TEMA 1: PROCESADORES SEGMENTADOS

1.3 Diferencias entre procesadores RISC y procesadores CISC.

PROCESADOR CISC: Complex Instruction Set Computer.

Los procesadores fueron dotados de conjuntos de instrucciones muy potentes que realizaban gran cantidad de operaciones internas.

Estas instrucciones se decodifican internamente en la unidad de control y se ejecutan mediante unas microinstrucciones almacenadas en una ROM interna, llamada memoria de control.

Para ello se requieren varios ciclos de reloj. Debido a la gran cantidad de operaciones internas que realizaban las instrucciones se empezaron a llamar CISC.

Simultáneamente se introducen los modos de direccionamiento para disminuir la cantidad de instrucciones a almacenar en memoria.

Está motivado porque el tiempo de acceso a memoria era mucho menor que el tiempo de procesamiento.

La arquitectura tipo CISC dificulta el paralelismo a nivel de instrucciones, para realizar la ejecución de una instrucción se debe de cargar desde memoria y traducirla a microoperaciones.

PROCESADOR RISC: Reduced Instruction Set Computer.

Para aumentar la velocidad de procesamiento se descubrió que con una determinada arquitectura de base la ejecución de programas compilados con instrucciones simples era más eficiente.

El avance en la fabricación de procesadores centró el objetivo de diseño en las arquitecturas RISC en la disminución de accesos a memoria mediante un aumento de la cantidad de registros internos en el procesador. El hardware de control es más sencillo debido al conjunto de instrucciones simplificado.

La decodificación de instrucciones puede ser directamente implementada por hardware dentro de la CPU, disminuyendo la lógica de control y obteniendo una mayor velocidad de ejecución.

La alternativa RISC permite la ejecución segmentada de instrucciones, pipeline, y el diseño de procesadores cableados.

Ventajas de un diseño RICS frente a un CISC.

Tienen una mayor velocidad de ejecución.

Reduce el tamaño de la CPU, por lo que se dispone de más espacio para recursos, mayor cantidad de ***registros*** o memoria caché. En arquitectura de ordenadores, un ***registro*** es una memoria de alta velocidad y poca capacidad, integrada en el microprocesador, que permite guardar transitoriamente y acceder a valores muy usados, generalmente en operaciones matemáticas.

Al ser la lógica de control más simple, se facilita el diseño.

Permite máquinas más compactas y con menor consumo al reducirse el tamaño de la CPU.

Posibilita la segmentación y el paralelismo en la ejecución de instrucciones.

Reduce los accesos a memoria debido a que los operandos se cargan en registros.

El rendimiento de la implementación pipeline en un procesador RISC depende directamente de los riesgos que se generan durante la ejecución de dos o más instrucciones que entran en conflicto.

Para minimizar los riesgos se plantea una relación muy estrecha entre los compiladores y la arquitectura. Las operaciones complejas que aparecen en el código fuente se descomponen en multitud de operaciones sencillas RISC.

El objetivo es mantener el hardware tan simple como podamos a base de hacer más complejo el compilador.

La ganancia de velocidad se debe a que predominan las instrucciones más frecuentemente utilizadas, mientras que las menos frecuentes se descomponen en operaciones simples.

La cantidad de instrucciones en un procesador RISC es mayor que en un CISC.

Un factor importante de los procesadores RISC es la compatibilidad con el software preexistente.

Tienen filosofías de diseño opuestas.

A igual tecnología y frecuencia de reloj, un procesador RISC tiene mayor capacidad de procesamiento, con una estructura de hardware más simple, teniendo como beneficio disminución de potencia y costes.

El aumento de rendimiento de un RISC se sustenta en el diseño de un compilador eficiente.

1.4 Clasificación y características generales de las arquitecturas paralelas.

El paralelismo se implementa siguiendo dos líneas:

Replicación de elementos: incluye unidades funcionales, procesadores, módulos de memoria entre los que se distribuye el trabajo.

A nivel de sistema están los multiprocesadores y los canales/procesadores de E/S.

A nivel de procesador se tiene el uso de varias unidades funcionales en los procesadores superescalares, los procesadores VLIW y los procesadores vectoriales.

Segmentación (*pipelining*): es una técnica en la que un elemento se divide en una serie de etapas que funcionan de forma independiente y por las que van pasando los operandos y las instrucciones que procesa cada elemento. De esta forma dicho elemento puede realizar simultáneamente operaciones distintas en las diferentes etapas en que se encuentra dividido.

Taxonomía de Flynn.

Computadores SISD (Single Intruction Single Data): Un único flujo de instrucciones procesa operandos y genera resultados definiendo un único flujo de datos. Computadores monoprocesador.

Computadores SIMD (Single Intruction Multiple Data): Un único flujo de instrucciones procesa operandos y genera resultados definiendo varios flujos de datos. Computadores matriciales y vectoriales.

Computadores MIMD (Multiple Intruction Multiple Data): El computador ejecuta varias secuencias o flujos distintos de instrucciones y cada uno de ellos procesa operandos y genera resultados de forma que existen también varios flujos, uno por cada flujo de instrucciones. Computadores multiprocesadores y multicomputadores.

Computadores MISD (Multiple Intruction Single Data): Se ejecutan varios flujos de instrucciones aunque todos actúan sobre el mismo flujo de datos.

Tipos de paralelismo según la Taxonomía de Flynn.

Paralelismo de datos.

Se utiliza cuando una misma función, instrucción, etc… se ejecuta repetidas veces en paralelo sobre datos diferentes.

Se utiliza en máquinas de la clase SIMD y MIMD.

Paralelismo funcional.

Se aprovecha cuando las funciones, bloques, instrucciones, etc… que intervienen se ejecutan en paralelo.

Existen distintos niveles de paralelismo funcional.

- **Nivel de instrucciones u operaciones**. (ILP), las instrucciones de un programa se ejecutan en paralelo, es de bajo nivel, se explota por el hardware de forma transparente o por el compilador. Es el nivel de granularidad más fina en la arquitectura de computadores la granularidad es la cantidad de trabajo asociado a cada tipo de tarea candidata a la paralelización.

- **Nivel de bucle**. Se ejecutan en paralelo distintas iteraciones de un bucle o secuencias de instrucciones de un programa, la granularidad es fina-media.

- **Nivel de funciones**. Los distintos procedimientos que constituyen un programa se ejecutan simultáneamente, la granularidad es media.

- **Nivel de programas**. Se ejecutan en paralelo programas diferentes que pueden ser o no de una misma aplicación, la granularidad es gruesa.

1.5 Medidas para evaluar el rendimiento de un computador.

Las medidas más utilizadas para el rendimiento de un computador son:

- **Tiempo de respuesta**: tiempo que tarda el computador en procesar una entrada.
- **Productividad**: número de instrucciones procesadas por unidad de tiempo.
- **Funcionalidad**: tipos de entradas diferentes que es capaz de procesar.
- **Expansibilidad**: posibilidad de ampliar la capacidad de procesamiento añadiendo bloques a la arquitectura existente.
- **Escalabilidad**: posibilidad de ampliar el sistema sin que esto suponga una devaluación de las prestaciones.
- **Eficiencia**: relación entre el rendimiento obtenido y el coste que ha supuesto conseguirlo. (eficiencia = rendimiento/coste).

$$T_{CPU} = NI \cdot CPI \cdot T_{ciclo} = NI \cdot \left(\frac{CPI}{f} \right)$$

TCPU : tiempo de CPU de un programa o tarea.
NI : número de instrucciones máquina del programa que se ejecutan.
CPI : número medio de ciclos por instrucción.
Tciclo : periodo del reloj del procesador.

$$CPI = \frac{Ciclos\ de\ reloj\ del\ \ programa}{NI} = \frac{\sum_{i=1}^{n} NI_i \cdot CPI_i}{NI}$$

CPIi :número medio de ciclos de las instrucciones de tipo i.
NIi : número de instrucciones de ese tipo.

$$T_{CPU} = NI \cdot \left(\frac{CPE}{IPE} \right) \cdot T_{ciclo}$$

CPE : número medio de ciclos entre inicios de ejecución de instrucciones.
IPE : número medio de instrucciones que se emiten.

$$T_{CPU} = \left(\frac{N_{operaciones}}{Op_{instrución}} \right) \cdot CPI \cdot T_{ciclo}$$

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

MIPS : Millones de instrucciones por segundo.

$$MFLOPS = \frac{Operaciones\ en\ coma\ \ flotante}{T_{CPU}}$$

$$S_p = \frac{rendimiento(p)}{rendimiento\_original} = \frac{T_{CPU\_original}}{T_{CPU\_mejorada}}$$

$$S_p \leq \frac{p}{1 + f(p-1)}$$ ley de Amdahl

$$T_p \geq f \cdot T_1 + (1-f)\frac{T_1}{p}$$

1.6 Características de los procesadores segmentados.

La segmentación (***pipelining***) es una técnica empleada en el diseño de procesadores que trata de explotar el paralelismo intrínseco que existe entre las instrucciones de un flujo secuencial.

Mediante la segmentación se puede solapar la ejecución de múltiples instrucciones. Cada etapa de la segmentación completa una parte de la tarea total. La salida de un segmento es la entrada del siguiente.

La segmentación puede procesar las subtareas de forma simultánea, aunque sea sobre diferentes tareas, ***lo más importante*** es la posibilidad de comenzar una nueva tarea sin que la anterior haya terminado.

La ***medida de eficacia de un procesador*** es el tiempo máximo que pasa entre la finalización de dos tareas consecutivas.

La velocidad de emisión de tareas es el ritmo al que salen las tareas del procesador.

La profundidad de segmentación es el número de n etapas en las que puede dividirse el procesamiento de una instrucción.

Para que el tiempo de latencia del procesador segmentado sea el mínimo posible es necesario que el procesador esté equilibrado.

El procesador está equilibrado si todas las subtareas en que se haya dividido la tarea total tarden en procesarse el mismo tiempo. Las tareas no pueden avanzar a la etapa siguiente hasta que no se haya terminado la subtarea más lenta, en el caso que el procesador no esté equilibrado las etapas más rápidas estarán un tiempo sin hacer trabajo, disminuyendo el rendimiento total del procesador.

La relación de precedencia de un conjunto de subtareas $T_1,\ldots,T_n$ que componen cierta área T, especifica para cada subtarea T_j que no puede comenzarse hasta que hayan terminado ciertas subtareas T_i.

Las relaciones de precedencia para todas las subtareas de T forman su grafo de precedencia.

1.7 Arquitectura segmentada genérica.

La arquitectura segmentada genérica ASG, es de tipo de registros de propósito general donde los operandos se referencian explícitamente.

La ventaja de estas máquinas surgen del uso efectivo de los registros por parte del compilador al calcular expresiones aritmético lógicas y al almacenar datos.

Los registros permiten una gestión más flexible de los datos, además se reduce el tráfico de memoria, se acelera la ejecución del programa y se mejora la densidad de código.

Dos características importantes del repertorio de instrucciones que clasifican las arquitecturas de propósito general:

- El número de operandos que puede tener las instrucciones aritmético-lógicas.
- El número de operandos que se pueden direccionar en memoria en las instrucciones aritmético-lógicas.

Las instrucciones aritmético-lógicas de la ASG utilizan en total tres operandos y ninguno de ellos se referencia a memoria.

Las máquinas en la que los operandos no se referencian en memoria se les denominan máquinas registro-registro o máquinas de carga/almacenamiento.

En la ASG el modo de direccionamiento es con desplazamiento y la Dirección Efectiva se calcula sumando al contenido de un registro el Operando declarado en la instrucción, que es un desplazamiento respecto al contenido del registro.

1.7.1 Repertorio de instrucciones de la ASG.

La ASG tiene un total de 32 registros de 32 bits identificados desde R0 a R31.
Cada registro puede contener un valor entero a excepción de R0 que siempre contiene el valor 0.
Los registros de coma flotante de 64 bits se identifican como F0,… F30
La longitud de las instrucciones en procesadores segmentados de la ASG es de 32 bits.

Los cuatro tipos básicos de operaciones son:

- Aritméticas y lógicas
- Transferencia de datos.
- Bifurcaciones o saltos incondicionales.
- Saltos condicionales.

Las instrucciones de la ALU son operaciones aritméticas sencillas (suma, resta, multiplicación, división y comparación)y lógicas (AND, OR, XOR).

En las instrucciones de comparación, si la condición es cierta se almacena un 1 en el bit menos significativo del registro de destino y los restantes se fijan a 0, en caso contrario se coloca un 0 en todos los bits.

Las operaciones de transferencia de datos entre los registros y la memoria se realizan exclusivamente a través de instrucciones de carga (LD) y almacenamiento (SD).

En una instrucción de carga se almacena el contenido de una dirección de memoria en un registro y al contrario en una instrucción de almacenamiento.

LD Rd, desplazamiento (Rf).

Rd: registro destino donde se almacena el dato leído de memoria.
desplazamiento(Rf): dirección de memoria a la que acceder para recuperar el dato, Rf es el registro base.

SD desplazamiento (Rd), Rf.

Rf: registro cuyo contenido se va a escribir en memoria.
desplazamiento(Rd): dirección de memoria donde se va a escribir, Rd es el registro base.

El valor del desplazamiento y el valor del registro base serán siempre valores enteros positivos o negativos.

Si utilizamos registros en coma flotante las expresiones son:

LD Fd, desplazamiento (Rf).
SD desplazamiento (Rd), Ff.

Una carga o un almacenamiento sencillo se lee o escribe una palabra de 4 bytes.
Una carga o un almacenamiento en coma flotante se lee o escribe una palabra de 8 bytes.

Instrucción Ejemplo	Nombre	Significado
LD R1, 4(R2)	Carga	R1 ← M[R2 + 4]
SD 0(R3), R5	Almacenamiento	M[R3 + 0] ← R5
LD F6, 5(R3)	Carga flotante	F6 ← M[R3 + 5]
SD 6(R1), F4	Almacenamiento flotante	M[R1 + 6] ← F4

La modificación del flujo de control se realiza mediante las instrucciones de bifurcación y salto.

La bifurcación ocurre cuando el cambio en el flujo de control sea incondicional y de salto cuando sea condicional, es decir cuando establece una condición (R5>0).

En los saltos la condición que determina el resultado del salto se especifica en el código de operación de la instrucción que examina el registro fuente para compararlo con cero, distinto de cero, mayor de cero o menor de cero.

La forma más común de especificar el destino del salto es suministrar un desplazamiento que se sume al contador del programa, este tipo de saltos son los saltos relativos.

Un salto es efectivo cuando la condición probada por la instrucción de salto es verdadera y la siguiente instrucción que se vaya a ejecutar es el destino del salto.

Las bifurcaciones son siempre efectivas.

Instrucción Ejemplo	Nombre	Significado
JUMP etiqueta	Bifurcación	PC ← etiqueta
JUMP R4	Bifurcación a registro	PC ← R4
BEQZ R3, etiqueta	Salto igual a cero	If (R3 = 0) PC ← PC + etiqueta
BNEZ R2, etiqueta	Salto distinto de cero	If (R2 ≠ 0) PC ← PC + etiqueta
BGT R5, etiqueta	Salto mayor que cero	If (R5 > 0) PC ← PC + etiqueta
BLT R7, etiqueta	Salto menor que cero	If (R7 < 0) PC ← PC + etiqueta

1.7.2 Implementación de la segmentación de instrucciones en la ASG.

En un procesador segmentado el cálculo a segmentar es el trabajo que es necesario realizar en cada ciclo de instrucción, que es el número de ciclos de reloj que consume su procesamiento.

En la ASG ***un ciclo de una instrucción se descompone en cinco etapas*** básicas:

1. IF (Instruction Fetch): lectura de la instrucción de la caché de instrucciones. La caché de instrucciones puede admitir la lectura de una instrucción en cada ciclo de máquina y un fallo en la caché detiene la segmentación
2. ID (Intruction Decoding): decodificación de la instrucción y lectura de sus operandos del fichero de registros.
3. EX (Execution): ejecución de las operaciones si se trata de una instrucción aritmético-lógica y del cálculo de la condición y de la dirección de salto si se trata de una bifurcación o salto condicional.
4. MEM (Memory Access): Acceso a la caché de datos para lecturas (cargas) o escrituras (almacenamientos).
5. WB (Write-Back result): Escritura del resultado en el fichero de registros. Las instrucciones de almacenamiento y salto no realizan ninguna acción en esta etapa ya que no necesitan escribir en el fichero de registros y se encuentran liberadas.

En cada ciclo de máquina el fichero de registros debe admitir dos lecturas en la etapa ID y una escritura en la etapa WB.

La ventaja de la segmentación es la posibilidad de empezar a ejecutar una nueva instrucción en cada ciclo de reloj.

El tiempo total de la instrucción segmentada es ligeramente superior al de su equivalente no segmentada debido al tiempo que se consume en el control de la segmentación.

Este tiempo viene determinado por varios factores:

- Los cerrojos o buffers de contención con el objeto de aislar la información entre etapas.
- La duración de todas las etapas viene determinada por la duración de la etapa más lenta.
- Los riesgos que se producen en la segmentación y producen detenciones.

1.8 Riesgos en la segmentación.

Los riesgos de la segmentación son consecuencia tanto de la organización como de las dependencias entre las instrucciones.

Un ***riesgo*** es la situación que impide a una instrucción acceder a la ejecución de sus etapas al depender de otra anterior.

Los riesgos provocan una parada en el flujo de las instrucciones dentro de las etapas de la segmentación hasta que la dependencia se resuelva.

1.8.1 Riesgos estructurales.

Surgen por conflictos en los recursos, debido a que el hardware que necesita una instrucción está siendo utilizado por otra.

Otras situaciones en las que pueden aparecer riesgos estructurales son:

- No todas las etapas de la segmentación tienen la misma duración.
- Hay instrucciones más complejas que otras.

Para evitar las detenciones en las etapas de segmentación podemos plantear dos soluciones:

- Duplicar la unidad funcional aritmética para poder solapar dos etapas de dos instrucciones en el mismo ciclo de reloj.
- Separar las instrucciones si se puede, se conoce como ***planificación de código***.

1.8.2 Riesgos por dependencia de datos.

Se produce cuando una instrucción necesita los resultados de otra anterior por no haberse terminado de ejecutar.

Los riesgos por dependencia de datos se clasifican en función del orden de los accesos de escritura y lectura de las instrucciones en tres tipos, consideraremos las instrucciones *i* ocurre primero y *j* después.

- Riesgo tipo **RAW** (Read After Write): también conocido como ***dependencia verdadera***, se produce cuando la instrucción j intenta leer un dato antes de que la instrucción *i* lo escriba.
- Riesgo tipo **WAR** (Write After Read): también conocido como ***antidependencia o dependencia falsa***, se produce cuando la instrucción *j* intenta escribir en su destino antes de que sea leído por la instrucción *i*.
- Riesgo tipo **WAW** (Write After Write): también conocido como ***dependencia de salida o dependencia falsa***, se produce cuando la instrucción *j* intenta escribir un operando antes de que sea escrito por la instrucción *i*.

Hay que tener en cuenta que el caso **RAR** (Read After Read) no es un riesgo, ya que cuando dos instrucciones necesitan leer desde el mismo registro, lo hacen sin problemas en la etapa de decodificación de la instrucción.

En la ASG sólo se puede presentar el riesgo RAW.

El riesgo WAR no se puede presentar ya que todas las lecturas de registro se realizan en la etapa ID y todas las escrituras de registro tienen lugar en la etapa WB, siendo la ID anterior a la WB.

El riesgo WAW se presenta en segmentaciones que escriben en más de una etapa. La ASG evita este riesgo ya que solo escribe un registro en WB.

Riesgos por dependencia de datos en registros en el caso de instrucciones aritmético-lógicas.
Riesgos por dependencia de datos en memoria con instrucciones de carga y almacenamiento.

Las alternativas más importantes ***para evitar los problemas de los riesgos RAW*** son:

1.8.2.1 La reorganización de código.

Consiste en disponer las instrucciones en el programa de forma que entre las instrucciones con dependencias tipo RAW existan instrucciones que permitan retrasar la segunda instrucción con respecto a la primera, de esta forma la primera tendrá tiempo a escribir su resultado antes que la segunda la lea.

Debemos de mantener la semántica original del programa, el orden original de las lecturas y escrituras de los registros y la memoria.

En el caso de que el compilador no pueda reorganizar el código se deben insertar instrucciones NOP, entre las instrucciones que tengan dependencia de datos. Una ventaja es que no hace falta un hardware adicional, pero se necesita un compilador más complejo y pérdida de tiempo.

1.8.2.2 El interbloqueo entre etapas.

Se introducen elementos hardware en el cauce para detectar la existencia de dependencias, en el caso que detecte una la instrucción se detiene el número de ciclos necesarios. El programa finaliza correctamente pero seguimos perdiendo ciclos lo que nos lleva a una disminución del rendimiento.

1.8.2.3 El adelantamiento (Caminos de bypass o forwarding).

Con esta técnica aprovechamos los elementos de la técnica de interbloqueo y la utilizamos para habilitar una serie de buses para permitir que los resultados de una etapa pasen como entradas a la etapa donde son necesarios en caso de dependencias RAW, a la vez que prosigue su camino para almacenarse en el fichero de registros.

La función que tiene la lógica de bypass es comprobar si hay coincidencia entre el identificador del registro de destino de la instrucción que acaba su etapa EX y los identificadores de los registros fuente de los operandos de la instrucción siguiente que va a iniciar su etapa EX.

1.8.3 Riesgos de control.

Se producen a partir de las instrucciones de control de flujo, saltos y bifurcaciones, ya que no podemos leer la instrucción siguiente hasta que no se conozca su dirección.

Cuando se ejecuta un salto condicional el valor del contador del programa puede incrementarse automáticamente o cambiar su valor en función de que el salto sea efectivo o no.

