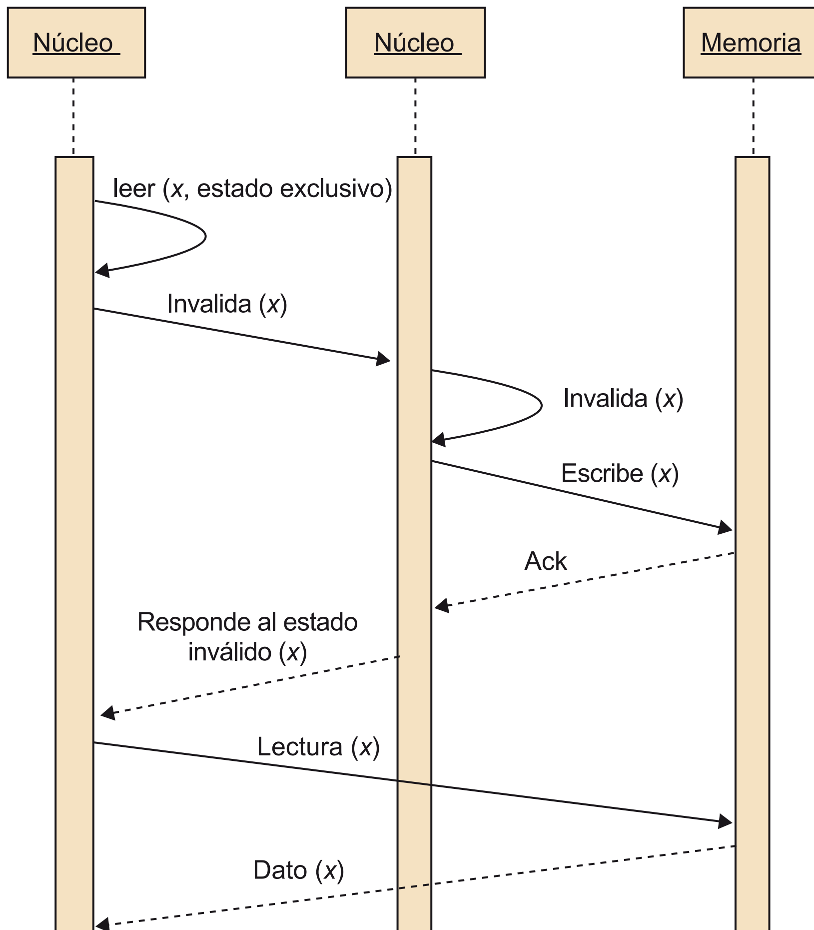
En arquitecturas de un solo núcleo, el acceso a estos tipos de estructuras puede tener un impacto relativamente menor. Con alta probabilidad todos los hilos están compartiendo los accesos a las mismas líneas de la L1 que las guardan. Sin embargo, en un sistema multinúcleo el acceso concurrente de diferentes hilos a estas estructuras puede llevar a problemas de escalabilidad y de rendimientos graves. De manera similar a lo que se ha mostrado con las barreras, el acceso a las líneas de memoria que contienen las variables empleadas para gestionar la exclusión mutua implica invalidaciones y movimientos de líneas entre los núcleos del procesador.

La figura 21 muestra un escenario en el que el uso frecuente de acceso a zonas de exclusión mutua entre los diferentes hilos puede reducir sustancialmente el rendimiento de la aplicación. En este ejemplo se asume un protocolo de coherencia entre los diferentes núcleos de tipo *snoop*; por lo tanto, cada vez que uno de los hilos de un núcleo quiere coger la propiedad de la variable de exclusión mutua *(lock)* tiene que invalidar todo el resto de núcleos. Hay que hacer notar que la línea de la memoria caché que contiene la variable en cuestión se encuentra en estado modificado cada vez que el núcleo lo actualice. Siempre que un hilo coge el *lock*, modifica la variable para marcarla como propia. En el supuesto de que el hilo solo lo esté consultando, no habría que invalidar los otros núcleos.

Se puede asumir que la variable se reenvía del núcleo que la tiene (el núcleo 0) al núcleo que la pide (el núcleo 3) en estado modificado. Ahora bien, dependiendo del tipo de protocolo de coherencia que implementara el procesador, la línea se escribiría primero en memoria i, entonces, el núcleo 3 la podría leer. En este caso el rendimiento sería extremadamente bajo: por cada lectura

del *lock x*, se invalidarían todos los núcleos; así, el que la tuviera en estado modificado lo escribiría en memoria y finalmente el núcleo que lo estuviera pidiendo la leería de memoria (figura 22).

Figura 22. Lectura del *lock x* en estado exclusivo



Como se ha podido ver, es recomendable un uso moderado de este tipo de mecanismos en arquitecturas con muchos núcleos y también en arquitecturas heterogéneas. Así pues, a la hora de decidir qué tipo de mecanismos de sincronización se usan, hay que considerar la arquitectura sobre la cual se ejecuta la aplicación (jerarquía de memoria, protocolos de coherencia e interconexiones entre núcleos) y cómo se implementan todos estos mecanismos. Por otro lado, también es recomendable reducir al máximo la cantidad de datos que comparten los diferentes núcleos. De este modo se disminuye la cantidad de mensajes de protocolo de coherencia que se envían entre núcleos y el número de invalidaciones y *snoops* que tienen que procesar los núcleos.

Lectura recomendada

Para profundizar en la creación de mecanismos de exclusión mutua escalables en arquitecturas multinúcleo se recomienda leer el artículo:

M. Chynoweth; M. R. Lee (2009). "Implementing Scalable Atomic Locks for Multe-Core".

Uno de los puntos más importantes cuando se hace el diseño de arquitecturas multinúcleo eficientes, para aplicaciones de supercomputación o HPC, es precisamente que faciliten mecanismos para poder implementar de manera eficiente barreras entre todos los hilos de las aplicaciones. Dependiendo del número de núcleos y de los mecanismos que facilite el procesador, una barrera puede tardar desde un centenar de ciclos de procesador hasta miles de ciclos.

Lectura recomendada

Los autores del artículo siguiente hablan de implementaciones escalables de mecanismos de sincronización entre hilos:

J. M. Mellor-Crummey; M. L. Scott (1991). "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems*.

5.2.4. Gestión de acceso concurrente a datos

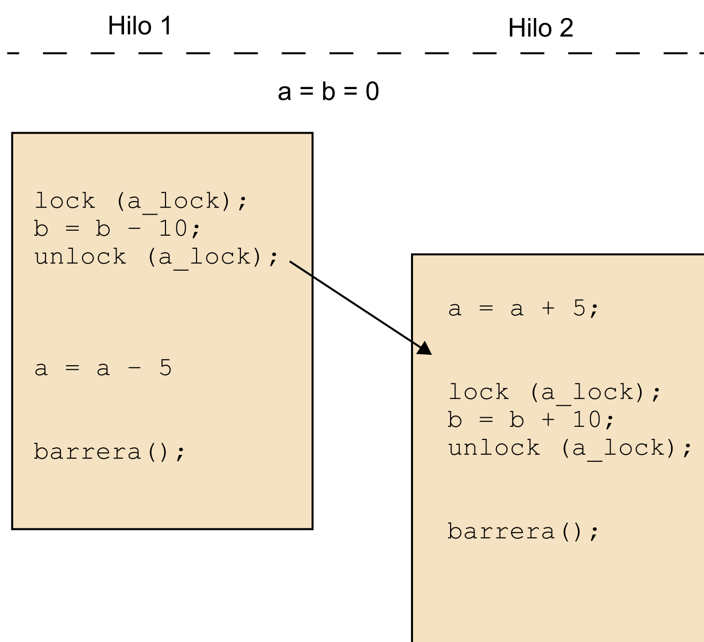
Acabamos de ver las implicaciones que tiene el hecho de usar diferentes técnicas de exclusión mutua en arquitecturas multiprocesador. Este tipo de técnicas son empleadas para asegurar que el acceso a los datos entre diferentes hilos es controlado. Si no se usan estas técnicas de manera adecuada, las aplicaciones pueden acabar teniendo carreras de acceso[17].

(17)En inglés, *race accesses*.

En general se pueden distinguir dos tipos de carreras de acceso que hay que evitar cuando se desarrolla un código paralelo:

1) Carreras de acceso a datos: Suceden cuando diferentes hilos están accediendo de manera paralela a las mismas variables sin ningún tipo de protección. Para que haya una carrera de este tipo, uno de los accesos tiene que ser en forma de escritura. La figura 23 muestra un código que potencialmente puede tener un acceso a carrera de datos. Suponiendo que los dos hilos se están ejecutando en el mismo núcleo, cuando los dos hilos llegan a la barrera, ¿qué valor tiene *a*? Como el acceso a esta variable no se encuentra protegido y se accede tanto en modo lectura como en escritura (añadiendo 5 y –5), esta variable puede tener los valores siguientes: 0, 5 y –5.

Figura 23. Carrera de acceso a datos



2) Carreras de acceso general: Este tipo de carreras suceden cuando dos hilos diferentes tienen que seguir un orden específico en la ejecución de diferentes partes de su código, pero no tenemos estructuras que fuercen este orden. El ejemplo siguiente muestra uno de estos tipos de carreras. Los dos hilos usan una estructura guardada en memoria compartida en la que el primer hilo pone un trabajo y el segundo lo procesa. Como se puede observar, si no se añade ningún tipo de control o variable de control, es posible que el hilo 2 se acabe de ejecutar sin procesar el trabajo a..

