# Arquitecturas multihilo

Francesc Guim
Ivan Rodero

PID_00215410

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** ..... 7

**1. Motivación** ..... 9

**2. Preliminares: límites del paralelismo a nivel de instrucción** ..... 10
- 2.1. Máquina perfecta ..... 11
- 2.2. Impacto de la ventana de instrucciones ..... 12
- 2.3. Impacto del predictor de saltos ..... 13
- 2.4. Impacto de los registros de renombre o *renaming* ..... 15
- 2.5. Conclusiones ..... 16

**3. Paralelismo a nivel de hilo de ejecución o *multithreading*** ..... 18

**4. Arquitecturas *super-threading*** ..... 21
- 4.1. La compartición fina ..... 21
- 4.2. La compartición gruesa ..... 23

**5. Arquitecturas con multihilo simultáneo** ..... 24
- 5.1. Conversión del paralelismo a nivel de hilo en paralelismo a nivel de instrucción ..... 25
- 5.2. Diseño de un SMT ..... 26
- 5.3. Complejidades y retos en las arquitecturas SMT ..... 28
- 5.4. Implementaciones comerciales de la SMT ..... 29
  - 5.4.1. El Hyper-Threading de Intel ..... 29
  - 5.4.2. El Alpha 21464 ..... 33
- 5.5. Conclusiones ..... 35

**6. Arquitecturas multinúcleo** ..... 37
- 6.1. Limitaciones del SMT y arquitecturas *super-threading* ..... 37
  - 6.1.1. Escalabilidad y complejidad ..... 37
  - 6.1.2. Consumo energético y área ..... 38
  - 6.1.3. Producción ..... 38
- 6.2. El concepto de multinúcleo ..... 39
- 6.3. Variante de multinúcleo con L3 y directorio ..... 42
- 6.4. Diseño de arquitecturas multinúcleo ..... 44
  - 6.4.1. Las interconexiones ..... 44
  - 6.4.2. Coherencia ..... 49
- 6.5. Conclusiones ..... 56

**Resumen**..... 57

**Actividades**..... 59

**Bibliografía**..... 60

# Introducción

Durante muchos años, el rendimiento de los procesadores se ha ido incrementando y ha explotado el nivel de paralelismo inherente al flujo de instrucciones de una aplicación. En 1971, se lanzó a la venta el primer microprocesador simple, el denominado Intel 4004. Este procesador, de 4 bits, estaba formado por una sola unidad aritmético-lógica: un banco de registros con 16 entradas y un conjunto de 46 instrucciones. En 1974, Intel fabricó un nuevo microprocesador 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, en los que tan solo se podían ejecutar 200.000 instrucciones por segundo, se ha llegado a arquitecturas tipo Sandy Bridge (Intel) o AMDfusion (AMD) en las que se pueden llegar a ejecutar aproximadamente 2.300 billones de instrucciones por segundo. Con el fin de lograr este incremento del rendimiento, a grandes rasgos, se distinguen dos tipos de mejoras importantes aplicadas a los primeros modelos antes mencionados: técnicas asociadas a explotar el paralelismo de las instrucciones y técnicas asociadas a explotar el paralelismo inherente de las aplicaciones multihilo.

El primer conjunto de técnicas consistió en tratar de explotar el nivel de paralelismo de las instrucciones que formaban una aplicación (*instruction level parallelism*, ILP, de ahora en adelante). Ejemplos de procesadores que las usaron son VAX 78032, PowerPC 601, los Intel Pentium o bien los AMD K5 o AMD K6.

Dentro de este primer grupo de mejoras hay, entre otros, arquitecturas superescalares, ejecución fuera de orden de instrucciones y técnicas de predicción o *very long instruction word* (VLIW) (Fisher, 1893). Estas últimas optimizaciones eran bastante interesantes desde el punto de vista del ILP, dado 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.

El segundo conjunto de técnicas intenta explotar el paralelismo asociado a los hilos que componen las aplicaciones. Durante las décadas entre 1990 y 2010, la cantidad de hilos que componen las aplicaciones ha aumentado notoriamente. Se pasó del uso de pocos hilos de ejecución al uso de miles de hilos por aplicación. Un claro ejemplo 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. Ejemplos de estas aplicaciones son máquinas virtuales como Parallels y VMWare o navegadores como Chrome o Firefox.

Para apoyar este tipo de aplicaciones, el hardware ha ido evolucionando de manera natural a causa de esta demanda creciente de paralelismo. Tal como se ve a continuación, esta tendencia se hizo patente con los primeros procesadores con *simultaneous multithreading* y se ha confirmado con la aparición de sistemas multinúcleo.

# Objetivos

Los principales objetivos que debe alcanzar el estudiante con este módulo didáctico son los siguientes:

1. Ver las limitaciones inherentes a las arquitecturas de procesador que explotan solo el paralelismo a nivel de instrucción.

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

3. Conocer qué tipo de arquitecturas multihilo hay y cuáles son las propiedades de cada una.

4. Entender cómo se puede mejorar el rendimiento de los procesadores explotando el paralelismo a nivel de hilo y a nivel de instrucción a la vez.

5. Saber en detalle los diferentes tipos de arquitecturas multihilo y el diseño de algunos procesadores comerciales que se han diseñado.

6. Entender en detalle las arquitecturas multinúcleo y las diferentes características que pueden hacer variar más el diseño y rendimiento.

# 1. Motivación

La tendencia de evolución del rendimiento de los procesadores es duplicarse cada año respecto al año anterior. Este rendimiento tiene lugar gracias a dos factores: una mejora del proceso y una mejora de la arquitectura. Estos factores se denominan tics y tacs, respectivamente. En las líneas de procesadores actuales (por ejemplo, la gama de servidores de Intel conocida como Xeon; Rusu, Tam, Muljono, Ayers y Chang, 2006), estos dos factores se van intercalando (un procesador es un tic y el siguiente es un tac).

El primer factor es un factor tecnológico asociado al proceso de fabricación. Los transistores cada vez son más pequeños y esto permite incrementar el tamaño de los recursos sin aumentar el del procesador (por ejemplo, duplicar el tamaño de la memoria caché o el número de bits que los buses pueden transmitir). Este incremento de capacidad se traduce claramente en una mejora del rendimiento.

El segundo pretende sacar rendimiento de cambios arquitectónicos importantes en el procesador (por ejemplo, pasar de una arquitectura en orden a una arquitectura de orden). En el caso de los procesadores ILP, este segundo factor acabaría siendo un límite si no se entrara en otro tipo de arquitecturas en las que se explotaran también otros factores. La limitación principal es que el rendimiento se intenta explotar en un solo flujo de ejecución. Eso, por sí mismo, ya es una limitación importante porque a menudo los flujos tienen características que por sí mismas ya limitan el rendimiento que se puede sacar (por ejemplo, errores de páginas, errores por kiloinstrucciones, cantidades de saltos o dependencias entre instrucciones).

En cualquier caso, la escalabilidad del rendimiento de estos sistemas es finita, de forma que, para obtener una mejora continua en el rendimiento de los procesadores, es básico optimizar la arquitectura, pero también es necesario mejorar el proceso de fabricación. El estudio de las técnicas de mejora de arquitectura es la motivación principal del módulo que se presenta a continuación.

# 2. Preliminares: límites del paralelismo a nivel de instrucción

Tal como se ha mencionado en el apartado anterior, durante las primeras décadas de desarrollo de los microprocesadores (1970 y 1980), el objetivo principal se centró en explotar al máximo el paralelismo de las aplicaciones a nivel de instrucción. El límite de este tipo de paralelismo es el que dio lugar a una segunda era de microprocesadores, en la que el paralelismo se mueve entre los hilos de ejecución. En este apartado, se presenta un estudio de límites en el que se pretende demostrar que realmente el rendimiento de las técnicas ILP logró su umbral máximo, teniendo en cuenta, está claro, el tipo de aplicaciones y ciencia de computación que se entiende actualmente.

Para llevar a cabo este estudio de límites, se analiza el impacto de correr seis *benchmarks* (Bailey, Barton, Lasinski y H., 1991) muy empleados para estudiar microprocesadores en una máquina ILP infinita, es decir, una máquina de recursos infinitos y perfectos. Sus aplicaciones se presentan en la tabla 1 y se explican con más detalle en Bailey, Barton, Lasinski y H. (1991). Así pues, el objetivo de este estudio es ver el resultado de correr los *benchmarks* mencionados en una máquina ILP perfecta. Las características de la máquina perfecta y los resultados que se desprenden del estudio se presentan en el primer subapartado de este análisis.

Tabla 1. Aplicaciones consideradas para el estudio

| Nombre de la aplicación | Tipo | *Benchmark* |
|---|---|---|
| **gcc** | enter | Spec89/92/95/2000 |
| **espresso** | enter | Spec89/92 |
| **Li** | enter | Spec89/92 |
| **fppp** | coma flotante | Spec89/95 |
| **doducd** | coma flotante | Spec89/95 |
| **tomcatv** | coma flotante | Spec89/95 |

Además, posteriormente se muestra y se estudia el impacto de limitar algunas de las características de este tipo de procesadores que más significación tienen en el rendimiento de las aplicaciones. Los resultados obtenidos, y también las características de la máquina que se van limitando, se muestran en los apartados posteriores.

## 2.1. Máquina perfecta

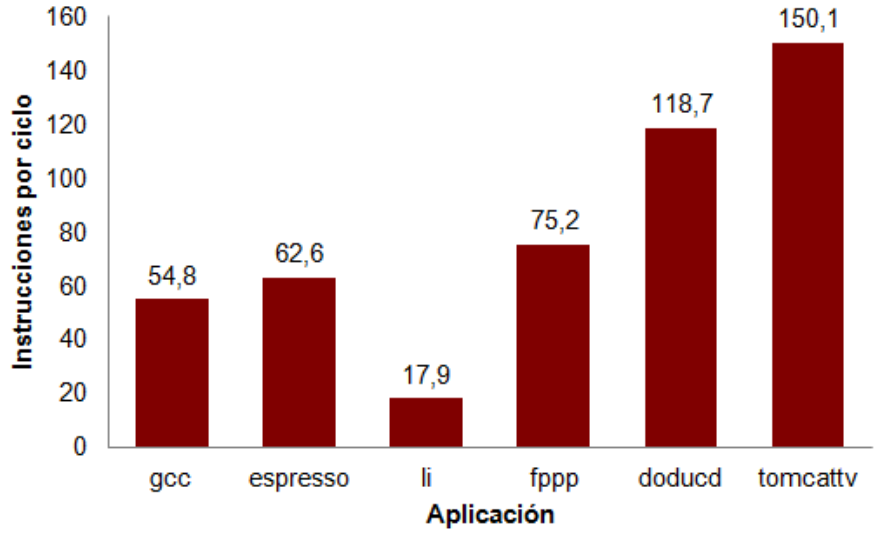
La tabla 2 presenta el modelo de máquina ILP infinita que se estudia. Entre otras características, se muestra la cantidad de instrucciones que se pueden generar por ciclo, el número de instrucciones que puede tener en vuelo o el número de registros de renombre o *renaming* que tienen. Por otro lado, para compararla con las características que presenta un procesador del mercado, al lado se muestran los valores que presenta una máquina Power 5 IBM (Chase y Doyle, 2001) en cada una de las características especificadas.

Tabla 2. Máquina ILP infinita frente a una Power 5

| | Modelo | Power 5 |
|---|---|---|
| **Instrucciones generadas por ciclo** | Infinitas | 4 |
| **Ventana de instrucciones** | Infinita | 200 |
| **Número de registros para hacer *renaming*** | Infinitos | 48 registros de tipo enter<br>40 registros de tipo coma flotante |
| **Algoritmo de predicción** | Perfecto | de un 2% a un 6% de error |
| **Memoria caché** | Perfecta | 64 kB de primer nivel de datos<br>64 kB de primer nivel de instrucciones<br>1,92 MB de segundo nivel<br>36 MB de tercer nivel |

La figura 1 muestra las instrucciones por ciclo que puede lograr cada uno de los *benchmarks* en esta máquina ideal. Tal como se puede observar, tanto el fpppp como el doducd como el tomcatv muestran un nivel de paralelismo de instrucción bastante elevado, dado que se pueden hacer más de setenta instrucciones por ciclo. El gcc, el espresso y li muestran unos niveles de paralelismo bastante inferiores, sobre todo teniendo en cuenta que disponen de un procesador con recursos infinitos.

Figura 1. Rendimiento en una máquina ILP ideal



A continuación, se presentan los estudios llevados a cabo y que limitan alguna o algunas de las características de la máquina perfecta.

### 2.2. Impacto de la ventana de instrucciones

Como parte del estudio de límites, a continuación se evalúa el impacto en los *benchmarks* anteriores al limitar los diferentes recursos que hemos presentado más arriba. En primer lugar, se analiza el efecto de tener una ventana de instrucciones limitada. Para este estudio, se consideran los valores siguientes: instrucciones infinitas, 2 K, 512, 128, 32, 8 y 4 instrucciones.

Hay que destacar los dos aspectos siguientes:

- El primero es que no se limita la cantidad de instrucciones generadas por ciclo: se quiere ver la limitación de esta arquitectura para procesar operaciones asumiendo que genera infinitas.

- El segundo es que la máquina Power 5 tiene una ventana de instrucciones de 200 entradas. Por lo tanto, se evalúan valores inferiores, pero también valores bastante superiores a este.

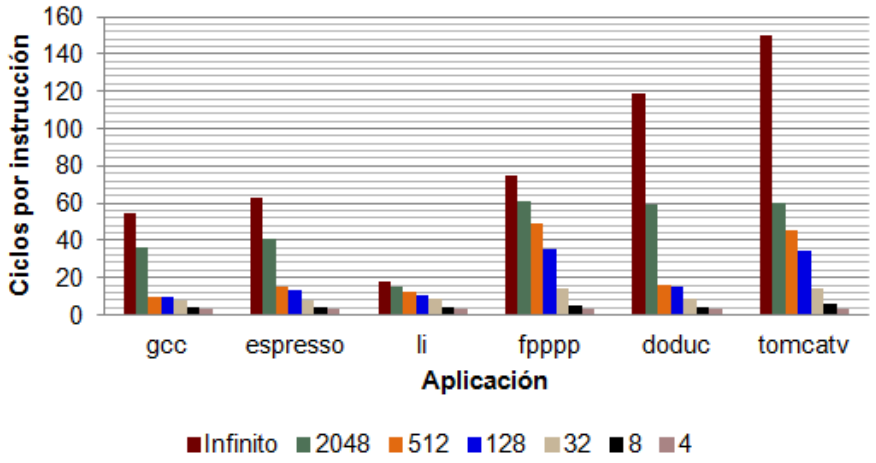
Así pues, las características de la máquina perfecta limitada son las descritas en la tabla 3.

Tabla 3. Máquina ILP perfecta (variante de la ventana de instrucción) frente a una Power 5

| | **Modelo** | **Power 5** |
|---|---|---|
| **Instrucciones generadas por ciclo** | Infinitas | 4 |
| **Ventana de instrucciones** | Infinita, 2 K, 512, 128, 32, 8, 4 | 200 |
| **Número de registros para hacer *renaming*** | Infinitos | 48 registros de tipo enter<br>40 registros de tipo coma flotante |
| **Algoritmo de predicción** | Perfecto | de un 2% a un 6% de error |
| **Memoria caché** | Perfecta | 64 kB de primer nivel de datos<br>64 kB de primer nivel de instrucciones<br>1,92 MB de segundo nivel<br>36 MB de tercer nivel |

La gráfica siguiente (figura 2) muestra los efectos de limitar la ventana de instrucciones en el procesador ILP perfecto antes introducido. La primera de las barras muestra el valor de referencia (una ventana infinita). Las otras seis barras de cada *benchmark* muestran los CPI que se obtienen reduciendo la ventana de instrucciones a 2.048, 512, 128, 32, 8 y 4 instrucciones.

Figura 2. Rendimiento en una máquina ILP ideal variante de la ventana de instrucciones




Como se puede observar, reduciendo la ventana a 2.048 instrucciones, la reducción de rendimiento ya es considerable. Este impacto es muy drástico para aquellas aplicaciones que presentaban un paralelismo de instrucción más elevado. Por ejemplo, tomcatv ha pasado de un CPI de 155 a un IPC 60. Es importante fijarse en que esta reducción aparece reduciendo esta ventana a un valor muy elevado (2.048 instrucciones). Los efectos son mucho más devastadores cuando se consideran los valores más bajos.