En la ASG si la instrucción *i* es un salto efectivo entonces ***el PC no se actualiza hasta el final de la etapa MEM***, hasta haya verificado la condición y calculado la nueva dirección del PC en la etapa EX.

El PC se incrementa en cada ciclo automáticamente para apuntar a la siguiente instrucción a buscar.

Se incrementa 4 bytes si la memoria se direcciona por bytes.

Se incrementa 1 byte si la memoria se direcciona por palabras de 4 bytes.

En las instrucciones de salto condicional el PC debe cargarse con la dirección de destino del salto, si el salto es efectivo controlando el valor de carga un multiplexor:

- El valor actual incrementado si el salto no es efectivo.
- El correspondiente a la dirección de destino si el salto es efectivo, calculado en la etapa EX.

El esquema más fácil de implementar en ASG es detener la segmentación hasta que se conozca el resultado del salto, introduciendo instrucciones NOP después de cada instrucción de salto condicional, el problema es que se reduce el rendimiento de la segmentación.

La alternativa es dejar que las instrucciones sigan ejecutándose. El compilador introduce en esos huecos instrucciones que se tienen que ejecutar antes de la instrucción de destino de salto siendo el efecto independiente si el salto es efectivo o no.

De esta forma el cauce puede terminar una instrucción por ciclo, mejorando el rendimiento, a esto lo llamamos **s*alto retardado***.

Un esquema mejor y un poco más complejo es predecir el salto como no efectivo, permitiendo que el hardware continúe procesando la siguiente instrucción en la secuencia del programa como si el salto fuese efectivo, se conoce como ***ejecución especulativa***.

Hay que tener cuidado de no cambiar el estado de la máquina hasta que no se conozca definitivamente el resultado del salto.

El estado de la máquina viene definido por el contador de programa, el fichero de registros y el espacio de memoria asignado al programa.

Si el salto es efectivo se debe detener la segmentación y recomenzar la búsqueda de la instrucción destino del salto. Se eliminan las instrucciones que se estaban ejecutando detrás de la instrucción de salto, perdiendo ciclos de reloj.

Si el salto no es efectivo no se penaliza con pérdidas de ciclo de reloj.

Se mejora considerablemente el rendimiento del procesador frente a los saltos si se adelanta el cálculo de la dirección y de la condición de la instrucción de salto a las primeras etapas del cauce

1.9 El algoritmo de Tomasulo como técnica de planificación dinámica en segmentación.

Hasta el momento utilizamos la planificación estática como única técnica en un procesador segmentado. Una de las principales limitaciones de la segmentación estática es que emiten las instrucciones en orden, si el procesador se detiene con una instrucción las posteriores no pueden proceder aunque no mantengan ninguna dependencia con las instrucciones que van por delante en el cauce. Una dependencia entre dos instrucciones puede dar lugar a un riesgo y a una detención.

En la planificación dinámica el hardware reorganiza la ejecución de la instrucción para reducir las detenciones mientras mantiene el flujo de datos y la consistencia del estado del procesador y de la memoria.

Entre las ventajas de la planificación dinámica está el aprovechamiento más óptimo en tiempo real de una etapa EX con múltiples unidades funcionales con lo que simplificamos el trabajo del compilador, aunque esto supone un aumento de la complejidad del hardware.

La ***ejecución fuera de orden*** introduce la posibilidad de tener que gestionar más riesgos que no existirían en un cauce de cinco etapas ni en un procesador segmentado con operaciones en coma flotante.

Para permitir la ejecución fuera de orden hay que desdoblar la etapa ID del procesador en:

- ***Decodificación*** (ID, Instruction Decoding): decodificación de instrucciones y comprobación de los riesgos estructurales.
- ***Emisión*** (II, Instruction Issue): la instrucción espera hasta que no hay riesgos de tipo RAW y cuando estén listos todos los operandos fuente, se leen y se emiten la instrucción hacia la unidad funcional.

El algoritmo de Tomasulo es una de las primeras técnicas basada en la planificación dinámica, de este algoritmo se derivan las técnicas de planificación dinámica que utilizan todos los procesadores superescalares actuales. El algoritmo de Tomasulo se utilizó en la unidad de coma flotante del IBM 360/91, cuyo objetivo era conseguir un alto rendimiento y evitar los grandes retrasos que se tenían en los accesos a memoria y en las operaciones de coma flotante.

La Unidad de Coma Flotante FPU contiene:

- Dos unidades funcionales, una de ***suma flotante*** y otra de ***multiplicación/división flotante***.
- Tres ficheros de registro:
  - Los registros de coma flotante ***FR***.
  - Los buffers de coma flotante ***FB***.
  - Los buffers de almacenamiento de datos ***SDB***.

La generación de direcciones y el acceso a memoria se realizan fuera de la FPU.

ÍNDICE

Índice Analítico

R

S

TEMA 2: PROCESADORES SUPERESCALARES

2.3 Características y arquitectura genérica de un procesador superescalar.

Un procesador basado en segmentaciones de un único cauce tiene ciertas limitaciones en su rendimiento. En ausencia de detenciones *el número máximo de instrucciones que se pueden emitir por ciclo de reloj es uno*.
Una alternativa para mejorar el rendimiento es reducir la duración del ciclo de reloj para aumentar el número de instrucciones ejecutadas por segundo, esto nos lleva a reducir el número de operaciones a realizar por el hardware y a aumentar la profundidad de segmentación para realizar todos los pasos que se hace en el procesamiento de una instrucción.
El aumento en el número de etapas nos lleva a un aumento en el número de buffers que las separan y un aumento en las dependencias entre instrucciones.
Todo esto se traduce en:

- Un aumento de los riesgos.
- Un aumento de las detenciones.
- Burbujas en la segmentación.

Resumiendo tenemos una pérdida mayor de ciclos.
Además un aumento en la frecuenta del reloj encarece el rendimiento global del procesador.
Otro factor importante es que cuanto más profunda es la segmentación intervienen más transistores, incrementándose la potencia consumida y el calor a disipar.

La principal diferencia de una segmentación superescalar con una clásica es que por las etapas de la segmentación pueden avanzar varias instrucciones simultáneamente, esto implica la existencia de varias unidades funcionales para que sea posible. Otra característica importante es que las instrucciones se pueden ejecutar fuera de orden, además de una segmentación más ancha.

Las segmentaciones superescalares se caracterizan por tres atributos:

- Paralelismo.
- Diversificación.
- Dinamismo.

Paralelismo.
Los procesadores segmentados presentan paralelismo de máquina temporal, es decir en un mismo instante de tiempo varias instrucciones se encuentran ejecutándose pero en diferentes etapas.
En un procesador superescalar se dan al mismo tiempo el paralelismo de máquina temporal y el espacial.
El paralelismo de máquina espacial replica el hardware permitiendo que varias instrucciones se procesen simultáneamente en una misma etapa.
El nivel de paralelismo espacial de un procesador se especifica con el término ancho o grado de la segmentación y especifica el número de instrucciones que se pueden procesar simultáneamente en una etapa.
El coste y la complejidad del hardware aumentan, hay que aumentar los puertos de lectura/escritura del fichero de registros para que varias instrucciones puedan acceder a sus operandos simultáneamente, lo mismo tenemos que hacer con las escrituras/lecturas en las cachés de datos e instrucciones, las unidades aritméticas, los buffers de contención entre etapas, etc..
Aunque un procesador superescalar es una réplica de una segmentación escalar rígida, no resulta sencillo replicar una segmentación escalar ya que tenemos que tener en cuenta varias complicaciones, hay que considerar la posibilidad de ejecutar instrucciones fuera de orden y el problema de la dependencia de datos y de memoria, los buffers entre etapas pasan a ser buffers complejos multientrada, que permiten pasar a unas instrucciones y a otras no. Otro elemento es incluir una red compleja de interconexión entre las entradas y las salidas de las unidades funcionales que permitan la resolución de la dependencia de datos.

Diversificación.
Los procesadores superescalares incluyen en la etapa de ejecución múltiples unidades funcionales diferentes e independientes, siendo habitual la existencia de varias unidades del mismo tipo.

Dinamismo.
Las segmentaciones superescalares se etiquetan como dinámicas al permitir la ejecución de instrucciones fuera de orden, respetándose la semántica del código fuente.
La segmentación superescalar cuenta con los mecanismos necesarios para garantizar que se obtengan los mismos resultados, es decir que se respeta la semántica del código fuente.
Una segmentación dinámica paralela utiliza buffers multientrada que permiten que las instrucciones entren y salgan de los buffers fuera de orden.
La ventaja que aporta la ejecución fuera de orden es que intenta aprovechar al máximo el paralelismo que permiten las instrucciones y el que permite el hardware al disponer de múltiples unidades funcionales.
Esto se traduce en:

- Unas instrucciones pueden adelantar a otras si no hay dependencias falsas, evitando ciclos de detención innecesarios.
- Una reducción de ciclos de detención por dependencias verdaderas de datos y memoria.

Los procesadores superescalares tienen la capacidad para especular, pueden realizar predicciones sobre las instrucciones que se ejecutarán tras una instrucción de salto.

2.4 Arquitectura de un procesador superescalar genérico.

El modelo de segmentación superescalar genérica consta de seis etapas:

- Etapa de lectura de las instrucciones IF (Instruction Fetch).
- Etapa de decodificación ID (Intruction Decoding).
- Etapa de distribución/emisión II (Instruction Issue)
- Etapa de ejecución EX (Execution).
- Etapa de terminación WR (Write-Back Results).
- Etapa de retirada RI (Retirement Instruction).

En una segmentación ***superescalar*** una instrucción ha sido:

- ***Distribuida*** (*dispatched*) cuando ha sido enviada a una estación de reserva asociada a una o varias unidades funcionales del mismo tipo.
- ***Emitida*** (*issued*) sale de la estación de reserva hacia una unidad funcional.
- ***Finalizada*** (*finished*) cuando abandona la unidad funcional y pasa al buffer de reordenamiento, los resultados se encuentran temporalmente en registros no accesibles al programador.
- ***Terminada*** (*completed*) o terminada arquitectónicamente, cuando ha realizado la escritura de los resultados desde los registros de renombramiento a los registros arquitectónicos, ya son visibles al programador. Se realiza la actualización del estado del procesador.
- ***Retirada*** (*retired*) cuando ha realizado la escritura en memoria, si no se necesita escribir en memoria la finalización de una instrucción coincide con su retirada.

Las ***etapas lógicas*** permiten agrupar un conjunto de operaciones que realizan una tarea completa, como por ejemplo la lectura de instrucciones.
En las implementaciones reales las etapas lógicas se encuentran segmentadas, es decir por varias etapas físicas donde cada segmento consume un ciclo de reloj.
Las dependencias de datos falsas, WAR y WAW, se resuelven en los procesadores superescalares recurriendo a un almacenamiento temporal en el que se realiza la escritura que tiene riesgo, para esto utilizamos una técnica que se llama ***renombramiento dinámico de registros*** que consiste en utilizar un conjunto de registros ocultos al programador en los que se realizan los almacenamientos temporales.

La técnica consta de dos pasos:

- ***Resolución de riesgos WAW y WAR***: se renombran de forma única los registros arquitectónicos que son objeto de una escritura, se resuelven las dependencias ya que manejan registros diferentes.
- ***Mantenimiento de las dependencias RAW***: se renombran los registros arquitectónicos fuente que corresponden a una escritura previa, el objetivo es mantener las dependencias RAW.

Una vez resueltas las dependencias en la etapa de distribución y las instrucciones se han procesado fuera de orden en la etapa de ejecución, se procede en la fase de terminación a deshacer el renombramiento y a actualizar ordenadamente los registros arquitectónicos.
Respecto a las dependencias entre instrucciones de carga/almacenamiento se presentan las mismas situaciones que con las instrucciones que operan con registros, suponiendo común la posición de memoria tenemos:

- Una carga seguida de un almacenamiento produce una dependencia RAW.
- Un almacenamiento seguida de una carga produce una dependencia WAR.
- Dos almacenamientos seguidos implican una dependencia WAW.

Una solución para respetar las dependencias que puedan existir entre las instrucciones de carga y almacenamiento es su ejecución ordenada, pero no es eficiente.
Si realizamos el renombramiento y la terminación ordenada de los almacenamientos se resuelven las dependencias WAR y WAW, pero no las RAW.
Igual que las operaciones con registros para resolver las dependencias RAW usamos adelantamiento de los operandos.
Otra forma para mejorar el rendimiento es adelantar la ejecución de las cargas.
Que se adelanten resultados o se renombren los registros para evitar dependencias WAR y WAW y aumentar así el rendimiento de las unidades funcionales no garantiza la consistencia semántica del procesador y la memoria.
Para lograr la consistencia una vez deshecho el renombramiento hay que realizar la escritura ordenada de los registros arquitectónicos en la etapa de terminación y de las posiciones de memoria en la etapa de retirada.
Tenemos que tener en cuenta que solo es posible terminar aquellas instrucciones que no sean el resultado de especular con una instrucción de salto.
El buffer de reordenamiento o terminación se convierte en la pieza fundamental para conseguir esta consistencia del procesador, ya que es el sitio en donde se realiza el seguimiento de una instrucción desde que se distribuye hasta que se termina.

2.5 Lectura de instrucciones.

La diferencia entre un procesador superescalar y uno escalar en la etapa de lectura es el número de instrucciones que se extraen de la caché de instrucciones en cada ciclo.
En un procesador escalar es una, mientras que en uno superescalar está determinado por el ancho de la etapa de lectura.
La caché de instrucciones tiene que proporcionar en un único acceso tantas instrucciones como el ancho de la etapa de lectura.
En los acceso a la caché pueden aparecer fallos de lectura deteniendo el suministro de instrucciones, los procesadores tienen mecanismos auxiliares de almacenamiento, llamados buffers de instrucciones, entre la etapa de lectura y decodificación, el objetivo es que aunque la etapa de lectura no suministre instrucciones durante uno o varios ciclos, la etapa de decodificación pueda continuar extrayendo instrucciones del buffer y continuar la tarea sin detenerse.
Los procesadores superescalares cuentan con mayor ancho de banda para la lectura de instrucciones que para la decodificación.
Al conjunto de instrucciones que se extrae simultáneamente de la I-caché se le denomina ***grupo de lectura***.
Aunque el ancho de la memoria caché de instrucciones sea suficiente pueden surgir dos problemas:

- La falta de alineamiento de los grupos de lectura.
- El cambio en el flujo de instrucciones debido a las instrucciones de salto.

2.5.1 Falta de alineamiento.

Un alineamiento incorrecto de un grupo de lectura implica que las instrucciones que forman el grupo superan la frontera que delimita la unidad de lectura de una caché.
Esto nos lleva a realizar dos accesos a la caché ya que el grupo de lectura ocupa dos bloques consecutivos.
Nos reduce el ancho de banda como mínimo a la mitad al ser necesarios dos ciclos de reloj para suministrar las instrucciones que forman el grupo de lectura y además puede provocar un fallo de lectura si la segunda parte del grupo de lectura pertenece a un bloque que no está en la caché.
Las máquinas se diseñan con ciertas restricciones de alineamiento para evitar problemas de lectura.
Si no hay restricciones de alineación podemos recurrir a utilizar un hardware adicional para realizar la extracción de la unidad de lectura con las instrucciones desordenadas y proceder a la colocación correcta de las instrucciones en el grupo de lectura mediante una *red de alineamiento* o una *red de desplazamiento*.
La *red de alineamiento* consiste en reubicar las salidas de la caché mediante multiplexores que conducen las instrucciones leídas a su posición correcta dentro del grupo de lectura.
La *red de desplazamiento* recurre a registros de desplazamiento para mover las instrucciones.
El *prefetching* o *prelectura* es una técnica utilizada para minimizar el impacto de los fallos de lectura en la caché de instrucciones.
Consiste en disponer de una cola de *prefetch* de instrucciones que usamos cuando hay un fallo de lectura en la caché de instrucciones, si las instrucciones se encuentran en la cola de *prefetch*, estas se introducen en la segmentación igual que si hubiesen sido extraídas de la I-caché.
Y al mismo tiempo el sector crítico que ha producido el fallo de lectura se trae y se escribe en la I-caché para evitar fallos futuros, en el caso que el grupo de instrucciones no esté tampoco en la cola de prefetch se lanza el fallo al siguiente nivel de la caché, el L2, y se procesa con la máxima prioridad.

2.5.2 Rotura de la secuencialidad.

Puede suceder que una de las instrucciones que forme parte de un grupo sea un salto incondicional o un salto condicional efectivo, esto provoca que las instrucciones posteriores del grupo sean inválidas, reduciéndose el ancho de banda de lectura, además de pagar un elevado coste en ciclos de reloj desperdiciados.
En una segmentación de ancho *s*, cada ciclo de parada equivale a no emitir *s* instrucciones o a leer *s* instrucciones NOP.

Esto es el ***coste mínimo de la oportunidad perdida*** y se expresa como el producto entre el ancho de la segmentación y el número de ciclos de penalización.
El *coste mínimo de la oportunidad perdida* es la cantidad mínima de ciclos que se desperdician como consecuencia de una rotura en la secuencia del código ejecutado y que obliga a anular el procesamiento de las instrucciones que hay en la segmentación, se expresa en ciclos de reloj.
Cuando la instrucción es un *salto incondicional* hay que insertar en el PC la dirección de salto dejando inservible el procesamiento de las instrucciones posteriores al salto que ya están en alguna etapa de la segmentación.
Si la instrucción es un *salto condicional* hay que esperar a conocer si el salto es efectivo o no y en caso afirmativo calcular la dirección de salto, proceder a su inserción en el PC y anular las instrucciones posteriores que estuviesen en el cauce.
La técnica de salto retardado que se aplica en los procesadores escalares para rellenar huecos en la segmentación no es válida en un procesador superescalar, por la capacidad que tienen los procesadores superescalares para ejecutar instrucciones fuera de orden cuando no existan dependencias que lo impidan.
La solución consiste en la detección temprana de las instrucciones que cambian el flujo de instrucciones para poder aplicar alguna técnica destinada a minimizar el impacto de este tipo de instrucciones.
Una técnica es efectuar una detección integrada con la extracción del grupo de lectura, que consiste en comenzar el análisis antes de que se extraiga el grupo de lectura de la I-caché, para ello se dispone de una pre-etapa de decodificación situada entre la I-caché y el siguiente nivel de memoria, el L2 de la caché. En esta pre-etapa se analiza la instrucción y como resultado se le concatenan unos bits con diversas indicaciones.

2.5.3 **Tratamiento de los saltos.**

El principal problema que plantea el tratamiento de los saltos no es descubrir que se trata de una instrucción de salto, sino los ciclos que hay que esperar para conocer el resultado de la evaluación de la condición que determina si el salto es efectivo o no y la dirección de destino, en caso de que sea efectivo.
El procesamiento de los saltos se realiza en una unidad funcional específica, ya que detener todo el cauce hasta saber cuál es la instrucción implica desperdiciar muchos ciclos.
Lo habitual es que el tratamiento de los saltos se inicie en la etapa IF durante la lectura de la I-caché, cuando las instrucciones han sido extraídas y se confirma en la etapa ID que la instrucción es un salto se puede mantener el tratamiento iniciado o anularlo.
La técnica que se utiliza en los procesadores superescalares para tratar las instrucciones de salto es especular, es decir se realizan predicciones. Al mismo tiempo que se están leyendo las instrucciones de la I-caché se comienza a predecir su efectividad y su dirección de destino, sin saber si se trata de un salto, cuestión que se confirma en la etapa ID.
La especulación también puede realizarse con los saltos incondicionales o bifurcaciones, en este caso la especulación siempre acaba coincidiendo con el resultado del salto.

2.5.4 **Estrategias de predicción dinámica**

Las técnicas que se emplean para predecir el comportamiento de las instrucciones de salto se dividen en dos grandes grupos, estáticas y dinámicas.
Las técnicas dinámicas se basan en que un salto es una instrucción cuya ejecución se repite con cierta frecuencia, lo que nos permite establecer un patrón basado en su comportamiento.
Las técnicas de predicción estática logran tasas de predicción de saltos condicionales entre un 70% y 80%.
Las técnicas de predicción dinámica obtienen tasas de acierto que oscilan entre un 80% y 95%, el inconveniente que tienen estas técnicas es el incremento del coste económico del procesador.
Para realizar una especulación completa de una instrucción de salto es necesario predecir los dos componentes que producen su procesamiento, *la dirección de destino* y *el resultado, efectivo o no efectivo*.

2.5.4.1 Predicción de la dirección de destino de salto mediante BTAC

Para reducir al máximo la penalización por salto mediante la ejecución especulativa de instrucciones debemos saber cuanto antes si una instrucción es un salto efectivo y su dirección de destino más probable.
Una técnica sencilla para predecir la dirección de destino es recurrir a una BTAC (Brach Target Address Cache), que es una pequeña ***memoria caché asociativa*** en la que se almacenan las ***direcciones de las instrucciones de saltos efectivos ya ejecutados*** o BIAs (***Brach Instruccion Addres***), y las ***direcciones de destino de esos saltos*** o BTAs (***Brach Target Address***).
El acceso a la BTAC se realiza en paralelo con el acceso a la I-caché utilizando el valor del PC. Si ninguna de las direcciones que forman el grupo de lectura coincide con alguna de las BIAs que hay en la BTAC es debido a que no hay ninguna instrucción de salto efectivo o se trata de un salto efectivo que nunca antes había sido ejecutado.
Si la BTAC no devuelve ningún valor, por defecto se aplica la técnica de ***predecir como no efectivo***, se procede normalmente con la ejecución de las instrucciones que siguen al salto y se deshace su procesamiento en caso de que el salto sea finalmente efectivo.

