Arquitecturas de altas prestaciones

Francesc Guim Bernat
Ivan Rodero Castro

PID_00209163

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 18

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. Arquitecturas multinúcleo 31
4.6.1. Limitaciones de la *SMT* y arquitecturas superthreading 31
4.6.2. Producción 33
4.6.3. El concepto de multinúcleo 34
4.6.4. Coherencia entre núcleos 36

5. Arquitecturas *many-core*: el caso de Intel Xeon Phi.......... 39
5.1. Historia de los Xeon Phi 39
5.2. Presentación de los Xeon KNC y KNF 41
5.3. Arquitectura y sistema de interconexión 43
5.4. Núcleos de los KNC 47
5.5. Sistema de coherencia 49
5.6. Protocolo de coherencia 52
5.7. Conclusiones 54

Resumen.......... 56

Actividades.......... 59

Bibliografía.. 60

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

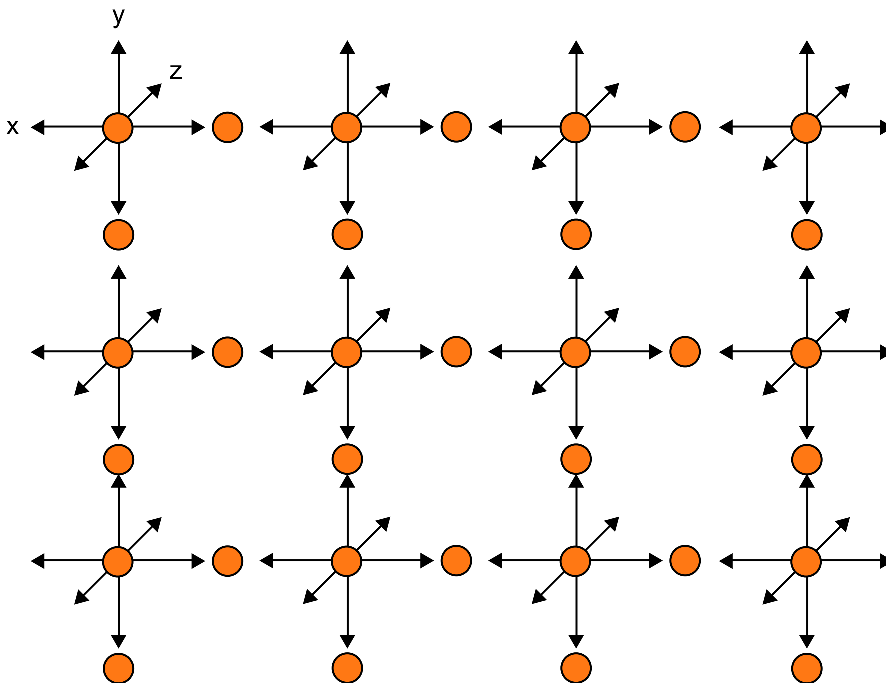
Como se muestra a lo largo de esta unidad didáctica, la mayoría de arquitecturas de altas prestaciones que hoy en día se pueden encontrar en el mercado permiten ejecutar más de un hilo de ejecución de manera paralela. Por otro lado, algunas también proporcionan acceso a unidades de proceso específicas,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

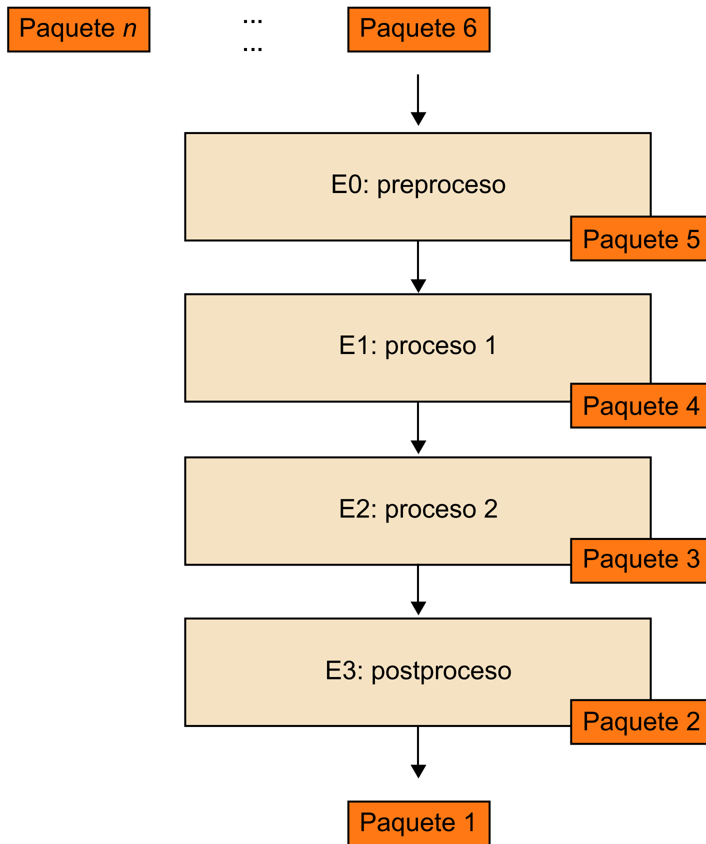
M. S. Lam; E. E. Rothberg; M. E. Wolf (1991).

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



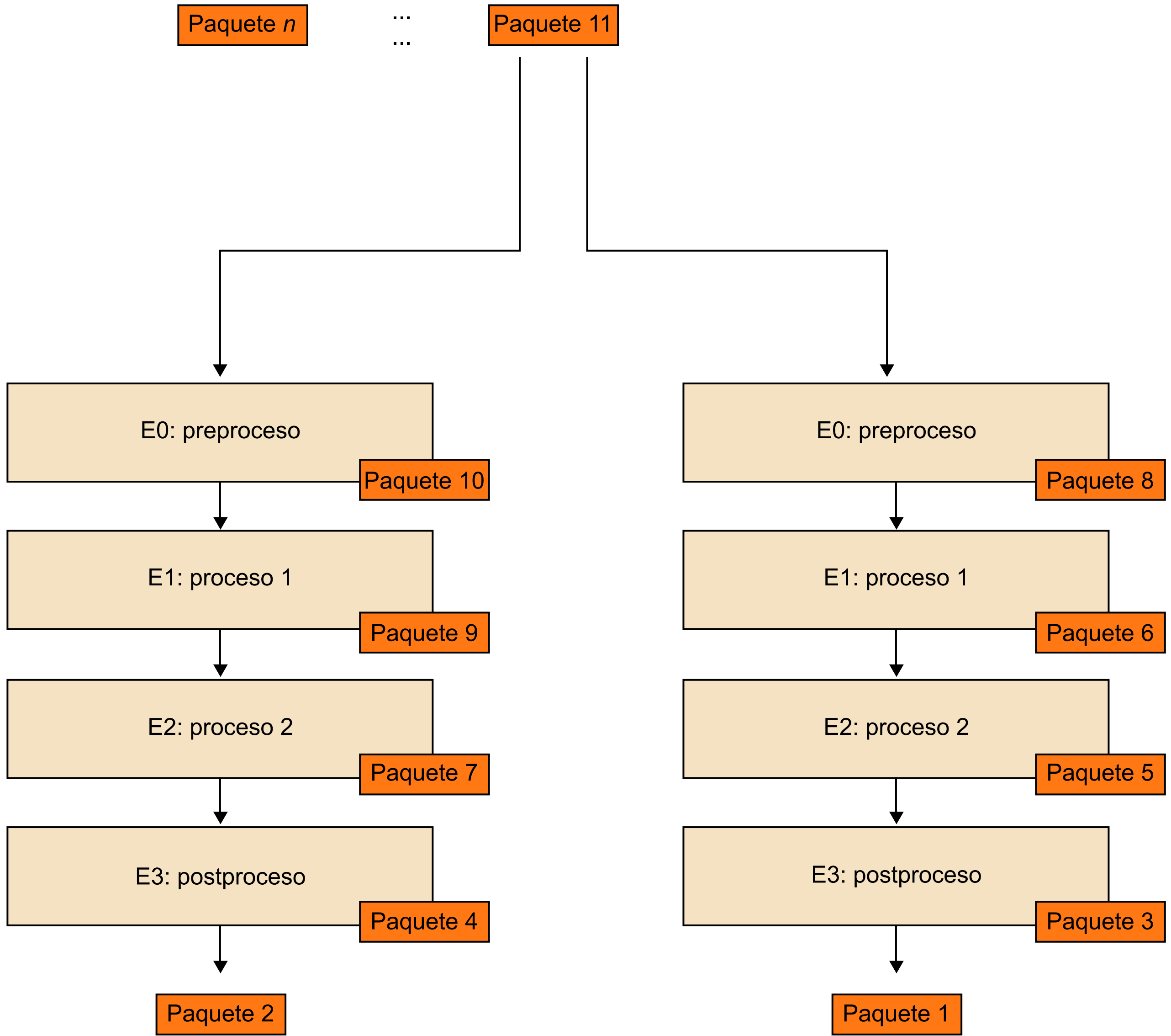
A pesar de que los dos ejemplos anteriores tratan la descomposición de datos y funcional separadamente, en la realidad estos dos problemas están directamente relacionados. A la hora de analizar la paralelización de una aplicación, se estudian tanto la descomposición funcional como la de datos. En general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

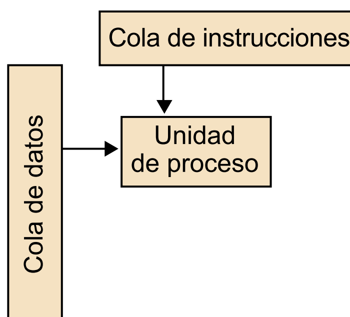
1) Una instrucción, un dato[1]**:** son las arquitecturas tradicionales formadas por una sola unidad de proceso. Como se puede observar a continuación, no explotan ni el paralelismo a nivel de hilo, ni el paralelismo a nivel de datos. En este caso, tenemos un solo flujo de datos y de instrucciones. La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

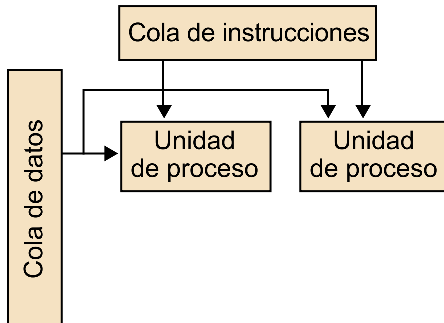
(2)En inglés, *single instruction, multiple data stream* (*SIMD*).

