

clean.sty

Gómez Macías [Marta](#)

Arquitectura de Computadores

Febrero-Junio 2015



Arquitectura de Computadores by [Marta Gómez Macías](#) is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional License](#).

Contents

Chapter 1

Arquitecturas Paralelas: Clasificación y Prestaciones

1.1 Clasificación del paralelismo implícito en una aplicación

1.1.1 Criterios de clasificaciones del paralelismo implícito en una aplicación

En una aplicación se pueden distinguir distintos niveles de paralelismo, que se aprovechan en distintos niveles del computador. Estos niveles de paralelismo se pueden clasificar en función del nivel de abstracción dentro del código secuencial de un programa en el que podemos encontrar implícito el paralelismo.

Cuando hacemos una aplicación podemos dividir el código en programas (*nivel de programas*), a su vez, podemos considerar que un programa está compuesto por funciones (*nivel de funciones*), éstas a su vez están compuestas por bucles (*nivel de bucle*) y éstos, se basan en operaciones (*nivel de operaciones*). Así, el paralelismo puede clasificarse en función de estos niveles. Esto se ve en la Figura ??.

El paralelismo puede clasificarse también en función de la *granularidad* o *magnitud de la tarea* candidata a la parallelización. El grano más pequeño (*grano fino*) se asocia generalmente al paralelismo entre operaciones o instrucciones y el *grano grueso*, al paralelismo entre programas. Entre ambos extremos está el *grano medio* asociado a los bloques funcionales lógicos de la aplicación. La granularidad se refiere al tamaño de los trozos de código que se ejecutan. No es un contexto muy preciso.

El paralelismo también puede clasificarse en paralelismo de datos y de tareas.

1.1.2 Niveles de paralelismo implícito en una aplicación

- ♡ **Nivel de programas:** los diferentes programas que intervienen en una aplicación o en diferentes aplicaciones que se pueden ejecutar en paralelo. No suele existir dependencia entre ellos.
- ♡ **Nivel de funciones:** un programa se constituye de funciones. Las funciones llamadas en un programa pueden ejecutarse en paralelo siempre que no existan dependencias entre ellas.
- ♡ **Nivel de bucle (bloques):** El código dentro de un bucle se ejecuta múltiples veces y en cada iteración, se completa una tarea. Se pueden ejecutar en paralelo las iteraciones de un bucle siempre que se eliminen los problemas derivados de dependencias, es decir, que en la siguiente iteración no se utilicen datos que se han usado en la anterior. Para detectar dependencias se deben analizar las entradas y salidas de las iteraciones del bucle.
- ♡ **Nivel de operaciones:** Las operaciones independientes se pueden ejecutar en paralelo. En los procesadores de propósito general y los de propósito específico, encontramos instrucciones compuestas de varias operaciones que se aplican en secuencia al mismo flujo de datos de entrada (por ejemplo: una instrucción que realiza una multiplicación seguida de una suma). En este nivel se puede detectar la posibilidad de usar instrucciones compuestas, que van a evitar las penalizaciones por dependencias verdaderas.

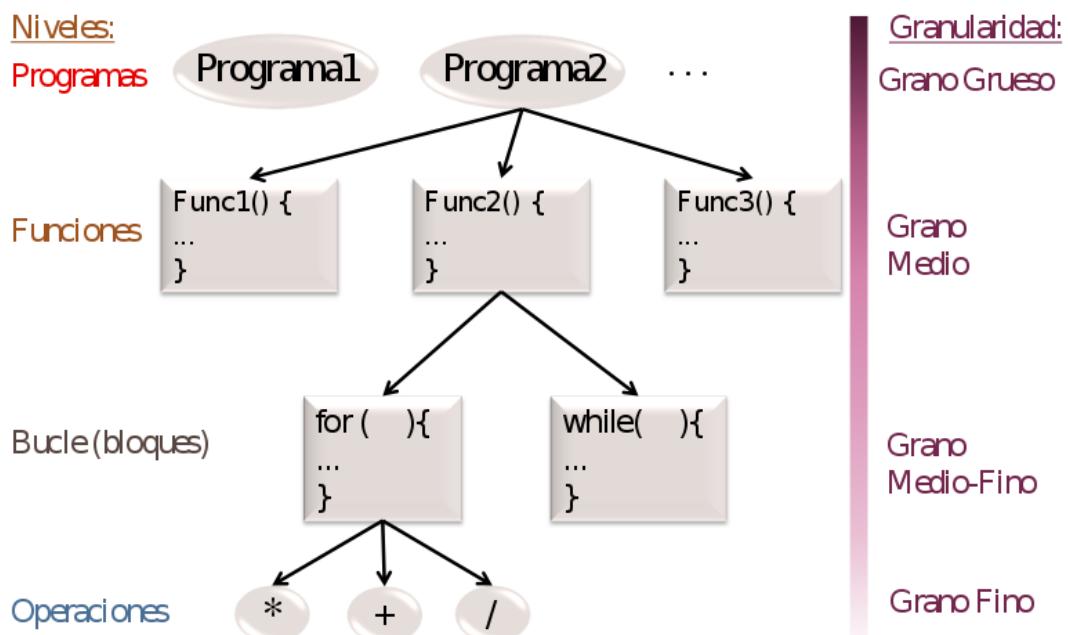


Figure 1.1: Niveles de paralelismo en el código de una aplicación y granularidad

Este paralelismo que se puede detectar en los niveles de un código secuencial se llama *paralelismo funcional*.

1.1.3 Dependencias de datos

Determinan si un código puede ejecutarse en paralelo o no. Son las condiciones que se deben

cumplir para que el bloque de código B_2 presente una dependencia de datos con respecto a B_1 :

- ♡ Deben hacer referencia a una misma posición de memoria M (variable)
- ♡ B_1 aparece en la secuencia de código antes que B_2 .

Los tipos de dependencias de datos (de B_2 respecto a B_1) son:

- ♡ RAW (*Read After Write*) o dependencia verdadera: se lee el contenido de una variable después de haber escrito en esa variable. Se pretende que lo que se lee sea el resultado de lo que se ha escrito anteriormente. Si se están ejecutando a la vez puede que se ejecute la operación de lectura antes que la de escritura y que por tanto, no se lea el valor deseado.

El siguiente código sería un ejemplo de dependencia RAW: $c \dots a = b + c; d = a + c; \dots$

Este código está hecho pensando en que primero se ejecutará $a=b+c$ y después, $d=(b+c)+c$. Si lo ejecutásemos en paralelo, podríamos hacer primero $d=a+c$ y luego, $a=b+c$ obteniendo así, un resultado erróneo.

- ♡ WAW (*Write After Write*) o dependencia de salida: el resultado de una operación se usará después en el código, y si después modificamos el valor de la variable, será porque queremos que se use después con ese valor (de hecho, los compiladores quitan el código muerto, es decir, variables que no se usan). Si ambas operaciones de escritura se ejecutan en paralelo podría ocurrir que se lea un valor incorrecto dependiendo del orden en el que se ejecuten las operaciones.

El siguiente código sería un ejemplo de dependencia WAW: $c \dots a = b + c; \dots a = d + e; \dots$

Primero, escribimos en la variable a el valor $b+c$ para usar dicho valor en alguna operación siguiente. Después, volvemos a usar la variable a para guardar en ella otro valor distinto, $d+e$ y usarlo de nuevo en cualquier otra operación. Si ejecutásemos esto en paralelo, podríamos obtener un resultado erróneo al usar en cualquier operación que espere el valor $b+c$ el valor $d+e$ y viceversa. Este tipo de dependencia la resuelve el compilador guardando ambos valores en registros diferentes.

- ♡ WAR (*Write After Read*) o antidependencia: Si ejecutamos una operación de lectura y otra de escritura posterior en paralelo, podemos escribir en la variable que se está leyendo antes de que dicha variable se lea.

El siguiente código sería un ejemplo de dependencia WAR: $c \dots b = a + 1; \dots a = d + e; \dots$

En primer lugar, leemos el valor de a y luego lo sobreescrivimos. Si ejecutásemos este código en paralelo podríamos ejecutar la instrucción $b=(d+e)+1$ y así, obtener un resultado erróneo. Este tipo de dependencia también la resuelve el compilador usando diferentes registros.

Estas dependencias de datos van a limitar la paralelización. WAW y WAR se pueden eliminar cambiando el destino de salida, poniendo una variable distinta. No sólo lo hace el compilador, el hardware también lo hace. Respecto a los RAW, se pueden eliminar en alguna ocasión a nivel de bucle pero no a nivel de operación.

Siempre que no haya dependencias RAW, podemos parallelizar las operaciones:

- ♡ *A nivel de bucle*: pueden hacerlo los compiladores si se trata de bucles sencillos y, por supuesto, el programador.
- ♡ *A nivel de función*: el compilador lo tiene algo más difícil por varias razones:

- El uso de punteros
- El no expresar en los argumentos de la función todas las variables que se van a usar en dicha función.
- Si además las funciones están en ficheros distintos, tendría que entrar en acción el enlazador y dificultaría aún más la paralelización.

♡ *A nivel de programa*: el compilador no puede paralelizar nada porque genera código máquina para cada programa de manera independiente (no podemos compilar dos programas a la vez).

En resumen, los compiladores extraen mejor el paralelismo a bajo nivel (granos finos) que a alto nivel, cuanto más bajo sea el nivel mejor trabajo hay. Pero este trabajo se puede superar.

1.1.4 Paralelismo implícito en una aplicación

Paralelismo de tareas

El **paralelismo de tareas (*task-parallelism*)** se encuentra extrayendo la estructura lógica de funciones de la aplicación. En esta estructura, los bloques son funciones, y las conexiones entre ellos reflejan el flujo de datos entre funciones. Equivaldría al paralelismo a nivel de función dentro del código de alto nivel.

Se extrae la estructura lógica de las funciones en un grafo: los nodos serían las funciones y las flechas, la secuencia en la que se deberían de ejecutar. Un ejemplo de esto se ve en la Figura ??

Paralelismo de datos

El **paralelismo de datos (*data-parallelism ó DLP-Data Level Parallelism*)** se encuentra implícito en las operaciones con estructuras de datos (vectores y matrices). Se puede extraer de una representación matemática de las operaciones de la aplicación. Las operaciones con vectores y matrices engloban múltiples operaciones con datos escalares, que se pueden realizar en paralelo. Éstas operaciones se implementan mediante bucles. Por tanto, el paralelismo de datos está relacionado con el paralelismo a nivel de bucle. El paralelismo de datos se puede extraer de los bucles analizando las operaciones realizadas con la misma estructura de datos en las diferentes iteraciones del bucle.

En la década de los 90 se han ido incorporando unidades funcionales que implementan paralelismo de datos usando procesamiento SIMD (arquitecturas multimedia). Las instrucciones multimedia aceleran el procesamiento vectorial aplicando la misma instrucción en paralelo a múltiples datos de un registro.

1.1.5 Estructura de funciones lógica de una aplicación. Ej: decodificador JPEG

Se aplica a imágenes donde se dividen las imágenes en bloques de 8x8 píxeles. Cada cuadradito representa un bloque de 8x8 píxeles. El decodificador parte del bloque codificado y lo decodifica a través de las funciones que se indican en el grafo de la Figura ??

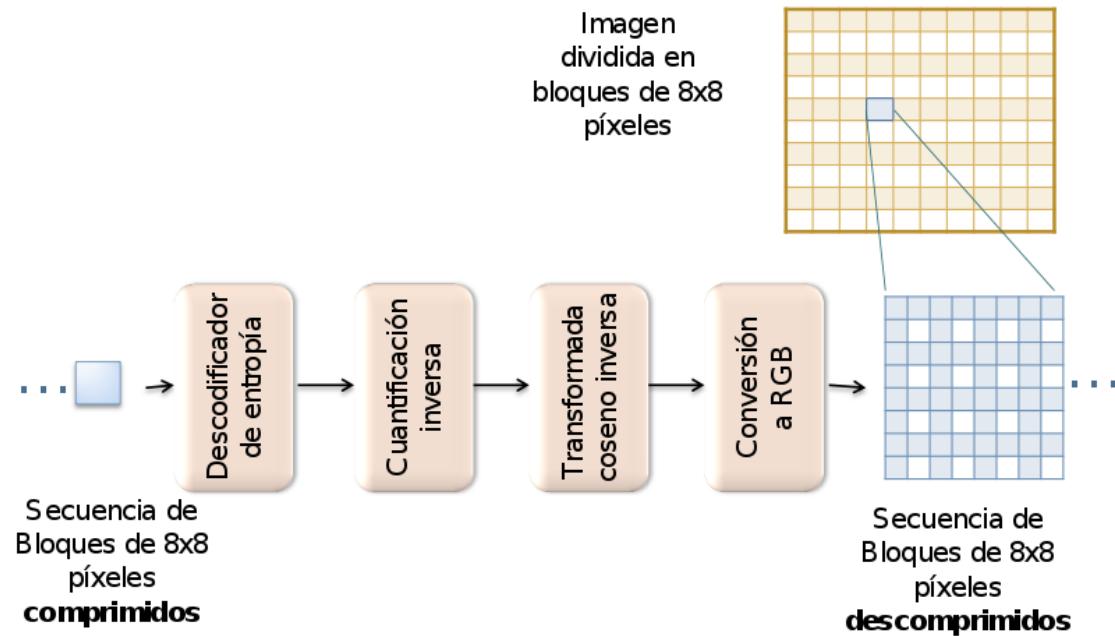


Figure 1.2: Secuencia lógica de funciones seguida por un decodificador JPEG

Una vez extraída la estructura lógica de las funciones, se programarían.

1.1.6 Paralelismo de datos y paralelismo de tareas en OpenMP (Prácticas 1, 2 y 3)

La herramienta de prácticas permite aprovechar el paralelismo a nivel de función y bucle, pero éste debe ser extraído por el programador. En el ejemplo de la Figura ??, las funciones se pueden ejecutar en paralelo, para ello se usan las directivas de la imagen. Con eso se añade lo necesario para ejecutar las funciones en paralelo. Si nos hemos equivocado y hay una función que necesita el resultado que otra produce, obtendremos resultado erróneo pero la herramienta no dirá nada. También podemos hacer esto con bucles añadiendo delante del bucle una frase y si hay dependencias RAW obtendremos resultado erróneo, igual que antes.

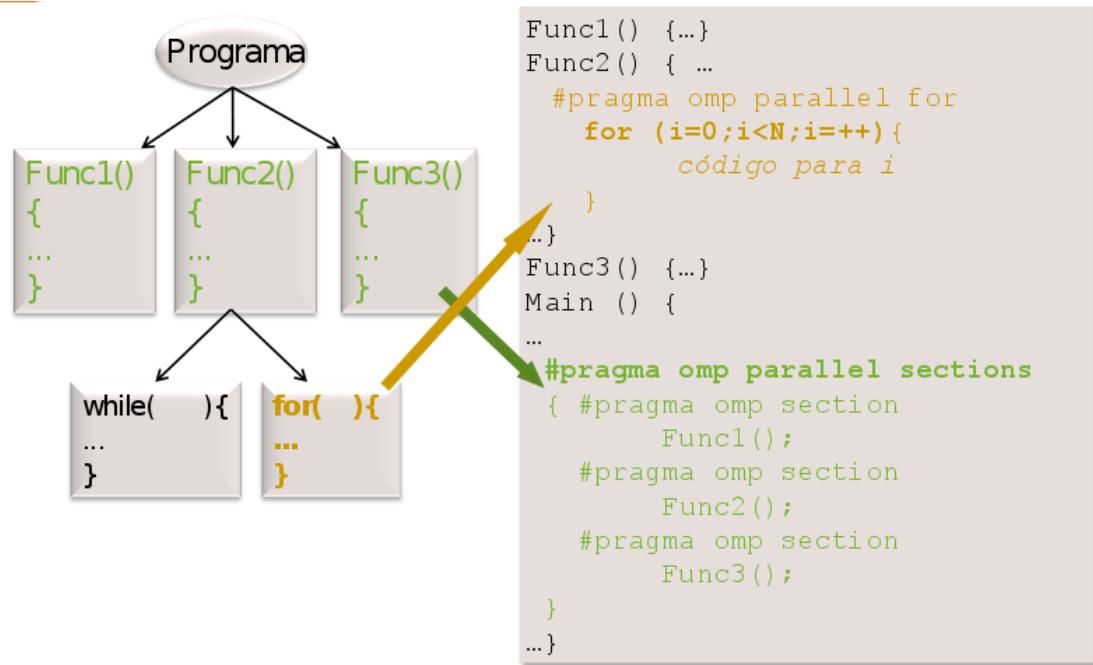


Figure 1.3: Paralelismo en OpenMP

1.1.7 Paralelismo implícito (nivel de detección), explícito y arquitecturas paralelas

En la Figura ?? se relacionan los distintos niveles en los que se encuentra el paralelismo implícito en el código, con los niveles en los que se puede hacer explícito y con arquitecturas paralelas que aprovechan el paralelismo.

El paralelismo entre programas se utiliza a nivel de procesos. En el momento en el que se ejecuta un programa, se crea el proceso asociado al programa. Este paralelismo es aprovechado por arquitecturas multicamputador y multiprocesador.

El paralelismo disponible entre funciones se puede extraer para utilizarlo a nivel de procesos o de hebras y lo aprovechan las arquitecturas multiprocesador y multithread.

El paralelismo dentro de un bucle, se puede extraer también a nivel de procesos o de hebras. Se puede aumentar la granularidad asociando un mayor número de iteraciones del ciclo a cada unidad a ejecutar en paralelo. El paralelismo dentro de un bucle también se puede hacer explícito dentro de una instrucción vectorial, es decir: instrucciones multimedia, para que sea aprovechado por arquitecturas SIMD o vectoriales.

El paralelismo entre operaciones se puede aprovechar en arquitecturas con paralelismo a nivel de instrucción ejecutando en paralelo las instrucciones asociadas a estas operaciones independientes.

Una visión más general de este gráfico se ve en la Figura ??

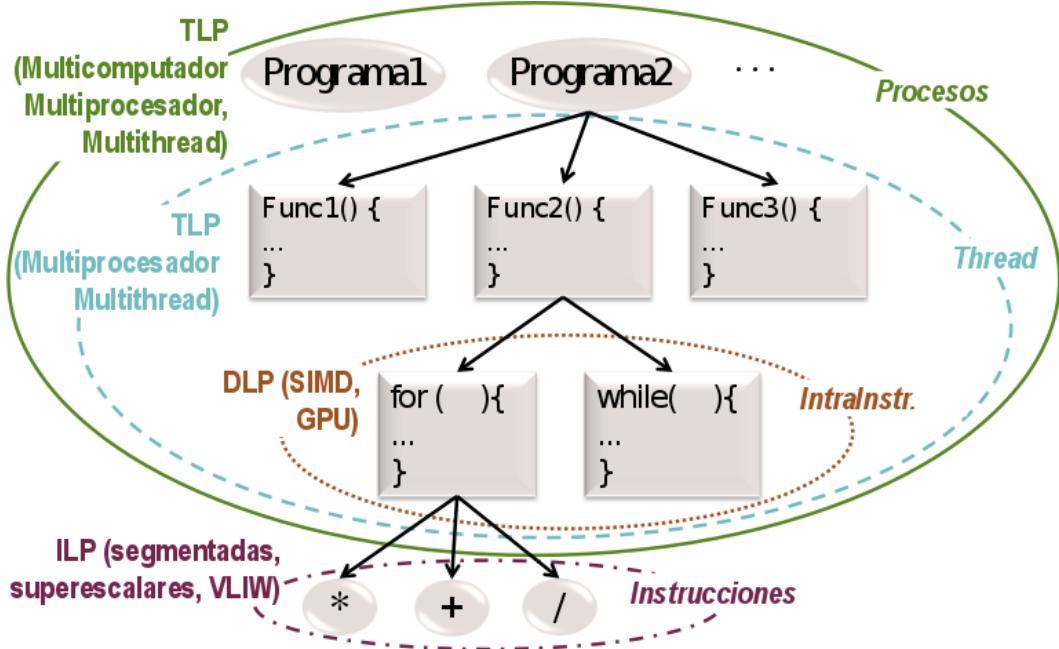


Figure 1.4: Niveles de detección en arquitecturas paralelas

1.1.8 Nivel de paralelismo explícito. Unidades de ejecución en un computador

El hardware se encarga de gestionar la ejecución de instrucciones. A nivel superior, el sistema operativo se encarga de gestionar la ejecución de unidades de mayor granularidad: los procesos y hebras.

- ♡ **Instrucciones**: la unidad de control de un core o procesador gestiona la ejecución de instrucciones por la unidad de procesamiento.
- ♡ **Thread o light process**: es la menor unidad de ejecución que gestiona el sistema operativo y la menor secuencia de instrucciones que se pueden ejecutar en paralelo.
- ♡ **Proceso o process**: es la mayor unidad de ejecución que gestiona el sistema operativo. Consta de uno o varios thread.

1.1.9 Nivel de paralelismo explícito. Thread versus procesos

Cada proceso en ejecución tiene su propia asignación de memoria con todo lo que hace falta para su ejecución: datos en pila, registros, tablas de páginas y ficheros abiertos, etc. Los sistemas operativos multihebra permiten que un proceso se componga de una o varias hebras. Una hebra

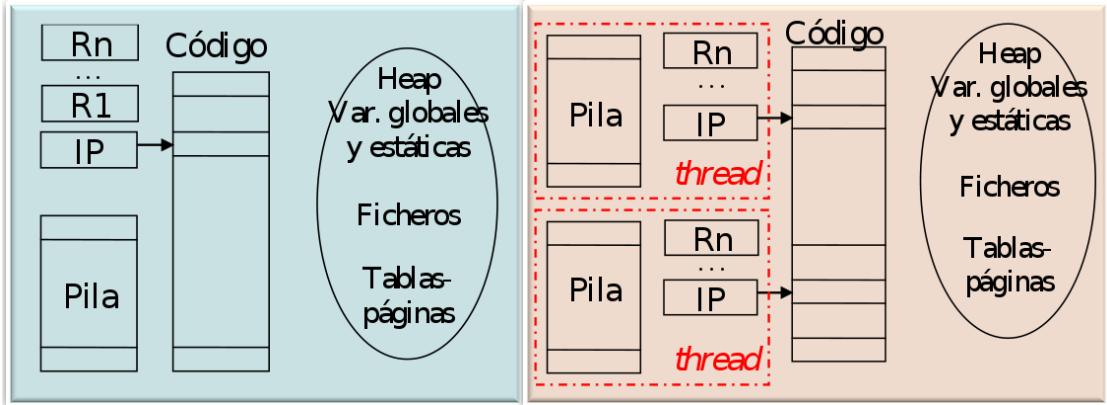


Figure 1.5: Estructura en memoria de un proceso secuencial y multihebrado

tiene su propia pila y contenido de registros pero comparte el código, las variables globales y otros recursos con las hebras del mismo proceso. Estas características hacen que las hebras se puedan crear y destruir en menos tiempo que los procesos, y que la comunicación, sincronización y conmutación entre hebras de un proceso sea más rápida que entre procesos. Todo ello permite que las hebras puedan tener una granularidad menor que los procesos, como se muestra en la Figura ??.



Figure 1.6: Razones por las que las thread tienen menor granularidad que los procesos

Para comunicar procesos tenemos que hacer llamadas al sistema operativo que suponen una mayor sobrecarga que la comunicación entre hebras, pues ésta se realiza en la memoria que comparten.

Las diferencias entre la estructura de las hebras y de los procesos se puede ver en la Figura ??.

El paralelismo implícito en el código de una aplicación se puede hacer *explícito* a nivel de instrucciones, hebras o procesos.

1.1.10 Detección, utilización, implementación y extracción del paralelismo

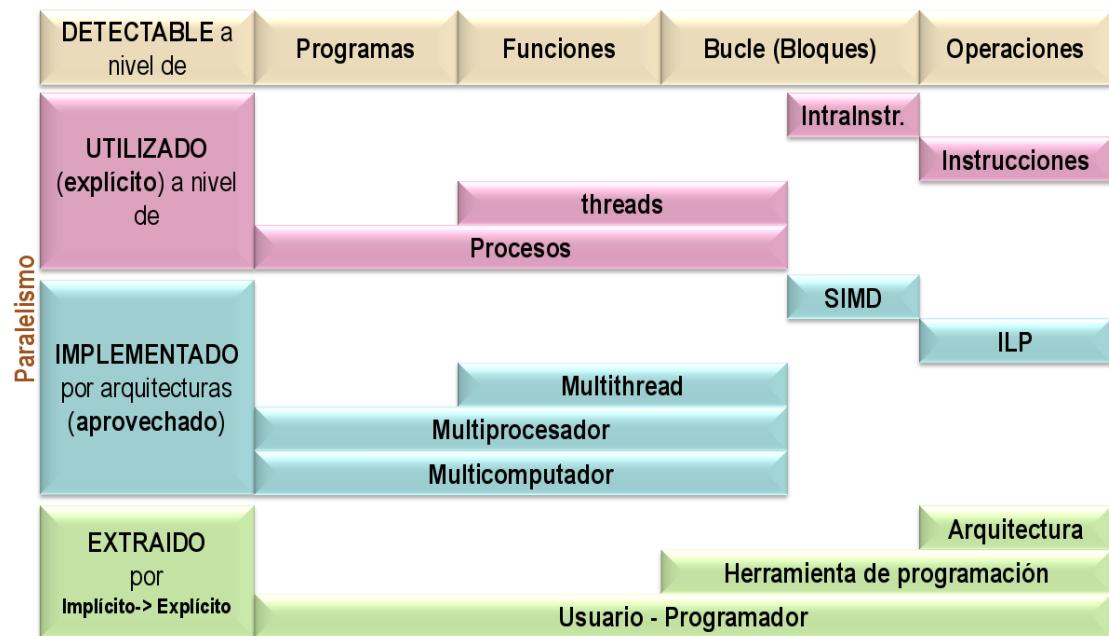


Figure 1.7: Detección, utilización, implementación y extracción del paralelismo

En la Figura ?? se relaciona el agente que extrae el paralelismo implícito en los diferentes niveles del código de la aplicación, con los niveles en los que se hace explícito y las arquitecturas que lo aprovechan.

En los procesadores ILP (superescalares o segmentados) la arquitectura *extrae* paralelismo. Para ello, eliminan dependencias de datos falsas entre instrucciones y evitan problemas debidos a dependencias de datos, de control y de recursos (columna 4). El *grado de paralelismo*¹ de las instrucciones aprovechado puede aumentar con la ayuda del compilador (herramienta de programación) y del programador.

Hay compiladores que extraen el paralelismo de datos implícito a nivel de bucle (columna 3). Algunos compiladores lo hacen explícito a nivel de hebra y otros dentro de una instrucción para que se pueda aprovechar en arquitecturas SIMD o vectoriales.

Aún es difícil para un compilador extraer paralelismo a nivel de función sin la ayuda del programador. El programador puede extraer el paralelismo implícito en un bucle o entre funciones definiendo hebras y/o procesos (columnas 2 y 3).

La distribución de las tareas independientes entre hebras y procesos dependerá:

- ♥ de la granularidad de las unidades de código independientes

¹ Podemos definir el grado de paralelismo de un conjunto de entradas a un sistema como el máximo número de entradas del conjunto que se puede ejecutar en paralelo, este número será menor al número de entradas debido a las dependencias

- ♥ de la posibilidad que ofrezca la herramienta de programación paralela disponible de definir hebras o procesos
- ♥ de la arquitectura disponible para aprovechar el paralelismo
- ♥ del sistema operativo disponible

Los usuarios del sistema al ejecutar programas están creando procesos que se pueden ejecutar en el sistema o bien concurrentemente, o bien en paralelo (columna 1).

1.2 Clasificación de arquitecturas paralelas

1.2.1 Computación paralela — Computación distribuida

Computación paralela : estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un sistema de cómputo compuesto por **múltiples procesadores/cores/computadores** que es visto externamente como una **unidad autónoma** (multicores, multiprocesadores, multicamputadores, cluster²).

El paralelismo en las arquitecturas se ha implementado siguiendo dos líneas fundamentales: *replicación de elementos* entre los que se distribuye el trabajo y la *segmentación de cauce* técnica que divide un elemento en una serie de etapas que funcionan de manera independiente y por las que van pasando instrucciones, operandos, etc.

Computación distribuida : estudia los aspectos hardware y software relacionados con el desarrollo y ejecución de aplicaciones en un **sistema distribuido**; es decir, en una **colección de recursos autónomos** (PC, servidores de datos, aplicaciones, supercomputadores...) situados en **distintas localizaciones físicas**

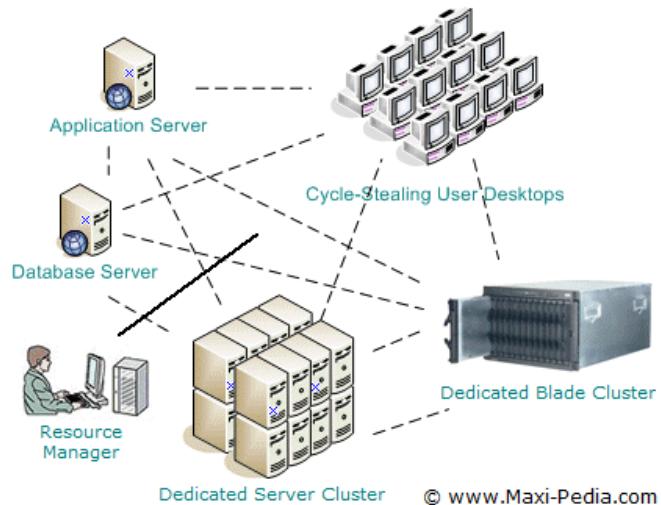


Figure 1.8: El concepto de computación distribuida es simple: reunir y emplear todos los recursos posibles para acelerar la computación.

²Los cluster son computadores paralelos basados en computadores (nodos) y redes disponibles comercialmente, lo que les permite ser los sistemas con mejor relación prestaciones/coste en los diferentes niveles del mercado en los que se encuentran

1.2.2 Computación distribuida a gran escala: Computación grid

Computación distribuida a baja escala: estudia los aspectos relacionados con el desarrollo y ejecución de aplicaciones en una colección de recursos autónomos **de un dominio administrativo** situados en **distintas localizaciones físicas** conectados a través de **infraestructura de red local**.

Computación grid: estudia los aspectos relacionados con el desarrollo y ejecución de aplicaciones en una colección de recursos autónomos de **múltiples dominios administrativos geográficamente distribuidos** conectados con **infraestructura de telecomunicaciones**

La computación grid virtualiza los recursos de procesamiento de múltiples ordenadores para usarlos en resolver un único problema, ya sea a través de hardware dedicado o compartido. Lo que esto significa es que tu aplicación grid no está “atada” a tu ordenador, sino que puede usar más de un ordenador y otros recursos más allá de tus paredes para mejorar su rendimiento.

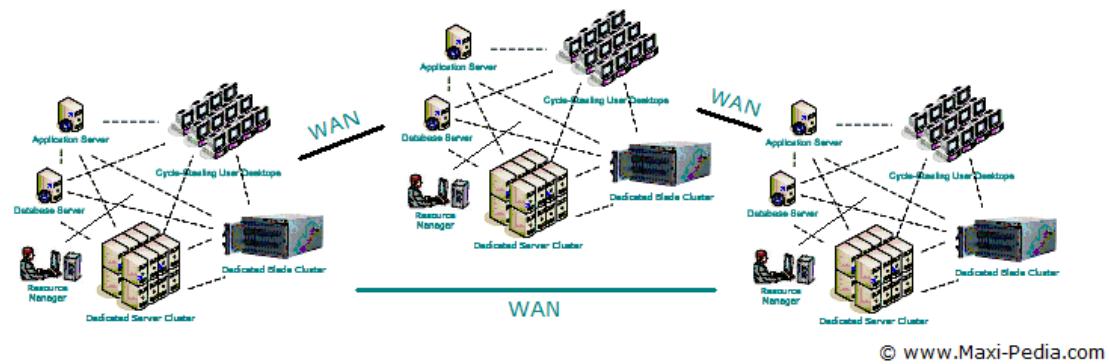


Figure 1.9: La computación grid no sólo emplea recursos sino sistemas enteros de varias localizaciones mientras cruzan límites geográficos y políticos

La clave para distinguir la computación grid y la computación distribuida está en la manera en la que se gestionan los recursos. La computación distribuida usa un gestor de recursos centralizado y todos los nodos trabajan juntos como un sistema unificado. La computación grid utiliza una estructura donde cada nodo tiene su propio gestor de recursos y el sistema no actúa como una sola unidad.

1.2.3 Computación distribuida a gran escala: Computación nube o cloud

La **computación cloud** comprende los aspectos relacionados con el desarrollo y ejecución de aplicaciones en un **sistema cloud**.

Sistema cloud

Ofrece **servicios de infraestructura** (Ej: una máquina), **plataforma** (Ej: una máquina con un sistema operativo y un software) y/o **software** (Ej: Dropbox), por los que se paga cuando se necesitan (*pay-per-use*) y a los que se accede típicamente a través de una **interfaz (web) de auto-servicio**.

Consta de **recursos virtuales** que:

- ♡ Son una *abstracción* de los recursos físicos.
- ♡ Parecen *ilimitados* en número y capacidad y son reclutados/liberados de forma inmediata sin interacción con el proveedor.
- ♡ Soportan el acceso de múltiples clientes (*multitenant*).
- ♡ Están conectados con métodos **estándar independientes** de la plataforma de acceso.

1.2.4 Criterios de clasificación de computadores

- ♡ **Comercial**: el *grado de accesibilidad* al hardware depende en gran medida de su precio de la accesibilidad del software. El precio no sólo depende del *coste de diseño* y del *coste de fabricación*, también del *volumen de venta* y de la *competencia*. El grado de accesibilidad al software depende de factores como el coste de desarrollo, volumen de venta y competencia.
- **Segmento de mercado**: Los sistemas con múltiples procesadores pueden encontrarse en los diferentes niveles del mercado de computadores.

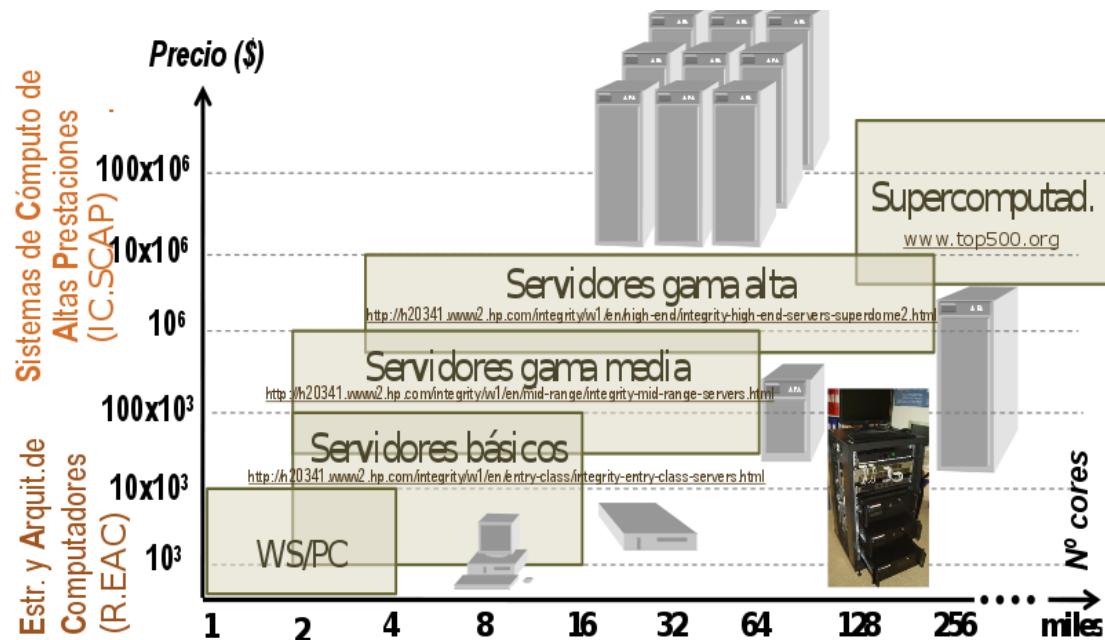


Figure 1.10: Clasificación del mercado de sistemas con múltiples procesadores

La clasificación ?? se ha hecho teniendo en cuenta la clasificación que hacen los propios vendedores con sus sistemas, el precio del sistema y la clasificación de servidores en función de su precio IDC (*International Data Corporation*). Conforme se baja de nivel disminuyen el precio y las prestaciones por unidad, y aumenta el volumen de ventas. El precio en un nivel del mercado depende del número de procesadores que incluye el sistema y del grado de utilización de componentes disponibles comercialmente ya que al tener mayor volumen de venta tienen el precio más bajo.

En el nivel más bajo están los sistemas empotrados, que se hacen para un propósito específico. Tienen restricciones de consumo de potencia, precio, tamaño y tiempo real, presentan dificultades para el programador pues tiene que programar eficientemente limitándose por un procesador más malo, un determinado consumo de potencia, etc para sacar partido máximo a la arquitectura. Después, los PC y estaciones de trabajo que se hacen para todo tipo de aplicaciones y generalmente tienen un procesador. El precio más alto de estos computadores es \$10.000. El precio más alto se corresponde con estaciones de trabajo SMP y el más bajo con PCs de baja gama.

En los siguientes niveles nos encontramos los **servidores**. IDC clasifica los servidores en tres grupos en función de su precio.

- servidores básicos (*entry-level* ó *volume servers*): incluyen aquellos con un precio inferior a \$25.000.
- servidores de gama media (*mid-range*): con precios entre \$25.000 y \$500.000
- servidores de gama alta (*high-end*): con precios mayores a \$500.000

Por último, se ha tomado como un grupo aparte de servidores de alta gama los **supercomputadores**. Esta denominación se asigna a los computadores que ofrecen mayores prestaciones, siendo también los de mayor coste y prestaciones. Dado su precio, tienen el grado de accesibilidad más bajo. El criterio utilizado para clasificar los supercomputadores son las prestaciones que ofrece el benchmark LINPACK³

♡ Educación, investigación (también usados por fabricantes y vendedores):

→ **Flujos de control y flujos de datos: clasificación de Flynn (1972)**: esta clasificación (Figura ??) divide los computadores en cuatro clases según el número de secuencias o flujos de instrucciones y secuencias o flujos de datos que pueden procesarse simultáneamente en el computador:

- **Computadores SISD**: un único flujo de instrucciones (SI, *Single Instruction*) procesa operandos y genera resultados, definiendo un único flujo de datos (SD, *Single Data*).
- **Computadores SIMD**: un único flujo de instrucciones (SI) procesa datos y genera resultados, definiendo varios flujos de datos (MD, *Multiple Data*), cada flujo de instrucciones ejecuta las mismas instrucciones pero las aplica sobre datos diferentes.
- **Computadores MIMD**: el computador ejecuta varias secuencias o flujos distintos de instrucciones (MI, *Multiple Instructions*), y cada uno de ellos procesa operandos y genera resultados definiendo un único flujo de instrucciones, de forma que también existen varios flujos de datos (MD) uno por cada flujo de instrucciones.
- **Computadores MISD**: se ejecutan varios flujos distintos de instrucciones (MI) aunque todos actúan sobre el mismo flujo de datos (SD).

³este benchmark ofrece una medida de la velocidad con la que la máquina ejecuta instrucciones de punto flotante.

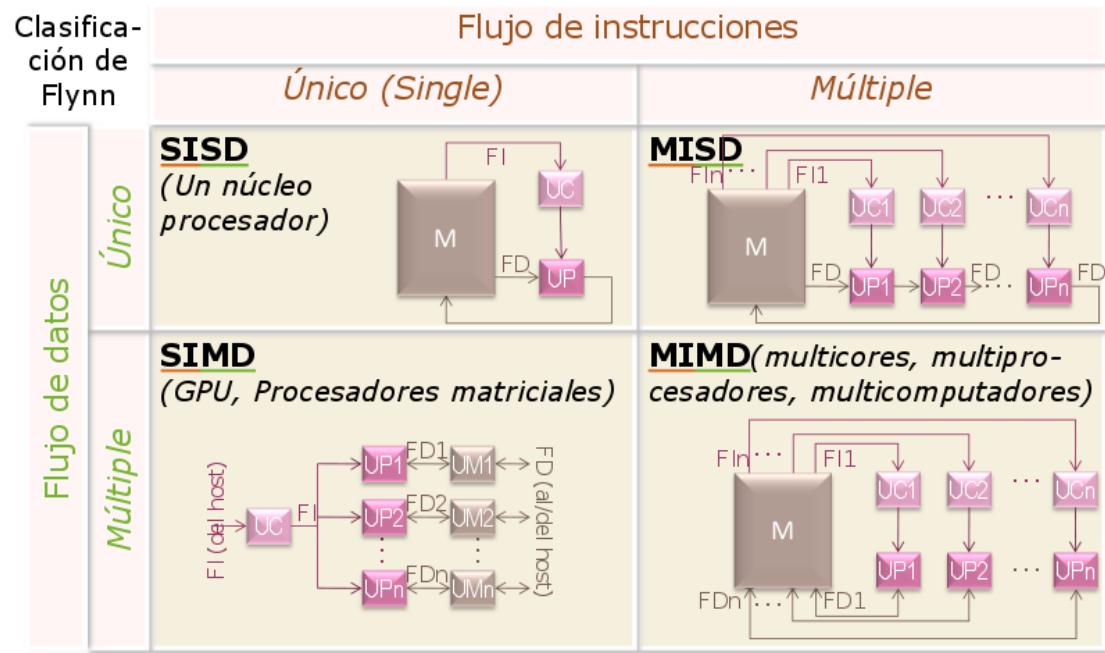


Figure 1.11: Clases de computadores en la taxonomía de Flynn: según el número de instrucciones y datos. M = memoria principal, UC = unidad de control, UP = unidad de procesamiento.

- ♡ **Sistema de memoria:** generalmente, los sistemas con múltiples procesadores o multiprocesadores se han clasificado atendiendo a la organización del sistema de memoria. Los sistemas de paralelismo de alto nivel se han clasificado en dos grupos:
 - **Sistemas con memoria compartida (SM, Shared Memory)** ó **multiprocesadores**: todos los procesadores comparten el mismo espacio de direcciones. El programador no necesita conocer dónde están almacenados los datos.
 - **Sistemas con memoria distribuida (DM Distributed Memory)** ó **multicomputador**: cada procesador tiene su propio espacio de direcciones particular. El programador necesita conocer dónde están almacenados los datos.

♡ **Flujos de control (propuesta de clasificación de arquitecturas con múltiples flujos de control):** cuando hablamos de flujos de control nos referimos a threads (Figura ??). Estas arquitecturas permiten paralelismo a nivel de thread sea real o concurrente y tiene dos categorías:

- **Implícito:** flujos de control gestionados por la arquitectura
- **Explícito:** flujos de control creados y gestionados por el sistema operativo, o bien con una instancia del sistema operativo (multiprocesadores, multicore, cores multithread...) ó con múltiples instancias del sistema operativo (multicomputadores).

♡ **Nivel de paralelismo aprovechado (propuesta de clasif.):** Esta clasificación se divide en:

- **Arquitecturas con Data Level Parallelism:** ejecutan las operaciones de instrucción concurrentemente o en paralelo. Son unidades funcionales vectoriales o SIMD.