En este caso las penalizaciones serían:

- 0 ciclos de detención si el salto finalmente no es efectivo, al aplicarse la predicción ***como no efectivo*** el procesamiento se realiza normalmente.
- Ciclos_ID + Ciclos_II + Ciclos_EX_salto de detención si el salto es efectivo suponiendo que el salto se resuelve en la etapa EX y el vaciado del cauce de las instrucciones que seguían al salto. A continuación se actualiza la BTAC con la dirección de la instrucción de salto y el destino real del salto.

Si como predicción por defecto se opta por la técnica ***predecir como efectivo***, no se obtiene ninguna mejora, ya que siempre tenemos que esperar a la resolución real del salto para poder efectuarlo.
Si la dirección de la instrucción que se busca en la BTAC coincide con una de las BIAs se trata de un salto efectivo que ha sido ejecutado con anterioridad y se procede a extraer el valor de BTA que tenga asociado, es decir la dirección de destino asociada a la última vez que fue ejecutada esa instrucción de salto.
Una vez que tenemos la predicción de la dirección de destino lo usamos como nuevo contador de programa, se leerá de la I-caché y comenzará la ejecución especulativa de esa instrucción y de las siguientes.
Al mismo tiempo la instrucción de salto debe finalizar su ejecución con el fin de validar que la especulación es correcta.
Se pueden dar dos situaciones al conocer el resultado del salto:

- La ***predicción realizada es incorrecta*** debido a que se trata de un salto no efectivo o a que la dirección de destino especulada no coincidía con la real.
- Si la ***predicción es correcta*** se continúa con el procesamiento de instrucciones, no es necesario modificar la tabla.

Puede surgir el problema de los ***falsos positivos***, la BTAC devuelve una dirección de destino pero en el grupo de lectura no hay realmente ningún salto, esto se descubre en la etapa ID.
Esto es lo que se conoce como ***saltos fantasmas*** *o* ***saltos en falso***, el tratamiento correcto es desestimar la predicción de la BTAC una vez que se confirma que se trata de un falso positivo. Estas instrucciones que el ***predictor de destinos*** identifica inicialmente como saltos producen una penalización al tener que expulsar del cauce la instrucción de destino especulada.
Los falsos positivos son debidos a que el campo BIA de la BTAC no almacena una dirección completa, sólo una parte. Esto provoca que a direcciones distintas se le asocie la misma BIA.
Los saltos fantasma se minimizan aumentando el número de bits que componen el campo BIA de forma que haya mayor coincidencia con la dirección de la instrucción que se busca en la tabla, si la longitud de la BIA llega a igualarse a la longitud de las direcciones, el problema de los falsos positivos deja de existir.

Otra técnica para predecir la dirección de salto es utilizar una **BTIB** (***Brach Target Intruction Buffer***), su estructura es similar a la de la BTAC pero se almacena la instrucción de destino **BTI** (***Brach Target Intruction***) y algunas posteriores en lugar de solo la dirección de salto efectivo BTA.
La BTIB entrega una secuencia de instrucciones al buffer de instrucciones, como si se hubiese extraído de la I-caché y en el PC se coloca la dirección correspondiente a la siguiente instrucción que hay después de la última BTI que forma el paquete.

2.5.4.2 Predicción de destino de salto mediante BTB con historial de salto

La predicción mediante **BTB** (***Brach Target Buffer***), es una técnica de predicción dinámica similar a la BTAC pero la diferencia es que junto con la dirección de destino predicha, la BTA almacena un conjunto de bits que representan el historial del salto y predicen la efectividad del salto, BH (***Branch History***).

Al mismo tiempo que se extrae el grupo de lectura de la I-caché, se accede a la BTB en busca de alguna de las instrucciones del grupo de lectura.
Si hay un acierto se analizan los bits del historial de salto y se decide si la dirección de destino predicha ha de ser utilizada o no. La instrucción de salto se continúa procesando para validar el resultado de la especulación.

Se pueden dar ***cuatro situaciones***:

- *Se predice como efectivo y no hay error en la predicción* del resultado ni en la dirección de destino, no hay penalización y se actualiza el historial de salto.
- *Se predice como efectivo pero hay algún error en la predicción*, en el resultado, en la dirección o ambos. Se vacía el cauce de las instrucciones especuladas, se actualiza el historial de salto, se actualiza la entrada de la BTB con la dirección de destino y se comienza a procesar la instrucción indicada por el resultado del salto.
- *Se predice como no efectivo y el salto no lo es*, no hay penalización y se actualiza el historial de salto.
- *Se predice como no efectivo y el salto sí lo es*, se vacía el cauce con las instrucciones especuladas, se salta a la dirección de destino obtenida, se actualiza el historial de salto y se actualiza la entrada de la BTB.

Si ninguna de las instrucciones del grupo de lectura proporciona una coincidencia en la BTB se predice como no efectivo y se pueden plantear dos situaciones:

- El ***salto no es efectivo***: no pasa nada en el cauce.
- El ***salto es efectivo***: se vacía el cauce con las instrucciones especuladas y se salta a la dirección de destino obtenida y se incluye una entrada en la BTB con la dirección de la instrucción de salto, dirección de destino e historial de salto.

Si la instrucción no es un salto no ocurre nada.

Y por último nos falta resolver la eliminación de las entradas en la tabla debido a fallos de capacidad o de conflicto, dependiendo de la organización de la tabla:

- ***Correspondencia directa***, se sustituye la entrada.
- ***Asociatividad***, descartamos la entrada que sea menos eficiente para mejorar el rendimiento. Podemos aplicar el algoritmo LRU, eliminando la entrada que lleva más tiempo sin utilizar o eliminar la entrada con mayor posibilidad de no ser efectiva según su historial de salto.

2.5.4.3 Predictor de Smith o predictor bimodal

El predictor de Smith, o predictor bimodal, es el algoritmo de predicción dinámica del resultado de salto más sencillo.
Se basa en asociar un contador de saturación de ***k*** bits a cada salto de forma que el bit más significativo del contador indica la predicción para el salto.

- Si el bit es 0, el salto se predice como no efectivo (Not Taken o NT).
- Si el bit es 1, el salto se predice como efectivo (Taken o T).

Los ***k*** bits que forman el contador de saturación constituyen el historial de salto y es la información que se almacena en el campo BH de la BTB, estos bits junto con el resultado actual del salto se utilizan para realizar la predicción de lo que sucederá cuando se ejecute de nuevo ese salto.
En un contador de saturación cuando está en su valor máximo y se incrementa no vuelve a 0, sino que se queda igual, a su vez cuando alcanza el valor 0 y se decrementa su valor continúa siendo 0.
Cuando se invoca un salto se extrae el historial del salto de la BTB, se mira la predicción, el bit más significativo, y se aplica.
Procedemos a actualizar el contador con el resultado real del salto para dejar preparada la próxima predicción:

- Si el salto es efectivo, el contador se incrementa.
- Si el salto no es efectivo, el contador se decrementa.

Para valores altos del contador, el salto se predecirá como efectivo y para valores bajos como no efectivo.
Podemos utilizar un contador de 1 bit ($Smith_1$), pero el más utilizado es el de 2 bits ($Smith_2$), el contador de dos bits presenta cuatro posibles estados:

- SN (Stongly Not Taken): 00 Salto no efectivo.
- WN (Weakly Not Taken): 01 Salto no efectivo.
- WT (Weakly Not Taken): 10 Salto efectivo.
- ST (Stongly Taken): 11 Salto efectivo.

Las transiciones del autómata representan el resultado real del salto.
Los estados representan el historial del salto.

2.5.4.4 Predictor de dos niveles basado en el historial global

Estos predictores mantienen en un primer nivel de información un historial de los últimos saltos ejecutados, ***historial global***, o de las últimas ejecuciones de un salto concreto, ***historial local***.
En un segundo nivel, la información del primer nivel en combinación con la dirección de la instrucción de salto se utiliza para acceder a una tabla que almacena contadores de saturación que son los que determinan la predicción.
Este predictor se basa en un registro en el que se almacena el resultado de los saltos más recientes.
El historial de los últimos *h* saltos se almacena en un registro de desplazamiento de *h* bits denominado registro del historial de saltos **BHR**, ***Branch History Register***.
Cada vez que se ejecuta un salto se introduce su resultado por el extremo derecho del registro, se desplaza el contenido una posición y se expulsa el resultado más antiguo por el extremo izquierdo.
Si el salto es efectivo se introduce un 1, caso contrario un 0.
Para conocer la predicción de un salto los ***h*** bits del BHR se concatenan con un subconjunto de ***m*** bits obtenido mediante la aplicación de una función hash a la dirección de la instrucción de salto.
La combinación de ***h+m*** bits se utiliza para acceder a una ***tabla de historial de patrones*** **PHT** (***Pattern History Table***).
Esta tabla almacena en cada una de sus 2^{h+m} entradas un contador de saturación de 2 bits.
El bit más significativo del contador representa la predicción del salto, 1 efectúa el salto y 0 no.
En cuanto evaluamos el salto y conocemos su resultado hay que proceder a la actualización de los dos componentes del predictor, se incrementa o decrementa el contador de la PHT y se actualiza el historial del BHR, quedando todo preparado para la próxima predicción.
La aplicación de la función hash es fundamental para reducir la longitud de la dirección de la instrucción a ***m*** bits.

2.5.4.5 Predictor de dos niveles basado en el historial local.

Es muy similar al anterior, pero la principal diferencia es que utiliza una tabla BHT en la que se almacena el historial concreto de cada salto en lugar de un único registro BHR que mantiene un historial común para todos los saltos.
Para obtener la predicción de un salto hay que recuperar el historial del salto, el acceso a la BHT se realiza mediante un hashing de la dirección de la instrucción de salto que los reduce a ***k*** bits ya que el número de entradas de la BHT es 2^k.
Concatenamos los ***h*** bits del historial con los *m* bits obtenidos mediante otro hashing de la dirección de la instrucción de salto.
Con los ***h+m*** bits accedemos a la PHT para recuperar el estado del contador de saturación de 2 bits.
La función hash puede ser muy básica y consistir en quedarse con los ***m*** o ***k*** bits menos significativos de la dirección, pudiendo recurrir a funciones más complejas.
En cuanto evaluamos el salto y conocemos su resultado hay que proceder a la actualización de los dos componentes del predictor.
Accedemos al PHT para actualizar el contador de saturación con el resultado, se suma 1 si fue efectivo, 0 en caso contrario.
Se accede al historial del BHT para introducir el resultado del salto, 1 si fue efectivo, 0 en caso contrario, desplazando su contenido.
Quedando actualizados los dos niveles de predictores para la predicción del siguiente salto.

2.5.4.6 Predictor de dos niveles de índice compartido gshare

El ***predictor gshare*** es una variante del predictor de dos niveles de historial global.
Se realiza una función XOR entre los *h* bits superiores de los *m*.
Los *h* bits obtenidos de la XOR se concatenan con los *m-h* bits restantes para poder acceder a la PHT.

2.5.4.7 Predictores híbridos

Los procesadores superescalares recurren a dos predictores que generan un resultado y un selector que se ocupa de decidir cuál de las dos predicciones hay que utilizar, se conoce como ***predicción híbrida***.

2.5.5 Pila de dirección de retorno

El ***retorno de subrutina*** es una instrucción especial de salto que no puede predecirse adecuadamente ya que cada vez que se la invoca puede saltar a una dirección completamente diferente, la BTB generaría predicciones de la dirección de destino con una elevada tasa de fallos.
Las invocaciones a una subrutina se pueden realizar desde distintos lugares de un programa, por eso la instrucción de salto que hay al final de la subrutina para devolver el control no tiene una dirección de destino fija.
Esta instrucción de salto nunca obtiene predicciones correctas de la BTB ya que corresponderán a la dirección de retorno de la invocación previa.
El tratamiento correcto de los retornos de subrutina se realiza mediante una ***pila de direcciones de retorno*** **RAS** (***Return Addres Stack***)o ***buffer de pila de retornos*** **RSB (*Return Addres Buffer*)**.
Cuando se invoca una subrutina mediante una instrucción de salto se efectúan tres acciones:

- Se accede a la BTB para obtener la predicción de la dirección de destino.
- Se especula el resultado del salto.
- Se almacena en la RAS la dirección de la instrucción siguiente al salto.

Una vez procesadas las instrucciones de la subrutina, se invoca una instrucción de retorno de subrutina, cuando se detecta que la instrucción es de retorno se accede a la RAS para obtener la dirección correcta de retorno y se desestima la predicción de la BTB.
Las instrucciones procesadas como consecuencia de una especulación incorrecta dada por la BTB son anuladas.
El problema que tiene la RAS es el desbordamiento cuando tratamos subrutinas con muchos anidamientos y la pila no ha sido dimensionada correctamente.

2.5.6 Tratamiento de los errores en la predicción de los saltos

Uno de los problemas de la ejecución especulativa de instrucciones es que el resultado de la predicción del salto no coincida con el resultado verdadero del salto. En este caso se deshace el procesamiento de todas las instrucciones y se continúa con el procesamiento correcto, esto es la ***recuperación de la secuencia correcta***.
La forma habitual para conocer y validar o anular las secuencias de instrucciones que se están ejecutando de forma especulativa desde que entran en la fase de distribución hasta que son terminadas es etiquetarlas durante todo el tiempo que estén en la segmentación, para conocer su estado en cualquier etapa del cauce.

Dos de esas etiquetas son:

- La especulativa.
- La de validez.

Las etiquetas son campos de uno o más bits que hay en el buffer de reordenamiento, aquí todas las instrucciones tienen una entrada en la que se actualiza su estado desde que son distribuidas y hasta que son terminadas arquitectónicamente, se dice que están ***en vuelo***.
Si la ***etiqueta especulativa*** es de un bit, el valor de 1 identifica la instrucción como ***instrucción especulada***.
Si el procesador permite varias rutas en paralelo, se denominan ***nivel de especulación***, la etiqueta dispondrá de más bits para identificar cada uno de los bloques de instrucciones especuladas.
A continuación hay que especificar la dirección de las instrucciones que se almacenan en una tabla junto con la etiqueta especulativa que se asocia a las instrucciones.

Posteriormente esta etiqueta permite identificar y validar las instrucciones especuladas si la predicción es correcta o realizar la recuperación de la secuencia correcta en caso de error.

La etiqueta ***validez*** permite saber si la instrucción debe o no terminarse arquitectónicamente, es decir si el resultado se escribe en un registro arquitectónico o en la memoria.
Inicialmente todas las instrucciones son válidas, aunque no puede terminarse arquitectónicamente hasta que la etiqueta que la define como especulativa cambie a no especulativa, todos los bits a 0.
En el momento en que se evalúa el salto, si la predicción coincide con el resultado, las etiquetas se cambian de manera que las instrucciones especuladas a ese salto son correctas y terminan arquitectónicamente.

Si la predicción es incorrecta hay que realizar dos acciones:

- ***Invalidar las instrucciones especuladas***, el bit de validez de todas esas instrucciones pasa a indicar invalidez y no son terminadas arquitectónicamente, no se accede al banco de registros o a la memoria para escribir sus resultados.
- ***Recuperar la ruta correcta***, implica iniciar la lectura de instrucciones desde la dirección de salto correcta. Si la predicción incorrecta fue:
    - No realizar el salto se utiliza el resultado del salto como nuevo valor del PC.
    - Realizar el salto, se accede a la tabla en la que se almacenó la dirección de la instrucción de salto y se utiliza para obtener el nuevo valor del PC, el de la siguiente instrucción (*ruta fall-through*).

2.6 Decodificación

Tras la extracción simultánea de varias instrucciones el siguiente paso es la decodificación. Es una de las etapas más críticas en un procesador superescalar.
En un procesador RISC superescalar en una etapa de decodificación tiene que decodificar varias instrucciones en paralelo e identificar las dependencias de datos con todas las instrucciones que componen el grupo de lectura y con las que se están ejecutando en las unidades funcionales.
Otra complicación es que serían varias las instrucciones que necesitarían leer todos sus operandos o parte de ellos. También se debe confirmar si hay saltos en falso en el grupo de lectura para anular la ruta especulada y minimizar el impacto del procesamiento erróneo.
Por todo esto los procesadores superescalares reparten estas tareas en dos etapas:

- ***Decodificación***, detecta los saltos en falso y realiza la adaptación del formato de las instrucciones a la estructura interna de control y datos del procesador.
- ***Distribución***, se ocupa del renombramiento de los registros, de la lectura de los operandos y del análisis de las dependencias verdaderas.

En los procesadores CISC las instrucciones pueden tener longitudes diferentes, hay que analizar el código de operación de cada instrucción para conocer su formato y longitud.
En los RISC la longitud de instrucción es fija, sabemos donde empieza y acaba una instrucción, además la extracción del buffer de instrucciones se realiza muy rápido sin importar si es una o varias instrucciones a decodificar.
Los factores que determinan la complejidad de la etapa de decodificación son:

- El tipo arquitectónico del procesador (RISC o CISC).
- Número de instrucciones a decodificar en paralelo (ancho de segmentación).

Se suelen adoptar tres soluciones:

- ***Descomponer la etapa*** de decodificación en varias subetapas o fases, aumentando el número de ciclos de reloj que se emplean.
- Realizar una parte de la decodificación antes que la extracción de instrucciones de la I-caché, esto se conoce como etapa de ***predecodificación o decodificación previa***.
- ***Traducción de instrucciones***, se descompone una instrucción en instrucciones más básicas, o unir varias instrucciones en una única instrucción interna.

Una vez completada la decodificación, las instrucciones pasan al buffer de distribución y a las estaciones de reserva para iniciar su verdadero procesamiento, desde los buffers se reparten las instrucciones entre las unidades funcionales del procesador.

2.6.1 Predecodificación

Esta etapa está situada antes de la I-caché de forma que las instrucciones que recibe la I-caché desde la caché de nivel 2 o memoria principal como consecuencia de un fallo de lectura pasan obligatoriamente por esta etapa.
Está constituida por hardware situado antes de la I-caché que realiza una decodificación parcial de las instrucciones, este proceso analiza cada instrucción y la concatena un pequeño conjunto de bits con información sobre ella.
Los ***bits de predecodificación*** son un conjunto de bits que se añaden a cada instrucción cuando son enviados a la I-caché para simplificar las tareas de decodificación y distribución.
Se utilizan posteriormente en la etapa de *fetch* para determinar el tipo de instrucción de salto que hay en el grupo de lectura y proceder o no a su especulación y en la etapa de decodificación para determinar la forma en que se agrupan para su posterior distribución.
Estos bits también identifican las instrucciones que son susceptibles de provocar una excepción.

Inconvenientes de la decodificación previa:

- La necesidad de un mayor ancho de banda.
- El incremento del tamaño de la I-caché.

Otra forma de hacer la predecodificación de instrucciones es situar esta fase entre el buffer interno de fetch que recibe las instrucciones de la I-caché y el buffer de instrucciones que alimenta la etapa de decodificación. Esta etapa se suele considerar parte de la etapa *fetch* y no de la de decodificación.

La técnica es propia de los CISC y el objetivo de estos bits es:

- Determinar la longitud de las instrucciones.
- Decodificar ciertos prefijos asociados a las instrucciones.
- Señalar determinadas propiedades de las instrucciones a los decodificadores.

La precodificación adelanta parte del trabajo que realiza la etapa de decodificación reduciendo la profundidad de su segmentación. Una segmentación menos profunda permite que la recuperación de un fallo en la especulación de un salto no encarezca los ciclos de reloj.

2.6.2 **Traducción de instrucciones**

En la fase de decodificación se realiza la traducción de una instrucción compleja en un conjunto de instrucciones más básicas de tipo RISC.
Estas operaciones básicas se llaman:

- En Intel microoperaciones (micro-ops).
- En PowerPC operaciones internas (IOPs –Internal OPerations).
- En AMD ROPs (Rips Operations).

Esta técnica es propia de arquitecturas CISC y también de las RISC para reducir la complejidad de ciertas instrucciones. La intención es simplificar el repertorio original de instrucciones para que su procesamiento hardware se pueda realizar directamente en el procesador.

La arquitectura Intel Core complementa la traducción de instrucciones con dos técnicas adicionales:

- ***Macro-fusión***, fusiona ciertos tipos de instrucción en la fase de predecodificación y las envía a uno de los decodificadores para producir una única *micro-ops*, denominada *macro-fused micro-op*. La macro-fusión permite realizar más trabajo con menos recursos hardware. Esto hace aumentar el ancho de banda de decodificación, ya que el buffer se vacía más rápido al retirar dos instrucciones por ciclo. También se produce un incremento virtual del ancho de la segmentación en el núcleo de ejecución dinámica (número de unidades funcionales).
- ***Micro-fusión***, decodifica una instrucción que genera dos *micro-op* y las funde en una *micro-op*. La nueva *micro-op* solo ocupa una entrada en el buffer de reordenamiento (ROB), al emitirse se desdobla en sus dos componentes originales ejecutándose en paralelo en unidades funcionales diferentes o en serie si es una. La aplicación más habitual es con instrucciones de almacenamiento ya que implica su desdoblamiento en dos *micro-ops*, una para el cálculo de la operación de destino y otra para la escritura del dato. Produce un aumento de la capacidad de decodificación y de la eficiencia energética de la arquitectura al emitir más *micro-op* con menos hardware.

En los PowerPC se denomina ***instrucciones rotas*** a las que se descomponen en dos IOPs, y las ***instrucciones microcodificadas*** a las que se descomponen en tres o más IOPs (Internal Operations).