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



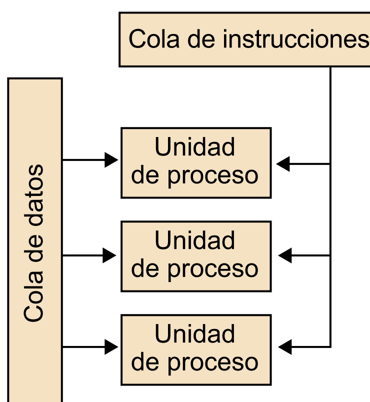
3) Múltiples instrucciones, un dato[3]**:** permiten ejecutar diferentes flujos de instrucciones sobre un mismo bloque de datos. En este caso, tenemos diferentes unidades de proceso que operan sobre el mismo dato. Este tipo de arquitecturas es muy específico y no se encuentra ninguno dentro del mundo 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

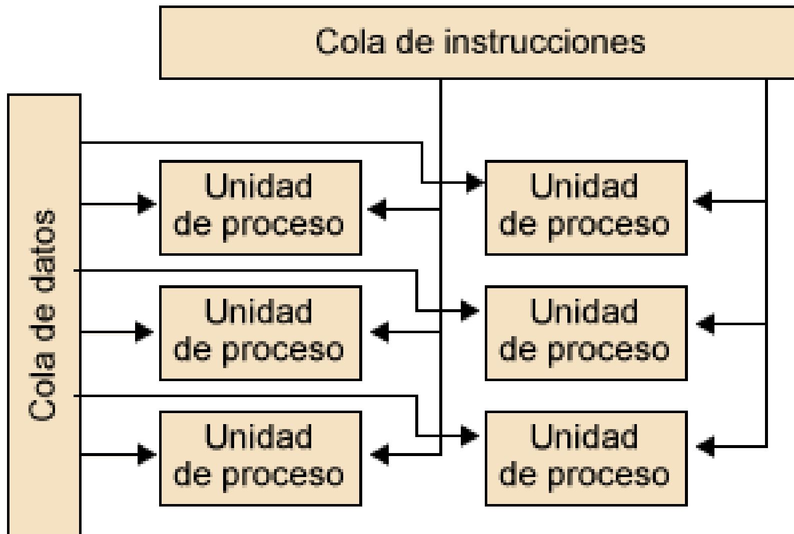


4) Múltiples instrucciones, múltiples datos[4]**:** están formadas por múltiples unidades de proceso que trabajan sobre diferentes flujos de datos de manera paralela. Como muestra la figura siguiente, la arquitectura está formada por diferentes unidades de proceso que tienen diferentes flujos de instrucciones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

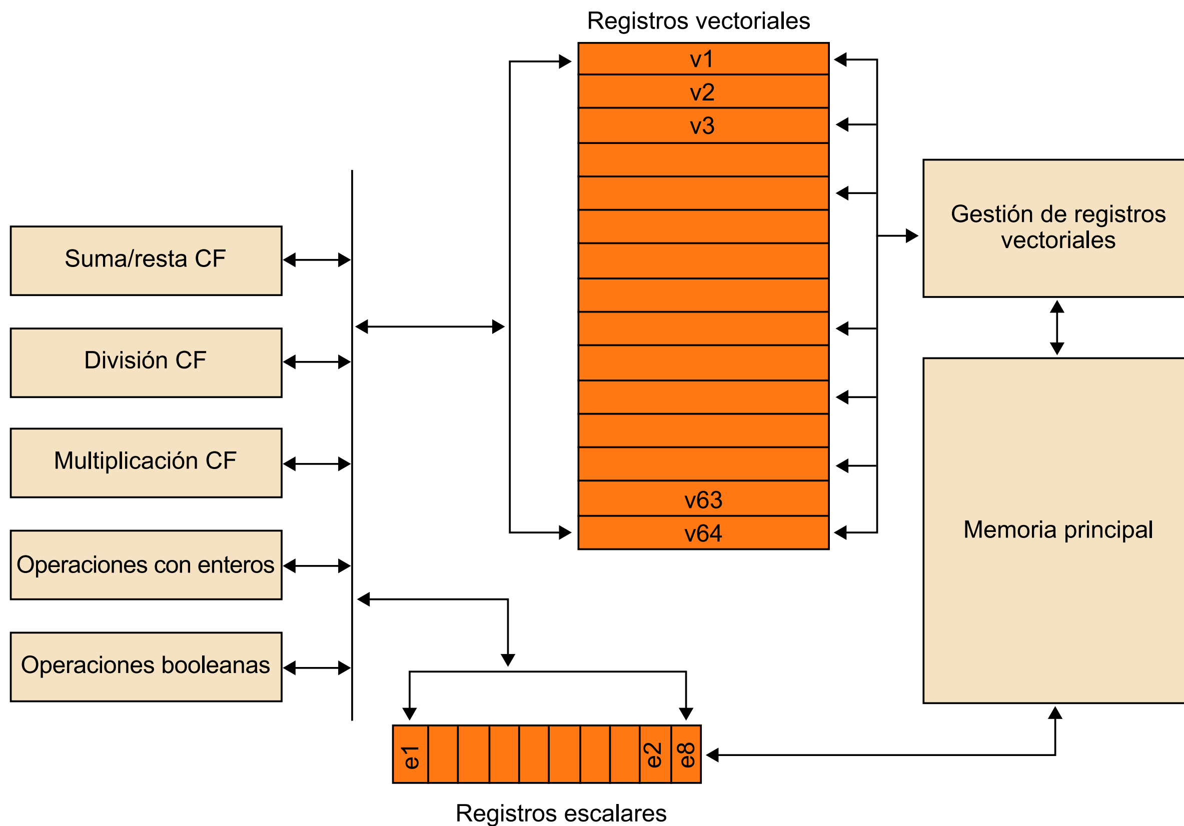
Un ejemplo de ello son los *broadcasts* o los *shuffles*. Algunos de los juegos de instrucciones más conocidos son las *advanced vector extensions* (AVX; I. Corp, 2011) o las *streaming SIMD extensions* (SSE; I. Corp, 2007).

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.

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.

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.

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 incluye 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.

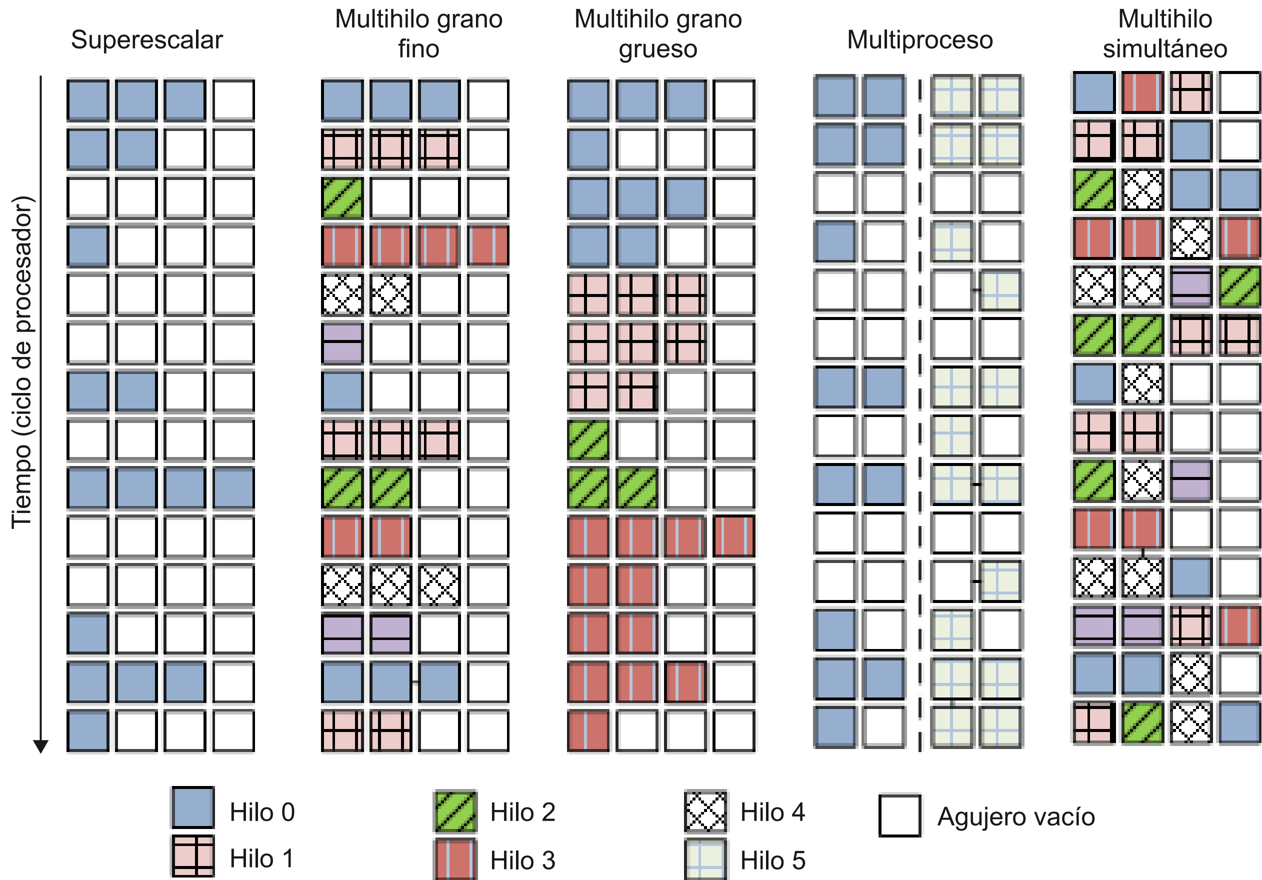
Aplicaciones paralelas

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

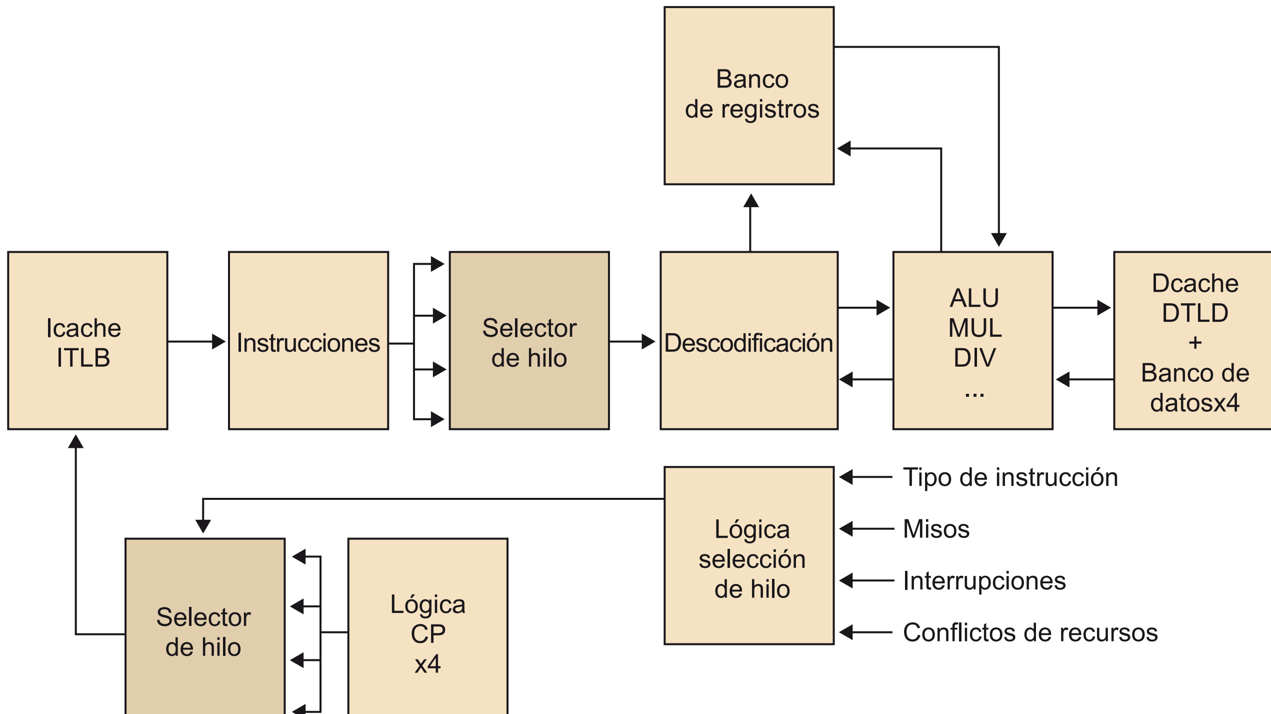
El UltraSPARCT1 (Kongetira, Aingaran y Olukotun, 2005) es un ejemplo de este tipo de procesador.