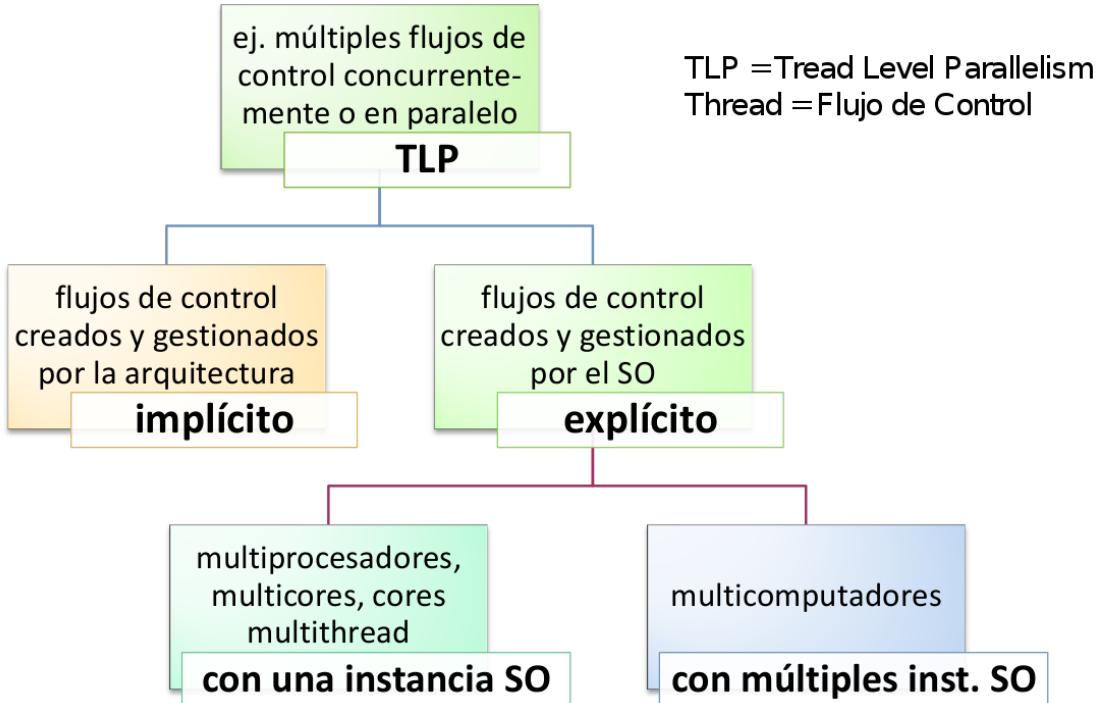


Figure 1.12: Clasificación de arquitecturas con múltiples threads

- **Arquitecturas con *Instruction Level Parallelism***: ejecutan múltiples instrucciones concurrentemente o en paralelo. Son los cores escalares segmentados, superescalares o VLIW/EPIC.
- **Arquitecturas con *Thread Level Parallelism explícito y una instancia del SO***: ejecutan múltiples flujos de control concurrentemente o en paralelo. Los hay de dos tipos
 - Cores que modifican la arquitectura escalar segmentada, superescalar o VLIW/EPIC para ejecutar threads concurrentemente o en paralelo
 - Multiprocesadores: ejecutan threads en paralelo en un computador con múltiples cores (incluye multicores).
- **Arquitectura con *Thread Level Parallelism explícito y múltiples instancias del SO***: ejecutan múltiples flujos de control en paralelo. Son los multiprocesadores que ejecutan threads en paralelo en un sistema con múltiples computadores.

1.2.5 Computadores SISD

En un computador (Figura ??) SISD existe una única unidad de control (UC) que recibe las instrucciones de memoria, las decodifica y genera los códigos que definen la operación correspondiente a cada instrucción que debe realizar la unidad de procesamiento (UP) de datos. El flujo de datos se establece a partir de los operandos necesarios para realizar la operación codificada en

cada instrucción, que se traen desde memoria, y de los resultados generados por las instrucciones que se almacenan en memoria. Se corresponde a computadores uniprocesador.

El siguiente código:

```
pascal for i:=1 to 4 do begin C[i]:=A[i]+B[i]; F[i]:=D[i]-E[i]; G[i]:=K[i]*H[i]; end;
```

Implica realizar doce operaciones, más las correspondientes al control del bucle. Una ejecución del código anterior en un SISD precisaría doce intervalos de tiempo.

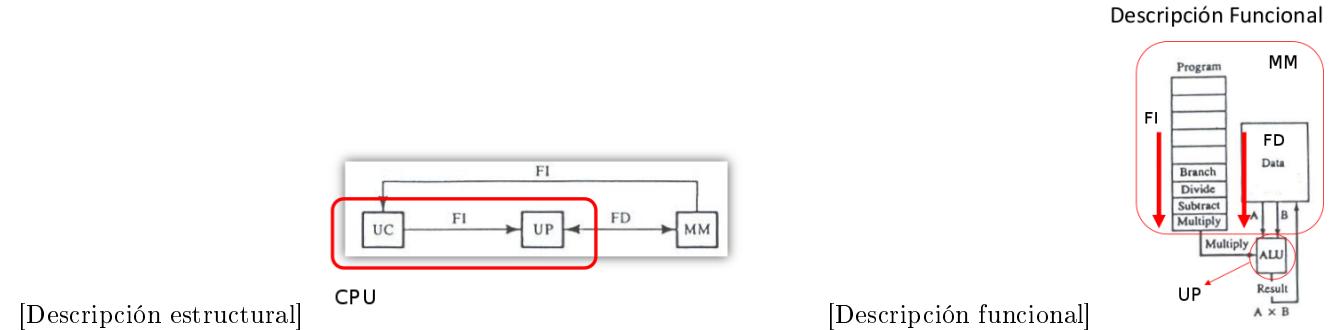


Figure 1.13: Descripción de un computador SISD

1.2.6 Computadores SIMD

En un computador SIMD (Figura ??) los códigos que genera la única unidad de control del computador a partir de cada instrucción actúan sobre varias unidades de procesamiento distintas (UP_i). Así, se pueden realizar varias operaciones similares simultáneas con operandos distintos. Cada una de las secuencias de operandos y resultados utilizados por las distintas unidades de proceso define un flujo de datos diferente. Aprovechan el **paralelismo de datos**.

El siguiente código: pascal for all EPi(i:=1 to 4) do begin C[i]:=A[i]+B[i]; F[i]:=D[i]-E[i]; G[i]:=K[i]*H[i]; end;

Correspondería a un procesador matricial con cuatro unidades de proceso (UP_i , $i = 1, \dots, 4$). En este caso, se necesitarían tres intervalos de tiempo. Es decir, cuatro veces menos que en el computador SISD, teniendo en cuenta que se están incluyendo cuatro unidades de proceso, tal y como se ve en la Figura ??

También tenemos procesadores vectoriales, que simulan el paralelismo ejecutando las instrucciones concurrentemente, es decir, todas las instrucciones usan el mismo recurso, no como en el matricial que hay una réplica del recurso y cada instrucción usa una. Los procesadores vectoriales se dividen en etapas (se podría decir que están segmentados) y en cada etapa del segmento se va calculando una suma diferente. El esquema de un procesador vectorial se representa en la Figura ??

1.2.7 Computadores MIMD

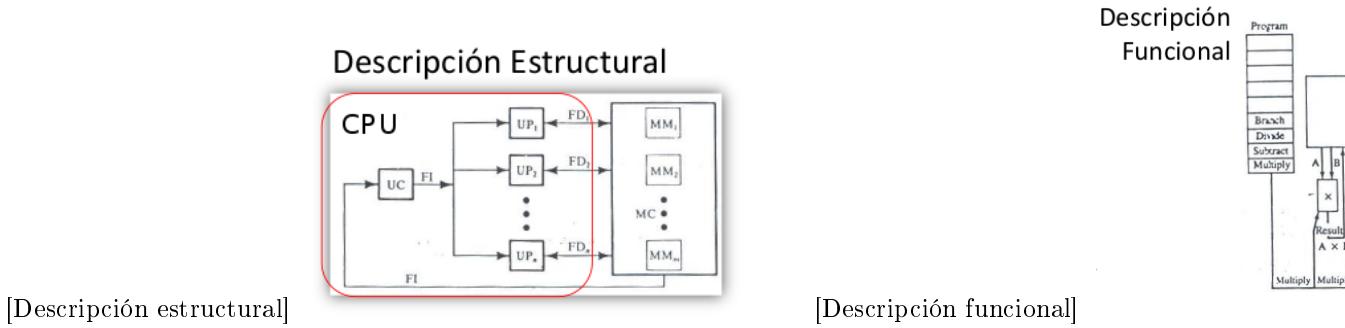


Figure 1.14: Descripción de un computador SIMD

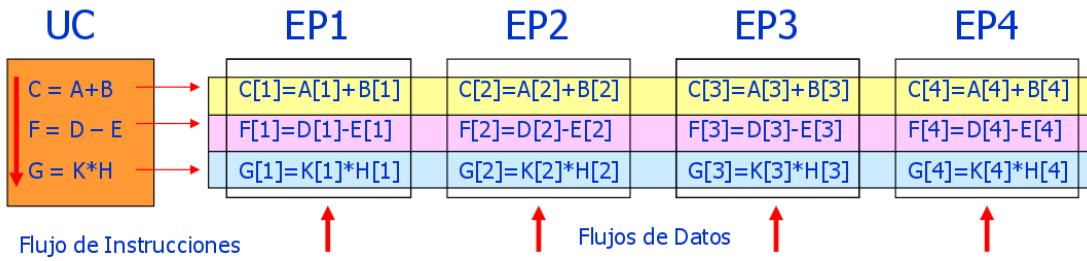


Figure 1.15: Procesador matricial

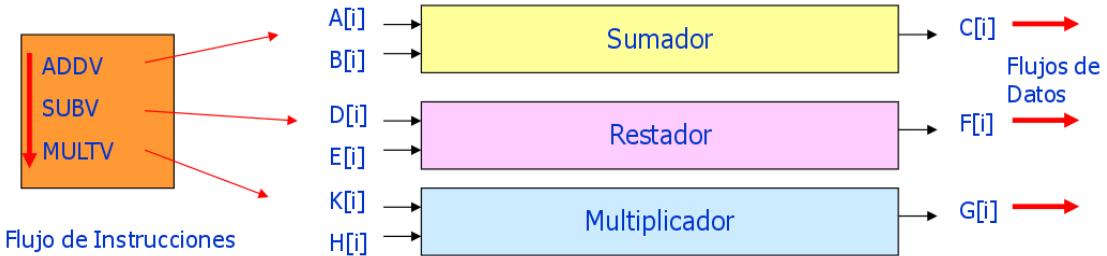


Figure 1.16: Procesador vectorial

En un computador MIMD (Figura ??) existen varias unidades de control que decodifican las instrucciones correspondientes a distintos programas. Cada uno de esos programas procesa conjuntos de datos diferentes, que definen distintos flujos de datos. Corresponde con multinúcleos, multiprocesadores y multicamputadores: puede aprovechar, además, **paralelismo funcional**.

En los computadores MIMD, para ejecutar los códigos anteriores se consideran tres procesadores (Proc1, Proc2 y Proc3) y cada uno ejecuta una operación distinta con los cuatro componentes de los distintos vectores: pascal for i:=1 to 4 do for i:=1 to 4 do for i:=1 to 4 do begin begin begin C[i]:=A[i]+B[i]; F[i]:=D[i]-E[i]; G[i]:=K[i]*H[i]; end; end; end; Proc 1 Proc 2 Proc 3

En este caso, el coste temporal sería de 4 unidades de tiempo: tres veces menos que en la implementación secuencial.

También se podría usar un MIMD con cuatro procesadores en el que cada uno realizaría las

tres operaciones con una de las cuatro componentes de los vectores. En este caso, el tiempo se reduciría a cuatro intervalos de tiempo, cuatro veces menos que en el caso de un SISD. En esta situación se dice que el computador MIMD utiliza una implementación SPMD (*Single Program Multiple Data*) del problema.

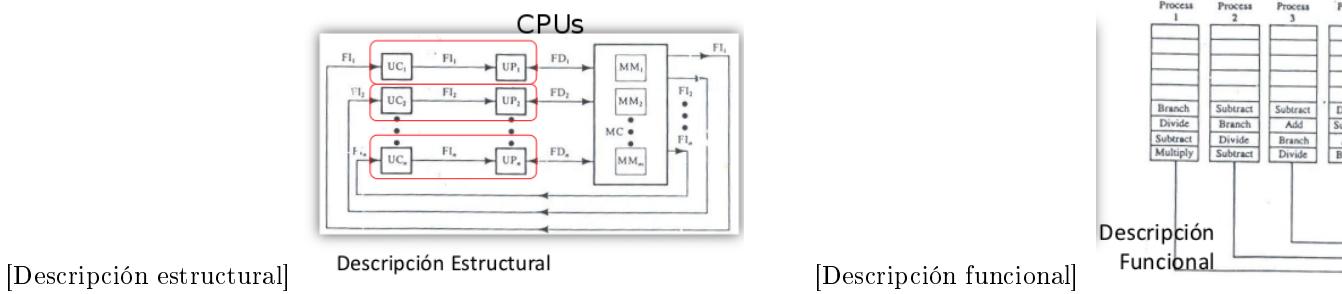


Figure 1.17: Descripción de un computador MIMD

1.2.8 Computadores MISD

Los computadores MISD (Figura ??) constituyen una clase de computadores cuyo comportamiento se puede implementar con iguales prestaciones en un computador MIMD en el que sus procesadores se sincronizan para que los datos vayan pasando desde un procesador a otro. Por eso, si bien se pueden identificar computadores de las demás clases (SISD, SIMD...) no existen computadores MISD específicos. Si la forma de procesamiento MISD es provechosa para algún problema, éste se puede implementar en un computador MIMD en el que los datos son procesados sucesivamente por los procesadores del sistema.

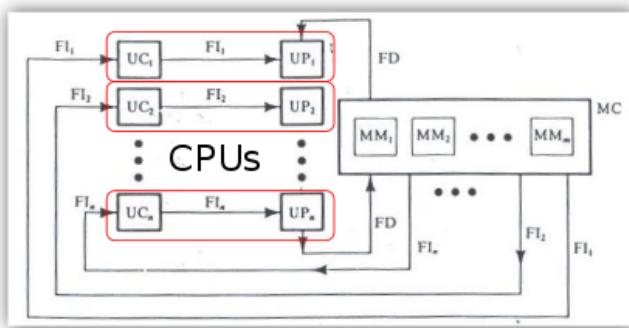


Figure 1.18: Descripción estructural de un computador MISD

1.2.9 Comparativa SMP (Symmetric Multi-Processor) y Multicomputadores y comunicación entre procesadores

En un multicomputador, la memoria asignada a un procesador se encuentra ubicada cerca del procesador. En un multiprocesador, toda la memoria se encuentra situada a igual distancia de todos los procesadores y físicamente enlazada.

En un multicomputador cada procesador tiene su propio espacio de direcciones, es por tanto lógico que en su estructura física, cerca de cada procesador haya un módulo de memoria local, en el que se encuentre su espacio de direcciones. También nos podemos encontrar distribuido el sistema de E/S. En un multiprocesador, todos los procesadores comparten el mismo espacio de direcciones por tanto, la estructura física para multiprocesadores tiene los módulos de memoria ubicados en la misma zona del sistema, separados de los procesadores por una red de conexión que arbitre el acceso a los módulos. También nos podemos encontrar centralizados los dispositivos de E/S.

Con esta estructura, el tiempo de acceso de los procesadores a memoria (se puede extender también a los dispositivos de E/S), será igual sea cual sea la posición de memoria a la que acceden, es una estructura simétrica. Los multiprocesadores con esta característica se denominan SMP (*Symmetric Multi-Processor*) o *multiprocesadores simétricos*. El acceso a memoria se realiza a través de la red de interconexión.

En multicomputadores, cada procesador tiene su propio módulo de memoria local al que puede acceder directamente. En esta configuración la red de interconexión se utiliza para la transferencia de datos compartidos (mensajes) entre nodos de la red. El tamaño de los mensajes a transferir depende del programador. En un multiprocesador, el tamaño de las transferencias con memoria depende del hardware, y será menor que el de las transferencias entre nodos en un multicomputador.

El comportamiento de ambos ante diferentes facetas es:

Latencia en el acceso a memoria : el tiempo de acceso a memoria es mayor en los multiprocesadores que en los multicomputadores. Esta mayor latencia se debe a:

- ♡ **La no localidad de los módulos de memoria**, en un multicomputador cada procesador tiene su espacio de direcciones en un módulo de memoria situado en su proximidad mientras que en multiprocesadores todos los módulos de memoria están separados de los procesadores.
- ♡ **La necesidad de acceder a memoria atravesando la red de interconexión**
- ♡ **El incremento en la latencia media debido a conflictos en la red entre accesos de diferentes procesadores**, ya que si varios procesadores en un momento dado necesitan utilizar el mismo recurso, tendrán que esperar a que quede libre y esto supone una penalización en la latencia de memoria para el procesador. Cuanto mayor sea el número de procesadores compitiendo en el acceso a memoria, la probabilidad de conflicto aumenta. En cambio, en un multicomputador como cada procesador tiene su propia memoria no se vería afectada la latencia en el acceso a memoria. Esto hace que los multiprocesadores sean poco escalables⁴

Mecanismos de comunicación : Al estar todos los procesadores colaborando en la ejecución de un programa, uno puede necesitar los datos que ha producido otro. En multiprocesadores esto se hace con un simple acceso a memoria (instrucciones **load** y **store**) ya que todos comparten la memoria. En multicomputadores se necesitan implementar primitivas

⁴Un sistema es escalable si al añadir más recursos al sistema éste incrementa sus prestaciones de forma proporcional al número de recursos utilizados. Idealmente, la productividad debería ser proporcional, y la latencia mantenerse constante.

software para copiar datos de un procesador a otro (instrucciones `send` y `receive`) lo cual es menos eficiente porque tenemos el mismo dato duplicado en el sistema.

Mecanismos de sincronización : Si un procesador A necesita un dato que produce B , debe esperar a que éste lo produzca. En multicomputadores estos mecanismos de sincronización se incluyen en las instrucciones de comunicación. En multiprocesadores, se usan implementaciones software para sincronizar (semáforos, monitores, cerrojos...). Los multiprocesadores proporcionan soporte hardware para incrementar las prestaciones en la implementación de primitivas software de sincronización.

Herramientas de programación : al tener los procesadores de los multicomputadores cada uno su memoria local, debemos ubicar en la memoria de cada procesador el código y los datos a ejecutar antes de ejecutar una aplicación en un multicomputador. Esto no es necesario en multiprocesadores. Además, las herramientas de programación deben ofrecer al programador herramientas para distribuir la carga de trabajo entre procesadores, lo ideal sería que esto lo hiciera el compilador pero no siempre se obtiene el mejor resultado. Estas herramientas deben ser más sofisticadas en multicomputadores que en multiprocesadores.

Programación : la programación de un multiprocesador es más sencilla que la de un multicomputador, ya que no debe pensar en la copia de datos entre nodos, ni en la asignación de trabajo a los procesadores. Los mecanismos de sincronización en multiprocesadores pueden dar lugar a situaciones de error en ejecución. La sincronización en multicomputadores es más fácil de entender debido a que está asociada a los mecanismos de comunicación. Si utilizamos el modo SPMD para programar multicomputadores, la distribución del código consiste en llevar el mismo programa a todos los nodos de procesamiento y habría que equilibrar únicamente los datos.

1.2.10 Incremento de escalabilidad en multiprocesadores y red de interconexión

- ♡ Incorporando *cachés* en el sistema de memoria, de forma que cada procesador disponga de una caché local. Debido a la localidad de instrucciones y datos, el número de accesos a memoria principal se reducirá, disminuyendo la latencia media y aumentando así la escalabilidad. Cuando se acceda a memoria principal, se transfiere al procesador una línea de caché completa en lugar de un único dato, aprovechando así en mayor medida el ancho de banda de la red. Se necesitan incluir mecanismos hardware que aborden la falta de coherencia y esto encarece el hardware y la escalabilidad.
- ♡ Utilizando *redes con menor latencia y mayor ancho de banda*. Para mejorar la latencia media y/o el ancho de banda en multiprocesadores manteniendo la simetría en el acceso a cualquier módulo de memoria principal, se puede usar:
 - En lugar de un bus, múltiples buses
 - Una red dinámica multietapa
 - Una red de barras cruzadas (ésta es la que mejor resultado da, ya que permite implementar cualquier aplicación biyectiva entre entradas y salidas.): permite a todos los procesadores acceder a memoria a todos los procesadores a la vez.

El único conflicto en el acceso a memoria es que dos o más procesadores quieran conectarse a la misma salida por lo que proporciona la menor probabilidad de colisión en el acceso a memoria.

- ♡ *Distribuyendo físicamente los módulos de memoria principal entre los procesadores*, se puede escalar hasta varios cientos y miles de procesadores, aunque perdiendo la propiedad de simetría. Así, cada procesador dispone de una serie de direcciones a las que podrá acceder con menor latencia. Acceder al espacio de direcciones de otro procesador costará mayor tiempo. Al perder la simetría adquiere importancia la distribución de la carga del trabajo. Estos multiprocesadores utilizan redes de interconexión con características comunes a las redes utilizadas en multicomputadores. Se denominan **multiprocesadores con memoria compartida distribuida** (DSM, *Distributed Shared Memory*).

1.2.11 Clasificación completa de computadores según el sistema de memoria

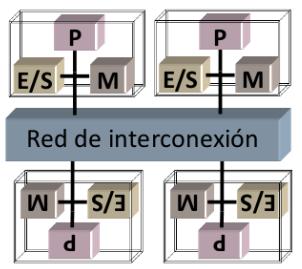
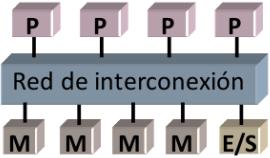
Multi-computadores Memoria no compartida	NORMA No Remote Memory Access	<i>e.g. cluster, red de computadores</i>	Memoria físicamente distribuida	+ +
Multi-procesadores Memoria compartida Un único espacio de direcciones	NUMA Non-Uniform Memory Access	NUMA		Nivel de empaquetamiento y conexión Escalabilidad
		CC-NUMA		
		COMA		
	UMA Uniform Memory Access	SMP Symmetric MultiProcessor	Memoria físicamente centralizada 	- -

Figure 1.19: Clasificación de sistemas con múltiples procesadores

A raíz de la aparición de multiprocesadores con memoria físicamente distribuida surgieron nuevas denominaciones para sistemas con múltiples procesadores. Los multiprocesadores se empezaron a clasificar entonces, en función de la uniformidad en el acceso a memoria, en dos grupos (Figura ??):

- ♡ **Multiprocesadores con acceso a memoria uniforme** o UMA (*Uniform Memory Access*): el tiempo de acceso de los procesadores a una determinada posición de memoria principal es

igual sea cual sea el procesador. El acceso a una posición de memoria caché es igual para todos los procesadores.

♡ **Multiprocesadores con acceso a memoria no uniforme** o NUMA (*Not-Uniform Memory Access*): el tiempo de acceso a una posición de memoria depende del procesador, ya que un procesador tardará menor tiempo en acceder al bloque de memoria local que al situado junto a otro procesador. Para que un NUMA sea escalable debe disminuirse la latencia media reduciendo el número de accesos de un procesador a posiciones de memoria situadas en bloques de memoria localizados en nodos remotos. Para ello necesitan distribuirse los códigos y datos de forma que en la memoria de un procesador estén los datos y código que utiliza. Si tenemos un hardware que evite incoherencias, podemos trasladar a la caché líneas de memoria y así aprovechar mejor el ancho de banda. Hay tres subgrupos:

- **NCC-NUMA** (*Non-Cache Non-Uniform Memory Access*): no incorporan hardware para evitar problemas por incoherencias entre cachés de distintos nodos. Esto hace que los datos modificables compartidos no se puedan trasladar a caché de nodos remotos, se debe acceder a ellos individualmente a través de la red. Para el programador es el más difícil de programar pues debe mantener la coherencia entre cachés él mismo a nivel software.
- **CC-NUMA** (*Cache-Coherent Non-Uniform Memory Access*): tienen hardware para mantener coherencia entre cachés de distintos nodos. El hardware añadido para mantenimiento de coherencia supone un coste añadido e introduce un retardo que hace que estos sistemas escalen en menor grado que un NUMA.
- **COMA** (*Cache Only Memory Access*): En estas arquitecturas la memoria local de los procesadores se gestiona como caché. El sistema de mantenimiento de coherencia se encarga de llevar dinámicamente el código y los datos a los nodos donde se necesiten. Permite *replicación*⁵ y *migración*⁶ de bloques de memoria en función de su frecuencia de uso en los nodos. Tienen mayor retardo hardware que los CC-NUMA. No existe actualmente ningún sistema comercial COMA.

1.2.12 Nota histórica

El paralelismo a nivel de datos surgió en los procesadores en los años 90.

El paralelismo a nivel de instrucción surgió en los procesadores en los años 80, aunque el primer prototipo es de los 60.

El paralelismo a nivel de thread explícito con una instancia del SO tuvo sus primer prototipo en 1975 y explícito con múltiples instancias del SO (multicomputadores), en 1985.

El objetivo de esta parte es ver cuando ha ido surgiendo cada cosa y tambien ver que a partir de los 2000 no ha surgido casi nada nuevo sino que se ha mejorado segun ha ido avanzando la tecnologia que ha ido surgiendo lo que ya habia.

1.3 Evaluación de prestaciones de una arquitectura

⁵Se prefiere cuando el bloque se usa en varios nodos

⁶Se prefiere cuando un bloque se usa en mayor medida por otro nodo diferente al nodo donde se encuentra

1.3.1 Medidas usuales para evaluar prestaciones

Para evaluar un sistema de computador necesitaremos ejecutar un programa que tenga unas entradas y devuelva unas salidas. Podemos evaluar distintas partes de un sistema computador como el procesador, el sistema de memoria, el sistema de comunicación entre distintos computadores (en este caso las entradas serían los mensajes a transferir). En resumen, para evaluar cualquier sistema, se utilizan siempre las mismas medidas: tiempo de respuesta (tiempo desde que se introducen las entradas al sistema hasta que se obtienen los resultados) y productividad (número de entradas procesadas por unidad de tiempo).

Tiempo de respuesta de un programa en una arquitectura

Si usamos el comando `time` en linux antes de ejecutar un programa obtenemos tres números: elapsed, user y time.

- ♡ **user**: se corresponde con el **tiempo de CPU de usuario**, es decir, el tiempo de ejecución en el espacio de usuario.
- ♡ **sys**: se corresponde con el **tiempo de CPU de sistema**, es decir, el tiempo en el nivel del kernel del sistema operativo.
- ♡ Y el tiempo asociado a las esperas devidas a I/O o asociados a la ejecución de otros programas.

```
bash time./program.exe
elapsed 5.4
user 3.2
sys 1.0
```

- ♡ **Real (wall-clock time, elapsed time, real time)**: tiempo que tarda en ejecutar todo el programa.
- ♡ **CPU time = user + sys**: tiempo que tardan en ejecutar el código correspondiente a llamadas al sistema y el código correspondiente al código del programa. El tiempo de ejecución real/elapsed también incluye esperas de E/S, que no forman parte del CPU time. Por eso el tiempo real es mayor o igual al de CPU.

Ahora bien, cuando tenemos una o varias hebras ejecutándose en el programa varían los resultados devueltos por `time`:

- ♡ **Con un flujo de control**: el tiempo que nos devuelve *elapsed* es mayor o igual al *CPU time*:

$$\text{elapsed} \geq \text{CPU time}$$
- ♡ **Con múltiples flujos de control**: esta vez tenemos que:

$$\begin{aligned} &\rightarrow \text{elapsed} < \text{CPU time} \\ &\rightarrow \text{elapsed} \geq \frac{\text{CPU time}}{\text{numero flujos control}} \end{aligned}$$

Ya que el elapsed será el thread que más tiempo tarde en ejecutarse y el CPU time será la suma de los tiempos de cpu de ambos thread.

En el ejemplo anterior, $3.2 + 1.0$ es el 78% del tiempo transcurrido (que en total ha sido 5.4).

Algunas alternativas para obtener tiempos de ejecución son:

Función	Fuente	Tipo	Resolución aprox (μs)
<code>time</code>	SO(/usr/bin/time)	elapsed, user, system	10000
<code>clock()</code>	SO (time.h)	CPU	10000
<code>gettimeofday()</code>	SO (sys/time.h)	elapsed	1
<code>clock_gettime() / clock_getres()</code>	SO (time.h)	elapsed	0.001
<code>omp_get_wtime() / omp_get_wtick()</code>	OpenMP (omp.h)	elapsed	0.001
<code>SYSTEM_CLOCK()</code>	Fortran	elapsed	1

`clock()` no obtiene el tiempo real. `clock_gettime()` y las funciones de OMP tienen muy buena precisión a la hora de calcular tiempo, a nivel de nanosegundos. Esta precisión depende de la plataforma.

Cuando el programa sea secuencial usaremos `clock_gettime()` y cuando sea paralelo, las de OMP.

Tiempo de CPU

Podemos calcular el tiempo de CPU que tendrá nuestro programa nosotros mismos, ya que el tiempo de CPU depende de número de instrucciones del programa y el tiempo medio por instrucción. El número de instrucciones es fácil de obtener y el tiempo medio por instrucción, se puede obtener como el número medio de ciclos por instrucción por el tiempo de cada ciclo, y éstos datos se pueden calcular dinámicamente.

$$T_{tarea} = NI \cdot t_{medio_por_instr} = NI \cdot (CPI \cdot T_{ciclo}) = NI \cdot \left(\frac{CPI}{f} \right) \quad (1.1)$$

donde T_{tarea} es el tiempo de CPU de un programa (también se denota como T_{CPU}), NI es el número de instrucciones máquina del programa, CPI es el número medio de ciclos por instrucción y T_{ciclo} , el periodo de reloj del procesador (inverso de la frecuencia del procesador, f). El valor de CPI se puede expresar como:

$$CPI = \frac{\text{Ciclos de reloj programa}}{NI} = \frac{\sum_{i=1}^n NI_i \cdot CPI_i}{NI} \quad (1.2)$$

donde CPI_i es el número medio de ciclos de las instrucciones de tipo i (uno de los n tipos de instrucciones, ya que $i = 1, \dots, n$), y NI_i es el número de instrucciones de ese tipo.

De esto, deducimos por tanto que el tiempo de CPU de un programa es:

$$T_{CPU} = \text{Ciclos del programa} \cdot T_{ciclo} = \frac{\text{Ciclos del programa}}{\text{Frecuencia de reloj}} \quad (1.3)$$

- ♡ NI depende del *repertorio de instrucciones* y del *compilador*
- ♡ CPI viene determinado por el *repertorio de instrucciones* y la *organización del computador*
- ♡ T_{ciclo} depende de las prestaciones que proporcione la *tecnología* y de la *organización del computador*.

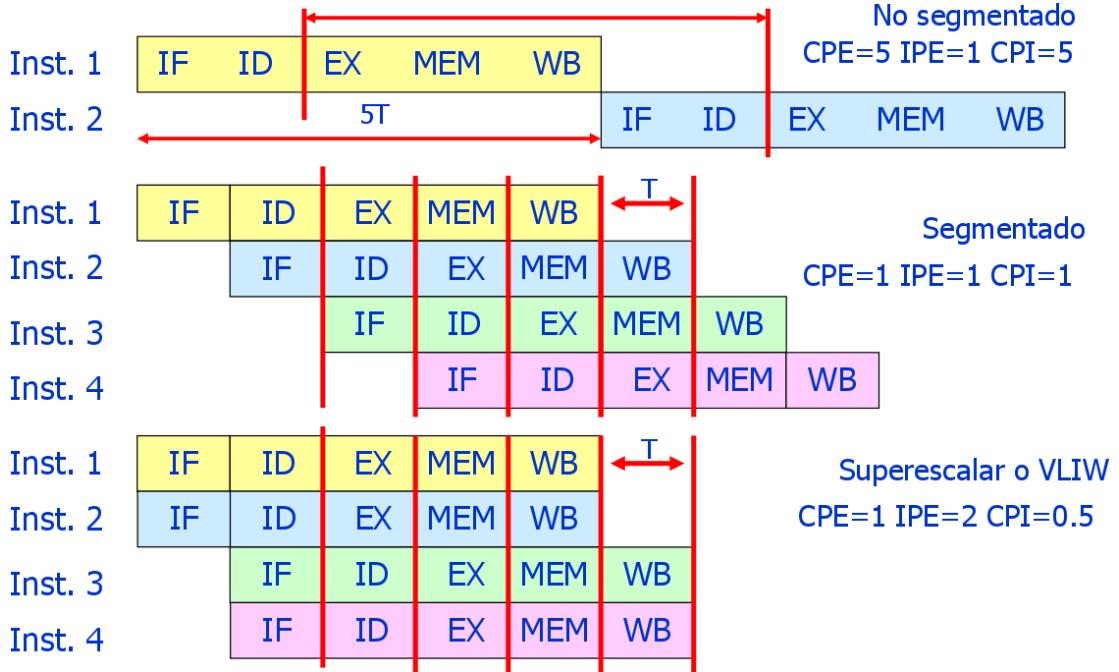


Figure 1.20: Ejemplos de valores característicos de CPE, IPE y CPI según las arquitecturas segmentadas, superescalares y VLIW

El valor de CPI se puede mejorar con procesadores superescalares ejecutando varias instrucciones en paralelo.

Al afectar la organización del computador tanto a *CPI* como a f , la métrica importante es el cociente $\frac{CPI}{f}$: aunque se utilicen frecuencias mayores, si se produce un incremento del mismo orden en CPI, las prestaciones no aumentarían.

La expresión (??) para T_{tarea} se puede utilizar para reflejar características de los procesadores en relación con el aprovechamiento del paralelismo entre instrucciones:

$$T_{CPU} = NI \cdot \underbrace{\left(\frac{CPE}{IPE} \right)}_{CPI} \cdot T_{ciclo} \quad (1.4)$$

donde *CPE* es el número medio de ciclos entre inicios de ejecución de instrucciones, es decir, ciclos entre *emisión de instrucciones* a ejecutar (*Ciclos Por Emisión*); e *IPE*, el número medio de instrucciones que se emiten (*Instrucciones Por Emisión*). El cociente entre ambos no es más que otra forma de contar el número de ciclos por instrucción.

Así, como se muestra en la Figura ??, en el caso de:

- ♡ Un procesador *no segmentado* en el que todas las instrucciones tienen igual duración (por ej: $CPI = 5$), el número de ciclos entre etapas de emisión es igual a $CPE = CPI$ (en este caso $CPE = 5$). Como cada vez se emite sólo una instrucción, $IPE = 1$. Tardaríamos un tiempo $5T$ en ejecutar una instrucción.

- ♡ *Un procesador segmentado* a pleno rendimiento se emitiría una instrucción por ciclo en cada ciclo de reloj, por lo que $CPE = 1$ e $IPE = 1$, teniéndose que $CPI = 1$ (en cada ciclo se termina de ejecutar una instrucción). El tiempo de ejecución sería igual al tiempo de ejecución de una instrucción entre el número de etapas en las que se ha segmentado.
- ♡ En *los procesadores superescalares* se podría emitir más de una instrucción por ciclo en cada ciclo. En el ejemplo de la Figura ?? se emiten dos instrucciones por ciclo, por lo que $CPE = 1$ e $IPE = 2$, por lo que $CPI = 0.5$.

También podemos recoger en la ecuación (??) las características del repertorio de instrucciones según las posibilidades de codificación que presenta, el número de instrucciones, NI , se puede expresar en términos del número de operaciones que realiza el programa, $Noper$, y del número medio de operaciones que puede codificar una instrucción, Op_instr :

$$T_{CPU} = \left(\frac{Noper}{Op_instr} \right) \cdot CPI \cdot T_{ciclo} \quad (1.5)$$

Así, si consideramos el ejemplo de procesamiento SIMD del procesador vectorial y el procesador matricial, el número de instrucciones que se ejecutan es $Noper = 12$. Como existen cuatro elementos de procesamiento que trabajan simultáneamente, cada instrucción codifica cuatro operaciones, $Op_instr = 4$, y el programa a ejecutar consta de tres instrucciones del repertorio del procesador matricial ($NI = 3$).

Productividad: MIPS y MFLOPS

Los **MIPS** (*Millones de Instrucciones por Segundo*) y los **MFLOPS** (*Millones de Operaciones en Coma Flotante por Segundo*) son medidas que pretenden caracterizar el nivel de prestaciones de un computador y facilitar la comparación entre distintos sistemas a través de una única cantidad.

La definición de los MIPS en términos de T_{CPU} se proporciona en la siguiente expresión:

$$MIPS = \frac{NI}{T_{CPU} \cdot 10^6} = \frac{f}{CPI \cdot 10^6} \quad (1.6)$$

Esta medida puede variar según el programa, por lo que no sirve como medida característica de una máquina. Además, *depende del repertorio de instrucciones*, por lo que tampoco permite comparar máquinas con repertorios distintos y, puede ser inversamente proporcional a las prestaciones (valores mayores de MIPS corresponden a peores prestaciones), ya que los MIPS miden la velocidad de ejecución de las instrucciones. Por ejemplo, si dos programas tardan lo mismo pero uno utiliza más instrucciones que el otro (porque uno utiliza repertorio CISC y el otro, el repertorio RISC) el que utiliza más instrucciones tendrá más MIPS y erróneamente diremos que proporciona más prestaciones.

Los MFLOPS plantean una problemática similar a los MIPS y su fórmula es:

$$MFLOPS = \frac{\text{Operaciones en coma flotante}}{T_{CPU} \cdot 10^6} \quad (1.7)$$

No es una medida adecuada para todos los programas ya que sólo tiene en cuenta las operaciones en coma flotante. Además, ni el conjunto de instrucciones en coma flotante es el mismo para

todas las máquinas ni tampoco el coste de dichas instrucciones (la precisión puede ser diferente, pueden no consumir el mismo número de ciclos, etc).

Para paliar este último problema, se utilizan los **MFLOPS normalizados**, que se obtienen dando un peso relativo a cada instrucción para poner de manifiesto su mayor coste con respecto a la instrucción en coma flotante con menor coste.

1.3.2 Conjunto de programas de prueba (Benchmark)

Las medidas de prestaciones que realicemos o que nos ofrezcan terceros deben cumplir las siguientes propiedades:

- ♡ **Fiabilidad:** las medidas deben obtenerse procesando un conjunto de entradas que sea representativo del que se pueden esperar en el funcionamiento habitual del sistema. También, para que sean fiables, los valores de las medidas deberían ser reproducibles.
- ♡ **Permitir comparar diferentes realizaciones de un sistema o diferentes sistemas:** para que se puedan comparar diferentes realizaciones de un sistema o diferentes sistemas, sobre la base de medidas de tiempo de respuesta y productividad, éstas se deben tomar con las mismas secuencias de entradas en las diferentes implementaciones. Por lo que se necesitan entradas por los interesados, tanto fabricantes e investigadores como vendedores o usuarios.

Tipos de Benchmarks

- ♡ **De bajo nivel o microbenchmark:** evalúan las prestaciones de distintos aspectos de la arquitectura o del software: el procesador, la memoria, etc. Un ejemplo de microbenchmark para redes de interconexión es el test ping-pong.
- ♡ **Kernels:** son trozos de código muy utilizados en diferentes aplicaciones. Por ejemplo: multiplicación de matrices, resolución de sistemas de ecuaciones, producto escalar... Junto con los microbenchmark, son útiles para encontrar los puntos fuertes de cada máquina y así poder explicar las diferencias en prestaciones entre diferentes máquinas.
- ♡ **Programas sintéticos:** Son también trozos de código, pero no pretenden obtener un resultado con significado. Es la peor opción, ya que no se utilizan en aplicaciones reales. Ejemplos: Dhystone ó Whetstone.
- ♡ **Programas reales:** Son programas disponibles comercialmente o gratuitos utilizados en el sector de computadores que pretenden evaluar, como por ejemplo, bases de datos, servidores web, etc.
- ♡ **Aplicaciones diseñadas:** Se diseñan aplicaciones que pretenden representar a aquellas para las que se utiliza el computador. Resultan útiles cuando no existen para estas aplicaciones programas disponibles. Cuanto más se acerquen a *aplicaciones reales*, mayor será la fiabilidad que ofrecen.

Benchmark suites

Las *benchmark suites* son un conjunto de programas de prueba promovidos por ciertas instituciones en los que se suplen las limitaciones de los benchmark con la presencia de otros. Estos conjuntos cambian periódicamente adaptándose a los nuevos avances que van surgiendo, mejorar sus resultados, etc. Las benchmark suites más destacadas son:

SPEC CPU2006 : se usa para la evaluación de operaciones con enteros (CINT2006) y con punto flotante (CFP2006) en un core. Es un benchmark del tipo aplicaciones reales:

- ♡ **CINT2006**: compilador gcc, compresor bzip2, planificación de vehículos de transporte, inteligencia artificial, compresión de vídeo...
- ♡ **CFP2006**: dinámica de fluidos, dinámica molecular, reconocimiento de voz...

Las herramientas que usa son C, C++ y Fortran.

SPEC OMP 2001 (SPEC OpenMP) : es un benchmark paralelo. Con este benchmark se evalúan prestaciones de aplicaciones para SMP programas con OpenMP. Permite evaluar el procesador, el sistema de memoria, el sistema operativo y el compilador. Está basado en el benchmark SPEC CPU2000. Ofrece los códigos fuente para poder analizar las prestaciones del compilador con diferentes optimizaciones. Se usa en el ámbito científico y su estilo es de variables compartidas.

SPEC HPC2002 : está basado en aplicaciones de altas prestaciones y tiene como objetivo la evaluación de computadores de altas prestaciones. Permite evaluar: los procesadores, el sistema de interconexión, los compiladores, la implementación de OpenMP y MPI, y el sistema de E/S. Su área de aplicación es científica. Su estilo se basa en variables compartidas, paso de mensajes y combinación de ambos.

TPC (Transaction Processing Performance Council) : El grupo TPC ha desarrollado un conjunto de benchmark que evalúa diferentes aplicaciones comerciales de servidores:

- ♡ **TPC-C** analiza procesamiento de transacciones OLTP.
- ♡ **TPC-H** y **TPC-R** evalúan cargas de trabajo de sistema de soporte de decisiones (DSS). TPC-H soporta modificaciones concurrentes de datos y peticiones improvisadas orientadas al comercio y a la industria. TPC-R permite optimizaciones adicionales basadas en el conocimiento de las peticiones.
- ♡ **TPC-W** evalúa aplicaciones de comercio electrónico, genera transacciones orientadas al comercio.
- ♡ **TPC-App** evalúa servidores web y de aplicaciones.

Los benchmark TPC son entradas software comercial (bases de datos, servidores de información de Internet) y carga de trabajo diseñada.

NPB2, NPB3 (NAS Parallel Benchmark) : son benchmark paralelos. Ambos constan de cinco núcleos y tres pseudo-aplicaciones diseñadas. NPB2 está implementado con MPI. NPB3 consta de implementaciones de OpenMP, Java y HPF. Se usa en el ámbito científico. Su estilo es paso de mensajes y variables compartidas.

Implementaciones de la biblioteca BLAS (Basic Linear Algebra Subprograms) : evalúan cores con operaciones del álgebra lineal. Incluye operaciones con:

- ♡ vectores, como producto escalar o AXPY (*Alpha X Plus Y*)
- ♡ vector-matriz, como producto matriz por vector
- ♡ matriz-matriz como producto de matrices (*GEMM-GEneral Matrix Multiply*)

Hay implementaciones con diferentes herramientas de programación y optimizadas para diferentes arquitecturas.

LINPACK : se utiliza para clasificar a los computadores en la lista top500 .el núcleo de este programa es una rutina denominada DAXPY (*Double precision-real Alpha X Plus Y*) que multiplica un vector por una constante y los suma a otro vector. Las prestaciones obtenidas se escalan con el valor del tamaño del vector: c++ for (i=0; i<N; i++) y[i] = alpha*x[i] + y[i];

1.3.3 Ganancia en prestaciones

Ganancia en velocidad ó Speed-up

Si en un computador se incrementan las prestaciones de un recurso haciendo que su velocidad sea *p veces mayor* (por ejemplo: *p procesadores* en lugar de uno), el incremento de velocidad que se consigue en la nueva situación con respecto a la previa (*máquina base*) se expresa mediante la **ganancia de velocidad** ó *speed-up*, S_p :

$$S_p = \frac{V_p}{V_1} = \frac{T_1}{T_p}$$

Siendo:

- ♡ V_1 la velocidad de la máquina base
- ♡ V_p la velocidad de la máquina mejorada (un factor p en uno de sus componentes)
- ♡ T_1 tiempo de ejecución en la máquina base
- ♡ T_p tiempo de ejecución en la máquina mejorada

Ley de Amdahl

La mejora de la velocidad, S , que se puede obtener cuando se mejora un recurso de una máquina en un factor p (o k según la Figura ??) está limitada por:

$$S \leq \frac{p}{1 + f(p - 1)}$$

donde f es la fracción del sistema que utiliza la mejora, y p indica en cuanto se consigue mejorar esa fracción.