2.7 Distribución

En esta etapa se establece el punto de partida para la ejecución de instrucciones en paralelo y fuera de orden en una segmentación superescalar.
Se reparten las instrucciones según su tipo entre las distintas unidades funcionales para proceder a la ejecución en paralelo.
La distribución es el último componente del *front-end* de un procesador superescalar después de la etapa de *fetch* y de decodificación.
Es el punto de inflexión entre el procesamiento centralizado de las instrucciones y el procesamiento distribuido.
Después de la decodificación, las instrucciones se depositan temporalmente en dos buffers llamados ***buffer de distribución*** o ventana de instrucciones y ***buffer de terminación*** o de reordenamiento.
Para ejecutar una instrucción necesitamos todos los operandos fuente y estar libre una de las unidades funcionales.
Puede ocurrir que los operandos no estén disponibles y haya que esperar por algún resultado de instrucciones que están ejecutándose.
Puede ocurrir que los operandos estén disponibles pero no su unidad funcional o también que todo esté disponible pero que no podamos enviar la instrucción a la unidad funcional debido a que no hay suficientes buses, hay un límite en la cantidad de instrucciones por ciclo que se pueden enviar desde la ventana de instrucciones a las unidades funcionales, son los riesgos estructurales.
Este problema lo resolvemos deteniendo la instrucción en la etapa de decodificación hasta que todo esté listo para poder emitirla, esta no es una buena solución ya que reduce el rendimiento de la decodificación y del procesador introduciendo burbujas en la segmentación.
La solución que vamos a utilizar es *desacoplar la etapa de decodificación de la de ejecución* utilizando la ventana de instrucciones.
El ***desacoplo*** consiste en decodificar la instrucción al máximo y no detenerla para que avance hacia la ventana de instrucciones.
En la ventana de instrucciones se deposita la instrucción con los identificadores de los operandos fuente y además se indica mediante un bit de validez por operando si está disponible.
Cuando se cumplan las condiciones necesarias se emitirá la instrucción que está a la espera en la ventana de instrucciones.
Resumiendo, este mecanismo permite distribuir y emitir las instrucciones de forma que se respeten las dependencias verdaderas.

2.7.1 Organización de la ventana de instrucciones

Existen varias formas de organizar la ventana de instrucciones:

- ***Estación de reserva centralizada***, es lo que hemos definido hasta ahora como ventana de instrucciones o buffer de instrucciones (Ventana de emisión o cola de emisión). Tiene un hardware de control muy complejo. Los términos distribución y emisión significan lo mismo ya que la asociación de la instrucción a la unidad funcional se produce en el momento del envío.
- ***Estaciones de reserva distribuidas o individuales***, cada unidad funcional dispone de una estación de reserva propia. Un buffer de distribución que recibe las instrucciones de la etapa de decodificiación se ocupa de distribuirlas a las estaciones de reserva individuales según su tipo. Esta estructura aumenta la complejidad de los buses que hay que utilizar para reenviar los resultados de las distintas unidades funcionales a las estaciones de reserva y bancos de registro para emitir nuevas instrucciones. La distribución es el envío desde el buffer de distribución a la estación de reserva individual y la emisión es el envío desde la estación de reserva individual a la unidad funcional para que se ejecute.
- ***Estaciones de reserva en clústers o compartidas***, las estaciones de reserva reciben las instrucciones del buffer de distribución pero una estación de reserva da servicio a varias unidades funcionales del mismo tipo. La distribución es el envío desde el buffer de distribución a una estación de reserva y la distribución/emisión se produce al enviar la instrucción a una de las unidades funcionales asignada.

La utilización de una configuración depende del equipo de diseño.

Distribución: asociar una instrucción a una unidad funcional.
Emitir: enviar la instrucción a la unidad funcional para comenzar la ejecución.

2.7.2 Operativa de una estación de reserva individual

Una estación de reserva es un buffer de almacenamiento con múltiples entradas en donde se almacenan las instrucciones ya decodificadas.
Cada entrada del buffer es un conjunto de bits agrupados por campos y representa una instrucción, está formado por los siguientes campos:

- ***Ocupado*** (O): la entrada está ocupada por una instrucción válida pendiente de emisión.
- ***Código de operación*** (CO): contiene el código de operación de la instrucción.
- ***Operando 1*** (Op1): si el registro correspondiente al primer operando fuente está disponible, este campo almacena el valor del registro o el identificador. Si no está disponible contiene el identificador del registro.
- ***Válido 1*** (V1): indica si el operando fuente está disponible o no.
- ***Operando 2*** (Op2): similar a Op1
- ***Válido 2*** (V2): similar a V1
- ***Destino*** (D): almacena el resultado de la operación de forma temporal.
- ***Listo*** (L): indica que todos los operandos ya están disponibles y la instrucción puede emitirse a la unidad funcional correspondiente.

El formato de las entradas de las estaciones de reserva es similar tanto si están organizadas de forma individual o compartida, la diferencia surge en las entradas del buffer de distribución que se ocupa de distribuir las instrucciones a las estaciones individuales o agrupadas.
En las estaciones individuales y en las compartidas las entradas del buffer de distribución no disponen de bits de validez ya que se asignan cuando las instrucciones son enviadas desde el buffer a las estaciones.
En una estación de reserva centralizada los bits de validez se establecen al salir de la etapa de decodificación.
El mecanismo de las estaciones de reserva resuelve el problema de las dependencias RAW, esto ocurre al forzar la espera de los operandos pero a la vez permitiendo la ejecución distribuida y fuera de orden de las instrucciones no dependientes.
El hardware de control para la asignación y emisión de instrucciones es bastante complejo.
Las fases por las que pasa una instrucción desde que sale del buffer hasta que se emite a la estación de reserva son: ***distribución, supervisión y emisión***.

2.7.2.1 Fase de distribución

Consiste en el envío de una instrucción desde el ***buffer de distribución*** *o* ***ventana de instrucciones*** a la estación de reserva individual que le corresponda según su tipo.
La introducción de instrucciones en la estación de reserva se realiza de forma ordenada por parte de la ***lógica de asignación***.

La lógica de asignación se ocupa de:

- Ubicar correctamente la instrucción recibidas, utiliza un registro de las instrucciones almacenadas y de las entradas que están libres.
- Establece inicialmente los bits de validez.

Si se utiliza una estación de reserva centralizada el origen de las instrucciones es la etapa de decodificiación.

2.7.2.2 Fase de supervisión

Cuando la instrucción está almacenada en la estación de reserva, comienza la fase de supervisión y termina cuando están los dos operandos fuente disponibles y está lista para ser emitida.
Las instrucciones que tienen algún operando fuente marcado como no disponible se encuentran en espera activa, supervisando los buses de reenvío CDB (Common Data Bus), que es donde cada unidad funcional publica el resultado y el identificador del registro destino en el que almacenar el resultado.
El hardware que realiza la supervisión de los buses se suele denominar ***lógica de activación***.
La lógica de activación compara continuamente los identificadores de los operandos no disponibles de todas las instrucciones que hay en la estación de reserva con los identificadores que se publican en los buses de reenvío.
En cuanto haya una coincidencia de identificadores se cambia el bit de validez del operando fuente correspondiente, se lee el valor del operando del bus de reenvío y si todos los operandos están listos se activa el bit que señala a la instrucción como preparada para ser emitida, *bit L*.
La activación del bit L se conoce como ***activación de la instrucción***.
La complejidad de la lógica de activación es elevada y aumenta con el tamaño y el número de estaciones de reserva.

2.7.2.3 Fase de emisión

Cuando la instrucción tiene todos sus operandos disponibles comienza la fase de emisión.
En esta fase las instrucciones se mantienen en espera activa hasta que la ***lógica de selección*** determina la que se puede emitir.
La fase de emisión da paso a la etapa de ejecución.
La lectura de los operandos se realiza en el momento en que la instrucción se emite a la unidad funcional.
Al mismo tiempo el código de la operación y el identificador del registro destino se envían a la unidad funcional desde la estación de reserva y los operandos se leen del fichero de registros y se remiten a la unidad funcional.
Cuando se emite una instrucción se libera la entrada asociada para que se pueda distribuir una nueva instrucción.
Para las instrucciones que pueden provocar algún tipo de interrupción se mantienen en la estación de reserva hasta que no haya concluido completamente su ejecución.
Si en una estación de reserva individual solo hay una instrucción lista para emitir no hay problema, se envía a la unidad funcional y se marca su entrada como libre para poder ser ocupada por otra instrucción.
El problema se presenta cuando hay varias instrucciones listas para ser emitidas en el mismo ciclo de reloj y la lógica de selección debe decidir cuál hay que emitir.
La lógica de selección es un ***algoritmo de planificación o planificador dinámico***.
El algoritmo más habitual selecciona la instrucción más antigua para emitirla a la espera de que la unidad funcional esté libre y pueda comenzar la ejecución de la instrucción.
Los primeros procesadores superescalares realizaban una ***emisión con bloqueo y ordenada,*** esto implica que las instrucciones salen en orden de la estación de reserva excepto que no tuviese todos sus operandos disponibles debiendo esperar y bloqueando las instrucciones posteriores.
Los procesadores actuales ya realizan ***emisión sin bloqueo y desordenada***, mejorando notablemente su rendimiento.

La emisión de instrucciones desde las estaciones de reserva puede realizarse de forma:

- ***Alineada***, la ventana de distribución no puede enviar nuevas instrucciones a la estación de reserva hasta que no esté completamente vacía.
- ***No alineada***, se pueden distribuir instrucciones desde el buffer de distribución siempre que queden entradas libres en las estaciones de reserva.

2.7.3 **Lectura de los operandos**

La primera forma de efectuar la lectura de los operandos se realizaba siempre en el momento en que se emitían las instrucciones desde las estaciones de reserva individuales a las unidades funcionales, se conoce como ***planificación sin lectura de operandos***.
Cuando la instrucción se emite por parte del planificador, todavía no se han extraído los valores de los operandos fuente del fichero de registros.
Cuando una instrucción se distribuye desde la ventana de instrucciones a una estación de reserva individual las entradas contienen únicamente identificadores de los registros fuente. Se analizan las instrucciones en busca de dependencias verdaderas y se marcan en el fichero de registros como inválidos los que resulten ser destino de una operación.
La lógica de asignación se encarga de asignar los bits de validez de los operandos fuente mediante una consulta al fichero de registros.
Consideraremos que la asignación de los bits de validez al fichero de registros y a las entradas a las estaciones de reserva se realizan en la fase de distribución y no en la de decodificación.
Una de las ventajas de este modo de organización es que el ancho de las estaciones de reserva y de los buses de reenvío se reduce mucho.
El inconveniente es que el tiempo desde la emisión de la instrucción hasta que comienza su ejecución es mayor por el tiempo necesario para extraer los operandos.
La segunda forma de efectuar la lectura de los operandos es cuando la instrucción se distribuye desde el buffer de distribución a las estaciones de reserva, esto es la ***planificación con lectura de operandos***.
Las instrucciones se distribuyen a las estaciones de reserva y los valores de los operandos fuente disponibles en el fichero de registros son leídos y copiados en la estación de reserva.
El código de operación y el identificador de registro se copian directamente en la estación de reserva desde el buffer de distribución y los identificadores de los operandos fuente se envían al fichero de registros.
Si el operando está disponible en el fichero de registros, la estación de reserva recibe el valor y coloca el bit de validez de su entrada a 1.
Si el registro no está disponible se reenvía el identificador a la estación y se coloca el bit de validez a 0.
Tras la emisión de la instrucción la unidad funcional publica en el bus de reenvío el resultado de su operación con el identificador del registro destino, se copia el resultado en el fichero de registros y en algunas entradas de la estación de reserva.
En el fichero de registros se actualiza el valor del registro y se modifica el bit de validez para indicar que ya está disponible y no es el registro destino de ninguna instrucción posterior.
Las estaciones de reserva reemplazan el identificador que presenta alguna coincidencia por el valor del operando y su bit de validez se pone a 1.
El identificador de los registros detecta la disponibilidad de un operando y procede a la activación de las instrucciones.

2.7.4 **Renombramiento de registros**

Aunque la ejecución fuera de orden permite maximizar el rendimiento de los procesadores superescalares tiene una serie de inconvenientes:

- La gestión de las dependencias falsas WAR, ***antidependencias*** y
- WAW ***dependencias de salida***.

Si el **ISA** (***Instruction Set Architecture***) tuviese un número infinito de registros no existirían las dependencias falsas, que surgen como consecuencia de reutilizar los registros accesibles por el repertorio de instrucciones para efectuar el almacenamiento temporal de los resultados.
El ***fichero de registros creados*** son los registros accesibles al programador, que son un recurso limitado y que pueden utilizarse por el ISA. También se conoce como ***fichero de registros arquitectónicos***.
Para generar el código objeto, el compilador utiliza un número ilimitado de registros simbólicos como almacenamiento temporal, manteniendo el máximo de datos en los registros.
De esta forma se minimizan los accesos al sistema de memoria ya que se consumen muchos ciclos.

Para general el código definitivo el compilador realiza una asignación del conjunto infinito de registros simbólicos al conjunto finito de registros arquitectónicos.
El compilador utiliza la siguiente regla, se escribe un nuevo valor en un registro cuando se detecta que el valor almacenado en el registro ya no es necesario para operaciones futuras.
El ***rango de vida de un registro*** es el tiempo durante el que el dato almacenado en un registro es válido, es decir el tiempo entre dos etapas consecutivas.
Dentro del rango de vida se pueden efectuar múltiples lecturas del valor ya que estas operaciones no le quitan validez al dato.
El final y comienzo del rango de vida del mismo registro se produce debido a que se realiza el reciclaje de ese registro.
Los rangos de vida de diferentes registros se pueden intercambiar y mezclar siempre que se respeten las dependencias verdaderas.
La solución adoptada para resolver las dependencias falsas de datos es el ***renombramiento dinámico de los registros de la arquitectura mediante hardware***.
Consiste en utilizar un conjunto de registros auxiliares, invisibles al programador, de forma que se restablezca la correspondencia única entre resultados temporales y registros.
Se conocen como ***registros físicos***, ***registros no creados*** o ***registros de renombramiento***.
Las instrucciones escriben sus resultados en los registros no creados para después deshacer el renombramiento y proceder a la escritura ordenada de los registros arquitectónicos utilizando los valores de los registros no creados.
Este restablecimiento permite eliminar todas las dependencias falsas entre las instrucciones emitidas.

Este proceso consta de dos pasos:

- ***Resolución de los riesgos WAW y WAR***. Se renombran de forma única los operandos destino de las instrucciones y de esta manera se resuelven las dependencias WAW y WAR.
- ***Mantenimiento de las dependencias RAW***, se renombran todos los registros fuente que son objeto de una escritura previa, el objetivo de este paso es respetar las dependencias RAW.

En realidad el procesador realiza el renombramiento instrucción tras instrucción, analizando si los dos operandos fuente, renombrados o no, están disponibles y efectuando el renombramiento del registro destino.
Para realizar esto se recurre a un hardware adicional que trabaja con el fichero de registros arquitectónicos.
La fase de renombramiento se puede realizar en la fase de decodificación o en la de distribución.
El renombramiento dinámico se realiza incluyendo en el procesador un nuevo fichero de registros denominado ***fichero de registros de renombramiento*** **RRF** (*Rename Register File*) o ***buffer de renombramiento***.
Al ***fichero de registros de la arquitectura*** se denominará **ARF** (*Architected Register File*).

Existen tres formas de organizar el **RRF**:

- Como un único fichero de registros formado por la suma del RRF y ARF.
- Como estructura independiente pero accesible desde ARF.
- Como parte del buffer de reordenamiento y accesible desde el ARF.

2.7.4.1 Organización independiente del RRF con acceso indexado

A continuación se describe la implementación del RRF como una estructura independiente del ARF.

Las entradas del ARF se componen de tres campos:

- **Datos**, contiene el valor del registro.
- **Ocupado**, indica si el registro ha sido renombrado.
- **Índice**, apunta a la entrada del RRF.

La estructura de los registros RRF tienen tres campos:

- **Datos**, se escribe el resultado de la instrucción que ha causado el renombramiento.
- **Ocupado**, tiene un bit de longitud, comprueba si el registro está siendo utilizado por instrucciones pendientes de ejecución y no pueden liberarse.
- **Válido**, tiene un bit de longitud, indica que aún no se ha realizado la escritura en el RRF.

La lectura de un registro puede encontrarse frente a tres situaciones:

- El registro ARF no está pendiente de ninguna escritura.
- El registro ARF es destinatario de una escritura por lo que ha sido renombrado. Una vez que se accede al RRF se pueden plantear dos situaciones según el estado del registro de renombramiento:
    - Campo *Válido = 0*. La actualización del contenido del registro RRF con el resultado de la instrucción que provocó el renombramiento está pendiente, el operando no está disponible y se procede a enviar el identificador del registro de renombramiento a la estación de reserva.
    - Campo *Válido = 1*. El valor del registro de renombramiento con el resultado de la instrucción de escritura se ha actualizado, el operando está disponible y se puede extraer el valor del RRF si una instrucción lo necesita.

El proceso de renombrar el registro destino de una instrucción cuando ésta se distribuye a la estación de reserva individual es sencillo.
Una vez que una instrucción finaliza su ejecución, se produce la escritura del resultado de la operación en el registro RRF y la colocación del campo *Válido* a 1.
Las situaciones que se pueden plantear el realizar el renombramiento en función del estado de la instrucción de escritura que forzó el renombramiento son:

- ***Instrucción pendiente de escritura***.
- ***Instrucción finalizada***.
- ***Instrucción terminada***.

Si al renombrar un registro el campo *Ocupado* ya está asignado a 1, se efectúa un nuevo renombramiento buscando un registro en el RRF que esté libre.
Si al terminar una instrucción de escritura y deshacer el renombramiento de su registro destino hay que distribuir una nueva instrucción cuyo operando fuente coincide con el destino de la escritura terminada, la instrucción de lectura leerá el valor del operando del ARF ya que en ese momento el registro ya no está renombrado.
Si la instrucción de lectura se distribuye mientras que la escritura que forzó el renombramiento ha finalizado pero no ha terminado, la instrucción de lectura se almacenará en la entrada de la estación de reserva utilizando como operando fuente el valor del registro de renombramiento no su identificador.
La técnica del renombrado de registros implica que el contenido de los operandos fuente en las estaciones de reserva puede ser de tres tipos:

- El valor del registro original, no hay escritura pendiente.
- El identificador del registro de renombramiento, escritura pendiente de finalizar.
- El valor del registro de renombramiento, escritura finalizada pero no terminada.

Las condiciones necesarias para poder distribuir una instrucción son tres:

- Una entrada libre en el RRF.
- Una entrada libre en la estación de reserva individual que le corresponda según su tipo.
- Una entrada libre en el buffer de renombramiento.

2.7.4.2 Organización independiente del RRF con acceso asociativo

Otra forma de organizar el RRF de forma independiente pero introduciendo cambios en su estructura es si se accede de forma asociativa mediante una búsqueda en el RRF del identificador del registro del ARF que provoca el renombramiento.
En este caso el ARF no tiene campo Índice ya que para acceder al registro de renombramiento se utiliza el identificador del registro destino ARF que provoca el renombrado.

La estructura de los registros RRF tienen cinco campos:

- ***Destino***, se almacena el identificador del registro ARF que provoca el renombramiento y que se utiliza para realizar la búsqueda y validar la coincidencia.
- ***Datos***, se escribe el resultado de la instrucción que ha causado el renombramiento.
- ***Ocupado***, tiene un bit de longitud, comprueba si el registro está siendo utilizado por instrucciones pendientes de ejecución y no pueden liberarse.
- ***Válido***, tiene un bit de longitud, indica que aún no se ha realizado la escritura en el RRF.
- ***Último***, efectúa la labor del campo índice del ARF con acceso indexado, de forma que si está a 1 indica que es el último registro de renombramiento asignado al registro del ARF.

2.7.4.3 Organización del RRF como parte del buffer de reordenamiento

Otro procedimiento para integrar el RRF en el núcleo del procesador es como parte del buffer de reordenamiento o terminación.
El buffer de reordenamiento permite recuperar el orden de las instrucciones y concluir su procesamiento manteniendo la semántica.
Para incluir el RRF en el buffer de reordenamiento añadimos los campos *Datos* y *Válido* con la misma función que ya hemos visto.
La diferencia está en el campo *Índice* del ARF que contiene un puntero a una de las entradas del buffer de reordenamiento ya que el buffer hace RRF.
Se prescinde del campo *Ocupado* ya que este campo existe en el buffer de terminación.
Si se produce un renombramiento de un registro por ser el operando destino de una instrucción, el campo *Índice* del registro apuntará a la entrada del buffer de reordenamiento que ocupa la instrucción que renombra el operando destino.
En el instante en que la instrucción termine y se libere su entrada en el buffer de terminación, se accederá con el identificador de esa entrada al ARF y si hubiese coincidencia se actualizará al campo *Datos* y el bit de *Ocupado* se establecerá a 0, finalizando el renombramiento.
Al realizar la distribución de una instrucción pueden darse las siguientes situaciones:

- Si el registro no está renombrado, se lee su valor del campo *Datos* del ARF y se utiliza como operando fuente en la estación de reserva.
- Si está renombrado, se sigue el puntero del campo Índice y se accede a una entrada del buffer de reordenamiento, puede ocurrir:

    - Si el bit de Válido está a 0, está a la espera de la finalización de la instrucción que tiene que actualizar el contenido del *registro de renombramiento*, el campo Datos de la entrada del buffer.
    - Si el bit de Válido está a 1, está pendiente de la terminación de la instrucción, como operando fuente para la estación de reserva se utiliza el valor almacenado en el campo *Datos* del buffer de terminación.

La inclusión del RRF en el buffer de terminación añade un campo Datos y un campo Válido al buffer de terminación cuando existen instrucciones que no provocan renombramiento de registros.
Lo importante al realizar el diseño del RRF es alcanzar una elevada velocidad de renombramiento, influyendo en el rendimiento de la etapa de distribución, ejecución y terminación.