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*.

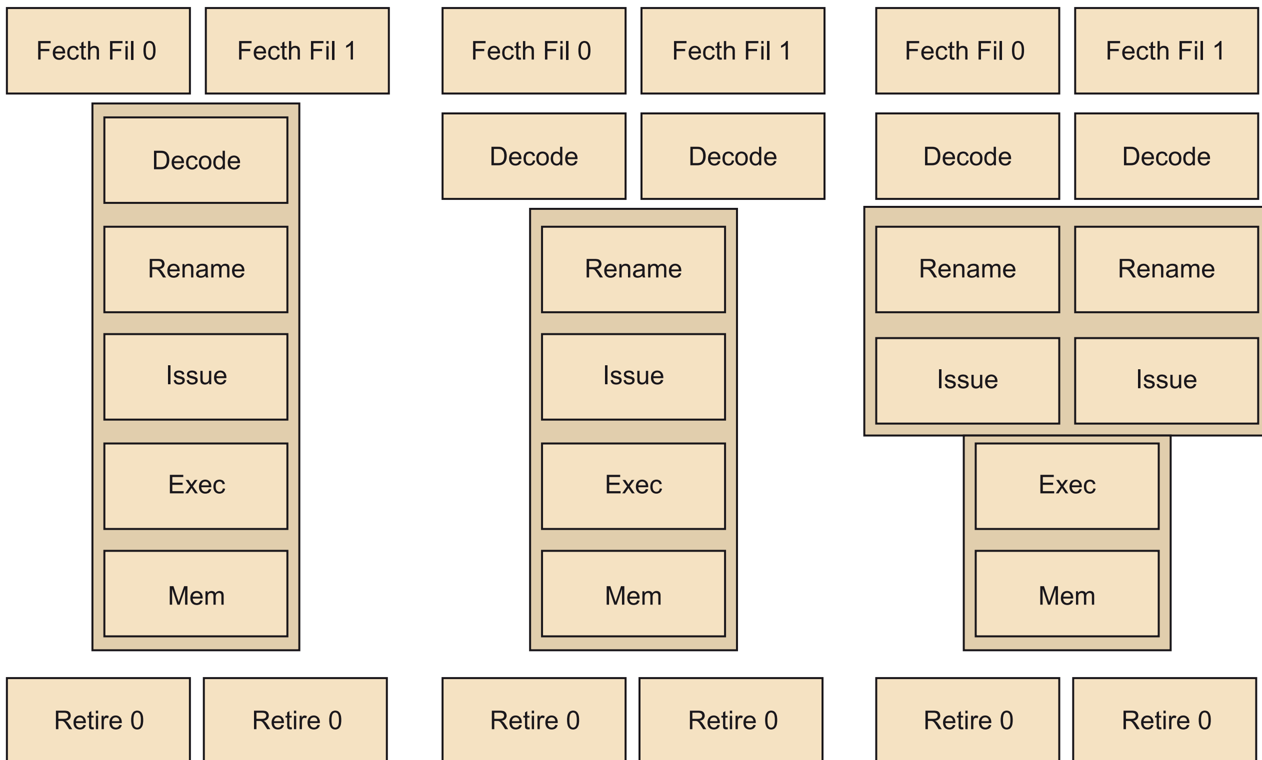
4.4. Diseño de un *SMT*

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. Arquitecturas multinúcleo

4.6.1. 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.

4.6.2. 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.

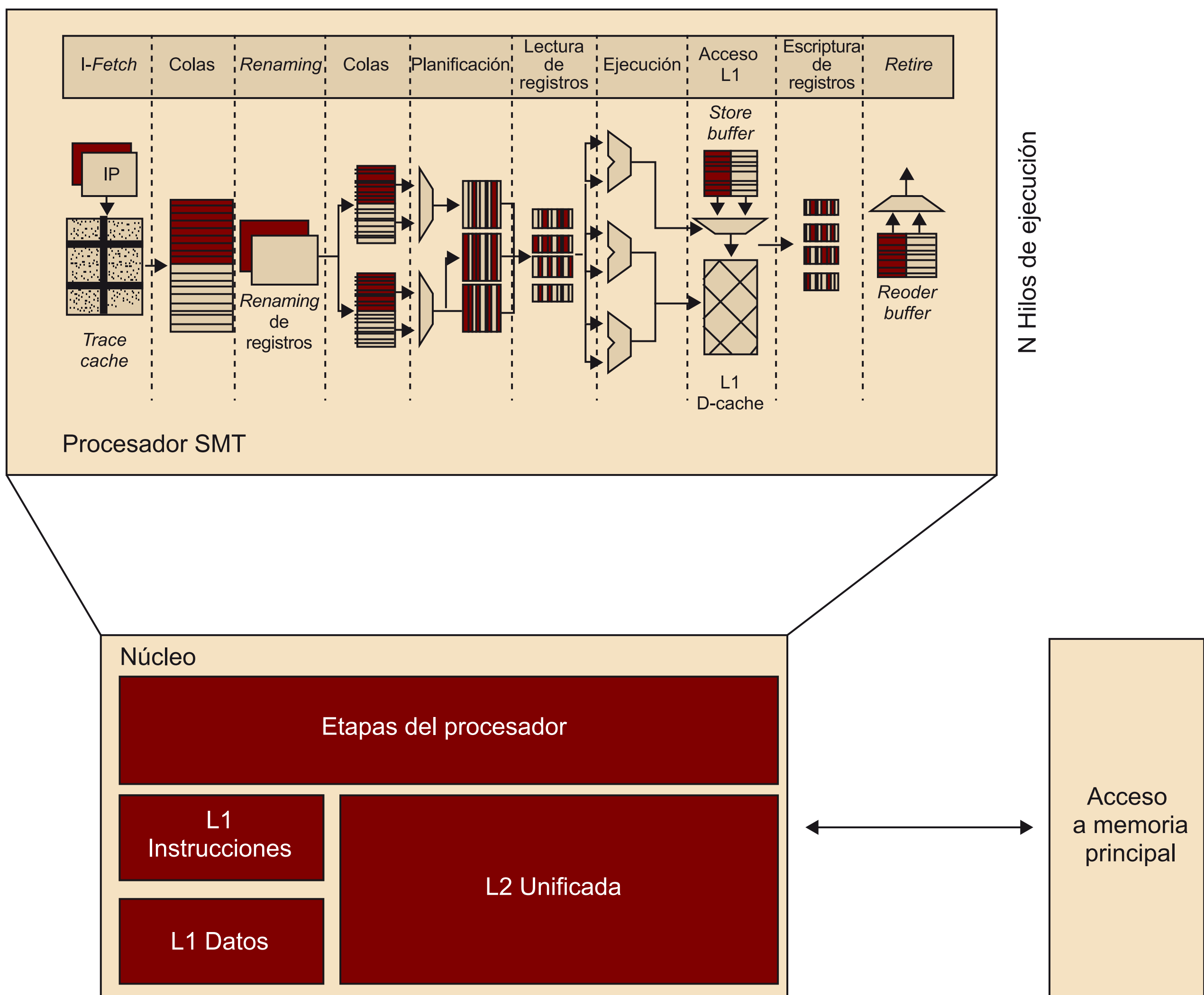
> Por ejemplo, en el caso de la familia SandyBridge, se pueden encontrar los procesadores domésticos con 12 hilos y 130 vatios de consumo, procesadores para dispositivos móviles de 8 hilos y 55 vatios de consumo y procesadores para servidores con 16 hilos y 150 vatios de consumo. Justo es decir que no 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.

4.6.3. 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 12. Abstracción de procesador multihilo

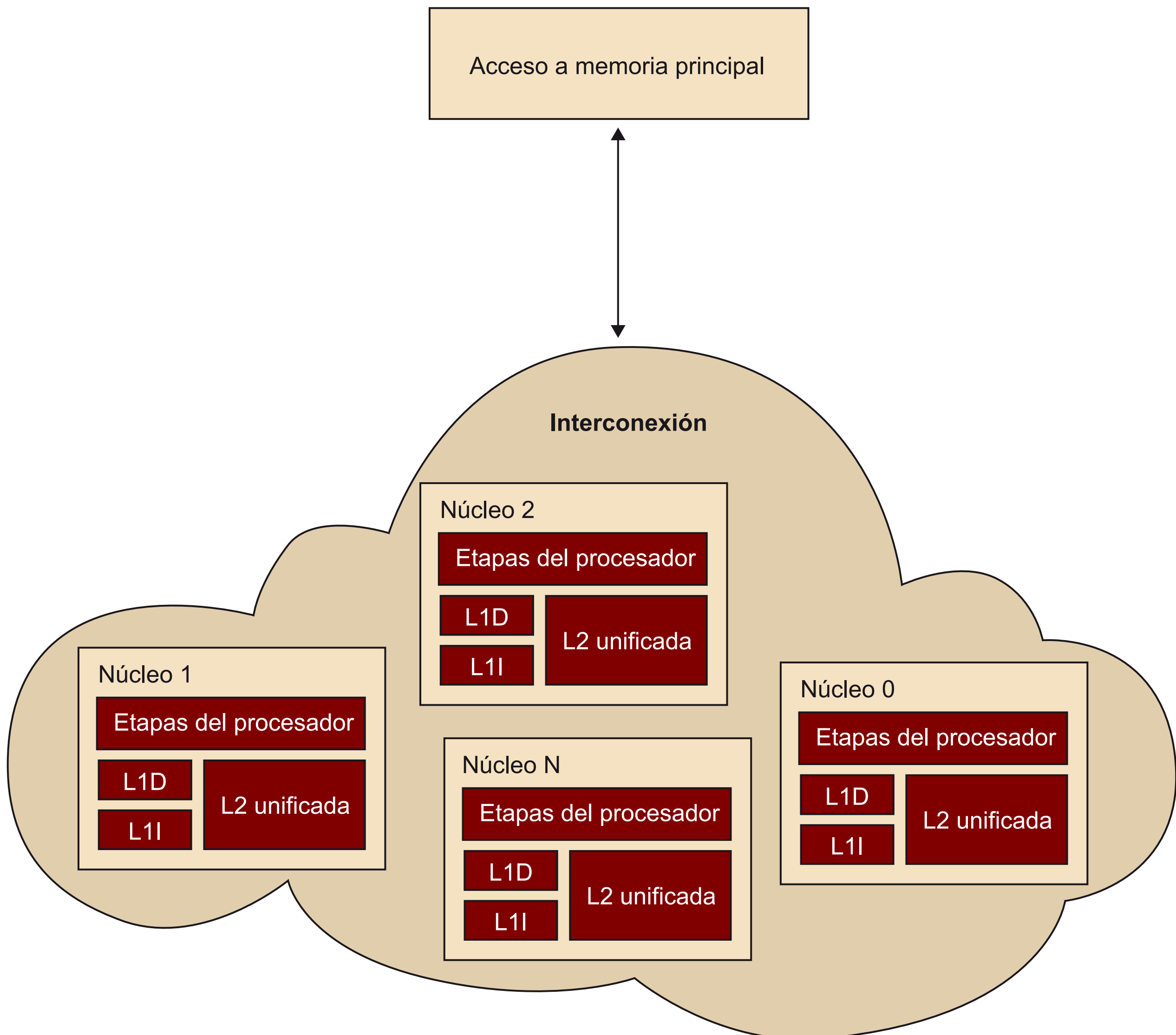


De hecho, el concepto de multinúcleo, como indica su nombre, consiste en replicar *m* núcleos diferentes dentro de un procesador (figura 16). Cada núcleo acostumbra a tener una memoria caché de primer nivel (de datos e instrucción), y puede tener una memoria de segundo nivel (acostumbra a ser unificada: 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 13. 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.