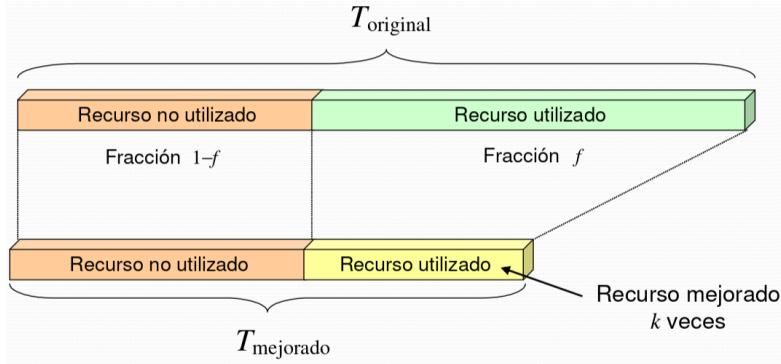


Figure 1.21: Tiempo original vs tiempo mejorado

Por ejemplo: si un programa pasa un 25% de su tiempo de ejecución en una máquina realizando instrucciones de coma flotante, y se mejora la máquina haciendo que estas instrucciones se ejecuten en la mitad de tiempo, entonces $p = 2$, $f = 0.75$ y:

$$S \leq \frac{2}{1 + 0.75} = 1.14$$

Hay que mejorar el caso más frecuente (lo que más se usa)

Para deducir esta ecuación debemos partir del *speedup* obtenido al cambiar el componente.

El tiempo sin mejorar sería:

$$T_{original} = f \cdot T_{original} + (1 - f) \cdot T_{original}$$

y el tiempo mejorado al aplicar el factor p sería:

$$T_{mejorado} = f \cdot T_{original} + \frac{(1 - f) \cdot T_{original}}{p}$$

Y el *speedup* obtenido sería:

$$S = \frac{T_{original}}{T_{mejorado}} = \frac{T_{original}}{f \cdot T_{original} + \frac{(1 - f) \cdot T_{original}}{p}} = \frac{T_{original}}{T_{original}} \cdot \left(\frac{1}{f + \frac{1 - f}{p}} \right) = \frac{1}{f + \frac{1 - f}{p}} = \frac{p}{1 + f \cdot (p - 1)}$$

Otra forma de ver esta ley es la siguiente: tenemos un componente que, para un determinado programa, se utiliza una fracción f de tiempo, ¿cómo mejora el tiempo de ejecución del programa al cambiar dicho componente por otro p veces más rápido?

$$T_{mejorado} = (1 - f) \cdot T_{original} + \frac{f \cdot T_{original}}{p}$$

Para deducirla, hacemos el *speedup* como antes:

$$S = \frac{T_{original}}{T_{mejorado}} = \frac{T_{original}}{(1-f) \cdot T_{original} + \frac{f \cdot T_{original}}{p}} = \frac{1}{1-f+\frac{f}{p}}$$

La mejora sólo se verá afectada por la fracción de tiempo que utiliza dicho recurso, lo demás no varía (Figura ??). Por tanto, cuando $f = 0$ no hay ninguna mejora y cuando $f = 1$ el tiempo de ejecución mejora k veces, igual que el componente.

En relación con la eficacia de los computadores paralelos, la ley de Amdahl dice que dado que en un programa hay código secuencial que no puede parallelizarse, los procesadores no se podrían utilizar eficazmente.

1.4 Ejercicios

1.4.1 Ejercicio 3

Resumen de los datos del enunciado

I_i	CPI_i^1	NI_i^1	CPI_i^2	NI_i^2
CMP	3	$0.2NI^1$	-	-
BR	4	$0.2NI^1$	-	-
CMP-BR	-	-	4	$0.2NI^1$
Resto	3	$0.6NI^1$	3	$0.6NI^1$

$$T_c^1 = T_c^2 - 0.25 \cdot T_c^2 = 0.75 \cdot T_c^2$$

El número de instrucciones de salto condicional en la alternativa 1 coincide con el número de instrucciones CMP+BR porque ejecutamos los mismos programas con ambas alternativas.

En el resto teníamos incluida las instrucciones CMP, como las hemos puesto aparte, en el resto tenemos $0.6NI^1$

Comparativa entre las dos alternativas

$$T_{CPU}^1 = \underbrace{(3 \cdot 0.8NI^1)}_{Resto+CMP} + \underbrace{4 \cdot 0.2NI^1}_{BR} \cdot T_c = 3.2 \cdot NI^1 \cdot T_c^1$$

$$T_{CPU}^2 = \underbrace{(3 \cdot 0.6NI^1)}_{Resto} + \underbrace{(4 \cdot 0.2NI^1)}_{CMP-BR} \cdot T_c^2 = 2.6 \cdot NI^1 \cdot T_c^2$$

Cogiendo la fórmula que hemos sacado a partir de los datos del enunciado, sustituimos en el tiempo de ciclo de la alternativa 1:

$$T_{CPU}^1 = \underbrace{((3 \cdot 0.8NI^1) + 4 \cdot 0.2NI^1)}_{Resto+CM\!P} \cdot T_c = 3.2 \cdot NI^1 \cdot T_c^1 = 3.2 \cdot NI^1 \cdot \underbrace{0.75 \cdot T_c^2}_{T_c^1} = 2.4 \cdot NI^1 \cdot T_c^2$$

Y concluimos que la alternativa 2 no merece la pena, pues aunque tenemos menos instrucciones, el tiempo de ciclo es mayor.

1.4.2 Ejercicio 4

Ahora tenemos este tiempo de ciclo para la alternativa dos:

$$T_c^2 = 1.1 \cdot T_c^1$$

Si sustituimos en el tiempo de cpu de la alternativa dos obtenemos:

$$T_{CPU}^2 = \underbrace{((3 \cdot 0.6NI^1) + (4 \cdot 0.2NI^1))}_{Resto} \cdot T_c^2 = 2.6 \cdot NI^1 \cdot T_c^2 = 2.6 \cdot NI^1 \cdot 1.1 \cdot T_c^1 = 2.86 \cdot NI^1 \cdot T_c^1$$

En este caso si merecería la pena.

EXTRA: calcular los MIPS de cada alternativa:

$$MIPS^1 = \frac{NI^1}{3.2 \cdot NI^1 \cdot T_c^1 \cdot 10^6}$$

$$MIPS^2 = \frac{NI^2}{2.86 \cdot NI^1 \cdot T_c^1 \cdot 10^6} = \frac{0.8NI^1}{2.86 \cdot NI^1 \cdot T_c^1 \cdot 10^6} = \frac{1}{3.575 \cdot T_c^1 \cdot 10^6}$$

La alternativa mejor según los MIPS sería la uno, lo cual es erróneo porque como hemos visto, la mejor es la dos.

Chapter 2

Programación paralela

2.4 Herramientas, estilos y estructuras en programación paralela

2.4.1 Problemas que plantea la programación paralela al programador. Punto de partida

Problemas que plantea la programación paralela

Con la programación paralela, nos surgen nuevos problemas que no teníamos en programación secuencial:

- ♡ **División en unidades de cómputo independientes (tareas)**: el programador debe dividir su código en unidades independientes unas de las otras, llamadas tareas.
- ♡ **Agrupación/asignación de tareas o carga de trabajo (código, datos) en procesos/threads**: después, debe asignar dichas tareas a distintos threads para que se ejecuten en paralelo.
- ♡ **Asignación a procesadores/núcleos**: para conseguir paralelismo real el programador debe asignar cada thread a un procesador/núcleo diferente porque si asigna todo el trabajo a un único thread no tendríamos programación paralela sino concurrente.
- ♡ **Sincronización y comunicación**: dichas tareas seguramente comparten datos, por tanto, deben estar sincronizadas para que el resultado de la ejecución sea correcto.

Estos nuevos problemas debe abordarlos la herramienta de programación (el compilador), el programador o ambos.

Punto de partida

Cuando se plantea obtener una versión paralela de una aplicación, se puede utilizar como punto de partida:

- ♡ Un **código secuencial** que resuelva el problema y, sobre este código de alto nivel buscar la paralelización. La versión paralela va a depender de la descripción del problema que se ha utilizado en la versión secuencial de partida. Podría ocurrir que el programa secuencial no adminta una paralelización inmediata. Una ventaja de esta opción es que facilita el reparto del trabajo equitativo entre procesadores.
- ♡ Otra posibilidad, sería partir de la **definición de la aplicación**. Habría que buscar una descripción para el problema que admita paralelización, incluso se podrían exportar varias descripciones alternativas.

Para facilitar el trabajo podemos apoyarnos en programas paralelos que aprovechen las características de la arquitectura y en bibliotecas de funciones paralelas. Hay diversas bibliotecas paralelas que proporcionan funciones que se utilizan en aplicaciones que requieren altas prestaciones como:

- ♡ Un programa paralelo que resuelva un problema parecido.
- ♡ Versiones paralelas u optimizadas de bibliotecas de funciones como pueden ser BLAS (*Basic Linear Algebra Subroutine*) o LAPACK (*Linear Algebra PACKAGE*)

2.4.2 Modos de programación

SPMD (Single-Program Multiple Data)

En el modo **SPMD** (*Single Program Multiple Data*), todos los códigos que se ejecutan en paralelo se obtienen compilando el mismo programa. Cada copia trabaja con un conjunto de datos distintos, y se ejecuta en un procesador diferente (Figura ??).

En la práctica, tanto en multiprocesadores como en multicomputadores, se aplica en mayor medida el modo SPMD.

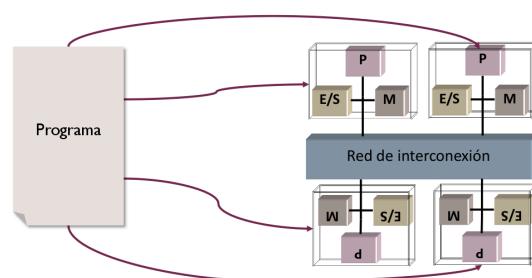


Figure 2.1: Programación SPMD

MPMD (Multiple Programs Multiple Data)

En el modo **MPMD** (*Multiple Programs Multiple Data*), los códigos que se ejecutan en paralelo se obtienen compilando programas independientes. En este caso la aplicación a ejecutar se divide en unidades independientes. Cada unidad trabaja con un conjunto de datos y se asigna a un procesador distinto (Figura ??).

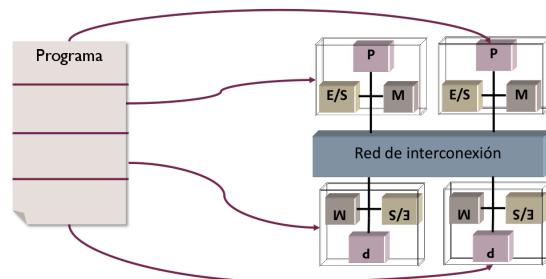


Figure 2.2: Programación MPMD

2.4.3 Herramientas para obtener código paralelo

Las herramientas para obtener programas paralelos deberían permitir de **forma explícita** (el trabajo lo haría el *programador*) o **implícita** (el trabajo lo haría la *propia herramienta*) la realización de las siguientes tareas:

- ♥ Localizar paralelismo
- ♥ Distribuir la carga de trabajo entre procesos
- ♥ Crear y terminar procesos
- ♥ Comunicación y sincronización entre procesos

La asignación de la carga de trabajo a procesadores puede quedar implícita en la asignación de ésta a procesos (la elección se deja al sistema operativo).

Hay varias alternativas para obtener un programa paralelo, la que requiere una mayor implicación del programador está en el nivel de abstracción más bajo (Figura ??).

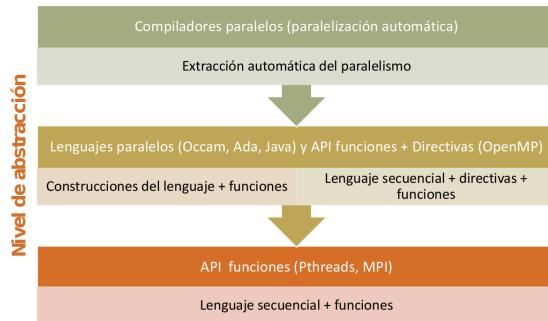


Figure 2.3: Herramientas de programación paralela clasificadas por su nivel de abstracción

Bibliotecas de funciones para programación paralela

El programador utiliza un lenguaje secuencial y una biblioteca de funciones. El cuerpo de los procesos y hebras se escribe con lenguaje secuencial. El programador explícitamente se encaga de:

- ♥ Distribuir las tareas entre los procesos usando el lenguaje secuencial
- ♥ Crear o gestionar procesos e implementar la comunicación y sincronización, utilizando funciones de la biblioteca
- ♥ Puede elegir la distribución de los procesos entre procesadores

Las ventajas de esta alternativa son:

- ♥ Los programadores están familiarizados con los lenguajes secuenciales
- ♥ Las bibliotecas están disponibles para todos los sistemas paralelos
- ♥ Las bibliotecas están más cercanas al hardware y dan al programador un control a más bajo nivel
- ♥ Se pueden utilizar a la vez bibliotecas para programar hebras y bibliotecas para programar procesos.

Por ejemplo: PThread y MPI. Esta última incluye directivas, el uso de éstas sitúa al programador en un nivel de abstracción superior.

```
Un ejemplo para calcular el número pi usando MPI sería: [linenos]c include <mpi.h> int main
(int argc, char **argv) double ancho, x, sum, lsum; int intervalos, i, nproc, iproc;
if (MPIInit(argc, argv)! = MPI_SUCCESS) exit(1); //Enrolar MPIComm_size(MPI_COMM_WORLD, nproc); MPI
atoi(argv[1]); ancho = 1.0/(double)intervalos; lsum = 0;
// Localizar y asignar for (i = iproc; i<intervalos; i+=nproc) x = (i+0.5)*ancho; lsum += 4.0/(1.0+x*x);

lsum*=ancho;
// Comunicar/sincronizar MPIReduce(lsum, sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
// Desenrolar MPIFinalize();
```

Lenguajes paralelos y directivas del compilador

El uso de lenguajes paralelos o lenguajes secuenciales con directivas sitúan al programador en un nivel de abstracción superior, facilitando un poco el trabajo, aunque puede que sólo aprovechen un tipo de paralelismo: paralelismo de tareas o de datos.

Los lenguajes paralelos ahorran estas tareas utilizando:

- ♥ Construcciones propias del lenguaje, que además de distribuir la carga de trabajo pueden crear y terminar procesos.
- ♥ Directivas del compilador.
- ♥ Funciones de biblioteca que implementan en paralelo algunas operaciones usuales.

Estos programas son más cortos y fáciles de entender que los programas secuenciales con bibliotecas para programación paralela.

```
Por ejemplo, para calcular el número pi con directivas de OpenMP en C: [linenos]c include
<omp.h> define NUM_THREADS 4 int main(int argc, char**argv) double ancho, x, sum = 0; int intervalos, i; intervalos
// Crear y terminar pragma omp parallel // Localizar pragma omp for reduction (+:sum) /* Co-
municar y sincronizar*/ private(x) schedule(dynamic) /* Agrupar y asignar*/ for (i=0;i<intervalos;i++)
x = (i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x); sum*=ancho; return 0;
```

Compiladores paralelos

El compilador extrae automáticamente paralelismo a nivel de bucle (paralelismo de datos) y de función (paralelismo de tareas) a partir de una versión secuencial del código. Para ello realizan análisis de dependencias entre bloques de código, con las dependencias de datos que vimos en el tema anterior.

Los compiladores paralelos aún están limitados a aplicaciones que exhiben un paralelismo regular, como cálculos a nivel de bucle.

2.4.4 Comunicaciones Colectivas

Las herramientas de programación paralela ofrecen comunicación entre procesos de distintos tipos para comunicar a los procesos que colaboran entre sí en la ejecución de un código.

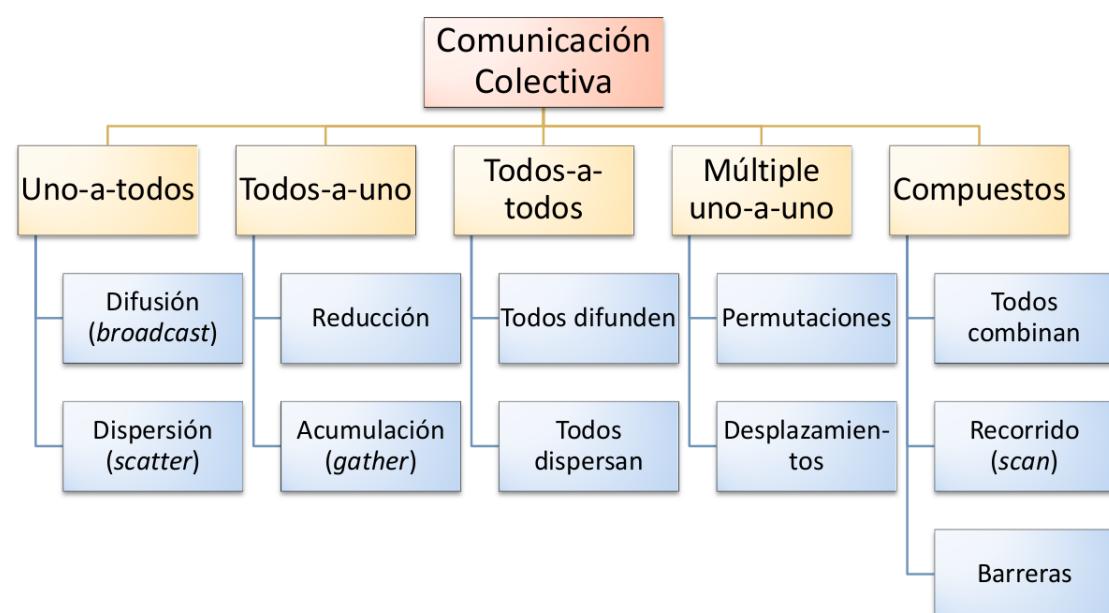


Figure 2.4: Comunicación paralela

La clasificación de las funciones colectivas (Figura ??) se presenta en cinco grupos:

- ♡ **Comunicación uno-a-todos:** Un proceso envía y todos los procesos reciben (Figura ??). Hay variantes en las que el proceso que envía no forma parte del grupo y otras en las que reciben todos los del grupo excepto el que envía. Dentro de este grupo hay varias variantes:
 - **Difusión:** todos los procesos reciben el mismo mensaje
 - **Dispersión (scatter):** cada proceso receptor recibe un mensaje diferente

- ♡ **Comunicación todos-a-uno:** todos los procesos en el grupo envían un mensaje a un único proceso (Figura ??).

Difusión (broadcast) Dispersión (scatter)

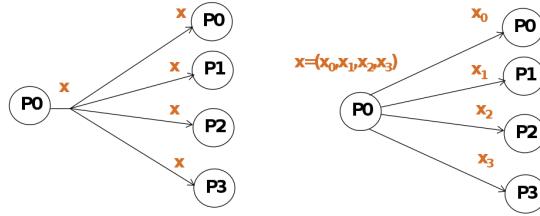


Figure 2.5: Comunicación uno-a-todos

- **Reducción:** los mensajes enviados se combinan en un sólo mensaje mediante algún operador. La operación de combinación suele ser conmutativa y asociativa.
- **Acumulación (gather):** los mensajes se reciben de forma concatenada en el receptor. El orden de concatenación depende del identificador del proceso.

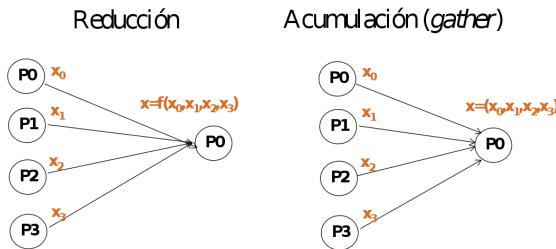


Figure 2.6: Comunicación todos-a-uno

- ♡ **Comunicación múltiple uno-a-uno:** hay componentes del grupo que envían un único mensaje y componentes que reciben un único mensaje (Figura ??). Si todos los componentes del grupo envían y reciben, se implementa una **permutación** (rotación, intercambio, barajes...).

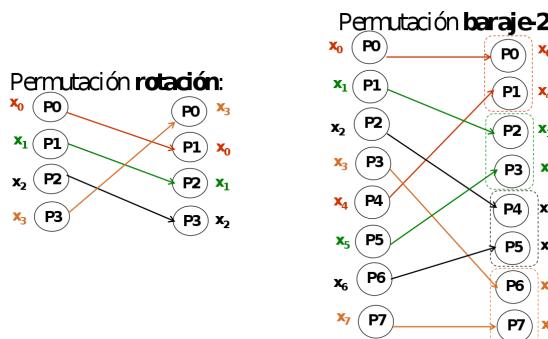


Figure 2.7: Comunicación múltiple uno-a-uno

- ♡ **Comunicación todos-a-todos:** Todos los procesos del grupo ejecutan una comunicación

uno-a-todos (Figura ??). Cada proceso recibe n mensajes, cada uno de un proceso diferente del grupo:

- **Todos-Difunden (all-broadcast)**: todos los procesos realizan una difusión. Usualmente las n transferencias recibidas por un proceso se concatenan en función del identificador del proceso que envía.
- **Todos-Dispersan (all-scatter)**: todos los procesos concatenan diferentes transferencias. En el ejemplo de la Figura ??, se ilustra una transposición de una matriz 4×4 : el procesador P_i dispersa la fila i (x_0, x_1, x_2, x_3); tras la ejecución, el procesador P_i tendrá la columna i (x_0, x_1, x_2, x_3).

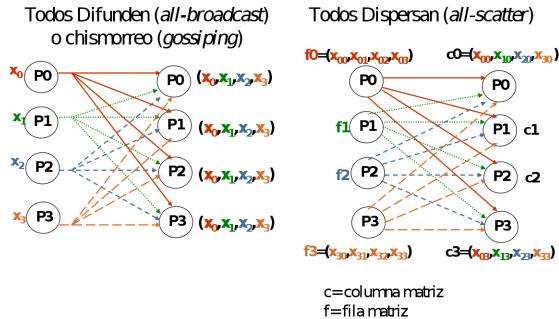


Figure 2.8: Comunicación todos-a-todos

♡ **Servicios compuestos**: hay servicios que resultan de una combinación de algunos de los anteriores, por ejemplo:

- **Todos combinan**: el resultado de aplicar una reducción se obtiene en todos los procesos porque se realizan tantas reducciones como procesos (Figura ??)

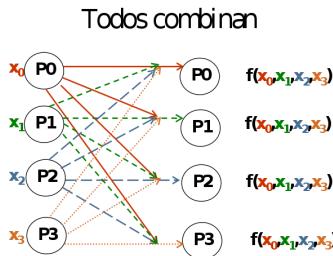


Figure 2.9: Todos combinan

- **Barrera**: es un punto de sincronización en el flujo de control de un programa que todos los procesos de un grupo (no enmascarados) deben alcanzar para que cualquier proceso (no enmascarado) del grupo pueda continuar con la ejecución de la instrucción que hay tras la barrera.
- **Recorrido (scan)**: todos los procesos envían un mensaje, recibiendo cada uno de ellos el resultado de reducir un conjunto de estos mensajes (Figura ??). Con recorrido prefijo paralelo, el proceso P_0 se queda con el dato que él mismo ha enviado; P_1 recibe la reducción de dos mensajes P_0 y P_1 ; P_2 recibe la reducción de los mensajes de P_0, P_1 y P_2 y así sucesivamente.

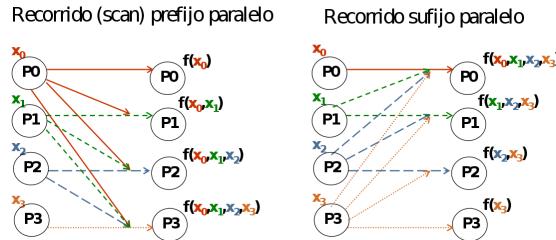


Figure 2.10: Recorrido

Ejemplo: comunicación colectiva en OpenMP

Uno-a-todos	Difusión	<ul style="list-style-type: none"> — Cláusula <code>firstprivate</code> (desde thread 0) — Directiva <code>single</code> con cláusula <code>copyprivate</code> — Directiva <code>threadprivate</code> y uso de la cláusula <code>copyin</code> en directiva <code>parallel</code> (desde thread 0)
Todos-a-uno	Reducción	<ul style="list-style-type: none"> — Cláusula <code>reduction</code>
Servicios compuestos	Barreras	<ul style="list-style-type: none"> — Directiva <code>barrier</code>

Ejemplo: comunicación en MPI

Uno-a-uno	Asíncrona	<ul style="list-style-type: none"> — <code>MPI_Send()</code> / <code>MPI_Receive()</code>
Uno-a-todos	Difusión Dispersión	<ul style="list-style-type: none"> — <code>MPI_Bcast()</code> — <code>MPI_Scatter()</code>
Todos-a-uno	Reducción Acumulación	<ul style="list-style-type: none"> — <code>MPI_Reduce()</code> — <code>MPI_Gather()</code>
Todos-a-todos	Todos difunden	<ul style="list-style-type: none"> — <code>MPI_Allgather()</code>
Servicios compuestos	Todos combinan Barreras Scan	<ul style="list-style-type: none"> — <code>MPI_Allreduce()</code> — <code>MPI_Barrier()</code> — <code>MPI_Scan()</code>

2.4.5 Estilos de programación y arquitecturas paralelas

Como se ve en la Figura ??, cada estilo de programación paralela se implementa mejor en distintos tipos de computadores.

Paso de mensajes

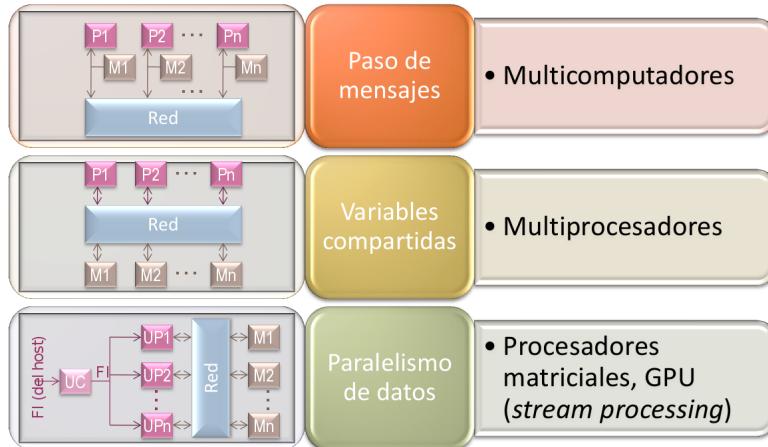


Figure 2.11: Estilos de programación y arquitecturas paralelas

Con **paso de mensajes** se supone que cada procesador en el sistema tiene su propio espacio de direcciones. Los mensajes llevan datos de una memoria a otra y se pueden usar para sincronizar procesos. Estos datos transferidos están duplicados en el sistema de memoria.

Disponemos de varias herramientas como *lenguajes de programación* (Ada) o *bibliotecas de funciones* que actúan de interfaz al sistema de comunicaciones (MPI).

Las funciones básicas de comunicación de paso de mensajes deben permitir la comunicación uno-a-uno entre procesos, con dos funciones:

- ♡ *Send (destino, datos)*, donde *datos* especifica la variable que contiene los datos a enviar.
- ♡ *Receive (fuente, datos)*, donde *datos* especifica la variable dónde se guardarán los datos recibidos.

Las transmisiones pueden ser:

- ♡ *Síncronas*: el proceso que ejecuta el envío de datos queda bloqueado hasta que el destino ejecute la función para recibirlos correspondiente. Igualmente, el proceso que ejecuta el receive queda bloqueado hasta que se ejecute el send correspondiente. Estas operaciones implementan las denominadas **citas**
- ♡ *Asíncronas*: el send no deja bloqueado al proceso que lo ejecuta, generalmente se ofrece una implementación de receive que deja bloqueado el proceso hasta que se realice en correspondiente send. Para que el send sea no bloqueante debe usarse un buffer donde guardar los datos que se envían al destino.

Variabes compartidas

Con **variables compartidas**, los procesadores del sistema comparten el mismo espacio de direcciones, por eso no se necesita transferir datos, sino que se realizan transferencias con instrucciones del procesador implícitamente. Para sincronizar procesos el programador usa primitivas que ofrece el software.

Disponemos de varias herramientas como *lenguajes de programación* (Java), *bibliotecas de funciones* (PThread) y alternativas que usan lenguaje secuencial *más directivas del compilador* (OpenMP).

La comunicación entre procesos se realiza accediendo a variables compartidas. Las hebras pueden compartir variables globales pero los procesos no porque tienen espacio de direcciones independiente.

Las herramientas software ofrecen mecanismos para implementar sincronización como cerros, semáforos ó monitores. Estas herramientas sitúan al programador en distintos niveles de abstracción

Paralelismo de datos

Con **paralelismo de datos**, las mismas operaciones se ejecutan en paralelo en múltiples unidades de procesamiento de forma que cada unidad aplica la operación a un conjunto de datos distinto. Sólo soporta paralelismo a nivel de bucle y la sincronización está implícita.

Permite aprovechar el paralelismo de datos en aplicaciones que organizan sus datos en vectores o matrices. El programador no debe preocuparse de las sincronizaciones pues están implícitas y las genera el compilador.

Los lenguajes de paralelismo de datos como C* aparecieron ligados a procesadores matriciales (arquitectura SIMD).

2.4.6 Estructuras típicas de códigos paralelos

Dueño-Esclavo (master-slave) o Granja de tareas (task-farming)

Se distingue un proceso dueño y múltiples esclavos. El dueño se encarga de distribuir las tareas de un conjunto (granja) entre el grupo de esclavos y de ir recolectando los resultados parciales que van calculando para obtener el resultado final. No suele haber comunicación entre los esclavos (Figura ??).

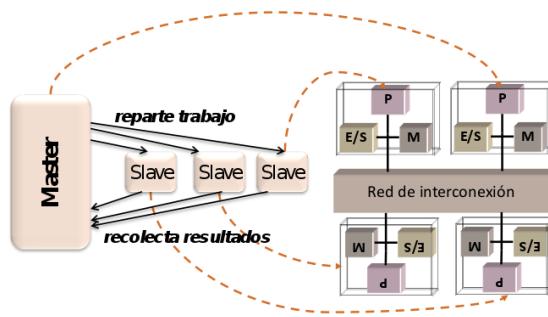


Figure 2.12: Esquema del dueño-esclavo

Si todos los esclavos ejecutan el mismo código, se puede hacer un programa para el dueño y otro para el esclavo (modo mixto MPMD-SPMD). También pueden juntarse ambos en un único programa (SPMD).

El reparto de trabajo puede hacerse de forma dinámica (durante la ejecución) ó de forma estática (por el programador). De forma estática sabemos a priori lo que va a hacer cada esclavo y el dueño sólo se encarga de reunir resultados.

La escabilidad dependerá del número de esclavos y de su comunicación con el dueño. Para aumentar la escabilidad puede haber múltiples dueños cada uno con un subconjunto de esclavos.

Esta estructura de la Figura ?? es a nivel de procesos

Cliente-Servidor

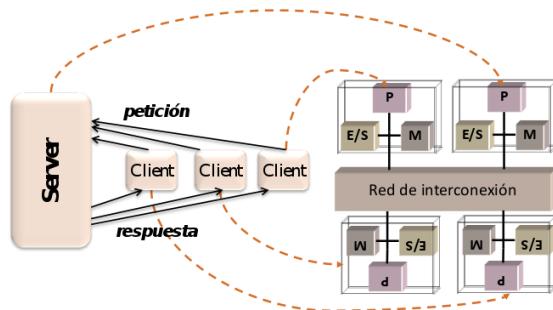


Figure 2.13: Esquema del cliente-servidor

Tiene una estructura parecida al dueño-esclavo pero cambia el sentido de las flechas. Los clientes envían una petición al servidor y éste les devuelve la respuesta.

Descomposición de dominio o descomposición de datos

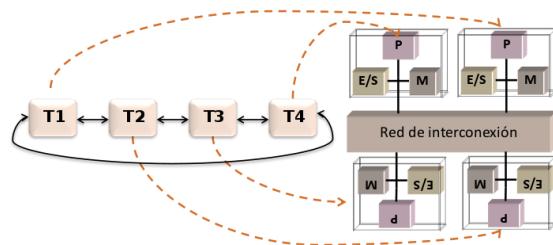


Figure 2.14: Esquema de la descomposición de datos

Se usa en problemas en los que se opera con estructuras de datos grandes. Éstas se dividen en partes y con esta división se derivan las tareas paralelas. Éstas suelen realizar las mismas operaciones (Figura ??). Los algoritmos con imágenes, por ejemplo, admiten descomposición de datos (Figura ??).

Para hacer el filtrado de una imagen cada pixel se sustituye por una combinación de los píxeles que tiene alrededor y él mismo. En el ejemplo se ve una imagen de 4×4 px. Para decodificar la

Imagen repartimos el trabajo por columnas, cada columna se asigna a un flujo de control distinto (en este caso tendremos cuatro threads). Para obtener el filtrado de un píxel necesitamos datos presentes en otras columnas. Las comunicaciones necesarias se harían por filas y columnas (los datos de las columnas los tiene ya ese thread por tanto sólo se comunicaría un thread con el anterior y el posterior en la matriz), por tanto, los threads envían y reciben de otros dos threads.

El código para hacer esto se ve en la Figura ??.

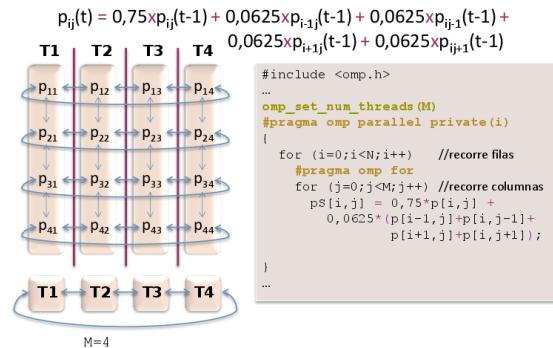


Figure 2.15: Ejemplo: filtrado de imagen

Los procesos pueden englobar varias tareas. Suelen ejecutar el mismo código (SPMD) aunque cada uno trabaja sobre un conjunto de datos diferente. Puede haber comunicaciones entre procesos, de hecho, si no hay comunicación entre procesos se conoce como *embarrassing parallel computation*.

Estructura segmentada (pipeline) o flujo de datos

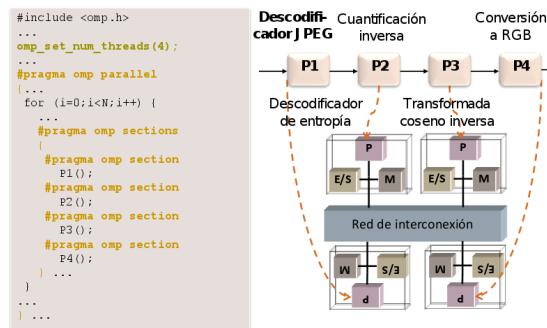


Figure 2.16: Esquema de flujo de datos con ejemplo de decodificador JPEG

Se usa para aplicar a un flujo de datos en secuencia distintas funciones (parallelismo de tareas). La estructura de los procesos y de las tareas es la de un cauce segmentado (Figura ??). Cada proceso ejecuta, por tanto, código distinto (MPDP).

Por ejemplo, en un decodificador de imágenes JPEG, se aplica en secuencia a bloques de 8×8 píxeles de una imagen de entrada, las siguientes funciones: decodificación de entropía, cuantifi-

cación inversa, transformada del coseno inversa y conversión RGB. Por tanto el decodificador se podría implementar con cuatro procesos en una estructura segmentada.

No se puede aplicar una descomposición de dominio porque cada bloque necesita los datos que se han calculado del bloque anterior.

Para traducir esto a código y ejecutar en paralelo las distintas etapas usamos la directiva de OpenMP **section**.

Divide y vencerás (divide and conquer) ó descomposición recursiva

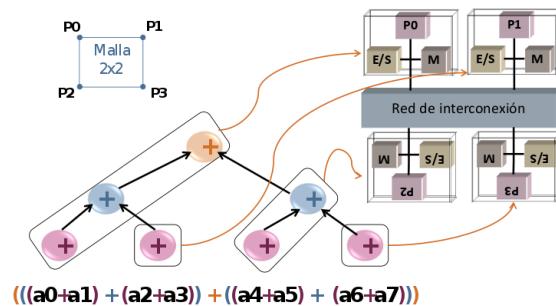


Figure 2.17: Esquema de divide y vencerás

Esta estructura se utiliza cuando un problema se puede dividir en subproblemas independientes. Si los subproblemas se pueden dividir, lo harán también, recursivamente. Las tareas presentan una estructura en forma de árbol y no habrá comunicación interacciones entre los hijos del mismo parente. (Figura ??).

Los programas secuenciales recursivos (paralelismo de tareas) y la descomposición de los datos de un problema recursivamente (paralelismo de datos) también tienen divide y vencerás. Por ejemplo, un vector cuyos datos se van a sumar puede descomponerse en vectores la mitad de componentes hasta llegar a vectores de dos componentes sólo.

La asignación de tareas puede hacerse asignando a cada proceso una tarea o dependiendo de la red de interconexión de la máquina, para disminuir el número de enlaces de la red que se utilizarán para cada comunicación.

En la Figura ?? se descompone la suma de 8 números, en primer lugar en la suma de cuatro números y, en el siguiente nivel del árbol, en la suma de dos números. Los resultados de cada nivel se deben sumar para obtener la combinación lineal del nivel superior.

El número mínimo de procesadores que necesitamos para tener paralelismo (para obtener paralelismo en una aplicación¹) viene dado por el número de hojas del árbol.

¹GRADO DE PARALELISMO: número máximo de flujos de control que un programa puede ejecutar en paralelo, donde para que la ejecución de nuestras tareas sea lo más eficiente posible. No puede superar este número.

2.5 Proceso de paralelización

Para obtener una versión paralela de una aplicación se podrían seguir los siguientes pasos:

- ♡ **Descomponer** la aplicación en tareas independientes
- ♡ **Asignar** tareas a procesos o a hebras
- ♡ **Redactar** código paralelo
- ♡ **Evaluuar** prestaciones

2.5.1 Descomposición en tareas independientes

En esta etapa se buscan unidades de trabajo independientes que, junto con los datos que utilizan, formarán las *tareas*. Se deben representar mediante un grafo dirigido las dependencias entre las tareas: los arcos en el grafo representan el flujo de datos y de control, y los vértices, tareas. Así, podemos detectar las tareas independientes y facilitarnos la vida en la etapa de asignación.

Esta búsqueda se realiza a partir de un código secuencial en dos niveles de abstracción:

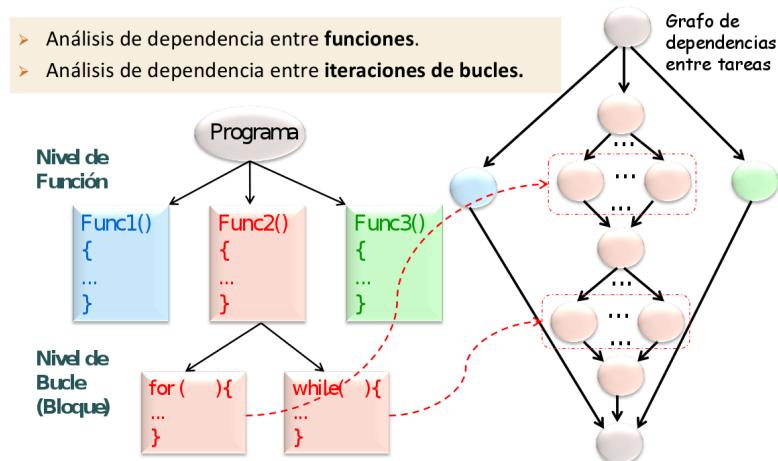


Figure 2.18: Descomposición en tareas

♡ **Nivel de función:** analizando las dependencias entre las funciones del código encontramos las que son independientes o las que se pueden hacer independientes y, por tanto, ejecutarse en paralelo. En el ejemplo de la Figura ?? se ha encontrado que las tres funciones son independientes.

♡ **Nivel de bucle:** analizando las iteraciones de los bucles dentro de una función, podemos encontrar si son o se pueden hacer independientes (es decir, asegurarnos de que no haya dependencias RAW entre las iteraciones del bucle). Con el análisis de los bucles podemos detectar el paralelismo de datos. Además, si una función tiene varios bucles podemos analizar la dependencia entre ellos. En el ejemplo de la Figura ??, una de las funciones consta de dos

bucles y con el grafo de dependencias se ha encontrado que son independientes las iteraciones de cada bucle, en cambio, la salida del bucle `for` es usada en el bucle `while`, por eso ambos ciclos están a diferente nivel en el grafo.

Si partimos de la definición de la aplicación se puede extraer paralelismo de tareas (*task-parallelism*). Para extraer paralelismo de datos (*data parallelism*) se podría estudiar dentro de cada función la estructura de los datos y realizar una descomposición recursiva.

Ejemplo de cálculo π : descomposición en tareas independientes

Para paralelizar el cálculo de π se puede partir de una versión secuencial o un planteamiento para que el problema se preste a paralelización.

Podemos definir el cálculo de π como un problema de integración numérica, π es la integral en el intervalo $[0, 1]$ de la derivada del arco tangente de x que es $\frac{\pi}{4}$:

$$\left. \begin{array}{l} \arctg'(x) = \frac{1}{1+x^2} \\ \arctg(1) = \frac{\pi}{4} \\ \arctg(0) = 0 \end{array} \right\} \Rightarrow \int_0^1 \frac{1}{1+x^2} = \arctg(x) \Big|_0^1 = \frac{\pi}{4} - 0$$

$(i + 0.5) \cdot A$ es la altura del rectángulo del que estamos calculando el área y A es la base de dicho rectángulo.

$$\pi = 4 \cdot \int_0^1 \frac{1}{1+x^2} \approx 4 \cdot \sum_{i=0}^{intervalos-1} Ancho \cdot \frac{1}{1+[(i+0.5) \cdot A]^2}$$

La fórmula que implementamos en el código secuencial es:

$$\pi = A \cdot \sum_{i=0}^{intervalos-1} \frac{4}{1+[(i+0.5) \cdot A]^2}$$

Si sacamos el ancho fuera obtenemos el área con mayor precisión y con menor número de intervalos. Así, calculamos la altura de cada rectángulo y al final, multiplicamos toda la sumatoria por el ancho del rectángulo.

El área de la derivada del arcotangente en $[0, 1]$ se aproxima dividiendo el intervalo en subintervalos, calculando el área de cada subintervalo y sumando las áreas obtenidas en cada subintervalo. El error será menor cuanto mayor sea el número de subintervalos.

El cálculo del área de los diferentes intervalos es independiente, por lo que podemos repartir estos cálculos en un conjunto de procesadores. Si se divide el intervalo en 100 y disponemos de 10 procesadores, a cada procesador podemos asignarle el cálculo de 10 áreas.