Ejemplo carrera de acceso general

```
Hilo 1:
Trabajo a = nuevo_trabajo();
Configurar_Trabajo(a);
EncolaTrabajo(a,Cola);
PostProceso();
Hilo 2:
Trabajo b  = CogeTrabajo(Cola);
si(b != INVALID)
ProcesaTrabajo(b);
Acaba();
```

Carrera de acceso a datos

Hay que mencionar que una carrera de acceso a datos es un tipo específico de carrera de acceso general. Los dos tipos pueden ser evitados empleando los mecanismos de acceso introducidos en el anterior subapartado (barreras, variables de exclusión mutua, etc.). Siempre que se diseñe una aplicación multihilo hay que considerar que los accesos en modo escritura y lectura a partes de memorias compartidas tienen que estar protegidos; si los diferentes hilos asumen un orden específico en la ejecución de diferentes partes del código hay que forzarlo vía mecanismos de sincronización y espera.

El primero de los dos ejemplos presentados (figura 23) se puede evitar añadiendo una variable de exclusión mutua que controle el acceso a la variable *a*.De este modo, independientemente de quien acceda primero a modificar el valor de la variable, el valor de esta una vez se llega a la barrera es 0. El segundo de los ejemplos de la figura 23 podría ser evitado con mecanismos de espera entre hilos, es decir, como muestra el ejemplo siguiente, el segundo hilo tendría que esperar que el primer hilo notifique que le ha facilitado el trabajo.

Evitar la carrera de acceso mediante sincronización

```
Fil 1:
Trabajo a = nuevo_trabajo();
Configurar_Trabajo(a);
EncolaTrabajo(a,Cua);
NotificaTrabajoDisponible();
PostProceso();
Hilo 2:
EsperaTrabajo();
Trabajo b  = CogeTrabajo(Cola);
si(b != INVALID)
ProcesaTrabajo(b);
Acaba();
```

Condiciones de carrera en arquitecturas multinúcleos

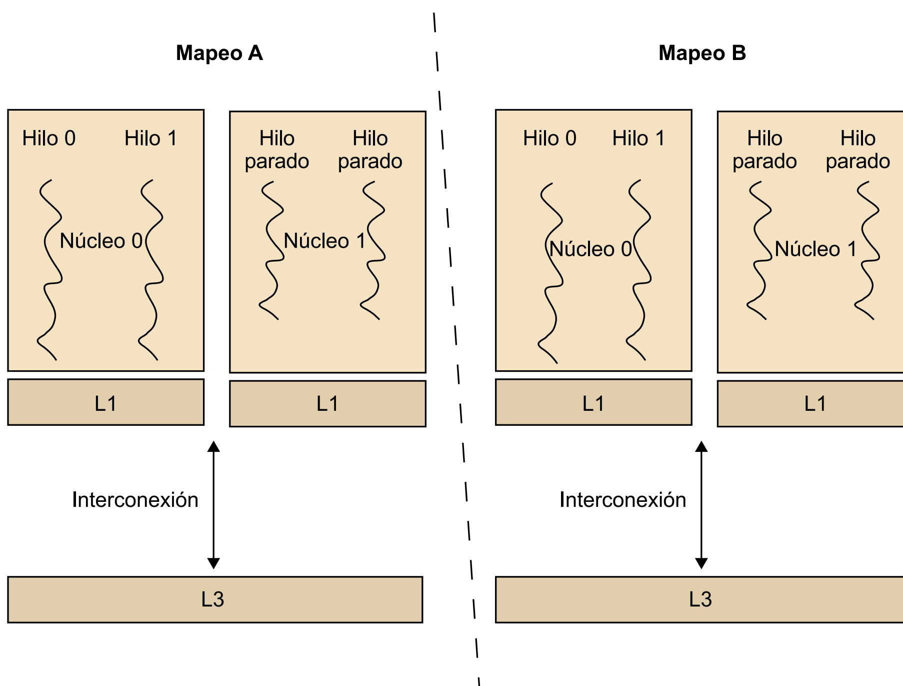
En el subapartado anterior se han introducido varias situaciones en las que el uso de memoria compartida entre diferentes hilos es inadecuado. En primer lugar, las carreras de acceso a datos, se da cuando dos hilos diferentes están le-

yendo y escribiendo de manera descontrolada en una zona de memoria. Se ha mostrado cómo el valor de una variable puede ser funcionalmente incorrecto cuando los dos hilos finalizan sus flujos de instrucciones. Ahora bien, esta cadena de acontecimientos ¿sucede independientemente de la arquitectura y del mapeo de hilos sobre el cual se ejecuta la aplicación?

En este subapartado se quiere mostrar cómo el mismo código se puede comportar de manera diferente dentro de un mismo procesador según como se asignen los hilos a los núcleos, puesto que, dependiendo de este aspecto, la condición de carrera analizada en el subapartado anterior sucede o no.

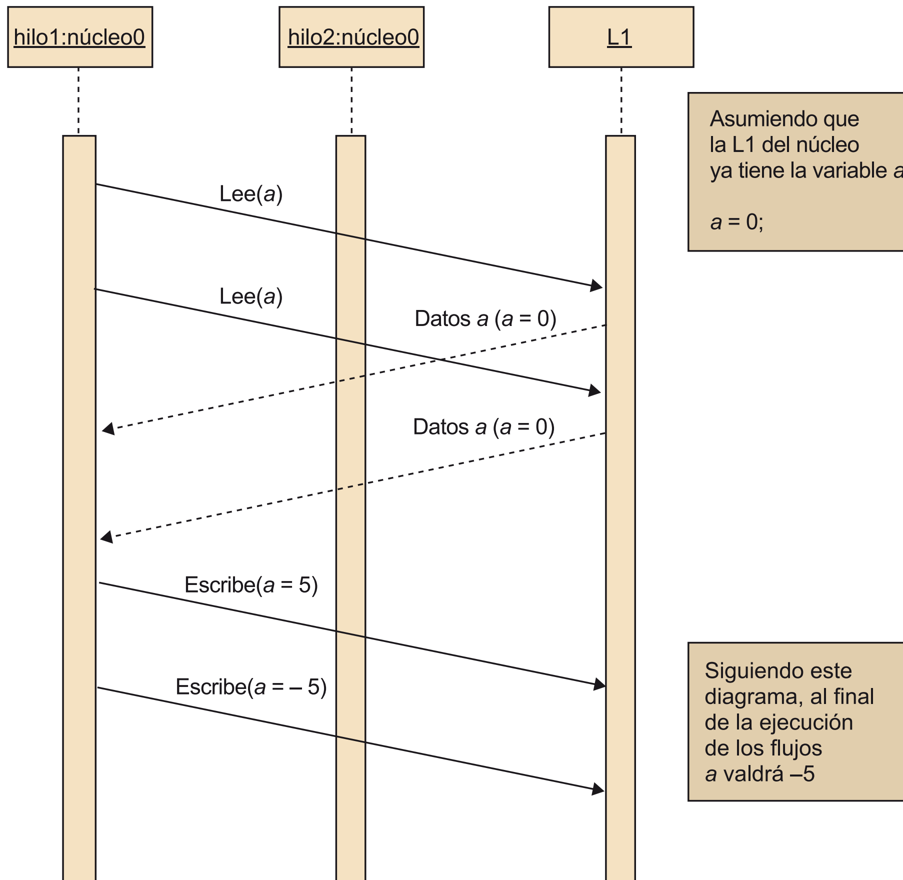
Suponemos los dos escenarios que muestra la figura siguiente:

Figura 24. Dos mapeos de hilos diferentes ejecutando el ejemplo de carrera de acceso general



En el primero de los dos escenarios, los dos hilos se están ejecutando en el mismo núcleo. Tal como muestra la figura 25, los dos hilos acceden al contenido de la variable *a de* la misma memoria caché de nivel dos. Primero el hilo 1 lee el valor.