2.8 Terminación

Una vez que las instrucciones han completado su etapa de ejecución, se quedan almacenadas en el buffer de reordenamiento a la espera de su terminación arquitectónica.
Una instrucción se considera terminada arquitectónicamente cuando actualiza el estado del procesador manteniendo la ***consistencia***.
Existe ***consistencia del procesador*** cuando las instrucciones concluyen su procesamiento en el mismo orden secuencial en el que se encuentran en el programa, o lo que es lo mismo en el mismo orden en el que iniciaron el procesamiento.
Mantener la consistencia es fundamental por dos motivos:

- Para garantizar el resultado final del programa, tenemos que respetar la semántica del programa que viene definida por el orden secuencial de las instrucciones.
- Para permitir un tratamiento correcto de las interrupciones, tenemos que conocer el estado del procesador antes de procesar una información.

Estar a la espera de la terminación arquitectónica es estar a la espera de copiar en un registro arquitectónico el resultado temporal almacenado en un registro de renombramiento.
Las instrucciones de almacenamiento necesitan pasar por la etapa de retirada, este es el momento en que escriben sus resultados ordenados en la D-caché.

- Una instrucción ha ***finalizado*** cuando abandona la unidad funcional y queda a la espera en el buffer de terminación.
- Una instrucción ha ***terminado*** cuando ha actualizado el estado de la máquina. Solamente se pueden terminar las instrucciones que han finalizado y no están marcadas como especulativas o como inválidas en el buffer de terminación.
- Una instrucción se ha ***retirado*** cuando ha escrito su resultado en memoria. Este estado sólo pueden alcanzarlo las instrucciones de almacenamiento.

La pieza clave que garantiza la consistencia del procesador es el buffer de reordenamiento o terminación ya que gestiona la terminación ordenada de todas las instrucciones que están en vuelo. La terminación ordenada asegura que las interrupciones se puedan tratar correctamente.
Cuando una instrucción fuerce una excepción, el buffer de reordenamiento permitirá expulsar del cauce todas las instrucciones posteriores para garantizar la terminación ordenada de las anteriores.
El *buffer de reordenamiento* pone fin a la ejecución fuera de orden de las instrucciones y forma el *back-end* de un procesador superescalar junto con la etapa de retirada.
Este buffer decide cuándo los resultados almacenados temporalmente en el RRF se tienen que escribir en el ARF y puede darse por concluida una instrucción.
El *buffer de reordenamiento* es una estructura que mantiene entradas con el estado de todas y cada una de las instrucciones que hay en vuelo.
La composición de una entrada del buffer de reordenamiento, si tenemos en cuenta que el RRF es una estructura independiente, con los campos más habituales son los siguientes:

- Ocupada (O).
- Emitida (E).
- Finalizada (F).
- Dirección (Dir).
- Registro de destino (Rd).
- Registro de renombramiento (Rr).
- Especulativa (Es).
- Validez (V).

En el caso que el RRF fuese parte del buffer de terminación hay que añadir los campos:

- Datos (D)
- Validez de datos (Vdatos)

Y eliminar el campo Rr, ya que la propia entrada haría de registro de renombramiento.

Si unimos los campos *Ocupada*, *Emitida* y *Finalizada* en uno de tres bits denominado Estado, una instrucción pasará por las siguientes situaciones:

Estado = 100: Instrucción en espera de emisión.
Estado = 110: Instrucción en ejecución.
Estado = 111: Instrucción pendiente de terminación.

Para que una instrucción pueda distribuirse son necesarias tres condiciones:

- Una entrada libre en el RRF,
- Una entrada libre en la estación de reserva individual
- Una entrada libre en el buffer de reordenamiento.

Cuando una ***instrucción es terminada arquitectónicamente*** ocurre lo siguiente:

- Su registro de renombramiento asociado Rr se libera.
- El registro de destino Rd se actualiza con el valor de su registro de renombramiento Rr asociado y se libera si no tiene renombramientos pendientes.
- El campo *Ocupado* se fija de nuevo a 0.
- El puntero de cola se incrementa para apuntar a la siguiente instrucción a terminar.

El número de instrucciones que pueden terminarse simultáneamente depende de la capacidad del procesador para transferir información desde el RRF al ARF o desde el buffer de terminación al ARF.
El ancho de banda se determina por el número de puertos de escritura del ARF y la capacidad de enrutamiento de datos hacia el ARF.
Tendremos que tener tantos puertos de escritura en el ARF como instrucciones se quieran terminar por ciclo de reloj.
La velocidad pico del procesador es:

- V_{pico} = Máx. instrucciones terminadas por ciclo * Frecuencia del procesador.

2.9 Retirada

La etapa de retirada es exclusiva de las ***instrucciones de almacenamiento,*** y es donde estas realizan el acceso a memoria para la escritura de sus resultados.
El objetivo de esta etapa es garantizar la consistencia de memoria, es decir, que las instrucciones de almacenamiento se completen en el orden establecido por el programa, debemos evitar los riesgos WAW y WAR.
El mecanismo que se utiliza en esta etapa para lograrlo es el ***buffer de almacenamiento*** o ***buffer de retirada*** que consta de dos partes:

- Finalización.
- Terminación.

El camino que siguen las instrucciones de carga/almacenamiento en una segmentación superescalar una vez que han sido emitidas consta de tres pasos:

- Cálculo de la dirección de memoria.
- Traducción de la dirección de memoria virtual en memoria física.
- Acceso a memoria.

La sintaxis de las instrucciones de carga y almacenamiento que emplean direccionamiento basado en el desplazamiento es:

- **`LD registro_destino, desplazamiento(registro_base)`**
- **`SD desplazamiento (registro_base), registro_fuente.`**

El primer paso en la generación de la dirección de memoria, implica acceder al registro base para leer su valor y sumar este valor de desplazamiento.
Una vez que se dispone de la dirección de memoria hay que realizar la traducción de memoria virtual a memoria principal.
La traducción se realiza consultando la **TLB** (*Translation Lookaside Buffer*), buffer de traducción adelantada, que es donde se mantienen las equivalencias entre direcciones virtuales y direcciones físicas.
Si al realizar el acceso a la TLB, la dirección virtual se encuentra mapeada en la memoria física la TLB devuelve la dirección física y el acceso se realizará con normalidad.
Si la dirección virtual no se encuentra en la memoria principal, se produce un fallo de página que da pie al levantamiento de una *excepción de fallo de página*.
Recuperada la página y el estado del procesador previo a la instrucción que provocó la excepción, la instrucción de carga/almacenamiento puede volver a emitirse sin que se produzca fallo de página.
El paso final de una instrucción de ***carga/almacenamiento*** es la ***lectura/escritura*** en memoria, es en este paso cuando el procesamiento de las instrucciones de carga y almacenamiento difiere.
Las instrucciones de carga realizan los tres pasos en la etapa de ejecución, en cambio las instrucciones de almacenamiento dejan el acceso a memoria para la etapa de retirada.
Las ***instrucciones de almacenamiento*** acceden a la memoria en la *etapa de retirada* para dar mayor prioridad al acceso de las instrucciones de carga a la D-caché.
Cuando finaliza una instrucción de almacenamiento el dato a escribir y la dirección de memoria se guardan en la entrada que se asigna a la instrucción en el buffer de almacenamiento y en esa entrada se marca como finalizada.
La actualización del buffer de almacenamiento se realiza al mismo tiempo que la actualización del buffer de reordenamiento.
Cuando la instrucción es terminada por mandato del buffer de reordenamiento, la instrucción de almacenamiento se marca en el buffer de almacenamiento como terminada, quedando lista para escribir en memoria en cuanto el bus de acceso esté libre.

La ***escritura diferida*** por parte de una instrucción de almacenamiento evita una actualización errónea y precipitada de la memoria en el caso de que una instrucción sufra una interrupción.

La ***escritura diferida*** permite dar mayor prioridad a las instrucciones de carga para acceder a la D-caché.

Una *instrucción de almacenamiento* puede estar marcada como especulativa en el buffer de reordenamiento, en el caso que se produzca una predicción incorrecta la instrucción de almacenamiento se invalida en el buffer de reordenamiento y se purga el buffer de almacenamiento.
Como nunca se pueden terminar las instrucciones especulativas, no existe el problema de que existan instrucciones marcadas como terminadas en el buffer de almacenamiento.
Si están marcadas como terminadas no son especulativas.

Los ***riesgos de memoria WAW*** se evitan mediante el buffer de almacenamiento que asegura que las escrituras en la D-caché se realizan respetando el orden en que aparecen en el programa.
Los ***riesgos de memoria WAR*** se evitan gracias a que una instrucción de almacenamiento posterior a una carga en el orden del programa nunca podrá escribir antes que la carga ya que lo evita la escritura diferida.

2.10 Mejoras en el procesamiento de las instrucciones de carga/almacenamiento

La ***lectura adelantada*** del dato por la carga permite que todas las instrucciones aritmético-lógicas dependientes de ella, directa o indirectamente, tengan el camino libre para su ejecución.
Orto mecanismo que mejora el rendimiento del procesador es el ***reenvío de datos*** desde una instrucción de almacenamiento hacia una de carga que tengan operandos destino y fuente comunes. Se permite que la carga no necesite acceder a la D-caché para su lectura ya que el almacenamiento le reenviará el dato directamente.
Tanto la terminación adelantada de las cargas como el reenvío son dos técnicas que se pueden aplicar simultáneamente.

2.10.1 Reenvío de datos entre instrucciones de almacenamiento y de carga.

El reenvío se puede realizar cuando existe una dependencia de memoria RAW entre una instrucción de almacenamiento (productora) y una instrucción de carga (consumidora).
Para realizar el reenvío es necesario comprobar que las direcciones de memoria de una instrucción de carga y de los almacenamientos pendientes sean coincidentes.
El dato que necesita la carga está en un registro del procesador y no hay que ir a la memoria para obtenerlo.
Los riesgos de datos RAW se resuelven en las estaciones de reserva pero los riesgos de memoria RAW todavía no han sido tratados y son los que hay que analizar para reenviar datos y adelantar cargas.
La coincidencia de direcciones no es posible conocerla cuando las instrucciones se encuentran en la estación de reserva, la comprobación se realiza cuando las instrucciones carga y almacenamiento han sido emitidas y finalizadas/terminadas.
En el proceso de búsqueda de direcciones coincidentes pueden plantearse dos situaciones:

- Hay coincidencia, hay una dependencia de memoria RAW.
- No hay coincidencia, no hay dependencia de memoria RAW, se realiza el acceso a caché de forma normal.

Puede surgir el problema de una aparición de una carga que presenta varias coincidencias en el buffer de almacenamiento, en este caso el procesador debe contar con el hardware adecuado para saber cuál es el almacenamiento más reciente.
Necesitamos mantener los almacenamientos ordenados según su orden de llegada y utilizar codificadores con prioridades para utilizar la dirección más reciente en caso de múltiples coincidencias.
Otra cuestión importante es el número de puertos necesarios en el buffer de almacenamiento para la lectura de datos.
Hasta ahora con un único puerto en el buffer de almacenamiento para la lectura de datos era suficiente.
Para evitar bloqueos es necesario un segundo puerto de lectura para garantizar que en un mismo ciclo se pueda leer y enviar datos a la caché.

2.10.2 **Terminación adelantada de las instrucciones de carga**

Adelantar la ejecución de una instrucción de carga frente a varias instrucciones de almacenamiento es más complicado ya que es necesario comprobar que las direcciones de memoria que manejan no son iguales.
Tenemos que comprobar la ausencia de dependencia de memoria RAW teniendo en cuenta la problemática de las *dependencias ambiguas*.

Vamos a suponer que las instrucciones de carga/almacenamiento se emiten en orden por la estación de reserva a las unidades funcionales de carga y almacenamiento.
La carga, mientras que realiza el acceso a la caché de datos puede comprobar la coincidencia de su dirección en el campo *Dirección* de las entradas del buffer de almacenamiento.
Tenemos dos posibles situaciones:

- Hay coincidencia de direcciones, hay dependencia de memoria RAW. No se puede adelantar la carga, se utiliza el dato del almacenamiento coincidente.
- No hay coincidencia de direcciones, no hay dependencia de memoria RAW, el dato recuperado de la D-caché es válido y se puede almacenar en su registro destino, la carga puede finalizar y terminarse aunque no lo hayan hecho los almacenamientos, se produce un adelantamiento.

La ventaja de la emisión ordenada de las instrucciones de carga/almacenamiento es que reduce la complejidad del hardware, imprescindible para resolver las dependencias ambiguas, pero tiene como inconveniente que reduce el rendimiento, aunque sería deseable que se permitiese el adelantamiento de cargas con emisión fuera de orden.
El empleo de un ***buffer de cargas finalizadas***, es una técnica utilizada para realizar el adelantamiento de cargas permitiendo una emisión desordenada de las cargas y de los almacenamientos.
Esta técnica es una estructura que posibilita que las instrucciones de carga se emitan de forma especulativa sin resolver las dependencias ambiguas que puedan existir con almacenamientos pendientes de ejecución pero previos a la carga en el orden del programa.
Se pueden plantear dos situaciones:

- No hay coincidencia, se permite que la carga continúe normalmente ya que no hay ningún tipo de dependencia de memoria RAW.
- Hay coincidencia, hay dependencia de memoria RAW con un almacenamiento, se anula la carga y todas las instrucciones posteriores para su emisión posterior, no se reenvían los datos ya que pueden quedar almacenamientos anteriores pendientes de ejecución.

Tras finalizar las cargas quedan almacenadas en el buffer de cargas finalizadas, quedan pendientes los almacenamientos que no conocen sus direcciones de acceso a memoria.
Cuando una instrucción de almacenamiento termina tiene que comprobar la existencia de una coincidencia con las cargas que estén en el buffer de cargas finalizadas.
Se pueden plantear dos situaciones:

- Hay coincidencia, la instrucción de carga que depende del almacenamiento se ha ejecutado de forma especulativa. Se ha violado una dependencia de memoria RAW.
- No hay coincidencia, la carga y todas las instrucciones posteriores a ella pueden terminarse ya que no hay ningún tipo de dependencia entre el almacenamiento y la carga.

2.11 Tratamiento de interrupciones

El tratamiento de las interrupciones en un procesador superescalar es más complejo que en un procesador de un único cauce, esto es debido a que la ejecución fuera de orden puede modificar el estado de un proceso en un orden diferente.
Cuando se produce una interrupción, el estado del proceso interrumpido se guarda por el hardware, el software o una combinación de ambos.
El estado del procesador está definido por:

- El contador de programa.
- Los registros arquitectónicos.
- El espacio de memoria asignado.

La precisión de excepción es cuando un procesador es capaz de tratar las interrupciones de forma que se pueda reanudar la ejecución de un programa en las mismas condiciones que había antes de producirse la interrupción.
Es necesario que el procesador mantenga su estado y el de la memoria de forma análoga como si las instrucciones se ejecutasen en orden.
El estado del procesador debe de evolucionar secuencialmente cuando una instrucción termina aunque se haya ejecutado fuera de orden.

Las interrupciones se clasifican en dos grandes grupos:

- ***Interrupciones externas***, se producen por eventos externos a la ejecución del programa.
- ***Excepciones, interrupciones de programa o traps***, suelen producirse en la etapa IF o por las instrucciones desprograma en ejecución.

Las técnicas para el tratamiento de instrucciones que permite la ejecución fuera de orden se pueden agrupar en cuatro categorías:

- Ignorar el problema y no garantizar la precisión de la excepción.
- Permitir que las interrupciones sean algo imprecisas.
- Permitir que una instrucción sea emitida solo cuando se está seguro de que las instrucciones que la preceden terminarán sin causar ninguna interrupción.
- Mantener los resultados de las operaciones en un buffer de almacenamiento temporal hasta que todas las operaciones previas hayan terminado correctamente y se puedan actualizar los registros arquitectónicos sin ningún tipo de riesgo.

Para lograr la precisión de excepción se diseñaron dos técnicas:

- ***El buffer de historia***, es similar al buffer de reordenamiento ya que permite una terminación desordenada de las instrucciones. En sus entradas se almacena el valor de los registros destino al comienzo de la ejecución de las instrucciones (Campo Vp), no al finalizar.
- ***El fichero de futuro***, es una técnica que combina este fichero con el buffer de reordenamiento. Los operandos fuente se leen del fichero de futuro y al finalizar las instrucciones los resultados se escriben en dos ubicaciones, en el fichero de fututo y en el buffer de terminación.

Las técnicas basadas en el buffer de historia y en el registro de futuro proporcionan precisión de excepción retrasando el almacenamiento definitivo de los resultados mientras que las instrucciones no sean terminadas arquitectónicamente y dadas por válidas.

2.11.1 Excepciones precisas con buffer de reordenamiento

El elemento clave que permite mantener la consistencia del procesador es el ***buffer de reordenamiento*** o ***buffer de terminación***, ya que posibilita que las instrucciones no interrumpidas terminen en el mismo orden en que se sitúan en el programa, almacenando sus resultados definitivos de forma ordenada.
Para saber si una instrucción ha sido interrumpida durante la etapa de ejecución, las entradas del buffer de reordenamiento cuentan con un campo *Int*.
Este campo está a 0 y pasa a valer 1 para indicar que ha sido interrumpida la instrucción.
Cuando no es una instrucción del programa durante la ejecución la que lanza la interrupción es posible realizar un tratamiento del problema diferente en función de dónde suceda la interrupción.
Si la interrupción ocurre en la etapa IF por un fallo de página de memoria virtual al intentar leer una instrucción, no se leen nuevas instrucciones y se terminarían todas las instrucciones ya existentes en el cauce.
Si se produce en la etapa de decodificación debido a un código de operación ilegal o indefinido, se terminaría la ejecución de las instrucciones alojadas en el buffer de distribución y siguientes.

2.12 Limitaciones de los procesadores superescalares

Hay dos factores que limitan el rendimiento de los procesadores superescalares:

- El grado de paralelismo intrínseco en el flujo de instrucciones del programa.
- La complejidad y el coste de la etapa de distribución.

En la actualidad se diferencian dos grandes líneas de trabajo en el diseño de procesadores:

- Una enfocada a aumentar el ***paralelismo a nivel de instrucción*** (ILP).
  - VLIW (Very Long Instruction Word).
  - EPIC (Explicitly Parallel Instruction Computing)
- Otra orientada a incrementar el ***paralelismo a nivel de hilo*** (Thread-Level Paralelism – TLP).
- Una tercera sería el paralelismo a nivel de proceso, pero se considera más orientada al desarrollo de multiprocesadores.

La tendencia actual para mejorar las prestaciones de un procesador superescalar pasa por su escalabilidad:

- Aumento del ancho y profundidad de las segmentaciones.
- Aumento del tamaño y niveles de la memoria caché.
- Aumento de la frecuencia de reloj.
- Técnicas avanzadas especulativas que mejoran el flujo de control.
- Otra mejora es el procesamiento vectorial, incluyen dentro del procesador una o varias unidades funcionales para procesar instrucciones SIMD.

Para aumentar el paralelismo a nivel de hilo se trabaja en procesadores:

- **SMT** (*Simultaneous MultiThreading*). Permite la ejecución en paralelo de instrucciones pertenecientes a hilos distintos.
- **CMT** (*Chip Multiprocessors*). Coloca varios procesadores similares en un único circuito integrado.

ÍNDICE

Índice Analítico

CAPÍTULO 3
Procesadores VLIW y procesadores vectoriales

3.3 El concepto arquitectónico VLIW.

Un procesador **VLIW** (***Very Long Instruction Word***) al igual que un procesador superescalar puede emitir y terminar varias operaciones en paralelo.

La diferencia es que es el compilador el encargado de generar el código binario formado por instrucciones que contengan operaciones paralelas.

El hardware se limitará a emitir una instrucción por ciclo de reloj, sin preocuparse de la dependencia de datos, esto implica que el creador del compilador debe saber el paraleleismo que puede proporcionar el hardware.

La diferencia clave entre el enfoque superescalar y el VLIW es cómo se realiza la planificación de las instrucciones.

En una arquitectura superescalar la planificación se realiza vía hardware y se denomina dinámica.

En la VLIW se realiza vía software y se denomina planificación estática. Es estática porque es el compilador el que establece la secuencia paralela de instrucciones con el objeto de no vulnerar la dependencia entre las instrucciones y reducir las detenciones de la segmentación.

En un procesador VLIW se emite una instrucción por ciclo y una detención en una unidad funcional significa detener todas las unidades para mantener la sincronía.

En los VLIW la complejidad del hardware se reduce considerablemente desplazándose hacia el software, esto se traduce en:

- Reducción de transistores.
- Reducción de la energía consumida.
- Reducción del calor disipado.
- Reducción de la inversión económica en horas de ingeniería.
- El desarrollo del compilador se paga una sola vez y no cada vez que se fabrica un chip.
- Se pueden introducir mejoras en el compilador aunque esté en fase de producción.

Pese a todas estas ventajas los procesadores VLIW no triunfaron debido a:

- La incapacidad para desarrollar compiladores que aprovechen al máximo las características de este estilo arquitectónico. VLIW es un ***código de baja densidad***.
- Por los problemas de compatibilidad entre procesadores VLIW.

3.4 Arquitectura de un procesador VLIW genérico.

La principal diferencia con respecto a un procesador superescalar es la ausencia de los elementos necesarios para, la distribución, la emisión y el reordenamiento de instrucciones, o lo que es lo mismo su planificación dinámica.

En un procesador VLIW estos elementos no son necesarios ya que es el compilador quien realiza el trabajo de planificación.

Los mecanismos que utilizan los procesadores superescalares para garantizar el flujo de instrucciones son válidos para un procesador VLIW.

La lógica de decodificación se utiliza para realizar la extracción de las operaciones de la instrucción VLIW y enviarlas a las unidades funcionales junto con los valores de los operandos fuente y el identificador del registro destino.

En las ventanas de emisión quedan preparados los códigos de operación y los valores de los operandos fuente para proceder a su emisión simultánea a las unidades funcionales.

El fichero de registros debe de disponer de los suficientes puertos de lectura para suministrar los operandos a todas las unidades funcionales en un único ciclo de reloj, análogamente sucede lo mismo con el número de puertos necesarios para la escritura de los resultados en el fichero de registros.