Otra alternativa es partir de una versión secuencial. Esta versión secuencial calcula π utilizando rectángulos en los que la altura es el valor de la derivada de la arcotangente en el punto medio. Usa un bucle `for` que escribe siempre en la misma variable `sum`, pero esta dependencia se puede eliminar suponiendo que cada hebra escribirá en una variable `sum` distinta:

```
[linenos, frame=single, label=Código secuencia]
[linenos, frame=single, label=Código con el paralelismo extraído]
c int main (int argc, char **argv) double
ancho, sum=0; int intervalos, i;
// Primer nodo del grafo // y primer nivel
intervalos = atoi(argv[1]); ancho = 1.0/(double)
intervalos;
// Segundo y tercer nodo // del grafo, en el
segundo nivel for (i=0;i<intervalos;i++) x =
(i+0.5)*ancho; sum = sum + 4.0/(1.0+x*x);
// Cuarto nodo del grafo, // en el tercer nivel
sum *= ancho;
return 0;
frame=single, label=Código con el paralelismo extraído]
c include <omp.h> define
NUM_THREADS 4
int main(int argc, char *
*argv) double ancho, sum = 0; int intervalos, i;
intervalos = atoi(argv[1]); ancho =
1.0/(double)intervalos;
#pragma omp parallel
#pragma omp for for
for (i=0;i<intervalos;i++) x = (i+0.5)*ancho; sum =
sum + 4.0/(1.0+x*x); sum *= ancho;
return 0;
```

2.5.2 Asignar tareas a procesos o hebras

Se realiza la asignación de las tareas del grado de dependencia a procesos y hebras. En general, no es conveniente asignar más de un proceso o hebra por procesador en una misma aplicación, por lo que la asignación a procesos y hebras (*scheduling*) está asignada con la asignación/mapeo de procesadores (*mapping*), incluso se puede realizar la asignación asociando los procesos/hebras a procesadores concretos.

La granularidad de los procesos y hebras dependerá de:

- ♡ Del número de procesadores que haya, ya que cuanto mayor sea su número menos tareas se asignarán a cada proceso. No se recomienda tener más threads que cores porque sino, perdemos tiempo comunicando threads de un mismo core y es más eficiente asignar el trabajo de esas dos threads a uno sólo y ejecutarlo en un core.
- ♡ Del tiempo de comunicación y sincronización, ya que el tiempo de ejecución paralelo no sólo depende del tiempo que supone la ejecución en paralelo de las tareas, sino que también depende del tiempo de comunicación y sincronización entre procesos/hebras. Para reducir este tiempo se asignan varias tareas a un mismo proceso/hebra y así reducir las interacciones. Si replicamos una tarea en todos los threads (que todos los threads ejecuten el mismo cálculo en vez de que lo ejecute uno y envíe el resultado al resto) disminuimos tiempo de comunicación y sincronización y podemos conseguir mayor velocidad.

La asignación debería *repartir la carga* optimizando la *comunicación y sincronización*, de forma que todos los procesadores empiecen y terminen a la vez, es decir, no deben hacerse esperar los unos a los otros.

En el grafo de tareas, éstas representan tiempo de cálculo, y los arcos, tiempo de comunicación y sincronización. Para realizar la asignación y cumplir estos objetivos, conviene saber la *arquitectura* en la que se va a ejecutar la aplicación y conocer el tiempo de ejecución de cada tarea y de las comunicaciones/sincronizaciones entre ellas.

El equilibrado depende de:

♡ **La arquitectura:**

- **Heterogénea:** consta de componentes con diferentes prestaciones, por lo que un buen equilibrado debería asignar más trabajo a los nodos más rápidos. Cuesta más trabajo asignar el trabajo para que todos empiecen y terminen a la vez.

→ **Homogénea:** Todos los nodos son iguales, con la misma velocidad y usando las mismas herramientas de programación. hay dos subtipos

- *Uniforme*: la comunicación de los procesadores con la memoria (multiprocesadores) o entre sí (multicomputadores) supone el mismo tiempo sean cuales sean los nodos que intervienen. En esta arquitectura no tiene mucha influencia el no considerar la arquitectura, aunque la distribución de las tareas de forma que las tareas de cada procesador estén en módulos de memoria distintos reduce las colisiones.
- *No Uniforme*: la asignación de tareas de forma que se minimice el tiempo de comunicación y sincronización es más difícil. Para mejorar el tiempo de ejecución se pueden eliminar transferencias a través de la red asignando tareas al mismo procesador y disminuir las distancias que se atraviesan en las transferencias.

♡ **La aplicación/descomposición:** porque hay aplicaciones en las que las tareas no tardan todas lo mismo, aunque sean iteraciones de un bucle.

La asignación de tareas a procesadores se puede realizar de dos formas:

♡ **Estática:** en tiempo de compilación (implícita) o al escribir el programa (explícita).

♡ **Dinámica:** en tiempo de ejecución. Su ventaja es que permite evitar repartos estáticos complejos, especialmente cuando no se conoce el número de tareas que se van a ejecutar antes de la ejecución, y permite que acabe una aplicación aunque falle un procesador. Aunque consume un tiempo extra de comunicación y sincronización. Aun sabiendo el número de tareas a ejecutar al programador puede resultarle más cómodo usar asignación dinámica para conseguir un buen equilibrio, por ejemplo, cuando tenemos una arquitectura heterogénea/no uniforme/las tareas no tardan lo mismo, ya que se va asignando trabajo conforme se van terminando las tareas.

Normalmente se deja al sistema operativo el mapeo de threads (*light process*). Lo puede hacer el entorno o sistema en tiempo de ejecución (*runtime system* de la herramienta de programación) aunque el programador puede influir.

Por ejemplo, en el filtrado de una imagen, se puede dividir el proceso en columnas (Figura ??) o en filas (Figura ??) y la asignación a procesadores, como tenemos en ambos casos cuatro tareas y cuatro procesadores, será de una tarea para cada procesador. Además, intentaremos al hacer el mapeo que procesadores cercanos tengan tareas que se comuniquen entre sí para que las comunicaciones y sincronizaciones sean más eficientes, ya que tenemos una arquitectura homogénea no uniforme.

Entonces, dependiendo de si repartimos el trabajo por columnas o por filas, tendremos los siguientes códigos paralelos:

```
perldoc [linenos]c void Func1() ... void Func2() ... pragma omp parallel for schedule(static) for (i=0;i<N;i++)
// código para i ... void Func3() ... int main () pragma omp parallel sections pragma omp
section Func1(); pragma omp section Func2(); pragma omp section Func3(); ...
```

El siguiente código es un ejemplo de asignación estática de paralelismo de tareas y datos en OpenMP:

```
[linenos]c void Func1() ... void Func2() ... pragma omp parallel for schedule(static) for (i=0;i<N;i++)
// código para i ... void Func3() ... int main () pragma omp parallel sections pragma omp
section Func1(); pragma omp section Func2(); pragma omp section Func3(); ...
```

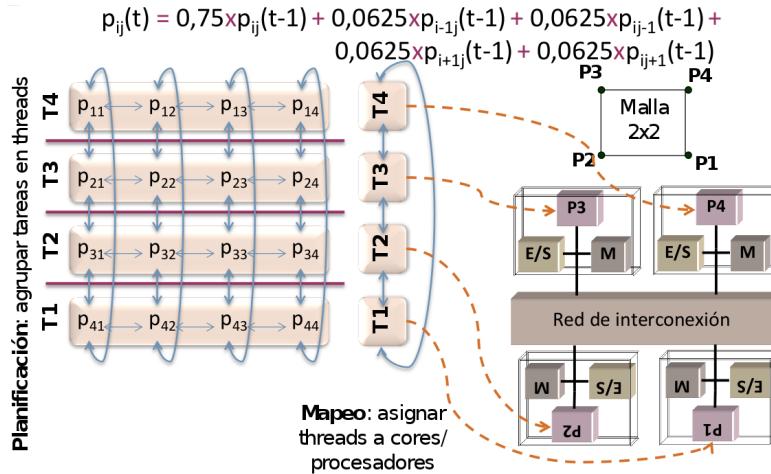


Figure 2.19: Filtrado de imagen por filas

Para repartir el trabajo nosotros mismos podemos usar en vez del `sections`, el identificador del thread. Así haríamos la asignación estática y explícita. Tal y como está en el código, como lo hace la herramienta de programación, es estática pero implícita.

Cada hebra ejecutará las funciones `Func1`, `Func2` y `Func3` en secuencial, pero las tres se ejecutarán en paralelo. Dentro de `Func2` se ejecutará el bucle `for` en paralelo, es decir, la hebra que ejecute `Func2` tendrá otras subhebras.

➤ Asignación **estática** y explícita de las iteraciones de un bucle:

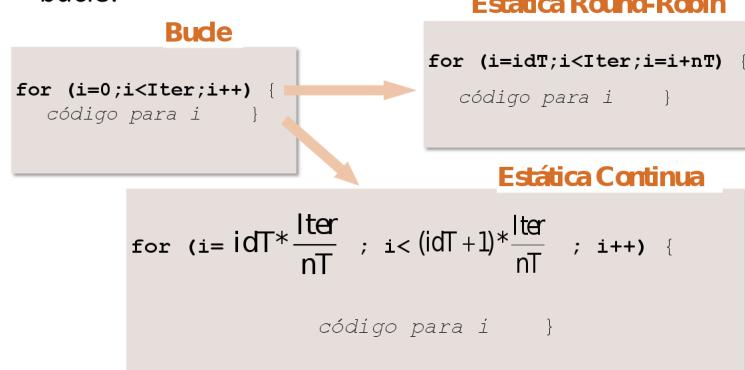


Figure 2.20: Asignación estática explícita de las tareas de un bucle a procesos.

En la Figura ?? se pueden ver dos alternativas que el programador puede utilizar explícitamente para asignar las iteraciones de un bucle a procesos de forma estática. En la asignación *round-robin*, iteraciones consecutivas del bucle se asignan a procesos consecutivos (con identificador consecutivo). En el otro ejemplo se asignan iteraciones consecutivas al mismo proceso.

Estas asignaciones serían explícitas porque las hace el programador variando los límites del bucle `for`.

En el ejemplo de π , sería mejor asignar las tareas consecutivas al mismo thread para no perder precisión al sumar los resultados, ya que la números en coma flotante con exponentes dispares puede hacer que perdamos decimales.

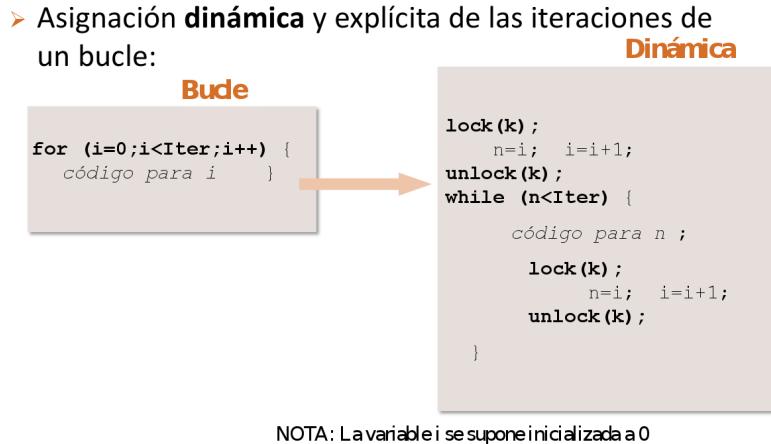


Figure 2.21: Asignación dinámica explícita de las tareas de un bucle a procesos/hebras

La Figura ?? muestra cómo el programador puede implementar explícitamente en el código una asignación dinámica para memoria compartida. Las iteraciones se reparten en orden: primero la 0, después la 1, y así sucesivamente hasta la última. El índice i del bucle es una variable compartida que consultan los procesos para “coger” la siguiente iteración que van a realizar e incrementan su valor en uno para que el siguiente proceso no coja una iteración ya asignada. Esto debe hacerse en exclusión mutua, para ello se usa un cerrojo k . Con paso de mensajes la asignación dinámica la haría el proceso dueño en un esquema dueño-esclavo.

Usando asignación dinámica añado tiempo de ejecución aunque la asignación de tareas es más equilibrada.

Asignación de tareas a dos threads estáticas por round-robin para el cálculo de π

Si tenemos dos cores en nuestra máquina, podemos dividir el trabajo de calcular áreas de rectángulos como se ve en la Figura ??:



Figure 2.22: Cálculo de π diviendo el área bajo la gráfica en 6 rectángulos

Así, cada core calculará tres áreas y luego se sumarán los resultados de cada área para saber el valor de pi.

Ejemplo: multiplicación matriz por vector

$$c = A \bullet b; \quad c_i = \sum_{k=0}^{M-1} a_{ik} \bullet b_k = a_i^T \bullet b, \quad c(i) = \sum_{k=0}^{M-1} A(i, k) \bullet b(k), \quad i = 0, \dots, N-1$$

\bullet

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}

b_1 $N=8$ $M=6$ $=$ c_1
 b_2 c_2
 b_3 c_3
 b_4 c_4
 b_5 c_5
 b_6 c_6
 c_7
 c_8

$c(1) = \sum_{k=0}^{M-1} A(1, k) \bullet b(k)$
 $c(2) = \sum_{k=0}^{M-1} A(2, k) \bullet b(k)$
 $c(3) = \sum_{k=0}^{M-1} A(3, k) \bullet b(k)$
 $c(4) = \sum_{k=0}^{M-1} A(4, k) \bullet b(k)$
 $c(5) = \sum_{k=0}^{M-1} A(5, k) \bullet b(k)$
 $c(6) = \sum_{k=0}^{M-1} A(6, k) \bullet b(k)$
 $c(7) = \sum_{k=0}^{M-1} A(7, k) \bullet b(k)$
 $c(8) = \sum_{k=0}^{M-1} A(8, k) \bullet b(k)$

Figure 2.23: Asignación por salida

$$c = A \bullet b; \quad c_i = \sum_{k=0}^{M-1} a_{ik} \bullet b_k = a_i^T \bullet b, \quad c(i) = \sum_{k=0}^{M-1} A(i, k) \bullet b(k), \quad i = 0, \dots, N-1$$

\bullet

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}

b_1 $N=8$ $M=6$ $=$ c_1
 b_2 c_2
 b_3 c_3
 b_4 c_4
 b_5 c_5
 b_6 c_6
 c_7
 c_8

$a_{11}b_1 + a_{12}b_2 + a_{13}b_3 + a_{14}b_4 + a_{15}b_5 + a_{16}b_6$
 $a_{21}b_1 + a_{22}b_2 + a_{23}b_3 + a_{24}b_4 + a_{25}b_5 + a_{26}b_6$
 $a_{31}b_1 + a_{32}b_2 + a_{33}b_3 + a_{34}b_4 + a_{35}b_5 + a_{36}b_6$
 $a_{41}b_1 + a_{42}b_2 + a_{43}b_3 + a_{44}b_4 + a_{45}b_5 + a_{46}b_6$
 $a_{51}b_1 + a_{52}b_2 + a_{53}b_3 + a_{54}b_4 + a_{55}b_5 + a_{56}b_6$
 $a_{61}b_1 + a_{62}b_2 + a_{63}b_3 + a_{64}b_4 + a_{65}b_5 + a_{66}b_6$
 $a_{71}b_1 + a_{72}b_2 + a_{73}b_3 + a_{74}b_4 + a_{75}b_5 + a_{76}b_6$
 $a_{81}b_1 + a_{82}b_2 + a_{83}b_3 + a_{84}b_4 + a_{85}b_5 + a_{86}b_6$

Figure 2.24: Asignación por entradas

Para paralelizar un proceso que calcule la multiplicación de una matriz por un vector, podemos:

♡ **Dividir las salidas de una tarea:** (Figura ??), así las operaciones para obtener $c(1)$ y $c(2)$, por ejemplo, las haría un thread y así. En esta alternativa no es necesaria comunicación entre los threads ya que cada uno calcula salidas definitivas

- ♡ **Dividir las entradas de una tarea:** (Figura ??), troceando la matriz por columnas (troceando las entradas también troceamos las salidas). En esta alternativa hay comunicación entre los threads ya que, por ejemplo, para calcular $c(1)$ tenemos cálculos en los tres threads. Esta alternativa es peor que la primera ya que en esta necesitamos comunicación y en la otra no.

2.5.3 Redactar código paralelo

El código paralelo va a depender de:

- ♡ **Estilo de programación utilizado:** variables compartidas, paso de mensajes, paralelismo de datos
- ♡ **Modo de programación:** SPMD, MPMD mixto
- ♡ **Punto de partida:** versión secuencial, descripción del problema
- ♡ **Herramienta software disponible para hacer explícito el paralelismo:** lenguajes de programación, directivas, bibliotecas de funciones, compilador
- ♡ **Estructura:** granja de tareas, segmentada, descomposición de datos, divide y vencerás, que depende a su vez de la aplicación.

Habrá que añadir o utilizar en el programa las funciones, directivas o construcciones del lenguaje que hagan falta para:

- ♡ *Crear y terminar* procesos, o en caso de que se creen de forma estática, *enrolar* y *desenrolar* procesos en el grupo que va a cooperar en el cálculo.
- ♡ *Localizar* el paralelismo
- ♡ *Asignar* la carga de trabajo conforme a la decisión tomada en el paso anterior
- ♡ *Comunicar y sincronizar* los diferentes procesos/hebras.

Cálculo de π en MPI

```
El código para el cálculo de  $\pi$  en MPI sería el siguiente: [linenos]c include <mpi.h> int main
(int argc, char **argv) double ancho, x, lsum, sum; int intervalos, i, nproc, iproc;
//Enrolar if (MPIInit(argc, argv)! = MPI_SUCCESS)exit(1);
MPICommSize(MPI_COMM_WORLD, nproc); MPICommRank(MPI_COMM_WORLD, iproc);
intervalos=atoi(argv[1]); ancho=1.0/(double)intervalos; lsum=0;
//Localizar/agrupar for (i=iproc; i<intervalos; i+=nproc) x = (i+0.5)*ancho; lsum+=4.0/(1.0+x*x);

lsum*=ancho;
//Comunicar/sincronizar MPIReduce(lsum, sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
//Desenrolar MPIFinalize();
```

A diferencia del código en OpenMP, en MPI al ser de un nivel de abstracción más bajo, tenemos que repartir las iteraciones del `for` y enrolar y desenrolar procesos manualmente.

2.5.4 Evaluación de prestaciones

Una vez redactado el código deben evaluarse sus prestaciones. Si no son las deseadas, debemos volver hacia atrás, se retrocederá más hacia atrás cuanto mayor sea la necesidad de mejora de prestaciones.

Si se vuelve a la etapa de escritura puede elegirse otra herramienta de programación, ya que no todas ofrecen las mismas prestaciones. Se puede volver a la etapa de asignación para realizar un mejor reparto de la carga. Se puede volver a la etapa de descomposición para mejorar la descomposición en tareas o modificar el punto de partida.

2.6 Evaluación de prestaciones en procesamiento paralelo

2.6.1 Ganancia en prestaciones y escabilidad

Evaluación de prestaciones

Las medidas que se usan para evaluar y comparar distintas realizaciones de una arquitectura son medidas genéricas, que se utilizan también en la evaluación de otros sistemas. En computadores paralelos suelen ser:

- ♡ **Tiempo de ejecución (respuesta):** tiempo que supone la ejecución de una aplicación en el sistema. Puede ser el tiempo real (*wall-clock time, elapsed time (/usr/bin/time)*) ó el tiempo de usuario, sistema o la suma de ambos: el tiempo de CPU.
- ♡ **Productividad:** número de aplicaciones que el computador es capaz de procesar por unidad de tiempo.

Dependiendo de la utilización del sistema se le dará más énfasis a mejorar una u otra (se mejorará la productividad en, por ejemplo, una red de computadores y el tiempo de respuesta, en aplicaciones HPC), o en algunos casos se intentará mejorar ambos aspectos (por ejemplo: servidores de internet).

Además se utilizan otras medidas de prestaciones adicionales:

- ♡ **Escabilidad:** evolución de la ganancia en prestaciones que se consigue en el sistema conforme se añaden recursos. Pretende medir el nivel efectivo de aprovechamiento efectivo de los recursos.
- ♡ **Eficiencia:** permite evaluar en qué medida las prestaciones que ofrece un sistema para un programa paralelo se acercan a las prestaciones máximas que idealmente debería ofrecer dado los recursos de los que dispone (por ejemplo, un programa secuencial que tarda en ejecutarse T , si hacemos una versión paralela y lo ejecutamos en un sistema con cuatro cores debería tardar $T/4$).

Las prestaciones que se obtienen con p recursos serían las prestaciones que se obtienen con un recurso multiplicado por p :

$$E(p, n) = \frac{\text{Prestaciones}(p, n)}{p \cdot \text{Prestaciones}(1, n)} = \frac{S(p, n)}{p}$$

La ganancia máxima es p por lo que la eficiencia máxima es 1. Por otro lado, la ganancia mínima es 1 por lo que la eficiencia mínima será $\frac{1}{p}$

Ganancia en prestaciones. Escalabilidad

Se utiliza la ganancia en velocidad para estudiar en qué medida se incrementan las prestaciones al ejecutar una aplicación en un sistema con múltiples procesadores frente a su ejecución en un sistema uniprocesador. Se puede obtener como:

$$S(p) = \frac{\text{Prestaciones}(p)}{\text{Prestaciones}(1)}$$

Es decir, dividiendo las prestaciones que se consiguen en un sistema con p procesadores entre las prestaciones conseguidas en una versión secuencial. Si usamos el tiempo de respuesta para evaluar las prestaciones, las prestaciones vendrían dadas por la inversa del tiempo:

$$S(p) = \frac{T_s}{T_p(p)} \quad \text{con } T_p(p) = T_c(p) + T_o(p)$$

Donde T_s sería el tiempo de ejecución (respuesta) del programa secuencial y $T_p(p)$, el tiempo de ejecución del programa paralelo con p procesadores. El tiempo de ejecución en paralelo depende del tiempo de cómputo $T_c(p)$ y del tiempo de penalización o sobrecarga (*Overhead*) $T_o(p)$ en el que influyen factores como:

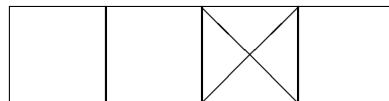
- ♥ Tiempo de comunicación/sincronización entre procesos
- ♥ Tiempo para crear/terminar (enrolar/desenrolar) los procesos
- ♥ Tiempo de ejecución de operaciones añadidas en la versión paralela no presentes en la secuencial.

La penalización que supone la *falta de equilibrio* en la distribución de tareas está incluída en la sobrecarga si no se ha tenido en cuenta al determinar el tiempo de cómputo paralelo.

Tanto el tiempo de cálculo como el *overhead* dependen del número de procesadores. Cuanto mayor sea éste mayor será el *grado de paralelismo* de la aplicación aprovechado. El *overhead* depende del número de procesos involucrados ya que la comunicación/sincronización y el crear/terminar procesos dependen de éste número.

La representación de la ganancia en función del número de procesadores nos permite evaluar la escalabilidad de una implementación paralela o de una arquitectura. Idealmente el tiempo se reduciría a $\frac{T_s}{p}$, pero para lograrlo, se debería poder distribuir el programa entre los procesadores y el *overhead* debería ser despreciable. En este caso, la ganancia sería p y la escalabilidad sería *lineal* (Figura ??). Para que la ganancia siempre pueda ser p , el *grado de paralelismo* debe ser ilimitado, es decir, siempre se podrá dividir el código entre los p procesadores disponibles sea cual sea el valor de p (Apartado a) de la Figura ??).

No obstante podemos encontrarnos a veces con una escalabilidad *superlineal* ($S(p) > p$), que se puede deber a que al aumentar el número de procesadores también aumentamos otros recursos (caché, memoria principal). Otra posibilidad es que estemos ejecutando una aplicación que consista en explorar un conjunto de posibilidades y en cuanto encuentre la posibilidad acertada termine. Por ejemplo, imaginemos que tenemos que explorar un vector de cuatro casillas donde la casilla tercera contiene la solución esperada:



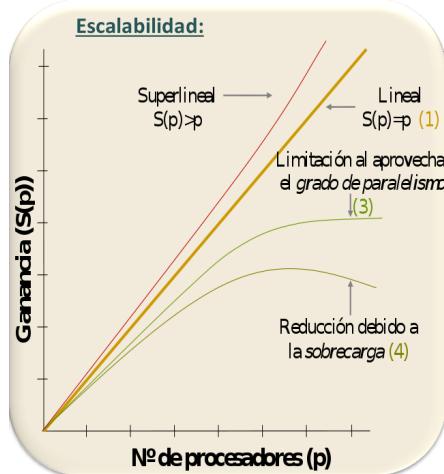
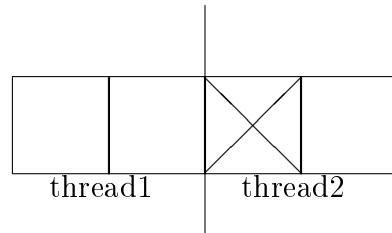


Figure 2.25: Escalabilidad lineal, superlineal y sublineal

En una solución secuencial tardaríamos $3T$ en llegar a la solución correcta, en cambio, si hacemos una versión secuencial con dos threads dividiendo el trabajo de la siguiente manera:



Sólo tardaríamos T en encontrar la solución correcta, es decir, tres veces menos que con la versión secuencial.

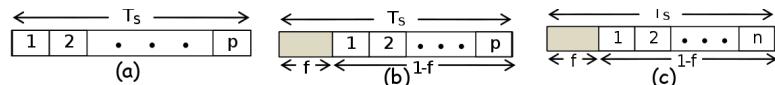


Figure 2.26: Modelos de código secuencial

Aunque hemos considerado que todas las tareas son paralelizables en la práctica siempre habrá una parte f que no sea paralelizable y que supone un tiempo no despreciable. Además, el grado de paralelismo, aunque lo hemos considerado ilimitado, estará limitado si se mantiene fijo el tiempo de ejecución secuencial T_s . Tampoco hemos considerado el tiempo de sobrecarga.

Modelo código	Fracción no paral. en T_s	Grado paralelismo	Overhead	Ganancia en función del número de procesadores p con T_s constante
a)	0	ilimitado	0	$S(p) = \frac{T_s}{T_p(p)} = p$ Ganancia Lineal (2.1)
b)	f	ilimitado	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}}$ $p \rightarrow \infty \frac{1}{f}$ (2.2)
c)	f	n	0	$S(p) = \frac{1}{f + \frac{(1-f)}{p}}$ $p = n$ $\frac{1}{f + \frac{(1-f)}{n}}$ (2.3)
b)	f	ilimitado	incrementa linealmente con p	$S(p) = \frac{1}{f + \frac{(1-f)}{p} + \frac{T_o(p)}{T_s}}$ $p \rightarrow \infty 0$ (2.4)

En la tabla anterior se incluyen expresiones de ganancia en velocidad para diferentes modelos de código secuencial. Las expresiones ??, ??, ?? y ?? se corresponden a las gráficas de la Figura ??.

En la expresión ?? se tiene en cuenta la presencia de código no parallelizable y que el grado de paralelismo puede ser menor que el número de procesadores disponibles ($n < p$), por tanto, la ganancia al incrementar el número de procesadores se limita cuando se aprovecha todo el grado de paralelismo de la aplicación.

En la expresión ?? se tiene en cuenta la presencia de código no parallelizable y el tiempo de sobrecarga, que se incrementa linealmente con p . Se considera el grado de paralelismo ilimitado. Esta función presenta un máximo para p , es decir, llega un momento cuando incrementamos el número de procesadores que la ganancia deja de crecer y comienza a decrecer. Este decremento se inicia cuando al incrementar p en 1 se incrementa el término de la sobrecarga ($\frac{T_o(p)}{T_s}$) en mayor medida que se decrementa la parte del cálculo paralelo ($f + \frac{1-f}{p}$) de forma que el denominador de la expresión aumenta en lugar de disminuir.

2.6.2 Ley de Amdahl

La ganancia en prestaciones utilizando p procesadores está limitada por la fracción de código que no se puede parallelizar:

$$S(p) = \frac{T_s}{T_p(p)} \leq \frac{T_s}{f \cdot T_s + \frac{(1-f) \cdot T_s}{p}} = \frac{p}{1 + f(p-1)} \rightarrow \frac{1}{f} (p \rightarrow \infty)$$

Esta expresión se obtiene a partir de la expresión ???. Donde:

- ♥ S es el incremento en velocidad que se consigue al aplicar una mejora (paralelismo)
- ♥ p es el incremento en velocidad máximo que se puede conseguir si se aplica la mejora todo el tiempo (número de procesadores)
- ♥ f es la fracción de tiempo en el que no se puede aplicar la mejora (fracción de tiempo no paralelizable).

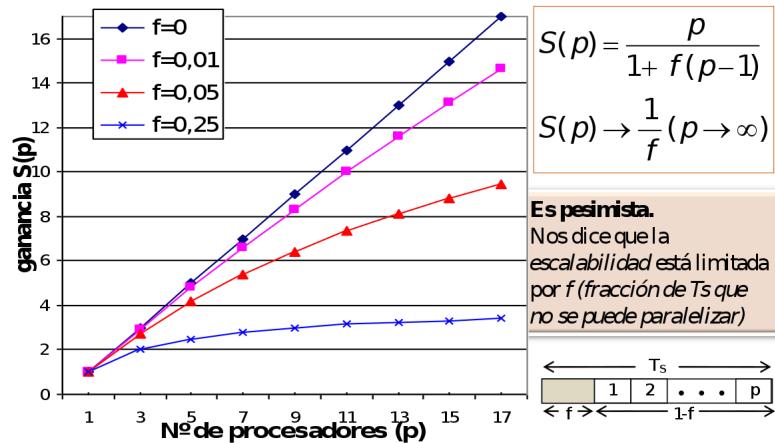
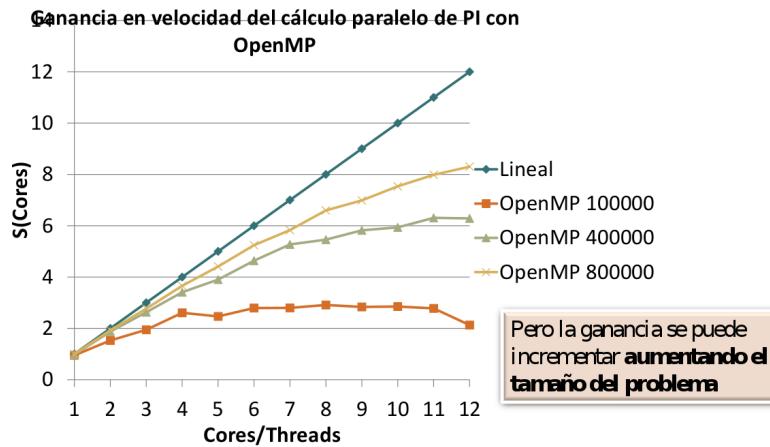


Figure 2.27: Ganancia en prestaciones para varias fracciones de código no paralelizable

La Ley de Amdahl nos da una visión pesimista de las ventajas de la parallelización, ya que nos dice que tenemos limitada la escalabilidad y que este límite depende de la fracción de código no paralelizable. En la Figura ?? se representa la ganancia en prestaciones (expresión ??) en función de p para diferentes fracciones de código no paralelizable. La escalabilidad decrece conforme se incrementa la fracción de código no paralelizable.

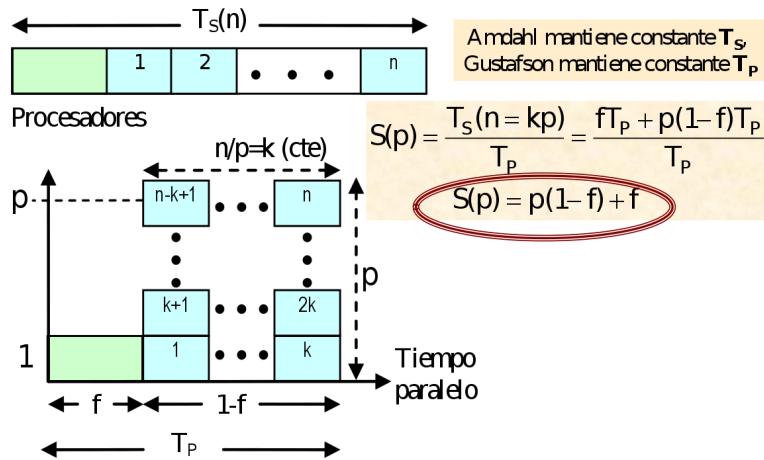
No obstante, se ha puesto constante el tiempo de ejecución secuencial T_s y con ello, se ha considerado la fracción de código paralelizable ($1 - f$) constante también. Pero en muchas aplicaciones se puede aumentar la parte de código paralelizable aumentando el tamaño del problema.

Por ejemplo, en el modelo basado en un bucle (Apartado c) de la Figura ??). Probablemente el bucle opere con una estructura matricial. Incrementando su tamaño, incrementaremos el número de iteraciones y, por tanto, el grado de paralelismo y la fracción de código paralelizable. En el ejemplo del cálculo de π , aumentar el tamaño del problema supone calcular π con mayor precisión, ya que el error al calcular el área de los rectángulos (Figura ??), disminuye el error (Figura ??)

Figure 2.28: Cálculo de π con distintos tamaños del problema

2.6.3 Ganancia escalable

Ley de Gustafson

Figure 2.29: Paralelización en p procesadores con $n = kp$. Si k es constante entonces el tiempo de ejecución en paralelo es constante (T_p)

Supongamos un modelo de código secuencial con una parte no paralelizable, y un número de tareas a paralelizar n que se puede incrementar aumentando el tamaño del problema (Figura ??). Considerando despreciable el tiempo de *overhead*, podemos mantener constante el tiempo de ejecución en paralelo T_p variando el número de procesadores p y el tamaño n de forma que

$n = kp$, con k constante. Así, el tiempo de ejecución sólo depende del tiempo de ejecución secuencial $T_s(n)$. Así, la ganancia en prestaciones sería:

$$S(p) = \frac{T_s(n)}{T_p} = \frac{fT_p + p(1-f)T_p}{T_p} = \frac{fT_p + p(1-f)T_p}{T_p} = p(1-f) + f$$

¡Ojo!, en este caso f es la fracción del programa NO PARALELIZABLE que corresponde a T_p , mientras que en la ley de Amdahl la f es la fracción del programa no paralelizable que corresponde a T_s , es muy importante no confundirlas y ver la diferencia clara. En la expresión lo único que varía es p .

La expresión viene del dibujo que tenemos en la Figura ??, como se ve en la foto tenemos por un lado una fracción de programa no paralelizable (fT_p) y por otro lado tenemos k conjuntos de p bloques de código paralelo que se con la fracción $1 - f$ ($p(1 - f)T_p$).

La ganancia depende linealmente del número de procesadores p con una pendiente $1 - f$, que es la fracción de tiempo que supone la ejecución de la parte paralela. Cuanto mayor sea $1 - f$ mayor será la escalabilidad.

Chapter 3

Arquitecturas con parallelismo de thread (TLP)

3.7 Arquitecturas TLP

3.7.1 Clasificación de arquitecturas con TLP explícito y una instancia del sistema operativo

♡ **Multicomputador**: puede ejecutar múltiples flujos de control en paralelo, cada flujo de control se ejecuta en un core distinto. Se puede ver empaquetado en diferentes niveles de encapsulación:

- a nivel de chip (multicores)
- a nivel de placa
- a nivel de sistema (un sistema está compuesto de armarios)

♡ **Multicore o multiprocesador en un chip o CMP (*Chip MultiProcessor*)**: es un multiprocesador en un chip, es decir, ejecuta varios threads en paralelo en un chip de procesamiento multicore (cada thread en un core distinto)

♡ **Core multithread**: Se puede modificar la arquitectura superescalar y segmentada de un core para que pueda ejecutar múltiples flujos de control concurrentemente o incluso, en paralelo (*Hyperthreading*).

3.7.2 Multiprocesadores

Criterio de clasificación: sistema de memoria

Esta clasificación la vimos en el Tema 1.

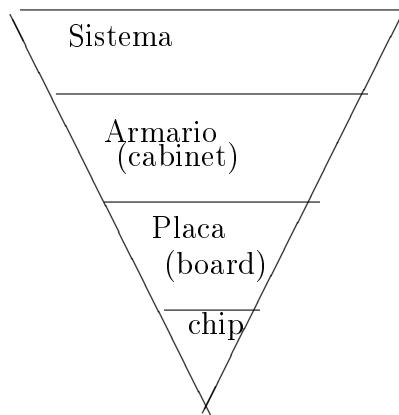
- ♡ **Multiprocesador con memoria centralizada (UMA)**: el acceso a memoria en los procesadores tarda el mismo tiempo independientemente de la dirección de memoria a la que se acceda y desde el procesador desde el que se acceda, esto quiere decir que todos los procesadores se encuentran a la misma distancia de las direcciones de memoria. Se accede a memoria a través de una red de interconexión que puede ser un bus, una multietapa o barras cruzadas.

Esta alternativa presenta una mayor latencia por el hecho de que todos los procesadores acceden a la memoria de manera compartida, por tanto, es bastante probable que haya colisiones (dos procesadores quieren acceder a la misma posición de memoria). También presentan poca escalabilidad puesto que cuanto mayor sea el número de procesadores mayor será el número de colisiones en el acceso a memoria.

- ♡ **Multiprocesador con memoria distribuida (NUMA)**: el acceso a memoria no es uniforme porque la memoria está distribuida entre los nodos de procesamiento. Cada uno de los procesadores tiene cerca un módulo de memoria, por tanto puede acceder a las direcciones de memoria que tiene cerca más rápido que las direcciones de memoria que tiene lejos. Esta alternativa es más escalable, puede llegar a cientos/miles de cores porque cada procesador accede normalmente a las posiciones de memoria que tiene más cerca suyo y por tanto, el número de colisiones es muy pequeño. Los diferentes tipos de red de interconexión para acceder a la memoria de los demás procesadores suelen ser barras cruzadas, multietapa, árbol grueso y anillo.

Criterio de clasificación: nivel de empaquetamiento/conexión

Como dijimos antes, los multiprocesadores pueden venir presentados de diferentes formas:



Cada una de las diferentes formas está formada por la forma anterior, así, por ejemplo, un sistema será un conjunto de armarios, un armario será un conjunto de placas y una placa será un conjunto de chips.

Evolución de multiprocesadores en una placa

Han evolucionado de UMA a NUMA.

En un UMA, la comunicación de la CPU con la memoria se hacía a través de un bus y el chipset (conjunto de chips que se usa para la conexión de la CPU con elementos externos: suele estar formado por dos chips, el *puente norte*, que conecta cpu con memoria y graficos, y el *puente sur*, que conecta CPU con dispositivos lentos).

Ahora lo que se suele ver más son estructuras NUMA: los chips de procesamiento están conectados directamente a la memoria. Ya no tenemos buses, sino conmutadores y enlaces. La interfaz de red está incluida en los chips junto a los cores. El puente norte ahora sólo está para la conexión con los gráficos. Aunque hay PCs que tienen directamente los gráficos conectados, en ese caso no habría puente norte y el chipset estaría formado por uno sólo, el puente sur.

La estructura de un *Intel Xeon 7300* (Figura ??) es una estructura intermedia entre las dos con un bus para cada CPU. Se corresponde con una placa de tipo UMA.

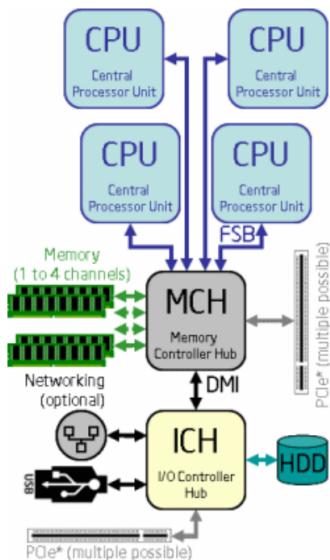


Figure 3.1: Estructura de un Intel Xeon 7300

Y en la Figura ?? nos encontramos una estructura más actual, de tipo NUMA. La red de interconexión que se usa para conectar los chips de procesamiento conecta los módulos de memoria de cada procesador dos a dos.

3.7.3 Multicores

Hay varias variantes para el diseño de multicores, que difieren en el número de cores, la estructura y tamaño de las cachés, el acceso de los cores a las cachés y la heterogeneidad de sus componentes.

Una posible estructura sería una en la que todos los cores compartieran al menos el último nivel de caché (Figura ??). Así, los cores pueden comunicarse con ellos a través de este último nivel

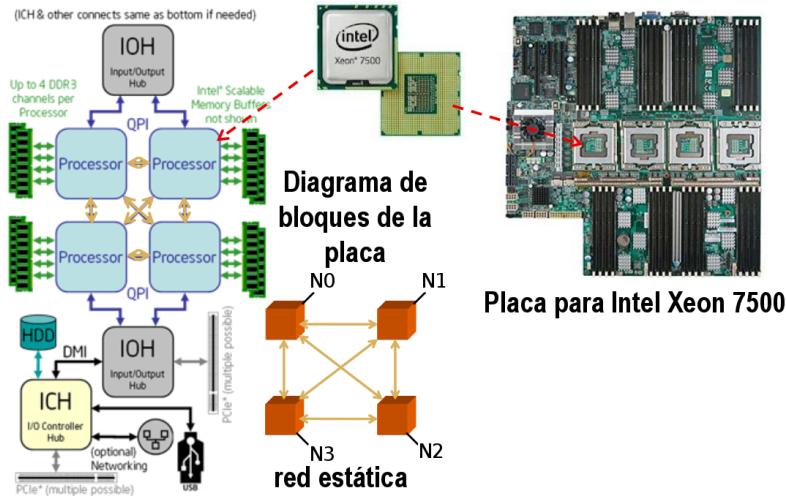


Figure 3.2: Estructura de un Intel Xeon 7500

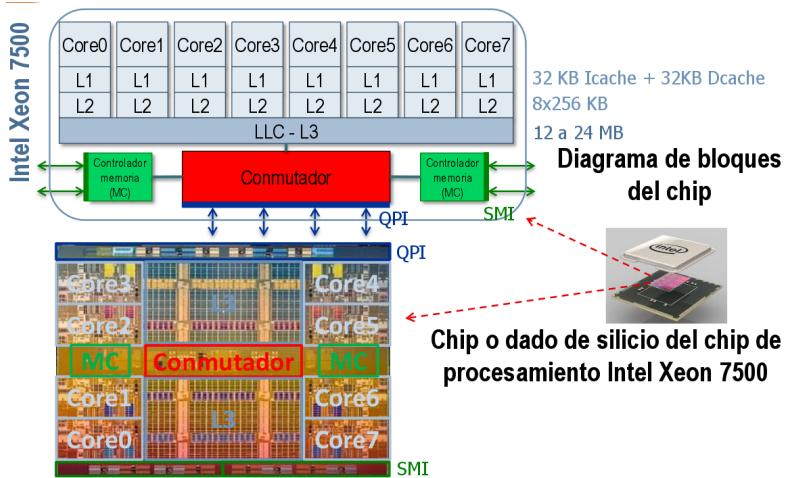


Figure 3.3: Estructura en forma de jerarquía

de caché sin acceder a memoria principal.

Otras posibles estructuras (Figura ??) pueden ser, por ejemplo, hacer que la mitad de los procesadores comparten su último nivel de caché y la otra mitad comparta un último nivel de caché diferente (Figura ??). En este caso, para implementar la comunicación entre cores tendremos que procurar que se comuniquen aquellos que comparten el último nivel de caché.

O directamente, no compartir ningún nivel de caché entre cores, para lo cual tendremos que acceder a memoria principal si queremos comunicarlos (Figura ??).

También podríamos poner un conmutador en entre los cores y el último nivel de caché compartido para regular el acceso al último nivel de caché (Figura ??)