Tomando como valor de referencia una ventana de instrucciones de 200 entradas (Power 5), podemos ver que las aplicaciones experimentarían una reducción de rendimiento muy considerable respecto a nuestra máquina ILP perfecta. Por ejemplo, en el caso del gcc, se reduce el rendimiento diez veces.

### 2.3. Impacto del predictor de saltos

El siguiente cambio aplicado a la máquina ILP perfecta es usar un algoritmo de predicción de saltos real. Como es sabido, los costes asociados a cometer un error en la predicción de un salto son bastante elevados. Estos costes se ven acentuados cuanto más largo es el *pipeline* del procesador y cuanto más instrucciones en vuelo se pueden tener. En estos casos, cuando sucede un error de este tipo, implica hacer *rollback* de muchas transacciones en vuelo, aspecto altamente costoso.

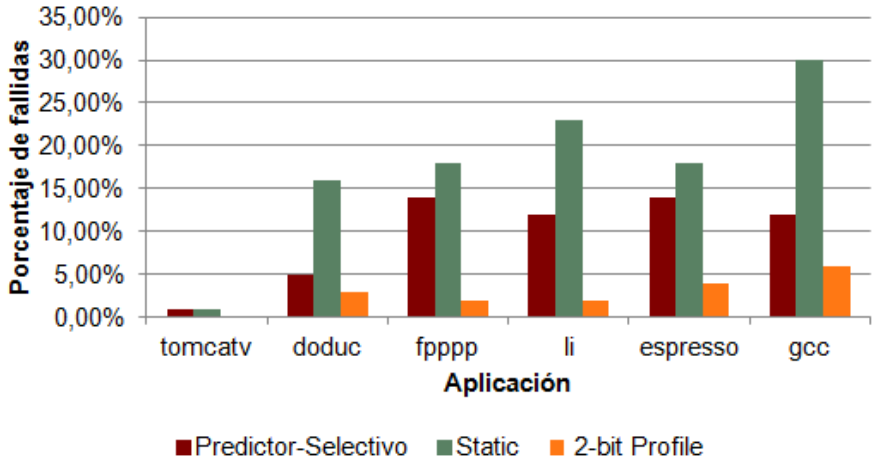
La tabla 4 presenta los seis escenarios considerados para este estudio, usando un predictor perfecto, tres predictores reales (uno selectivo, uno basado en perfil y uno estático) y ningún predictor. Como valores de referencia, podemos ver que el predictor de saltos de la Power 5 tiene un error medio situado entre un 2% y un 6%.

Tabla 4. Máquina ILP semiperfecta (variante del algoritmo de predicción de saltos) frente a una Power 5

| | Modelo | Power 5 |
|---|---|---|
| **Instrucciones generadas por ciclo** | 64 | 4 |
| **Ventana de instrucciones** | 2.048 | 200 |
| **Número de registros para hacer *renaming*** | Infinitos | 48 registros de tipo enter<br>40 registros de tipo coma flotante |
| **Algoritmo de predicción** | Perfecto, predictor selectivo, 2 bits, estático frente a ninguno | de un 2% a un 6% de error |
| **Memoria caché** | Perfecta | 64 kB de primer nivel de datos<br>64 kB de primer nivel de instrucciones<br>1,92 MB de segundo nivel<br>36 MB de tercer nivel |

Como se muestra en la figura 3, vemos el porcentaje de errores de predicción que cada uno de los *benchmarks* usados en este estudio logra para cada uno de los tres predictores empleados. Como podemos observar, el que tiene mejor rendimiento es el que usa dos bits para intentar capturar el perfil del flujo ejecutado.

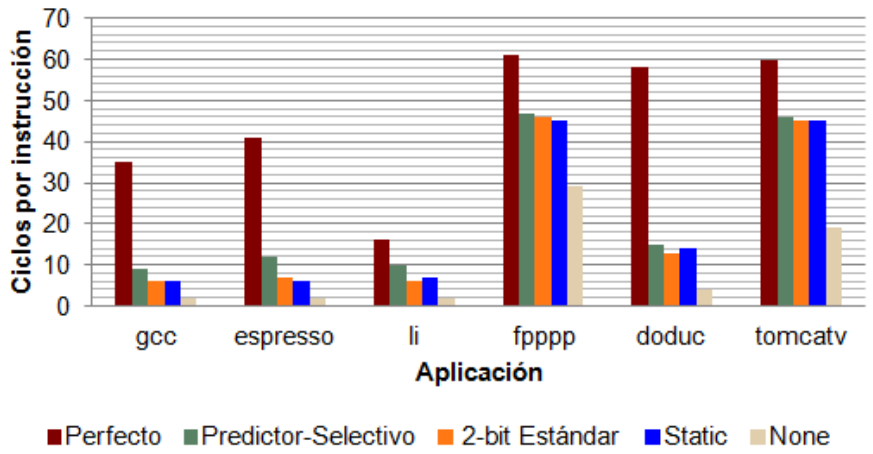
Figura 3. Porcentaje de errores en la predicción por aplicación variante de algoritmo de predicción de saltos




A continuación, en la figura 4, se muestra el impacto en las instrucciones por ciclo (IPC) en cada uno de los *benchmarks* considerados. Como podemos ver, exceptuando el fppp y el tomcatv, el impacto es considerable. Por ejemplo, el gcc pasa de unas IPC de 35 usando un predictor perfecto a unas IPC de 5 usando un predictor basado en perfiles de dos bits. Es importante resaltar que este último predictor mostraba un error de un 5% en la predicción del salto. Estos resultados demuestran que el impacto de los predictores de salto en este tipo de arquitecturas es muy elevado en la mayoría de casos, puesto que reduce hasta seis veces su número de instrucciones por ciclo retiradas.

Hay que destacar que el modelo que se está considerando tiene una jerarquía de memoria perfecta. Si se considerara un modelo de memoria real, los errores de memoria serían mucho más costosos. Los accesos a memoria son extremamente costosos, por lo tanto, en los casos en los que hubiera errores en las memorias caché locales y las operaciones se tuvieran que revertir a causa de un error de predicción, se estaría perdiendo rendimiento de una manera considerable. Por lo tanto, las IPC mostradas a continuación podrían ser sustancialmente inferiores si se considerara un modelo más cercano a la realidad.

Figura 4. Rendimiento en una máquina ILP semiperfecta variante del algoritmo de predicción de saltos




## 2.4. Impacto de los registros de renombre o *renaming*

En los últimos subapartados se ha mostrado el impacto de variar la ventana de instrucciones y el algoritmo de predicción de saltos en un procesador en el que solo hay un flujo de ejecución. Como se ha mencionado, el rendimiento está marcado por las no dependencias entre las instrucciones que se ejecutan. Una de las técnicas más empleadas en estos tipos de arquitectura es el renombre (*renaming*) de registros.

A continuación, en la tabla 5, se muestra una nueva extensión de la máquina ILP anterior variando el número de registros de renombre (tanto de coma flotante como de enter). Tal como se puede observar, en este caso, se ha fijado el algoritmo de predicción de saltos en el perfil de 2 bits. La Power 5 tiene un total de 48 y 40 registros de renombre, de tipo enter y de coma flotante, respectivamente. El modelo que vamos a estudiar en este apartado tiene registros de renombre infinitos, 256, 128, 64 y 32 registros tanto de coma flotante como de enter.

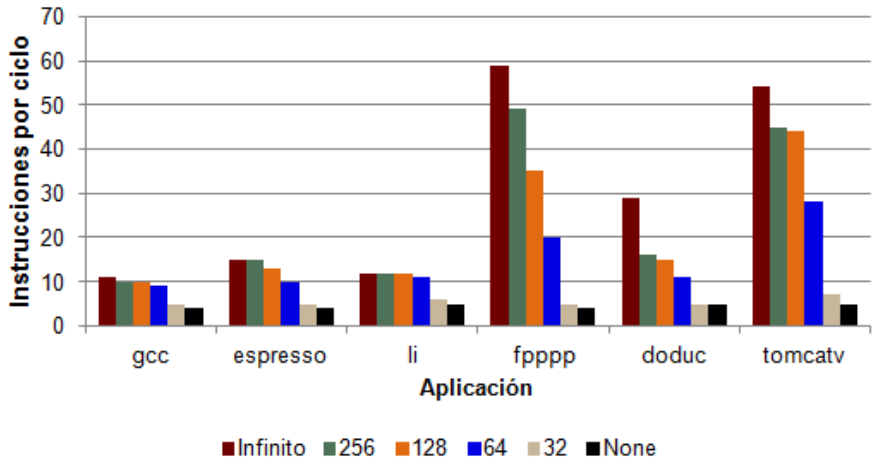
Tabla 5. Máquina ILP semiperfecta (variante del número de registros de renombre) frente a una Power 5

| | Modelo | Power 5 |
|---|---|---|
| **Instrucciones generadas por ciclo** | 64 | 4 |

| | Modelo | Power 5 |
|---|---|---|
| **Ventana de instrucciones** | 2.048 | 200 |
| **Número de registros para hacer *renaming*** | Infinitos, 256, 128, 64, 32, ninguno (tanto enter como coma flotante) | 48 registros de tipo enter<br>40 registros de tipo coma flotante |
| **Algoritmo de predicción** | Perfil 2 bits | de un 2% a un 6% de error |
| **Memoria caché** | Perfecta | 64 kB de primer nivel de datos<br>64 kB de primer nivel de instrucciones<br>1,92 MB de segundo nivel<br>36 MB de tercer nivel |

La figura 5 muestra la variación en las IPC para cada uno de los *benchmarks* considerados y las diferentes cantidades de registros de renombre. Es interesante destacar el impacto de esta variable en las tres aplicaciones que mostraban un índice de paralelismo más alto en la aplicación. Tanto el fppp, como el doduc, como el tomcatv muestran una degradación importante al reducir la cantidad de registros de renombre. Si se toma como referencia el de 64 registros (cercano a la cantidad de registros que tiene la Power 5), se puede observar que las IPC se reducen en un 50% respecto a uno de tamaño infinito. Si se reduce a 32 entradas, las IPC ya se reducen notoriamente. En este caso, el rendimiento cae hasta diez veces respecto a un tamaño infinito.

Figura 5. Rendimiento en una máquina ILP semiperfecta variante de la cantidad de registros de renombre




## 2.5. Conclusiones

Durante este apartado, se ha visto el impacto de pasar de una máquina con una arquitectura perfecta con recursos infinitos a una máquina acotada, pero con recursos considerablemente altos. Hay que destacar que tener más recursos no es gratuito, es decir, tener muchos registros de renombre o una ventana de instrucción muy grande implica que el área que requiere el procesador se

incrementa notablemente y el consumo de energía se dispara. Esto se contrapone a las tendencias actuales, en las que se suele intentar reducir el consumo y sacar rendimiento del paralelismo a nivel de hilo.

Así pues, el modelo resultante de todos los cambios (ventana de instrucciones de 2.048, predictor de perfil de 2 bits, generación de 64 instrucciones por ciclo, memoria caché infinita) ha mostrado que se pasa rápidamente de unas IPC muy elevadas de la máquina infinita a unas IPC mucho más moderadas en una máquina limitada más cercana a la realidad. Hay que destacar que, a pesar de ser una máquina ILP limitada, los valores empleados por estos estudios han sido muy elevados. Por ejemplo, se ha asumido que podía generar 64 instrucciones por ciclo, tenía un banco de registro de 256 elementos (tanto coma flotante como enteros) y tenía una ventana de instrucción de 2.048 instrucciones. De la máquina infinita y perfecta original se han reducido las IPC sustancialmente en todos los *benchmarks* considerados (de cinco a diez veces).

# 3. Paralelismo a nivel de hilo de ejecución o *multithreading*

En el apartado anterior, se han estudiado los límites arquitectónicos de los procesadores que intentaban sacar rendimiento con el paralelismo a nivel de instrucción. Sin embargo, el rendimiento de las aplicaciones no escala proporcionalmente en la cantidad de recursos añadidos. Incluso, en muchos casos, por más recursos que se añadan, el rendimiento de la aplicación se ve incrementado solo ligeramente.

Este factor es el que motivó la aparición de arquitecturas que consideran la ejecución simultánea de diferentes hilos de ejecución en un mismo procesador: paralelismo a nivel de hilo o *thread level parallelism* (TLP) (Lo, 1997). En estas arquitecturas:

- Diferentes hilos de ejecución comparten las unidades funcionales del procesador (por ejemplo, unidades funcionales).

- El procesador debe tener estructuras independientes para cada uno de los hilos que ejecuta, como registro de renombre o contador de programa, entre otros.

- Si los hilos pertenecen a diferentes procesos, el procesador debe facilitar mecanismos para que puedan trabajar con diferentes tablas de páginas.

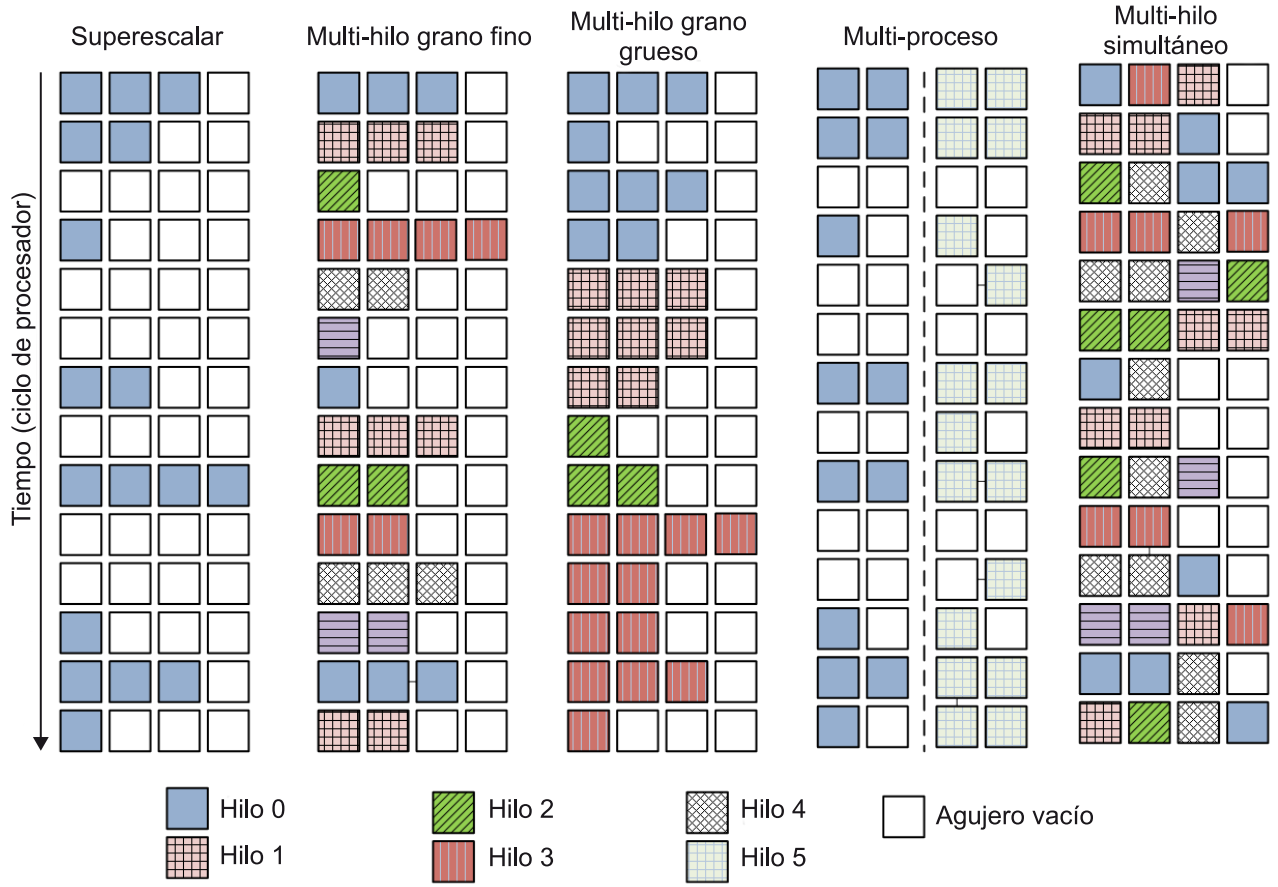
Tal como se muestra durante los apartados siguientes, las arquitecturas actuales explotan este paradigma de muchas formas diferentes. Sin embargo, la motivación es la misma: la mayoría de aplicaciones actuales tienen un alto nivel de paralelismo y su rendimiento escala añadiendo más paralelismo a nivel de procesador. El objetivo es mejorar la productividad de los sistemas que pueden correr aplicaciones que son inherentemente paralelas. Ejemplos de este tipo de aplicaciones son, entre otros, navegadores, bases de datos o servidores web.