Los repertorios de instrucciones de las arquitecturas VLIW siguen una filosofía RISC, pero con mayor tamaño ya que contienen múltiples operaciones o mini-instrucciones.

Una instrucción VLIW equivale a la concatenación de varias instrucciones RISC que se pueden ejecutar en paralelo, son implícitamente paralelas.

Las operaciones recogidas dentro de una instrucción VLIW no presentan dependencias de datos, de memoria y/o control entre ellas.

El tamaño más habitual de una instrucción VLIW es de 256 bits.

El número y tipo de operaciones que contiene una instrucción VLIW se corresponde con el número y tipo de unidades funcionales existentes en el procesador. Si se tienen menos operaciones que unidades funcionales el procesador deja unidades ociosas y no obtiene el máximo rendimiento.

Otra cuestión importante es si las operaciones que forman una instrucción pueden situarse en cualquier posición dentro de la instrucción o tiene posición fija, ya que ésta simplifica al máximo la red de interconexión.

Por la ausencia de hardware, los procesadores VLIW no detienen las unidades funcionales en espera de resultados.

No existen interbloqueos por dependencias de datos, ya que el compilador se encarga de evitarlos.

Para esto se insertan operaciones NOP forzando el avance del contenido del cauce de las unidades funcionales, evitando así la detención.

Uno de los problemas de los procesadores VLIW es la necesidad de que el compilador encuentre instrucciones fuente independientes para poder rellenar todas las operaciones de una instrucción VLIW.

El no poder rellenar completamente una instrucción tiene dos consecuencias:

- El código objeto de un procesador VLIW es de mayor tamaño que un superescalar, aunque quede vacío el slot correspondiente a la operación, el espacio que ocupa la instrucción en memoria no se reduce, se coloca como código de operación NOP.
- No se aprovechan al máximo los recursos del procesador ya que se quedan unidades funcionales ociosas.

Predecir qué accesos a memoria producirán fallos de caché es complicado.

Las memorias caché son ***bloqueantes***, esto quiere decir que pueden detener todas las unidades funcionales ante la aparición de un fallo de caché.

Cuanto mayor es el ancho de instrucción VLIW mayor es el número de referencias a memoria llegando a ser inaceptable las detenciones de las unidades funcionales, esto encarece el rendimiento del procesador.

Se han desarrollado unas técnicas para explotar al máximo el paralelismo de un procesador VLIW, el inconveniente es que aumenta el tamaño del código.

3.5 Planificación estática o basada en el compilador.

Cuando un compilador VLIW recibe como entrada un código fuente realiza una serie de tareas para optimizar el código objeto que produce como salida para aprovechar al máximo el paralelismo del procesador, produciendo tres elementos:

- Un ***código intermedio***.
- Un ***grafo de flujo de control***.
- Un ***grafo de flujo de datos***.

Un ***código intermedio***. Está formado por instrucciones RISC, posee dependencias verdaderas RAW. Las dependencias de salida WAW y antidependencias WAR se eliminan gracias a que el compilador tiene un número infinito de registros simbólicos. Al reasignar los registros simbólicos en registros arquitectónicos aparecen dependencias falsas, el compilador las resuelve al generar el código VLIW planificando las instrucciones que componen el código intermedio. Lo ideal sería encontrar las suficientes operaciones en el código intermedio para formar instrucciones VLIW y maximizar el rendimiento del procesador, de esta manera todas las unidades funcionales trabajarían en paralelo, evitando paradas en la segmentación por dependencias.

Un ***grafo de flujo de control***. Para generarlo debemos de conocer los ***bloques básicos*** del código intermedio. Un *bloque básico* se compone de un grupo de instrucciones que forman una línea de ejecución secuencial, no contiene instrucciones de salto excepto la última, no hay punto intermedios de entrada y salida.

Para obtener los bloques básicos se analiza el código intermedio teniendo en cuenta que:

- Una instrucción etiquetada o la siguiente instrucción a una instrucción de salto establecen el comienzo de un bloque básico o instrucción inicial.
- El bloque básico se compone por todas las instrucciones que hay desde la instrucción inicial hasta la siguiente instrucción de salto que se detecte.
- Los bloques se numeran de forma secuencial, la interconexión de las distintas entradas y salidas de los diferentes bloques básicos son el diagrama de flujo de control.

Cuando ya conocemos los bloques básicos, las instrucciones de cada bloque se combinan para formar instrucciones VLIW.

Esto lo hacemos con el ***grafo de flujo de datos*** que tiene asociado cada bloque.

Un *grafo de flujo de datos* es un grafo dirigido, los *nodos* son las instrucciones de un bloque básico y los *arcos* se inician en una instrucción de escritura en un registro, ***instrucción productora***, y el destino es una instrucción que lee el valor de ese registro, ***instrucción consumidora***.

El *grafo de flujo de datos* nos muestra la secuencia de instrucciones que no presentan dependencias entre ellas y que podemos combinar para formar instrucciones VLIW.

A esta combinación de instrucciones de un bloque básico se le denomina ***planificación local***.

Algunas técnicas de planificación local son el ***desenrollamiento de bucles*** y la ***segmentación software***.

El paralelismo que se obtiene mediante planificación local está limitada, se recurre a la ***planificación global*** con el objeto de combinar instrucciones de diferentes bloques básicos produciendo una planificación con mayor grado de paralelismo. Una técnica es la ***planificación de trazas***.

3.6 Desenrollamiento de bucles (*loop unrolling*).

Es una técnica de planificación local que permite aprovechar el paralelismo existente entre las instrucciones del bucle.

Esta técnica replica muchas veces el cuerpo del bucle utilizando diferentes registros en cada réplica y ajusta el código de replicación en función de las veces que se replique el cuerpo del bucle.

Tiene tres ventajas:

- Se reduce el número de iteraciones del bucle, reduciéndose las dependencias de control al ejecutarse menos instrucciones de salto.
- El total de instrucciones ejecutadas es menor ya que se reduce el número de instrucciones de incremento/decremento de los índices que se utilicen en el bucle.
- Se proporciona al compilador más oportunidades para planificar las instrucciones, el compilador realiza una planificación más efectiva y genera código VLIW más compacto.

En cada ciclo de reloj, la lógica de emisión del procesador emite una instrucción del código VLIW hacia las unidades funcionales, el proceso de emisión no se detiene como consecuencia de una dependencia de datos entre instrucciones, salvo por fallos de acceso a caché.

La emisión continua se consigue por la planificación estática que realiza el compilador y asegura que las dependencias se satisfacen al mantener la separación necesaria en ciclos de reloj entre parejas de operaciones que plantean riesgos.

3.7 Segmentación software.

Es una técnica que se utiliza para intentar aprovechar al máximo el paralelismos a nivel de instrucción que hay en el cuerpo de algunos bucles.

Esta técnica consiste en producir un nuevo cuerpo del bucle compuesto por la intercalación de instrucciones correspondientes a diferentes iteraciones del bucle original.

Al reorganizar el bucle se añaden unas instrucciones de arranque, ***el prólogo***, antes del cuerpo del bucle y otras de terminación al finalizar el bucle, ***el epílogo***.

Una vez que se dispone del nuevo bucle ya reorganizado y dado que las operaciones que lo forman pertenecen a diferentes iteraciones del bucle original y son dependientes, es posible planificarlas y emitirlas en paralelo bajo la forma de instrucciones.

A diferencia del desenrollamiento esta técnica no tiene más instrucciones que el bucle original salvo las que forman el prólogo y el epílogo.

En esta técnica se ejecutan al mismo tiempo instrucciones provenientes de múltiples iteraciones, también se la conoce como ***planificación policíclica***.

Una vez que se aplica la técnica de segmentación software a un bucle, las instrucciones que componen el prólogo, el patrón y el epílogo se utilizan para generar la versión VLIW del bucle original.

Cuando hay instrucciones condicionales en el cuerpo del bucle que impiden la aparición de un patrón de comportamiento regular se complica la aplicación a un procesador VLIW.

Del mismo modo cuando no se conoce a priori el número de iteraciones o cuando el número y tipo de operaciones que forman el patrón de ejecución no se ajusta al formato de la instrucción VLIW.

Tendremos que recurrir a técnicas de planificación que permitan reubicar operaciones de diferentes bloques básicos para intentar ocupar huecos que queden libres en el código VLIW.

Es el objetivo de las técnicas de planificación global, entre las que se encuentran:

- La planificación de hiperbloques.
- La planificación de supebloques.
- La planificación de trazas.

Tras las iteraciones del bucle original aparece un ***patrón*** de ejecución compuesto por instrucciones pertenecientes a iteraciones diferentes del bucle original y que ya no presentan dependencias RAW entre ellas.

Este *patrón* de ejecución o núcleo constituye el cuerpo del nuevo bucle reorganizado.

Las instrucciones que hay antes de la aparición del primer patrón forma el ***prólogo*** del nuevo bucle y las que se encuentran tras el último patrón son el ***epílogo***.

El *prólogo* y el *epílogo* están constituidos por las instrucciones necesarias para completar las iteraciones afectadas por las instrucciones que conforman el patrón de ejecución

3.8 Planificación de trazas.

La planificación de trazas (***trace scheduling***) es una técnica de planificación global que permite tratar una secuencia de bloques básicos como si fuese uno único y con ese nuevo bloque con un mayor número de operaciones producir código VLIW más óptimo mediante una adecuada planificación de las operaciones.

Una ***traza*** es un conjunto de bloques básicos que forman una secuencia de código sin bucles, representan el camino de ejecución más probable ante una bifurcación.

La planificación de trazas es una combinación de dos pasos que se efectúan de forma consecutiva:

- La selección de la traza.
- La compactación o selección de la traza.

La selección de los bloques básicos dentro del grafo de flujo de control se realiza especulando sobre cuales serán las rutas o caminos más probables.

Se utiliza un ***grafo de flujo de control con pesos o ponderado***, este lleva una etiqueta que indica la probabilidad o frecuencia de ejecución.

La reorganización de operaciones dentro de la traza debe de garantizar que se preserva la corrección del programa con independencia de cual sea la ruta más probable.

Cuando se realiza la planificación de trazas colocamos un ***código de compensación*** para deshacer los efectos producidos al ejecutar incorrectamente las operaciones que han sido desplazadas al considerar que su ejecución era altamente probable. Este código se coloca en la ruta de ejecución con menos probabilidad de ser ejecutada y elimina los efectos de las instrucciones que se han ejecutado de forma especulativa.

Las consideraciones que debe realizar el compilador para desplazar operaciones dentro de una traza son las siguientes:

- Conocer cual es la secuencia de ejecución más probable.
- Conocer las dependencias de datos existentes para garantizar su mantenimiento.
- La cantidad de código de compensación que es necesario añadir.
- Saber si compensa el desplazamiento de operaciones dentro de la traza, midiendo el coste tanto en ciclos de ejecución como en espacio de almacenamiento.

Expresión genérica que determina cuándo el código planificado es mejor que la secuencia de código original:

$$a \cdot p + b \cdot (1 - p) < c \cdot p + d \cdot (1 - p)$$

p: frecuencia de ejecución de la rama más probable.
a: ciclos de ejecución de la rama más probable de la secuencia planificada.
b: ciclos de ejecución de la rama menos probable de la secuencia planificada.
c: ciclos de ejecución de la rama más probable de la secuencia original.
d: ciclos de ejecución de la rama menos probable de la secuencia original.

Uno de los problemas que presenta la planificación de trazas es que a mayor cantidad de operaciones desplazadas, mayor es la penalización en que se incurre cuando se realiza una predicción errónea.

La planificación de trazas proporciona mejores resultados cuando la traza seleccionada es correcta, es decir cuando se cumple la predicción que se ha realizado sobre cuál es la ruta de ejecución más probable.

3.9 Operaciones con predicado.

Son instrucciones en la que su resultado se almacena o se descarta dependiendo del valor de un operando que tiene asociado, este operando recibe el nombre de ***predicado***, se implementa como un registro de un bit que se añade como un nuevo operando de lectura en cada una de las operaciones que conforman una instrucción VLIW.

Este registro es la condición que determina el que una operación tenga efecto o no sobre el estado del procesador y de la memoria.

El objetivo de las operaciones con predicado es permitir al compilador reducir el número de saltos condicionales que hay en el código de forma que un diagrama de flujo de control compuesto por diferentes ramas con pequeños bloques básicos pueda transformarse en un único bloque básico extendido y aplicarle técnicas de planificación local.

Este tipo de operaciones condicionadas no se expulsan cuando entran en la segmentación si no se cumple su condición de guarda.

Las operaciones condicionadas siempre se ejecutan, pero si su predicado sigue sin cumplirse al finalizar el procesamiento, el resultado se deshecha, y y resultan equivalentes a las instrucciones NOP.

La técnica ***if-conversión*** es el proceso de eliminar los saltos condicionales de un programa y reemplazarlos con predicados, tiene por objetivo generar un único bloque con instrucciones condicionadas, que vienen de un conjunto de bloques básicos que contienen varias rutas de ejecución.

La representación de una instrucción de código intermedio que tiene asociado un predicado es:

- Instrucción (*p*)

Si *p* vale 1 el resultado de la operación que realice la instrucción se almacenará.
Si *p* vale 0 no se almacena.

La utilización del segundo predicado, ***p2***, es opcional.
Si se utilizan dos predicados ***p1*** y ***p2***, nunca ambos podrán ser true/true, solo está permitido trae/false, false/true y false/false.

Formato para las instrucciones de manipulación de predicados.

```
PRED_CLEAR      p1, p2              % p1:= false
                                    % p2:= false

PRED_EQ         p1, p2, reg, valor  % p1:= (reg ==valor)
                                    % p2:= NOT p1

PRED_NE         p1, p2, reg, valor  % p1:= (reg <>valor)
                                    % p2:= NOT p1

PRED_LT         p1, p2, reg, valor  % p1:= (reg <=valor)
                                    % p2:= NOT p1

PRED_GT         p1, p2, reg, valor  % p1:= (reg >=valor)
                                    % p2:= NOT p1
```

Es posible incluso asignar un predicado a las operaciones de manipulación de predicados de forma que se ejecuten en función de una condición previamente establecida.

Un predicado es un operando origen o destino que está sujeto a las dependencias de datos al igual que sucede con los operandos fuente y destino de cualquier instrucción.

3.10 Tratamiento de excepciones.

Una de las estrategias aplicadas por los procesadores VLIW es la utilización de centinelas, es una técnica análoga al buffer de reordenamiento de los procesadores superescalares.

El ***centinela*** es un fragmento de código que indica que la operación ejecutada de forma especulativa con la que está relacionado ha dejado de serlo.

El *centinela* se ubica en la posición del código en el que se encontraba originalmente la instrucción especulada.

La ejecución del código del centinela es lo que determina que los resultados temporales se pueden escribir en sus ubicaciones definitivas, *fichero de registro o de memoria*.

Para garantizar el tratamiento correcto de las excepciones se utiliza un buffer similar al de terminación en el que las instrucciones especuladas se mantienen marcadas hasta que se llega al punto del programa en que se considera que ya no son instrucciones especulativas.

Mientras permanezcan marcadas en el buffer, sus resultados se almacenan de forma temporal.

3.11 El enfoque EPIC.

El aumento de ***ILP*** (***Paralelismo implícito a nivel de instrucción***) en procesadores VLIW puro presenta una serie de problemas:

- Los repertorios de instrucciones VLIW no son compatibles entre diferentes implementaciones.
- Las instrucciones de carga presentan comportamientos no determinísticos por la propia naturaleza de sistema de memoria.
- La importancia de disponer de un compilador que garantice una planificación óptima del código de forma que se maximice el rendimiento del computador y se minimice el tamaño del código.

Para resolver estos problemas surgió la arquitectura **EPIC** (*Explicitly Parallel Instruction Computing)*

EPIC representa una filosofía para construir procesadores que se apoya en ciertas características arquitectónicas.

El principal objetivo de EPIC es retener la planificación estática del código pero introduciendo mejoras con características arquitectónicas que permitan resolver dinámicamente diferentes situaciones como retardos en las cargas o unidades funcionales nuevas o con diferentes latencias.

El enfoque EPIC sitúa en un punto intermedio entre los procesadores superescalares y los VLIW, el compilador determina el agrupamiento de instrucciones, a la vez comunica de forma explícita en el propio código cómo se ha realizado el agrupamiento.

Las características arquitectónicas más destacadas del estilo EPIC son:

- Planificación estática con paralelismo explícito.
- Operaciones con predicado.
- Descomposición o factorización de las instrucciones de salto condicional.
- Especulación de control.
- Especulación de datos.
- Control de la jerarquía de memoria.

En ciencias de la computación, un ***algoritmo determinista*** es un algoritmo que, en términos informales, es completamente predictivo si se conocen sus entradas. Dicho de otra forma, si se conocen las entradas del algoritmo siempre producirá la misma salida, y la máquina interna pasará por la misma secuencia de estados. Este tipo de algoritmos ha sido el más estudiado durante la historia y por lo tanto resulta ser el tipo más familiar de los algoritmos, así como el más práctico ya que puede ejecutarse en las máquinas eficientemente.

3.12 Procesadores vectoriales.

Son máquinas con una arquitectura que permite la manipulación de vectores.

Las arquitecturas escalares se basan en los procesadores clásicos, segmentados superescalares o los VLIW.

Los procesadores vectoriales cuentan con un repertorio de instrucciones en los que los operadores fuente y destino de las instrucciones son vectores almacenados en registros vectoriales.

Típicas operaciones vectoriales:

- Suma, resta, multiplicación y división de dos vectores
- Suma o multiplicación de todos los elementos de un vector por un escalar.

Existen dos tipos de operandos:

- Los operandos vectoriales.
- Los operandos escalares.

Las operaciones vectoriales pueden tener como fuente dos operandos vectoriales o uno vectorial y otro escalar, pero el resultado será siempre un operando vectorial.

Pueden utilizar unidades funcionales vectoriales con segmentaciones muy profundas sin tener que preocuparse por la existencia de la dependencia de datos, ya que procesan de forma independiente pero continua cada uno de los elementos de que constan los operandos fuente vectoriales.

El compilador genera el código formado por las instrucciones vectoriales a partir de las operaciones que se realizan en el código original.

Una única instrucción vectorial equivale a un bucle completo de instrucciones escalares.

La ventaja de esto es que el ancho de banda de las instrucciones es menor ya que el tamaño del código es más reducido y no hay que extraer tantas instrucciones de la I-caché, por otro lado no existen riesgos de control al eliminarse las instrucciones de salto condicional.

Los factores que determinan que un programa se pueda aprovechar de las características de un procesador vectorial son:

- La estructura del flujo de datos del programa.
- La estructura de flujo de control del programa.
- La capacidad del compilador para detectar las estructuras de datos y de control susceptibles de ser vectorizadas.

3.13 Arquitectura vectorial básica.

Un procesador vectorial básico consta de una unidad de procesamiento escalar y de una unidad de procesamiento vectorial.

Todo programa se compone de una parte de instrucciones escalares y de otra parte de instrucciones vectoriales y cada unidad se ocupa de procesar las instrucciones de su misma naturaleza.

La ***unidad de procesamiento escalar*** está compuesta por el fichero de registros escalares y por las unidades funcionales en las que se ejecutan las instrucciones escalares.

La unidad de procesamiento escalar dispone de unidades segmentadas para realizar operaciones enteras, en coma flotante, saltos, cargas/almacenamientos, etc.

La ***unidad de procesamiento vectorial*** tiene un fichero de registros vectoriales, unidades funcionales vectoriales y una unidad de carga almacenamiento vectorial. También tiene acceso al fichero de registros de la unidad de procesamiento escalar.

El ***fichero de registros vectoriales*** está compuesto por un conjunto de entradas, en donde cada entrada almacena los elementos de que consta un vector. Este fichero es esencial para los algoritmos de vectorización. El número de registros vectoriales oscila entre 64 y 256.

Las especificaciones técnicas de un fichero de registro vectorial incluyen los siguientes parámetros:

- El número de registros vectoriales. **MVL** (Maximum Vector Length).
- El número de elementos por registro. El número de elementos por registro son de 8 a 256.
- El tamaño en bytes de cada elemento, suele ser de 8 bytes. Almacena valores en coma flotante en doble precisión.

Otro parámetro a tener en cuenta es el número de puertos de lectura/escritura necesarios para intercambiar datos con las unidades funcionales vectoriales sin provocar detenciones por riesgos estructurales.

La conexión de todos los puertos de lectura y escritura con las unidades funcionales vectoriales se realiza mediante una red de interconexión total de tipo *crossbar*.

Son necesarios mecanismos hardware que detecten los riesgos estructurales y de datos que puedan aparecer entre las instrucciones vectoriales y escalares del programa.

Las unidades funcionales vectoriales pueden estar a nivel interno organizadas en varios ***lanes*** o ***carriles*** para aumentar las operaciones que pueden procesar en paralelo por ciclo de reloj.

Una unidad funcional vectorial con *n* ***carriles*** implica que el hardware necesario para realizar la operación se ha replicado *n* veces de forma que el número de ciclos que se emplea en procesar un vector se reduce en un factor de *n*.

El problema que tiene utilizar unidades sementadas organizadas en carriles es el incremento del número de puertos de lectura y escritura de todos los registros vectoriales para poder suministrar en cada ciclo de reloj los elementos a los carriles.

La solución que se propone para resolver este problema es especializar los carriles de forma que solo tengan acceso a un subconjunto de elementos de cada registro vectorial.

La unidad funcional de carga/almacenamiento vectorial se ocupa de cargar los registros vectoriales con los datos extraídos de la memoria y de almacenar en ella el contenido de los mismos.

La principal característica es que es capaz de mantener un ancho de banda sostenido de una palabra por ciclo de reloj tras la latencia de acceso inicial.