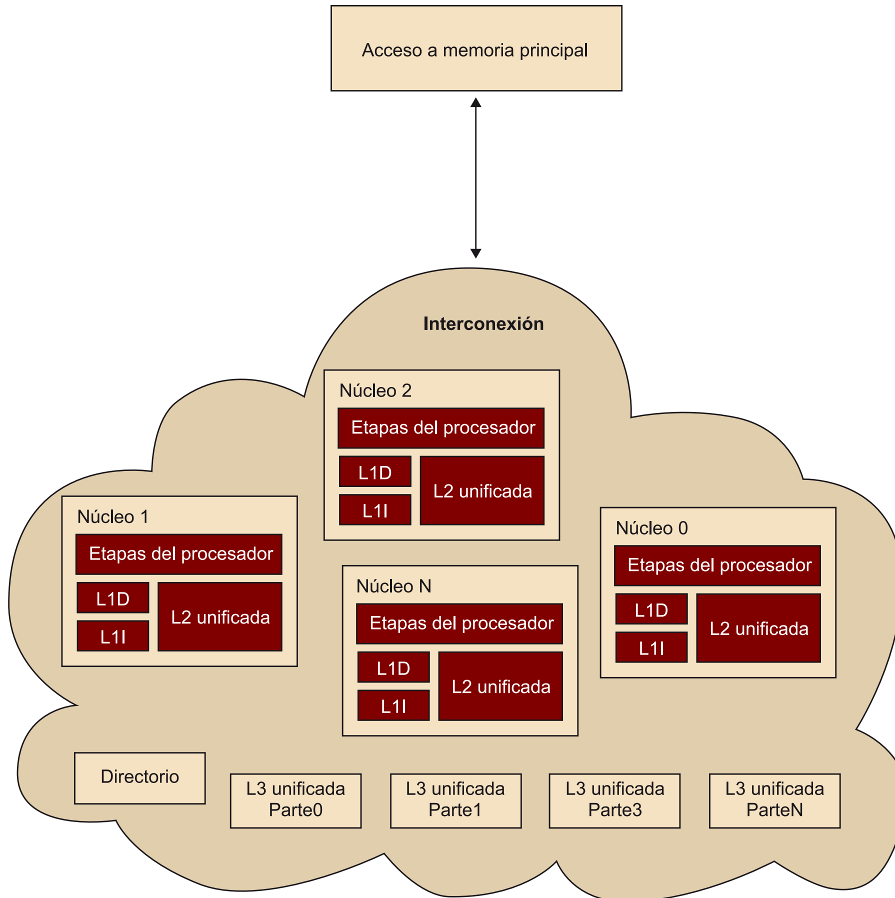
4.6.4. Coherencia entre núcleos

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 14 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 14. 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[13].

(13)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).

5. Arquitecturas *many-core*: el caso de Intel Xeon Phi

En este apartado, nos centraremos en arquitecturas *many-core* utilizando una de las familias de procesadores *many integrated core* de Intel como hilo conductor. Como se verá durante los próximos apartados, estos procesadores se caracterizan por dar acceso a un número muy elevado de hilos de ejecución. De este modo, aplicaciones que son altamente paralelas pueden sacar un rendimiento bastante más elevado en comparación con otras arquitecturas disponibles en el mercado. Por otro lado, un aspecto que las hace muy atractivas respecto al uso de GPU (que también da acceso a un número muy elevado de hilos de ejecución) es que mantienen una visión coherente del espacio de memoria, mientras que la mayoría de las GPU actuales no lo hace.

En primer lugar, se presentará la evolución histórica (hasta el momento de escribir este material) de la familia Xeon Phi, así como sus orígenes. A continuación, se mostrarán las características más importantes de los dos procesadores que se han presentado hasta este momento (Knighs Ferry y Knights Corner). Finalmente, se dará un conjunto de referencias recomendadas para profundizar en las arquitecturas Xeon Phi.

5.1. Historia de los Xeon Phi

En el año 2010, la compañía de procesadores Intel anunció el primero de los procesadores Intel que incluiría un gran número de núcleos integrados. Este número sería mucho más elevado de núcleos respecto a los procesadores diseñados hasta aquel momento. Este concepto, ya conocido en la literatura de computación de altas prestaciones, se conoce como *many integrated cores* ('muchos núcleos integrados').

Esta familia de procesadores, denominada Intel Xeon Phi, heredaba su diseño conceptual del proyecto Larrabee. El objetivo del mismo era diseñar una GPU (*graphical processing unit* o tarjeta gráfica) donde los cómputos se llevaran a cabo en unidades de proceso de propósito generales (x86). El proyecto en cuestión no salió a la luz, pero toda la investigación efectuada se empleó para transformar este sistema multinúcleo en un procesador de propósito general que diera acceso a un número muy elevado de hilos de ejecución. Para más detalles sobre el proyecto Larrabee, se recomienda la lectura del artículo de Selier, Carmean y Sprangle (2008). Otra de las ventajas más importantes de estas arquitecturas era el bajo consumo energético respecto a los competidores de otras compañías.

Desde el 2010, Intel ha anunciado tres nuevos procesadores dentro de la familia de los Xeon Phi: Knights Ferry (KNF), Knights Corner (KNC) y Knights Landing (KNL).

El primero de todos, KNF, fue un prototipo que solo se entregó a centros de investigación o centros de supercomputación para empezar a hacer pruebas y evaluaciones de rendimientos con esta nueva familia de procesadores. No se hizo ninguna venta comercial del mismo. El objetivo era que sus potenciales usuarios finales empezaran a ver cómo había que adaptar las aplicaciones (si era necesario) e hiciesen proyecciones de qué rendimiento obtendrían de las mismas. Como características más importantes, hay que destacar que estaba compuesto por 32 núcleos con cuatro hilos por núcleo, estaba construido sobre un proceso de 45 nm y venía con 2 Gb de memoria integrada. Este era el primero de los procesadores con objetivos comerciales que Intel facilitaba con 128 hilos de ejecución.

El segundo, KNC, fue la versión ya comercial de KNF y se presentó en el 2012. Esta arquitectura se fundamentaba en el mismo diseño que KNF. No obstante, daba acceso un número más elevado de núcleos (50 núcleos en la primera versión), incorporaba una memoria integrada mucho más elevada (hasta 16 Gb) y usaba un proceso de producción más eficiente (22 nm). Utilizando esta nueva arquitectura, centros de computación como el Texas Advanced Computing Center o el Guangzhou construyeron supercomputadores que se colocaron en las primeras posiciones del Top 500 (*Top 500*) y del Green Top 500 (*500*). El primero de los dos construyó Stampede (TACC), un sistema formado por un total de 102,400 núcleos y capaz de 9.5 petaflops. El segundo construyó el computador denominado Thiane-2 (Top500, *China's Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500*). Este es capaz de llegar a un rendimiento máximo de 33.8 petaflops y en su momento de lanzamiento ocupó la primera posición de los computadores más potentes del mundo (*Top 500*).

Finalmente, el último de todos, KNL, se anunció a finales del 2013. No obstante, en el momento de escribir este documento la información que Intel ha hecho pública es muy escasa. Lo más destacable es que esta arquitectura sería un cambio importante respecto a la última generación (KNC). Hay que remarcar que KNC había sido una evolución de KNF.

El diseño interno de KNL probablemente será bastante distinto al que se verá durante el próximo apartado. Como característica más remarcable, este procesador llevará integrado un sistema de memoria denominado *high memory bandwidth* y que aparte de dar acceso a una mayor cantidad de memoria que sus predecesores, permitirá acceder a la misma a través de un ancho de banda mucho más elevado. Esto resulta especialmente importante teniendo en cuenta que muchas de las aplicaciones HPC se encuentran limitadas, por un

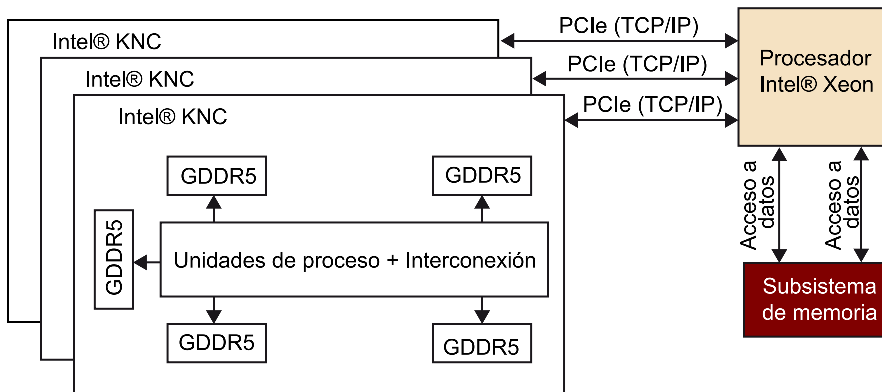
lado, por la cantidad de memoria que pueden usar pero por otro lado (y en algunos casos, más importante) por la cantidad de datos por segundo que el procesador puede seguir.

Como ejercicio de lectura e investigación, se recomienda que una vez acabada la lectura de este apartado busquéis en la Red las novedades de esta familia de procesadores: nuevos modelos, prestaciones etc. Hay que tener presente que estamos hablando de un mercado muy reciente y con constante evolución. Por lo tanto, lo que se cubre puede no incluir novedades más recientes o cambios importantes incorporados en nuevas generaciones.

5.2. Presentación de los Xeon KNC y KNF

Como ya se ha comentado en la última subsección, la arquitectura de un KNC es una extensión comercial de la de KNF aumentando la cantidad de núcleos y de memoria integrada. En este apartado, se cubren las características arquitectónicas comunes a las dos arquitecturas. Como ya se ha mencionado, en el momento de escritura de esta documentación la arquitectura de de los Xeon KNL no se había hecho pública. Por lo tanto, no se discutirán sus detalles.