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

A continuación, se presentan los diferentes tipos de arquitecturas TLP que se han propuesto durante las últimas décadas. La figura 6 muestra cómo se ejecutarían diferentes hilos de ejecución en cada una de las arquitecturas introducidas a continuación.

**a) Arquitecturas multiprocesador (MP)**. Esta es la extensión más sencilla en 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 tener muchos hilos de ejecución, cada uno de estos hilos sigue teniendo las limitaciones de un ILP. Sin embargo, como se muestra más adelante, existen variantes en las que cada procesador es a la vez multihilo.

Figura 6. Modelo ILP frente a modelos TLP




**b) Arquitecturas multihilo**, también conocidas como ***super-threading***. Esta fue la siguiente de las variantes TLP que apareció. 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 empezará en este ciclo) tiene la posibilidad de escoger cuál de los hilos de ejecución empezará la instrucción siguiente en el ciclo siguiente. Por ejemplo, TERA Systems (Alverson, Callahan, Cummings y Koblenz, 1990) podía trabajar con 128 hilos a la vez.

**c) Arquitecturas con ejecución de hilo simultánea o *simulatenous multithreading* (SMT).** Estas son una variación de las arquitecturas multihilo que permite a la lógica de planificación escoger instrucciones de cualquier hilo en cada ciclo de reloj. Esta condición hace que el uso de los recursos sea mucho más elevado y eficiente. La mayor desventaja 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. Per este motivo, el número de hilos que estas arquitecturas acostumbran a usar es relativamente bajo. Por ejemplo, las arquitecturas Intel con *hyperthreading* (Marr, 2002) implementan dos o más hilos de ejecución, o bien el Alpha 21464 (Seznec, Felix, Krishnan y Sazeide, 2002) implementa cuatro hilos de ejecución.

**d) Arquitecturas *multicore*, denominadas *chip-multiprocessor* (CMP) o *system-on-chip* (SOC).** 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 que no comparten recursos entre ellos, como ahora 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 de *m* núcleos en los que pueden correr hilos que pueden estar relacionados o incluso pueden compartir recursos.

Durante los próximos apartados, se estudian cada una de estas arquitecturas.

# 4. Arquitecturas *super-threading*

El TLP saca rendimiento por el hecho de compartir los recursos entre diferentes hilos de ejecución. Sin embargo, existen dos maneras de llevar a cabo esta compartición. La primera consiste en 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 en 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 se conoce como *super-threading*.

Dentro de las arquitecturas *super-threading*, existen dos maneras de compartir los recursos en el tiempo entre los diferentes hilos. Las más habituales son la compartición fina y la compartición gruesa.

## 4.1. La compartición fina

En la compartición de recursos fina, el procesador cambia de hilo en cada instrucción que este ejecuta. De este modo, 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 $n + 2$ y así sucesivamente. En los casos en los que un hilo se encuentra 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 debe ser capaz de hacer un cambio de hilo por ciclo.

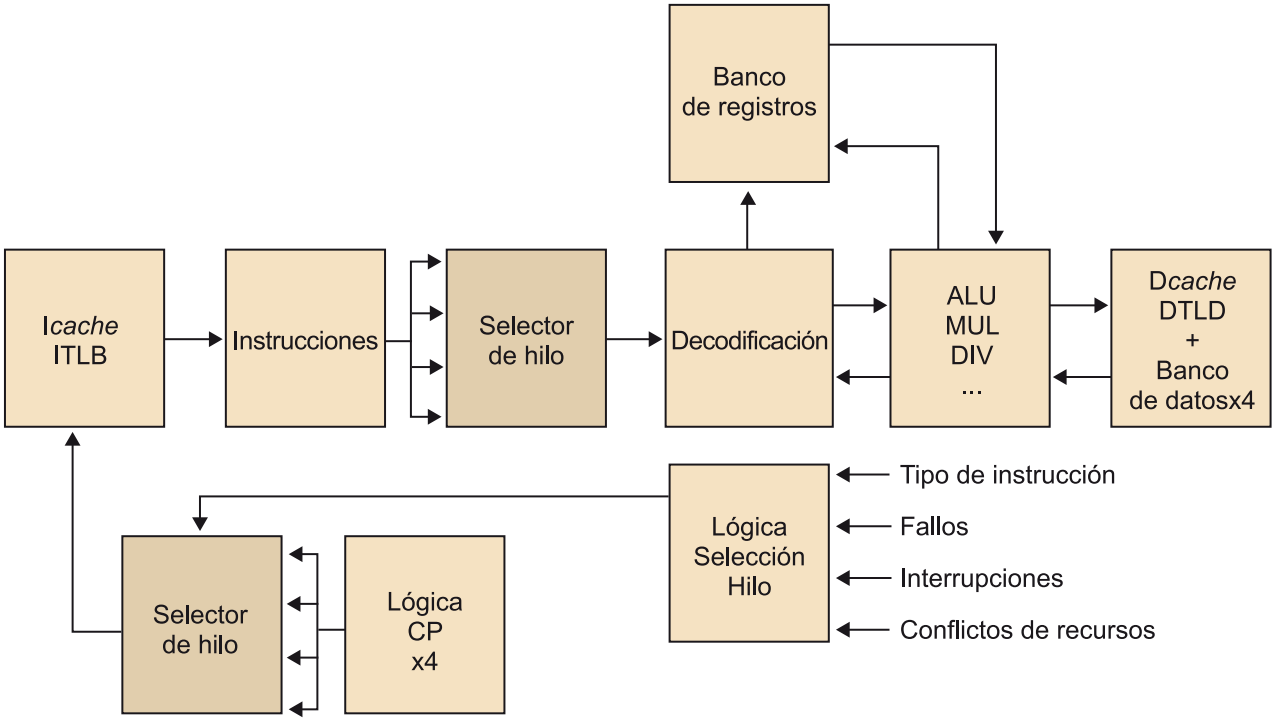
La ventaja de esta opción es que puede amortiguar bloqueos cortos y largos de los hilos que se están ejecutando, dado que en cada ciclo se cambia de hilo. La desventaja principal es que se reduce el rendimiento de los hilos de manera individual, puesto que hilos preparados para ejecutar instrucciones se verán retrasados por las instrucciones de los demás hilos. Esto tiene implicaciones de complejidad de diseño y de consumo de energía, dado que el procesador tiene que poder cambiar de hilo. Un ejemplo de este tipo de procesador es el UltraSPARCT1 (Kongetira, Aingaran y Olukotun, 2005).

El procesador UltraSPARC T1, 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. Antes, la empresa Sun ya había introducido dos arquitecturas multinúcleo: los UltraSPARC IV y IV+. Sin embargo, el T1 fue el primer microprocesador que era multinúcleo y multihilo. Se pueden

encontrar versiones con cuatro, seis u ocho núcleos, cada uno de los cuales puede manejar cuatro hilos de forma concurrente. Esto implica que se pueden correr hasta un máximo de 32 hilos concurrentemente.

La figura 7 muestra las etapas (también llamado *pipeline*) que cada uno de estos núcleos tiene. 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 sucede un acontecimiento de larga latencia (como, por ejemplo, un error de memoria caché que desencadena una petición en memoria), el hilo que lo ha causado se saca de la cadena de rotación (*round robin*). Una vez este acontecimiento de larga duración acaba, el hilo se vuelve a añadir a la rotación para acceder a las unidades funcionales. Por el hecho de que el *pipeline* se comparta con diferentes hilos 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 error de las memorias caché es mucho más reducido: en el caso de que uno de los hilos cause un error, los demás pueden seguir usando los recursos y progresando.

Figura 7. *Pipeline* UltraSPARC T1




Tal como se puede observar en la figura anterior, considerando un ciclo determinado, todos los elementos del *pipeline* son usados por solo un hilo. No obstante, algunas de las estructuras se encuentran replicadas por el número de hilos que el núcleo tiene, como, por ejemplo, la lógica de gestión del contador de programa. En este caso, el núcleo debe 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. Al contrario, el resto de lógica, como, por ejemplo, la lógica de descodificación, es única y solo usada por un solo hilo en un ciclo determinado.

### 4.2. La compartición gruesa

En la compartición gruesa, los cambios de hilos se hacen cuando en el hilo que está corriendo sucede un bloqueo de larga duración. Un ejemplo de este tipo de bloqueo es un error en la memoria caché de último nivel. En este caso, haría falta ir a la memoria, hecho que implicaría muchos ciclos de bloqueo.

La ventaja de esta opción es que no se reduce el rendimiento del hilo individual, ya que solo cambia de hilo cuando este queda bloqueado por un acontecimiento que requiere una larga latencia 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 quedan bloqueadas o congeladas. Estas se volverán a emplear un vez el hilo pueda volver a procesar instrucciones.

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

Un ejemplo de este tipo de procesador es el IBM AS/400 (IBM, 2011).

# 5. Arquitecturas con multihilo simultáneo

Las arquitecturas con multihilo simultáneo, *simultaneous multithreading* (SMT) (Tullsen, Eggers y Levy, 1995), son una variación de las arquitecturas tradicionales multihilo. Estas nuevas arquitecturas 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 sucesivas del *pipeline*) como vertical (por ejemplo, recursos de una misma etapa del procesador).

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 bastante compleja. Hay que resaltar que, por defecto, los hilos de diferentes procesos no se tienen que poder ver entre ellos, tanto por cuestiones 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 demás procesos, siempre que no se usen técnicas explícitas para compartir recursos.

Las arquitecturas que son totalmente SMT y que son capaces de trabajar con muchos hilos son extremamente costosas en términos de complejidad. Es importante tener en cuenta que las estructuras necesarias para soportar este tipo de compartición crecen proporcionalmente al 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, es necesario lograr un equilibrio de rendimiento ante consumo energético, complejidad y área. Por ejemplo, el Hyper-Threading de Intel implementa solo contextos y dos hilos. Otro ejemplo es el Alpha 21464, que dispone de hasta cuatro hilos paralelos.

En el resto del apartado, se trata el diseño de este tipo de arquitecturas, así como también algunas de las arquitecturas SMT más relevantes de la bibliografía.

### 5.1. Conversión del paralelismo a nivel de hilo en paralelismo a nivel de instrucción

Tal como se ha comentado, las arquitecturas SMT permiten la compartición de diferentes unidades funcionales por diferentes hilos de ejecución. Para permitir este tipo de compartición, el estado de cada uno de los hilos se tiene que guardar de manera independiente. 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 hilo. En el caso de no tener estas estructuras duplicadas, la ejecución funcional de cada uno de los hilos interferiría con la de otro hilo. Por otro lado, podría haber problemas de seguridad graves. Per ejemplo, compartir la tabla de páginas implicaría que un hilo compartiría el mapeo de direcciones virtuales a físicas. Esto implicaría que un hilo podría acceder a la memoria del otro hilo sin ningún tipo de restricción. Sin embargo, existen otros recursos que no hay que replicar, como, por ejemplo, el acceso a las unidades funcionales para acceder a memoria (dado que los mecanismos de memoria virtual ya la apoyan por multiprogramación).

La cantidad de instrucciones que un procesador SMT puede generar por ciclo se encuentra limitada por los desbalanceos en los recursos necesarios para ejecutar los hilos y la disponibilidad de esos recursos. Sin embargo, también hay otros factores que limitan esta cantidad. Por ejemplo, el número de hilos activos, posibles limitaciones en el tamaño de las colas disponibles, la capacidad de generar suficientes instrucciones de los hilos disponibles o limitaciones sobre qué tipo de instrucciones se pueden generar desde cada hilo y cuáles pueden generar todos los hilos.

Las técnicas SMT asumen que el procesador facilita un conjunto de mecanismos que permiten la explotación del paralelismo a nivel de hilo. En particular, estas arquitecturas tienen un conjunto grande de registros virtuales que se pueden usar para guardar los registros de cada uno de los hilos de manera independiente (asumiendo, por supuesto, diferentes tablas de renombre para cada hilo). El renombre de registros facilita identificadores únicos de registro; sin este tipo de renombre, dos hilos podrían tener interferencias entre sus ejecuciones.

Por ejemplo, el flujo de ejecución de los dos hilos que se presentan a continuación no funcionaría correctamente (tabla 6). Si los registros r2, r3 y r4 no fueran renombrados para cada uno de los hilos, la ejecución funcional de los hilos sería errónea. En los instantes 1 y 4, los valores que ambos 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, también sucedería en el ciclo 3, en el que el hilo 1 estaría sumando un valor r3 modificado por otro hilo.

Tabla 6. Ejemplo de flujo de instrucciones multihilo

| | | | |
|---|---|---|---|
| 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 renombre 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 resaltar que este tipo de técnica es la que se emplearía en los procesadores fuera de orden, en los que se tiene una tabla de renombre por hilo de ejecución. Sin embargo, en este caso, también hay una lógica diferente con el fin de guardar los diferentes contadores de programa (uno por hilo), existe la capacidad de finalizar más de una instrucción por ciclo o se tienen predictores de saltos por hilo, por ejemplo.

Así pues, el proceso de renombre de registros es exactamente el mismo proceso que hace un procesador fuera de orden. Por ese motivo, un SMT se puede considerar una extensión de este tipo de procesadores (añadiendo, por supuesto, 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 fin de una instrucción (*commit*, en inglés) no es tan sencillo como en un procesador no SMT (donde en la finalización solo se 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 de forma independiente a los demás. Esto se puede llevar a cabo con las estructuras que permiten este proceso para cada uno de los hilos (por ejemplo, teniendo un *reorder buffer* por hilo).

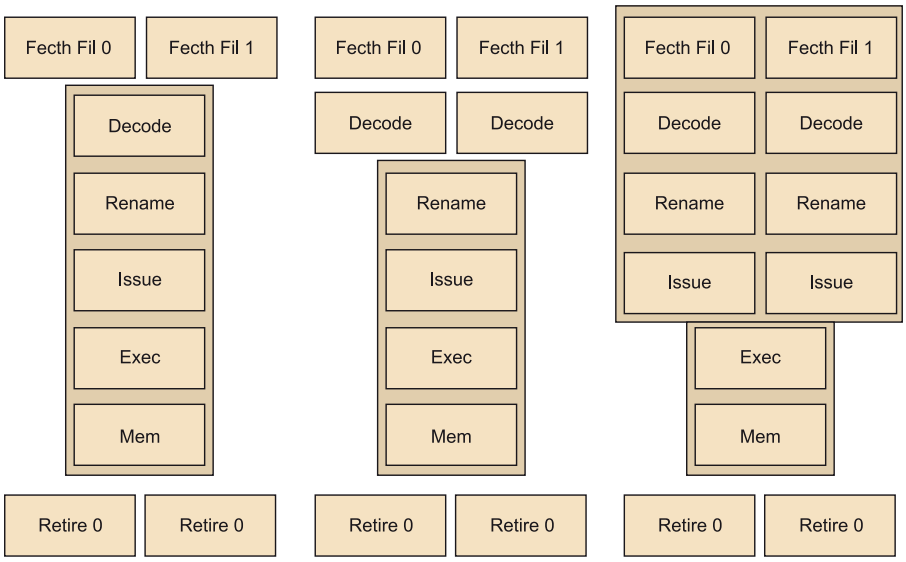
### 5.2. Diseño de un SMT

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 (como etapa de búsqueda de instrucción, etapa de descodificación y lectura de registros) como las técnicas o los algoritmos empleados (por ejemplo, Tomasulo).

Sin embargo, tal como se ha indicado en el apartado anterior, muchas de las estructuras tienen que ser replicadas para apoyar 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 (por ejemplo, tener contadores de programas separados o tablas de páginas separadas). Aun así, se pueden encontrar variantes arquitectónicas que no son estrictamente necesarias, pero que pueden dar más rendimiento en ciertas situaciones.

La figura 8 muestra algunas de las opciones que se pueden tener en cuenta cuando se considera la arquitectura global de un procesador SMT. En esta figura, se muestran diferentes posibilidades de cómo los diferentes hilos comparten o no las diferentes etapas del procesador (se han asumido las etapas típicas de un procesador superescalar fuera de orden). La primera de todas asume que tan solo la búsqueda de instrucción se encuentra dividida por hilo, el resto de etapas son compartidas. Cuanto más a la derecha nos movamos en la figura, las etapas se encuentran más divididas por hilo. Hay que destacar que el hecho de que una etapa se encuentre separada por hilos ejemplifica que el procesador hace la etapa dividida por hilos y que cada hilo tiene estructuras separadas para llevarla a cabo. No obstante, es lógico pensar que todos los hilos tendrán una lógica compartida dado que el mecanismo es común entre todos ellos.