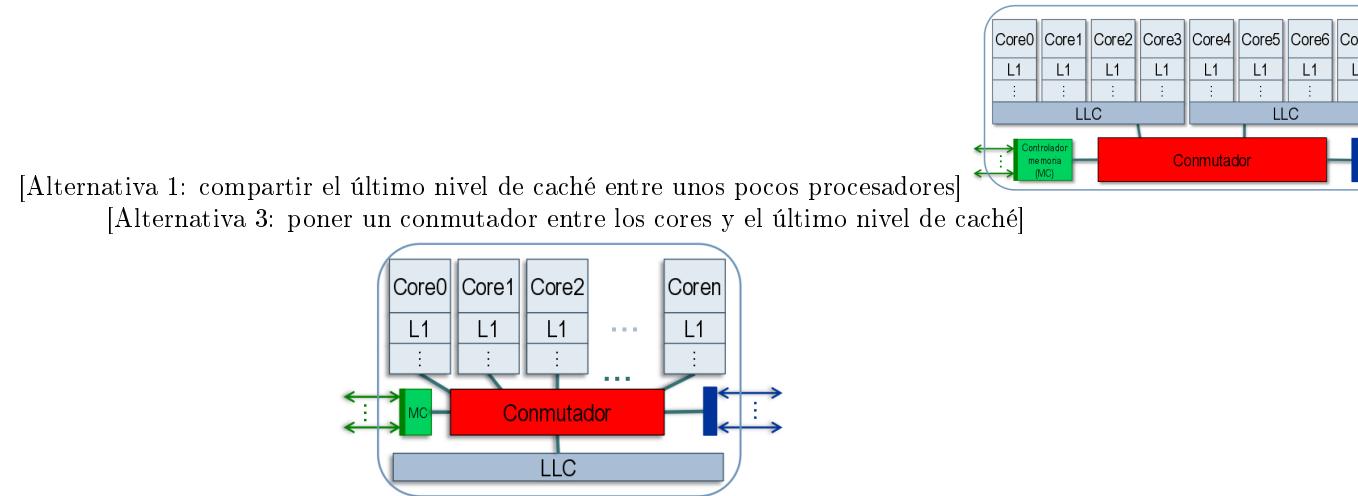


Figure 3.4: Alternativas de estructuración de un multicore

3.7.4 Cores Multithread

Arquitecturas ILP

La arquitectura ILP puede ser de tres tipos: segmentada, VLIW y superescalar.

En un procesador segmentado se insertan registros para crear un cauce. En un RISC es típico un cauce de cinco etapas:

- ♡ **IF (Instruction Fetch)**: etapa de captación de instrucciones. De la caché de instrucciones se coge la siguiente instrucción a ejecutar y se guarda en un registro intermedio (*Instruction Buffer*).
- ♡ **ID (Instruction Decode)**: etapa de decodificación de instrucciones y emisión a unidades funcionales. Se decodifica la instrucción y se guarda en un registro intermedio.
- ♡ **EX (Execution)**: etapa de ejecución. Se ejecuta la instrucción y se guardan los resultados en un registro intermedio.
- ♡ **MEM (Memory)**: etapa de acceso a memoria. Se recogen los resultados de dicho registro intermedio y se guardan en memoria.
- ♡ **WB (Write-Back)**: etapa de almacenamiento de resultados. Se modifican los registros de la arquitectura (tales como `rax`).

Un procesador segmentado permite ejecutar instrucciones concurrentemente, en la Figura ?? 5 instrucciones concurrentes cada una en una etapa del cauce. Es concurrente porque todas comparten el mismo hardware.



Figure 3.5: Arquitectura escalar segmentada

En las otras dos alternativas podemos tener instrucciones ejecutándose a la vez porque la unidad dispone de varias unidades funcionales a las que se les pueden emitir múltiples instrucciones en paralelo a las distintas unidades funcionales

Hay una unidad para punto fijo, otra para punto flotante, otra para cargar y guardar datos, etc. La unidad de punto flotante suele estar segmentada al igual que la unidad de memoria (calcular la dirección y acceder). Para decir que un core es superescalar, debe emitir varias instrucciones en el mismo ciclo de reloj a cada una de las unidades de ejecución.

Las diferencias entre VLIW y superescalar son:

♡ VLIW:

- Las instrucciones que se ejecutan en paralelo se captan juntas de memoria
- Este conjunto de instrucciones conforman la palabra de instrucción muy larga a la que hace referencia la denominación VLIW
- El hardware presupone que las instrucciones de una palabra son independientes: no tiene que encontrar instrucciones que pueden emitirse y ejecutarse en paralelo.

♡ Superescalares:

- Tiene que encontrar instrucciones que puedan emitirse y ejecutarse en paralelo (tiene hardware para extraer paralelismo a nivel de instrucción).

Es decir, las instrucciones en VLIW se captan juntas en la memoria, porque no hay hardware para evitar problemas (WAW, WAR, RAW). Los superescalares, tienen hardware para eliminar problemas de paralelismo.

Son segmentados también: ejecutan instrucciones concurrentemente y en paralelo.

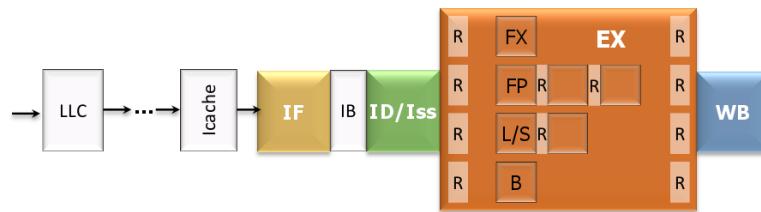


Figure 3.6: Arquitecturas VLIW y superescalar

3.7.5 Modificación de la arquitectura ILP en Core Multithread

Las modificaciones hardware permiten a los threads multiplexar (unas veces en un ciclo de reloj lo usa un thread y en otro ciclo de reloj otro) el uso del hardware, repartir o compartir (dinámicamente se va asignando los recursos a cada thread en función de sus necesidades, con la alternativa de repartir, si un thread tiene asignado dos recursos pero sólo usa uno, el recurso que no usa no puede ser usado por otro thread.) entre cada thread, por último, se puede replicar un recurso (en los cores multithread se suele replicar el fichero de registro de la arquitectura: uno por thread).

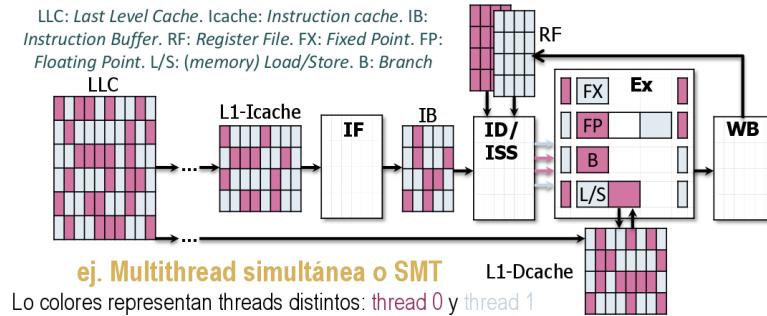


Figure 3.7: Modificación de la arquitectura ILP en un core multithread

Por ejemplo, en la Figura ??, se han repartido las cachés (tanto de datos como de instrucciones) y se ha replicado el fichero de registro, uno para cada thread.

3.7.6 Clasificación de cores multithread

♡ **Multihread temporal:** se ejecutan concurrentemente, deben hacer un uso mayor de multiplexación. Aquí hay conmutación entre threads porque en un ciclo de reloj no hay varios threads ejecutándose a la vez. La conmutación la controla el hardware, no el sistema operativo. Hay varios tipos:

- **Grano fino:** cada ciclo de reloj se cambia de thread, la conmutación se puede hacer con turnos rotatorios (no hay decisión sobre a qué thread se va a comutar) con un hardware muy sencillo (no tiene hardware para eliminar dependencias de datos, el número de etapas se fija para que no haya dependencias de WAR y WAW. Si tenemos menos threads de los predeterminados perdemos ciclos de reloj). También se puede comutar a otros threads por eventos¹ de latencia combinado con alguna técnica de planificación
- **Grano grueso:** se comuta entre threads pero no cada ciclo de reloj, la conmutación entre threads la decide el hardware tras intervalos de tiempo prefijados (*timeslice multithreading*) ó por eventos de cierta latencia (*switch-on-event multithreading*). Los granos gruesos a su vez se clasifican según su conmutación por eventos:
 - **Estática:** la conmutación puede hacerse *explícita* (instrucciones explícitas para la conmutación, el programador o el compilador pueden definir de forma explícita cuando cambiar de contexto) ó *implícita* con instrucciones de carga, almacenamiento o salto. El coste del cambio de contexto es bajo, pero se hacen cambios de contexto innecesarios.

¹dependencia funcional, acceso a datos a cache L1, salto no predecible, operación con cierta latencia

- **Dinámica:** la commutación se hace cuando es necesario por un fallo en la última caché dentro del chip de procesamiento (comutación por fallo de cache), interrupción (comutación por señal),... Reduce cambios de contexto innecesarios pero tiene una mayor sobrecarga al cambiar de contexto. En este caso, los cambios de contexto son acciones propias de la instrucción.

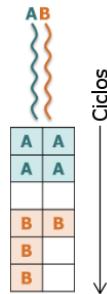


Figure 3.8: Esquema de dos threads ejecutándose en un multithread temporal

La Figura ?? refleja la ejecución de dos threads en un multithread temporal: ambos se van conmutando según indique el hardware. En el caso de B, deja sin usar un recurso durante tres ciclos de reloj pues no lo necesita.

- ♡ **Simultaneo:** se ejecutan, en un core superescalar, varios threads en paralelo y comparten la unidad de ejecución. Pueden emitir (para su ejecución) instrucciones de varios threads en un ciclo.

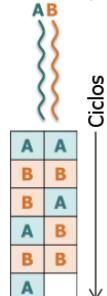


Figure 3.9: Esquema de dos threads ejecutándose en un multithread simultáneo

La Figura ?? muestra el esquema de dos threads ejecutándose en un thread simultáneo: ambos se van ejecutando paralelamente y así conseguimos un mayor aprovechamiento de los recursos.

El objetivo de los cores multithreads es que las pérdidas de tiempo debido a accesos a memoria, dependencias RAW, etc. no sean visibles durante la ejecución de la instrucción, sino que si un thread falla o tiene que acceder a memoria, pueda seguir la ejecución otro thread, ya sea de forma concurrente o paralela, para evitar pérdidas de ciclos de reloj por esperar para poder seguir con la ejecución. Estos ciclos se evitan ejecutando instrucciones de threads distintos.

En un thread se pueden producir:

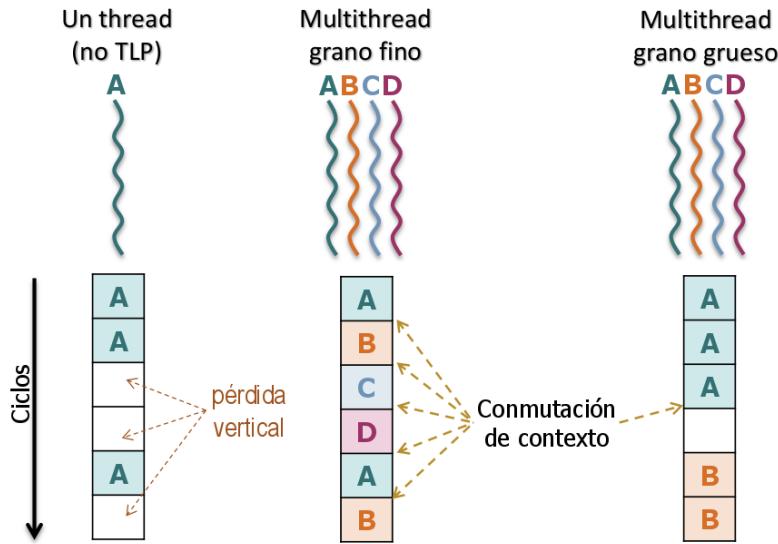


Figure 3.10: Alternativas en un core escalar segmentado

- ♡ *Pérdidas verticales*: eliminables con threads de grano grueso y fino
- ♡ *Pérdidas horizontales*: no eliminables porque no se pueden ejecutar a la vez instrucciones de threads diferentes. Se pueden evitar con threads simultáneos gracias al multithread, ya que en el mismo ciclo se pueden ejecutar instrucciones de threads distintos.

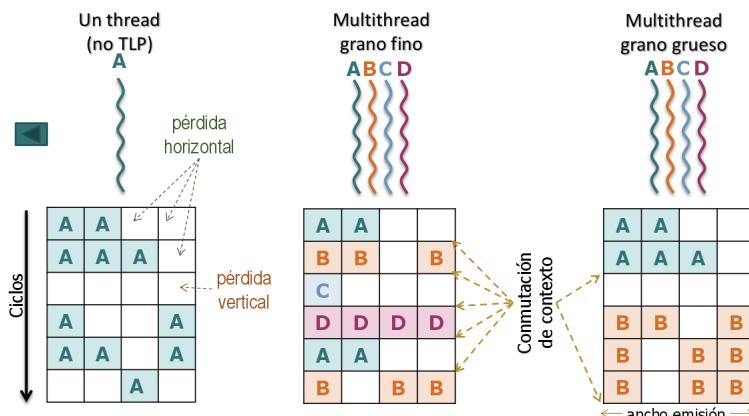


Figure 3.11: Alternativas en un core con emisión múltiple de instrucciones de un thread

En la Figura ?? podemos ver una ejecución de cuatro procesos en un core de cuatro threads y la misma ejecución en dos cores con dos threads. ¿Si las dos pueden ejecutar cuatro threads en paralelo, cuál de las dos es más ventajosa?

En la opción de la izquierda, para que haya 4 threads simultáneos, necesitan darse cuatro instrucciones diferentes para ejecutar (es decir, que se necesite usar el registro de punto fijo, punto

flotante, de salto y acceso a memoria), lo que puede dar lugar a pérdidas horizontales. Sin embargo, en la segunda opción, ejecutar todos los threads es más fácil porque sólo tenemos que tener en cuenta dos instrucciones por core al haber sólo dos threads por core.

Respecto a la replicación, algunos diseños de grano grueso sólo replican el contador de programa, pero en la mayoría de los casos, se replica todo, es decir, los registros.

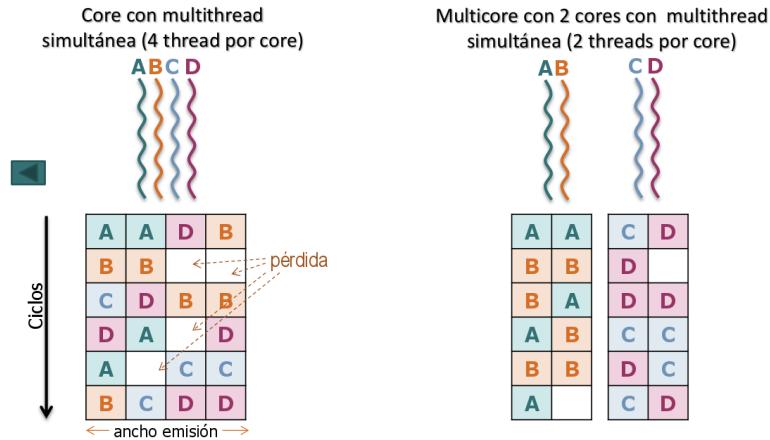


Figure 3.12: Core multithread simultánea y multicore

3.8 Coherencia del sistema de memoria

En los CC-NUMA, los UMA y los COMA se implementa coherencia por hardware

3.8.1 Sistema de memoria en multiprocesadores

El sistema de memoria incluye las caches de todos los nodos, la memoria principal, los buffers (tanto de escritura/almacenamiento como los buffer que combinan escrituras/almacenamientos) y la red de interconexión para acceder desde un procesador a los módulos de memoria directamente conectados a él y para acceder a direcciones remotas, conectadas a otro procesador. En resumen, el sistema de memoria incluye todo lo resaltado en rojo de la figura Figura ??.

Dentro del chip de procesador está incluido el conmutador, los niveles de cache y los buffers.

La comunicación de datos entre procesadores la realiza el sistema de memoria. La lectura de una dirección debe devolver lo último que se ha escrito, desde el punto de vista de todos los componentes del sistema.

3.8.2 Concepto de incoherencia

La utilización de jerarquía de memoria, con el fin de acercar la velocidad de acceso a memoria a la velocidad del procesador, posibilita que pueda haber varias copias de un mismo bloque

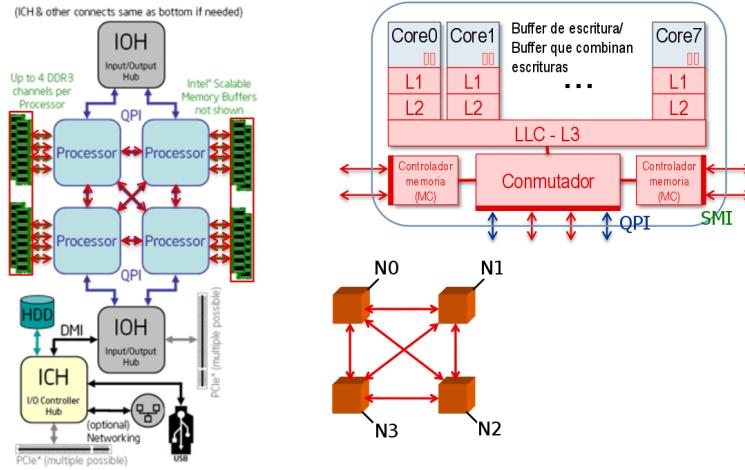


Figure 3.13: Sistema de memoria

de memoria. Si dichas copias no tienen el mismo contenido, tendremos una *incoherencia en el sistema de memoria*.

Se pueden dar situaciones de incoherencia entre la memoria principal y las caches, o entre las distintas caches:

Clases de estructuras de datos	Causa de problemas por falta de coherencia	Falta de coherencia
Datos modificables	E/S	Cache–MP
Datos modificables compartidos	Fallo de cache	Cache–MP
Datos modificables privados	Emigra proceso → Fallo de cache	Cache–MP
Datos modificables compartidos	Lectura de cache no actualizada	Cache–Cache

En el ejemplo de la Figura ??, el procesador P_k lee de la posición de memoria D (1L), lo que provoca que el bloque de memoria donde se encuentra esta dirección se copie a su cache. Tras la transferencia, el contenido de la posición de memoria D en la cache es 3, coincide con el contenido de la dirección en memoria principal. Si a continuación P_k escribe un nuevo valor, 4, en la dirección D (2E), escribirá en la copia que tiene en su cache y dará lugar a una incoherencia en el sistema de memoria, ya que la posición D no tiene el mismo contenido en memoria principal y en la caché.

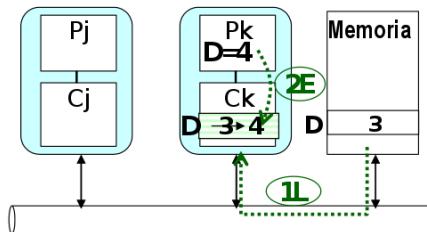


Figure 3.14: Incoherencia entre memoria principal y cache. L denota lectura y E escritura. Mediante un número se indica el orden entre operaciones

La memoria principal se suele dividir en bloques, que suelen ser de 64 bytes. Las caches también están divididas en bloques del mismo tamaño, llamados marcos de bloque. La unidad de acceso a memoria es un bloque. Si transferimos el bloque tenemos mayor velocidad de transferencia que transfiriendo palabras.

Métodos de actualización de memoria principal implementados en caches

Para resolver las incoherencias entre memoria principal y caches hay dos métodos:

- ♡ **Escritura inmediata:** nunca habrá incoherencia porque al escribir en un bloque de cache se escribe en memoria principal. Cada escritura supone la utilización de la red y supone además un desaprovechamiento del ancho de banda de la red al transferir sólo datos aislados. Además, cuando se escribe en una dirección de memoria, suele escribirse varias veces en dicha dirección o en direcciones cercanas, por tanto, si la cache implementa esta alternativa tendríamos que hacer 64 transferencias a través de la red. Sería más deseable transferir todo el bloque de una vez.
- ♡ **Posescritura:** se actualiza la memoria principal cuando se reemplaza el bloque en cache. Cuando se va a traer un bloque desde memoria principal y no queda espacio en cache, se reemplaza uno de los bloques que hay en cache. Si el bloque a reemplazar ha sido modificado (su “bit sucio” está activo), se escribe en memoria principal. Reduce comunicaciones pero permite que haya incoherencia, eso sí, con un sistema hardware que devuelve los valores actualizados de dicha posición de memoria. Para saber que un bloque ha sido modificado en la cache, se guarda en una tabla qué bloques de memoria principal están en cache y, para cada bloque, se guarda información sobre su estado, más concretamente si se ha modificado o no (*bit sucio*)

Alternativas para propagar una escritura en protocolos de coherencia de cache

En las caches no se admite incoherencia. Siempre se propaga lo que se escribe en una cache al resto. Para realizar dicha propagación tenemos dos alternativas:

- ♡ **Escritura con actualización:** siempre que se modifica una dirección en la copia de un bloque en la cache de un procesador, se modifica la dirección en todas las copias del bloque que se encuentren en todas las caches de otro procesadores.
- ♡ **Escritura con invalidación:** cuando se va a modificar una dirección en la cache de un procesador, primero se invalidan las copias del bloque que contiene esa dirección en otras caches. Así, el procesador que va a modificar la dirección obtiene acceso exclusivo al bloque que la contiene. Cuando otra cache quiera leer ese dato tendrá que acceder a memoria principal obteniendo así una copia actualizada. Invalidar es más rápido que actualizar porque al invalidar sólo transferimos la dirección en la que se va a escribir mientras que al actualizar se deben transferir también los datos. Además, la invalidación permite compartir un bloque de memoria mientras se lee del bloque. También se reduce el número de transferencias puesto que si el bloque vuelve a modificar la dirección no originará transferencias. Sin embargo, si se va a actualizar un dato para que lo lean los demás procesadores puede ser más eficiente la actualización.

Situación de incoherencia aunque se propagan las escrituras (se usa difusión)

La propagación sólo sirve si usamos buses, pues no pueden hacerse accesos a memoria en paralelo. Sin embargo, hoy en día apenas se usan buses. Por ello, aun usando difusión podemos llegar a una situación de incoherencia.

Por ejemplo, el procesador de la Figura ?? tiene una variable A con valor inicial 0 y todas las caches tienen una copia de dicha variable.

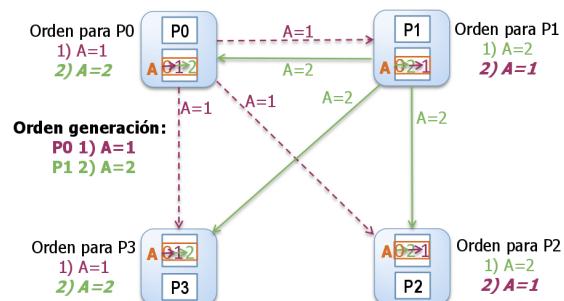


Figure 3.15: Situación de incoherencia usando difusión

Como el tiempo de transferencia depende de la longitud del enlace, el tiempo de transferencia entre $P0$ y $P2$ será mayor que entre $P0$ y $P1$.

P_0 escribe en A un 1 y después, P_1 escribe en A un 2. La actualización del valor de A llega en distinto tiempo debido al tiempo de propagación distinto y, por tanto, P_0 y P_3 finalmente ven $A = 2$ mientras que P_2 y P_1 ven $A = 1$. Por tanto, tenemos una situación de incoherencia.

Requisitos del sistema de memoria para evitar problemas por incoherencia

Para garantizar la coherencia se deben garantizar:

Propagar las escrituras en una dirección : la escritura en una dirección debe hacerse visible en un tiempo finito a otros procesadores.

- ♡ Si los componentes están conectados mediante un bus, los paquetes de actualización o invalidación son visibles a todos los nodos conectados al bus (controladores de cache).
- ♡ Si los componentes no están conectados mediante un bus y usamos difusión, los paquetes de actualización ó invalidación se envían a todas las caches. Para conseguir mayor escalabilidad debería enviar paquetes de actualización ó invalidación sólamente a caches con copia del bloque, ya que enviar una copia a nodos que no la van a necesitar ocupa enlaces de forma innecesaria. Para saber qué caches tienen una copia de un bloque, se mantiene en un directorio, para cada bloque, los nodos con copia del mismo.

Serializar las escrituras en una dirección : las escrituras en una dirección deben verse en el mismo orden por todos los procesadores (el sistema de memoria debe parecer que realiza en serie las operaciones de escritura en la misma dirección).

- ♡ Si los componentes están conectados con un bus, el orden en que los paquetes aparecen en el bus determina el orden en que se ven por todos los nodos.
- ♡ Si los componentes no están conectados con un bus, el orden en el que las peticiones de escritura llegan a su *home* (nodo que tiene en la memoria principal la dirección) o al directorio centralizado sirve para serializar en sistemas de comunicación que garantizan el orden en las transferencias entre dos puntos.

Directorio de memoria principal

El directorio informa sobre qué nodos tienen una copia de un bloque en memoria en forma de vector. Si el procesador i tiene una copia, la posición i del vector de bits tendrá un 1, y si no la tiene, tendrá un 0. También se guarda el estado del bloque de memoria, 0 si no ha sido modificado y 1 si se ha modificado.

Alternativas para implementar el directorio

Las distintas formas de implementar un directorio son:

- ♡ **Centralizado**: el directorio es compartido por todos los nodos y contiene información de todos los bloques de todos los módulos de memoria.

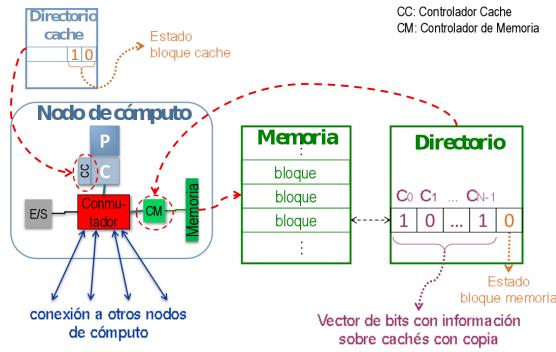


Figure 3.16: Estructura de un directorio

♡ **Distribuido:** las filas se distribuyen entre los nodos. Típicamente, el directorio de un nodo contiene información de los bloques de sus módulos de memoria. Esta es la alternativa más usada.



Figure 3.17: Alternativas para implementar los directorios

El orden en el que se realizan las peticiones de escritura sobre un bloque será el orden real en el que harán los procesadores los accesos a memoria.

Serialización de las escrituras por el home. Usando difusión

Siguiendo con el ejemplo de la Figura ??, cuando llega una petición de escritura al home, esta se confirma a todos los demás nodos. Así, todos ven las operaciones en el mismo orden. Es decir, el home difunde el nuevo contenido de esa dirección de memoria a todos los demás nodos.

Primero se generaría $A = 1$, pero como P_0 está más lejos que P_1 (que genera $A = 2$) de P_2 que actúa como home, primero llega $A = 2$, se confirma $A = 2$ y, después, llegaría $A = 1$ y se confirmaría $A = 1$. Lo importante es que todos los nodos al final ven a A con el mismo valor.

Serialización de las escrituras por el home. Sin difusión y con directorio distribuido

En este caso, el procesador P_3 de la Figura ?? no tiene una copia del valor A , es decir, la casilla C_3 del directorio tendrá un 0 y por tanto, no se enviarían confirmaciones del cambio de valor de A a P_3 .

En el resto, sería todo lo mismo. Confirmar los distintos valores de A que van llegando a los nodos que tengan una copia de A .

3.8.3 Protocolos de mantenimiento de coherencia: clasificación y diseño

Clasificación de protocolos para mantener coherencia en el sistema de memoria

- ♡ **Protocolos de espionaje (snoopy)**: para buses, y en general sistemas con una difusión eficiente (bien porque el número de nodos es pequeño o porque la red implementa difusión). Aprovechan que todos los componentes conectados al bus pueden ver (espiar) el contenido del bus. El controlador de cache espía los paquetes del bus y actúa en consecuencia.
- ♡ **Protocolos basados en directorios**: para redes sin difusión o escalables (multietapa y estáticas)
- ♡ **Esquemas jerárquicos**: para redes jerárquicas: jerarquía de buses, jerarquía de redes escalables, redes escalables-buses.

Facetas de diseño lógico en protocolos para coherencia

- ♡ *Política de actuación de memoria principal*: escritura inmediata, posescritura (la más usada) y mixta.
- ♡ *Política de coherencia entre caches*: escritura con invalidación (la más usada), escritura con actualización y mixta.

Debemos definir el comportamiento de nuestro protocolo de coherencia:

- ♡ Definir posibles estados de los bloques en cache, y en memoria.
- ♡ Definir transferencias (indicando nodos que intervienen y orden entre ellas) a generar ante eventos: lecturas o escrituras del procesador del nodo ó como consecuencia de la llegada de paquetes de otros nodos
- ♡ Definir transiciones de estados para un bloque en cache, y en memoria.

3.8.4 Protocolo MSI de espionaje

Los protocolos MSI se basan en posescritura e invalidación. Cada bloque en MSI puede tener tres estados diferentes en cache:

- ♡ **Modificado (M)**: es la única copia válida del bloque en todo el sistema.

- ♡ **Compartido, Shared (C,S)**: el bloque está válido (sin modificar) y puede haber copias de dicho bloque en otras caches.
- ♡ **Inválido (I)**: se ha invalidado el bloque pues ha sido modificado por un nodo o no está físicamente.

Cada bloque en memoria puede tener estos estados (en realidad se evita almacenar esta información):

- ♡ **Válido**: puede haber copia válida (sin modificar) en una o varias caches
- ♡ **Inválido**: habrá una copia válida en la cache del nodo que ha modificado dicho bloque.

Protocolo de espionaje de tres estados (MSI)

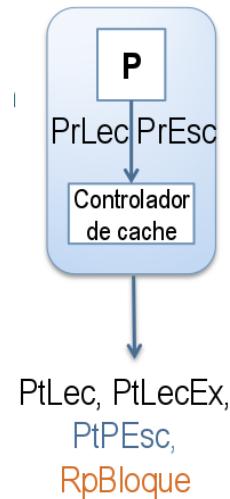


Figure 3.18: Tipos de paquetes MSI

Las transferencias generadas por un nodo con cache, es decir, los tipos de paquetes, son:

- ♡ **Petición de lectura de un bloque (PtLec)**: Cuando un nodo quiere leer un bloque y no lo tiene disponible en su cache (**PrLec**) genera un paquete de petición de lectura de dicho bloque (**PtLec**).
- ♡ **Petición de acceso exclusivo (PtLecEx)**: Cuando un nodo quiere escribir en un bloque (**PrEsc**) y no tiene dicho bloque disponible en su cache, genera un paquete de petición de acceso exclusivo para modificar dicho bloque (**PtLecEx**)
- ♡ **Petición de posescritura (PtPEsc)**: cuando se va a reemplazar el bloque en cache se genera un paquete de este tipo para actualizar su contenido en memoria.
- ♡ **Respuesta con bloque (RpBloque)**: cuando se recibe una petición de acceso exclusivo o de lectura, se debe responder con el bloque solicitado.

Transiciones de estados

La siguiente tabla representa todas las transiciones de estados MSI:

Estado Actual	Evento	Acción	Siguiente
Modificado (M)	PrLec/PrEsc		Modificado
	PtLec	Genera paquete respuesta (RpBloque)	Compartido
	PtLecEx	Genera paquete respuesta (RpBloque). Invalida copia local	Inválido
	Reemplazo	Genera paquete posescritura (PtEsc)	Inválido
Compartido (S)	PrLec		Compartido
	PrEsc	Genera Paquete PtLecEx	Modificado
	PtLec		Compartido
	PtLecEx	Invalida Copia Local	Inválido
Inválido (I)	PrLec	Genera paquete PtLec	Compartido
	PrEsc	Genera paquete PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

Cuando tenemos un bloque modificado y el propio nodo va a leerlo o a modificarlo, al ser la única copia válida en el sistema, el bloque no cambia de estado pues sigue estando en el propio nodo que lo modificó.

En cambio, un nodo envía una petición de lectura, se genera un paquete respuesta con el bloque y éste pasa a estado compartido, pues está como mínimo en una cache. Como el bloque está modificado el nodo que tiene la única copia válida del bloque responde con dicha copia.

Si un bloque está modificado y un nodo envía una petición de acceso exclusivo para modificarlo, el nodo debe primero invalidar su copia, pues el bloque se va a modificar, y enviar el bloque.

Si un bloque en cache esta modificado y se va a reemplazar por otro, se genera un paquete de posescritura y dicho bloque pasa a estar en estado inválido, pues no se encontrará físicamente en dicho nodo.

Cuando un bloque está compartido y el nodo que lo tiene va a leerlo, sigue estando compartido y no se genera ninguna acción, pues al estar el bloque en dicho nodo no hace falta ni siquiera generar un paquete respuesta.

En cambio, si un nodo está compartido y el nodo va a modificarlo, genera un paquete de acceso exclusivo invalidando las copias del resto de nodos y éste pasa a estado modificado.

Si un paquete está compartido y un nodo que no dispone de dicho paquete genera una petición de lectura, al no estar modificado se deja que la memoria principal responda con dicho bloque y sigue estando en estado compartido.

Cuando un bloque compartido recibe una petición de acceso exclusivo, se invalida la copia local de dicho bloque y se deja que la memoria envíe dicho bloque al nodo que lo ha solicitado.

Si un bloque está inválido y el nodo quiere leerlo, genera un paquete de petición de lectura y el bloque pasa a estado compartido.

Si un bloque está en estado inválido y el nodo quiere escribirlo, genera una petición de acceso exclusivo, se invalida el bloque en todos los nodos menos en ese y el bloque pasa a estado modificado.

Y por último, si el bloque está inválido y el nodo recibe una petición de lectura o de acceso exclusivo, no hace nada pues él no tiene el bloque y se queda en inválido.

3.8.5 Protocolo MESI de espionaje

Cuando sólo ejecutamos código secuencial, solo habrá copias de un código en una cache. Por tanto, el paquete para invalidar copias de otras caches sería inútil. Esto Intel lo resuelve con el protocolo MESI.

El protocolo MESI divide el estado *Shared* en dos:

- ♡ **Exclusivo (E)**: el nodo tiene la única copia válida en cache y la memoria está actualizada.
- ♡ **Compartido (S)**: el bloque está en varias caches y está válido en todas, es decir, la memoria está actualizada.

Así, si un bloque que estaba en estado Exclusivo se modifica, no haría invalidar nada.

Para saber si un bloque está en una cache o en varias se guarda en un vector los nodos que tienen copia de dicho bloque y luego se hace una operación OR.

Transiciones de estados

A la tabla de transiciones MSI se le añade el estado Exclusivo y una fila en Inválido:

Estado Actual	Evento	Acción	Siguiente
Modificado (M)	PrLec/PrEsc		Modificado
	PtLec	Genera paquete respuesta (RpBloque)	Compartido
	PtLecEx	Genera paquete respuesta (RpBloque). Invalida copia local	Inválido
	Reemplazo	Genera paquete posescritura (PtEsc)	Inválido
Exclusivo (E)	PrLec		Exclusivo
	PrEsc		Modificado
	PtLec		Compartido
	PtLecEx	Invalida copia local	Inválido
Compartido (S)	PrLec		Compartido
	PrEsc	Genera Paquete PtLecEx	Modificado
	PtLec		Compartido
	PtLecEx	Invalida Copia Local	Inválido
Inválido (I)	PrLec($C = 1$)	Genera paquete PtLec	Compartido
	PrLec($C = 0$)	Genera PtLec	Exclusivo
	PrEsc	Genera paquete PtLecEx	Modificado
	PtLec/PtLecEx		Inválido

Si una copia exclusiva recibe una petición de lectura, deja que la memoria responda con dicha petición y no hace nada.

Si recibe una petición de acceso exclusivo, invalida su copia local pero deja que la memoria responda con dicha petición.

Y si una copia inválida recibe una petición de lectura y dicha copia no está en ninguna otra cache, genera el paquete de petición de lectura y dicho bloque pasa a estado Exclusivo.

Ejercicio 1 - Relación ejercicios tema 3

En un multiprocesador SMP con 4 procesadores o nodos (N0-N3) basado en un bus, que implementa protocolo MESI para mantener la coherencia, supongamos una dirección de memoria incluida en un bloque que no se encuentre en ninguna cache. Indique los estados de este bloque en las caches y las acciones que se producen en el sistema en la siguiente secuencia de eventos para dicha dirección:

1. Lectura generada por el procesador 1
2. Lectura generada por el procesador 2
3. Escritura generada por el procesador 1
4. Escritura generada por el procesador 2
5. Escritura generada por el procesador 3

Evento	Acción generada	Estado siguiente
PrLec ($P = 1$)	La memoria responde con el bloque de memoria (RpBloque) y se comprueba que dicho bloque no está en ninguna otra cache	Exclusivo en P1 e Invalido en P2 y P3
PrLec ($P = 2$)	La memoria responde con el bloque de memoria (RpBloque) y se comprueba si dicho bloque está en otra caché.	Compartido en P1 y P2 e Inválido en P3
PrEsc ($P = 1$)	Se invalida la copia del procesador dos para que el procesador uno pueda modificar el bloque	Modificado en P1 e inválido en P2 y P3
PrEsc ($P = 2$)	Se inhibe la respuesta de la memoria pues tiene una copia inválida. Responde el primer procesador con el bloque y actualiza la memoria, después, el procesador uno invalida su copia local y para que el dos pueda modificarlo.	Modificado en P2 e inválido en P1 y P2
PrLecEx ($P = 3$)	El procesador dos responde con el bloque e invalida su copia local. Ahora la única copia válida se encuentra en el procesador tres	Modificado en P3 e inválido en P1 y P2

3.8.6 Protocolo MSI basado en directorios con o sin difusión

MSI con directorios sin difusión

Al igual que en el protocolo MSI de espionaje, los estados de un bloque en cache pueden ser Modificado (M), Compartido (S) o Inválido (I) y los estados de un bloque en memoria principal pueden ser válido e inválido.

En las transferencias, hay varios tipos de nodos:

- ♡ **Solicitante (S)**: No tienen una copia válida del bloque en su caché
- ♡ **Origen (O)**: También conocidos como *home*. Tienen el bloque en su memoria principal (los directorios se usan en sistemas con memoria distribuida).
- ♡ **Propietario (P)**: Tienen una copia válida del bloque en su cache.
- ♡ **Modificado (M)**
- ♡ **Compartidor (C)**

Además, hay tres tipos de paquetes:

- ♡ **Petición**: petición de lectura (*PtLec*), petición de lectura con acceso exclusivo (*PtLecEx*), petición de acceso exclusivo (*PtEx*) y postescritura (*PtPesc*)
- ♡ **Respuesta**: respuesta con bloque (*RpBloque*), respuesta con o sin bloque confirmando invalidación (*RpInv*, *RpBloqueInv*)
- ♡ **Reenvío**: reenvío de invalidación (*RvInv*), reenvío de lectura (*RvLec*, *RvLecEx*). Este tipo de peticiones sirven para cuando el nodo home no puede responder, debido a que tiene el bloque en estado inválido, reenviar la petición a un bloque que sí pueda responder con el bloque.

Los tipos de comunicaciones que se pueden hacer entre los distintos tipos de nodos se puede ver en la Figura ???. Un nodo solicitante puede hacer una petición al nodo origen para acceder al bloque (lectura, acceso exclusivo, lectura y acceso exclusivo, postescritura). El nodo origen, si no dispone de dicho bloque, reenviará la petición a un nodo propietario y éste responderá a dicha petición al nodo origen que responderá al nodo solicitante.

Por tanto, un acceso puede suponer como máximo cuatro envíos de paquetes. Si usamos difusión sólo serían necesarios tres envíos de paquetes.

Como ejemplo, haremos el ejercicio 3 de la relación de problemas del tema 3.

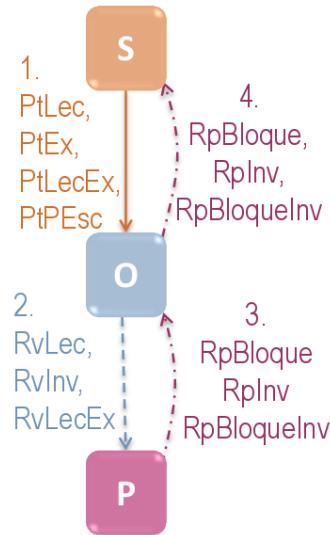


Figure 3.19: Comunicaciones a través de la red entre los distintos tipos de nodos

Ejercicio 3 - Relación de ejercicios tema 3

Suponga que en un CC-NUMA de red estática de 4 nodos (N0-N3) se implementa un protocolo MSI basado en directorios sin difusión con dos estados en el directorio (válido e inválido). Cada nodo tiene 8GB de memoria y una línea de cache supone 64B. Considere que el directorio utiliza vector de bits completo.

- Calcule el tamaño del directorio de un nodo en bytes
- Indique cuál sería el contenido del directorio, las transiciones de estados (en cache y directorio) y la secuencia de paquetes generados por el protocolo de coherencia en los siguientes accesos sobre una dirección D que se encuentra en la memoria del nodo 3 (initialmente D no está en ninguna cache):
 - Lectura generada por el procesador del nodo 1
 - Escritura generada por el procesador del nodo 1
 - Lectura generada por el procesador del nodo 2
 - Lectura generada por el procesador del nodo 3
 - Escritura generada por el procesador del nodo 0

- En primer lugar, el tamaño del directorio depende del número de entradas y del tamaño de cada entrada. Tenemos una entrada en el directorio por cada bloque de memoria principal y cada bloque tiene un tamaño de 64B:

$$Tam_{subdirectorio} = \underbrace{\frac{Tam_{memoria}}{Tam_{linea\ cache}}}_{\substack{\text{numero entradas}}} \cdot \underbrace{(4 + 1\ bits)}_{tam\ cada\ entrada} = \frac{2^3 \cdot 2^{30}B}{2^6B} \cdot 5bits \cdot \frac{1}{2^3bits/B} = 5 \cdot 2^{24}B = 2^4 \cdot 5 \cdot 2^{20} = 80MB$$

Como el directorio está distribuido en cuatro nodos, el tamaño total del directorio será:

$$Tam_{director} = 4 \cdot Tam_{subdirector} = 320MB$$

En sistemas con muchos procesadores, los directorios no tienen entradas para todos los nodos pues sino, el directorio tendría un tamaño demasiado grande.

- b) El estado inicial del sistema sería el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	I	I	I	0	0	0	0	V

El bloque está inválido en todas las caches pero está válido en la memoria principal de N3.

1. En este caso, el solicitante sería N1 y el Origen, N3. N1 envía a N3 un paquete petición de lectura ($PtLec(k)$) y N3 le responde con el bloque ($RpBloque(k)$). El estado del sistema pasaría a ser el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	S	I	I	0	1	0	0	V

El bloque sigue inválido en todos los demás nodos pero, ahora está en estado compartido en N1. N1 pasa a ser propietario del bloque.

2. Al igual que antes, el solicitante sería N1 y el origen N3. Como N1 ya tiene el bloque en su cache, lo que envía a N3 es un paquete de petición de acceso exclusivo ($PtEx(k)$). N3 comprueba que N1 es el único nodo que tiene ese bloque y le responde con la confirmación de la invalidación del bloque ($RpInv(k)$). El estado del sistema pasaría a ser el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	M	I	I	0	1	0	0	I

3. En este caso, N2 sería el solicitante, N3 el origen y N1 el propietario. N2 mandaría una petición de lectura ($PtLec(k)$) a N3. Al tener N3 el bloque inválido, reenvía esta petición a N1 ($RvLec(k)$) y éste responde con el bloque a N3 ($RpBloque(k)$) que le responde finalmente a N2 ($RpBloque(k)$). El estado del sistema pasaría a ser el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	S	S	I	0	1	1	0	V

Además de los estados estables (válido e inválido) también hay estados “pendientes” que sirven para cuando tenemos el nodo esperando un paquete para pasar su bloque de un estado a otro. Por ejemplo, si el nodo tiene el bloque inválido y ha hecho una petición de lectura, mientras llega el bloque lo tendría en estado “pendiente de válido”.

4. En este caso no hay comunicaciones a través de la red porque el nodo solicitante y el origen son el mismo. Pero sí que hay cambios de estado:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	S	S	S	0	1	1	1	V

5. En este caso tenemos varios nodos propietarios (N1, N2 y N3). N3 a parte de propietario también sería Origen. El nodo solicitante sería N0. N0 envía una petición de acceso exclusivo y lectura a N3, y N3 envía una respuesta de invalidación a N1 y N2. Una vez ha recibido la invalidación por parte de N1 y N2, N3 envía la confirmación a N0. El estado del sistema tras esto sería el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
M	I	I	I	1	0	0	0	I

En este caso, mientras N3 espera la confirmación por parte de N1 y N2, tendría su bloque de memoria principal en estado “Pendiente de Invalidación”.