Figura 25. Acceso compartido a *a* en un mismo núcleo



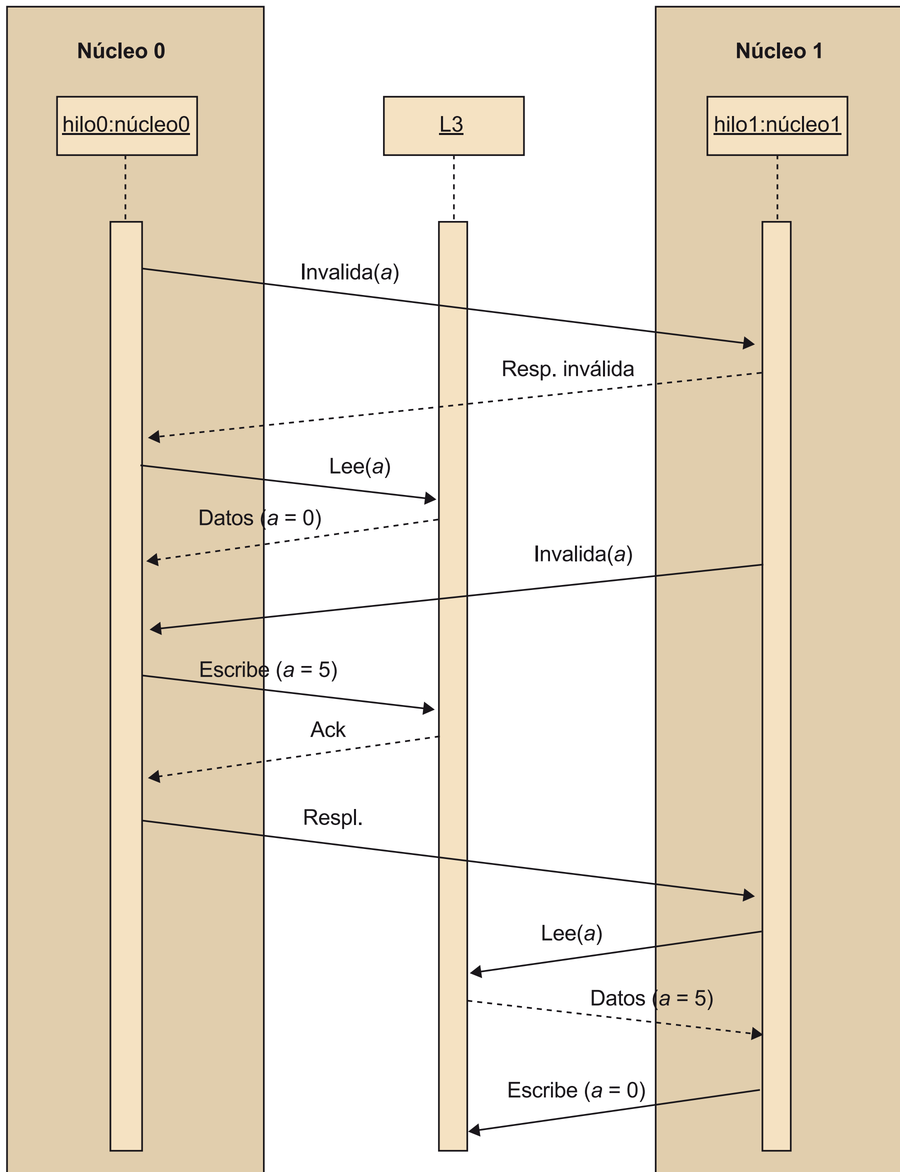
A continuación, a pesar de que el hilo 1 ya está modificando la variable, el segundo hilo lee el valor original. Finalmente, los dos hilos escriben los valores en memoria. Sin embargo, el segundo hilo sobrescribe el valor actual (5) por el valor resultante de la operación aritmética que el hilo ha aplicado (–5). Este flujo de acontecimientos, como ya se ha visto, es funcionalmente incorrecto, es decir, el resultado de la ejecución de los diferentes hilos no es el esperado por la aplicación en cuestión.

Asumimos ahora el segundo de los escenarios, en el que cada uno de los hilos se ejecuta en un núcleo diferente. En las arquitecturas de procesadores con memoria coherente, si dos núcleos diferentes están accediendo en un mismo momento a una misma línea de memoria, el protocolo de coherencia asegura que solo un núcleo puede modificar el valor de esta línea. Por lo tanto, en un instante de tiempo tan solo un núcleo puede tener la línea en estado exclusivo para modificarla.

En este escenario nuevo, y gracias al protocolo de coherencia, la carrera de acceso mostrada en el caso anterior no sucede. Como muestra la figura 26, el protocolo de coherencia protege el acceso en modo exclusivo a la variable *a*.Cuando el primer hilo quiere leer el valor de la variable en modo exclusivo para ser modificado, invalida todas las copias de los núcleos del sistema (en

este caso solo uno). Una vez el núcleo 0 notifica que tiene la línea en estado inválido, pide el valor a la memoria caché de tercer nivel. Para simplificar el ejemplo, suponemos que modifica el valor de *a* tan pronto como la recibe.

Figura 26. Acceso compartido a *a* en núcleos diferentes



A continuación, el hilo 0 que se está ejecutando en el núcleo 1, para coger la línea en modo exclusivo, invalida la línea de los otros núcleos del sistema (núcleo 0). Cuando el núcleo 0 recibe la petición de invalidación, este valida el estado. Como se encuentra en estado modificado, escribe el valor en la memoria caché de tercer nivel. Una vez recibe el *acknowlegement*[18], responde al núcleo 1 que la línea se encuentra en estado inválido. Llegado a este punto, el núcleo 1 pide el contenido a la L3, recibe la línea, la modifica y la escribe de vuelta con el valor correcto.

(18)De ahora en adelante *ack*.

Por un lado, es importante remarcar que si bien en este caso no se da la carrera de acceso, la implementación paralela sigue teniendo un problema de sincronización. Dependiendo de qué tipo de mapeo se aplique, en los hilos de la aplicación se encuentra el error ya estudiado.

Por otro lado, hay que tener presente que, dependiendo de qué tipo de protocolo de coherencia implemente el procesador, el comportamiento de la aplicación podría variar. Por este motivo, emplear las técnicas de sincronización para hacer que la ejecución sea determinista y no ligada a factores arquitectónicos o de mapeo es realmente relevante.

5.2.5. Otros factores que hay que considerar

Este subapartado presenta diferentes factores que hay que considerar a la hora de diseñar, desarrollar y ejecutar aplicaciones paralelas en arquitecturas multihilo, como también las características principales y las implicaciones que estas aplicaciones tienen sobre las arquitecturas multihilo.

Como ya se ha mencionado, el diseño de aplicaciones paralelas es un campo en el que se ha hecho mucha investigación (tanto académica como industrial). Es, por lo tanto, aconsejable profundizar en algunas de las referencias facilitadas. Otros factores que no se han mencionado, pero que también son importantes, son los siguientes:

- Los *deadlocks*. Suceden cuando dos hilos se bloquean esperando un recurso que tiene el otro. Los dos hilos quedan bloqueados para siempre, por lo tanto bloquean la aplicación (Kim y Jun, 2009).

- Composición de los hilos paralelos. Es decir, la manera como se organizan los hilos de ejecución. Esta organización depende del modelo de programación (como OpenMP o MPI) y de cómo se programa la aplicación (por ejemplo, depende de si los hilos están gestionados por la aplicación con *PosixThreads*).

- Escalabilidad del diseño. Factores como por ejemplo el número de hilos, baja concurrencia en el diseño o bien demasiada contención en accesos a los mecanismos de sincronización pueden reducir sustancialmente el rendimiento de la aplicación.

Lectura recomendada

Para profundizar en la escalabilidad del diseño, se recomienda la lectura siguiente:

S. Prasad (1996). *Multithreading Programming Techniques*. Nueva York: McGraw-Hill, Inc.

5.3. Factores ligados a la arquitectura

Este subapartado presenta algunos factores que pueden impactar en el rendimiento de las aplicaciones paralelas que no están directamente ligadas al modelo de programación. Como se verá a continuación, algunos de estos factores aparecen dependiendo de las características del procesador multihilo sobre el

cual se ejecuta la aplicación. Así pues, trataremos algunos de los factores más importantes que hay que tener en cuenta cuando se desarrollan aplicaciones paralelas para arquitecturas multihilo.

Por un lado, la compartición de recursos entre diferentes hilos puede llevar a situaciones de conflictos que degradan mucho el rendimiento de las aplicaciones. Un caso de compartición es el de las mismas entradas de una memoria caché.

Por otro lado, el diseño de las arquitecturas multihilo implica ciertas restricciones que hay que considerar al implementar las aplicaciones. Por ejemplo, un procesador multinúcleo tiene un sistema de jerarquía de memoria en el que los niveles inferiores (por ejemplo, la L3) se comparten entre diferentes hilos y los superiores se encuentran separados por el núcleo (por ejemplo, la L1). Los fallos en los niveles superiores son bastante menos costosos que en los niveles inferiores. Sin embargo, la medida de estas memorias es bastante menor, por lo tanto, hay que adaptar las aplicaciones para que tengan el máximo de local posible en los niveles superiores.

Este subapartado está centrado en aspectos ligados a los protocolos de coherencia y gestión de memoria, a pesar de que hay otros muchos factores que hay que considerar si se quiere sacar el máximo rendimiento del algoritmo que se está diseñando.

Ejemplo

Optimizaciones del compilador (Lo, Eggers, Levy, Parekh y Tullsen, 1997), características de las memorias caché (Hily y Seznec, 1998) o el juego de instrucciones del procesador (Kumar, Farkas, Jouppi, Ranganathan y Tullsen, 2003).

5.3.1. Compartición falsa

En muchas situaciones se quiere que diferentes hilos de la aplicación compartan zonas de memoria concretas. Como hemos visto, a menudo se dan situaciones en las que diferentes hilos comparten contadores o estructuras donde se guardan datos resultantes de cálculos hechos por cada uno y donde se usan variables de exclusión mutua para proteger estas zonas.

Los datos que los diferentes hilos están usando en un instante de tiempo concreto se guardan en las diferentes memorias caché de la jerarquía (desde la memoria caché de último nivel, hasta la memoria caché de primer nivel del núcleo que lo está usando). Por lo tanto, cuando dos hilos están compartiendo un dato, también comparten las mismas entradas de las diferentes memorias caché que guardan este dato.