Figura 8. Posible diseño de una arquitectura SMT




En todos los casos, las etapas de ejecución y acceso a memoria son compartidas por todos los hilos. En un estudio académico, se podría considerar que se encuentran replicadas por hilo, 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 los que todos los hilos los compartan. Así pues, cuando los unos estén bloqueados, los otros lo usarán y viceversa, o bien cuando los unos estén haciendo accesos a memoria, los otros podrán usar las unidades aritmético-lógicas. Por lo tanto, para maximizar la eficiencia energética del procesador, es necesario que aquellas sean compartidas.

En estos ejemplos, la etapa de búsqueda de instrucción se separa por hilo. Aun así, esto no es común en todos los diseños posibles. Como se va a ver más adelante, el Hyper-Threading 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, así como el predictor de saltos. También hay que

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

Las etapas de descodificación y renombre identifican las fuentes y destinos de las operaciones, del mismo modo que calculan las dependencias entre las instrucciones en vuelo. En este caso, como las instrucciones de los diferentes hilos serán independientes, podrían tener la lógica correspondiente de manera separada. Sin embargo, tener las estructuras de renombre 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 renombre para ambos hilos y las estructuras están separadas, cada hilo podrá tener acceso a 25 registros como máximo. En aquellos casos en los que uno de los hilos solo pueda emplear 10, pero el otro necesite 40, el sistema estará infrautilizado, de forma que el rendimiento del segundo hilo se verá reducido de forma sustancial.

### 5.3. Complejidades y retos en las arquitecturas SMT

Las arquitecturas SMT incrementan notoriamente el rendimiento del sistema al aumentar la ventana de instrucciones que pueden gestionar. De este modo, en un mismo ciclo, el procesador puede escoger un abanico amplio de instrucciones de los diferentes hilos disponibles en el sistema. Aun así, el resto de etapas se encuentran también más utilizadas por el mismo motivo. Sin embargo, hay que tener en cuenta que estas mejoras se vuelven en contra del rendimiento individual del hilo. En este caso, el rendimiento que un solo hilo puede conseguir puede ser inferior al 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 (*preferred thread*). La unidad encargada de generar o de empezar instrucciones dará preferencia a los hilos preferidos. 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.

Ahora bien, esto no es del todo cierto, ya que dando preferencia 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 un número suficiente de hilos independientes que permitan amortiguar los bloqueos que cada uno experimenta durante su ejecución.

También hay que comentar que existen algunas arquitecturas en las que solo se consideran los hilos preferidos, siempre que estos no se bloqueen. En el momento en el que uno de estos no puede seguir adelante, el procesador considera los demás hilos. En este caso, 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 en el momento de planificar la ejecución de los diferentes hilos que corren sobre el sistema operativo.

Además del reto que las arquitecturas SMT muestran hacia la mejora del rendimiento individual de los hilos, existe una variedad de otros retos que hay que afrontar en el diseño de un procesador de estas características. Por ejemplo:

- 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 cuanto más hilos haya la bolsa de instrucciones que se pueden elegir es más grande. Lo mismo sucede en la etapa de fin de instrucción, donde el procesador debe escoger cuál de las instrucciones acabadas finalizará en el próximo ciclo.

- Tener diferentes hilos de ejecución implica que hay que tener un banco de registros bastante grande 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 haber circunstancias en las que los mismos hilos estén haciendo lo que se denomina compartición falsa. Esta sucede 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 en los que haya mucha compartición falsa, el rendimiento del sistema se degradará de manera sustancial.

Durante los apartados siguientes, se van a presentar dos tecnologías comerciales que incorporaron el concepto de multihilo compartido en sus diseños, el Hyper-Threading de Intel y el procesador 21464 de Alpha.

Otros ejemplos de procesadores fueron MemoryLogix MLX1 (Song, 2002), ClearwaterNetwroks CNP810SP (Melvin, 2000) y FlowStromPorthos (Melvin, Flowstorm porthos massive multithreading (mmt) packet processor, 2003).

## 5.4. Implementaciones comerciales de la SMT

### 5.4.1. El Hyper-Threading de Intel

Hyper-Threading fue el nombre comercial con el que la empresa Intel introdujo en el mercado las tecnologías SMT en sus productos. La primera familia de productos en incluir *hyperthreading* fue la gama de procesadores Xeon orientada a servidores, lanzada en el 2002. En noviembre de ese año, Intel lanzó el procesador Intel Pentium 4. Este fue el primero en incluir SMT en su diseño.

En el 2010, Intel lanzó el procesador Atom. Este también incluye la tecnología SMT. Sin embargo, este producto está orientado al mercado de los dispositivos de bajo consumo. Esto incluye portátiles de bajo consumo, tabletas y teléfonos móviles, entre otros. Por este motivo, a pesar de ser una tecnología SMT, no es fuera de orden. Por lo tanto, no incluye técnicas de reordenamiento de instrucciones, ejecución especulativa o renombre de registros.

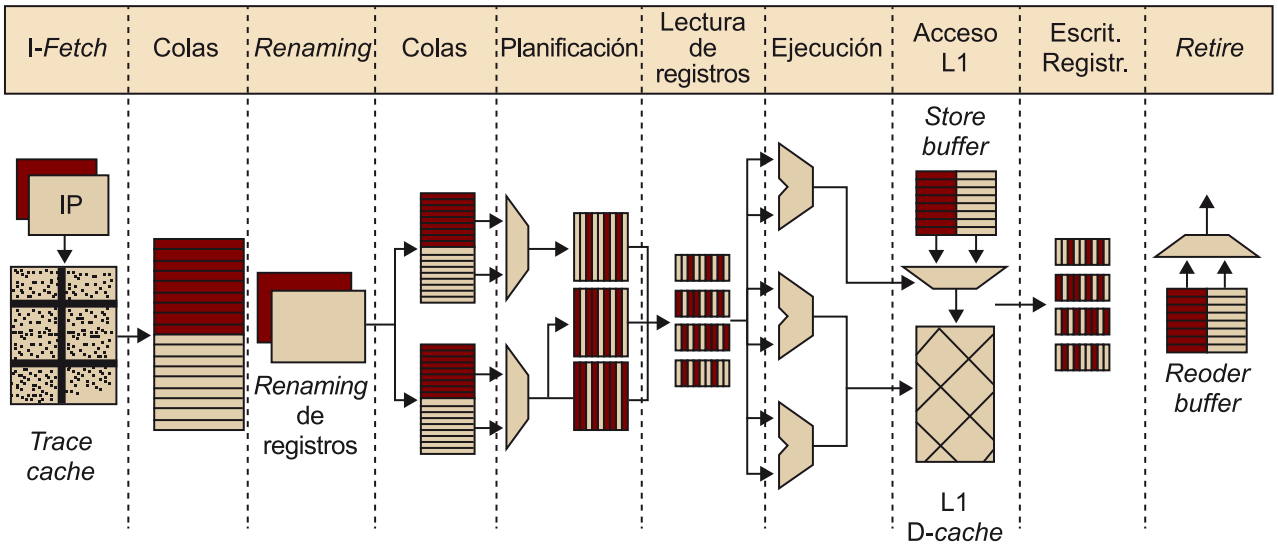
Dentro de la gama de los productos Xeon (Intel), y de forma más reciente, Intel reintrodujo el concepto de *hyperthreading* en la gama de procesadores Nehalem. En este caso, como el segmento de mercado destinado son los servidores, los procesadores ofrecen un nivel de paralelismo mucho más elevado y todas las funcionalidades de un procesador fuera de orden. En este caso, nos movemos del procesador Bloomfield, con cuatro núcleos y un total de ocho hilos de ejecución, hasta el procesador Beckton, con ocho núcleos y un total de dieciséis hilos de ejecución. En estos casos, hay que hacer notar que la tecnología SMT se incluye dentro de cada núcleo. Más adelante, se presentarán las arquitecturas multinúcleo.

Tal como se puede observar, la tecnología SMT es aplicable a muchos tipos de segmentos, como el segmento de los dispositivos móviles o segmentos de computación de altas prestaciones. El concepto de base en todos los casos es el mismo, potenciar el rendimiento de los sistemas aumentando el nivel de paralelismo de las aplicaciones. En unos casos, se necesita un rendimiento muy elevado (por ejemplo, servidor de bases de datos) y, en otros, basta con poder tener diferentes hilos que se ejecuten en paralelo (por ejemplo, el navegador y el gestor de correo). El consumo y la complejidad en ambos casos también son bastante diferentes. Por ejemplo, un procesador Atom puede consumir unos 10 vatios, mientras que un procesador Beckton puede consumir hasta 130 vatios.

### La arquitectura

La figura 9 presenta las diferentes etapas de la microarquitectura del Pentium 4, conocida también como *Netburst microarchitecture* (Intel). El *pipeline* consta de diez etapas diferentes. Las cuatro primeras van en orden y son las relacionadas con la búsqueda y preparación de las instrucciones (búsqueda, renombre y traducción a microoperaciones). Las cinco siguientes pueden ser fuera de orden y son las encargadas de llevar a cabo la ejecución funcional de las operaciones. Finalmente, la última se ejecuta en orden y es donde se finalizan las instrucciones.

Figura 9. *Pipeline* del Pentium 4




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

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

- El puntero en la instrucción siguiente.

- El *buffer* de instrucciones del hilo, es decir, el flujo de instrucciones que el hilo ejecutará potencialmente.

- El *translation look-aside buffer* (TLB) usado para hacer la traducción de direcciones virtuales a físicas. Cada hilo tendrá un mapeo de dirección virtual a una dirección física diferente. De lo contrario, habría problemas de seguridad potenciales.

- El *return stack predictor* o predictor de dirección de retorno. Como su nombre indica, predice las direcciones de retorno de las funciones.

- El *advanced programmble interrupt controller* (APIC) orientado a la gestión de interrupciones.

- La tabla de renombre, necesaria para poder llevar a cabo el fuera de orden y poder hacer el remapeo de registros virtuales en registros físicos.

A pesar de que hay otros recursos que se encuentran replicados, los más importantes son los mencionados anteriormente. También hay que destacar que esta descripción equivale a una arquitectura *hyperthreading* general. Cada implementación concreta, por ejemplo la del procesador Atom, puede tener variaciones en función del tipo de requerimientos que el procesador tiene y del diseño que los arquitectos han llevado a cabo.

En segundo lugar, encontramos ciertos recursos que se encuentran divididos entre los diferentes hilos. En general, la mayoría son *buffers* como, por ejemplo, el *reorder buffer*, los *load* y *store buffers* o las colas entre las diferentes etapas.

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

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

- Las memorias caché.

- Los mecanismos de ejecución fuera de orden.

- Los predictores de salto.

- La lógica de control y los buses de comunicación.

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

**La perspectiva del sistema**

El *hyperthreading*, tal como se ha introducido antes, hacía que un solo procesador se viera desde el punto de vista del sistema como diferentes procesadores lógicos (o hilos de ejecución). El procesador tiene una copia del estado arquitectónico para cada uno de los procesadores lógicos y todos los procesadores lógicos comparten un conjunto de recursos físicos que permiten la ejecución de los diferentes flujos de instrucciones.

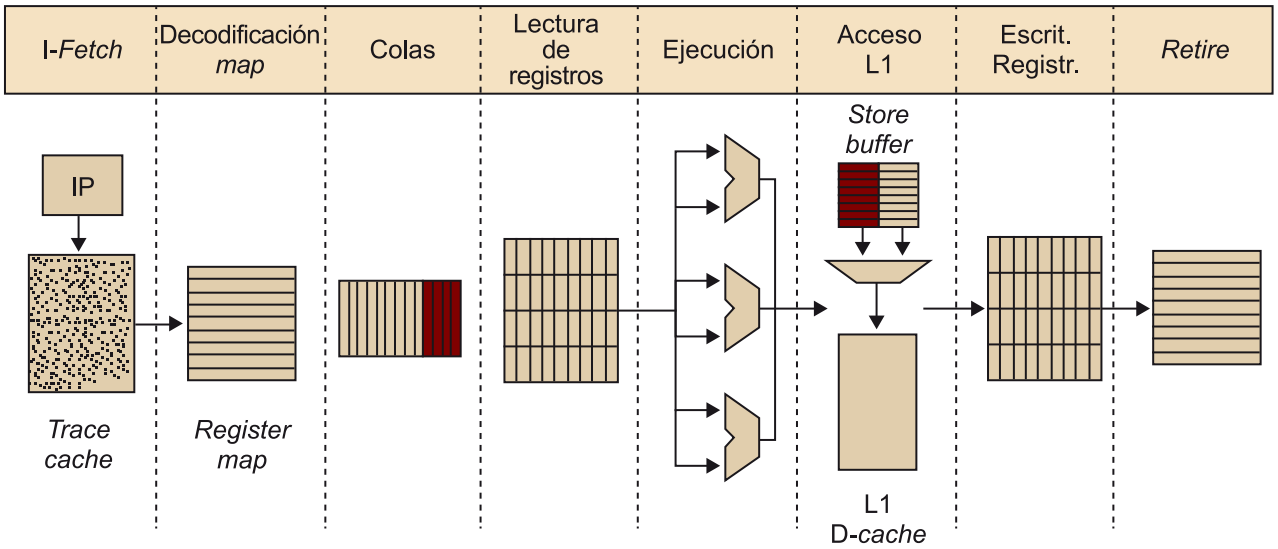
Desde la perspectiva del sistema operativo y de las aplicaciones, el hardware les ofrece *N* procesadores independientes. El software puede gestionar los diferentes hilos de ejecución encima de los procesadores lógicos en función de sus políticas de administración. Lo que es importante recalcar es que,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

Desde la perspectiva del modelo de programación, el *hyperthreading* se puede ver como una arquitectura multiprocesador con tiempo de acceso uniforme a memoria (conocido como NUMA = *non uniform memory access*). Todos los hilos tendrán, de media, un tiempo de acceso a memoria similar.

### 5.4.2. El Alpha 21464

El Alpha 21464 (Rusu, Tam, Muljono, Ayers y Chang, 2006), también conocido como EV8, fue la evolución del Alpha 21364 (figura 10). Respecto a este último, la mejora más importante del EV8 fue la incorporación de técnicas SMT. A pesar de que su rendimiento estimado era bastante más elevado que el de su predecesor, la línea de procesador Alpha se canceló en el 2001. Esto incluyó la cancelación del EV8.

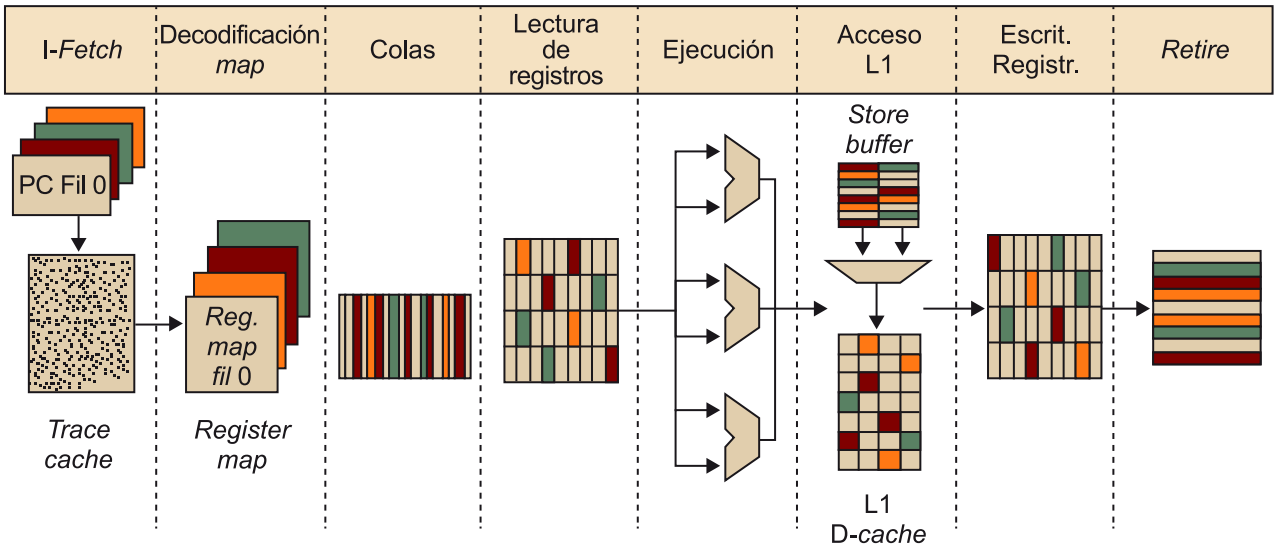
Figura 10. *Pipeline* del Alpha 21364




**La arquitectura**

La figura 11 muestra las diferentes etapas de las que se componía. Para añadir SMT a la arquitectura Alpha, el EV8 incluye una nueva lógica para mantener cuatro contadores de programas independientes (uno por hilo). En cada ciclo, la etapa de *fetch* selecciona uno solo de los hilos disponibles de la memoria caché de instrucciones. Una vez la instrucción es seleccionada y cargada, esta se marca con el identificador del hilo al que pertenece. Este identificador irá emparejado con la instrucción durante todas las etapas diferentes del procesador (como la etapa de descodificación o las unidades funcionales).