Figura 15. Visión global de un KNC



La figura 15 muestra una visión global de cómo un sistema de computación puede construirse empleando KNC (a partir de este punto, se empleará KNC para hacer referencia a KNC o KNF de manera indistinta; en caso contrario, se especificará). Como se puede observar, los KNC se encuentran conectados a un procesador Xeon tradicional como podría ser un Sandy Bridge (*Corporation*, *Sandy Bridge Server Products*) o *Haswell* (*Corporation*, *Haswell Server Products*) a través de PCIexpress. Esto se debe a que tanto KNC como KNF se diseñaron como coprocesadores y no son capaces de arrancar un sistema operativo convencional (por ejemplo, Linux o Windows). Por lo tanto, necesitan un procesador completo que permita arrancar un sistema operativo y que el usuario interactúe con el sistema (ejecutar aplicaciones, gestión de ficheros etc.).

Del mismo modo que se hace con los ordenadores convencionales y las tarjetas gráficas (GPU), los KNC se colocan en los PCIexpress del computador. Cuando se ejecuta una aplicación, esta se instancia al procesador principal (denomi-

nado también *host*). El hilo principal de la aplicación se ejecuta en el mismo. No obstante, usando un conjunto de librerías que Intel facilita (Corporation, *Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide*, 2013), la aplicación es capaz de mover datos al KNC e instanciar hilos de ejecución en este.

```
float reduction(float *data, int size)
{
    float ret = 0.f;
    #pragma offload target(mic) in(data:length(size))
    for (int i=0; i<size; ++i)
    {
        ret += data[i];
    }
    return ret;
}
```

Código 1. Ejemplo de *offload*

El código 1 muestra un ejemplo de cómo el programador puede especificar qué parte del código se quiere ejecutar en el *host* y cuál se ejecuta en el KNC. Como se puede observar, por defecto el código se ejecutará en el procesador principal (*host*). Ahora bien, si se quiere ejecutar una parte de este en el coprocesador, hay que especificarlo usando lo que en el mundo de programación se conoce como pragmas. Estos permiten especificar al compilador y a las librerías qué conjunto de líneas se quieren ejecutar fuera y cómo se tienen que ejecutar. En el ejemplo anterior, se pide ejecutar el bucle en el KNC y se especifica que hay que transferir el contenido apuntado por la variable *data* que tiene un tamaño *size*.

Tal y como es posible observar en la figura 15, se pueden construir sistemas compuestos por un solo procesador *host* y tantos coprocesadores KNC como entradas PCIexpress tenga la placa base que se está empleando. En ningún caso es posible comunicar dos KNC conectados a la misma placa: siempre se tiene que hacer a través del procesador principal. Este es el encargado de interaccionar con todos los KNC que forman parte del sistema.

Una de las novedades importantes del más reciente de los Xeon Phi KNL es que este sí será *bootable,* es decir, será capaz de arrancar un sistema operativo completo y ejecutar aplicaciones por sí solo. Esta es una propiedad muy interesante, pues se podrán construir sistemas de altas prestaciones sin tener que usar forzosamente procesadores de servidor convencionales. Estos sistemas podrán estar compuestos únicamente por procesadores KNL.

Durante los últimos párrafos, se ha mencionado que los datos se transmiten del procesador principal al coprocesador. Esto tiene tres implicaciones importantes. La primera es que los KNC han de tener memoria propia para almacenar estos datos. Tal y como se puede observar, los KNC emplean memoria GDDR5 (Jedec) para almacenar datos. Las diferentes versiones disponibles en el mercado comprenden versiones desde 6 Gb hasta 16 Gb las más potentes.

La segunda implicación importante es que el *host* y el KNC no comparten espacio de memoria virtual. Es decir, las variables que se instancien al código ejecutado en el *host* no serán visibles por las partes de código ejecutadas dentro del KNC, y viceversa. Si se quiere compartir datos, será necesario especificar dentro del código qué variables se desea mover del *host* hacia el KNC y de los KNC hacia el *host*.

La tercera implicación puede afectar a la eficiencia de las aplicaciones que se diseñen. Es decir, hay que minimizar y optimizar al máximo las comunicaciones que se hacen del *host* al KNC. Es preciso estudiar cuál es la mejor manera de mover los datos (por ejemplo, moverlos todos a la vez, a trozos, etc.). Hay que tener presente que el ancho de banda que un KNC puede dar de memoria principal es 352 Gb/s en la versión de 16 Gb (*Pcwire*, 2012), y el que puede dar el PCIe puede ser de 8 Gb/s en un PCIexpress x16 (*Anandtech*). Por lo tanto, el subsistema de memoria de un KNC dará un ancho de banda 40 veces más rápido que el que da el PCIexpress.

5.3. Arquitectura y sistema de interconexión

En el apartado anterior se han dado las características más importantes del funcionamiento de un sistema de computación construido con procesadores de la MIC. Como se ha visto, la comunicación entre diferentes nodos KNC y el procesador principal se lleva a cabo a través de una conexión PCIexpress. Ahora bien, ¿cómo está diseñado internamente un KNC? ¿Cómo están interconectados los diferentes núcleos y la memoria interna del mismo, y cómo se conectan con el mundo anterior?

La figura 16 muestra la arquitectura interna de un KNC. Como se puede observar, este contiene cuatro tipos diferentes de agentes:

- El agente PCIe es el encargado de comunicarse con el *host*. Cada vez que uno de los núcleos accede al espacio de memoria que se encuentra mapeado en el *host*, la petición del núcleo acabará llegando al *host* a través de este agente. Del mismo modo, cada vez que el *host* quiere enviar datos o peticiones al KNC, lo hará usando este agente.

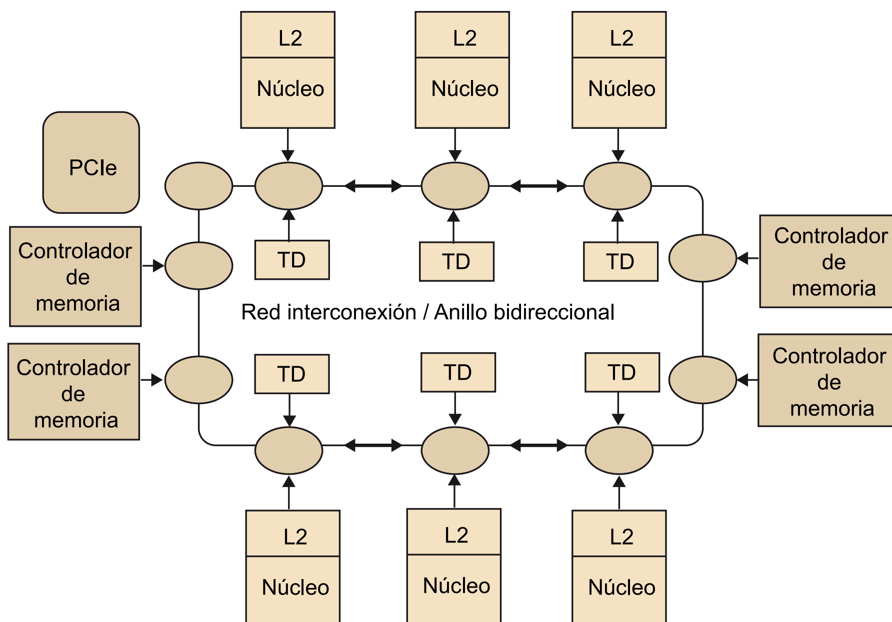
- El agente núcleo es el encargado de llevar a cabo los cálculos y las acciones de los distintos hilos que la aplicación ha instanciado. Como ya se ha mencionado anteriormente, cada núcleo tiene cuatro hilos de ejecución y contiene dos niveles de memoria caché (más adelante, se presentan más detalles de esto).

- El agente TD o *tag directory* se encarga de mantener la coherencia de memoria. Cada vez que un núcleo quiere acceder a una dirección de memoria, lo tiene que pedir al TD pues es el encargado de gestionar la coherencia de la dirección en cuestión. Cada dirección del espacio de memoria es gestionada por tan solo un TD. Como se verá más adelante, cuando este

recibe una petición para acceder a un dato tiene que controlar que ningún otro núcleo lo tenga antes de pedirlo a memoria. En caso contrario, deberá notificarlo al núcleo que lo tenga para que este actualice su estado.

- Finalmente, el último agente es el controlador de memoria. Este agente se encarga de satisfacer las peticiones de acceso a memoria principal que reciben del TD y que han sido originadas por uno de los núcleos. Cuando este recibe una petición de lectura o escritura, el controlador traduce esta petición a los pedidos específicos que el dispositivo de GDDR5 entiende. Para más información de cómo un controlador de memoria interacciona con un dispositivo GDDR5, se recomienda la lectura de la introducción a esta tecnología (Jedec).

Figura 16. Arquitectura interna de un KNC



Los diferentes agentes se encuentran conectados a través de un anillo bidireccional. Este es el responsable de llevar a cabo la comunicación entre los distintos núcleos, TD, controladores de memoria y agentes PCIexpress necesarios para mantener la coherencia del espacio de memoria y comunicarse con el *host*. Es importante remarcar que pese a parecer una pieza sencilla de hardware, se trata de una pieza altamente compleja que ha de tener en cuenta muchos aspectos importantes: mecanismos de balanceo, mecanismos de descongestión en caso de mucha carga, mecanismos para evitar bloqueos (también denominados *deadlocks*), etc.

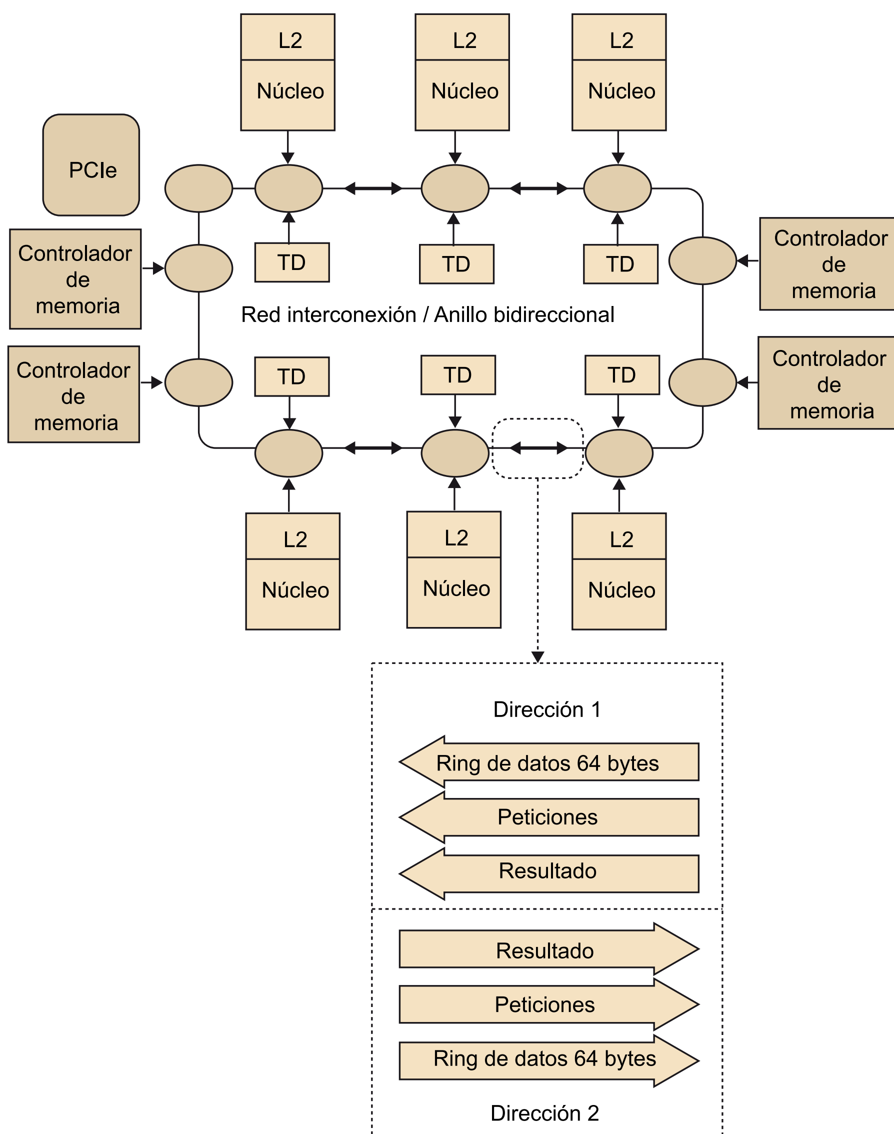
Este anillo bidireccional tiene que ser capaz de transmitir información de tres tipos distintos:

- Peticiones entre los diferentes agentes. Por ejemplo, la petición de una línea de memoria en exclusiva de un núcleo a un TD; o bien una petición de lectura de una línea de memoria de un TD a un controlador de memoria.

- Datos que se envían entre los diferentes agentes y generados como consecuencia de una petición que se ha iniciado anteriormente. Por ejemplo, después de la petición de lectura de una línea de memoria por parte de un núcleo alguno de los controladores de memoria acabará enviando los datos pedidos al núcleo que ha enviado la petición.

- Respuestas de confirmación o resultado que, como en el caso de los datos, han sido generadas como consecuencia de una petición. Por ejemplo, cuando un TD recibe una petición de lectura de una línea de memoria, este tiene que responder al núcleo que la transacción se está procesando correctamente y cuál es el estado con el que se retornará la línea en cuestión (exclusiva o compartida).

Figura 17. Anillo del *ring* bidireccional



Tal y como se muestra en la figura 17, el anillo bidireccional está formado por tres subanillos distintos (tres en cada dirección). El primero de todos, denominado AD, es el responsable de transportar las peticiones. El segundo, denominado BL (nombre diminutivo de ***bl**ock*), es el responsable de transportar

los datos. Y finalmente, el tercero, denominado ACK (nombre diminutivo de ***ack****nowledgement*), es el responsable de transmitir confirmaciones y resultados de las peticiones.

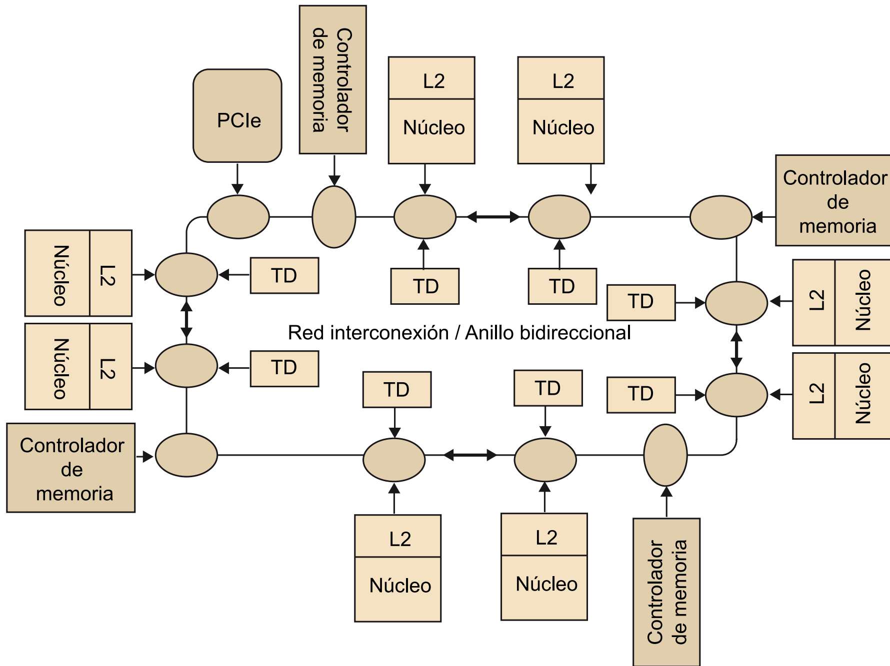
Cada uno de estos subanillos tiene un tamaño bastante diferente. El más ancho de todos es el de BL. Este tiene un ancho total de 512 bits (tamaño de la línea de 64 bytes de memoria) más la cabecera (necesaria para incorporar información de ruta: destinatario, origen, identificador de la transacción, etc.). El siguiente, el AD, es menos ancho que este primero. Este solo tiene que transportar el tipo de petición, la dirección física de la línea a la que hace referencia (48 bits) y, como en el caso anterior, la información de ruta. Finalmente, el anillo de AK es el que menos ancho necesita. Este tan solo tiene que llevar la respuesta, el identificador de la transacción a la que hace referencia y la información de ruta.

Es importante remarcar que el diseño de este anillo bidireccional tiene como objetivo dar bastante ancho de banda para que sea posible satisfacer todas las transferencias entre núcleos y el ancho de banda que cada uno de los dispositivos de GDDR5 da. Es muy importante tener en cuenta que si cada uno de los dispositivos puede dar un total de 80 Gb/s de lectura más escritura en memoria, el diseño del sistema debe ser capaz de soportar el ancho de banda que todos los controladores pueden dar (en la versión más potente, hasta 350 Gb/s). En caso contrario, no se estaría usando toda la potencia que da el sistema de memoria. Esto no sería aceptable teniendo en cuenta que se trata de uno de los componentes más caros de un sistema.

Tal y como muestra la figura 18, los KNC tienen un total de cuatro controladores de memoria. Cada controlador da acceso a una cuarta parte de la memoria total disponible. Por ejemplo, en el caso de tener la versión con 8 Gb, cada controlador dará acceso a dispositivos de GDDR5 de 2 Gb con un ancho de banda de 80 Gb/S. Tal y como se ha mencionado, el diseño de la red de interconexión tendrá que ser capaz de saturar los cuatro controladores.

En este apartado no se profundizará en los detalles de este diseño. En cualquier caso, si este es de interés para el lector, se recomienda la lectura del módulo de redes de interconexión así como la extensión de los conocimientos a través de lecturas de trabajo relacionado disponibles en la Red. Como ya se ha mencionado, un diseño de un sistema de comunicación de estas características requiere tener en cuentas muchos factores decisivos a la hora de obtener el rendimiento esperado.

Figura 18. Arquitectura de un KNC



5.4. Núcleos de los KNC

Tal y como ya se ha mencionado anteriormente, la primera generación de Xeon Phi incorporaba un número más limitado de núcleos respecto a KNC (un total de 32 núcleos). No obstante, la arquitectura de los dos era exactamente la misma. En este apartado, se presentarán las características más generales de este así como sus prestaciones.

Antes de discutir las prestaciones de estos núcleos, hay que remarcar que la segunda generación –los procesadores KNC– ofrece un abanico más abierto de configuraciones. Como se puede observar en la tabla 1, la opción más simple incorpora un total de 57 núcleos que pueden correr a una frecuencia de 1.1 GHz, una memoria con un ancho de banda máximo de 240 Gb/s con una capacidad de 6 Gb y un rendimiento máximo (en doble precisión) de 1003 gigaflops. Por otro lado, la opción más agresiva de este modelo incorpora un total de 61 núcleos que pueden correr a 1.2 GHz y ofrecen un rendimiento máximo de 1208 gigaflops, e incluye también una memoria con un ancho de banda máximo de 352 Gb/s y capacidad de 16 Gb/s.

Tabla 1. Opciones de KNC disponibles (finales del 2013)

Modelo	Consumo (vatios)	Número núcleos	Frecuencia (GHZ)	Rendimiento máximo (GFLops)	Ancho de banda memoria	Capacidad memoria
3120P	300	57	1.1	1003	240	6
3120A	300	57	1.1	1003	240	6
5110P	225	60	1.053	1011	320	8
5120D	245	60	1.053	1011	352	8

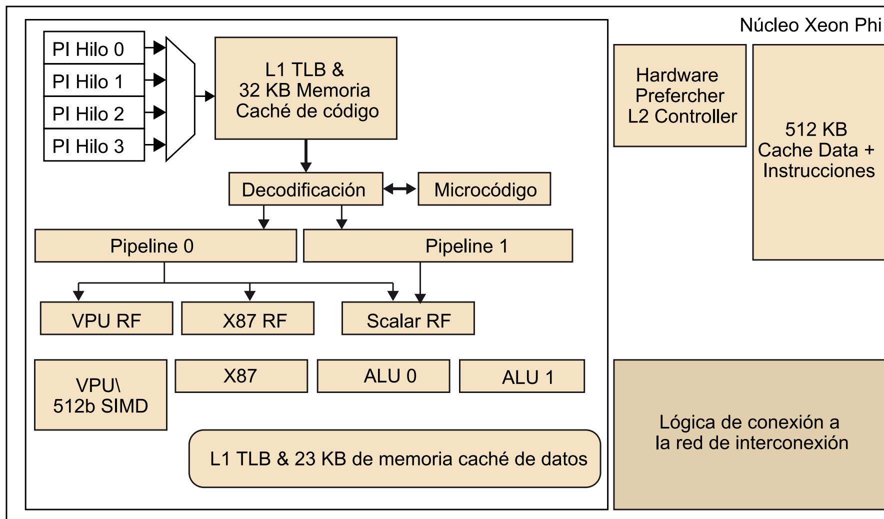
Modelo	Consumo (vatios)	Número núcleos	Frecuencia (GHZ)	Rendimiento máximo (GFLops)	Ancho de banda memoria	Capacidad memoria
7110P	300	61	1.238	1208	352	16
7120X	300	61	1.238	1208	352	16

La diferencia fundamental entre cada una de las opciones mostradas en el punto anterior en lo que respecta a los núcleos es la frecuencia a la que estos se ejecutan, así como la cantidad de memoria a la que dan acceso. Su arquitectura interna es exactamente la misma.

La figura 19 muestra los elementos fundamentales que definen su diseño. Como se puede observar en la parte superior, este contiene una estructura que almacena la información necesaria para ejecutar cuatro hilos de ejecución. A modo simbólico solo se muestra el puntero de instrucción (PI), pero este almacena otros datos necesarios para gestionar su flujo de ejecución. Como ya se ha visto en otros módulos, esto permite que las aplicaciones puedan instanciar hasta cuatro *hardware threads* o hilos para cada uno de los núcleos.