Compartición de entradas

En la figura 25, los dos hilos acceden a la misma entrada de la L1 que guarda la variable a.

Desde el punto de vista de rendimiento, lo que se busca es que los accesos de los diferentes hilos acierten alguna de las memorias caché (cuanto más cercana al núcleo mejor), puesto que el acceso a memoria es muy costoso. A esto se le llama mantener la localidad en los accesos (Grunwald, Zorn y Henderson, 1993).

Ahora bien, hay situaciones en las que las mismas entradas de la memoria caché las comparten datos diferentes. El cálculo de qué entrada de una memoria caché se usa se define según la asociatividad y el tipo de mapeo de la memoria (Handy, 1998).

Memoria 2-asociativa

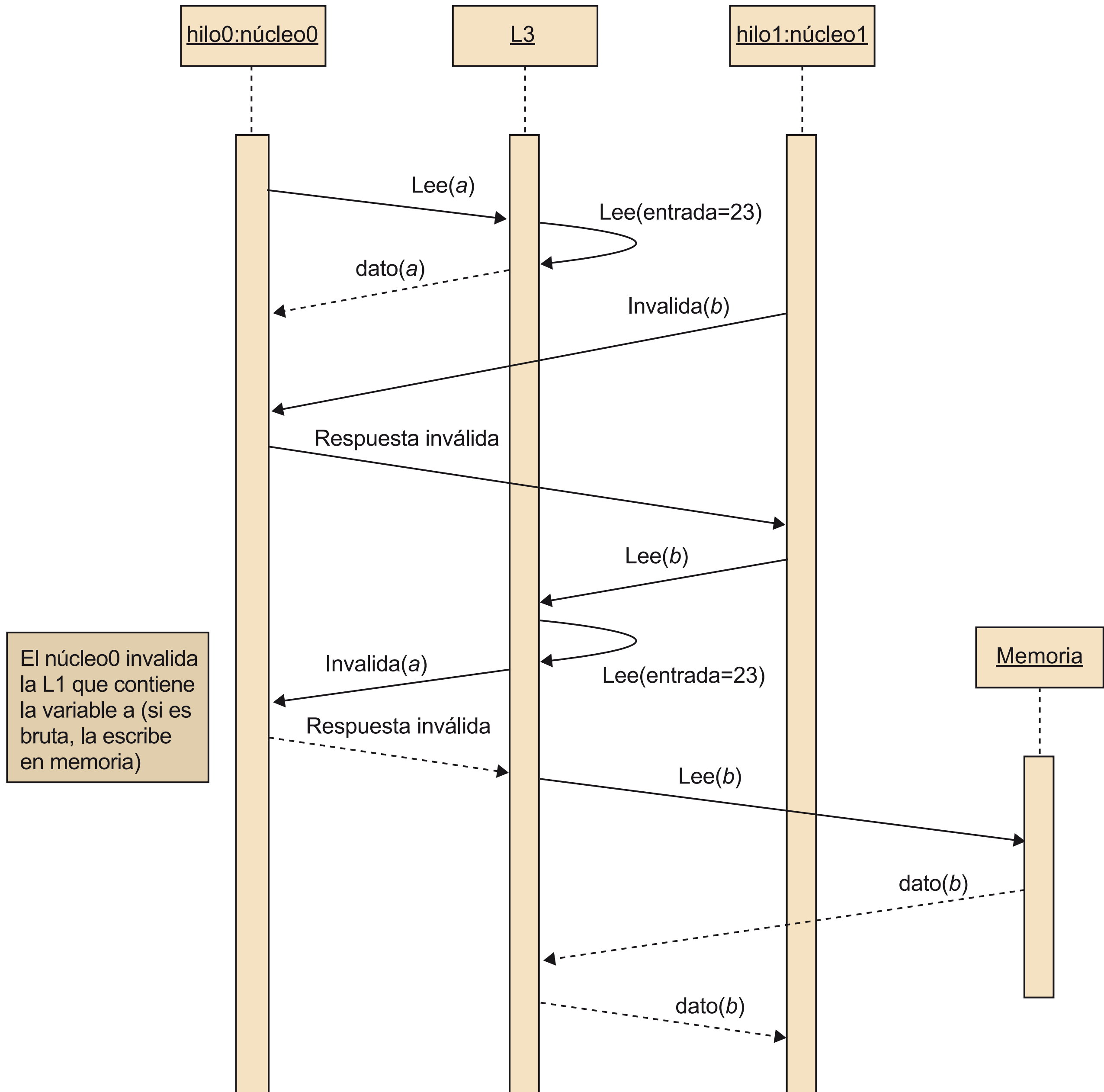
Por ejemplo, una memoria 2-asociativa (Seznec, 1993) divide la memoria caché en conjuntos de dos entradas. Primero se calcula a cuál de los conjuntos pertenece la dirección y después se escoge cuál de las dos entradas conjunto se usa.

Puede pasar que dos hilos accedan a dos bloques de memoria diferentes, que se mapeen en el mismo conjunto de una memoria caché. En este caso, los accesos de un hilo a su bloque de memoria invalidan los datos del otro hilo guardados en las mismas entradas de la memoria. A pesar de que las direcciones coinciden en las mismas entradas de la memoria, los datos y las direcciones son diferentes. Por lo tanto, para cada acceso se invalidan los datos del otro hilo y se pide el dato al siguiente nivel de la jerarquía de memoria (por ejemplo, la L3 o la memoria principal). Como se puede deducir, este aspecto implica una reducción importante en el rendimiento de la aplicación. Esto se denomina *compartición falsa*[19].

(19) En inglés, *false sharing.*

La figura siguiente muestra un ejemplo de lo que podría pasar en una arquitectura multinúcleo en la que dos hilos de núcleos diferentes están experimentando *false sharing* en el mismo set de la L3. Para cada acceso de uno de los hilos se invalida una línea del otro hilo, que es guardada en el mismo conjunto. Como se observa, para cada acceso se genera un número importante de acciones: invalidación de dirección en el otro núcleo, lectura en la L3, se victimiza el dato del otro hilo del otro núcleo y se escribe en memoria si es necesario, se lee el dato en memoria y se envía al núcleo que la ha pedido.

Figura 27. Compartición falsa en la L3



En el supuesto de que los dos hilos no tengan conflicto en los mismos conjuntos de la L3, con una probabilidad bastante alta aciertan en la L3 y se ahorran el resto de transacciones que suceden por culpa de la compartición falsa.

La compartición falsa se puede detectar cuando una aplicación muestra un rendimiento muy bajo y el número de invalidaciones de una memoria caché concreta y de misos es mucho más elevado de lo que se esperaba. En este caso, es muy probable que dos hilos compitan por los mismos recursos de la memoria caché. Hay herramientas como el Vtune de Intel (Intel, *boost performance optimization and multicore scalability*, 2011) que permiten detectar este tipo de problemas.

Evitar la compartición falsa es relativamente más sencillo de lo que puede parecer. Una vez se han detectado cuáles son las estructuras que probablemente causan este efecto, hay que añadir un desplazamiento para que caigan en posiciones de memoria diferentes para cada uno de los hilos. De este modo los datos que antes compartían un mismo set de la memoria caché, esta vez se guardan en sets diferentes.

5.3.2. Penalizaciones para fallos en la L1 y técnicas de *prefetch*

Muchas aplicaciones paralelas tienen una alta localidad en las memorias caché del núcleo sobre las cuales se ejecutan. Es decir, la mayoría de accesos que se hacen a la memoria aciertan la memoria caché de primer nivel.

Ahora bien, los casos en los que las peticiones en memoria no aciertan la memoria caché del núcleo tienen que hacer un proceso mucho más largo hasta que el dato está disponible para el hilo: petición en la L3, fallo en la L3, petición en memoria, etc. Evidentemente, cuanto más alto es el porcentaje de fallos, más penalizada se encuentra la aplicación. La causa es que un acceso a memoria que falla en la L1 tiene una latencia mucho más elevada que uno que lo acierta (uno o dos órdenes de magnitudes más elevados, dependiendo de la arquitectura y protocolo de coherencia).

Una de las técnicas que se usa para mirar de esconder la latencia más larga de las peticiones que fallarán en la memoria caché se denomina *prefetching*. Esta técnica consiste en pedir el dato a la memoria mucho antes de cuando la aplicación la necesita. De este modo, cuando realmente la necesita, esta ya se encuentra en la memoria local del núcleo (L1). Por lo tanto, lo acierta, la latencia de la petición en memoria es mucho más baja y la aplicación no se bloquea.

Hay dos tipos de técnicas de *prefetching* usadas en las arquitecturas multinúcleo:

1) *Hardware prefetching*. Es una técnica que se implementa en las piezas de los núcleos que se denominan *hardware prefetchers*. Estos intentan predecir cuáles son las próximas direcciones que pedirá la aplicación y las piden de manera proactiva antes de que la aplicación lo haga.

Este componente se basa en técnicas de predicción que no requieren una lógica muy compleja, como las cadenas de Markov (Joseph y Grunwald, 1997). El problema principal de este tipo de técnicas es que son agnósticas respecto de lo que la aplicación está ejecutando. Por lo tanto, a pesar de que en algunas aplicaciones puede funcionar bastante bien, en otras las predicciones pueden ser erróneas y hacer que el rendimiento de la aplicación empeore.