MSI con directorios con difusión

Con difusión, el nodo solicitante envía la petición a todos los nodos. Así, si el nodo origen no tiene copia válida pero la tiene otro nodo, dicho nodo recibiría la petición y contestaría directamente con el bloque solicitado. Así, la latencia de acceso a memoria es menor. Ahora bien, la productividad dependería del tipo de enlaces (de si podemos mandar paquetes paralelamente o no). Pero aunque pudiésemos mandar paquetes paralelamente, estaríamos ocupando enlaces innecesariamente. Por tanto, con esta alternativa conseguimos menor latencia pero también, menor ancho de banda.

En esta alternativa no tenemos reenvíos de peticiones desde el nodo origen.

Las placas de Intel usan esta alternativa.

El ejercicio anterior usando difusión sería:

El estado inicial del sistema sería el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	I	I	I	0	0	0	0	V

El bloque está inválido en todas las caches pero está válido en la memoria principal de N3.

1. En este caso, el solicitante sería N1 y el Origen, N3. N1 envía a todos los nodos un paquete petición de lectura ($PtLec(k)$) y N3 le responde con el bloque ($RpBloque(k)$). El estado del sistema pasaría a ser el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	S	I	I	0	1	0	0	V

2. Al igual que antes, el solicitante sería N1 y el origen N3. Como N1 ya tiene el bloque en su cache, lo que envía a todos los nodos es un paquete de petición de acceso exclusivo (PtEx(k)). N3 comprueba que N1 es el único nodo que tiene ese bloque y le responde con la confirmación de la invalidación del bloque (RpInv(k)). El estado del sistema pasaría a ser el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	M	I	I	0	1	0	0	I

3. En este caso, N2 sería el solicitante, N3 el origen y N1 el propietario. N2 mandaría una petición de lectura (PtLec(k)) a todos los nodos. Al tener N3 el bloque inválido, no podría responder a esta petición, pero como N1 sí la tiene, responde con el bloque a N3 (RpBloque(k)) que le responde finalmente a N2 (RpBloque(k)). El estado del sistema pasaría a ser el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	S	S	I	0	1	1	0	V

En este caso, nos hemos ahorrado el reenvío de la petición a N1 gracias a la difusión.

4. En este caso no hay comunicaciones a través de la red porque el nodo solicitante y el origen son el mismo. Pero sí que hay cambios de estado:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
I	S	S	S	0	1	1	1	V

5. En este caso tenemos varios nodos propietarios (N1, N2 y N3). N3 a parte de propietario también sería Origen. El nodo solicitante sería N0. N0 envía una petición de acceso exclusivo y lectura, y N3 envía una respuesta de invalidación a N1 y N2. Una vez ha recibido la invalidación por parte de N1 y N2, N3 envía la confirmación a N0. El estado del sistema tras esto sería el siguiente:

Caches				Directorio				
N0	N1	N2	N3	N0	N1	N2	N3	
M	I	I	I	1	0	0	0	I

3.9 Consistencia del sistema de memoria

3.9.1 Concepto de consistencia de memoria

El concepto de consistencia de memoria se refiere al orden en el que todos los procesadores ven los accesos a memoria a todas las direcciones de todos ellos. Es decir, especifica (las restricciones en) el orden en el cual las operaciones de memoria (escritura o lectura) deben parecer haberse

realizado. Estas operaciones pueden ser a las mismas o a distintas direcciones de memoria y pueden ser emitidas por el mismo o por un distinto procesador.

La coherencia sólo abarca operaciones realizadas por múltiples componentes (proceso/procesador) en una misma dirección.

Por un lado están los modelos de consistencia del hardware y por otro lado, los modelos de consistencia que ofrecen las distintas herramientas de programación.

Si usamos, por ejemplo, OpenMP con sus directivas nos ahorramos problemas. Si programamos nosotros mismos la sincronización entre procesos obtendremos código más eficiente pero tendremos que lidiar con el modelo de consistencia del hardware.

El programador de una herramienta de programación tiene que implementar la sincronización él mismo a bajo nivel.

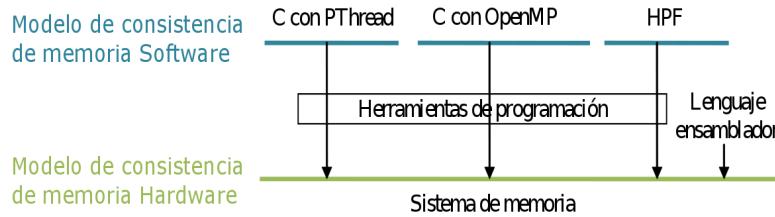


Figure 3.20: Modelos de consistencia hardware y de las herramientas de programación

3.9.2 Consistencia secuencial

No lo implementa ningún multiprocesador pero sí las herramientas de programación. Es el modelo de consistencia que espera el programador de las herramientas de alto nivel.

La consistencia secuencial requiere que:

- ♡ Todas las operaciones de un único procesador parezcan ejecutarse en el orden descrito por el programa de entrada al procesador (*orden del programa*). Es decir, el orden en el que todos los procesadores ven los accesos a memoria de un procesador debe ser el mismo orden en el que se encuentra en el código que ejecuta ese procesador.
- ♡ Todas las operaciones de memoria parezcan ser ejecutadas una cada vez (*ejecución atómica*). Es decir, que todos los procesadores ven el mismo orden en el acceso a memoria.

La consistencia secuencial presenta el sistema de memoria a los programadores como una *memoria global* conectada a todos los procesadores a través de un *conmutador central*.

Por ejemplo, si tenemos inicialmente que $A = 0$ y dos procesadores ejecutando estos códigos a la vez:

```
[linenos,
frame=single,
label=P1]c A=1;
k=1;
```

```
[linenos, frame=single, label=P2]c while (k = 0) ;
copia = A;
```

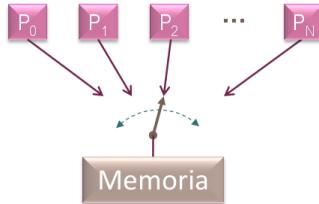


Figure 3.21: Esquema consistencia secuencial

¿Cuál es el valor que el programador espera ver en **copia**? Como no se le asigna valor a copia hasta que $k = 1$, y antes de que P_1 haga $k = 1$ ha hecho $A = 1$, el valor que esperamos ver en **copia** será, por tanto, 1.

El orden en el se pueden suceder las operaciones puede ser cualquiera que cumpla con varias condiciones:

- ♡ Escribir $A = 1$ debe ejecutarse antes que Escribir $k = 1$
- ♡ Leer k mientras que sea igual a 0 debe ejecutarse antes que leer A
- ♡ Escribir $k = 1$ debe ejecutarse antes que leer A .

Así por ejemplo, teniendo:

- ♡ Escribir $A = 1$ como (1)
- ♡ Escribir $k = 1$ como (2)
- ♡ Leer k mientras que sea distinto de 0 como (a)
- ♡ Y leer A como (b)

Estos órdenes serían válidos:

1. (1)(a) ... (a)(2)(a)(b)
2. (1)(2)(a)(b)
3. (a) ... (a)(1)(a) ... (a)(2)(a)(b)
4. etc

Pero sea cual sea el orden, todos los procesadores verán ese orden. Si P_1 ve (1)(2)(a)(b) pero P_2 ve (1)(a) ... (a)(2)(a)(b) se incumpliría el segundo requisito de consistencia secuencial.

Veamos otro ejemplo. ¿Qué espera el programador que se almacene en **reg1** si llega a ejecutarse **reg1=A**? Teniendo en cuenta que inicialmente $A = B = 0$

```
[linenos,
frame=single,      [frame=single, label=P2]c if [frame=single, label=P3]c if
label=P1]c A=1;    (A=1) B=1;           (B=1) reg1=A;
```

El programador esperaría $A = 1$, pero para ello debería cumplirse el segundo requisito. Esto puede no ocurrir si usamos redes que no sean buses. Esto se debe a las caches.

P2 y P3 tienen A en su cache. Cuando P1 escribe $A = 1$, esto se propaga a los demás procesadores que tengan A guardado en su cache mediante un paquete. Este paquete puede llegar antes a P2 que a P3 y entonces, poner $B = 1$. Tras esto P2 enviaría esta modificación a P3 y, puede ser que llegue antes que el paquete de $A = 1$. Si esto ocurriese, tendríamos que finalizaría la ejecución con $A = 0$.

Para que la atomicidad (segundo requisito) dé problemas, debe haber como mínimo tres procesadores.

3.9.3 Modelos de consistencia relajados

El modelo secuencial es muy poco eficiente, por tanto, los multiprocesadores implementan los modelos relajados, que relajan uno de los dos requisitos de la consistencia secuencial para incrementar prestaciones. Para describir el modelo de consistencia debemos describir qué requisitos relajamos:

- ♡ **Orden del programa:** hay modelos que permiten que se relaje en el código ejecutado en un procesador el orden entre dos accesos a distintas direcciones ($W \rightarrow R$, $W \rightarrow W$, $R \rightarrow RW$). Es decir, se permite que se adelanten accesos a memoria anteriores a variables distintas (para que no haya dependencias de datos).
- ♡ **Atomicidad:** hay modelos que permiten que un procesador pueda ver el valor escrito por otro antes de que este valor sea visible al resto de los procesadores del sistema.

Los modelos relajados comprenden:

- ♡ Las órdenes de acceso a memoria que no garantiza el sistema de memoria (tanto órdenes de un mismo procesador como atomicidad en las escrituras).
- ♡ Mecanismos que ofrece el hardware para garantizar un orden cuando sea necesario, es decir, cuando necesitamos sincronización.

Cuando necesitamos sincronización, el no garantizar un orden es problemático. El resto del tiempo se puede no garantizar un orden. Los accesos a memoria pueden desordenarse si se hacen a variables distintas.

Todos los modelos de consistencia hardware relajados permiten hacer una lectura adelantada de datos del propio procesador. Esto es posible porque todos los procesadores tienen un buffer de escritura, al que sólo puede acceder el propio procesador. Por eso puede leer los valores próximos de sus variables cuando aún no han sido escritas.

Todos los procesadores ofrecen instrucciones máquina para garantizar orden. Los programadores las usan para implementar herramientas de programaciones tales como semáforos y monitores.

Modelo que relaja $W \rightarrow R$

Rejala solamente los órdenes de escritura-lectura, es decir, son los que permiten que una lectura pueda adelantar a una escritura previa en el orden del código que ejecuta un flujo de control, pero evitan dependencias RAW.

Todos los procesadores del mercado implementan este nivel de relajación. Para ello, tienen un *buffer de escritura*. El buffer evita que las escrituras retarden la ejecución del código bloqueando lecturas posteriores.

Generalmente, permiten que el procesador pueda leer una dirección directamente del buffer y así leer antes que otros procesadores una escritura propia.

Para garantizar un orden correcto se pueden utilizar instrucciones de serialización, que son las instrucciones que, antes de ejecutarse, esperan a que se vacíe el cauce. Por ejemplo, las instrucciones máquina que se utilizan para programar herramientas de sincronización de la línea x80_86 de Intel, serializan.

Para un core de procesamiento, una escritura ha terminado cuando la escritura se coloca en el buffer. Las escrituras se colocan en el buffer en el orden en el que se encuentran en el código que ejecuta el core. Esto permite procesar una lectura antes de que ésta llegue a memoria y así poder adelantar a una escritura.

Hay sistemas en los que se permite que un procesador pueda leer la escritura de otro antes que el resto de procesadores (acceso no atómico). Para garantizar acceso atómico se puede utilizar instrucciones de lectura-modificación-escritura atómicas.

Modelo que relaja $W \rightarrow R$ y $W \rightarrow W$

Además de los modelos que relajan $W \rightarrow R$ podemos encontrar modelos que relajan escrituras. Una escritura puede adelantar a una escritura anterior del mismo flujo de control.

Los procesadores *Sun Sparc* tienen modelos de consistencia de este tipo (PSO). Para garantizar este orden, los procesadores Sparc usan la instrucción máquina STBAR. Esta instrucción se debe poner entre las dos escrituras para garantizar que hasta que no se han hecho los almacenamientos anteriores a la barrera en el flujo de control que ejecuta el core, no se van a hacer las escrituras que hay detrás de la barrera. La instrucción máquina 1-m-e garantiza los dos órdenes que se relajan.

Los modelos de consistencia que no garantizan el orden entre escrituras van a hacer que el siguiente código no dé un resultado correcto:

```
[linenos,
frame=single,
label=P1]c A=1;
k=1;                                [linenos, frame=single, label=P2]c while (k = 0) ;
                                         copia = A;
```

Esto de debe a que $k = 1$ podría adelantar a $A = 1$ y tener en la variable `copia` un 0 (valor inicial de A) en vez de un 1.

Ahora bien, el orden puede no estar garantizado por el core o por el sistema de memoria. Ya que los accesos a memoria de escritura suponen un tiempo considerable, pues debemos transmitir paquetes en secuencia a través de la red: el número de paquetes depende de si el bloque a escribir está en otras caches o no. Si está tendríamos que mandar cuatro paquetes (para invalidar otras copias) y si no, sólo necesitamos mandar dos paquetes.

Por ejemplo, si A está en otras caches y k no, la escritura de k puede adelantarse a la de A aunque el procesador lance las escrituras en orden y por tanto, acabar la variable `copia` con valor 0.

Modelo de ordenación débil

Hay modelos de consistencia que relajan todos los órdenes ($W \rightarrow R$, $W \rightarrow W$ y $R \rightarrow RW$). Al relajar todos los órdenes tenemos mayores prestaciones pues no tendremos órdenes en el acceso a memoria entre variables distintas.

Los procesadores *PowerPC* implementan este modelo, por tanto, no garantizan ningún orden pero tienen instrucciones máquina que permiten mantenerlo.

Por ejemplo, si S es una operación de sincronización, podemos garantizar que no se harán accesos a memoria posteriores a S en el código, hasta que no se hayan terminado los anteriores.

♡ $S \rightarrow WR$

♡ $WR \rightarrow S$

Por ejemplo, en el siguiente código:

```
c for (i = iproc; i<n; i+=nproc) sump += a[i]; lock(k); // adquisicion (isync)
// sección critica unlock(k) // liberación (sync)
```

se accede a la variable compartida `sum` entre varios flujos de control. La variable `sum` es privada en cada thread y una vez que todos los flujos han ejecutado el bucle `for`, acumulan su resultado en `sum`. Esto último debe hacerse secuencialmente. Para garantizar esto se puede usar un cerrojo (`lock` y `unlock`).

A la hora de implementar el cerrojo se debe usar una instrucción barrera (`isync` en el caso de PowerPC) que garantiza que hasta que no se han terminado los accesos a memoria anteriores al cerrojo no se hacen los posteriores. Lo mismo ocurre con el `unlock` (`sync` en PowerPC). Son instrucciones diferentes porque `sync` garantiza atomicidad e `isync` no. Podríamos usar `sync` en ambos casos, pero en el primero no nos hace falta y es una operación más costosa.

Existen dos tipos de operación de sincronización:

♡ **Adquisición:** se da cuando un procesador quiere adquirir el derecho a acceder a unas variables compartidas para modificarlas. El acceso a las variables compartidas se hace tras la operación de adquisición, no antes.

♡ **Liberación:** el flujo de control accede a una operación de liberación cuando ha terminado de acceder a unas variables compartidas para que otro flujo de control pueda acceder a ellas. El acceso se hace después de la operación de liberación y no antes.

El resultado de nuestro código sería erróneo si más de un thread escribiese a la vez en la variable compartida. Por tanto, debemos garantizar un orden en la operación de adquisición y la variable compartida pero no debemos garantizar el orden con los accesos a memoria anteriores a la adquisición. Es decir, que podríamos ejecutar nuestro código en exclusión mutua y después el código que venía antes sin ningún problema.

El orden que garantizan las instrucciones de ordenación débil no es necesario garantizarlo pues no accedemos a variables compartidas.

Es decir, garantizamos el orden de ejecución (Figura ??)

$$1 \rightarrow SA \rightarrow 2 \rightarrow SL \rightarrow 3$$

Cuando sólo es necesario garantizar

$$SA \rightarrow 2 \rightarrow SL$$



Figure 3.22: Orden garantizado en el modelo de ordenación débil

Consistencia de liberación

Este problema lo resuelve el modelo de consistencia de liberación. Este modelo permite mantener el orden entre `lock`, `sum` y `unlock` pero no entre operaciones previas a `lock` y posteriores a `unlock`. Así, el programa tiene un tiempo de ejecución más rápido.

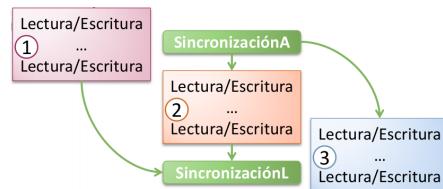


Figure 3.23: Orden garantizado en la consistencia de liberación

Así, podríamos ejecutar en paralelo la sección crítica (2) y tanto 3 como 1 y tener un menor tiempo de ejecución.

Así, si `SA` es una operación de adquisición y `SL`, de liberación, se garantiza el siguiente orden:

$\heartsuit SA \rightarrow WR$

$\heartsuit WR \rightarrow SL$

Lo implementan procesadores Itanium.

Ejercicio 4 - Relación de ejercicios tema 3

Supongamos que se va a ejecutar en paralelo el siguiente código (initialmente x e y son 0):

```
[linenos, frame=single, label=P1]c x=1; // W(x) x=2;
// W(x) print y; // R(y)
```

```
[linenos, frame=single, label=P2]c y=1; // W(y) y=2;
// W(y) print x; // R(x)
```

Qué resultados se pueden imprimir si (considere que el compilador no altera el código):

- Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial.
- Se ejecutan en un multiprocesador basado en un bus que garantiza todos los órdenes excepto el orden $W \rightarrow R$. Esto es debido a que los procesadores tienen buffer de escritura, permitiendo el procesador que las lecturas en el código que ejecuta adelanten a las escrituras que tiene su buffer. Obsérvese que hay varios posibles resultados.

- Pueden pasar dos cosas: que en primer lugar imprima P1 o que en primer lugar imprima P2.

Si imprimiese primero P1, tendrían que haberse hecho también las escrituras en la variable x , y puede que se hubiese hecho alguna escritura en la variable y , por tanto, podríamos obtener:

P1	P2
0	2
1	2
2	2

En cambio, si imprimiese primero P2, sería al revés.

P1	P2
2	0
2	1
2	2

- En este caso, las lecturas se pueden ejecutar antes que las escrituras, podría la lectura de y en P1 adelantar a las dos escrituras de x o adelantar sólo a una. De igual forma en P2. Por tanto, podemos obtener todas las combinaciones de 0, 1 y 2 posibles:

P1	P2
0	0
0	1
0	2
1	0
1	1
1	2
2	0
2	1
2	2

Ejercicio 5 - Relación de ejercicios tema 3

Supongamos que se va a ejecutar en paralelo el siguiente código (initialmente x e y son 0):

```
[linenos, frame=single, label=P1]c
x=30; // W(x)
y=40; // W(y) flag=1; //
W(f)

[linenos, frame=single, label=P2]c
while (flag==0); // R(f) r1 = x; //
R(x) r2 = y; // R(y)
```

Qué datos puede obtener P2 en $r1$ y $r2$ (considere que el compilador no altera el código):

- a) Se ejecutan P1 y P2 en un multiprocesador con consistencia secuencial
 - b) Se ejecutan en un multiprocesador con consistencia de liberación. Razone su respuesta.
-
- a) Se esperaría tener $r1 = 30$ y $r2 = 40$ porque para poder salir del `while`, `flag` debe ser igual a 1 y para ello, antes deben haberse hecho las escrituras en x y en y .
 - b) Se podría obtener $r1 = 0$ y $r2 = 0$ si la escritura $W(f)$ se adelanta a las otras dos. O se podría obtener $r1 = 30$ y $r2 = 0$ si $W(f)$ se adelanta a la escritura $W(y)$. También se podría obtener los mismos resultados que en un modelo secuencial si no se adelanta ninguna escritura.
- También se pueden adelantar las lecturas, por tanto, podríamos tener también $r2 = 40$ y $r1 = 0$.
- Como el orden de las operaciones es totalmente aleatorio pues no se conserva ningún orden, se puede obtener cualquier resultado.

3.10 Sincronización

3.10.1 Comunicación en multiprocesadores y necesidad de usar código de sincronización

Comunicación en un multiprocesador

Se necesita sincronización porque los procesadores se comunican a través de la memoria compartida y, para que la comunicación se lleve a cabo sin problemas, se debe introducir código de sincronización.

Para realizar la comunicación entre el productor de un dato y un consumidor se usa la memoria, en particular, la memoria compartida. La comunicación se lleva a cabo a través de una dirección de memoria compartida por los diferentes nodos.

Así, el productor escribirá el dato en memoria y el consumidor lo leerá de memoria. Para que ésto se lleve a cabo el consumidor debe esperar a que el dato esté escrito en memoria.

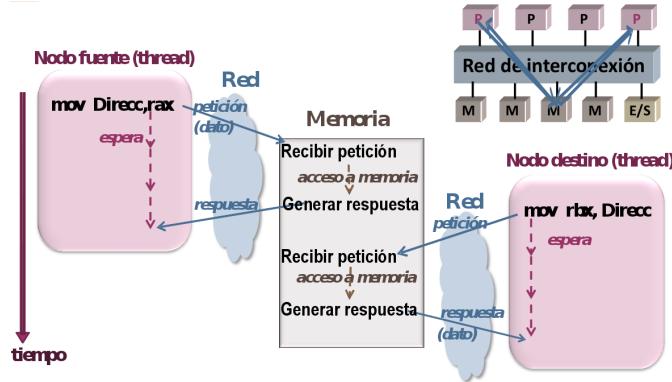


Figure 3.24: Esquema de la comunicación entre dos procesadores

Comunicación uno a uno

Secuencial	Paralela
<code>c ... A = valor; ... copia = A; ...</code>	P1 <code>c ... A = valor; ...</code> P2 <code>c ... copia = A; ...</code>

Se debe garantizar que el proceso que recibe lea la variable compartida cuando el proceso que envía haya escrito en la variable el dato a enviar. Si se reutiliza la variable para comunicación, se debe garantizar que no se envía un nuevo dato en la variable hasta que no se haya leído el anterior.

Para que la ejecución paralela devuelva el mismo resultado que la secuencial, P2 debe esperar a que P1 escriba en A. Para ello, podemos añadir una variable barrera (`while`) y así, P2 esperaría mientras que `k=0` hasta que `A=1`. Para que esto funcione bien el modelo de consistencia del multiprocesador debe ser un modelo que garantice el orden entre escrituras y lecturas, de no ser así debemos añadir código extra para garantizar los órdenes:

[linenos, frame=single, label=P1]
`c ... A=1; k=1; ...`

[linenos, frame=single, label=P2]
`c ...
while (k==0) ; copia = A; ...`

Comunicación colectiva

Secuencial	Paralela(<code>sum=0</code>)
<code>c for (i = 0; i < n; i++) sum += a[i];</code>	<code>c for (i = ithread; i < n; i += nthread) sum += a[i]; sum += sum; /* SC, sum compartida */ if (ithread == 0) printf(sum);</code>

El código lo deben ejecutar todos los flujos de control. En algunas ejecuciones cuando el thread 0 imprima el valor de `sum` obtendremos un valor correcto y en otras no. Esto se debe a que accedemos a la variable compartida `sum` de forma paralela cuando debemos de acceder de forma secuencial, pues tenemos una lectura y una escritura de `sum`. Si entre medias de la lectura y de la escritura de un thread, otro thread lee el valor de `sum`, los dos leerán el mismo valor y acumularán su suma parcial en el mismo valor, por tanto, perderemos la suma parcial de uno de ellos.

Por ejemplo: Tenemos dos procesadores ($nthread = 2$) y cada uno ha acumulado como suma parcial $sum_1 = 2$ y $sum_0 = 1$. Un posible orden de ejecución sería: $R_0 W_0 R_1 W_1$.

♥ R_0 devolverá un cero, y lo que hará a continuación el thread 0 será sumarle (W_0) su $sum = 1$, dejando la variable $sum = 1$.

♥ Después, R_1 leerá un uno y le sumará (W_1) $sum = 2$ dejando $sum = 3$.

Así, obtendremos un resultado correcto.

En cambio, si obtenemos este orden de ejecución: $R_1 R_0 W_0 W_1$, obtendríamos los siguientes resultados:

♥ $R_1 \rightarrow sum = 0 \rightarrow sum = 2$

♥ $R_0 \rightarrow sum = 0 \rightarrow sum = 1$

Así, obtendremos un resultado erróneo, ya que hemos perdido la suma parcial del thread 1.

Estas lecturas y escrituras deberían realizarse en *exclusión mútua* ya que es una *sección crítica*², para ello, podemos utilizar *cerrojos*.

Podría ocurrir que aun usando cerrojos, al imprimir el resultado no se haga correctamente. Ya que cuando el thread 0 terminase su ejecución y saliese de la exclusión mútua pasaría a ejecutar el `printf`. Para evitar esto, debemos poner tras la exclusión mútua una *barrera* y así hacer que los threads se esperen entre sí.

Los cerrojos y las barreras son las dos primitivas mínimas que necesitamos para garantizar que las comunicaciones se llevan a cabo sin problemas ya que los cerrojos no tienen barrera. Podemos usar cerrojos para hacer comunicaciones uno a uno, aunque es más costoso que la alternativa con el bucle `while`.

Otro ejemplo sería el siguiente código paralelo: [linenos]c pragma omp parallel for for (i=0; i<n; i++) suma += a[i];

Hacemos un acceso paralelo a la variable compartida `suma` cuando éste acceso debería hacerse secuencialmente. La solución sería poner un cerrojo o una directiva `atomic` o `critical`, pero si hacemos eso el código resultante sería secuencial ya que todas las iteraciones del `for` deberían hacerse una detrás de otra. Tendríamos que implementar una alternativa de sumas privadas como las del ejemplo anterior para ejecutar la sección crítica tantas veces como sea el número de hebras, pero no tantas veces como iteraciones tenga el `for`.

3.10.2 Soporte software y hardware para sincronización

El nivel de abstracción es mayor cuando usamos monitores que cuando usamos semáforos, por tanto, es menos probable que tengamos fallos con monitores, pero los programas obtenidos usando monitores son más lentos que los obtenidos usando semáforos. Las primitivas software se implementan usando instrucciones máquina que implementan una lectura y escritura de forma atómica. La mayor parte de los procesadores ofrecen instrucciones de este tipo. Otros en cambio ofrecen instrucciones LL/SC con las que comprobamos que cuando hacemos una lectura y una

²Secuencia de instrucciones con una o varias direcciones compartidas(variables) que se deben acceder en exclusión mútua

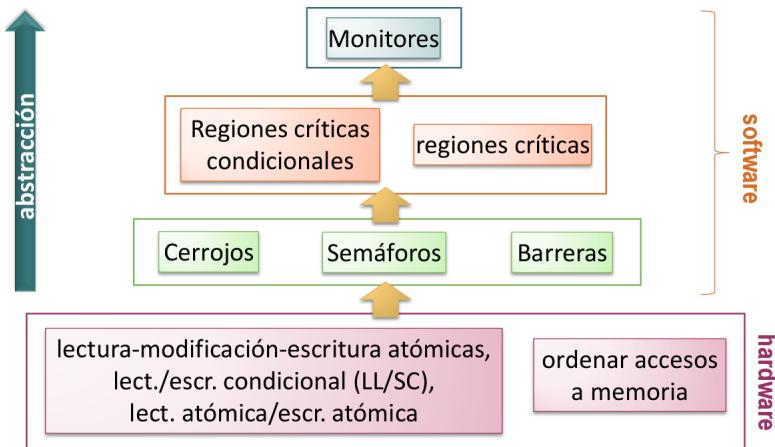


Figure 3.25: Jerarquía de mecanismos de sincronización según su nivel de abstracción

escritura no han habido accesos de otros threads entre medias. Si los ha habido, la operación debe repetirse.

Las primitivas software aprovechan el hecho de que las lecturas y las escrituras son atómicas.

Cuando implementemos primitivas software debemos añadir instrucciones máquina que garanticen el orden que no garantiza el modelo de consistencia a la hora de implementar cerrojos y barreras.

3.10.3 Cerrojos

Los cerrojos permiten sincronizar mediante dos operaciones:

- ♡ `lock(k)` o **cierre** del cerrojo: intenta *adquirir* el derecho a acceder a una sección crítica (cerrando o adquiriendo el cerrojo *k*).
- ♡ `unlock(k)` o **apertura** del cerrojo: *libera* a uno de los threads que esperan el acceso

Si varios procesos intentan la adquisición a la vez, sólo uno de ellos lo debe conseguir, el resto debe pasar a una etapa de espera. Además, todos los procesos que ejecuten el `lock(k)` deben quedar esperando. Una vez ejecutado `unlock(k)`, si hay threads en espera, se permitirá el acceso a la sección crítica a uno de los threads que están esperando.

Así, nuestro ejemplo anterior con cerrojos quedaría de la siguiente forma:

```
[linenos]c for (i = ithread; i < n; i += nthread) sump += a[i];
lock(k); sum += sump; /* SC, sum compartida */ unlock(k);
if (ithread == 0) printf(sum);
```

Para implementar la espera, tenemos dos opciones:

- ♡ Espera ocupada
- ♡ Suspensión del proceso o thread, éste queda esperando en una cola, el procesador conmuta a otro proceso-thread.

La decisión de cuál es mejor o peor depende de lo larga que sea nuestra sección crítica. En este ejemplo sería más conveniente hacer una espera ocupada, pues el tiempo para suspender los threads es mucho mayor y la sección crítica es muy corta.

Cerrojos simples

Se implementan con una variable compartida k que toma dos valores: abierto(0) y cerrado(1). Para abrir el cerrojo bastaría con escribir un 0 en k y para cerrarlo, un 1.

Así, sin tener en cuenta detalles sobre el modelo de consistencia bastaría con la siguiente implementación:

$[linenos, frame=single, label=lock(k)]c$ $lock(k)$ $while$ $(leer-asignar_1 - escribir(k) == 1); /* kcompartida */$	$[linenos,$ $frame=single,$ $label=unlock(k)]c$ $unlock(k)$ $k = 0;$
---	--

Así, si varios threads llegan a la vez al cerrojo y está abierto, sólo uno de ellos podrá pasar. Esto sólo puede ser posible haciendo que `leer-asignar_1-escribir(k)` sea una operación indivisible, para que sólo devuelva un 0 la primera vez que se ejecute y un 1 para todas las posteriores.

Al implementar estas primitivas debemos hacerlo en ensamblador.

Ejercicio 6 - Relación de ejercicios tema 3

Se quiere implementar un cerrojo simple en un multiprocesador SMP basado en procesadores de la línea x86 de Intel, en particular, procesadores Intel Core.

- a) Teniendo en cuenta el modelo de consistencia de memoria que ofrece el hardware de este multiprocesador, ¿podríamos implementar la función de liberación del cerrojo simplemente usando el siguiente código, siendo k la variable del cerrojo? Razona su respuesta: `gas unlock(k): mov k, 0`
 - b) ¿Cómo se debería implementar la función de liberación de un cerrojo simple si se usan procesadores itanium? Razona su respuesta.
-
- a) El modelo de consistencia de la línea x86 de Intel sólo permite que las lecturas adelanten a escrituras anteriores, por tanto, ese código sería completamente válido para ser cerrojo pues la escritura $k = 0$ no se adelantaría a las operaciones que estuviera haciendo el thread en la sección crítica y por tanto, no se abriría el cerrojo hasta que el thread no terminase todas sus operaciones en la sección crítica.
 - b) El modelo de consistencia en los Itanium permiten que las escrituras adelanten a lecturas anteriores, por tanto, con esa implementación, el cerrojo podría abrirse mientras el thread estuviera aún en la sección crítica. Por tanto, deberíamos añadirle una instrucción máquina (`ST.REL`) para asegurarnos de que se realizan todos los accesos a memoria antes de ejecutar $k = 0$.

Cerrojos con etiqueta

Los cerrojos simples tienen un problema: puede que haya threads que jamás entren en la sección crítica y que otros que lleguen después pasen. No hay un orden entre los threads que ejecutan el `lock` y esto puede producir inhanición.

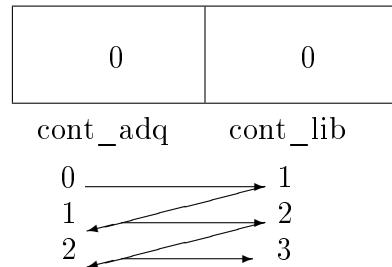
Hay otra implementación de cerrojos en los que van entrando en la sección crítica en el orden en el que ejecutan el `lock`. Esta implementación garantiza un orden FIFO, para ello, usa dos contadores:

- ♡ Contador de adquisición, implementado por el `lock`
- ♡ Contador de liberación, implementado por el `unlock`

Ambos contadores son variables compartidas. Cuando cada thread ejecuta el `lock` se guarda el valor del contador de adquisición y se incrementa en uno. Así, cada uno guarda su orden de acceso a la sección crítica.

```
[linenos, frame=single, label=lock(contadores)]c contador_local_adq =  
contadores.adq; contadores.adq = (contadores + 1)while(contador_local_adq <> contadores.lib);  
[linenos, frame=single, label=unlock(contadores)]c contadores.lib = (contadores.lib + 1)
```

Los threads se quedan esperando hasta que su orden de adquisición coincide con el contador de liberación:



3.10.4 Barreras

Las barreras son puntos del código en el que todos los threads se esperan entre sí. Hasta que no llegan todos los threads a ese punto no se continua con la ejecución. Para saber si han llegado todos o no se usa un contador, así, al llegar a la función barrera se incrementa el contador en uno, se comprueba si coincide con el número de threads: si no coincide se deben quedar esperando en la espera ocupada hasta que la bandera sea igual a cero; si coincide, habría que liberar a los que están esperando escribiendo en la bandera algo distinto de cero para que los threads que estaban esperando salgan de la espera ocupada. El último thread no espera y sale con los demás de la función.

Si la función barrera se va a reutilizar varias veces debemos poner el contador y la bandera a cero.

Así, el procedimiento descrito se correspondería con el siguiente código:

```
[linenos]c Barrera (id, numprocesos, cerrojo)//Accesoexclusionmutualock(bar[id].cerrojo); if(bar[id].cont == 0)bar
if (contlocal == numprocesos)bar[id].cont = 0; bar[id].bandera = 1;
// Implementar espera. Espera ocupada: else while (bar[id].bandera == 0) ;
```

Hemos supuesto en esta implementación que tenemos múltiples barreras, por eso usamos un vector de barreras (cada barrera se compondría de su contador, su cerrojo y su bandera).

Esta implementación tiene un problema a la hora de reutilizar barreras y suspender threads.

Supongamos la siguiente situación: tenemos tres threads distintos, en los que tenemos dos llamadas a la misma barrera:

```
[linenos, frame=single, la- [frame=single, label=T2]c [frame=single, label=T3]c
bel=T1]c ... Barrera(g,3); ... ... Barrera(g,3); ... Bar- ... Barrera(g,3); ... Bar-
Barrera(g,3); ... rera(g,3); ... rera(g,3); ...
```

Imaginemos, por ejemplo, que T1 llega el primero a la barrera, comprueba que su contador local no coincide con el global y entra a la espera ocupada. Después llega T2 y hace lo mismo. En el instante en el que entra T3, el sistema operativo suspende T2.

Después de eso, T3 entra comprueba que su contador local es igual al número de hebras y pone la bandera a 1. T1 y T3 continúan con la ejecución, pero T2 al estar suspendido no ve dicha modificación.

Al rato, T3 vuelve a entrar en la barrera e inicializa de nuevo el valor del contador y la bandera. En ese momento el sistema operativo reanuda T2 y, al ver la bandera a 0, se queda en la espera ocupada eternamente.

Barreras sin problema de reutilización

Este problema se resuelve de forma muy sencilla: en el primer uso de la barrera todos los threads esperan mientras la bandera sea 0 y en el segundo, esperan mientras sea 1. Esta modificación nos lleva al siguiente código:

```
[linenos]c Barrera(id, numprocesos, cerrojo)banderalocal =!(banderalocal); //complementamos la bandera lock(bar[id]
if (contlocal == numprocesos)bar[id].cont = 0; //se hace 0 el cont de la barrera bar[id].bandera = banderalocal; //para la
banderalocal); //espera ocupada
```

Esta modificación permite que en cada uso de la barrera hagamos la hagamos con un valor complementario en la bandera.

El acceso a las variables compartidas de la barrera debe hacerse en exclusión mútua.

3.10.5 Apoyo hardware a primitivas software

Instrucciones de lectura-modificación-escritura atómicas

El hardware nos ofrece instrucciones para leer y escribir una modificación de forma atómica, es decir, de forma que entre la lectura y la escritura no se van a producir accesos a memoria a esa variable por parte de ningún otro thread.

Hay tres tipos:

- ♡ **Test&Set(x):** lee el contenido actual de la variable compartida y la modifica escribiendo un 1. En la línea x86 se usa una instrucción de intercambio de los valores de dos variables.

```
[linenos]c Test_and_set(x)temp = x; x = 1; return([linenos]gas mov reg, 1 xchg reg, mem  
xcompartida * / reg ←→ mem
```

- ♡ **Fetch&Oper(x, a):** tiene dos parámetros: la variable compartida y una local. Lee el valor de la variable compartida y la modifica con el valor que resulta de una operación con el segundo parámetro (ejemplos: fetch&add, fetch&zor...). En la línea x86 debemos añadir un *lock* para garantizar la atomicidad. El siguiente ejemplo suma los dos números:

```
[linenos]c Fetch_and_oper(x, a)temp = x; x = x + a; [linenos]gas lock xadd reg, mem  
xcompartida, alocal * / reg ←→ mem  
mem ←→ reg + mem
```

- ♡ **Compare&Swap(a, b, x):** tiene tres parámetros: la variable compartida y dos locales. Lo que hace es comparar el primer parámetro (a) con la variable compartida (x) y si coinciden, intercambia el valor de la variable compartida (x) y el segundo parámetro (b). En la línea x86 sólo tenemos dos parámetros implícitos, el tercero debe ser explícito usando el registro `%eax`.

```
[linenos]gas lock cmpxchg mem, reg  
[linenos]c Compare_and_swap(a, b, x)if(a == x)temp if eax:x mem b = temp; /*  
xcompartida, aylocales * / then mem ←→ reg  
else eax ←→ mem
```

Cerrojos simples con Test&Set y Fetch&Or

Para implementar un cerrojo simple necesitamos una instrucción que nos permita leer el valor del cerrojo y cerrarlo de forma atómica. Esto se puede hacer con un Test&Set o un Fetch&Or cuyos parámetros fuesen la variable cerrojo y un 1, ya que $1 \text{ OR } 1 = 1$ (true) y $1 \text{ OR } 0 = 1$ (false).

```
[linenos, frame=lines, label=Fetch_and_Oper]c lock(k) while (fetch_and_or(k, 1) == 1); /*  
kcompartida * /
```

Para la línea x86, usamos el intercambio de un registro cuyo valor sea 1 y el cerrojo. Después comprobamos si el valor del cerrojo es un 1 para seguir esperando hasta que sea 0. Esta implementación no da problemas con el modelo de consistencia del x86.

<pre>[linenos, frame=lines, label=Test_and_Set]c lock(k) while(test_and_set(k) == 1); /* kcompartida * /</pre>	<pre>[linenos, frame=lines, label=x86]gas lock mov eax, 1 repetir: xchg eax, k cmp eax, 1 jz repetir</pre>
--	--

Cerrojos simples con Compare&Swap

Con esta implementación, le ponemos un 1 al segundo parámetro (b). Si el cerrojo está abierto se intercambian b y k para obtener en b el valor actual del cerrojo. Si fuese un 1 esperaríamos en la espera ocupada.

```
[linenos, frame=lines, label=Compare_and_swap]c lock(k) b=1; do compare_and_swap(0, b, k); while(b == 1); /* kcompartida, blocal * /
```

Cerrojo simple en Itanium (consistencia de liberación) con Compare&Swap

Hay procesadores que tienen registros de propósito específico llamados *predicados* que se pueden usar delante de instrucciones. Si el predicado es 1, la instrucción se ejecuta y si es 0, no. Los itanium tienen predicados, así, nos ahorraremos instrucciones de salto y ahorraremos tiempo de ejecución debido a la penalización de los saltos.

En el itanium no se garantiza ningún orden en el acceso a memoria, pero ofrece instrucciones que garantizan que hasta que no se han hecho los accesos a memoria anteriores, no se realiza dicha operación. Estas operaciones de acceso a memoria de adquisición y liberación nos las ofrece el itanium para garantizar el orden en los accesos a memoria.

En el unlock se usa un acceso a memoria de liberación, ya que si sólo usáramos `st8`, ésta puede adelantar a operaciones anteriores. Con `st8.rel` ya no:

```
c unclock: // unlock(M[lock]) st8.rel [lock] = r0;; // liberar asignando un 0, en itanium r0 es siempre 0
```

En el lock, se utiliza la alternativa Compare&Swap con sufijo de adquisición para garantizar que hasta que no se hace el acceso a memoria en la operación no se realizan los accesos a memoria que hay detrás.

```
[linenos]c lock: // lock(M[lock]) mov ar.ccv = 0 // cmpxchg compara siempre con ar.ccv // que es un registro de propósito específico mov r2 = 1 // cmpxchg utilizará r2 para poner el cerrojo a 1 spin: // se implementa espera ocupada ld8 r1 = [lock] ;; // carga el valor actual del cerrojo en r1 cmp.eq p1,p0 = r1,r2; // si r1=r2 entonces el cerrojo está a 1 y se hace p1=1 (p1) br.cond.spnt spin; // si p1=1 se repite el ciclo; spnt, indica que se // usa una predicción estática para el salto de no tomar cmpxchg8.acq r1 = [lock],r2;; // intento de adquisición escribiendo 1 // if[lock]=ar.ccv then [lock]<-r2; siempre r1<-[lock] cmp.eq p1,p0 = r1,r2 // si r1!=r2 (r1=0) -> cer. era 0 y se hace p1=0 (p1) br.cond.spnt spin; // si p1=1 se ejecuta el salto
```

Tenemos dos saltos a la etiqueta `spin`, es decir, hay dos bucles. La implementación de un bucle es muy inefficiente porque todos los threads que puedan estar esperando en el lock están ejecutando un Test&Set haciendo una lectura y escritura constantemente en una variable compartida. Si no tienen dicha variable en sus caches, se la traen y al escribir en ella, deben invalidar las copias de las otras caches. Por tanto, tenemos un constante envío de paquetes a través de la red. Como sólo hace falta escribir en el cerrojo si está abierto, en el primer bucle comprobamos si está abierto leyendo su valor (lo cual implica sólo que si el nodo no tiene la variable en su cache se la traiga). Cuando alguien escriba en la variable, volverán a acceder a la red para intentar obtener derecho a la sección crítica. Solamente el primero que ejecute el Test&Set será el primero que entre, y el resto, lo volverán a encontrar cerrado. Así conseguimos que sólo haya una escritura por thread.

3.10.6 Instrucción LL/SC (Load Linked / Store Conditional)

Algunos procesadores ofrecen dos instrucciones, una de escritura y otra de lectura, que permiten simular las anteriores (instrucciones de carga enlazada y almacenamiento condicional). Cuando ejecutamos estas instrucciones, leemos el valor de la variable (LL) y cuando vamos a escribir (SC) comprobamos si algún otro thread ha accedido a dicha variable, esto se hace comprobando un bit que SC resetea tras escribir en dicha variable. Por tanto, si el bit no está a 1 sino a 0 quiere decir que otro thread ha escrito en esa dirección de memoria antes y no se hace el almacenamiento condicional.