Dentro de la categoría SIMD se pueden englobar los procesadores vectoriales y matriciales.

Un procesador matricial consta de las siguientes unidades:

- Unidad de procesamiento vectorial.
- Unidad escalar.
- Unidad de control, discrimina según el tipo de instrucción.

La diferencia entre un procesador vectorial y uno matricial está en la organización de la unidad vectorial.

En un procesador matricial la unidad vectorial consta de ***n*** elementos de procesamiento o **EPs**, constituidos por una ALU, un conjunto de registros (**REP**) y una memoria local (**MEP**).

Los EPs se comunican a través de una red de interconexión y pueden funcionar de forma independiente y sincronizada.

Un procesador matricial con *n* elementos de procesamiento puede procesar de forma simultánea en el mismo ciclo de reloj un vector de *n* elementos.

La distribución de los elementos de un vector entre las MEPs es el factor clave para aprovechar al máximo el paralelismo de los EPs.

En condiciones ideales con n EPs se podrían producir n resultados simultáneamente al realizar una operación vectorial y en el peor de los casos todos los resultados podrían generarse por un único EP.

Resumiendo, un vector de m elementos que actúe como operando fuente de una operación vectorial podría estar distribuido entre todas las MEP si $m \leq n$ y en caso contrario habría que distribuir los m elementos del vector de forma cíclica entre las n MEPs para paralelizar la operación correctamente.

3.14 Repertorio genérico de instrucciones vectoriales.

El ***registro de longitud vectorial*** **VLR** (Vector Length Register) controla la longitud de cualquier operación vectorial, ya sean cargas, almacenamientos u operaciones aritméticas.

Si el ***tamaño del vector a procesar es menor*** que el valor del MVL (Maximum Vector Length), se almacena en el registro VLR la longitud del vector.

Si el ***tamaño del vector a procesar es mayor*** que el valor del MVL el compilador recurre a una técnica que se llama ***strip mining*** que es un troceado del vector o seccionamiento.

El *strip mining* consiste en procesar un vector de longitud mayor al MVL en secciones de longitud igual o menor al MVL.

Para esto el compilador genera un bucle con instrucciones escalares y vectoriales que se ocupa de procesar el vector en secciones de longitud MVL, y de longitud menor al MVL si el vector no es múltiplo de MVL. Se utiliza (*n mod* MVL)

Para solucionar el problema de cargar y almacenar datos que se encuentran ubicados en memoria de forma no consecutiva pero que siguen un patrón uniforme se utilizan las siguientes instrucciones:

- LVWS Vi,(Ri,Rj) Carga Vi desde la posición M [Ri] con separación en Rj.
- SVWS (Ri,Rj),Vi Almacena Vi a partir de la posición M [Ri] con separación en Rj.

Para solucionar el problema de vectorizar un bucle en cuyo cuerpo hay instrucciones que se ejecutan condicionalmente recurrimos a una máscara almacenada en el ***registro de máscara*** VM (Vector Mask), que establece qué elementos de un vector tienen que ser manipulados por la unidad funcional al realizarse la operación.

El valor del bit que ocupa la posición *i* en la máscara determina si se realizan o no, *bit a 1 o bit a 0*, las operaciones sobre los elementos que ocupan la posición *i* en los registros vectoriales.

Existen unas instrucciones especiales que gestionan el contenido de ese registro.

S__V Vi,Vj
S__SV Fi,Vi
RVM
MOVF2S VM,Fi
MOVS2F Fi,VM

La mayoría de los procesadores incluyen varios registros de máscara para controlas las operaciones que se realizan sobre los registros vectoriales, pero dependiendo de cómo esté implementado el control de operaciones mediante máscara pueden surgir algunos inconvenientes.

Puede ocurrir que el tiempo de procesamiento no se reduzca pese a que no se realicen las operaciones.

También puede ocurrir que el enmascaramiento afecte únicamente al almacenamiento del resultado pero no evite la operación con lo que se podría dar lugar a la aparición de excepciones.

3.15 Medida del rendimiento de un fragmento de código vectorial.

Tres factores afectan al tiempo de ejecución de una única instrucción vectorial:

- La latencia en producir el primer resultado o ***tiempo de arranque***.
- El número de elementos ***n*** a procesar por la unidad funcional.
- El tiempo que se tarda en completar cada resultado o ***tiempo por elemento***.

Expresión que indica el número de ciclos que se tarda en completar una instrucción vectorial que procesa un vector de ***n*** elementos:

$$T_n = T_{arranque} + n \cdot T_{elemento}$$

El $T_{arranque}$ es similar al número de segmentos de que constan. Si una unidad vectorial está compuesta por 6 etapas y cada etapa consume 1 ciclo, el tiempo de arranque es equivalente a la ***latencia inicial de la unidad*** es decir 6 ciclos.

El $T_{elemento}$ que se tarda en completar cada uno de los restantes *n* resultados es 1 ciclo.

Cuando calculamos el tiempo de ejecución de una secuencia de instrucciones vectoriales es necesario tener en cuenta si las instrucciones afectan o no a diferentes unidades funcionales y si existen dependencias verdaderas entre ellas, es decir, si son operaciones independientes.

Convoy o paquete de instrucciones vectoriales, es un conjunto de instrucciones vectoriales que se pueden emitir simultáneamente debido a que no existen dependencias verdaderas entre ellas y sin riesgos estructurales.

Se puede emitir un convoy cuando todas las instrucciones del convoy anterior han finalizado.

La velocidad de emisión es de 1 convoy por ciclo de reloj.

Por lo general el tiempo por elemento expresado en ciclos es igual al número de convoyes.

Otra medida que permite expresar el rendimiento de un procesador vectorial es el número de operaciones en coma flotante, **FLOP,** realizadas por ciclo de reloj.

$$R_n = \frac{\left(Operaciones\ en\ coma\ flotante \cdot n\ elementos\right)}{T_n}$$

A mayor valor de R_n mejor será el rendimiento obtenido al ejecutar el código vectorial.

El encadenamiento de resultados entre unidades funcionales, ***chaining***, permite mejorar el rendimiento del procesador. Consiste en reenviar el resultado de una unidad funcional a otra sin esperar a que se complete la operación sobre un registro vectorial.

Si se permite encadenamiento dentro de un convoy pueden existir instrucciones dependientes pero nunca deben existir ***riesgos estructurales*** entre ellas.

El encadenamiento permite reducir el tiempo por elemento gracias al solapamiento pero no los tiempos de arranque.

La ***ejecución solapada*** de diferentes convoyes permite que una instrucción vectorial utilice una unidad funcional antes que una operación previa finalice.

El ***solapamiento de convoyes*** complica la lógica de emisión pero permite ocultar los tiempos de arranque para todos los convoyes excepto para el primero.

3.16 La unidad funcional de carga/almacenamiento vectorial.

El elemento hardware más crítico en un procesador vectorial para alcanzar un rendimiento óptimo es la unidad de carga/almacenamiento.

Tiene que tener la capacidad de intercambiar datos con los registros vectoriales de forma sostenida y a velocidad igual o superior a la que las unidades funcionales aritméticas consumen o producen nuevos elementos, es decir, a $\boldsymbol{T_{elemento}}$.

Esta velocidad de transferencia se consigue organizando físicamente la memoria en varios bancos o módulos y distribuyendo el espacio de direccionamiento de forma uniforme entre todos ellos.

Dimensionando de forma adecuada el número de bancos de memoria en función del tiempo de acceso a un banco T_a, se consigue que la latencia para extraer un dato de memoria quede oculta mientras se transfieren los demás elementos que componen un vector a un registro vectorial.

En un procesador vectorial un *dato o elemento*, ***palabra***, suele tener una *longitud de 8 bytes*.

En una ***operación de carga*** de un registro vectorial el ***tiempo de arranque*** es igual a T_a, siendo el tiempo que transcurre desde que se solicita el primer elemento del vector al sistema de memoria hasta que está disponible en el puerto de lectura del banco para transferirlo al registro vectorial.

El ***tiempo por elemento*** se considera como el número de ciclos que se consumen en transferir el dato ya disponible en el banco de memoria el registro vectorial, suele ser inferior a 1 ciclo pero se iguala a 1 para equipararlo al $\boldsymbol{T_{elemento}}$ de las unidades funcionales.

En una ***operación de escritura*** en memoria, el ***tiempo por elemento*** se considera como el tiempo que se emplea en transferir un elemento desde un registro vectorial al puerto de escritura del banco de memoria.

El ***tiempo de arranque*** es el tiempo que emplea el banco en escribir el último elemento del vector en una posición del banco de memoria.

El *tiempo de arranque* y el *tiempo por elemento* se ven de forma opuesta según se trate de una operación de carga o de almacenamiento, para ambos el tiempo que consume una operación de acceso a memoria para un vector de $\boldsymbol{n}$ elementos es similar $T_a + n \cdot \boldsymbol{T_{elemento}}$ ciclos.

Cada vez que se solicita un dato de un vector al sistema de memoria supone un coste de $n \cdot \boldsymbol{T_a}$ ciclos, esto es demasiado si pretendemos mantener un ancho de banda igual a $\boldsymbol{T_{elemento}}$.

Para ocultar esta latencia debemos dimensionar correctamente el número de bancos de memoria de forma que solo se vea el tiempo de acceso correspondiente al primer elemento del vector y que los tiempos de acceso de los demás elementos queden ocultos.

Distribuimos los n elementos de un vector entre los m bancos de memoria para que se solapen los $(n-1) \cdot \boldsymbol{T_a}$ ciclos de acceso de $n - 1$ elementos del vector con los $n \cdot \boldsymbol{T_{elemento}}$ ciclos que se emplean en enviar los n elementos a un registro vectorial.

Esta latencia inicial se considera como el tiempo de arranque de la unidad de carga/almacenamiento.

Un dimensionamiento correcto siempre tiene que cumplir que el número de bancos de memoria m sea mayor o igual que la latencia de acceso al banco de memoria expresada en ciclos de reloj, es decir, que $\boldsymbol{m \geq T_a}$.

Existen dos formas de realizar el solapamiento de las latencias de acceso a los n elementos de un vector:

- De forma síncrona,
- De forma asíncrona,

Se considerará que $\boldsymbol{T_a}$ es igual a m y que se está realizando una operación de carga vectorial.

Acceso síncrono a los bancos de memoria de un procesador vectorial.

Se solicita simultáneamente un dato a los ***m*** bancos cada T_a ciclos.

Una vez que realizamos la primera petición y transcurridos T_a ciclos estarán disponibles los ***m*** primeros elementos del vector para ser transferidos al registro vectorial, consumiendo $m \cdot \boldsymbol{T_{elemento}}$ ciclos.

Cada T_a ciclos se realizan dos acciones:

- Se efectúa una nueva petición simultánea a todos los bancos para extraer los ***m*** elementos siguientes al vector.
- Se comienza a transferir ciclo a ciclo los ***m*** elementos ya disponibles y que fueron solicitados en la petición anterior.

Acceso asíncrono a los bancos de memoria de un procesador vectorial.

Se solicitan los elementos del vector a cada uno de los m bancos de forma periódica con periodo T_a con un desfase entre bancos consecutivos de $\boldsymbol{T_{elemento}}$ ciclos.

En el momento en que un banco tiene ya el dato disponible realiza dos acciones:

- Efectúa una nueva solicitud de dato.
- Transfiere el dato ya disponible al registro vectorial.

Aplicamos el mismo razonamiento para las escrituras de forma inversa:

- Primero se transfieren los elementos.
- Luego se realizan los accesos al banco.

Las palabras que componen un vector deben de estar distribuidas correctamente entre los bancos de memoria para que todo funcione bien.

Los bancos de memoria tienen un ancho de palabra de 8 bytes, se direccionan por bytes y son un número que es potencia de 2.

La distribución de los elementos de un vector en los bancos de memoria se realiza de forma consecutiva y cíclica a partir de una posición de memoria inicial que es múltiplo del ancho de palabra en bytes.

Cuando se va a leer un vector de memoria se analiza la posición de memoria del primer elemento para así conocer el número del banco en que se encuentra, iniciándose la lectura de todo el vector.

Dada una dirección de memoria, el número de banco está determinado por los log_2 (m) bits de orden inferior de la dirección de memoria una vez que se ignoran los bits correspondientes a la longitud de un dato expresada en bytes. Si la separación de los datos es por palabras de k bytes, los log_2 (k) bits de orden inferior son ignorados, y los siguientes log_2 (m) especifican el banco de memoria.

3.17 Medida del rendimiento de un bucle vectorizado.

Los costes de ejecución de las instrucciones vectoriales y escalares obtenidas de la vectorización del bucle se pueden desglosar en cuatro componentes.

T_{base}: es el tiempo que consumen las instrucciones escalares de preparación antes de abordar el bucle que secciona el vector en secciones de longitud igual o inferior a MVL.

T_{bucle}: es el coste de ejecutar las instrucciones escalares necesarias para realizar el seccionamiento en cada iteración del bucle mediante el que se procesa el vector sección a sección.

$T_{arranque}$: es la suma de los tiempos de arranque visibles de las unidades funcionales que se utilizan en los convoyes de instrucciones que se ejecutan en cada iteración del bucle que secciona el vector.

$T_{elemento}$: es el número de convoyes en que se organizan las instrucciones vectoriales que se derivan de vectorizar el bucle.

La expresión que permite determinar el tiempo total de ejecución en ciclos de reloj de un bucle vectorizado que realiza operaciones sobre vectores de longitud *n* es:

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \cdot \left(T_{bucle} + T_{arranque} \right) + n \cdot T_{elemento}$$

$\left\lceil \frac{n}{MVL} \right\rceil$, representa el siguiente valor entero superior a $\left\lceil \frac{n}{MVL} \right\rceil$.

En la expresión de T_n, este operador obtiene el número total de secciones en que se dividen los vectores a procesar con independencia de que haya una sección de longitud inferior a MVL.

El resultado del operador se multiplica por los costes asociados al procesamiento de cada sección, T_{bucle} y $T_{arranque}$.

Para obtener el número total de ciclos hay que añadir el $T_{elemento}$ que consumen los *n* elementos del vector y los costes adicionales que introducen las instrucciones escalares iniciales, es decir T_{base}.

Existe la posibilidad que $T_{elemento}$ sea inferior al número de convoyes cuando se permite el solapamiento de instrucciones vectoriales en las unidades funcionales.

Otra medida del rendimiento de un bucle vectorizado es la velocidad de procesamiento de un bucle de longitud infinita expresada en **FLOP** por ciclo de reloj:

$$R_\infty = \lim_{n \to \infty} \left(\frac{Operaciones\ vectoriales \cdot n}{T_n} \right)$$

Si quisiéramos expresar el rendimiento en **FLOPS** (FLOP por segundo), tendríamos que multiplicar el resultado de la expresión previa por la frecuencia de reloj del procesador vectorial.

Para calcular $T_{elemento}$ en situaciones de solapamiento tenemos:

$$T_{elemento} = \frac{\left(T_n - T_{arranque} \right)}{n}$$

ÍNDICE

Índice Analítico

Capítulo 4. Procesamiento paralelo

4.2 Introducción.

El procesamiento paralelo es el método de organización de las operaciones en un sistema de computación donde más de una operación es realizada de manera concurrente.

La finalidad inicial del procesamiento paralelo es obtener una mayor capacidad de cómputo.

Se intenta dividir el problema en múltiples tareas que se ejecutarán de manera paralela.

Para poder realizar varias tareas de manera simultánea es necesario disponer de múltiples procesadores.

Características de un sistema multiprocesador:

- Debe de estar compuesto por dos o más procesadores.
- Los procesadores deben de compartir el acceso a una memoria común.
- Los procesadores deben compartir acceso a canales de E/S, unidades de control y dispositivos.
- El sistema es controlado por un único sistema operativo.

4.3 Tipos de plataformas de computación paralela.

La estructura de los tipos de plataformas de computación paralela depende de su organización lógica y física:

La ***organización lógica*** se refiere a la visión que el programador tiene de la plataforma, son las capacidades para expresar tareas paralelas (la estructura de control) y los métodos de comunicación entre dichas tareas (el modelo de comunicación).

La ***organización física*** se refiere a la estructura del hardware que compone la plataforma, existen dos modelos para implementar el espacio de direcciones común:

- ***Sistemas de memoria compartida***, un único sistema de memoria física es compartido por todos los procesadores.
- ***Sistemas de memoria distribuida***, cada procesador tiene su propia memoria física, a la que el resto de procesadores no tiene acceso directo.

4.3.1 Organización basada en la estructura de control.

Un ***paradigma*** es una clase de algoritmos que tienen la misma estructura de control.

La elección del paradigma a utilizar depende de la disponibilidad de los recursos computacionales paralelos que se tengan y del tipo de paralelismo inherente al problema.

Los recursos computacionales definen el grado de acoplamiento o nivel de granularidad que puede soportar el sistema de forma eficiente.

El tipo de paralelismo refleja la estructura de la aplicación y de los datos.

El paralelismo debido a la estructura de la aplicación se denomina ***paralelismo funcional***. Las diferentes partes de un programa pueden realizar distintas tareas de una manera concurrente y cooperativa.

Cuando el paralelismo se encuentra en la estructura de datos, permite la ejecución de procesos paralelos con idéntico modo de operación pero sobre distintas partes de los datos, esto es el ***paralelismo estructural*** o ***paralelismo a nivel de datos***.

Clasificación de paradigmas en computación paralela:

- Propiedades del proceso (estructura, topología y ejecución).
- Propiedades de interacción.
- Propiedades de los datos (división y localización).

Clasificación de paradigmas en las aplicaciones paralela:

- Descomposición iterativa.
- Paralelismo algorítmico.
- Descomposición geométrica.
- Descomposición especulativa.
- Descomposición funcional.
- Maestro / Esclavo.
- SPMD (Single Program Multiple Data).
- Descomposición recursiva o divide y vencerás.

4.3.1.1 Paradigma Maestro/Esclavo.

Es el más utilizado en las aplicaciones paralelas y en la mayoría de los casos, consta de dos tipos de entidades, ***un maestro*** y ***varios esclavos***.

El ***maestro*** es el responsable de:

- La descomposición del problema en pequeñas tareas.
- Distribuir estas tareas entre el conjunto de procesadores esclavos.
- Recoger los resultados parciales obtenidos de cada procesador esclavo para ordenarlos y obtener el resultado final del problema.

Los procesadores esclavos reciben un mensaje con la tarea a realizar, realizan la tarea y envían el resultado al maestro.

La comunicación es únicamente entre el maestro y los esclavos.

El paradigma Maestro/Esclavo puede utilizar:

- Un balance de carga estático.
- Un balance de carga dinámico.

Utilizando un ***balance de carga estático***, la distribución de las tareas se realiza al comienzo de la computación, permitiendo al maestro participar en la computación una vez que haya asignado una fracción del trabajo a cada esclavo.

La asignación de tareas se puede realizar de una sola vez o de manera cíclica.

Utilizando un ***balance de carga dinámico***, es útil cuando el número de tareas es mayor que el número de procesadores disponibles o cuando el número de tareas es desconocido al comienzo de la aplicación. Puede alcanzar elevadas aceleraciones computacionales y un interesante grado de escalabilidad.

4.3.1.2 Paradigma SPMD (Single Program Multiple Data).

En este paradigma cada procesador ejecuta básicamente el mismo código pero sobre distintas partes de los datos.

Se dividen los datos de la aplicación entre los procesadores disponibles, en este paradigma la comunicación se establece entre esclavos.

También se denomina paralelismo geométrico, estructural o paralelismo a nivel de datos.

Este paradigma también engloba aquellas aplicaciones donde un algoritmo secuencial se ejecuta simultáneamente en diferentes procesadores pero con diferentes datos de entrada, en estas aplicaciones no hay dependencias entre las tareas de los distintos procesadores esclavos, por lo que no hace falta coordinación entre ambos.

El fallo de un único procesador es suficiente para bloquear la aplicación, ya que ningún procesador podrá avanzar más allá del punto de sincronización global.

4.3.2 Organización basada en el modelo de comunicación.

Existen dos modelos de comunicación de la información entre tareas paralelas:

- El espacio de mensajes único y compartido (memoria compartida)
- El paso de mensajes.

4.3.2.1 Espacio de direcciones compartido.

Los procesadores con memoria compartida comparten físicamente la memoria a través de una red de interconexión, es decir, que todos acceden al mismo espacio de direcciones.

Son fundamentales dos parámetros que caracterizan la velocidad de transferencia entre elementos del sistema paralelo:

- ***Latencia de red***, es el tiempo que se tarda en enviar un mensaje a través de la red de interconexión del sistema paralelo.
- ***Ancho de banda,*** definido como el número de bits que se pueden enviar por unidad de tiempo.

Para garantizar el funcionamiento óptimo es necesario que un ancho de banda elevado, puede darse la situación que en cada ciclo de instrucción cada procesador necesite acceder a la memoria.

Lo ideal sería que la latencia sea baja y el ancho de banda alto, excepto que varios procesadores intenten acceder al medio utilizado para la transmisión de datos simultáneamente.

Por este motivo y algún otro relacionado con el mismo, el número de procesadores en este tipo de sistemas suele ser pequeño y la arquitectura más común para la comunicación es la de bus, en ella todos los procesadores y módulos de memoria se conectan a un único bus.

Esta arquitectura se conoce como **UMA** (*Uniform Memory Access*), ya que el tiempo de acceso a memoria es el mismo para cualquier palabra accedida por un procesador.

Los sistemas de memoria compartida incorporan una memoria caché local en cada procesador, aumentando así el ancho de banda entre el procesador y la memoria local.

También puede existir una memoria caché para la memoria global.

Cuando utilizamos cachés es muy importante asegurar la coherencia de la información en la memoria caché, ya que si un procesador modifica el valor de una variable, los demás tienen que tenerlo en cuenta y actualizarse.