2) ***Software prefetching.*** Algunos de los sistemas multinúcleo facilitan instrucciones para que las aplicaciones puedan pedir de manera explícita los datos a la memoria, usando la instrucción de *prefetch*.

La instrucción vprefetch

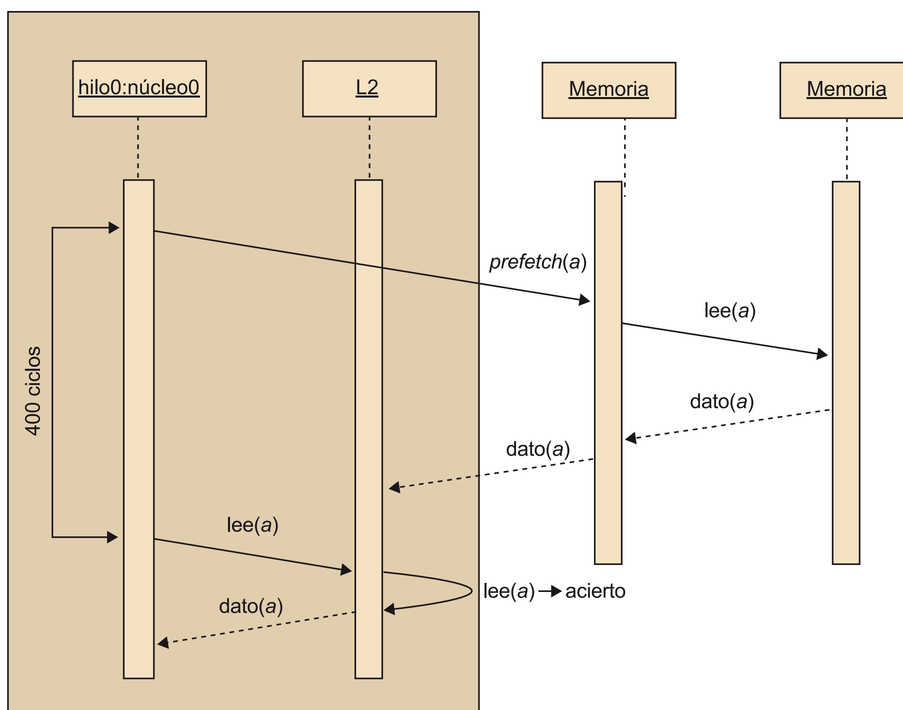
En la arquitectura Intel se usa la instrucción *vprefetch* (Intel, *Use Software Data Prefetch on 32-Bit Intel / IA-32 Intel® Architecture Optimization Reference Manual*, 2011). La aplicación es responsable de pedir los datos a la memoria de manera anticipada usando esta instrucción.

La ventaja de este mecanismo es que la aplicación sabe exactamente cuándo necesitará los datos. Por lo tanto, basándose en la latencia de un acceso a memoria, tiene que pedir los datos con el número de ciclos suficiente para que cuando la necesite la tenga. El uso de este tipo de *prefetch* es el que acostumbra a permitir obtener el rendimiento más elevado de la aplicación.

La desventaja principal es que el código se encuentra ligado a una arquitectura concreta. Es decir, la mayoría de arquitecturas multinúcleo tienen latencias diferentes. Por lo tanto, cada vez que se quiere ejecutar esta aplicación en una plataforma nueva hay que calcular las distancias de *prefetch* adecuadas al sistema nuevo.

La figura siguiente presenta un ejemplo de cómo funcionaría un *prefetch* en un sistema formado por un multinúcleo con una memoria caché de tercer nivel. En este caso la aplicación necesita el dato *a* en el ciclo *X*; asumiendo que la latencia de un acceso a memoria que falla en la L3 es de 400 ciclos, tiene que hacer *el prefetch X*-400 ciclos antes.

Figura 28. Ejemplo de *prefetch*.



Sin embargo, hay que tener en cuenta las consideraciones siguientes:

- Los *prefetchs* acostumbran a poder ser eliminados del *pipeline* del procesador si no hay suficientes recursos para llevarlos a cabo (por ejemplo, si la cola que guarda los datos que vuelven de la L3 está llena). Por lo tanto, si el número *de prefetchs* que hace una aplicación es demasiado elevado, pueden ser eliminados del sistema.

- La latencia de las peticiones a memoria puede variar dependiendo del camino que sigan. Por ejemplo, un acierto en la L3 hace que un dato esté disponible en la misma memoria L3 mucho antes de lo previsto, y en el supuesto de que haya una víctima interna, esta será mucho más tardía. Cuando se usan este tipo de peticiones hay que tener en cuenta el uso de toda la jerarquía de memoria, como también las características de la aplicación que se usa.

- Los *prefetchs* pueden tener impactos de rendimiento tanto positivos como negativos en la aplicación desarrollada. Hay que estudiar qué requisitos tiene la aplicación desarrollada y cómo se comportan en la arquitectura que ejecuta la aplicación.

Lectura recomendada

El uso de técnicas de *prefetch* en arquitecturas multihilo es un recurso frecuente para sacar rendimiento en aplicaciones paralelas. Por este motivo, se recomienda leer el artículo:

Intel (2007). *Optimizing Software for Multe-core Processors*. Portland: Intel Corporation - White Papel.

5.3.3. Impacto del tipo de memoria caché

Cada vez que un núcleo accede a una línea de memoria por primera vez, lo tiene que pedir al nivel de memoria siguiente. En los ejemplos estudiados, los fallos en la L2 se piden a la L3, y los fallos en la L3 se piden a la memoria. Cada uno de estos fallos genera un conjunto de víctimas en las diferentes memorias caché: hay que liberar una entrada por la línea que se está pidiendo.

Para sacar rendimiento a las aplicaciones paralelas que se desarrollan, es importante mantener al máximo la localidad en los accesos a las memorias caché. Es decir, maximizar el reúso (porcentaje de acierto) en las diferentes memorias caché (L1, L2, L3, etc.). Por este motivo es importante considerar las características de la jerarquía de memoria caché: inclusiva, no inclusiva/exclusiva y medidas.

A continuación se discuten los diferentes puntos mencionados desde el punto de vista de la aplicación.

Inclusividad

Si dos memorias (Lx y Lx – 1) son inclusivas quiere decir que cualquier dirección @X que esté en Lx – 1 se encuentra siempre en la Lx. Por ejemplo, si la L1 es inclusiva con la memoria L2, esta incluye la L1 y otras líneas. En este caso hay que considerar lo siguiente:

1) Cuando una línea de la Lx – 1 se victimiza, al final de la transacción esta está disponible en el nivel siguiente de memoria Lx.

2) Cuando una línea de la Lx se victimiza, al final de la transacción esta línea ya no está disponible en el nivel superior Lx – 1. Por ejemplo, en el caso de victimizar la línea @X en L3, esta se invalida también en la memoria caché L2. Y si la L1 es inclusiva con la L2, la primera también invalida la línea en cuestión.

3) Cuando un núcleo lee una dirección @X que no se encuentra en la memoria Lx – 1, al final de la transacción esta también se encuentra incluida en la Lx. Por ejemplo, en el supuesto de que tengamos una L1, L2 y L3 inclusivas, @X se escribe en todas las memorias. Hay que remarcar que cada una de estas entradas usadas potencialmente ha generado una víctima. Es decir, una dirección @Y que usa el *way* y el *set* en el cual se ha guardado @X. La selección de esta posición depende de la política de gestión de cada memoria caché, como también de la medida.

Los puntos 2 y 3 pueden causar un impacto bastante significativo en el rendimiento de la aplicación. Por lo tanto, es importante diseñar las aplicaciones para que el número de víctimas generadas en niveles superiores sea lo más bajo posible y maximizar el porcentaje de aciertos de las diferentes jerarquías de memoria caché más cercanas al núcleo (por ejemplo, L1).

Exclusividad y no inclusividad

Que dos memorias caché sean exclusivas implica que si la memoria caché Lx tiene una línea @X, la memoria caché Lx – 1 no la tiene. No se acostumbra a tener arquitecturas en las que todos los niveles son exclusivos. Habitualmente, las jerarquías que tienen memorias caché exclusivas acostumbran a ser híbridas.

Procesadores con memoria exclusiva

Un ejemplo de procesador con memoria exclusiva es el AMD Athlon (AMD, 2011) y el Intel Nehalem (Intel, *Nehalem Processor*, 2011). El primero tiene una L1 exclusiva con la L2. El segundo tiene una L2 y L1 no inclusivas y la L3 es inclusiva de la L2 y la L1.

En los casos en que una memoria caché Lx es exclusiva con Lx – 1, hay que considerar lo siguiente:

- Cuando se accede a una línea @Y en la memoria Lx – 1, esta se mueve de la memoria Lx.

- Cuando una línea se victimiza de la memoria Lx – 1, esta se mueve a la memoria caché Lx. En estos casos, como la memoria es exclusiva, hay que

encontrar una entrada en la memoria Lx. Por lo tanto, hace falta victimizar la línea que esté guardada en el *way* y el *set* seleccionado.

- Cuando una línea se victimiza de la memoria Lx, no hace falta victimizar la memoria caché Lx – 1.