Cerrojo simple en PowerPC (consistencia débil) con LL/SC implementando Test&Set

El modelo de consistencia de los PowerPC no garantiza ningún orden, al igual que el del itanium. La diferencia entre ambos está en las instrucciones que ofrecen para garantizar el orden. En el caso del PowerPC tenemos las instrucciones barrera: hasta que no se realizan los accesos anteriores no se hacen los que hay después. Es mejor por tanto el modelo del Itanium pues permite aprovechar más paralelismo.

[linenos]gas lock: lock(M[r3]) li r4, 1 para cerrar el cerrojo bucle: lwarx r5,0,r3 carga y reserva: r5 <- M[r3] cmpwi r5,0 si esta cerrado (a 1) bne- bucle esperar en el bucle, en caso contrario stwcx. r4,0,r3 poner a 1 (r4=1): M[r3] <- r4 bne- bucle el thread repite si ha perdido la reserva isync accede a datos compartidos cuando sale del bucle

Cargamos el valor del cerrojo en r5 y comprobamos si es 0, si no es tenemos que seguir esperando y si no, accedemos a la sección crítica.

[linenos]gas unlock: unclock(M[r3]) sync espera hasta que terminen los accesos anteriores li r1,0 stw r1,0(r3) abre el cerrojo

sync e isync son las instrucciones barrera de PowerPC.

Chapter 4

Arquitecturas con Paralelismo a nivel de Instrucción (ILP)

4.11 Microarquitecturas ILP. Cauces superescalares

4.11.1 Microarquitecturas ILP. Cauces Superescalares

Arquitecturas con DLP, ILP y TLP (thread = flujo de control)

Las arquitecturas con paralelismo a nivel de instrucción son las que estudiaremos en este tema. Abarcan los cores actuales (escalares segmentados, superescalares, VLIW). Ejecutan instrucciones concurrentemente y en paralelo.

Paralelismo entre Instrucciones

Como ya vimos en el Tema 1, en un procesador no segmentado se ejecutan cada una de las etapas de ejecución de una instrucción (captación de instrucción, decodificación de instrucción, ejecución, etc) se ejecutan secuencialmente. Por tanto, la ejecución de una instrucción tarda tiempo de un ciclo de reloj por el número de etapas.

Si tenemos un core segmentado, el procesamiento de las instrucciones se hace en etapas. Entre cada etapa se colocan registros para que las etapas puedan trabajar en paralelo. Así, podemos tener hasta cinco instrucciones en paralelo. Cada etapa supone un ciclo de reloj del procesador. La ejecución de las instrucciones es concurrente porque cada una está en una etapa distinta de procesamiento. Así, la primera instrucción sí que tardará lo mismo que en un procesador no segmentado, pero el resto finalizarán a cada ciclo de reloj. La ejecución de las instrucciones es concurrente porque cada instrucción está en una etapa distinta de procesamiento.

La ejecución paralela de instrucciones se consigue con arquitecturas superescalares o VLIW porque permiten tener varias instrucciones procesándose al mismo tiempo en la misma etapa.

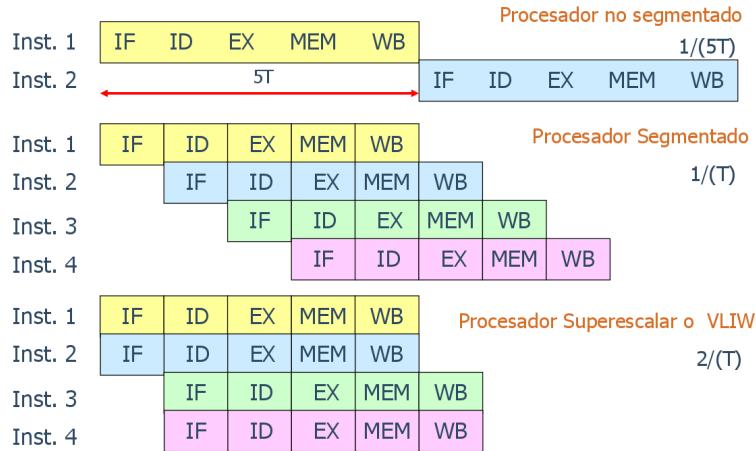


Figure 4.1: Comparativa de la estructura de cauce de los distintos tipos de arquitecturas

El hardware debe estar preparado para trabajar con varias instrucciones a la vez, es decir, para realizar las distintas etapas con varias instrucciones. El tiempo de ejecución se reduce por tanto a la mitad de la duración de un ciclo de reloj.

Procesadores superescalares y VLIW

Los procesadores Superescalares y VLIW comparten las siguientes características (además del hecho de ser procesadores segmentados):

- ♡ Disponen de varias unidades de ejecución
- ♡ Pueden ejecutar varias instrucciones simultáneamente en esas unidades de ejecución
- ♡ Pueden emitir instrucciones en paralelo a unidades de ejecución

Pero en los procesadores superescalares es el hardware quien debe descubrir el paralelismo que se puede aprovechar a partir de las instrucciones que se van captando, **en los procesadores VLIW el paralelismo es explícito** (se captan juntas las instrucciones que se van a emitir juntas a unidades de ejecución).

La micro-arquitectura de los procesadores VLIW es más sencilla (sobre todo en lo que respecta a las etapas de emisión y finalización: buffers de renombrado, ROBs,...) ya que es el compilador el que debe detectar paralelismo al seleccionar las instrucciones que se captarán juntas en la misma palabra de instrucción.

Mejora de las prestaciones de los procesadores

Se han ido añadiendo a los chips de procesamiento nuevos cambios debidos a las mejoras en tecnología de fabricación de circuitos integrados basada en el silicio que han contribuido a la mejora del tiempo de ejecución: la reducción del tamaño de los transistores y aumento del tamaño del dado de silicio:

- ♡ Al haber más transistores por circuito integrado las microarquitecturas son más complejas en un circuito integrado y permiten paralelismo entre instrucciones (procesadores superescalares) (*IPC*)
- ♡ Se reduce la longitud de puerta del transistor y con ello el tiempo de conmutación, así conseguimos mayores frecuencias de funcionamiento ya que el transistor tarda menos tiempo en cambiar de 0 a 1 (*F*).

$$V_{CPU} = IPC \times F$$

Evolución de microarquitecturas

A nivel de core, las mejoras han sido añadir una implementación escalar o segmentada que permiten ejecutar instrucciones en paralelo. Con la reducción de la frecuencia de reloj conseguimos menor tiempo de ejecución. Dividir las etapas en otras subetapas nos consigue también una frecuencia menor, pues la frecuencia de reloj es la duración de una etapa.

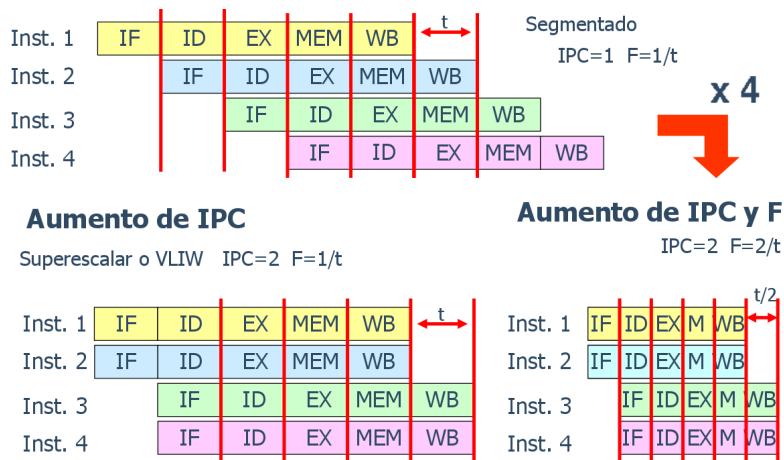


Figure 4.2: Si aumentamos el número de instrucciones ejecutándose concurrentemente y disminuimos la frecuencia de reloj obtenemos un tiempo de ejecución cuatro veces menor

Un core actual puede tener 10 o 15 etapas, esto se debe a que una operación en coma flotante tiene más etapas que cualquier otra operación. Intel implementó un superpipeline con 20 etapas, pero no fue rentable por las penalizaciones que hacían que se redujese el tiempo de ejecución.

4.11.2 Paralelismo entre instrucciones (ILP). Orden en Emisión y Finalización

Paralelismo entre instrucciones (ILP) y paralelismo de la máquina

El paralelismo de instrucciones depende de la frecuencia de las dependencias de datos y control, y del retardo de la operación (tiempo hasta que el resultado de una operación esté disponible). El que se puedan ejecutar más o menos instrucciones en paralelo en un core va a depender de la arquitectura del core y del paralelismo del código.

Por ejemplo, en el siguiente código: [linenos]gas load add add

tenemos un grado de paralelismo de 3, porque las instrucciones no tienen dependencias de datos (RAW, WAW, WAR) entre ellas.

En el siguiente código, en cambio: [linenos]gas add add store (r4),

hay un RAW entre las dos primeras instrucciones pues la segunda requiere del resultado de la primera. Pero es que después hay otro RAW entre la segunda y la tercera. Por tanto, el grado de paralelismo es 1: solo se puede ejecutar una instrucción cada vez debido a las dependencias.

El primer trozo de código se podrá ejecutar en paralelo pero dependiendo del hardware del core se van a poder ejecutar más o menos instrucciones en paralelo. Si sólo tenemos una unidad que haga operaciones de sumas con enteros y otra que haga accesos a memoria, podremos ejecutar dos de las tres instrucciones en paralelo.

El paralelismo de la máquina, viene determinado por el número de instrucciones que pueden captarse y ejecutarse al mismo tiempo (número de cauces paralelos) y por la velocidad y los mecanismos que usa el procesador para encontrar las dependencias entre instrucciones.

Ordenaciones en una secuencia de instrucciones

En un core de procesamiento no tiene porqué coincidir el orden en el que se captan las instrucciones de memoria (orden en el que las ha generado el compilador) con el orden en el que las instrucciones se ejecutan. El orden en el que las instrucciones terminan modificando los registros de la arquitectura tampoco tiene que coincidir. Hay cores en los que el orden en el que se captan las instrucciones coincide con el orden en el que se terminan (por ejemplo, en la línea x86 de Intel). Aunque se terminen en el mismo orden en el que se captan, el orden en el que se ejecutan puede ser distinto. En los PowerPC se terminan las instrucciones en un orden distinto al que se captan.

En una secuencia de instrucciones se pueden distinguir tres tipos de ordenaciones:

- ♡ El orden en que se captan las instrucciones (el orden de las instrucciones en el código)
- ♡ El orden en que las instrucciones se ejecutan
- ♡ El orden en que las instrucciones cambian los registros y la memoria.

El procesador superescalar debe ser capaz de identificar el paralelismo entre instrucciones (ILP) que exista en el programa y organizar la captación, decodificación y ejecución de instrucciones en paralelo, utilizando eficazmente los recursos existentes (el paralelismo de la máquina).

Cuanto más sofisticado sea un procesador superescalar, menos tiene que ajustarse a la ordenación de las instrucciones según se captan, para la ejecución y modificación de los registros, de cara a mejorar los tiempos de ejecución. La única restricción es que el resultado del programa sea correcto.

4.11.3 Cauces superescalares

Decodificación paralela y predecodificación

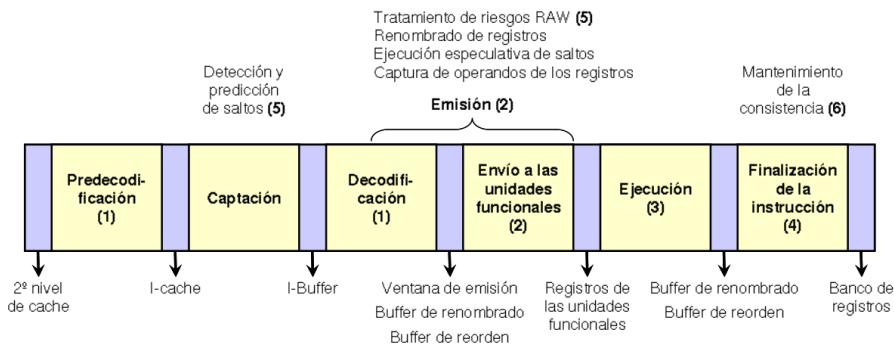


Figure 4.3: Etapas típicas en un cauce

Las etapas típicas de un cauce se muestran en la Figura ?? (podemos dividirlo en más subetapas):

- ♡ **Predecodificación:** captar instrucciones más lentas con antelación. Estas instrucciones suelen ser instrucciones compuestas por otras subinstrucciones, en esta etapa se descomponen en dichas subinstrucciones. Es decir, pasamos de instrucciones CISC a instrucciones RISC.
- ♡ **Captación:** se cogen las instrucciones de la cache L1 y se guardan en el buffer de instrucciones. Se detectan instrucciones de salto y se predice si se va a saltar o no (en el caso de ser condicional) para reducir la penalización que supone el salto.
- ♡ **Decodificación:** se predice qué unidades de ejecución y qué operandos van a usar las instrucciones. Los resultados se meten en el buffer de reorden, de renombrado o en la ventana de emisión.
- ♡ **Envío a unidades funcionales:** se podrán enviar instrucciones si tienen los operandos que necesitan disponibles. Se colocan las instrucciones en los registros de entrada a las unidades funcionales. Esta etapa junto con la anterior forman la etapa de **Emisión**. En la etapa de emisión se tratan los riesgos RAW y se renombran registros para tratar los riesgos WAR y WAW.
- ♡ **Ejecución:** al final de la ejecución, se modifica el buffer de renombrado y el de reorden
- ♡ **Finalización de la instrucción:** por último, se finaliza la ejecución de las instrucciones modificando el banco de registros de la arquitectura. Estos registros son los que usamos cuando programamos en ensamblador. En esta última etapa se mantiene la consistencia.

La etapa de decodificación de un procesador superescalar debe ser capaz de decodificar varias instrucciones por ciclo y comprobar la disponibilidad de sus operandos. También hay que determinar a qué unidades funcionales enviamos cada instrucción. Por ello, mientras que en un procesador segmentado hay una etapa de decodificación de la instrucción y de búsqueda de los operandos, en un procesador superescalar existen unidades de decodificación y de emisión de instrucciones separadas.

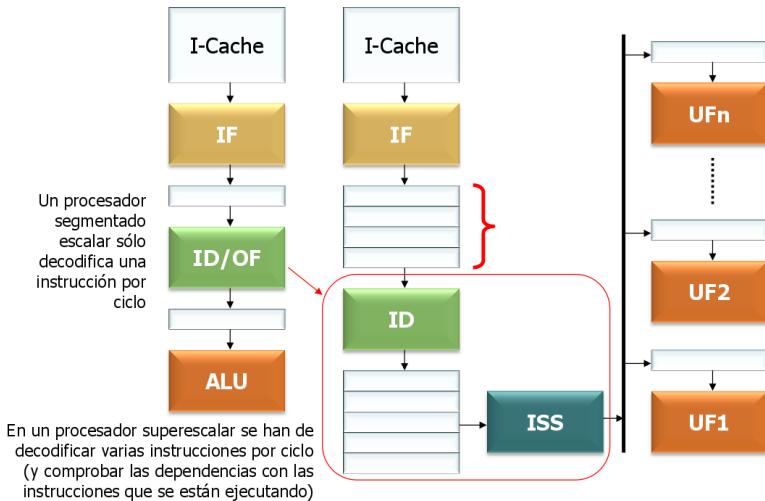


Figure 4.4: Comparación entre las primeras etapas de un procesador escalar y un procesador superescalar

Como se ve en la Figura ??, en un procesador superescalar los registros entre cada etapa son mayores que en un procesador escalar, pues tenemos que meter múltiples instrucciones. Además, en el caso de un superescalar, la ALU se divide en varias unidades funcionales con sus registros propios de entrada.

Dada la cantidad de instrucciones que hay que decodificar por ciclo en un superescalar, se usan varias etapas de decodificación (sobre todo en procesadores CISC). Una de las etapas de decodificación se implementa entre la cache de segundo nivel y la de instrucciones de primer nivel (*etapa de predecodificación*), como se ve en la Figura ??.

En la Figura ?? tenemos con más detalle un cauce superescalar. La emisión usa también (junto a la última etapa) los registros de la arquitectura.

Los bits que se añaden en la etapa de predecodificación (*bits de predecodificación*) suelen indicar:

- ♥ Si es una instrucción de salto o no (se puede empezar su procesamiento antes)
- ♥ El tipo de unidad funcional que va a utilizar (se puede emitir más rápidamente si hay cauces para enteros o coma flotante...)
- ♥ Si hace referencia a memoria o no

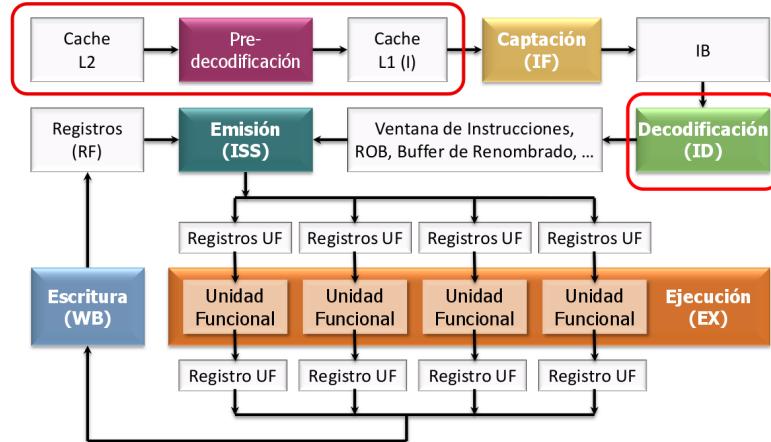


Figure 4.5: Etapas del cauce de un procesador superescalar

Emisión paralela de instrucciones. Estaciones de reserva

La ventana de instrucciones almacena las instrucciones pendientes: todas si la ventana es centralizada, o las de un tipo determinado si es distribuida.

Las instrucciones se cargan en la ventana una vez decodificadas. Se utiliza un bit para indicar si un operando está disponible (se almacena el valor o se indica el registro desde donde se lee) o no (se almacena la unidad funcional desde donde llegará el operando).

Una instrucción puede ser emitida cuando tiene todos sus operandos disponibles y la unidad funcional donde se procesará. Hay diversas posibilidades en el que varias instrucciones estén disponibles (características de los buses, etc.)

Ejemplo de Ventana de Instrucciones							
#	opcode	address	rb_entry	operand1	ok1	operand2	ok2
2	MULTD	loop + 0x4	2	1	0	0	0
1	LD	loop	1	0	0	0	1

Explicación de los campos:

- Lugar donde se almacenará el resultado:** Indicado por un cuadro rojo en la columna rb_entry de la fila LD.
- Dato no válido (índica desde dónde se recibirá el dato):** Indicado por un cuadro rojo en la columna ok1 de la fila LD.
- Dato válido (igual a 0):** Indicado por un cuadro rojo en la columna ok2 de la fila LD.

Figure 4.6: Ejemplo de ventana de instrucciones

En la Figura ?? se ve un ejemplo de ventana de instrucciones. Sus campos son:

- ♡ **codop:** indica la operación correspondiente a la instrucción que se ha decodificado y almacenado en esa línea de la ventana.
- ♡ **dest:** indica el lugar donde se escribirá el resultado de la instrucción tras ejecutarse, dicho lugar no tiene porqué ser el registro que se indique en la instrucción sino que podría almacenarse en un registro temporal, que se indicará en este campo.
- ♡ Los siguientes campos están dedicados a almacenar los operandos de dicha instrucción, habrá tantos como operandos puedan intervenir en una instrucción (en la Figura ?? se han supuesto dos). Asociado a cada operando hay dos campos:

- **ok**: indica si el valor almacenado en el campo de operando corresponde al valor de ese operando, es decir, indica si el operador está disponible para iniciar la ejecución (si está disponible su valor será 1 y si no, 0). Cuando todos los campos **ok** de un operando sean 1 la instrucción podrá emitirse siempre y cuando la unidad funcional donde se va a ejecutar esté disponible.
- **operando**: si el valor del campo **ok** relacionado con dicho operando es 1, este campo guardará su valor; en caso contrario, guardará el código correspondiente a la unidad funcional, de almacenamiento, etc que proporcionará ese valor cuando haya sido calculado.
- ♡ **tipo**: indica si el registro al que hace referencia el campo **operando** cuando **ok=0** es un registro para enteros o para coma flotante o es un dato inmediato.

Las políticas de emisión de instrucciones pueden clasificarse atendiendo al:

- ♡ **Alineamiento** de la ventana de instrucciones:
 - **emisión alineada**: no se pueden introducir nuevas instrucciones en la ventana de instrucciones hasta que la ventana quede vacía, es decir, hasta que no se hayan emitido todas las instrucciones del ciclo anterior.
 - **emisión no alineada**: mientras exista espacio en la ventana se pueden ir introduciendo instrucciones para ser emitidas.
- ♡ **Orden** en que se emiten las instrucciones a las unidades funcionales:
 - **emisión ordenada**: se respeta el orden en el que las instrucciones se han ido introduciendo en la ventana coincide con el orden en el que las instrucciones están en el programa. Así, si una instrucción de la ventana no puede emitirse las siguientes tampoco podrán, es decir, existe **bloqueo** entre instrucciones.
 - **emisión desordenada**: no existe bloqueo ya que pueden emitirse todas las instrucciones que dispongan de sus operandos y de su unidad funcional.

Ahora pongamos un ejemplo para ver el efecto de la emisión ordenada y la emisión desordenada. La secuencia de instrucciones es la siguiente: [linenos]gas add mult sub sub

En la emisión ordenada que se muestra en la Figura ??, primero se emiten las instrucciones de las dos primeras líneas ya que tienen sus operandos y las unidades funcionales disponibles. Las dos últimas instrucciones no pueden lanzarse porque la instrucción de la línea 3 bloquea a la de la línea 4. Si tuviésemos emisión desordenada (Figura ??) podríamos emitir la instrucción de la cuarta línea antes que la de la tercera.

No obstante, los tiempos de ejecución obtenidos son iguales en ambos casos, es decir, no afecta al tiempo de ejecución el tener una emisión ordenada o desordenada porque ambas tardan 9 ciclos en ejecutarse.

Sin embargo, si consideramos que sólo tenemos una unidad de suma y resta (Figura ?? y Figura ??) sí que se producen diferencias en los tiempos de ejecución (tres ciclos menos en el caso de la emisión desordenada): si las instrucciones no se emiten porque hay bloqueos, puede ocurrir que cuando desaparezcan los bloqueos aparezcan colisiones porque no hay suficientes recursos para todas las instrucciones que pueden emitirse. Si se permite emisión desordenada podemos utilizar la unidad de emisión con más eficiencia.

En la práctica la emisión tiene dos etapas ya que se pasa entre medias en lo que se llama estación de reserva:

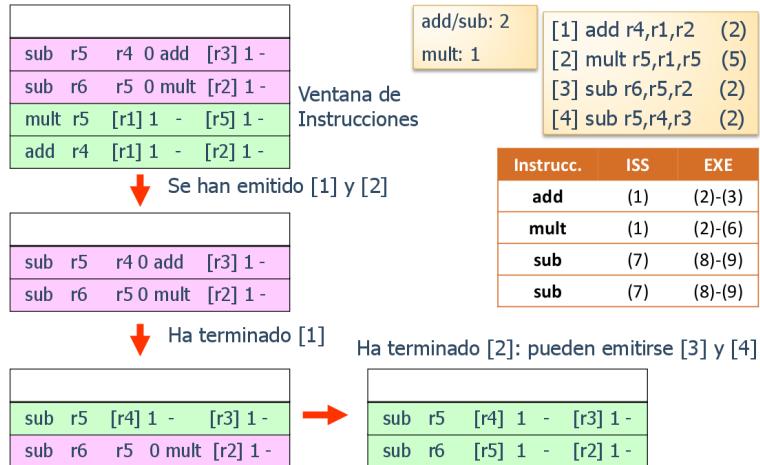


Figure 4.7: Emisión paralela de instrucciones ordenada con dos unidades de suma y resta y una de multiplicación

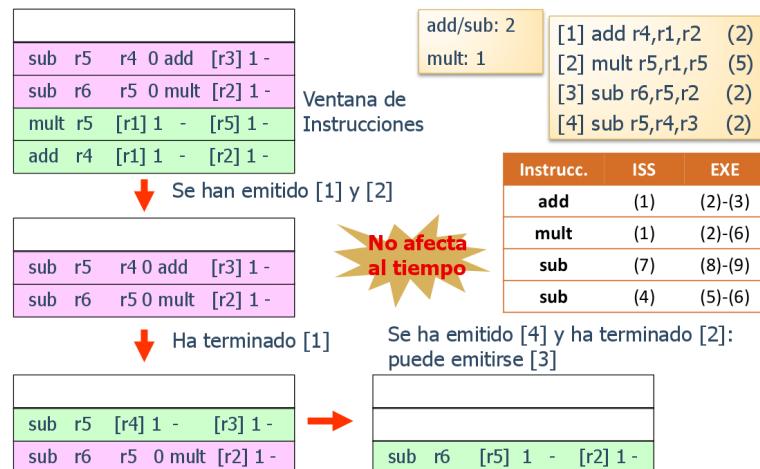


Figure 4.8: Emisión paralela de instrucciones desordenada con dos unidades de suma y resta y una de multiplicación

♡ Emisión

♡ Envío de reserva a las unidades funcionales

Las *estaciones de reserva* son como una especie de ventana pero específica para las unidades funcionales. Hay varias alternativas para su implementación: una única estación de reserva para todas las unidades funcionales (equivale a tener una ventana normal y corriente), una estación de reserva por unidad funcional, una estación de reserva compartida entre varias unidades funcionales, etc (Figura ??).

Si no existen limitaciones en el hardware las instrucciones se emiten a las estaciones de reserva independientemente de las dependencias. Las instrucciones esperan hasta que se resuelvan las

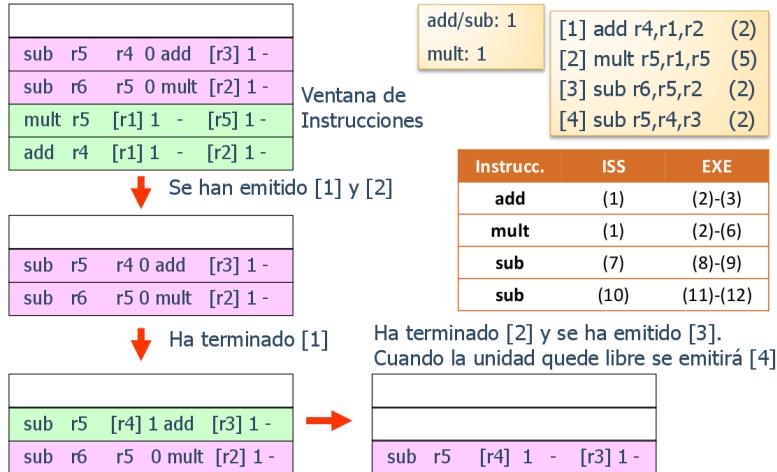


Figure 4.9: Emisión paralela de instrucciones ordenada con una unidad de suma y resta y otra de multiplicación

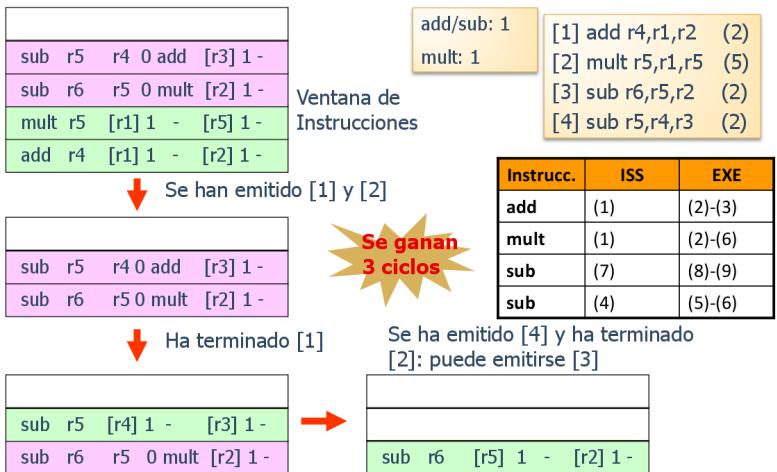


Figure 4.10: Emisión paralela de instrucciones desordenada con una unidad de suma y resta y otra de multiplicación

dependencias y se envían a las unidades funcionales una vez comprobada la disponibilidad de la unidad, cuando sea su turno.

En la figura Figura ?? se ve la decodificación/emisión de instrucciones desde la cola de instrucciones a las estaciones de reserva. Se ha supuesto una estación de reserva para el multiplicador y otra para la suma/resta, que cada estación puede guardar dos instrucciones y que se pueden emitir tres instrucciones desde la cola de instrucciones. Inicialmente, de las cuatro instrucciones que hay sólo podemos emitir tres.

Las distintas alternativas para el envío a las unidades funcionales son:

⌚ **Reglas de selección:** se determina las instrucciones que pueden enviarse, es decir, las que

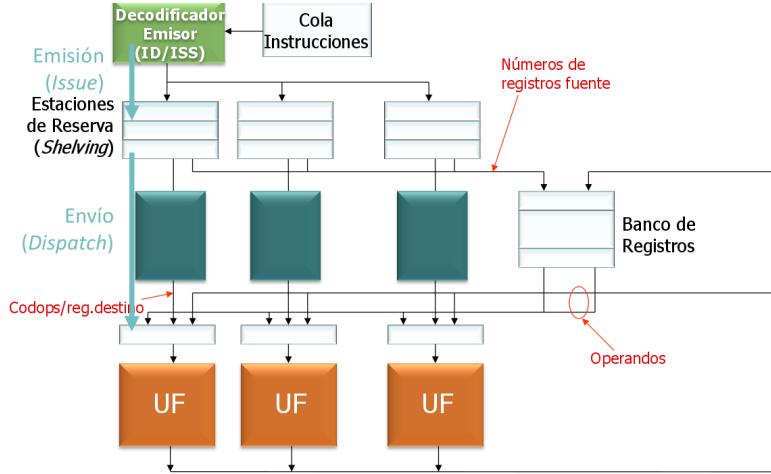


Figure 4.11: Microarquitectura con estaciones de reserva

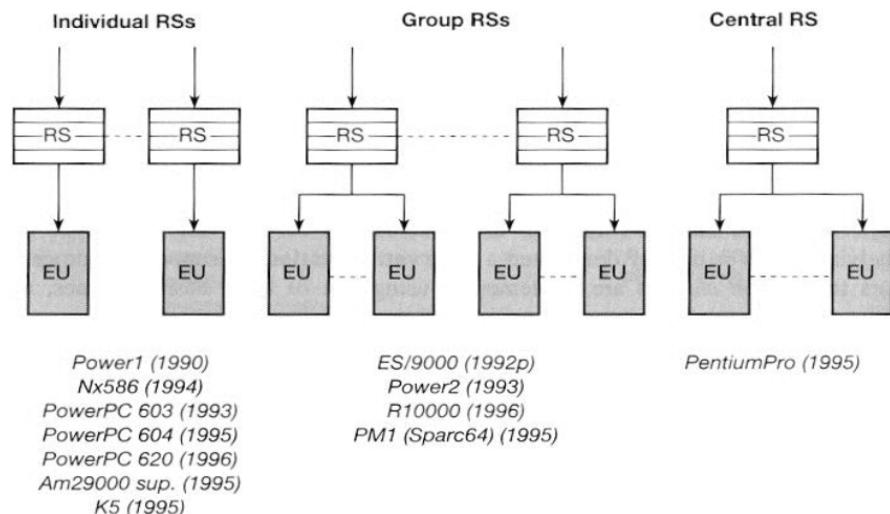


Figure 4.12: Tipos de estaciones de reserva

se pueden ejecutar.

- ♡ **Reglas de arbitraje:** instrucción que se envía si hay varias ejecutables, normalmente se suele enviar la que lleva más tiempo esperando
- ♡ **Orden de envío:** hay varias alternativas: ordenadas, desordenadas o parcialmente ordenadas (ciertas instrucciones no ejecutables bloquean instrucciones de un tipo, pero no de otros).
- ♡ **Velocidad de envío:** se corresponde con el número de instrucciones que se envían por ciclo. Podremos enviar más de una instrucción si no tenemos la implementación de una estación de reserva por unidad funcional.

Suponiendo una captación de operando en la emisión en lugar del envío y que las instrucciones

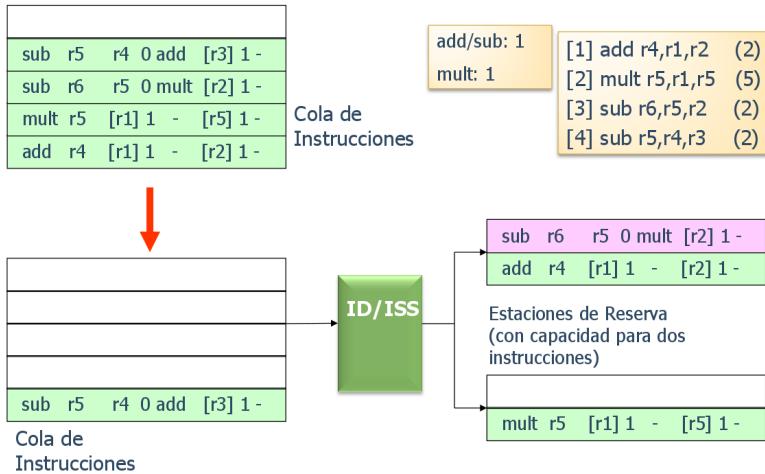


Figure 4.13: Ejemplo de emisión de instrucciones a las estaciones de reserva

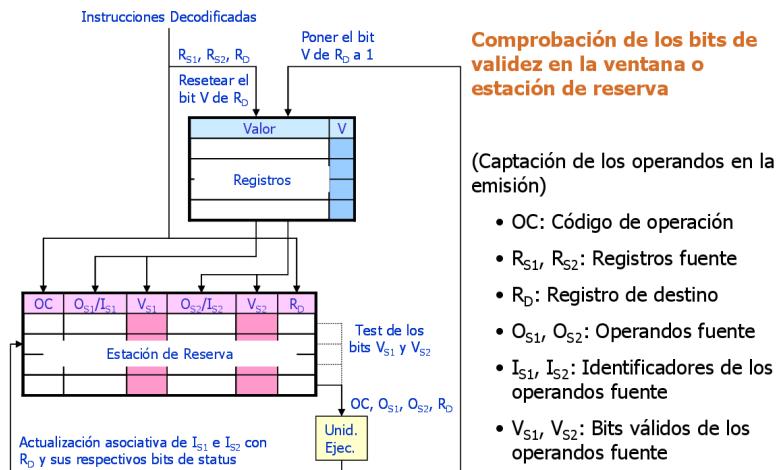


Figure 4.14: Comprobación de los operandos

tienen dos campos de entrada y uno de salida, tenemos los mismos campos que teníamos en la ventana respecto a los operandos: uno de estado y otro con su valor. Los registros de la arquitectura tienen un valor de válido si el valor que contienen es válido, será inválido cuando hayamos emitido una instrucción cuyo resultado se almacenará en dicho registro y no se haya terminado la operación.

Como ejemplo, tenemos la situación de la Figura ??, donde suponemos que la multiplicación necesita cinco ciclos y la suma, dos. En el ciclo i se va a emitir la multiplicación. Sólo podremos emitir la multiplicación debido a las dependencias. En el siguiente ciclo, podremos emitir las dos siguientes instrucciones a la vez, que pasan a las estaciones de reserva. Se envían al banco de registros los registros que usarán para invalidarlos. También necesitamos que el banco de registros nos responda con los parámetros de entrada de las instrucciones para colocar su valor y saber si están válidos o no.

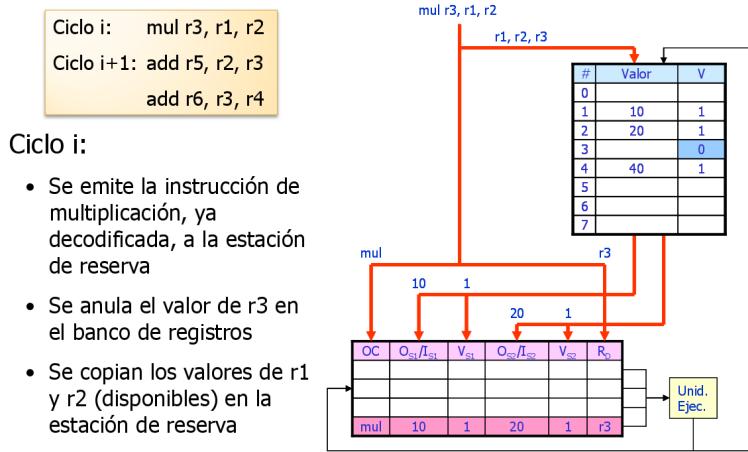


Figure 4.15: Ejemplo de uso de estaciones de reserva. Estado inicial

Los dos operandos que necesita la multiplicación son válidos, por tanto en el siguiente ciclo ($i+1$) la multiplicación se ejecuta, las operaciones de suma pasan a las estaciones de reserva cogiendo de los registros de la arquitectura los valores de sus parámetros de entrada y dejan como inválidos los registros que modifican. Respecto a los datos que necesitan las sumas, ambas necesitan el registro r3, pero como está siendo usado por la multiplicación deberán esperar a que ésta termine (Figura ??)

En $i+5$ ciclos, se almacenará el resultado de la multiplicación tanto en el banco de registros como en la estación de reserva y en el siguiente, las dos instrucciones restantes estarán válidas, pero como sólo tenemos una unidad para sumas/restas no podemos ejecutarlas a la vez. Por tanto, se ejecutará la primera suma, guardará su resultado en r5 y pasará a la otra.

Renombramiento de registros

Todos los cores usan renombramiento de registros para eliminar riesgos WAR y WAW.

Por ejemplo, en el siguiente código:

[linenos]pascal R3 := R3 - R5 R4 := R3 + 1 R3 := R5 + 1 R7 := R3 * R4

tenemos una dependencia WAW entre la primera y la tercera instrucción y una WAR entre la segunda y la tercera. Por tanto, no podemos ejecutar ambas instrucciones en paralelo porque tendríamos resultados erróneos. Si no usásemos renombramiento de registros tendríamos que esperar tres ciclos para ejecutar la tercera instrucción. Para ello, se añade un buffer a la arquitectura con los registros que se van a usar y así, cuando se emite una instrucción se le asigna una entrada del buffer de renombrado:

[linenos]pascal R3b := R3a - R5a R4a := R3b + 1 R3c := R5a + 1 R7a := R3c * R4a

En este código sólo tendremos dependencias RAW.

El renombramiento de registros se puede implementar de dos maneras:

♡ **Implementación estática:** el renombramiento se hace durante la compilación

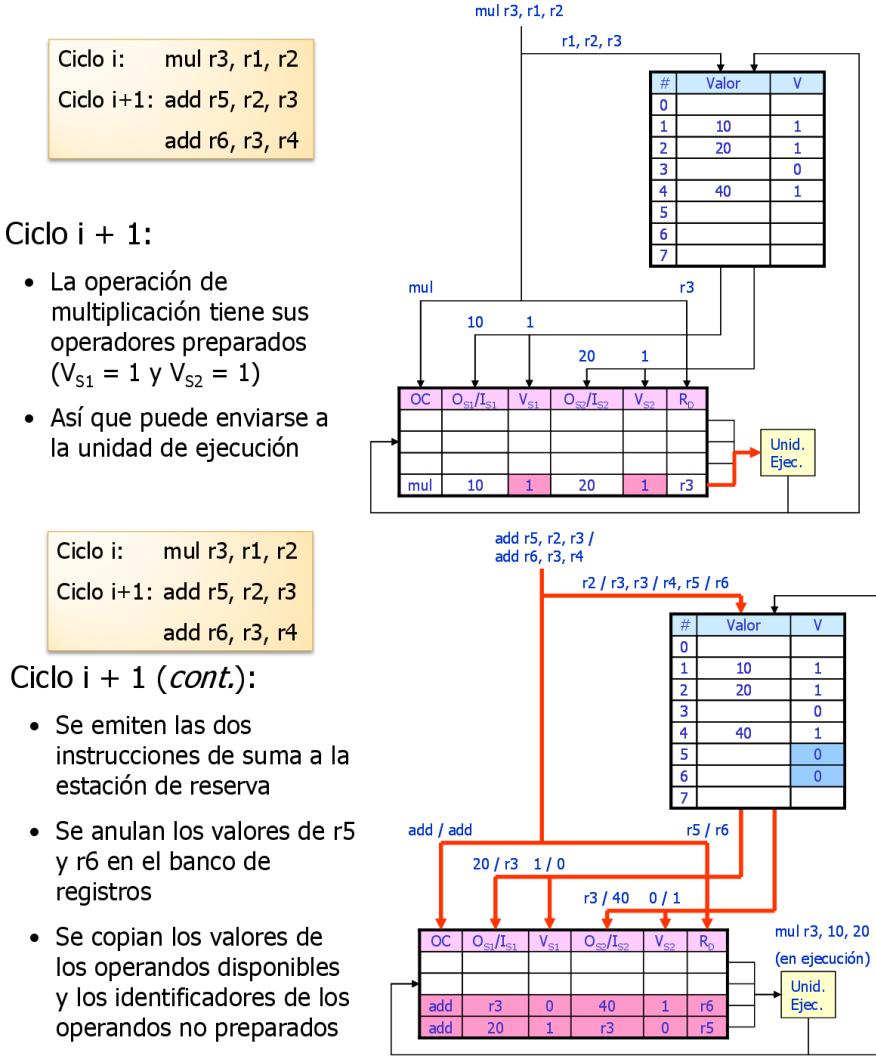


Figure 4.16: Ejemplo de uso de estaciones de reserva. Segundo paso

♡ **Implementación dinámica:** el renombramiento se hace durante la ejecución, lo cual implica añadir circuitería adicional y registros extra a nuestra arquitectura.

Los buffers de renombrado se caracterizan por:

- ♡ **Tipos de buffers**, que pueden estar separados o mezclados con los registros de la arquitectura
- ♡ **Número de buffers de renombrado**
- ♡ **Mecanismos para acceder a los buffers** que pueden ser asociativos o indexados

La *velocidad de renombrado* es el máximo número de nombres asignados por ciclo que admite el procesador.

	Entrada Válida	Registro Destino	Valor	Valor Válido	Último
	1	5	50	1	1
	1	12	1200	1	1
Búsq. asoc. de r2	1	2	20	1	1
	1	1	3	1	1
	:	:	:	:	:

Figure 4.17: Estructura del buffer de renombrado

Los campos que tiene el buffer de renombrado son (Figura ??):

- ♡ *Entrada válida*: Campo que indica si se está usando una entrada o no
- ♡ *Registro destino*: Registro de la arquitectura al que está renombrando
- ♡ *Valor*: Campo para almacenar el resultado de la operación
- ♡ *Valor válido*: Campo para saber si el dato está calculado o no
- ♡ *Último*: el bit de último se utiliza para marcar cuál ha sido la escritura más reciente en ese registro

El buffer de renombramiento permite varias escrituras pendientes a un mismo registro, pero el hardware sólo necesita conocer la última vez que se ha escrito en dicho registro porque el valor de la última vez que se ha usado será el que se pase a los registros de la arquitectura. Los registros de la arquitectura se modifican de forma ordenada (en el orden del programa). Si hay instrucciones en el buffer que están después en el código que otras que aún no se han calculado, no se podrán escribir hasta que las anteriores se calculen.

En la Figura ?? tenemos un ejemplo de renombramiento. Entre la instrucción primera y la tercera hay una dependencia WAW. En la situación inicial tenemos renombrados los registros 4, el 0, dos veces el 1.

En el ciclo i (Figura ??) se emite la multiplicación y renombramos el registro $r2$ pues también es accedido por la tercera operación. En esta figura se ve también que el buffer de renombramiento devuelve valores: si los registros aparecen en las operaciones devuelven los últimos valores de esos registros si están válidos y se envían a las estaciones de reserva. Si los registros no están en el buffer se cogen sus valores del banco de registros.