Figura 11. *Pipeline* del Alpha 21464




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

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

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

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

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

### La perspectiva del sistema

En el caso del EV8, el sistema también tiene acceso a un solo procesador físico. No obstante, en este caso, también hay un solo procesador lógico. Cada hilo se ve como una unidad de proceso o *thread processing unit* (TPU), que comparte recursos como por ejemplo la memoria caché de instrucciones, la memoria caché de datos, el TLB o la memoria caché de segundo nivel con los demás hilos.

## 5.5. Conclusiones

Durante este apartado, se han estudiado diferentes técnicas para explotar el paralelismo a nivel de hilo. En primer lugar, se han presentado las arquitecturas *super-threading*, en las que las diferentes etapas del procesador son compartidas por los diferentes hilos disponibles en el procesador. Dentro de este tipo de compartición, se entra en detalle en compartición de grano fino y de grano grueso. Del último tipo se estudia la arquitectura del UltraSPARCT1.

En segundo lugar, se han estudiado las arquitecturas de tipo *simultaneous multithreading*, en las que los diferentes hilos del procesador comparten las diferentes etapas. Dentro de estas técnicas, se han presentado diferentes posibilidades de diseño y dos implementaciones reales.

Por un lado, la implementación SMT de Intel, el Hyper-Threading. El procesador incluye diferentes procesadores lógicos (uno por hilo). Cada uno contiene una copia independiente de su estado arquitectónico (incluyendo TLB, registros de renombre y contador de programa, entre otros) y comparten los recursos de las diferentes etapas. En algunos casos, estos recursos se encuentran divididos por hilo (por ejemplo, colas o el *store buffer*). En otros casos, los recursos se encuentran totalmente compartidos por los hilos (por ejemplo, registros de renombre).

Por otro lado, se ha presentado la implementación SMT de Alpha, el EV8. En este caso, el procesador incluye diferentes unidades de procesamiento (también una por hilo). Cada una contiene una copia separada de sus registros y de los contadores de programa. Ahora bien, a diferencia del *hyperthreading*, los hilos del EV8 comparten todo el resto de recursos del procesador (por ejemplo, acceso a la L1, TLB o *store buffer*). Esto implica que los hilos no son independientes y desde el punto de vista del sistema ven el espacio de direcciones.

Durante el próximo apartado, se presentan los límites del paralelismo que los SMT y los *hyperthreading* muestran cuando se quieren construir arquitecturas con un número más elevado de hilos. Como solución a estos problemas de escalabilidad, se van a estudiar las arquitecturas multinúcleo. Estas arquitec-

turas permiten escalar el número de hilos de manera lineal sin topar con incrementos tan costosos como en el caso de las arquitecturas estudiadas en este apartado.

# 6. Arquitecturas multinúcleo

## 6.1. Limitaciones del SMT y arquitecturas *super-threading*

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 logra incrementando la lógica del procesador, analizando el diferente número de hilos que se quiere considerar.

Por ejemplo, si se quieren tener ocho hilos, se replicarán ocho veces las estructuras necesarias (más o menos veces en función del 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 tratan las más representativas.

### 6.1.1. Escalabilidad y complejidad

Los modelos SMT son adecuados para un número relativamente pequeño de hilos (por ejemplo, dos, cuatro u ocho hilos). No obstante, para un número más grande de hilos esto puede comportar ciertos problemas de escalabilidad. Como se ha podido ver en los apartados anteriores, por cada uno de los hilos de los que dispone un procesador hay mucha lógica que se encuentra replicada o compartida.

Este hecho podría implicar tan solo un problema de espacio. Es decir, duplicar el número de hilos implica duplicar el número de entradas del *store buffer* o duplicar el número de tablas de renombre. Aun así, el incremento en la cantidad y tamaño del número de recursos también tiene 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 realizar el *commit* dos instrucciones por ciclo. De lo contrario, el sistema no es sostenible. Poder finalizar dos instrucciones por ciclo es algo 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 su 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 se necesitarían 4.096). Cuando menos, considerando el estado actual de la tecnología de procesadores.

### 6.1.2. Consumo energético y área

Para apoyar un número elevado de hilos, hay que incrementar las estructuras en proporción. 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 del número de puertos de lectura o escritura de las diferentes memorias caché es altamente costoso (tanto en cuanto a la complejidad como en cuanto al área). Añadir un puerto de lectura nuevo a una memoria caché puede equivaler a un incremento de un 50% de su área (Handy, 1998).

Como se ha mencionado antes, si se quiere incrementar el rendimiento de manera más o menos proporcional al número de hilos, se necesita también tenerlo en cuenta en la manera como se accede a la memoria caché. Por lo tanto, si se quisiera aumentar el número de hilos, por ejemplo a 256 hilos, haría falta redimensionar (tanto en área como en lógica) toda la jerarquía de memoria de forma coherente. Tal como se puede ver, y con la tecnología actual, esto es impracticable.

El consumo energético de un procesador SMT que apoye un número muy elevado de hilos es alto. En aquellas situaciones en las que no se usaran todos los hilos disponibles o el uso de los recursos correspondientes fuera ineficiente, el consumo del procesador en vatios 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 el *dynamic voltage scaling* (Yao, Demers y Shenker, 1995), es decir, reducir la frecuencia y el voltaje de algunas partes del procesador, puesto que esto permite reducir sustancialmente el consumo del procesador en situaciones como la planteada.

### 6.1.3. Producción

Durante las últimas décadas y dada una gama de procesadores que siguen un diseño arquitectónico similar (por ejemplo, los Sandy Bridge de Intel), se sacan diferentes versiones de un mismo procesador. Dada 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.

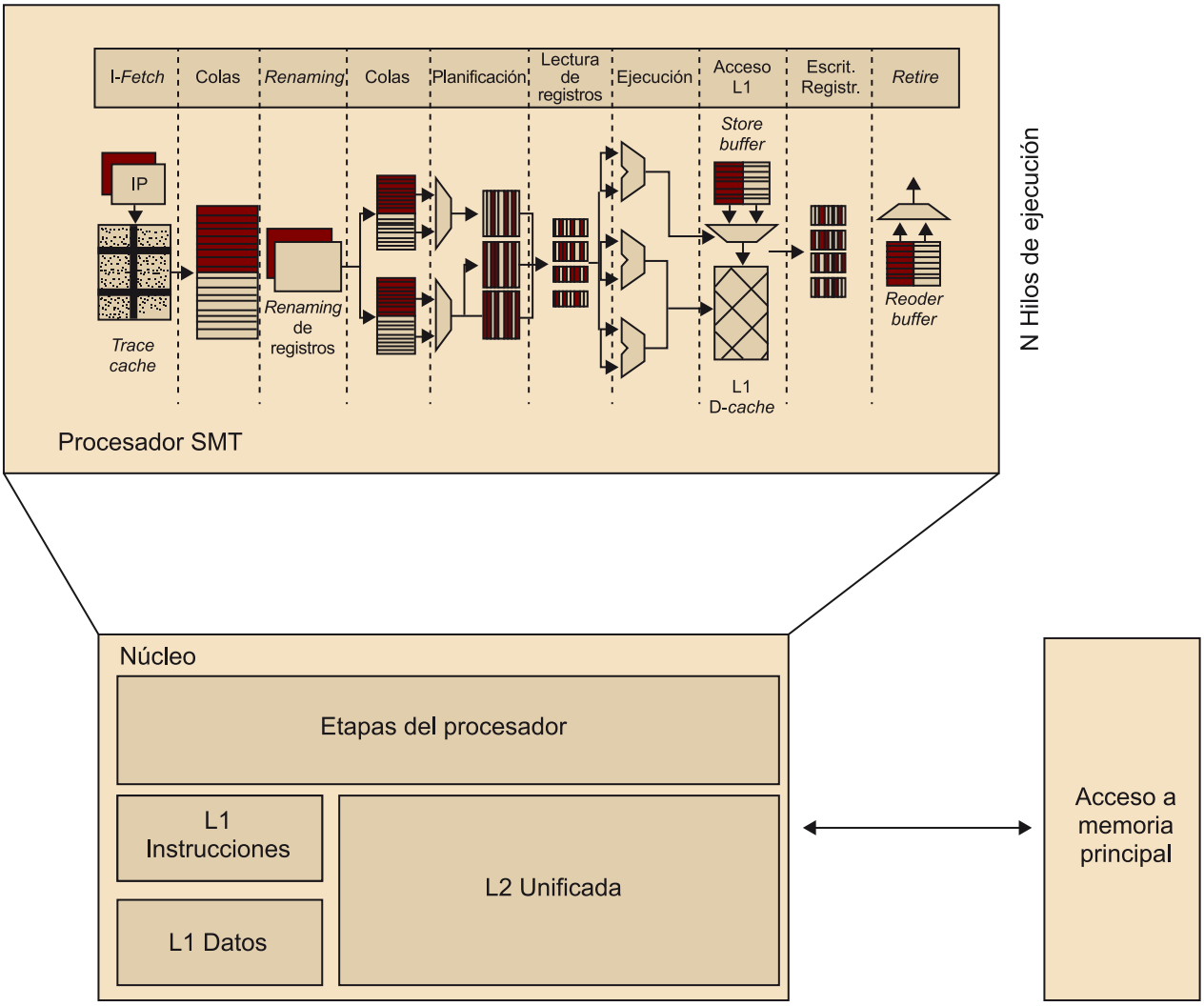
Por ejemplo, en el caso de la familia Sandy Bridge se pueden encontrar los procesadores domésticos con doce hilos que consumen 130 vatios, procesadores para dispositivos móviles de ocho hilos que consumen 55 vatios y procesadores para servidores con dieciséis hilos y 150 vatios de consumo. Cabe decir que no solo el número de hilos varía entre los diferentes tipos de procesadores, sino que también varían los tamaños de memora caché y prestaciones específicas (por ejemplo, la capacidad de conectividad con otros procesadores). Dentro de un mismo tipo de procesadores (por ejemplo, los clientes), se encuentran muchas variantes (por ejemplo, dentro de la familia Sandy Bridge de tipo cliente, se pueden encontrar más de treinta variantes).

Tratar de apoyar toda esta variedad de número de hilos limitándose a escalar la cantidad de hilos que la arquitectura SMT soporta sería extremamente costoso desde el punto de vista de la producción. Es decir, la complejidad de tener tantos diseños diferentes encarecería mucho más el proceso de diseño, producción y validación de los procesadores. Tal como se va a estudiar en el apartado siguiente, usando tecnologías multinúcleo este proceso se vuelve menos costoso y más factible.

### 6.2. El concepto de multinúcleo

Durante los últimos apartados se han estudiado diferentes estructuras multihilo. Cada una implementaba un procesador superescalar fuera de orden añadiendo el concepto de hilo. Independientemente del tipo de arquitectura multihilo (*super-threading* o SMT), estos procesadores se podrían ver como un elemento de computación, con *n* hilos y una jerarquía de memorias caché. Esta abstracción (como se puede ver en la figura 12) se puede denominar núcleo. Hay que destacar que, en este caso, no incluye otros elementos que un procesador superescalar sí incluiría, como el sistema de memoria o el acceso a la entrada y salida.

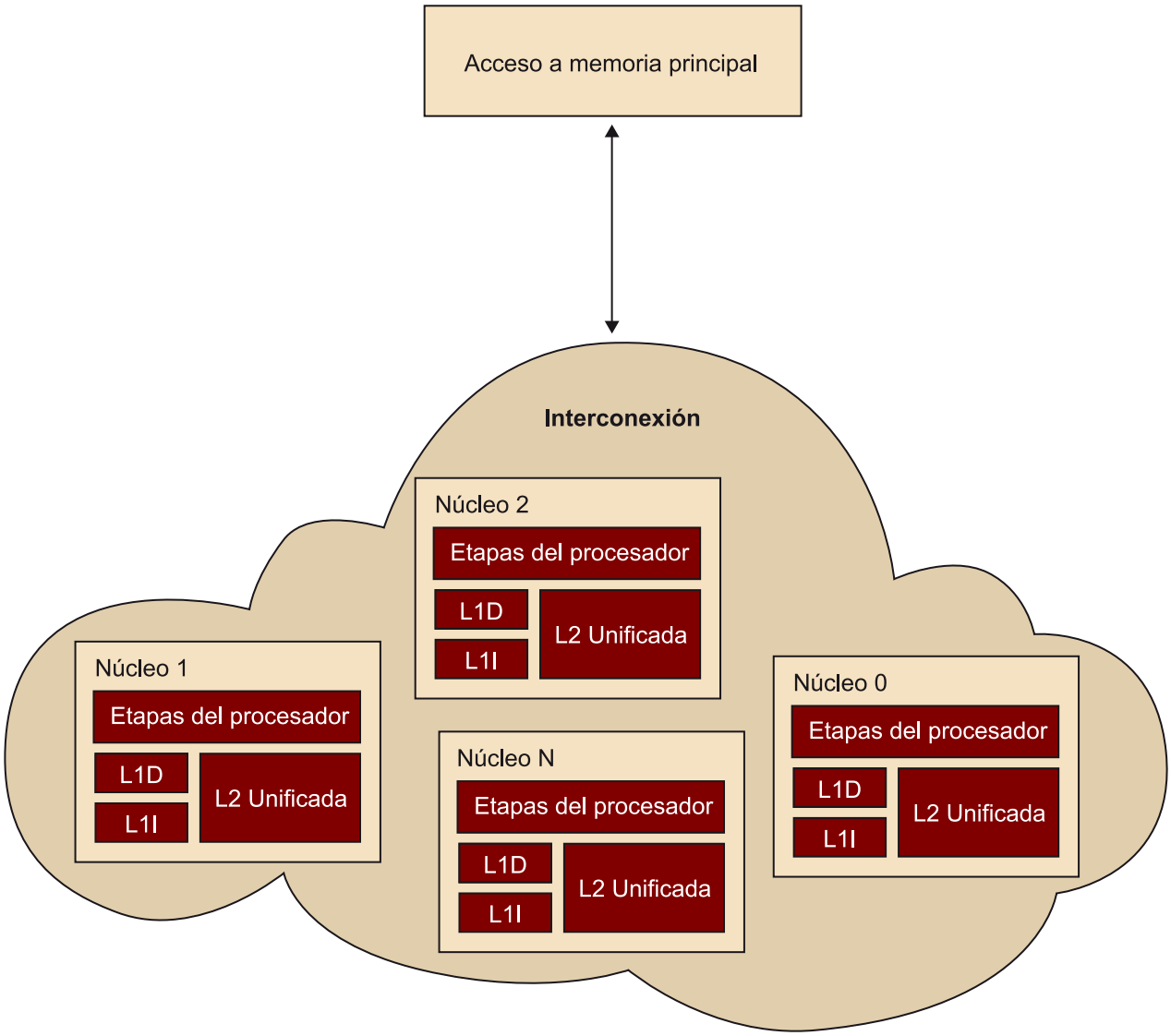
Figura 12. Abstracción de procesador multihilo




De hecho, el concepto de multinúcleo, como su nombre indica, consiste en replicar *m* núcleos diferentes dentro de un procesador (figura 13). Cada núcleo acostumbra a tener una memoria caché de primer nivel (de datos e instrucción) y puede tener una memoria de segundo nivel (suele ser unificada: datos más instrucciones). Aparte de los núcleos, el procesador acostumbra a tener otros componentes especializados y ubicados fuera de estos. Habitualmente, existen los siguientes: una memoria de tercer nivel, un controlador de memoria y componentes para hacer procesamiento de gráficos, entre otros.