Hay ciertas situaciones en las que hay que saber en detalle qué tipos de jerarquía de memoria y protocolo de coherencia implementa el procesador. Por ejemplo, si se considera una arquitectura multinúcleo en la que la L1 es exclusiva con la L2, en los casos en que los diferentes hilos estén compartiendo el acceso a un conjunto elevado de direcciones el rendimiento de la aplicación se puede ver deducido. En esta situación, cada acceso a un dato @X en un núcleo podría implicar la victimización de esta misma dirección en otro núcleo y pedir la dirección a la memoria.

Algunas memorias caché exclusivas permiten que en ciertas situaciones algunos datos se encuentren en dos memorias que son exclusivas entre sí por defecto. Por ejemplo, en las líneas de memoria compartidas por diferentes núcleos.

Medida de la memoria

El rendimiento de la aplicación depende en gran medida de la localidad de los accesos de los diferentes hilos en las memorias caché. En situaciones en las que los hilos piden varias veces las mismas líneas a la memoria por mala praxis de programación: el rendimiento de la aplicación caché sustancialmente. Esto pasa cuando un núcleo pide una dirección @X, esta línea se victimiza y se vuelve a pedir más tarde.

Un ejemplo sencillo es acceder a una matriz de enteros de 8 bytes por filas cuando esta se ha almacenado por columnas. En este caso, cada 8 enteros consecutivos de una misma columna se encontrarían mapeados en la misma línea. Ahora bien los elementos *i y i* + 1 de una fila se encontrarían guardados en líneas diferentes. Por lo tanto, en el caso de recorrer la matriz por columnas, el número de fallos en la L1 será mucho más elevado.

Por este motivo, es importante usar técnicas de acceso a los datos que intenten mantener localidad en las diferentes memorias caché.

Sin embargo, en muchos casos las aplicaciones paralelas diseñadas no siguen ninguno de los patrones que hemos analizado en otros estudios académicos (por ejemplo, en técnicas de partición de matrices). Para estos problemas hay aplicaciones disponibles que permiten analizar cómo se comportan las aplicaciones paralelas y ver de qué manera se pueden mejorar.

Lecturas recomendadas

De manera parecida a otros factores ya introducidos anteriormente, se ha hecho mucha búsqueda en este ámbito. Una referencia en técnicas de partición en bloques por la multiplicación de matrices es la siguiente:

K. Kourtis; G. Goumas; N. Koziris (2008). "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression". *37th International Conference on Parallel Processing*.

Y en técnicas de compresión en el proceso de grafs:

R. Jin; T. S. Chung (2010). "Node Compression Techniques Based on Cache-Sensitive B+-Tree". *9th International Conference on Computer and Information Science* (ICIS) (pág. 133-138).

Ejemplo

Algunos ejemplos son Vtune (Kishan Malladi, 2011) o Cachegrind (Valgrind, 2011).

5.3.4. Arquitecturas multinúcleo y multiprocesador

En algunas situaciones, las arquitecturas en las que se ejecutan las aplicaciones paralelas no solo dan acceso a múltiples hilos y múltiples núcleos, sino a múltiplos procesadores. En la mayoría de casos, estas arquitecturas contienen una placa madre o más en las que cada una contiene un conjunto de procesadores que están conectados por medio de una conexión de alta velocidad. Si la arquitectura de computación está formada por más de una placa, acostumbran a estar conectadas por redes de conexión más lentas.

Tipo de conexiones

Un ejemplo de conexión de alta velocidad es la Intel Quickpath Interconnect (Intel, *Intel® Quickpath Interconnect Maximizes Multe-Core Performance*, 2012), también conocido como *QPI*. Y como ejemplo de conexión más lenta, tenemos la Myrinet (Flich, 2000).

En estas arquitecturas se pueden asignar los diferentes hilos de la aplicación a cada uno de los hilos disponibles en cada uno de los núcleos de los diferentes procesadores conectados a una misma placa. Todos los diferentes hilos acostumbran a compartir un espacio de memoria coherente. Desde el punto de vista de la aplicación, esta tiene acceso a *N* núcleos diferentes y a un espacio de memoria común.

Por otro lado, los hilos que se han ubicado en placas diferentes no acostumbran a compartir el espacio de direcciones de memoria. Por lo tanto, si se quiere que compartan información, hay que emplear un entorno que permita hacerlo. Hay modelos de programación que lo permiten. El modelo de paso de mensajes[20] es el ejemplo más conocido. Este modelo de programación permite la comunicación de procesos ubicados en placas diferentes (que, por lo tanto, no comparten el espacio de direcciones) por medio de envío de mensajes. En

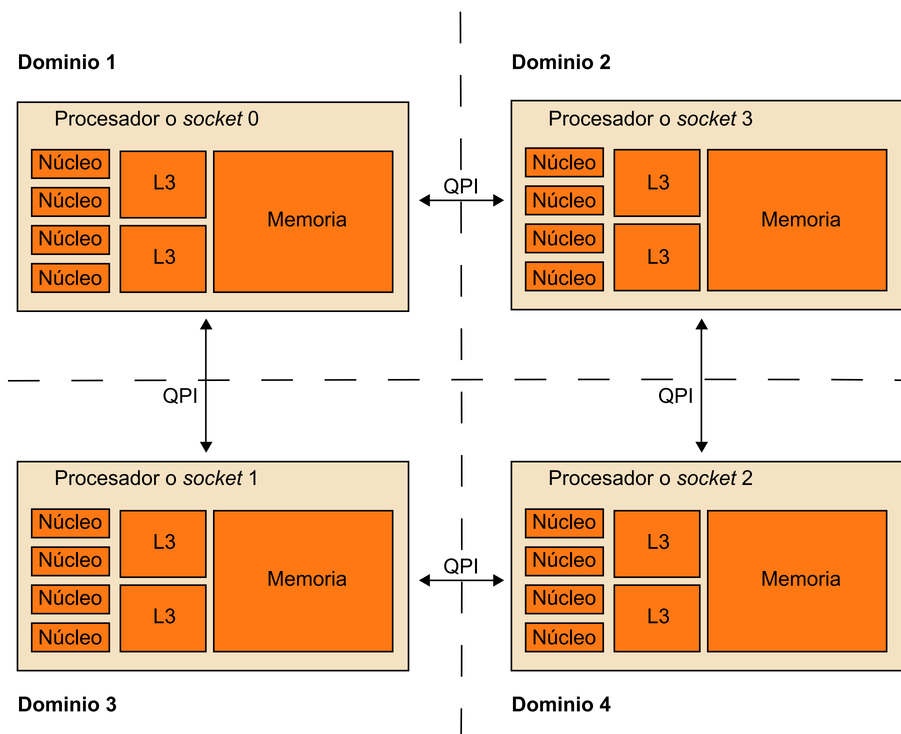
[20] En inglés, *message passing interface* (*MPI*).

muchos casos, este modelo se combina con OpenMP. Este último permite definir paralelismos dentro de una misma placa, asumiendo que los diferentes hilos comparten el mismo espacio de direcciones.

A pesar de que este espacio de memoria puede ser compartido por todos los hilos de una misma placa, el acceso de un hilo a determinadas zonas de memoria puede tener latencias diferentes. Cada procesador dispone de una jerarquía de memoria propia (desde la L1 hasta la memoria principal) y este gestiona un rango de direcciones de memoria concreto. Así, si un procesador dispone de una memoria principal de 2 GB, este procesador gestiona el acceso del espacio de direcciones asignado a estos 2 GB (por ejemplo, de 0x a FFFFFFFF).

Cada vez que un hilo accede a una dirección que se encuentra asignada a un espacio que gestiona otro procesador, tiene que enviar la petición de lectura de esta dirección al procesador que la gestiona a través de la red de interconexión (como en una QPI). Estos accesos son mucho más costosos puesto que tienen que enviar la petición a través de la red, llegar al otro procesador y hacer el acceso (siguiendo el protocolo de coherencia que siga la arquitectura).

Figura 29. Arquitectura multihilo



Este modelo de programación sigue el paradigma de lo que se denomina *non-uniform memory access* (NUMA), en el que un acceso de memoria puede tener diferentes tipos de latencia dependiendo de donde se encuentre asignado.

Cuando se programe por este tipo de arquitectura, habrá que considerar todos los factores que se han explicado en este apartado, pero en una escalera superior. Así pues, en el acceso a variables de exclusión mutua se tiene que considerar que, por cada vez que se coja el *lock*, habrá que invalidar el resto de núcleos del sistema. Los que están fuera del procesador van por la red de interconexión y tardan más en devolver la respuesta. Justo es decir que el comportamiento depende del protocolo de coherencia y de la jerarquía de memoria del sistema.

En este módulo se han introducido algunos de los aspectos más importantes que hay que tener en cuenta a la hora de diseñar y desarrollar aplicaciones para arquitecturas de computación de altas prestaciones. Como se ha podido ver con la gran cantidad de referencias facilitadas, este ámbito es muy complejo y, para profundizar en el mismo, hay que ocupar mucho esfuerzo y dedicación.