Cuando acabe la multiplicación, se actualizará el buffer de renombrado y podremos ejecutar la suma que leía el valor $r2$ que devolvía la multiplicación. Pero en el banco de registros, el valor de $r2$ que se escribirá será el que devuelva la última operación (la resta).

	Entrada Válida	Registro Destino	Valor	Valor Válido	Último
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	0
3	1	1	15	1	1
4	0				
5	0				
6	0				
7	0				
8	0				
9	0				
10	0				
11	0				
12	0				
13	0				
14	0				

mul r2, r0, r1
add r3, r1, r2
sub r2, r0, r1

Situación Inicial:

- Las instrucciones se emiten de forma ordenada y de una en una, pero se pueden enviar de forma desordenada
- Existen dos renombrados de r1 en las entradas 2 y 3 del buffer
- La última está en la entrada 3
- Una estación de reserva para add/sub y otra para mul

Figure 4.18: Ejemplo de renombramiento. Situación inicial

	Entrada Válida	Registro Destino	Valor	Valor Válido	Último
0	1	4	40	1	1
1	1	0	0	1	1
2	1	1	10	1	0
3	1	1	15	1	1
4	1	2		0	1
5	0				
6	0				
7	0				
8	0				
9	0				
10	0				
11	0				
12	0				
13	0				
14	0				

mul r2, r0, r1
add r3, r1, r2
sub r2, r0, r1

Ciclo i:

- Se emite la multiplicación
- Se accede a los operandos de la multiplicación, que tienen valores válidos en el buffer de renombrado
- Se renombra r2 (el destino de mul)

Figure 4.19: Ejemplo de renombramiento. Ciclo *i*

4.12 Consistencia del procesador y procesamiento de saltos

4.12.1 Consistencia. Reordenamiento

En el procesamiento de una instrucción se puede distinguir entre:

- ♡ El *Final de la Ejecución de la Operación* codificada en las instrucciones, se dispone de los resultados generados por la unidad funcional pero no se han modificado los registros de la arquitectura
- ♡ El *Final del procesamiento de la instrucción* o momento en el que se completa la instrucción, se escriben los resultados de la operación en los registros de la arquitectura. Si se usa un

123 de ??

buffer de reorden se usa el término de retirar la instrucción, en lugar de completar.

La consistencia de un programa se refiere a:

- ♡ El orden en que se completan las instrucciones
- ♡ El orden en que se accede a memoria para leer (LOAD) o escribir (STORE)

Cuando se ejecutan instrucciones en paralelo, el orden en el que termina (finish) esa ejecución puede variar según el orden que las correspondientes instrucciones tenían en el programa pero debe existir consistencia entre el orden en que se completan las instrucciones y el orden secuencial que tienen en el código del programa.

Puede ocurrir que en los procesadores, el orden en el que terminan de ejecutarse no sea el orden el que modifican los registros. La tendencia es que haya un tratamiento distinto en las operaciones que son accesos a memoria y las que operan con registros de la arquitectura. En el caso de las operaciones con registros, la tendencia es terminarlas en el orden en el que las ha generado el compilador, para conseguirlo, se usa el buffer de reorden. Con el resto de instrucciones de acceso a memoria se ejecutan desordenadamente porque ningún procesador garantiza consistencia secuencial. Estos accesos desordenados evitan penalizaciones por fallos de cache.

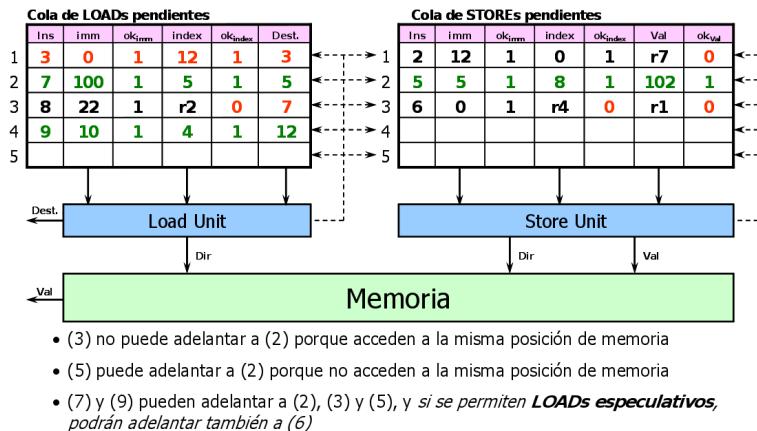


Figure 4.20: Reordenamiento load/store

En la Figura ?? vemos la cola de LOADs y STOREs pendientes. En el caso de los almacenamientos tenemos un campo donde se almacena el dato a escribir en memoria y otro donde se indica si el campo ya está calculado o no. Si no está calculado tenemos el número de registro donde se calculará. La dirección de acceso la calculamos con $im + index$.

En el caso de las lecuras, tenemos los mismos campos, menos el campo destino.

La primera instrucción en el orden del programa no se puede ejecutar porque le falta el valor a guardar en memoria. La siguiente no podría ejecutarse aunque el procesador permitiese que las lecturas adelantases a las escrituras anteriores porque hay una dependencia RAW entre ambas: la primera escribe en la dirección 12 y la segunda lee el valor de dicha dirección. La siguiente sí podría ejecutarse porque tiene los datos disponibles, si el procesador relajase los órdenes $W \rightarrow W$ y $W \rightarrow R$.

La instrucción séptima se podría ejecutar porque tiene los datos disponibles pero tiene un problema, no se sabe qué registro modificará la operación anterior (su campo ok_{index} está a 0) y puede ser que modifique el registro que lee la operación séptima. Sólo podría ejecutarse si el core permite **cargas especulativas**.

Cuando el core implementa *especulación*, hace la lectura pero debe añadir hardware que compruebe que se ha modificado dicha dirección. Si una instrucción anterior escribe en la dirección donde se acababa de leer, se volvería a emitir dicha lectura.

Buffer de renombramiento

Para ejecutar las instrucciones que están en memoria en orden, se usa el *Buffer de Renombramiento (ROB)*. El ROB se gestiona como si fuese una cola, el puntero de cabecera apunta a la siguiente posición libre y el puntero de cola a la siguiente instrucción a retirar.

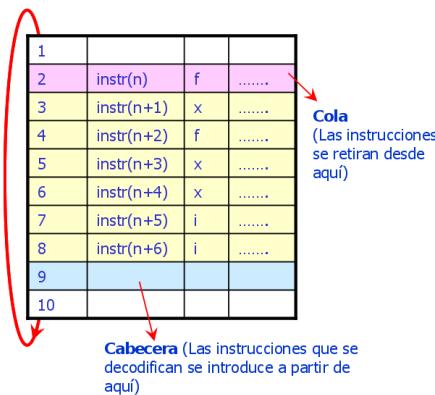


Figure 4.21: Esquema del ROB

Las instrucciones se van colocando en el ROB en el orden del programa, es decir, en el orden en el que las ha generado el compilador. Las instrucciones pueden estar marcadas como:

- ♡ **Emitidas (issued):** i
- ♡ **En ejecución:** x
- ♡ **Finalizada su ejecución:** f

Una instrucción sólo se pueden retirar (se produce la finalización con la escritura en los registros de la arquitectura) si ha finalizado su ejecución y han finalizado también todas las instrucciones que le preceden, ya que deben retirarse de forma ordenada. Por ejemplo, en la Figura ?? no podríamos retirar la instrucción 4, porque la 3 aún no ha terminado su ejecución. Se pueden retirar varias instrucciones a la vez.

La consistencia se mantiene porque sólo las instrucciones que se retiran del ROB se completan (escriben en los registros de la arquitectura) y se retiran en el orden estricto del programa.

La última etapa del cauce (Figura ??), coge las instrucciones del ROB que están en la cola y han terminado y guarda los resultados en el banco de registros de la arquitectura.

El ROB se usa también como buffer de renombrado. En el ROB tenemos dos campos para mantener el orden de las instrucciones que sirven para el renombre: uno para ver si el valor ha sido calculado y otro con el valor, o en su defecto, la línea del ROB o registro de la arquitectura en el que se guardará.

Ejemplo de uso del ROB

[linenos]gas mult st add xor

Entre las dos primeras instrucciones hay un RAW, de igual manera que lo hay entre las dos últimas. Hay un WAW entre la primera y la tercera instrucción y un WAR entre la segunda y la tercera.

Ciclo 7

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	-	0	x
6	xor	10	r1	int_alu	-	0	i

Ciclo 9 No se puede retirar add aunque haya finalizado su ejecución

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	-	0	x

Ciclo 10 Termina xor, pero todavía no se puede retirar

#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	-	0	x
4	st	8	-	store	-	0	i
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

Figure 4.22: Ejemplo de uso del ROB

Estación de Reserva (Unidad de Almacenamiento)

codop	dirección	op1	ok1
st	0x1ca	3	0

Estación de Reserva (ALU)

codop	dest	op1	ok1	op2	ok2
xor	6	5	0	[r3]	1

Líneas del ROB

Figure 4.23: Estaciones de reserva del ejemplo ROB

Imaginemos que ya se han emitido todas las instrucciones, es decir, están todas en el ROB (Ciclo 7 de la Figura ??). Además, las que tenían sus operandos disponibles (la primera y la tercera) ya han pasado a las unidades funcionales (es decir, su estado es *x*). Las demás están en estado de emisión (*i*) pues necesitan datos que aún no han sido calculados, están en las estaciones de reserva (Figura ??).

Con respecto a los operandos, r3 está disponible, r1 no, por lo que en su campo ok hay un cero. Cuando no están disponibles los operandos y usamos ROB, ponemos el campo del ROB en el que se guardará el dato una vez calculado.

Con respecto al almacenamiento, está en la estación de reserva de almacenamiento, como en el campo de **ok** tenemos un cero, en el campo del operando tenemos la línea del ROB donde se almacenará el resultado cuando esté calculado, la entrada de la multiplicación que es el número 3.

Cuando la suma termine, el dato que ha calculado se enviará no sólo al ROB sino también a las estaciones de reserva por lo cual tendremos un 1 en el campo **ok** y el dato ya calculado. Cuando todos los campos **ok** estén a 1, la **xor** pasa a calcularse.

La multiplicación tarda 5 ciclos, mientras que la suma sólo 2, por tanto, en el ciclo 9 ya ha terminado. Su resultado se guarda en el ROB y en la estación de reserva en la que tenemos el campo de operando con la línea de ROB de la suma. Como la **xor** ya tiene sus operandos disponibles, pasa a ejecutarse y como sólo tarda un ciclo, en el ciclo 10 ya ha terminado.

Aunque hemos terminado la suma no podemos retirarla porque las instrucciones se retiran en el orden del programa, es decir, tenemos que esperar a que termine la multiplicación para modificar los registros de la arquitectura en el orden del programa y garantizar la consistencia.

Ciclo 12							
#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
3	mult	7	r1	int_mult	33	1	f
4	st	8	-	store	-	1	f
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

Ciclo 13							
#	codop	Nº Inst.	Reg. Dest.	Unidad	Resultado	ok	marca
5	add	9	r1	int_add	17	1	f
6	xor	10	r1	int_alu	21	1	f

- Se ha supuesto que se pueden retirar dos instrucciones por ciclo.
- Tras finalizar las instrucciones **mult** y **st** en el ciclo 12, se retirarán en el ciclo 13.
- Despues, en el ciclo 14 se retirarán las instrucciones **add** y **xor**.

Figure 4.24: Segunda parte del ejemplo

Cuando termina la multiplicación (Ciclo 12 Figura ??) se guarda su resultado en el ROB y en la estación de reserva de almacenamiento para que comience a ejecutarse la instrucción **st**.

Así, podemos empezar a retirar instrucciones (2 por ciclo, aunque hoy en día podrían ser 4) de la cabecera del ROB.

4.12.2 Procesamiento especulativo de saltos

Clasificación de saltos

Hay saltos condicionales e incondicionales. Dentro de los saltos incondicionales, hay dos tipos: llamadas a subrutinas y retorno de subrutinas.

Con respecto a los condicionales, se usan para implementar bucles, se implementan con saltos hacia delante y hacia detrás.

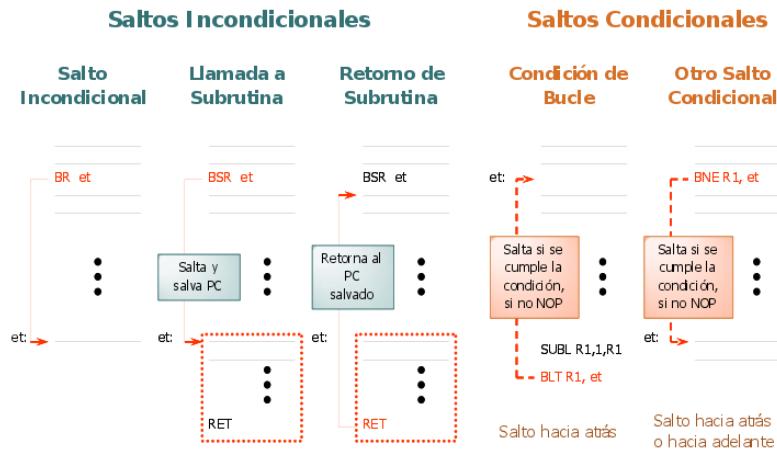


Figure 4.25: Clasificación de los saltos

Por ejemplo, un salto hacia delante podría usarse para implementar un bucle **for**:

[linenos]gas for: cmp esi, N comparamos el indice del bucle con el numero de iteraciones jz fin _____ instrucciones dentro del bucle _____ incl esi jmp for fin: _____ instrucciones fuera del bucle _____

Como se ve, hacemos un salto hacia delante para salir del bucle. Para la implementación de bucles **do-while** se usan saltos hacia atrás, pues como mínimo ejecutamos las instrucciones del bucle una vez:

[linenos]gas dowhile: _____ instrucciones dentro del bucle _____ cmp esi, N jnz dowhile

Alternativas para la condición de salto

♡ **Estado del resultado:** en la línea x86 de Intel, cuando se ejecuta una instrucción se modifica un bit de estado de forma que, en realidad, la instrucción **cmp** modifica los bits de estado y **jz** consulta el valor de dichos bits. Por ejemplo:

[linenos]gas add r1, r2, r3 beq cero div r5, r4, r1 _____ cero:

Para evitar dividir por cero se saltaría a la etiqueta **cero**. Lo que hace la instrucción **beq** es consultar el bit de estado generado por la instrucción anterior. Es un RAW implícito pues necesita el resultado de la anterior operación para ejecutarse.

♡ **Comprobación directa:** la comprobación necesita una o dos instrucciones. No se usan registros de estado:

→ En caso de usar dos instrucciones, se usa la comparación modificando un registro con el resultado de la comparación. En este caso, la dependencia RAW es explícita:

[linenos]gas add r1, r2, r3 cmpeq r7, r1, 0 bt r7, cero div r5, r4, r1 _____ cero:

→ En una instrucción se indica qué registro se quiere consultar, el registro consultado se usa para guardar resultados y NO es un registro de propósito específico. También tenemos una dependencia RAW explícita.

[linenos]gas add r1, r2, r3 bz r1, cero div r5, r4, r1 —————— cero:

Aspectos del Procesamiento de Saltos en un procesador Superescalar

Los saltos penalizan la ejecución de instrucciones sobre todo en procesadores superescalares. Las instrucciones de salto requieren calcular la dirección a la que se va a saltar y el cálculo de dicha dirección no se produce hasta la etapa de ejecución del cauce. Esto quiere decir que hasta la etapa de ejecución no podremos captar la siguiente instrucción.

Cuando se capta una instrucción de salto, al pasar a la segunda etapa se capta la siguiente instrucción a no ser que cambiemos el contador del programa, como no se puede cambiar hasta la etapa de ejecución, tendremos que descartar las dos instrucciones que captamos entre medias y serán dos ciclos de reloj perdidos. En los procesadores superescalares, como captamos más de una instrucción por ciclo, perderemos más instrucciones aún.

El efecto de los saltos en procesadores superescalares es más pernicioso ya que, al emitirse varias instrucciones por ciclo, prácticamente en cada ciclo puede haber una instrucción de salto.

Para disminuir la penalización de los saltos, se detectan las instrucciones de salto en la etapa de captación o incluso antes. También debemos calcular la dirección de salto, podemos guardar la dirección a la que se ha saltado en ejecuciones anteriores en una cache del procesador que no es más que una tabla en la que hay una entrada para almacenar saltos identificándolos por su dirección en el código.

Los saltos condicionales tienen un problema adicional: resolver la condición de salto para saber si se va a ejecutar la instrucción a continuación o la instrucción a partir de la dirección de salto. Se suele añadir hardware en la etapa de captación para predecir si se va a saltar o no, si la predicción es errónea, habrá penalización.

Actualmente, los cores superescalares incluyen *procesamiento especulativo de saltos*. Como alternativas están:

1. **Múltiples caminos**: se decodifican y ejecutan instrucciones de las dos ramas posibles, disminuyendo la velocidad con la que se ejecutan las instrucciones, pues tendremos que descartar una de las dos ramas.
2. **Bloqueo del procesamiento del salto**: en los procesadores antiguos se bloqueaba la captación de instrucciones hasta que no se resolvía la condición.
3. **Ejecución vigilada**: se han añadido instrucciones para evitar instrucciones de salto (`cmove`, `set...`). Estas instrucciones se ejecutarán si se cumple una determinada condición.
4. **Salto retardado**: se usaba en procesadores segmentados y consistía en, sabiendo que no podíamos resolver el salto hasta la etapa de ejecución, se captaba el salto dos ciclos de reloj antes y durante esos ciclos de reloj, se captaban instrucciones anteriores al salto que no tuvieran dependencias entre sí.

Esquemas de predicción de saltos

♡ **Predicción fija:** Siempre se considera que se va a saltar (*taken*) o siempre se considera que no se va a saltar (*not taken*). Es decir, siempre se toma la misma decisión.

♡ **Predicción verdadera:** la decisión de si se realiza o no el salto se decide mediante:

→ *Predicción dinámica:* se tiene en cuenta el historial del salto (lo que ha pasado en ejecuciones anteriores de ese salto). La predicción para cada instrucción de salto puede cambiar cada vez que se va a ejecutar ésta según la historia previa de saltos tomados/no tomados para dicha instrucción. El presupuesto básico de la predicción dinámica es que es más probable que el resultado de una instrucción de salto sea similar al que se tuvo en la última (o en las últimas n) ejecuciones. Presenta mejores prestaciones de predicción, aunque su implementación es más costosa. Hay dos tipos:

- **implícita:** en la que sólo se tiene en cuenta lo que ocurrió en la anterior ejecución del salto y sólo se almacena la dirección de salto de la ejecución anterior, si se equivoca, almacena la nueva dirección de salto.
- **explícita:** se usa lo que pasó en varias ejecuciones anteriores. Para cada instrucción de salto existen unos bits específicos que codifican la información de historia de dicha instrucción de salto.

→ *Predicción estática:* predicción fija que hace el compilador o la arquitectura. Hay varios tipos

- **Basada en el código de la operación:** para ciertos códigos de operación (ciertos saltos condicionales específicos) se predice que el salto se toma, y para otros, que el salto no se toma.
- **Basada en el desplazamiento del salto:** si se salta hacia atrás, lo normal es saltar (en un bucle `do-while` siempre se saltará excepto en la última iteración del bucle) y si el salto es hacia delante, lo normal es no saltar (en un bucle `for` sólo se saltará hacia delante en la última iteración del bucle).
- **Dirigida por el compilador:** El compilador es el que establece la predicción fijando, para cada instrucción, el valor de un bit específico que existe en la instrucción de salto (bit de predicción)

Es frecuente ver en los procesadores que la primera vez que se ejecuta una instrucción de salto condicional, si no se tiene historial de esa instrucción, se le aplica una predicción estática y las siguientes veces, aplicamos el historial.

Ejemplos de procedimientos explícitos de predicción dinámica de saltos

♡ **Predicción con 1 bit de historia:** la designación de estado, *taken* (1) o *not taken* (0), indica lo que se predice, y las flechas indican las transiciones de estado según lo que se produce al ejecutarse la instrucción.

♡ **Predicción con 2 bits de historia:** existen cuatro posibles estados. Dos para predecir *taken* y otros dos para predecir *not taken*. La primera vez que se ejecuta un salto se inicializa el estado con predicción estática, por ejemplo 11. Las flechas indican las transiciones de estado según lo que se produce al ejecutarse la instrucción (*taken* ó *not taken*).

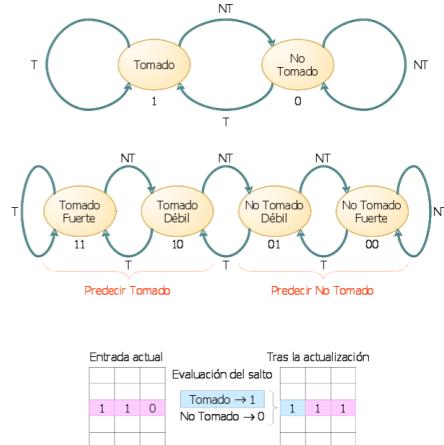


Figure 4.26: Procedimientos explícitos de predicción dinámica de saltos

- ♡ **Predicción con 3 bits de historia:** cada entrada guarda las ultimas ejecuciones del salto. Se predice según el bit mayoritario (por ejemplo, si hay mayoría de unos en una entrada se predice salto). La actualización se realiza en modo FIFO, los bits se desplazan, introduciéndose un 0 o un 1 según el resultado final de la instrucción de salto.

Extensión del procesamiento especulativo y recuperación de predicción incorrecta

Tras la predicción, el procesador continua ejecutando instrucciones especulativamente hasta que se resuelve la condición.

El intervalo de tiempo entre el comienzo de la ejecución especulativa y la resolución de la condición puede variar considerablemente y ser bastante largo.

Como máximo, una instrucción puede terminar su ejecución pero no puede escribir en los registros de la arquitectura, por tanto, las instrucciones especulativas pueden llegar como mucho a la etapa de ejecución del pipeline. Estras instrucciones permanecen en el ROB hasta que no se conoce si se ha fallado o no en la predicción. Si la predicción ha sido errónea, se descartan.

En los procesadores superescalares, que pueden emitir varias instrucciones por ciclo, pueden aparecer más instrucciones de salto condicional no resueltas durante la ejecución especulativa. Pueden haber varias instrucciones de salto condicional que se ejecuten una detrás de la otra. Los cores pueden predecir varias a la vez. Para disminuir la penalización en caso de error, lo que se hace es captar instrucciones de ambas ramas y dejarlas en el buffer de instrucciones, diviendo dicho buffer en dos. Como podemos procesar varios saltos, podemos almacenar incluso más ramas, teniendo en cuenta los saltos que pueda haber dentro de otros saltos (Figura ??)

Si el número de instrucciones que se ejecutan especulativamente es muy elevado y la predicción es incorrecta, la penalización es mayor.

Así, cuanto mejor es el esquema de predicción mayor puede ser el número de instrucciones ejecutadas especulativamente.

♥ **Nivel de especulación:** número de instrucciones de salto condicional sucesivas que pueden ejecutarse especulativamente (si se permiten varias, hay que guardar varios estados de ejecución).

♥ **Grado de especulación:** hasta qué etapa se ejecutan las instrucciones que siguen en un camino especulativo después de un salto.

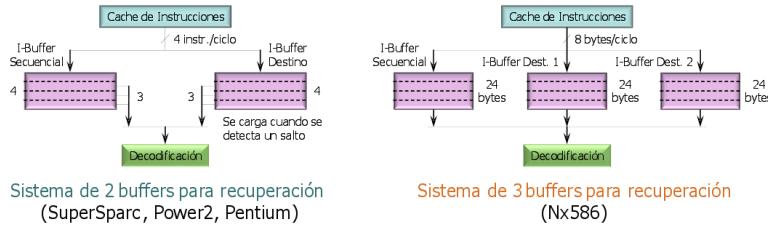


Figure 4.27: Recuperación de predicción incorrecta

La recuperación de una predicción incorrecta comprende:

- ♥ Descartar los resultados de la ejecución especulativa
- ♥ Continuar la ejecución de la secuencia de instrucciones alternativa (la correcta).

Recuperación desde un salto efectuado :

- ♥ El procesador debe guardar la dirección de la instrucción siguiente a la de salto para utilizarla si la predicción es incorrecta
- ♥ La recuperación es más rápida si no se descartan las instrucciones que se habían precaptado junto con la de salto.

Recuperación cuando no se ha saltado :

- ♥ Pre-calcular la dirección de salto y almacenarse para permitir la recuperación
- ♥ La recuperación es más rápida si se precaptan instrucciones de la secuencia que empieza a partir de la dirección a la que se salta.

Acceso a la secuencia de salto

Si se detecta una instrucción de salto, se calcula su dirección de destino para acceder a la posición de memoria correspondiente si se produce el salto.

Los saltos condicionales efectuados (*taken*) son más frecuentes que los no efectuados (*not taken*). Por ello, sería interesante reducir al máximo el tiempo de acceso a la secuencia de instrucciones a partir de la dirección de salto y reducir la penalización para las predicciones incorrectas de los saltos efectuados.

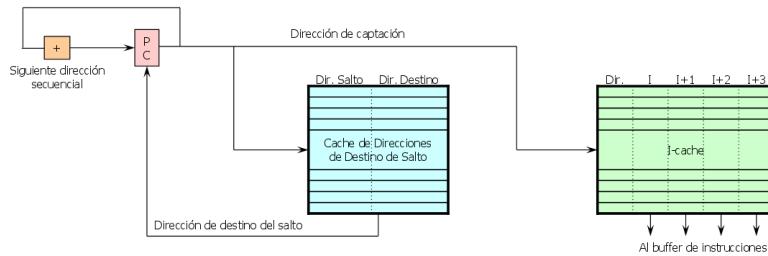


Figure 4.28: Esquema de la cache de saltos

La rapidez de acceso a la secuencia de instrucciones que empieza en la dirección a donde se salta es fundamental para mejorar las prestaciones del esquema de gestión de los saltos condicionales.

Esquema de cache de direcciones de destino de salto (BTAC) : como comentábamos antes, se añade una cache que contiene las direcciones de las instrucciones destino de los saltos, junto con las direcciones de las instrucciones de salto.

Se leen las direcciones al mismo tiempo que captan las instrucciones de salto.

Instrucciones de ejecución condicional (Guarded Execution)

Se pretende reducir el número de instrucciones de salto incluyendo en el repertorio máquina instrucciones con operaciones condicionales (“*conditional operate*” o “*guarded instructions*”). Estas instrucciones tienen dos partes:

- ♡ La **condición**, denominada *guardia*
- ♡ Y la **operación**

Por ejemplo, la instrucción `cmove xx` donde `xx` es una condición:

`gas cmove ra, rc, rd`

- `ne` es la condición, “*not equal*”.
- `ra` y `rb` son registros que guardan un entero cada uno
- `rc` es un registro para escritura
- El registro `ra` comprueba se en relación a la condición `ne` y si se verifica la condición, `rb` se copia en `rc`.

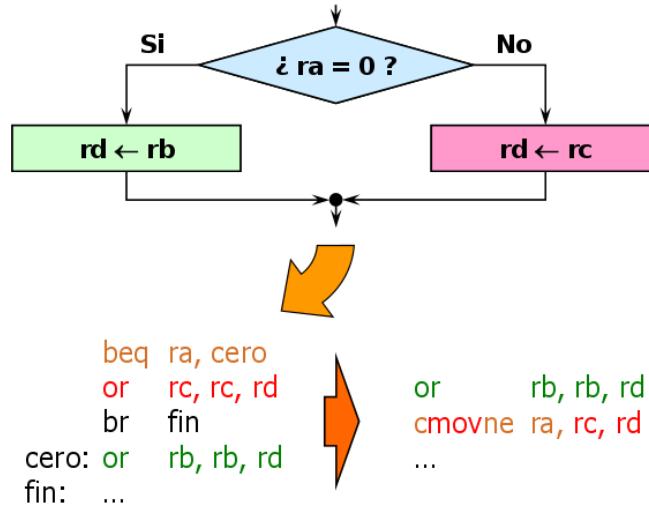


Figure 4.29: Comparación entre saltos y ejecución condicional

4.13 Procesamiento VLIW

4.13.1 Características generales y motivación (ILP hardware vs ILP software)

Características generales de los VLIW

El cauce de un VLIW es mucho más sencillo que el de un superescalar porque no tenemos estaciones de reserva, ni hardware de predicción de saltos... E incluso, podemos guardar los resultados en los registros de la arquitectura nada más terminar la ejecución, sin pasar por registros intermedios.

Las etapas de un VLIW (Figura ??) son:

- ♡ Captación (IF)
- ♡ Decodificación (ID)
- ♡ Ejecución (EX)

Los VLIW no tienen hardware para detectar paralelismo entre instrucciones. Las instrucciones que se emiten en paralelo a las unidades funcionales se captan juntas de memoria, es el compilador o el programador quien las pone juntas en memoria.

Las instrucciones se agrupan en palabras de instrucción largas, de ahí viene su nombre (*Very Long Instruction Word*), porque lo que se capta de memoria es una agrupación de instrucciones.

En esa agrupación de instrucciones, hay una instrucción de cada tipo. Los tipos dependen de la implementación, en el ejemplo de la Figura ?? una palabra se compondría de una instrucción de punto fijo, una de punto flotante, una de salto y otra de carga y almacenamiento en memoria.

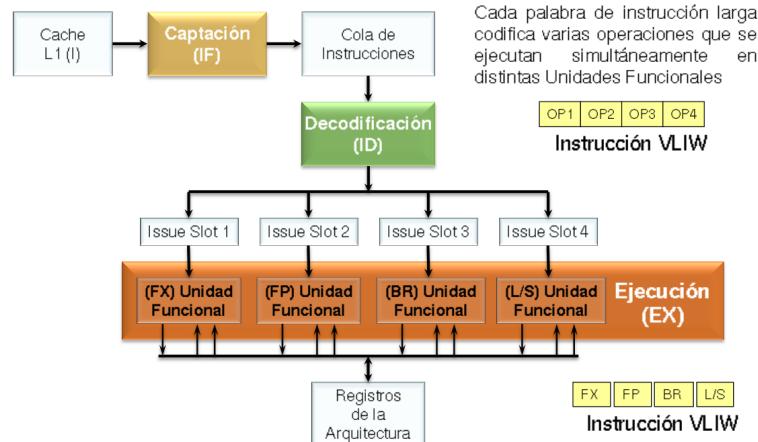


Figure 4.30: Etapas de un procesador VLIW

Las instrucciones de una palabra deben no deben tener ningún tipo de dependencia entre sí. Es el compilador o el programador quien evita las dependencias.

Como es difícil encontrar instrucciones independientes, es normal que haya vacíos en las palabras y esto hace que el código VLIW sea más largo.

Motivación: ILP hardware VS ILP software

ILP intensivo en hardware :

- ♥ Capaz de tener en cuenta los eventos que se producen dinámicamente durante la ejecución
- ♥ Mayor portabilidad de los códigos entre plataformas y mejor aprovechamiento de la memoria

ILP intensivo en hardware :

- ♥ Capaz de aprovechar la mayor visibilidad del código que tiene el compilador
- ♥ Mayor simplicidad en el hardware y menor consumo de energía.

Los VLIW representan la tendencia hacia el uso de un hardware lo más sencillo posible y la responsabilidad del compilador en la extracción del máximo ILP.

Estos procesadores se usan en productos empotrados. Se utilizan porque tienen un hardware más sencillo y por tanto, consume menos energía. Justo lo que queremos en productos empotrados, sobre todo si funcionan con batería. Además tienen un tamaño más pequeño y no necesitan ventiladores.

En el mercado de procesadores de propósito general, el Itanium de Intel (arquitectura x64) que se usa en servidores es una mezcla de VLIW y superescalar ya que en el itanium tenemos predicción de salto, detección de instrucciones para emitir en paralelo, etc.

Las instrucciones a emitir las encuentra el compilador, mientras que en superescalares las encuentra el hardware. Esto quiere decir que en los VLIW el paralelismo es extraído por el software mientras que en los superescalares lo extrae el hardware.

El compilador de un superescalar es más sencillo y los códigos generados son más portables.

Si tenemos un VLIW con cuatro unidades de ejecución, el compilador agrupa las instrucciones en grupos de cuatro. Por tanto, si tenemos otro VLIW con ocho, ya no nos serviría el código generado. Por esta razón, el código es menos portable.

4.13.2 Planificación estática

El papel del compilador

El VLIW tiene planificación estática porque se determinan las instrucciones en paralelo antes de la ejecución. En el caso de un superescalar, usamos planificación dinámica aunque podemos aprovechar también la estática.

Planificación estática : necesita asistencia del compilador, que puede realizar renombrados, reorganizaciones de código, etc para mejorar el uso de los recursos disponibles, el esquema de predicción de saltos, etc.

Planificación dinámica : requiere menos asistencia del compilador pero más coste hardware. Facilita la portabilidad del código entre la misma familia de procesadores.

El compilador construye paquetes de instrucciones, llamados *ventanas de emisión*, sin dependencias, de forma que el procesador no necesita comprobarlas explícitamente.

Planificación Estática

En el siguiente ejemplo existen dependencias RAW cada dos instrucciones consecutivas y además existe una instrucción de salto que controla el final del bucle: parece que no se puede aprovechar mucho ILP.

```
[linenos,    frame=single]c    for
(i=1000; i>0; i=1) x[i] = x[i] + [frame=single]gas loop: ld addd sd subui jne
s;
```

Tenemos un bucle con 1000 iteraciones que suma a cada uno de los componentes del vector *x* de doubles un valor escalar *s*. En el ensamblador generado tendríamos instrucciones de carga y almacenamiento. Estamos suponiendo que la arquitectura redirecciona a nivel de byte y como un double tendría 8B, por eso decrementamos el índice en 8.

Suponiendo que hay suficientes unidades funcionales para la suma, y que cuando hay dependencias RAW (las WAR y WAW podemos evitarlas con el renombrado) los retardos introducidos son los siguientes:

Instr que produce el resultado	Instr que usa el resultado	Latencia
Operación FP	Operación FP	3
Operación ALU	Store	2
Load	Operación FP	1
Load	Store	0
Operación Fx	Salto	1
Operación Salto	-	1

Tentemos un cauce segmentado sin hardware de planificación (el compilador tiene que asegurarse de que no empieza una instrucción hasta que no acaba la anterior introduciendo instrucciones de no operación entre medias). Así, sin desenrollar el bucle obtendríamos el siguiente código:

```
[linenos]gas ld _____ addd _____ sd subui _____ jne _____
```

Entre `1d` y `addd` hemos añadido una instrucción de no operación porque `1d` tiene una latencia de un ciclo de reloj. Entre un punto flotante (`addd`) y un almacenamiento (`sd`) tenemos que esperar dos ciclos, por tanto, hemos tenido que añadir los ciclos de penalización entre instrucciones con el fin de que las instrucciones esperen los datos que necesitan.

En total, nos supone la ejecución de una iteración 10 ciclos de reloj. Como tenemos 1000 iteraciones, en total necesitaríamos 10000 ciclos de reloj. Pero, si nos damos cuenta, el cálculo del índice siguiente (instrucción `subui`) no tiene dependencias RAW por lo que podríamos ponerla después de `1d`. Esto nos obliga a que en `sd` tenemos que acceder a $i + 8$:

```
[linenos]gas ld subui addd _____ jne sd
```

También aprovechamos el ciclo de reloj extra que tenemos que esperar tras la instrucción de salto para hacer el almacenamiento del valor de $x[i]$ calculado. Con estas modificaciones, reducimos el número de ciclos a 6000.

Ahora supongamos que tenemos un VLIW con tres unidades de ejecución: una para punto fijo, otra para punto flotante y otra para carga y almacenamiento.

En la primera palabra podremos poner `subui` y `1d`, las dos leerían el mismo valor de `r1`. A continuación no podríamos poner ninguna operación porque tenemos que esperar 1 ciclo de reloj para que termine `1d` y pueda ejecutarse `addd`. En la siguiente palabra, sólo tendríamos una instrucción, `addd` por lo que en el resto de unidades de ejecución tendríamos instrucciones de no operación.

La instrucción de salto está en la penúltima palabra porque tiene un retardo de un ciclo de reloj. En ese tiempo tenemos garantizado que no cambiará el contador de programa. No podemos ponerlo en la palabra anterior porque entonces no se ejecutaría la última instrucción (`sd`):

```
[linenos]gas inst FX/ BR inst FP inst LD/ST subui _____ addd _____
_____ jne _____ sd
```

Con esta implementación desaprovechamos espacio de almacenamiento con tanta instrucción de no operación. Para solucionar esto, podemos usar o bien el desenrollado de bucle o bien el software pipelining.

Planificación estática y global

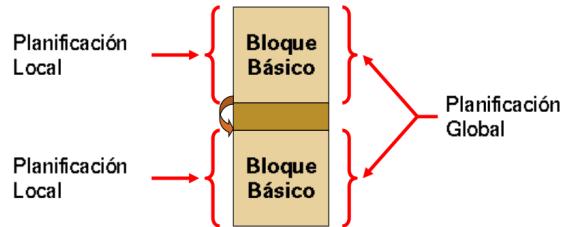


Figure 4.31: Optimizaciones locales y globales

Para hacer optimizaciones, los compiladores dividen el código en bloques básicos que obtienen entre instrucciones de salto y etiquetas de salto de manera que, por ejemplo, el código de un bucle es un código básico. Los compiladores hacen optimizaciones a nivel de bloque básico pero también hacen optimizaciones entre bloques básicos, llamadas optimizaciones globales.

Planificación estática local

Desenrollado de bucles :

- ♡ Al desenrollar un bucle se crean bloques básico más largos, lo que facilita la planificación local de sus sentencias.
- ♡ Además de disponer de más sentencias, éstas suelen ser independientes, ya que operan sobre diferentes datos.

Segmentación software(software pipelining) :

- ♡ Se reorganizan los bucles de forma que cada iteración del código transformado contiene instrucciones tomadas de distintas iteraciones del bloque original
- ♡ De esta forma se separan las instrucciones dependientes en el bucle original entre diferentes iteraciones del bucle nuevo.

En resumen, con ambas técnicas se pasa de un bucle original a un nuevo bucle que combina varias iteraciones del bucle original. Pero en el caso del desenrollado de bucle el bucle resultante contiene todas las instrucciones de varias iteraciones del bucle original mientras que con el software pipelining tenemos una instrucción de cada iteración.

Planificación estática con desenrollado de bucles

Al aplicar un desenrollado de bucle de 5 iteraciones, obtenemos el siguiente código:

```
[linenos]gas loop: ld ld ld ld ld addd addd addd addd sd sd sd sd sd subui jne
```

Al aplicar un desenrollado de 5 iteraciones, conseguimos 5 cargas, 5 sumas, 5 almacenamientos e instrucciones de control de bucles. Decrementamos i en 40 porque procesamos las 5 componentes del vector. Todas las operaciones de iteraciones distintas son independientes por lo que se podrían ejecutar en paralelo. Tenemos más paralelismo que aprovechar en el VLIW.

En el código resultante para el VLIW anterior tendremos que tener en cuenta los retardos anteriores:

[linenos]gas ————— ld ————— ld —————
 addd ————— addd ————— addd ————— addd subui —————
 ————— sd jne ————— sd

Se tardarían $10 \times \frac{1000}{5} = 2000$ ciclos.

Planificación estática con segmentación software

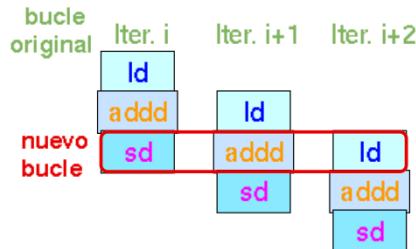


Figure 4.32: Desenrollado de bucle VS Segmentación software

[linenos, frame=single]gas iteracion i ld addd sd
 iteracion i+1 ld addd sd [frame=single]gas loop: sd addd ld subui jne
 iteracion i+2 ld addd sd

En el bucle original, la iteración i tendrá una instrucción de carga, otra de suma y otra de almacenamiento. En el nuevo bucle, tendremos la carga de la iteración i , la suma de la iteración $i + 1$ y la carga de la $i + 2$. Estas tres instrucciones son independientes. Para que tenga mayor ventaja, es bueno combinar esta técnica con el desenrollado de bucle.

En la arquitectura VLIW que tenemos de ejemplo no podremos poner en la misma palabra la carga y el almacenamiento porque ambas instrucciones se procesan en la misma unidad de ejecución, por lo que tendremos que procesar la carga junto a la resta:

[linenos]gas ————— addd subui ————— ————— jne —————
 ————— ————— —————

En total tardamos $5 \times 1000 = 5000$ ciclos. Si se desenrolla el bucle ya segmentado, se pueden mejorar mucho más las prestaciones. Esta rebaja adicional podemos conseguirla subiendo `subui` y sumando 8 a `sd`. Así podemos subir también la instrucción de salto y quedarnos en 4000 ciclos. Si lo combinamos con el desenrollado podemos llegar a 2000.

Planificación estática global

La planificación global mueve código a través de los saltos condicionales (que no correspondan al control del bucle).

Se parte de una estimación de las frecuencias de ejecución de las posibles alternativas tras una instrucción de salto condicional.

Las instrucciones con predicado y la especulación son un apoyo para facilitar la planificación global.

Instrucciones de ejecución condicional

Para reducir las instrucciones de salto que van a frenar los ciclos que podemos conseguir en un VLIW, podemos usar instrucciones de ejecución condicional.

Instrucciones con Predicado

Los predicados están tanto en arquitecturas VLIW como en superescalares. Pueden tomar dos valores: 0 o 1.

Para evitar instrucciones de salto cuando tenemos un **if** dentro de otro, podemos usar predicados de forma que, cuando una instrucción va acompañada de un predicado, sólo se ejecuta si el predicado es verdadero.

Hay instrucciones específicas para modificar su contenido, como por ejemplo **cmp**. Podemos usar dos predicados o uno:

♥ Si usamos dos predicados al hacer una comparación, el primer predicado se coge el valor de dicha comparación y el segundo, el complementario. Así por ejemplo si $a = 3$:

$c|P1, P2 = cmp (a == 0)$

P1 tendría valor 0 y P2, valor 1.

♥ Si solo tenemos un predicado, éste coge el valor de la comparación.

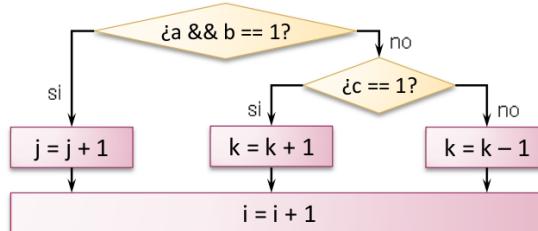


Figure 4.33: Flujo de ejecución del ejemplo

Por ejemplo, en el siguiente código (Figura ??):

[linenos]c if (a b) j++; else if (c) k++ else k- i++

Podríamos usar predicados para no obtener ni un sólo salto en nuestro código:

[linenos]c P1, P2 = cmp (a == 0) P3 = cmp (a != a) // Inicializa P3 a 0 P4 = cmp (a != a) // Inicializa P4 a 0 P5 = cmp (a != a) // Inicializa P5 a 0 <P2> P1, P3 = cmp(b == 0) // Si a == 1 <P3> add j, j, 1 // Si a == 1 y b == 1 <P1> P4, P5 = cmp(c != 0) // Si a == 0 o b == 0 <P4> add k, k, 1 // Si c == 1 <P5> sub k, k, 1 // Si c == 0 add i, i, 1

Las instrucciones con predicado reducen el número de saltos condicionales. Esto es bastante importante, sobre todo si no hay una opción más frecuente que otra en el salto.

El uso general de predicados es muy útil en planificación global ya que puede eliminar todos los saltos condicionales que no sean de control de bucle (facilita la *construcción de superbloques*).

Las instrucciones de movimiento condicional de datos son las más utilizadas, aunque pueden ser ineficientes si se dispone sólo de ellas para transformar trozos de código largos que dependen de saltos.