Todos estos componentes (incluidos los núcleos) se encuentran conectados mediante una red de interconexión. Esta 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 se ha introducido, uno de los componentes de un multinúcleo fundamental es el controlador de memoria. Este gestiona las peticiones de acceso al subsistema de memoria que hace el resto de componentes (tanto lecturas como escrituras).

Por sí misma, una arquitectura multihilo puede parecer sencilla. Sin embargo, tras este tipo de arquitecturas hay mucha complejidad escondida que no se ve de manera directa, como protocolos de coherencia, escalabilidad en la interconexión, sincronización o desbalanceos entre hilos.

Durante los próximos subapartados se profundiza en los diferentes conceptos que definen los multinúcleo. En primer lugar, se presentan algunas variantes de la arquitectura del modelo que se ha descrito. En segundo lugar, se estudian las dos características más importantes que definen un multihilo, las arquitecturas de interconexión y el mecanismo de coherencia. Finalmente, se presentará una de las arquitecturas multinúcleo comercial disponibles en el mercado.

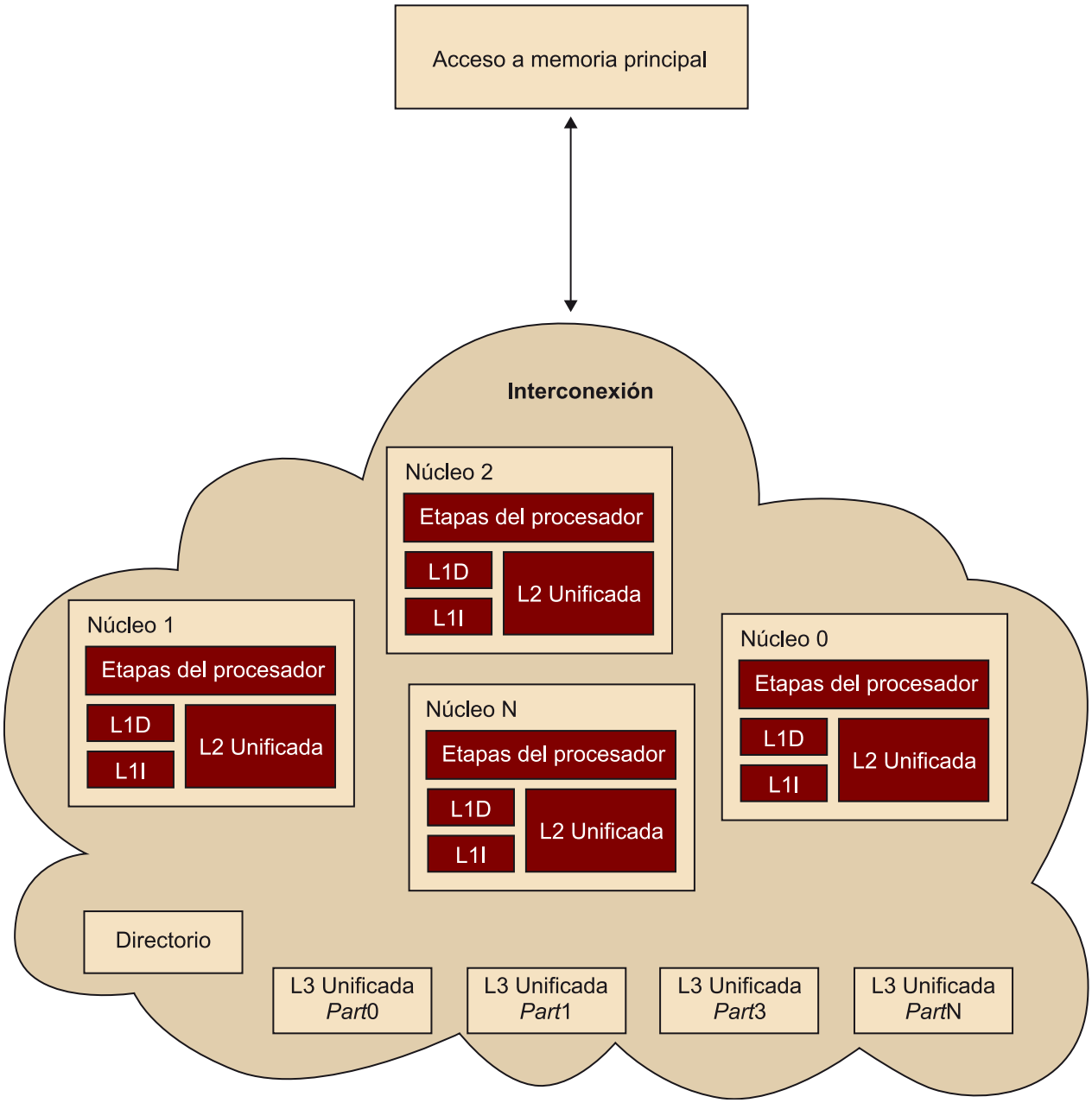
### 6.3. Variante de multinúcleo con L3 y directorio

El modelo tratado en el subapartado anterior describe la estructura más sencilla de un multinúcleo: núcleos, interconexión y acceso a la memoria principal. Sin embargo, como 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 primer nivel y otra 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 la L2, esta L3 se encuentra dividida en bloques del mismo tamaño y ubicada en componentes separados. Tal como se presenta 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, 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 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 el caso de que ningún componente la tenga, se encargará de pedirla a la memoria, guardarla en el bloque correspondiente de la L3 y enviarla al núcleo. Como se va a ver más adelante, esto se hace a partir 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 al directorio y este gestiona la petición a 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 se está mostrando en la tabla 7. 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, recibirá un valor incorrecto. En el caso del ejemplo, el núcleo 2 recibiría un valor erróneo.

Tabla 7. 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. Estos están diseñados para evitar situaciones como la presentada y mantener la coherencia entre todos los núcleos del procesador.

## 6.4. Diseño de arquitecturas multinúcleo

En el apartado anterior, se han introducido dos arquitecturas multihilo, así como sus aspectos más relevantes. En este apartado, se presentan dos aspectos representativos de un sistema multihilo y que definen en buena parte su rendimiento, la interconexión y los mecanismos de coherencia.

### 6.4.1. Las interconexiones

Dentro del ámbito académico, se han propuesto muchas maneras diferentes de conectar los diferentes elementos o componentes que forman un procesador multinúcleo. La mayoría de estas propuestas provienen de búsqueda ya hecha en el entorno de computación de altas prestaciones o *High Performance Computing* (HPC).

En este ámbito, el problema de conectar diferentes componentes también aparece, pero a una escala mayor. Es decir, en vez de conectar núcleos, se conectan procesadores entre ellos o clústeres. El entorno HPC tiene mucho más rodaje en la búsqueda de este tipo de problemas dado que es una ciencia que trabaja en estos problemas desde bien entrada la década de 1980, cuando los primeros ordenadores HPC fueron diseñados.

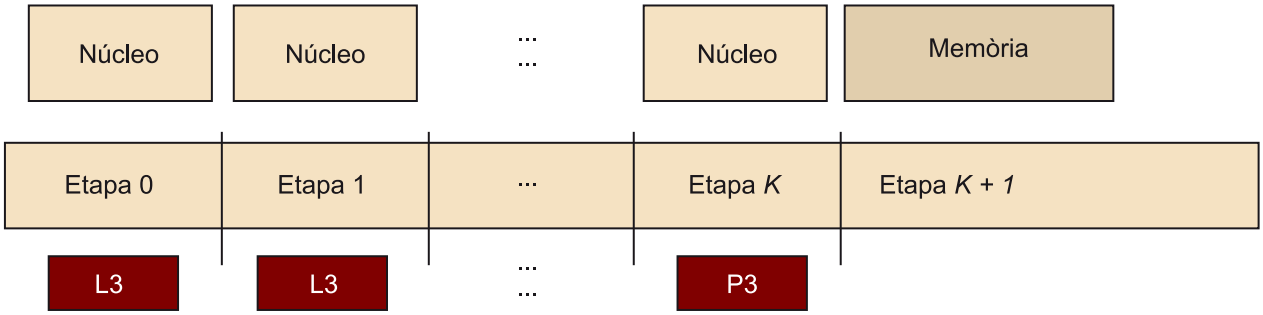
En cuanto a las redes de interconexión, se han propuesto de muchos tipos (Agrawal, Feng y Wu, s. f.), desde simples buses hasta estructuras tridimensionales complejas como las topologías *Torus* o el *Crossbar*. Sin embargo, por temas de complejidad o coste, no todas son aplicables al mundo de los multinúcleos,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

A continuación, se presentan tres de los diferentes modelos de redes de interconexión. Es importante destacar que la bibliografía en este ámbito es muy extensa. En este apartado, se hará una introducción a los modelos más básicos y se tratarán las características y los problemas más generales que tienen. Para una profundización mayor, es necesario analizar las publicaciones referenciadas.

**Red de tipo bus**

Este tipo de red se caracteriza porque usa un solo bus bidireccional en el que solo se pueden enviar transacciones de un punto a otro durante un intervalo de tiempo (Ceder y Wilson, s. f.). Es un bus compartido, por lo tanto, si un componente (como un núcleo lateral) quiere enviar información a otro (por ejemplo, la memoria), ningún otro componente puede inyectar datos al bus. Todos los núcleos se encuentran conectados a un árbitro que gestiona el acceso al bus.

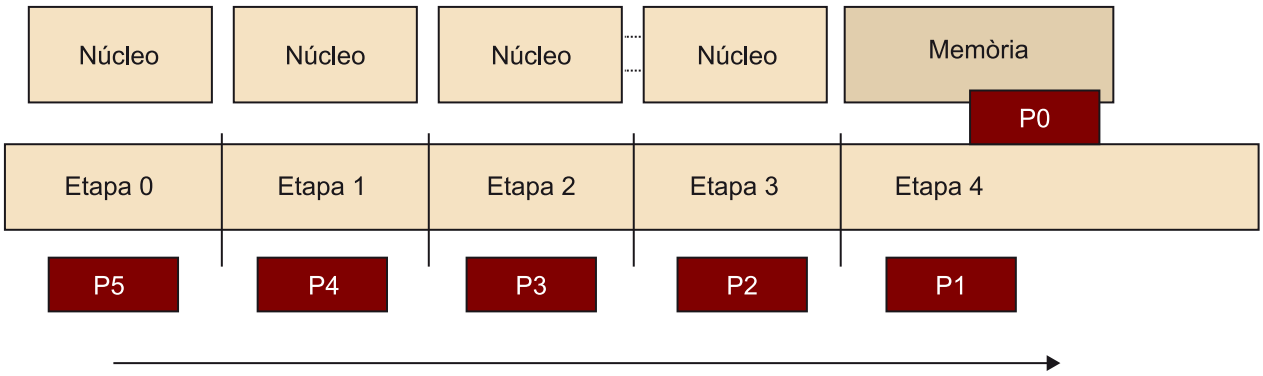
Figura 15. Interconexión de tipo bus




El bus tiene una capacidad de *k* bits. Por lo tanto, si un componente quiere enviar una transacción que ocupa más de *k* bits dividirá los datos que hay que enviar en *m* paquetes (en inglés *flit*). Por otro lado, en la mayoría de casos, no es posible enviar un paquete entre dos componentes en un ciclo. Por lo tanto, en estas ocasiones, el bus se encuentra segmentado. Un paquete tardará varios ciclos en llegar al destinatario. Aun así, como se encuentra segmentado, se podrá inyectar un paquete por ciclo. La figura 16 muestra un ejemplo: el núcleo cero envía una transacción que ocupa seis *flits*. Cada *flit* tarda cuatro ciclos en llegar a la memoria y la transacción tarda un total de nueve ciclos en llegar entera a la memoria (el último paquete se inyecta seis ciclos más tarde que el primero).

Cuando un componente del procesador quiere enviar una transacción o dato a otro componente, antes es necesario que tome la propiedad del testigo del bus. Una vez recoja el testigo, puede emplear el bus durante *n* ciclos consecutivos (no fijados). Hay que tener presente que una transacción puede necesitar un número arbitrario de paquetes. Una vez ha acabado, devolverá el testigo al árbitro.

Figura 16. Transmisión de una transacción del núcleo a la memoria




La ventaja de este tipo de interconexión es la simplicidad. No obstante, el rendimiento respecto a las transacciones por segundo que se pueden enviar es muy bajo. Uno de los otros problemas que presenta es la falta de escalabilidad. Esta arquitectura puede funcionar para un número relativamente bajo de elementos. Ahora bien, si se quieren construir sistemas con muchos núcleos y componentes (bloques L3, directorio) el ancho de banda que el bus puede dar no escala lo suficiente. En estos casos, las peticiones de los núcleos experimentarán latencias muy altas, puesto que todos los demás componentes también estarán intentando acceder al bus. Hay ciertas variantes de esta arquitectura que dan más escalabilidad, como es el caso de los multibuses (Reed y Grunwald, 1987), que dan más rendimiento que la presentada en este subapartado.

Como veremos, los dos tipos de redes siguientes, a pesar de ser más complejos, permiten sacar mucho más rendimiento y son más escalables.

**Red de tipo anillo**

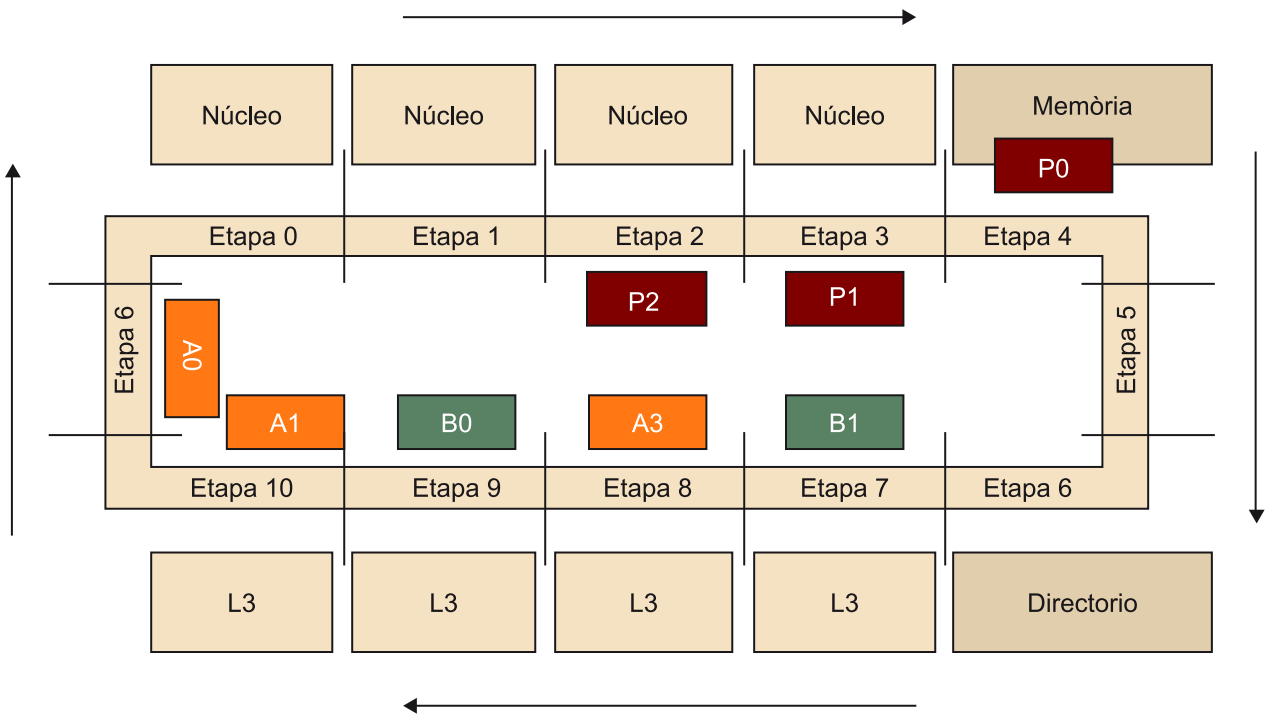
Un bus tan solo permite acceder a dos componentes en el canal de comunicación en un mismo instante de tiempo (Hong y Payne, 1989). Una extensión trivial de este modelo es definir un bus que permita inyectar a todos los componentes de manera simultánea.

Sin embargo, por construcción, esto tiene problemas tanto físicos como lógicos. Por ejemplo, si dos componentes quisieran enviar paquetes de manera cruzada eso no sería posible, dado que sus señales toparían en el medio físico y el resultado sería algo indeterminado.