Lecturas recomendadas

Dentro del ámbito de programación multihilo, para este tipo de arquitecturas se pueden encontrar muchas referencias interesantes. Algunas son las siguientes:

G. R. Andrews (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, Massachusetts: Addison-Wesley.

M. Herlihy; N. Shavit (2008). *The Art of Multiprocessor Programming*. Burlington, Massachusetts: Morgan Kaufmann.

D. E. Culler; J. Palo Singh (1999). *Parallel computer architecture: a hardware/software approach*. Burlington, Massachusetts: Morgan Kaufmann.

Resumen

En este módulo didáctico se han presentado los fundamentos arquitectónicos de procesadores necesarios para poder entender la computación de altas prestaciones. Primeramente, después de una breve introducción que trataba de la descomposición funcional y de datos, se ha presentado la taxonomía de Flynn, de la cual se deriva una caracterización de las diferentes arquitecturas de computadores que se pueden encontrar en el ámbito de la computación. A continuación se han tratado las arquitecturas *single instruction, multiple data* (*SIMD*) y las *multiple instruction, multiple data* (*MIMD*).

A continuación se ha presentado la arquitectura de uno de los procesadores más extendidos dentro del mundo de la computación: las arquitecturas vectoriales. Más adelante, al tratar de las *MIMD*, se ha hecho un análisis detallado de sus diferentes variantes. Empezando por las arquitecturas *superthreading* y acabando por las arquitecturas multinúcleo.

Como se ha podido ver, mejorar el rendimiento que las diferentes arquitecturas proporcionan no es una tarea sencilla. En muchos casos hay factores que limitan su escalabilidad o que hacen sus mejoras extremadamente costosas. Por ejemplo, en un procesador multinúcleo, dependiendo de qué tipo de red de interconexión se use, el número de núcleos que puede soportar no escala hasta un número muy elevado (por ejemplo, una red de interconexión de tipo bus).

Después se han estudiado algunos de los factores más importantes que hay que tener en cuenta a la hora de evaluar la escalabilidad y rendimiento de un procesador de altas prestaciones (como por ejemplo, el protocolo de coherencia). Para poder sacar el máximo rendimiento a las aplicaciones, hay que tener en cuenta ciertas limitaciones o restricciones que presentan este tipo de arquitecturas. En el último apartado hemos mostrado un conjunto de aspectos importantes que hay que tener en cuenta a la hora de diseñarlas y desarrollarlas. Por ejemplo, se ha presentado el impacto que tiene la compartición falsa en el acceso a datos. En este caso, diferentes hilos de ejecución acceden a datos que se encuentran ubicados en diferentes direcciones físicas, pero que comparten la misma entrada de la memoria caché. Esto hace que compitan por el mismo recurso de manera continuada, cosa que degrada mucho su rendimiento. Tal como ya se ha mencionado, la compartición falsa se puede evitar añadiendo lo que se denomina *padding*, es decir, uno de los dos datos se declara desplazado para que se mapee en una dirección física y, por lo tanto, no coincida con la misma entrada de memoria caché.

Tal como se ha hecho patente durante todo el módulo didáctico, la complejidad de este ámbito de búsqueda es bastante elevada. Las generaciones nuevas de computadores que demanda el mercado requieren cada vez más capacidad de proceso. Esta capacidad va tan ligada a la cantidad de procesamiento que estas facilitan (el número de núcleos e hilos de ejecución) como a la capacidad de procesar datos (por ejemplo, por medio de unidades vectoriales). Esto es especialmente importante en la computación de altas prestaciones.

Es recomendable, pues, que el alumno lea algunas de las referencias proporcionadas a lo largo del módulo. Una lectura en profundidad de los artículos recomendados ofrece una perspectiva más detallada de los factores que se consideran más importantes dentro de la búsqueda hecha.

Actividades

1. En el subapartado 5.1 se han tratado diferentes factores que hay que tener en cuenta a la hora de programar arquitecturas multihilo. Sin embargo, las arquitecturas vectoriales también tienen retos y factores que hay que tener en cuenta a la hora de programarlas. Para esta actividad se propone profundizar en las arquitecturas vectoriales con la lectura del artículo siguiente: "Intel® AVX: New Frontiers in Performance Improvements and Energy Efficiency" (Firasta, Buxton, Jinbo, Nasri y Kuo, 2008).

2. "El ejemplo de suma vectorial" del apartado 3 muestra una comparativa de un mismo algoritmo de suma usando un código escalar y un código vectorial. Asumiendo que hacer una *load* y un *store* tardan 10 ciclos y 15 ciclos en escalar y vectorial, respectivamente, y que una suma escalar y vectorial tardan 1 y 2 ciclos respectivamente, ¿qué *speedup* obtendríamos en la versión vectorial respecto de la escalar?

3. Para esta actividad se propone buscar y explicar dos problemas o algoritmos diferentes que se puedan beneficiar de la computación *SIMD*. En el apartado 3, dedicado a las arquitecturas *SIMD*, se han enumerado algunos de estos algoritmos. Sin embargo, no se ha detallado cómo han adaptado sus algoritmos para emplear las operaciones vectoriales y procesar los datos que usan. Hay que estudiar y explicar los detalles y hacer una estimación de las diferencias en cuanto a rendimiento de la versión vectorial respecto de la escalar.

4. Explicad cuáles son las diferencias más importantes entre un protocolo basado en directorio y uno basado en *snoops* y en qué situaciones creéis que un protocolo basado en directorio es más adecuado que uno basado en *snoops* y viceversa. Razonad la respuesta.

5. Una de las métricas más importantes que hay que optimizar en el diseño de procesadores es el consumo energético. En ninguno de los apartados anteriores se ha detallado el impacto de la complejidad de los procesadores multihilo o multinúcleo en el consumo energético. Enumerad las aportaciones más interesantes del artículo siguiente en este aspecto: "Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency" (Lee y Brooks, 2005).

6. Implementad un código multihilo en el que pueda haber un problema de carrera de acceso. Deberéis razonar el problema que puede aparecer en tiempo de ejecución. Haced el mismo ejercicio, pero en un código en el que el programa se pueda bloquear (es decir, acabar en *un deadlock*).

7. Las arquitecturas multinúcleo han cobrado mucha importancia durante los últimos años. Permiten que aplicaciones que pueden escalar su rendimiento si tienen más fuente de paralelismo se beneficien mucho. Larrabee fue un multinúcleo que la empresa Intel propuso en el 2008. Para esta actividad hay que leer el artículo siguiente y hacer una lista con 15 características positivas de esta arquitectura y 15 de negativas: "Larrabee: A Many-Core x86 Architecture for Visual Computing" (Seiler y otros, 2008).

Bibliografía

Agrawal, D. P.; Feng, T. Y.; Wu, C. L. (1978). “A survey of communication processors systems”. *Proc. COMPSAC* (pág. 668-673).

Alverson, R.; Callahan, D.; Cummings, D.; Koblenz, B. (1990). “The Tera computer system”. *International Conference on Supercomputing* (pág. 1-6).

AMD (2011). *AMD Athlon™ Processor*. Recuperado el 4 de enero de 2012 d'AMD Athlon™ Processor.

Andrews, G. R. (1999). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, Massachusetts: Addison-Wesley.

Bailey, D.; Barton, J.; Lasinski, T.; Simon, H. (1991). “The NAS parallel benchmarks”. *Technical Report RNR-91-002 Revision 2, 2*. NASA Ames Research Laboratory.

Baniwal, R. (2010). “Recent Trends in Vector Architecture: Survey”. *International Journal of Computer Science & Communication* (vol.1, núm. 2, pág. 395-339).

Bell, S.; Edwards, B.; Amann, J.; Conlin, R.; Joyce, K.; Leung, V. y otros (2008). “TILE64 Processor: A 64-Core SoC with Mesh Interconnect”. *IEEE International Solid-State Circuits Conference*.

Ceder, A.; Wilson, N. (1986). “Bus network design”. *Transportation Design* (pág. 331-344).

Chaiken, D.; Fields, C.; Kurihara, K.; Agarwal, A. (1990). “Directory-based cache coherence in large-scale multiprocessors”. *Computer* (pág. 49-58).

Chapman, B.; Huang, L.; Biscondi, E.; Stotzer, E.; Shrivastava, A. G. (2008). “Implementing OpenMP on a High Performance Embedded Multicore MPSoC”. *IPDPS*.

Chase, J.; Doyle, R. (2001). “Balance of Power: Energy Management for Server Clusters”. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*.

Chynoweth, M.; Lee, M. R. (2009). “Implementing Scalable Atomic Locks for Multi-Core”. Recuperado el 28 de diciembre de 2011 d'Implementing Scalable Atomic Locks for Multi-Core.

Corp, I. (2007). “Intel SSE4 Programming Reference”. *Intel, Intel® SSE4 Programming Reference* (pág. 1-197). Dender: Intel Corporation.

Corp, I. (2011). *AVX Instruction Set*. Recuperado el 23 de marzo de 2012: http://software.intel.com/en-us/avx/

Dixit, K. M. (1991). “The SPEC benchmark”. *Parallel Computing* (pág. 1195-1209).

Culler, D. E.; Pal Singh, J. (1999). *Parallel computer architecture: a hardware/software approach*. Burlington, Massachusetts: Morgan Kaufmann.

Firasta, N.; Buxton, M.; Jinbo, P.; Nasri, K.; Kuo, S. (2008). “Intel® AVX: New Frontiers in Performance Improvements and Energy Efficiency”. *Intel White Paper*.