A continuación, podemos ver que cada uno de los hilos se extrae de una memoria caché de primer nivel (solo de instrucciones) de 32 KBS. Como ya se sabe, las etapas de un procesador no trabajan con direcciones virtuales, sino físicas. Por este motivo, también en esta primera parte encontramos el *TLB* (*translation lookaside buffer*). Esta estructura permite traducir las virtuales a físicas. Una vez la instrucción se extrae de la L1 de instrucciones y se ha hecho el TLB, las instrucciones entran en la fase de decodificación. En esta fase, el núcleo se encarga de llevar a cabo todas las comprobaciones necesarias para ejecutar la instrucción (dependencias, riesgos estructurales, etc.) y de traducir las instrucciones denominadas instrucciones marco en instrucciones más simples, denominadas microinstrucciones. Esta es una característica común a todas las arquitecturas x86 (*Corporation*, *Introduction to x64 Assembly*).

Figura 19. Arquitectura de un núcleo de KNC



Las siguientes etapas ya son propiamente las etapas de cálculo y de acceso a memoria en caso de ser necesario. Una de las características más remarcables de esta arquitectura son las unidades vectoriales que esta incorpora, denominadas *vector processing unit* (*VPU*). Estas unidades vectoriales permiten ejecutar 16 operaciones de precisión simple u 8 de doble precisión por ciclo, y permiten ejecutar instrucciones FMA. Las FMA (*fuse-multiply and add*) permiten llevar a cabo la suma de un elemento más un segundo elemento multiplicado por un tercer elemento (A = A + B*C). Dado que estamos hablando de una unidad vectorial, tanto A como B y C son vectores. Por lo tanto, las FMA permiten efectuar hasta 32 operaciones de precisión simple o bien 16 de doble precisión por ciclo. Estas operaciones, altamente empleadas en el mundo de las aplicaciones de altas prestaciones, permiten obtener un buen rendimiento de cálculo.

Aparte de la memoria caché de instrucciones de primer nivel, cada núcleo contiene una memoria de datos de primer nivel de 32 Kb (junto con su TLB). Como segundo nivel de memoria, contiene una L2 de 512 Kb. En este caso, la L2 es unificada. Es decir, contiene tanto datos como instrucciones.

Uno de los elementos interesantes que un núcleo KNC tiene es el *hardware prefetcher*. Este, tal y como ya se ha introducido en otros módulos, es el encargado de pedir datos de memoria antes de que un hilo los pida. Esto se hace por medio de algoritmos que intentan predecir a qué direcciones de memoria accederá en un futuro el hilo en cuestión. De este modo, cuando este las pida, ya se encontrarán almacenadas en la memoria caché y no será necesario ir a buscarlas a la memoria principal (lo cual implicaría una cantidad de ciclos muy superior).

5.5. Sistema de coherencia

Los KNC implementan un sistema de coherencia *MESI* (*modified, exclusive, shared, invalid*) que implementa un sistema *GOLS* (*globally owned locally shared*). Como ya se ha discutido anteriormente, el hecho de tener un sistema de cohe-

rencia permite asegurar que en todo momento la visión de memoria por parte de todos los núcleos es coherente. Por lo tanto, se asegura que dos núcleos que acceden a la línea de memoria @X tendrán una visión coherente de la misma (el mismo valor). También se asegura que si un núcleo quiere modificar su valor hacia otro núcleo, tendrá una copia que sea diferente a esta.

Tabla 2. Definición de MESI

Estado	Definición	Propietarios de la línea	Estado respecto a memoria
M	Modificada	Solo un núcleo es propietario de la línea.	Modificada.
E	Exclusiva	Solo un núcleo es propietario de la línea.	No modificada.
S	Compartida	Un conjunto de núcleos contienen la línea.	Puede estar modificada o no.
I	Invalida	Ningún núcleo contiene la línea.	-

La tabla 2 muestra los diferentes estados en los que se puede encontrar una línea de memoria en un momento determinado de la ejecución dentro de una L1 o L2 de un núcleo. Por el hecho de ser un sistema multinúcleo, una línea de memoria se puede encontrar ubicada en diferentes memorias caché de distintos núcleos. El sistema GOLS, definido en la tabla 3, extiende la definición de MESI añadiendo cuatro estados más que se usan en un ámbito global de sistema. Cada núcleo guarda el estado de la línea siguiendo un protocolo MESI y los TD, encargados de mantener una visión global coherente, implementan el protocolo GOLS.

El estado GOLS permite saber al TD que a pesar de que los diferentes núcleos tienen una línea en estado compartido o S, había sido previamente modificada por un núcleo (cuando solo esta la poseía). Por lo tanto, cuando el último núcleo que contenga la línea la quiera invalidar, será necesario escribirla en memoria. Por ejemplo:

1) El núcleo 1 pide la línea @Y al TD 1 y la modifica. Esta pasa de estado E en el núcleo a M.

2) El núcleo 2 pide la línea @Y al TD 1. El TD primero enviará una notificación al núcleo 1 para que pase en estado S y este le responderá que la tenía modificada al final de esta transacción. Los núcleos 1 y 2 tendrán la línea en estado S (por lo tanto, no la podrán modificar) y el TD 1 tendrá apuntado que se encuentra en modo GOLS.

3) El núcleo 1 invalida la línea y lo notifica al TD 1.

4) El núcleo 2 invalida la línea y lo notifica al TD 1. Esta vez, dado que el núcleo 2 es el último en poseer la línea y el TD sabe que se encontraba en modo GOLS, le pedirá al núcleo 2 que envíe los datos a memoria.

Tabla 3. Definición de GOLS

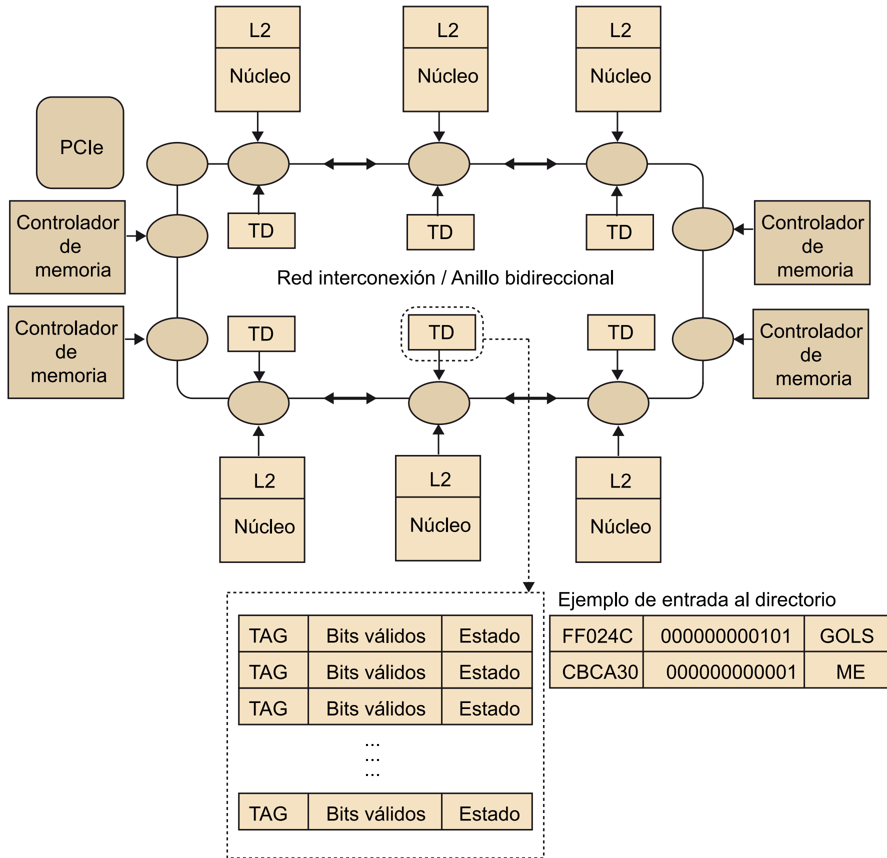
Estado	Definición	Propietarios de la línea	Estado respecto a memoria
GOLES	*Globally owned Locally Shared*	Diferentes núcleos la pueden tener.	Modificada
GM/GE	*Globally Modified / Exclusive*	Solo este núcleo la tiene.	Puede estar modificada o no
GS	*Globally Shared*	Diferentes núcleos la pueden tener.	No ha sido modificada
GI	*Globally Invalid*	Ningún núcleo contiene la línea.	-

Al principio de este apartado, se ha explicado que los TD son los agentes encargados de gestionar la coherencia de memoria. Como ya se ha mencionado, cada línea de memoria es gestionada por un solo TD. Cada núcleo implementa un función que le permite calcular cuál es el TD que gestiona una dirección en cuestión td_id = f_calculo_td(@X). Todos los núcleos implementan la misma función y, de este modo, el sistema se asegura de que las peticiones sobre una dirección irán siempre al mismo TD.

Cada TD contiene una estructura que le permite seguir el estado de cada línea y cuál de los núcleos la tienen. La figura 15 es un ejemplo de esto. Aparte del estado, cada línea tiene asociados lo que se denominan bits válidos. Cada bit corresponde a uno de los núcleos del sistema. Si un núcleo tiene la línea, el bit correspondiente estará a 1. En este ejemplo, la línea con TAG FF024C se encuentra en estado GOLS y la tienen los núcleos 0 y 2. Hay que remarcar que el TD no guarda la dirección de la línea entera, sino el TAG (como ya se ha discutido anteriormente en módulos que introducen la gestión de memorias caché).

Los sistemas de coherencia son una de las piezas más importantes a la hora de determinar el rendimiento que un sistema puede dar. Este es especialmente importante por aplicaciones en las que los diferentes hilos comparten de manera muy frecuente datos o bien tienen que acceder mucho al sistema de memoria. Se recomienda la lectura del artículo de Ramos Garea y Hoefler, 2013. Este presenta una comparativa del sistema de coherencia de un KNC respecto a un Sandy Bridge. No solo muestra una descripción detallada del mismo, sino que también presenta un estudio de rendimiento bastante interesante.

Figura 20. Implementación del directorio del TD



5.6. Protocolo de coherencia

El KNC implementa un protocolo de mensajes entre los diferentes agentes para mantener el estado MESI y GOLS. Este protocolo se implementa sobre los tres subanillos explicados en las últimas subsecciones (AD, BL y AK). Como ya se ha mencionado, las peticiones se generan y se envían a través de AD, las respuestas a través de AK y los datos a través de BL.