Una manera de hacer más escalable la arquitectura presentada antes es conectando los extremos de los buses. En este caso, como se puede observar en la figura 17, la red resultante tiene forma de anillo y los paquetes circulan siempre en una misma dirección para evitar estas colisiones. Por un lado, la dirección tanto puede ir en el sentido de las agujas del reloj como en el contrario. Por el otro lado, el anillo se encuentra segmentado en diferentes etapas, de forma que diferentes paquetes de diferentes transacciones pueden circular a la vez. Los diferentes componentes del procesador se encuentran conectados en una de las etapas del anillo y podrán inyectar a la red siempre que la etapa correspondiente se encuentre libre.

En este tipo de arquitectura, cuando un componente quiere enviar una transacción (dividida en diferentes paquetes) a otro componente, debe verificar que la etapa del anillo en la que se encuentra ubicado no se encuentra ocupada por ningún otro paquete. Por ejemplo, en la figura 17, se pueden observar tres paquetes de la transacción P que se mueven del núcleo 2 a la memoria (P0, P1 y P2) y cuatro paquetes de dos transacciones que van de la memoria a los núcleos 1 y 2. Hay que destacar que, en este caso, el núcleo 3 no podría inyectar ningún paquete y que los paquetes de la transacción A y B se encuentran intercalados.

Figura 17. Arquitectura de tipo anillo




Una de las ventajas claras de este tipo de arquitecturas respecto a un bus es el incremento del ancho de banda disponible de manera sustancial. En este caso, todas las etapas de la red pueden ser usadas por los diferentes componentes conectados al anillo. De este modo, se consigue reducir significativamente la latencia de las transacciones que los diferentes núcleos quieren inyectar.

Del mismo modo, hay que comentar que esta interconexión es más escalable que el bus. Tal como se ha mencionado antes, el ancho de banda de un bus rápidamente queda limitado, en cambio, una estructura en anillo tiene mucha más escalabilidad.

Un ejemplo de procesador comercial que usa esta arquitectura como diseño de interconexión es el Sandy Bridge de Intel. Este tiene un total de cuatro núcleos, cuatro memorias caché L3, un controlador de memoria y un componente de proceso gráfico. Todos ellos se encuentran conectados con una interconexión de tipo anillo.

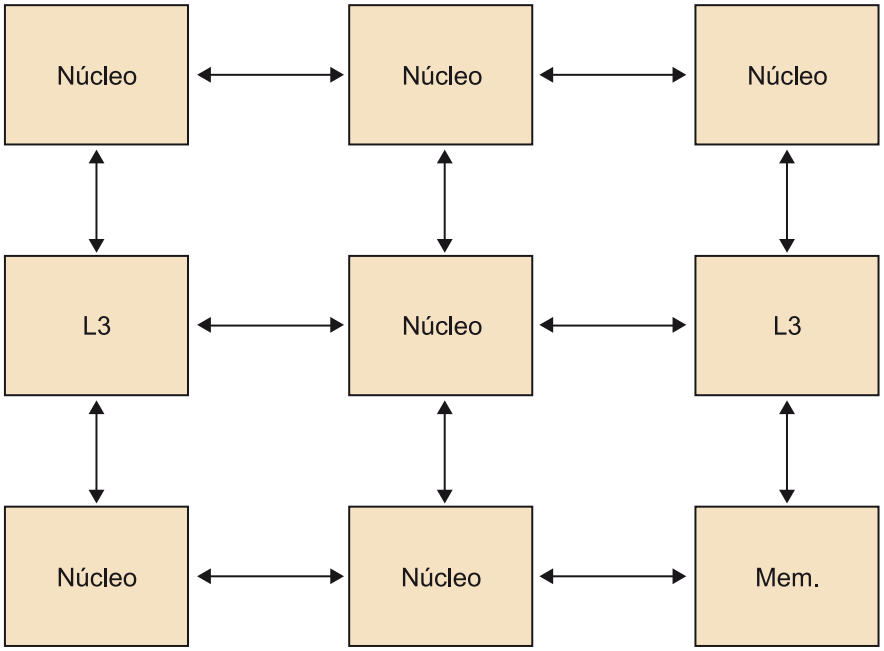
En este subapartado, se ha presentado un anillo unidireccional. Ahora bien, en el ámbito académico, se han propuesto otras soluciones que todavía dan más rendimiento como, por ejemplo, el anillo bidireccional. Esta extensión permite enviar los paquetes en ambas direcciones: en las agujas del reloj y en sentido contrario. De este modo, se está multiplicando el ancho de banda de la arquitectura por dos y se reduce el tiempo de viaje de los paquetes de manera notoria. Aun así, este descenso no es gratuito puesto que se necesita el doble de cables para enviar la información. Por lo tanto, hay que tener en cuenta que esto implica el doble de área y el doble de consumo energético.

**Red de tipo malla**

El incremento de escalabilidad, ancho de banda y la reducción de latencia de un bus en un anillo es sustancial. No obstante, en sistemas de dimensiones muy grandes incluso un anillo puede no escalar. Algunos estudios académicos (Bell y otros, 2008) han analizado la posibilidad de emplear modelos altamente escalables cuando tenemos requerimientos muy elevados. Un ejemplo es la malla o *mesh*.

Tal como se puede observar en la figura 18, los diferentes componentes del procesador se encuentran conectados en forma de malla, es decir, tienen conexiones en el eje de las abscisas y de las ordenadas. Cada componente se conecta a la malla con un router. Este se encarga de dirigir los paquetes que le vienen de los routers a los que está conectado hacia los routers adecuados siguiendo una política de asignación de ruta concreta.

Figura 18. Arquitectura de tipo malla




Las mallas tienen mucha escalabilidad y dan un ancho de banda muy elevado. De hecho, se pueden encontrar muchas variantes dado que, por ejemplo, hay muchos tipos de políticas de asignación de rutas. Ahora bien, estas arquitecturas son altamente complejas y, debido a esta complejidad, se encuentran problemas altamente difíciles de solucionar, como es el caso de los interbloqueos (*deadlocks*).

### 6.4.2. Coherencia

Al principio de este apartado se ha introducido el concepto de coherencia entre núcleos. Estos mecanismos permiten que los accesos a la memoria sean coherentes entre ellos en todo momento. Por lo tanto, siempre que un núcleo está leyendo o escribiendo un dato se puede asegurar que es su última versión.

En este subapartado, se presentan los tipos de mecanismos de coherencia propuestos, así como un ejemplo.

#### Protocolos de coherencia

Los protocolos de coherencia deben garantizar que en todo momento la visión global del espacio de memoria es coherente entre todos los núcleos. Dada una línea de memoria @X:

- Si uno de los núcleos quiere hacer una lectura recibirá la última copia. No se puede dar el caso de que otro núcleo lo esté modificando mientras este tiene una copia.

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

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

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

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

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

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

**1) Los protocolos de tipo snoop**

En estos protocolos (Katz, Eggers, Wood, Perkins y Sheldon, 1985), cuando un núcleo quiere ejercer alguna acción sobre una dirección de memoria, de lectura o de escritura, tiene que llevar a cabo las notificaciones pertinentes al resto de núcleos para obtener el estado querido. Por ejemplo, si quiere tener una dirección de memoria en exclusiva para modificarla, el núcleo primero debe invalidar todas las copias que los demás núcleos tengan. Cuando todos los demás núcleos hayan respondido notificando que han procesado la petición, el núcleo ya podrá usar la línea. Sin embargo, antes, lo tendrá que leer de la memoria o de la última memoria caché (L3).

**2) Los protocolos basados en directorio**

A diferencia de los protocolos de tipo snoop, aquí los núcleos no se encargan de gestionar la coherencia cuando quieren usar una dirección de memoria (Chaiken, Fields, Kurihara y Agarwal, 1990). En este caso, aparece un compo-

nente nuevo que se encarga de gestionarla, el directorio. Cuando un núcleo quiere una línea en un estado concreto, la pide al directorio. El directorio suele tener una lista concreta de los núcleos que tienen la dirección en cuestión y en qué estado la tienen. Por lo tanto, cuando procesa una petición ya sabe a qué núcleos tiene que avisar y a cuáles no.

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

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

**Ejemplo: protocolo de coherencia**

En este subapartado, se estudia un ejemplo de protocolo de coherencia basado en snoop. Como modelo de procesador se considera uno de cuatro núcleos, una memoria caché de tercer nivel dividida en bloques, un directorio que controla qué componentes de la L3 tienen las direcciones y un controlador de memoria. Para entender cómo funciona este protocolo, se estudia el flujo de mensajes que genera cada operación que quiere iniciar el procesador. El protocolo que se explica es un protocolo sencillo. Este se podría mejorar con ciertas optimizaciones que se dejan como ejercicio del estudiante.

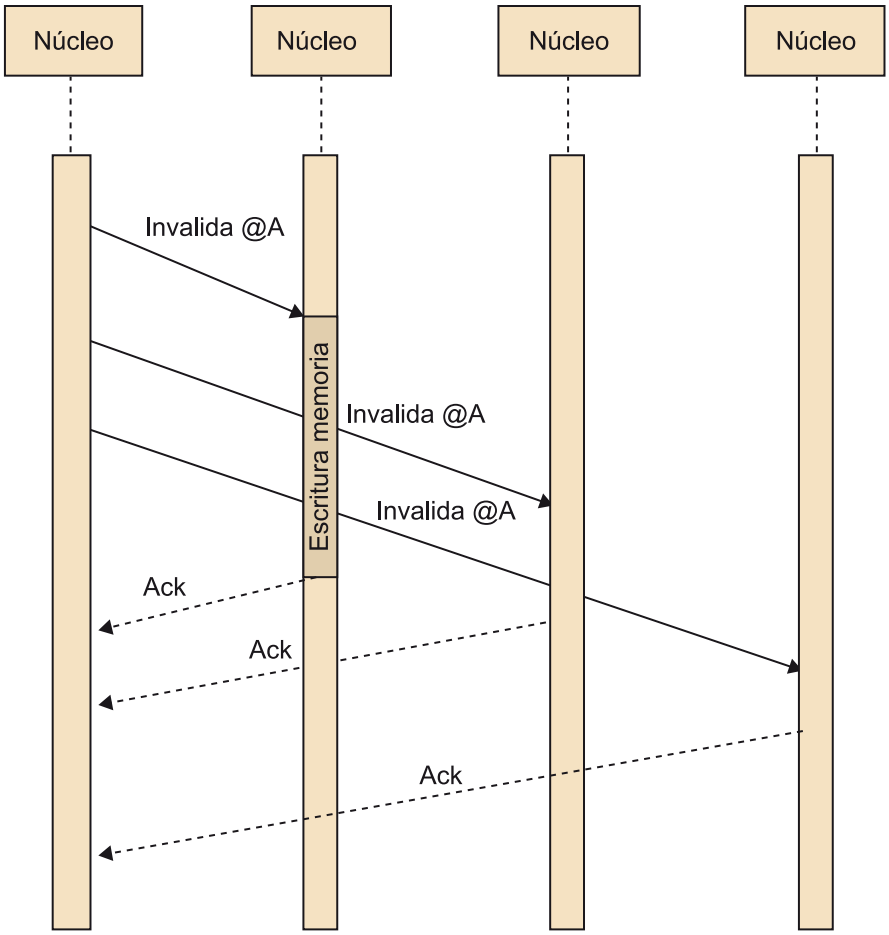
**Invalidación de copias**

Como se ha mencionado en los últimos puntos, cuando un núcleo quiere leer una línea de memoria para modificarla, antes tiene que validar que ninguno de los demás núcleos la tenga. Para hacerlo, tendrá que enviar *snoops* para invalidar todos los demás núcleos que la tengan (figura 19). Los núcleos que tengan la línea en estado compartido o exclusivo, la tendrán que marcar como inválida en sus L1 y L2.

En el caso de que uno de los núcleos la tenga modificada, lo tendrá que escribir en la memoria caché de último nivel y, entonces, invalidarla en sus L1 y L2. Hay que recordar que, si la línea se encuentra modificada, solo la puede tener un núcleo. Dependiendo del tipo de política de memoria caché del pro-

cesador, la línea se escribirá en la L3 o en la memoria: inclusiva/no inclusiva y *write-back*/*write-through* (Handy, 1998). En este ejemplo se asume que es inclusiva y *write-back*. Queda como tarea del lector estudiar qué diferencias existen entre estos cuatro tipos de políticas.

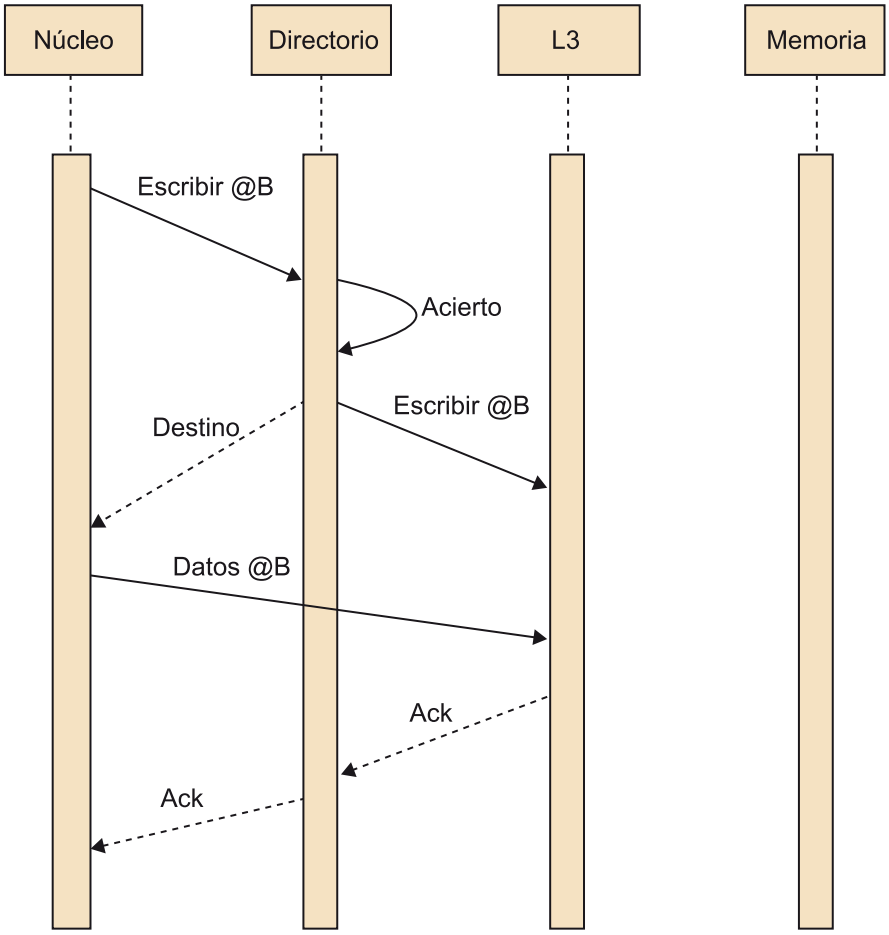
Figura 19. Invalidación de la línea @A




**Escritura en la memoria**

Cuando un núcleo recibe una petición de una línea que tiene guardada en su L2 en estado modificado, la tiene que escribir en la L3. Tal como se muestra en la figura 20, el núcleo primero notifica al directorio que quiere escribir la línea en la L3.

Figura 20. Escritura en la memoria




El directorio, usando las estructuras internas, busca cuál de los bloques de memorias caché L3 contiene la dirección @B. Una vez esta búsqueda ha finalizado, notifica al núcleo la dirección del bloque en cuestión y notifica al bloque que recibirá datos correspondientes a la victimización de la dirección @B. Hay que resaltar que este paso no es estrictamente necesario, pero sí puede dar mejor rendimiento, puesto que permite que la L3 prepare las estructuras pertinentes para tramitar la transacción.

Una vez el núcleo recibe la dirección donde tiene que enviar el dato, genera los paquetes necesarios hacia la L3 enviando los datos de @B. Cuando la memoria caché recibe los datos hace la escritura en el dispositivo. Una vez finalizado, notifica al directorio que ha acabado y el directorio lo notifica al núcleo. En este punto, las estructuras relacionadas con la transacción son liberadas del núcleo y del directorio.