Fisher, J. (1893). “Very Long Instruction Word Architectures and the ELI-512”. *Proceedings of the 10th Annual International Symposium on Computer Architecture* (pág. 140-150).

Flich, J. (2000). “Improving Routing Performance in Myrinet Networks”. *IPDPS*.

Gibbons, A.; Rytte, W. (1988). *Efficient Parallel Algorithms*. Cambridge: Cambridge University Press.

Grunwald, D.; Zorn, B.; Henderson, R. (1993). “Improving the cache locality of memory allocation”. *ACM SIGPLAN 1993 Conference on Programming Language Design And Implementation*.

Handy, J. (1998). *The cache memory book*. Londres: Academic Press Limited.

Hennessy, J. L.; Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Burlington, Massachusetts: Morgan Kaufmann.

Herlihy, M.; Shavit, N. (2008). *The Art of Multiprocessor Programming*. Burlington, Massachusetts: Morgan Kaufmann.

Hewlett-Packard (1994). “Standard Template Library Programmer's Guide”. Recuperado el 9 de enero de 2012 d’Standard Template Library Programmer’s Guide

Hily, S.; Seznec, A. (1998). “Standard Memory Hierarchy Does Not Fit Simultaneous Multithreading”. *Workshop on Multithreaded Execution, Architecture, and Compilation*.

Hong, Y.; Payne, T. H. (1989). “Parallel Sorting in a Ring Network of Processors”. *IEEE Transactions on Computers* (vol. 38, núm. 3).

IBM (2011). “AS/400 Systems”. Recuperado el 20 de diciembre de 2011: http://www-03.ibm.com/systems/i/.

IBM. “System/370 Model 145”. Recuperado el 12 de marzo de 2012 d’IBM Corporation Web Site.

IEEE (2011). “IEEE POSIX 1003.1c standard”. Recuperado el 11 de enero de 2012 d’IEEE POSIX 1003.1c standard.

insideHPC. “InsideHPC: A Visual History of Cray”. Recuperado el 1 de marzo de 2012 d’insideHPC Web Site.

Intel (2007). *Intel® Threading Building Blocks Tutorial*.

Intel (2007). *Optimizing Software for Multi-core Processors*. Portland: Intel Corporation - White Paper.

Intel (2011). *1971-2011. 40 años del microprocesador*. Portland: Intel Corporation.

Intel (2011). “Boost Performance Optimization and Multicore Scalability”. Recuperado el 3 de enero de 2012 de Boost Performance Optimization and Multicore Scalability.

Intel (2011). “Intel Cilk Plus”. Recuperado el 21 de diciembre de 2011 d’Intel Cilk Plus.

Intel (2011). *Intel® Array Building Blocks 1.0 Release Notes*.

Intel (2011). “Nehalem Processor”. Recuperado el 4 de enero de 2012 de Nehalem Processor.

Intel (2011). *Use Software Data Prefetch on 32-Bit Intel / IA-32 Intel® Architecture Optimization Reference Manual*. Portland: Intel Corporation.

Intel (2012). “Intel Sandy Bridge - Intel Software Network”. Recuperado el 8 de enero de 2012 d’Intel Sandy Bridge - Intel Software Network.

Intel (2012). “Intel® Quickpath Interconnect Maximizes Multi-Core Performance”. Recuperado el 8 de enero de 2012 d’Intel® Quickpath Interconnect Maximizes Multi-Core Performance.

Intel. “Sandy Bride Intel”. Recuperado el 10 de diciembre de 2011 de Sandy Bride Intel.

Jin, R.; Chung, T. S. (2010). “Node Compression Techniques Based on Cache-Sensitive B+-Tree”. *9th International Conference on Computer and Information Science* (*ICIS*) (pág. 133-138).

Joseph, D.; Grunwald, D. (1997). “Prefetching using Markov predictors”. *24th Annual International Symposium On Computer Architecture*.

Katz, R. H.; Eggers, S. J.; Wood, D. A.; Perkins, C. L.; Sheldon, R. G. (1985). “Implementing a cache consistency protocol”. *Proceedings of the 12th Annual International Symposium on Computer Architecture.*

Kim, B. C.; Jun, S. W. H. K. (2009). “Visualizing Potential Deadlocks in Multithreaded Programs”. *10th International Conference on Parallel Computing Technologies*.

Kishan Malladi, R. (2011). *Using Intel® VTune™ Performance Analyzer Events/ Ratios & Optimizing Applications*. Portland: Intel Corporation.

Kongetira, P.; Aingaran, K.; Olukotun., K. (2005). “Niagara: A 32- Way Multithreaded SPARC Processor”. *IEEE MICRO Magazine*.

Kourtis, K.; Goumas, G.; Koziris, N. (2008). "Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication Using Index and Value Compression". *37th International Conference on Parallel Processing.*

Kumar, R.; Farkas, K. I.; Jouppi, N. P.; Ranganathan, P.; Tullsen, D. M. (2003). "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction". *Microarchitecture, MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (pág. 81-92).

Kwak, H.; Lee, B.; Hurson, A.; Suk-Han, Y.; Woo-Jong, H. (1999). "Effects of multithreading on cache performance". *IEEE Transactions on Computer* (pág. 176-184).

Lee, B. C.; Brooks, D. (2005). "Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency". *Workshop on Complexity Effective Design 2005* (WCED2005, held in conjuction with ISCA-32).

Linux (2010). *Linux Programmer's Manual*. Recuperado el 3 de enero de 2012 de Linux Programmer's Manual.

Lo, J. L. (1997). *ACM Transactions on Computer Systems. Converting Thread-level Parallelism to Instructionlevel Parallelism via Simultaneous Multithreading.*

Lo, J. L.; Eggers, S. J.; Levy, H. M.; Parekh, S. S.; Tullsen, D. M. (1997). "Tuning Compiler Optimizations for Simultaneous Multithreading". International Symposium on Microarchitecture (pág. 114-124).

Mellor-Crummey, J. M.; Scott, M. L. (1991). "Algorithms for scalable synchronization on shared-memory multiprocessors". *ACM Transactions on Computer Systems.*

Marr, D. (2002). "Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History". *Intel Technology J.* (vol. 8, núm. 1).

Martorell, X.; Corbalán, J.; González, M.; Labarta, J.; Navarro, N.; Ayguadé, E. (1999). "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors". *13th International Conference on Supercomputing.*

Melvin, S. (2000). "Clearwater networks cnp810sp simultaneous multithreading (smt) core". Recuperado el 19 de diciembre de 2011: www.zytek.com/~melvin/clearwater.html.

Melvin, S. (2003). "Flowstorm porthos massive multithreading (mmt) packet processor".

ParaWise (2011). "ParaWise - the Computer Aided Parallelization Toolkit". Recuperado el 27 de diciembre de 2011 de ParaWise - the Computer Aided Parallelization Toolkit.

Philbin, J.; Edler, J.; Anshus, O. J.; Douglas, C.; Li, K. (1996). "Thread scheduling for cache locality". *7th International Conference on Architectural Support for Programming Languages and Operating Systems.*

Prasad, S. (1996). *Multithreading Programming Techniques*. Nueva York: McGraw-Hill, Inc.

Reed, D. i Grunwald, D. (1987). "The Performance of Multicomputer Interconnetion Networks". *Computer* (pág. 63-73).

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly.

Rusu, S.; Tam, S.; Muljono, H.; Ayers, D.; Chang, J. (2006). "A dual-core multi-threaded Xeon(r) processor with 16 Mb L3 cache*". Proc. 2006 IEEE Int. Solid-State Circuits Conf.* (pág. 315-324).

Sabot, G. (1995). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, Massachusetts: Addison-Wesley.

Seiler, L.; Carmean, D.; Sprangle, E.; Forsyth, T.; Dubey, Junkins, S. P. y otros (2008). "Larrabee: A Many-Core x86 Architecture for Visual Computing". *ACM Transaction on Graphics.*

Seznec, A. (1993). "A case for two-way skewed-associative caches". *20th Annual International Symposium on Computer Architecture.*

Seznec, A.; Felix, S.; Krishnan, V.; Sazeide, Y. (2002). "Design tradeoffs for the Alpha EV8 conditional branch predictor". *Proceedings of the 29th International Symposium on Computer Architecture*.

Song, P. (2002). "A tiny multithreaded 586 core for smart mobile devices". *2002 Microprocessor Forum (MPF)*.

Stenström, P. (1990). "A Survey of Cache Coherence for Multiprocessors". *IEE Computer Transactions*.

Tullsen, D. M.; Eggers, S.; Levy, H. M. (1995). "Simultaneous multithreading: Maximizing on-chip parallelism". *22th Annual International Symposium on Computer Architecture*.

Valgrind (2011). "Cachegrind: a cache and branch-prediction profiler". Recuperado el 3 de enero de 2012 de Cachegrind: a cache and branch-prediction profiler.

Villa, O.; Palermo, G.; Silvano, C. (2008). "Efficiency and scalability of barrier synchronization on NoC based many-core architectures". *CASES '08 Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*.

Yao, E.; Demers, A.; Shenker, S. (1995). "A scheduling model for reduced CPU energy". *IEEE 36th Annual Symposium on Foundations of Computer Science* (pág. 374-382).