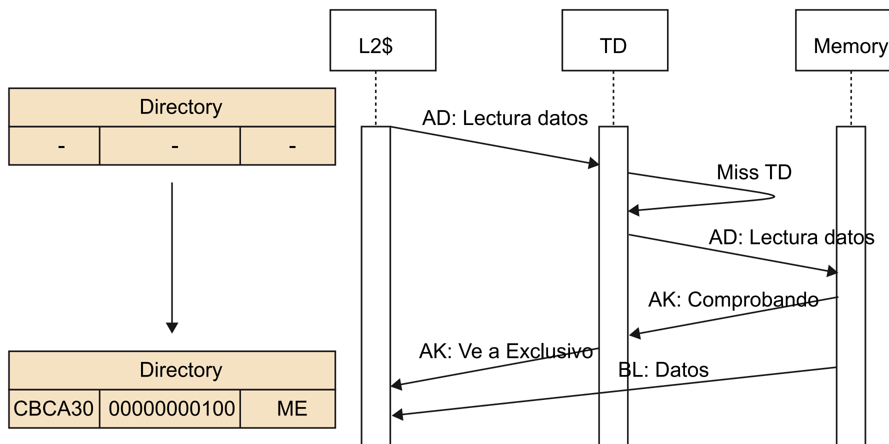
El protocolo que permite definir el estado de coherencia es complejo y contempla muchas situaciones y tipos de peticiones distintos. Hay que tener presente que no solo hay peticiones de acceso a memoria. Podemos encontrar transacciones que piden acceso a zonas de memorias no coherentes, a la zona PCIexpress etc. Cada tipo de transacción tiene asociados unos conjuntos de mensajes y dependencias que tienen que suceder para que esta se lleve a cabo.

A modo de ejemplo, a continuación se describen tres posibles transacciones que pueden pasar en el caso de de una petición de lectura de una línea de memoria por parte de un núcleo.

Petición de una línea de memoria no ubicada en ningún núcleo

El núcleo pide la línea en cuestión al TD a través del anillo de AD. El TD busca en su directorio y ve que esta no se encuentra dentro de ninguna de las L2 del resto de los núcleos. A continuación, el mismo TD pide al controlador de memoria que envíe los datos de la línea al núcleo que la ha pedido. El controlador enviará una confirmación de inicio de transacción al directorio. Este, cuando la recibe, comunica al núcleo que tiene la línea que ha pedido en modo exclusivo (a través de el anillo de AK). En este punto, el director actualiza el estado de la línea a ME y el núcleo lo actualizará a exclusiva una vez reciba, a través del anillo de BL, los datos de memoria. Como se puede observar, el TD ha activado el bit correspondiente de los bits válidos.

Figura 21. Petición de línea de memoria que no tiene ningún otro núcleo

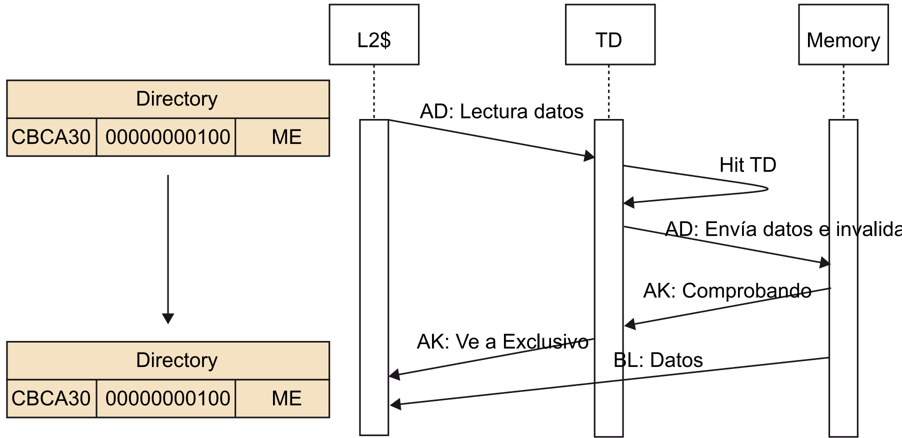


Petición de una línea en estado exclusivo que ya tiene otro núcleo

A diferencia del ejemplo anterior, en este caso se asume que la línea ya estaba ubicada en otro núcleo. Por otro lado, también se asume que el núcleo pide la línea en exclusivo. Esto es necesario en todos los casos en los que el núcleo tiene que modificar el contenido de la línea.

En este caso, cuando el TD procesa la petición del núcleo, la búsqueda dentro de los TAGS es positiva. Como se puede observar en la figura 22, la búsqueda retorna que la línea se encuentra ya ubicada en el núcleo 1 y que se encuentra en estado ME. Dado que el núcleo la está pidiendo en exclusiva, el TD envía una petición al núcleo que tiene la línea de invalidarla y enviar los datos al núcleo que la ha pedido. Como se puede observar, bits válidos se modifican de tal manera que el bit correspondiente al primer núcleo pasa a ser cero y el bit del tercer núcleo (el que ha pedido la línea) pasa a ser 1.

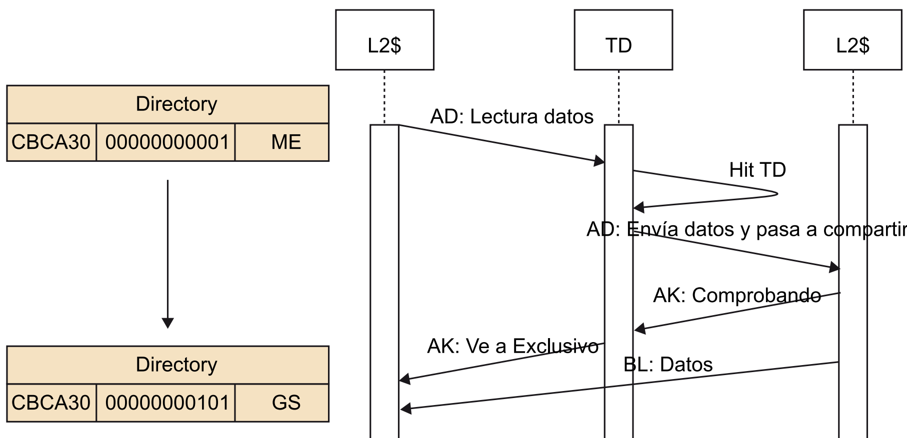
Figura 22. Petición de una línea de memoria que ya tiene otro núcleo



Petición de una línea que ya tiene otro núcleo

En este último ejemplo, a diferencia del caso anterior, el núcleo que pide la línea no pide la exclusividad. Por lo tanto, el TD pide al núcleo que ya posee la línea que la reenvíe al núcleo que la ha pedido. En este caso, sin embargo, notifica al núcleo que poseía la línea en estado exclusivo que pase a compartido o *shared*. Tal y como se puede observar en la parte izquierda de la figura 23, los bits válidos pasan de tener el bit del núcleo 1 a 1 y mantienen el estado del bit del núcleo 1 a 1. De este modo, el TD se apunta que los dos núcleos tienen la línea en cuestión y que esta se encuentra en estado GS.

Figura 23



5.7. Conclusiones

Durante este apartado, se han presentado las pinceladas generales y más representativas de las primeras arquitecturas *many integrated core* que Intel sacó al mercado bajo la familia con nombre Xeon Phi. Se han discutido sus características generales en lo que respecta a sistema, arquitectura, coherencia y núcleo.

No obstante, como ya se ha mencionado, estas contienen una cantidad sustancialmente más elevada de detalles y definiciones. Por este motivo, se recomienda la lectura de otras fuentes que profundicen más en los detalles mencionados en este documento (por ejemplo, *Corporation*, *Intel® Xeon Phi™ Coprocessor - the Architecture*, *The first Intel® Many Integrated Core (Intel® MIC) architecture product*, 2013). También se recomienda buscar en la red nuevos documentos que den más detalles de esta familia o bien que expliquen arquitecturas de las que aún no hay detalles a la hora de escribir esta documentación (por ejemplo, el procesador Knights Landing).

En esta introducción a la familia Xeon Phi, no se han cubierto detalles de su modelo de programación. A pesar de haber introducido su modelo general en el primer apartado, Intel ha extendido modelos de programación ya existentes como por ejemplo OpenMP (*OpenMP*), Cilck (*Corporation*, *Intel® Cilk™ Plus*) o Intel-TBB (*Corporation, Threading Building Blocks*). Por otro lado, ha facilitado nuevas directivas para trabajar con estas nuevas arquitecturas. Estas extensiones (mayoritariamente accesibles vía directivas de tipo pragma) permiten, por un lado, interaccionar con los KNC (por ejemplo, mover datos al MIC, crear o destruir hilos, hacer *map* o *reduce* de variables etc.); y por otro lado, utilizar nuevas funciones facilitadas por esta arquitectura (por ejemplo, algunas ya disponibles en otros sistemas: *software prefetchs*, FMA etc.). Por este motivo, también se recomienda la lectura de documentos que dan más detalles del modelo de programación (*Corporation*, *Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide2013* ; *Corporation, An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors 2013*; Roth, 2013).

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

500, G. T. (n.d.). *Green Top 500*. Retrieved 8 28, 2013, from http://www.green500.org/

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.

anandtech (n.d.). *AMD Radeon HD 7970 Review: 28nm And Graphics Core Next, Together As One*. Retrieved 08 29, 2013, from http://www.anandtech.com/show/5261/amd-radeon-hd-7970-review/10

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/

Corporation, I. (2013). *An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors* . Portland: Intel .

Corporation, I. (2013). *Intel® Xeon Phi™ Coprocessor - the Architecture, The first Intel® Many Integrated Core (Intel® MIC) architecture product*. Portland: Intel.

Corporation, I. (2013). *Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide*. Portland: Intel.

Corporation, I. (n.d). *Haswell Server Products*. Retrieved 08 29, 2013, from http://ark.intel.com/products/codename/42174/Haswell

Corporation, I. (n.d.). *Intel® Cilk™ Plus*. Retrieved 09 1, 2013, from Intel® Cilk™ Plus: http://software.intel.com/en-us/intel-cilk-plus

Corporation, I. (n.d.). *Introduction to x64 Assembly*. Retrieved 09 01, 2013, from Introduction to x64 Assembly: http://software.intel.com/en-us/articles/introduction-to-x64-assembly

Corporation, I. (n.d.). *Sandy Bridge Server Products*. Retrieved 08 29, 2013, from http://ark.intel.com/products/codename/29900

Corporation, I. (n.d.). *Threading Building Blocks*. Retrieved 08 29, 2013, from Threading Building Blocks: www.threadingbuildingblocks.org/

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

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

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.

Jedec (n.d.). *Jedec*. Retrieved 08 29, 2013, from www.jedec.org/sites/default/files/docs/JESD212.pdf

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”.

OpenMP (n.d.). *The OpenMP® API specification for parallel programming*. Retrieved 09 01, 2013, from The OpenMP® API specification for parallel programming.

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

pcwire (2012, 05 12). *Intel Brings Manycore x86 to Market with Knights Corner*. Retrieved 08 29, 2013, from http://www.hpcwire.com/hpcwire/2012-11-12/intel_brings_manycore_x86_to_market_with_knights_corner.html

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.

Ramos Garea, S.; Hoefler, T. (2013). *Modelling Communications in Cache Coherent Systems*.

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.

Roth, F. (2013). *System Administration for the Intel® Xeon Phi™ Coprocessor*. Portland: Intel.

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*.

TACC, T. A. (n.d.). *Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors*. Retrieved 09 01, 2013, from Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors: http://www.tacc.utexas.edu/resources/hpc/stampede

Top500 (n.d.). Retrieved 08 28, 2013, from http://www.top500.org/

Top500 (n.d). *China’s Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500* . Retrieved 09 01, 2013, from China’s Tianhe-2 Supercomputer Takes No. 1 Ranking on 41st TOP500: http://www.top500.org/blog/lists/2013/06/press-release/

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).