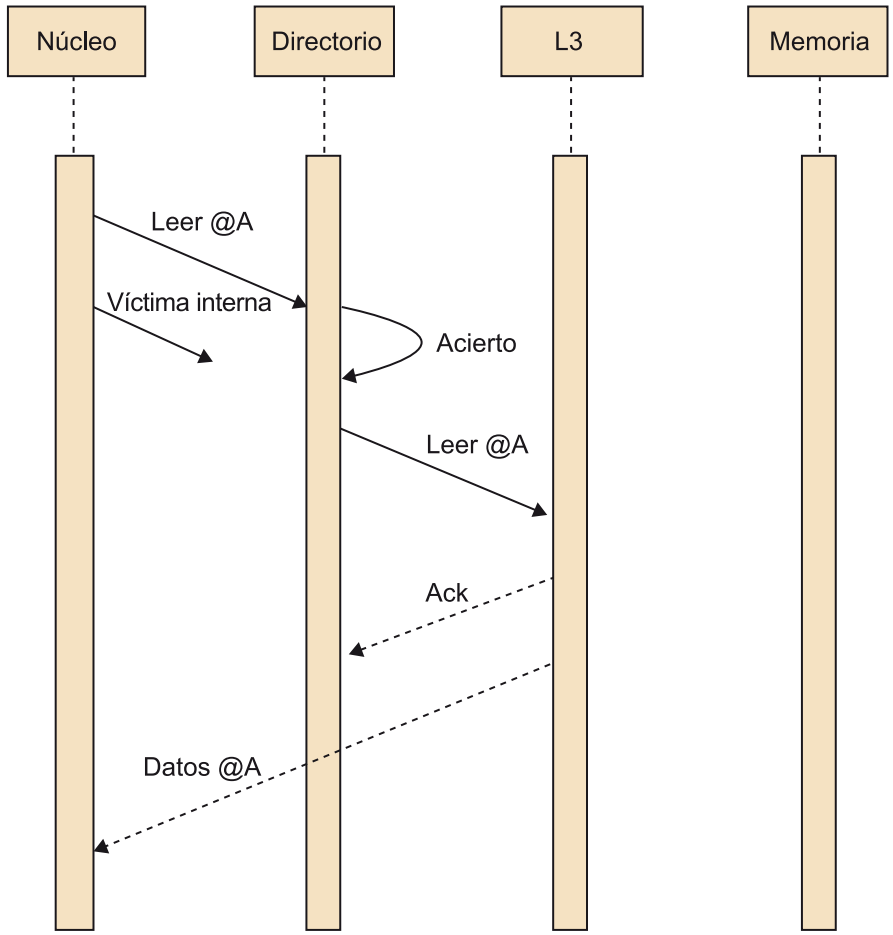
**Lectura de una dirección con acierto en la L3**

La figura 21 presenta el flujo de mensajes que genera una transacción de lectura de un núcleo que acierta la memoria caché de tercer nivel. El núcleo envía la petición al directorio. Este, otra vez usando las estructuras internas, busca la línea con dirección @A. Una vez finaliza la búsqueda, ve que ha acertado y del resultado extrae el bloque de memoria caché que tiene la dirección en

cuestión. El directorio envía la petición de lectura al bloque. El bloque procesa la transacción y envía los datos al núcleo y la confirmación de que lo ha procesado al directorio. El directorio libera todas las estructuras asociadas a la transacción una vez ha recibido la confirmación y el núcleo escribe los datos en su L2.

Como se puede observar, al principio de todo el núcleo ha generado una víctima interna. Siempre que se pide un dato a la L3, hay que liberar una entrada de la misma L2 (un *way* y un *set*; Handy, 1998). De lo contrario, no tendría espacio para guardar el dato que se está pidiendo. Este dato victimizado, en caso de estar modificado, genera una transacción de escritura hacia la L3 siguiendo el flujo de escritura de la memoria. En el caso de que la línea en cuestión no esté modificada, el núcleo no generará ninguna transacción hacia el directorio, simplemente liberará los recursos de la L2 en los que está guardada.

Figura 21. Lectura en la L3 con acierto




**Lectura de una dirección con error en la L3**

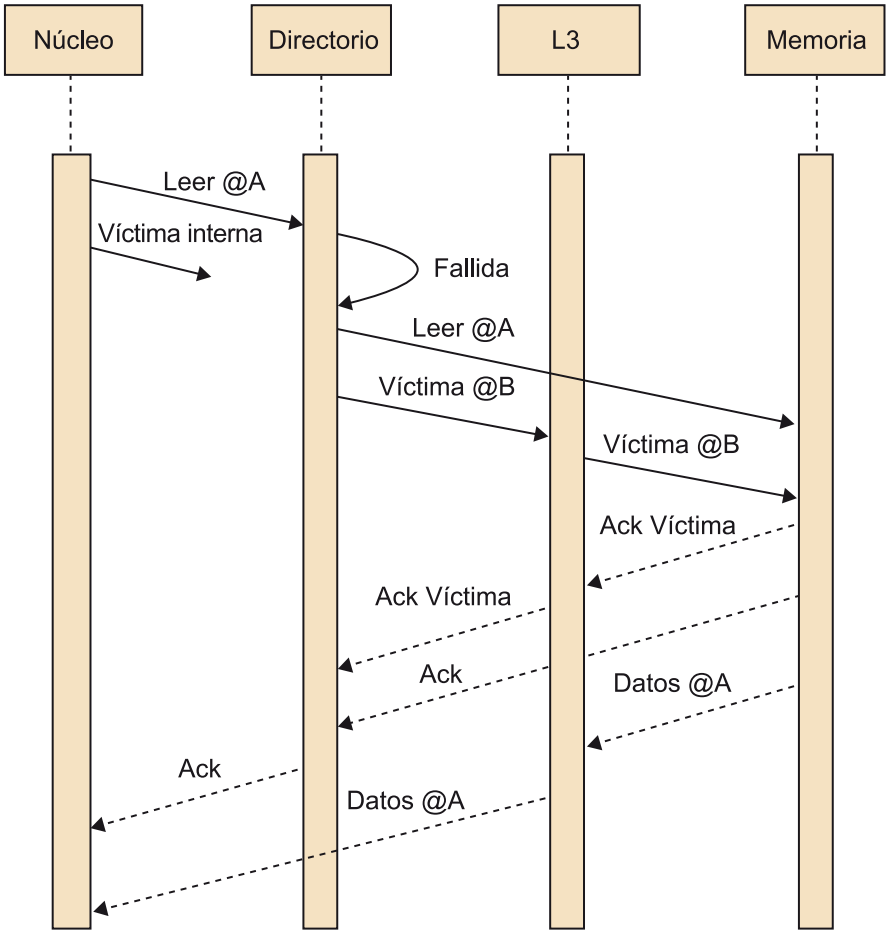
En el caso anterior se ha visto el flujo de instrucciones que genera una lectura en caso de acierto en la memoria caché de último nivel. En este caso, se analiza qué pasa cuando falla en el directorio.

Tal como se puede observar en la figura 22, el flujo de mensajes es exactamente igual hasta la llegada de la lectura al directorio. Tanto la lectura como la generación potencial de una víctima interna en la L2 (para dejar espacio para los datos de @A) son comunes.

Ahora bien, en este caso, el resultado de la búsqueda en la L3 es un error. Aun así, llegado este punto, el directorio también ha podido calcular cuál de las L3 tendría que guardar los datos de @A. Hay que recordar que, asumiendo un modelo inclusivo de memoria, la línea @A tendrá que estar presente tanto en la L2 del núcleo que lo ha pedido como en la L3. Por lo tanto, cuando el directorio pida los datos al componente de memoria, este los tendrá que enviar tanto a la L3 como al núcleo.

Hay que destacar que, de manera similar a la generación de una víctima en el núcleo, para escribir @A la L3 deberá generar una víctima para dejar espacio para la nueva dirección. Por lo tanto, tal como se puede observar en la misma figura, el directorio envía una transacción al bloque correspondiente de la L3 donde notifica que tiene que victimizar una línea concreta (@B) de la memoria caché para dejar espacio para @A. Una vez la L3 ha liberado las estructuras pertinentes de @B, envía los datos a la memoria.

Figura 22. Lectura en la L3 con error



En paralelo, el directorio envía la petición de lectura al bloque de memoria para la dirección @A. Cuando la memoria ha finalizado el proceso de escritura, enviará la notificación de fin a la L3 y esta al directorio (esto se puede hacer de formas diferentes).

Una vez finalizado el proceso de lectura de @A, el bloque de memoria enviará los datos al núcleo y a la L3 de manera paralela. Así, la transacción de lectura acabará cuando los datos hayan llegado al núcleo y el directorio haya notificado que también ha acabado la transacción.

### 6.5. Conclusiones

Durante este apartado, se han analizado las limitaciones que las estructuras SMT presentan. Sobre todo, se ha destacado la complejidad y los problemas de escalabilidad que podrían aparecer si se quisiera escalar el rendimiento de estas arquitecturas en sistemas con un número de hilos más elevados (por ejemplo, 32 hilos de ejecución).

Como evolución de estas arquitecturas, y siguiendo explotando el paralelismo a nivel de hilo, se ha presentado el concepto de arquitectura multinúcleo. Este tipo de arquitectura consiste en replicar elementos de proceso varias veces dentro de un mismo procesador. Estos elementos de proceso pueden ser a su vez núcleos multihilo que implementan las etapas de un procesador superescalar fuera de orden SMT. Por lo tanto, estos sistemas pueden tener $n$ elementos de proceso diferentes y, cada uno de ellos, $m$ hilos diferentes. Por lo tanto, se tendría un total de $n \times m$ hilos. Este tipo de arquitecturas son muy escalables, puesto que permiten aumentar el nivel de paralelismo sin aumentar exponencialmente su complejidad.

Ahora bien, tal como se ha podido ver durante todo el apartado, estas arquitecturas también presentan retos interesantes. Los dos puntos más críticos que se han presentado para poder tener un sistema escalable y robusto han sido las redes de interconexión y los mecanismos de coherencia. El primero de todos es relativo a la manera como los diferentes componentes del procesador se encuentran conectados. Y el segundo es relativo a qué mecanismos tiene el procesador para asegurar que los accesos a los datos de la memoria sean coherentes entre los diferentes núcleos.

La temática de los multinúcleo es altamente extensa y compleja. Tal como se ha apuntado durante todo el apartado, se han elaborado muchos trabajos académicos relacionados con este ámbito. Este tipo de arquitecturas ha marcado tendencias presentes dentro del mercado de procesadores y dentro del ámbito de la búsqueda. Por lo tanto, es interesante profundizar en las diferentes referencias proporcionadas para entender más en detalle los diferentes puntos tratados durante este apartado.

# Resumen

En este módulo, se ha profundizado en el ámbito de las arquitecturas multihilo. En primer lugar, se han analizado las limitaciones que presentan las arquitecturas superescalares fuera de orden con un solo hilo. Como se ha visto, el rendimiento que podemos extraer explotando el paralelismo a nivel de instrucción se encuentra altamente limitado por el tipo de aplicaciones que se quieren ejecutar. Para algunas de ellas, a pesar de aumentar infinitamente sus recursos, el rendimiento se encuentra limitado por su característica inherentemente no paralela. Aun así, para las aplicaciones que más escalaban suponiendo recursos infinitos hemos podido ver que su rendimiento no aumentaba de manera infinita.

Este estudio previo motiva la aparición de las arquitectura multihilo. En este tipo de arquitecturas, el rendimiento del procesador aumenta al incrementar el número de hilos de ejecución independientes que este puede correr. Estas explotan lo que se denomina paralelismo a nivel de hilo (TLP). En este caso, aumentando el número de hilos infinitos podemos lograr un procesador teórico con rendimiento infinito, dado que estos hilos pueden ser independientes entre sí. Sin embargo, tal como se ha visto durante el resto de apartados, este tipo de arquitecturas presentan otras complejidades y retos.

Una vez explicada la motivación de las arquitecturas, se han presentado los diferentes tipos multihilo que se han considerado más relevantes durante las últimas décadas. Así, hemos empezado por las arquitecturas *super-threading* para continuar por las arquitecturas *simultaneous multithreading* (SMT) y finalizar por las estructuras multinúcleo.

Como se ha visto, las arquitecturas SMT extienden el concepto de procesador fuera de orden al añadir el modelo hilo. Los hilos pueden ser totalmente independientes, como en las arquitecturas Intel de *hyperthreading*, o bien compartir el mismo espacio de direcciones como el caso del Alpha 21464. Por otro lado, pueden compartir más o menos estructuras de las diferentes etapas (por ejemplo, el *reorder buffer*). En el caso de que las arquitecturas no sean compartidas, cada hilo tiene una copia, de forma que obtiene un rendimiento más elevado pero también un consumo y un espacio mucho más alto.

Las arquitecturas SMT añaden una mejora sustancial respecto a los procesadores que tan solo explotan el paralelismo a nivel de instrucción. Sin embargo, también se ha podido ver que la complejidad de estos diseños es bastante elevada. Además, esta aumenta exponencialmente en función del número de hilos que se quieren facilitar. Por lo tanto, la escalabilidad de este diseño es relativamente limitada.

Como evolución de las arquitecturas multihilo, se han tratado las arquitecturas multinúcleo. Estas extienden el concepto de procesador multiplicando el número de unidades de procesamiento. Este se encuentra compuesto por *n* diferentes núcleos de proceso, en los que cada uno puede ser a la vez un SMT o un *super-threading*. Los diferentes componentes de procesador se encuentran conectados con una red de interconexión y en general se comunican usando protocolos de coherencia. Estos últimos aseguran que en todo momento todos los núcleos diferentes tienen una visión coherente del espacio de direcciones, es decir, que dada una dirección de memoria todos ven el mismo contenido y estado.

Durante la última parte se ha explicado cuáles son las características y cuáles son los puntos de diseño más importantes en estas arquitecturas. Se han explicado algunas de las redes de interconexión y se han tratado las características principales de los mecanismos de coherencia. Cabe decir que tanto en el ámbito académico como en el comercial se ha hecho mucha investigación en el área de los multinúcleos. Por lo tanto, en este módulo tan solo se han cubierto algunos de los aspectos más relevantes. Para lograr un conocimiento más profundo de cada una de sus partes diferentes, hay que profundizar en las referencias propuestas.

# Actividades

**1.** Para profundizar en las técnicas asociadas a la generación y selección de instrucciones en un SMT, leed el artículo indicado a continuación. Hay que enumerar las principales diferencias y propuestas respecto a las mismas etapas de un procesador escalar fuera de orden SMT.

M. Tullsen. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous-Multithreading Processor Dean".

**2.** Enumerad quince características del Hyper-Threading que no se hayan enumerado durante esta unidad didáctica. Como fuente de datos, usad la fuente siguiente:

http://software.intel.com/en-us/articles/introduction-to-hyper-threading-technology/

**3.** 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 hemos entrado en detalle en el impacto de la complejidad de los procesadores multihilo o multinúcleo en el consumo energético. Enumerad las aportaciones más interesantes del artículo siguiente sobre este aspecto.

Benjamin Lee; David Brooks. "Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency".

**4.** Proponed una variante del protocolo presentado en el apartado "La perspectiva del sistema" suponiendo que en la arquitectura considerada suprimimos la memoria caché de último nivel y el directorio. En este caso, solo habrá memoria y núcleos.

# Bibliografía

**Agrawal, D. P.; Feng, T. Y.; Wu, C. L.** (s. d.). "A survey of communication processors systems". *COMPSAC* (págs. 668-673).

**Alverson, R.; Callahan, D.; Cummings, D.; Koblenz, B.** (1990). "The Tera computer system". En: *International Conference on Supercomputing* (págs. 1-6).

**AMD** (s. d.). *Página AMD Fusion* [en línea]. [Fecha de consulta: 21 de diciembre del 2011]. <fusion.amd.com>

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

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

**Ceder, A.; Wilson, N.** (s. d.). "Bus network design". *Transportation Design* (págs. 331-344).

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

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

**Dixit, K. M.** (1991). "The SPEC benchmark". *Parallel Computing* (págs. 1.195-1.209).

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

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

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

**IBM** (2011). *AS/400 Systems*. Recuperado el 20 de diciembre del 2011: <http://www-03.ibm.com/systems/i/>

**Intel** (s. d.). *Sandy Brid'intel*. Recuperado el 10 de diciembre del 2011: <http://software.intel.com/en-us/articles/sandy-bridge/>

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

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

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

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

**Melvin, S.** (2003). "Flowstorm porthos massive multithreading (mmt) packet processor" [en línea]. <www.zytek.com/~melvin/flowstorm.html>

**Melvin, S.** (2000). *Clearwater networks cnp810sp simultaneous multithreading (smt) core* [en línea]. [Fecha de consulta: 19 de diciembre del 2011]. <www.zytek.com/~melvin/clearwater.html>

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

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

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

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

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

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