La ***escalabilidad*** es la capacidad del sistema para mejorar la potencia de cálculo cuando el número de componentes del sistema aumenta, aunque limita a la arquitectura UMA.

En el caso ideal la escalabilidad sería lineal, con dos procesadores tendríamos el doble de potencia y así sucesivamente, pero la escalabilidad no suele ser lineal y además llega a saturarse.

Podemos hacer mejoras dotando a cada procesador de una memoria local en la que se almacena el código que se está ejecutando en el procesador y a los datos que sean locales a ese procesador y no los compartan con otros.

Mediante esta estrategia se evitan accesos a memoria a través de la red de interconexión relacionados con búsquedas de código y datos locales, mejorando la eficiencia.

Este tipo de arquitectura se conoce como **NUMA** (*Non Uniform Memory Access*), el tiempo de acceso a memoria no es uniforme dependiendo de si el acceso se realiza a la memoria local del procesador o a alguna otra memoria remota.

Los sistemas NUMA también se conocen como sistemas de memoria compartida distribuida **DSM** *Distributed Shared Memory*.

Es posible manejar un espacio común de direcciones y que sea el programador el encargado de manipular el contenido de esas direcciones de memoria evitando cualquier falta de coherencia.

Las arquitecturas NUMA que incluyen mecanismos hardware dedicados a mantener la coherencia de caché son **ccNUMA** (Caché coherent NUMA) y **COMA** (Caché Only Memory Access).

En los sistemas ccNUMA cada nodo contiene una porción de la memoria total del sistema. Las variables compartidas se reparten por el programador o la máquina, existiendo una copia de cada variable en memoria principal.

Cada nodo puede constar de uno o varios procesadores con sus cachés y una memoria principal.

La coherencia se mantiene de dos formas:

- Mediante el escaneo de las operaciones que realizan los procesadores y que se transmiten por la red (***protocolos snoopy***).
- Manteniendo un registro de las variables en cada procesador.

En los sistemas COMA los procesadores que componen cada nodo no incluye memoria local, únicamente caché.

Se emplea la memoria local de cada nodo del multiprocesador como si fuese una caché del resto del sistema, transfiriendo los datos de cada nodo en función de las necesidades del código en ejecución.

La memoria local de un nodo actúa como caché y el conjunto de la memoria local de los otros nodos sería el equivalente a la memoria principal.

La ventaja de estos sistemas es que tratan los fallos de acceso remoto distribuyendo los datos que está utilizando la aplicación automáticamente por el sistema.

El inconveniente es la complejidad para mantener la coherencia de las copias de las variables a lo largo de todo el sistema.

4.3.2.2. Paso de mensajes.

Consiste en el intercambio de información en forma de mensajes entre los distintos procesadores que componen el sistema.

Tenemos como elementos necesarios para describir el sistema:

- ***Emisor***. Procesador que inicia la comunicación.
- ***Receptor***. Procesador que recibe la comunicación.
- ***Canal de comunicación***. Red de interconexión entre procesadores
- ***Mensaje***. Compuesto por la información enviada.

Existen cuatro operaciones básicas necesarias para establecer un mecanismo de paso de mensajes:

- ***Envío***: para enviar el mensaje.
- ***Recepción***: para recibir el mensaje.
- ***Identificación***: identifica cada uno de los procesadores del sistema con un identificador único.
- ***Número de participantes***: identifica el número de procesadores que participan en la comunicación

Con estas cuatro operaciones se puede escribir cualquier tipo de programa por medio de paso de mensajes.

Se puede emular un sistema de comunicación por paso de mensajes dividiendo el espacio de direcciones en fracciones iguales, cada una asignada a un procesador, las acciones de envío y recepción de mensajes estarán representadas por la escritura y lectura de información en dichas fracciones de memoria.

4.4 Sistemas de memoria compartida.

Cuando se realiza el diseño de un sistema de memoria compartida se deben tener en cuenta tres aspectos básicos:

- La organización de la memoria principal.
- El diseño de la red de interconexión.
- El diseño del protocolo de coherencia de la caché.

4.4.1. Redes de interconexión.

4.4.1.1. Redes estáticas.

También se denomina red directa, su topología queda definida de manera definitiva y estable durante la construcción de la máquina paralela.

Las redes estáticas presentan cuatro topologías:

- Redes unidimensinales.
- Redes bidimensinales.
- Redes tridimensinales.
- Hipervínculos.

Las ***redes unidimensinales***, se conecta cada procesador a dos procesadores vecinos, uno a la derecha y otro a la izquierda.

En la ***red lineal*** todos los procesadores salvo los extremos están enlazados a otros dos procesadores.

La diferencia de las redes lineales con el bus está en el hecho de que en un momento dado puede realizarse más de una transferencia simultáneamente siempre que sea a través de enlaces diferentes. Aunque es simple esta topología presenta problemas de comunicación cuando el número de procesadores es elevado.

En la ***red de anillo*** cada procesador enviará los mensajes por su izquierda o derecha dependiendo cual sea el camino más corto. Los extremos finales están enlazados y permiten notables mejoras en la comunicación.

Una ***estrategia bidimensional*** puede obtenerse a partir de un anillo incrementando el número de enlaces por nodo, disminuyendo los tiempos de transferencia entre los nodos de la red.

Esta topología se denomina de ***anillo cordal***.

Otras dos estrategias bidimensionales son la ***malla*** y la ***red sistólica o array sistólico***.

El esquema bidimensional ideal es la ***red completamente conectada***, aquí cada procesador se comunica directamente con cualquier otro, los mensajes se envían en un solo paso. Equivalen a las redes dinámicas ***crossbar***, son redes no bloqueantes.

La topología de estrella dispone de un procesador central y los demás procesadores establecen un camino de comunicación con él. Es similar a la de bus común.

En la ***redes árbol*** hay un procesador en cada nodo del árbol y solo un camino de conexión entre cualquier par de procesadores.

Las redes lineales y las redes de estrella son casos particulares de la topología de árbol.

El camino de comunicación se realiza de la siguiente forma:

Un procesador envía un mensaje y lo transmite hacia arriba en el árbol hasta que encuentra el procesador destino o llega al nodo raíz del menor subárbol que contiene tanto al procesador origen como al destino, en este caso una vez alcanzado el nodo raíz el mensaje desciende por el árbol hasta encontrar el destino.

Una de las *principales desventajas* de las topologías de red en árbol es que las comunicaciones pueden verse afectadas en un nodo cuando el número de procesadores es grande y se realizan comunicaciones entre procesadores situados en los niveles superiores.

La red de árbol grueso es una estrategia que se utiliza para solventar esta desventaja, consiste en aumentar el número conexiones de comunicación entre los procesadores de menor nivel, los cercanos al nodo raíz. Se conoce como ***red de árbol grueso***.

En las redes ***mesh bidimensionales*** cada procesador se conecta directamente con otros cuatro procesadores salvo en los extremos.

Cuando los procesadores forman una estructura cuadrada con igual número de procesadores en cada dimensión se denomina ***mesh cuadrado***.

Si el número de procesadores es diferente en las dos dimensiones se denomina ***mesh rectangular***.

En la topología ***mesh cerrada o toro*** los procesadores extremos pueden conectarse entre ellos.

En la topología ***mesh tridimensional*** puede establecerse con los procesadores periféricos conectados o no.

Las redes hipercubo son redes mesh multidimensionales con dos procesadores en cada dimensión, un hipercubo de dimensión ***d*** está constituido por $\boldsymbol{p = 2^d}$ procesadores.
Un hipercubo de dimensión *l* se forma conectando los procesadores correspondientes a dos hipercubos de dimensión *l -1*.

Propiedades de los hipercubos:

- Dos procesadores se conectan entre sí cuando sus etiquetas en binario tienen exactamente un bit distinto en una posición determinada.
- Un procesador de un hipercubo de dimensión *d* se conecta directamente a *d* procesadores.
- Todo hipercubo de dimensión *d* puede dividirse en dos de dimensión $d - 1$.
    - Seleccionamos la posición de un bit y se agrupan todos los procesadores que tengan un cero en esa posición, todos estos forman una partición y el resto la segunda partición.
    - Las etiquetas que identifican un procesador en un hipercubo de dimensión *d* constan de *d* bits.
    - Para cualquier grupo de *k* bits fijo, los procesadores que difieren en los demás $d - k$ bits forman un subcubo de dimensión $d - k$ por 2^{d-k} procesadores.
    - Con *k* bits obtenemos 2^k subcubos distintos.

- La distancia Hamming es el número total de posiciones de bits para los que las etiquetas de dos procesadores son diferentes. La distancia Hammin se realiza con la operación or-exclusiva $\oplus$ y es el número de bits que son 1. El número de enlaces por el camino más corto entre dos procesadores viene dado por la distancia de Hamming. Como $a \oplus b$ tiene

como máximo *d* bits que valen 1, el camino más corto entre dos procesadores de un hipercubo consta como máximo de *d* enlaces.

La nomenclatura de las topologías se define como redes *d-cubo k-arias*, *d* es la dimensión de la red y *k* es el número de procesadores en cada dimensión, a *k* se le denomina *radio*.

4.4.1.2. Caracterización de redes estáticas.

Hay cuatro parámetros que caracterizan a una red estática:

- El ***diámetro*** de la red es la máxima distancia entre dos procesadores cualesquiera, la distancia es el mínimo camino entre ellos con el menor número de enlaces, a menor distancia más rápidas las comunicaciones. El diámetro determina el peor de los casos.
- La ***conectividad*** de una red es una medida de la multiplicidad de caminos entre dos procesadores, a mayor tamaño mayores prestaciones, se suele tomar la *conectividad de arco*, que es el menor número de arcos que deben eliminarse para obtener dos redes disjuntas.
- Los ***parámetros*** para establecer la ***rapidez de las comunicaciones*** son:
  - El ***ancho de canal***, número de bits que pueden transmitirse simultáneamente por el canal que comunica dos procesadores, que es igual al número de cables físicos del enlace.
  - La ***velocidad de canal***, es la velocidad máxima con que se puede emitir por cada cable físico.
  - El ***ancho de banda de canal***, velocidad máxima con la que los datos pueden enviarse entre dos enlaces de comunicación, es el producto de la velocidad del canal y el ancho del canal.
- El ***ancho de banda de bisección***, es el menor número de enlaces de comunicación que deben eliminarse para que la red quede dividida en dos partes iguales. Es el menor volumen de comunicaciones permitidas entre dos mitades cualesquiera de la red con igual número de procesadores. Es el producto del ancho de bisección y del ancho de banda del canal. Se suele utilizar como medida de coste ya que establece el menor número de comunicaciones.
- El ***coste***, se cuentan el número de enlaces de comunicación o la cantidad de cableado necesario en la red.

4.4.1.3. Redes dinámicas.

Para diseñar un sistema paralelo de propósito general la mejor opción es una red de interconexión dinámica. Este tipo de redes pueden adaptarse para procesadores del sistema en diferentes ámbitos.

Clasificación de las redes dinámicas:

- Redes basadas en bus.
- Redes crossbar o matriciales.
- Redes multietapa.

Redes basadas en bus.

Es la topología de red más sencilla, todos los nodos comparten un único medio de comunicación que es el bus.

El bus está compuesto por un conjunto de líneas de conexión y conectores a cada procesador.

En un instante de tiempo solo un único procesador puede transmitir información por el bus.

Las colisiones que se producen para el acceso al bus se soluciona usando una lógica de arbitraje o módulo de arbitraje, este módulo se encarga de asignar el acceso al bus para los diferentes procesadores siguiendo una política que se haya determinado.

Las políticas más usadas son:

- **FIFO** (*First In First Out*)
- **Round Robin**
- **LRU** (*Last Recently Used*)

El ancho de banda del bus es limitado y es igual al producto de la frecuencia del reloj por el número de líneas de comunicación existentes. A más procesadores menos rendimiento en la comunicación, este rendimiento puede mejorar colocando una caché en cada procesador ya que la mayoría de accesos de un procesador se realizan a su memoria local. De esta manera sólo se transmiten por el bus los datos remotos.

Son escalables en coste pero no en rendimiento.

Redes crossbar.

Permite conectar ***p*** procesadores con ***q*** elementos de memoria utilizando una matriz de *conmutadores*, se puede utilizar también para conectar procesadores entre sí.

El número de conmutadores para hacer una red ***crossbar*** es *p x q*, donde ***q*** debe ser al menos igual que ***p***.

La complejidad y el coste de la red aumentan en un orden aproximado igual o mayor que $O(p^2)$.

El número de conmutadores y sus conexiones crecen a un ritmo cuadrático según aumenta el número de procesadores, es una gran desventaja y reduce la escalabilidad.

Son redes no bloqueantes.

La latencia de comunicación es constante, la comunicación entre elementos se puede considerar como un bus punto a punto.

Son escalables en rendimiento pero no en coste.

Redes multietapa.

Se componen de una serie de etapas G_i compuestas de conmutadores conectados a las etapas adyacentes mediante conexiones estáticas C_j.

El número de etapas y el tipo de conexiones depende de la implementación concreta de la arquitectura.

Un conmutador *a x b* es un dispositivo con *a* entradas y *b* salidas, *a* y *b* suelen ser potencias de 2. Tienen un *orden = a = b*.

La flexibilidad de las redes multietapa viene de la posibilidad de reconfigurar dinámicamente los modos de conmutación de sus diferentes etapas.

La principal diferencia entre las distintas redes multietapa es la utilización de los distintos tipos de conmutadores y el patrón de conexión entre etapas *G* y *C*.

Son redes bloqueantes, quiere decir que ciertas permutaciones o conexiones a través de la red pueden a su vez bloquear otras conexiones.

Las redes multietapa más comunes son:

- ***Red Omega***, se basa en la utilización de una ***permutación por barajamiento perfecto*** entre sus etapas, las etapas están compuestas por conmutadores binarios. Esta permutación supone un desplazamiento a la izquierda de los bits que representan el número *x*, también existe el ***barajamiento inverso*** que hace un desplazamiento a la derecha. Está compuesta por p nodos de entrada que son procesadores y p nodos de salida que son elementos de memoria. En total tenemos $log_2\ p$ etapas entre ambos. Las conexiones de barajamiento

perfecto entre etapas proporcionan una única conexión entre cada par de componentes. El número de conmutadores necesarios para construir la red es $\frac{p}{2}\log_2 p$, siendo el coste de la red de orden $O(p \log_2 p)$, su coste es menor que una red crossbar. El algoritmo de encaminamiento funciona de la siguiente manera:

- Cada paquete transmitido va precedido de la dirección del nodo destino en binario.
- Los conmutadores de cada etapa deciden el camino por el que transmitir el paquete dependiendo del valor del bit de la dirección de destino correspondiente a la etapa actual. Si el bit es 0 se encamina por la salida superior, si es 1 la inferior.

- ***Red Baseline***, se construye recursivamente por bloques, conectando los conmutadores de la etapa *i* con ambos sub-bloques de la etapa *i + 1*. La primera etapa se construye con un bloque que tendrá un tamaño *n x n*, siendo *n* el número de entradas. La segunda etapa se construye con dos sub-bloques de tamaño $\left(\frac{n}{2}\right)\cdot\left(\frac{n}{2}\right)$, así hasta llegar a la última etapa a $\left(\frac{n}{2}\right)$ sub-bloques de tamaño *2 x 2*. el encaminamiento de paquetes es análogo a la red omega.

- ***Red Butterfly***, se construye con un modelo de permutación básico. De forma genérica las salidas de un conmutador *j* en la etapa *i*, identificado como [*i, j*] se conectarán a los conmutadores [*i+1, j*] e [*i+1, j* $\oplus$ 2^i], difieren en el *i-ésimo* bit. El algoritmo de encaminamiento funciona de la siguiente manera:

  - Siendo A el conmutador conectado al nodo de origen y B el conmutador conectado al nodo destino, los dos representados en binario, calculamos la ruta R = A $\oplus$ B.
  - La ruta entre A y B se define eligiendo el camino directo en el conmutador de la etapa *i* si el bit $R_i = 0$, o el cruzado si el bit $R_i = 1$, siendo R_0 el bit menos significativo.

4.4.2. Protocolos de coherencia de caché.

En sistemas multiprocesador se utilizan memorias caché para mejorar el rendimiento del sistema, reduciendo el tiempo de latencia de la memoria.

En sistemas multiprocesador cada procesador puede necesitar una copia del mismo bloque de memoria y aquí nos surge el problema de consistencia y coherencia entre las distintas copias utilizadas.

Cuando varios procesadores guardan en sus respectivas cachés locales una copia de un mismo bloque de memoria y se modifica se deben comunicar los cambios al resto de procesadores para actualizar las distintas cachés locales.

Este problema se denomina ***problema de coherencia de caché***. Un sistema de memoria es coherente si el valor devuelto por una operación de lectura sobre una dirección de memoria es siempre el mismo valor almacenado por la última operación de escritura realizada sobre esa misma dirección, independientemente de qué procesador realice las operaciones.

La solución es no incorporar un protocolo de coherencia de caché y centrarse en la escalabilidad del sistema, de esta forma solo se permiten dos tipos de accesos a memoria:

- *Local*, el procesador accede a datos privados y puede utilizar una caché local.
- *Remoto*, el procesador accede a datos externos que no se almacenan en la caché.

Los problemas de coherencia de caché están provocados por tres factores:

- Por ***modificar datos compartidos***, cuando dos procesadores acceden a la misma estructura de datos, los datos se encuentran en la memoria principal compartida y en las cachés de cada procesador.
    - Si un procesador modifica su caché local, el contenido de la memoria principal y las copias de las cachés no serán coherentes.
    - Si el sistema utiliza una política de escritura directa, la memoria principal se actualizará pero no las copias del resto de procesadores.
    - Si el sistema utiliza una política de post-escritura, la memoria principal no se actualizará hasta que se reemplace el bloque de caché modificado.

- Por ***migración de procesos***, un proceso puede ser planificado en diferentes procesadores durante su tiempo de vida, si un proceso es intercambiado a otro procesador antes de que las modificaciones realizadas se actualicen en la memoria principal, los datos que cargue el proceso en la caché del nuevo procesador serán incoherentes con las modificaciones realizadas en el anterior procesador.

- Por el uso de ***entrada/salida mediante DMA***, con esta técnica se transportan datos desde los periféricos a la memoria principal del sistema.

    - En la entrada de datos, escritos por el procesador de E/S en memoria principal pueden ser inconsistentes con las copias inconsistentes en las cachés locales de los procesadores del sistema.
    - En la salida de datos, utilizando política de post-escritura se puede generar inconsistencia ya que el procesador de E/S estaría leyendo datos que puede que todavía no hayan sido actualizados desde las cachés locales.

  Una posible solución es configurar los procesadores de E/S para que actúen directamente sobre las cachés locales de los procesadores, así no se generan inconsistencias.
  El inconveniente es la pobre localidad de los datos de E/S en las cachés, incrementando su tasa de fallos.

Existen dos opciones para solucionar los problemas de incoherencia de caché en sistemas multiprocesador:

- ***Invalidar***, consisten en invalidar las copias en las cachés del dato modificado, como inconveniente requiere la espera de acceso al dato causada por la carga del valor correcto de la variable invalidada.
- ***Actualizar***, consiste en actualizar todas las copias en las cachés del dato que se acaba de modificar, como inconveniente provoca un mayor tráfico de datos entre los procesadores, debido a las operaciones de actualización de las cachés.

Un posible método para mantener la coherencia de las diferentes copias de caché podría ser la monitorización del número de copias existentes y el estado de cada copia.

Existen diferentes mecanismos hardware que implementan el protocolo de coherencia de chaché basado en invalidación:

- ***Sistemas snoppy*** o de vigilancia de bus. Cada procesador monitoriza el tráfico de la red en busca de transacciones para poder actualizar el estado de sus bloques de la caché. La caché de cada procesador tiene asociadas unas etiquetas que usa para determinar el estado de sus bloques de caché. Como inconveniente tiene la limitación de ancho de banda del bus. Una posible solución es propagar acciones de coherencia solo a los procesadores afectados, esto se propone en los sistemas basados en directorios.

- ***Sistemas basados en directorios***. Se usa un mapa de bits para almacenar la localización de las copias de caché que tiene cada procesador en su caché. Se puede incluir en memoria principal (con un sistema de directorio centralizado), o en las memorias locales de cada procesador (con un sistema de directorio distribuido), en el caso de sistemas escalables de memoria distribuida.

  Este tipo de sistemas usa el contenido del mapa de bits para propagar las acciones de coherencia del protocolo únicamente a los procesadores que mantienen una copia del bloque afectado.

  En los sistemas de directorio centralizado el principal cuello de botella está en la memoria, porque todas las acciones de coherencia representan accesos a memoria.

  El coste del mapa de bits en memoria crece con orden $O(mp)$, m es el tamaño de la memoria y p el número de procesadores.

  Los sistemas de directorio distribuido permiten p acciones de coherencia simultánea. La latencia y el ancho de banda de la red representan partes fundamentales del rendimiento del sistema.

4.5. Sistemas de memoria distribuida.

Arquitectura de memoria privada o arquitectura de paso de mensajes.

Cada procesador dispone de su propia memoria, se denomina local o privada, es independiente del resto y sólo accesible por su procesador.

La comunicación se realiza por paso de mensajes, es decir, el primer procesador construye un mensaje por software, lo envía a través de una red de interconexión y el segundo lo recibe.

La red de interconexión facilita el paso de mensajes entre los procesadores nodo.

El paso de mensajes se utiliza en sistemas con más de 100 procesadores, además resulta muy útil donde la ejecución de los programas puede dividirse en pequeños subprogramas independientes.

Para analizar el ancho de banda tenemos que tener en cuenta el factor de ***granularidad*** del computador paralelo.

La ***granularidad del computador paralelo*** es el cociente entre el tiempo requerido para realizar una operación básica de comunicación y el tiempo requerido para realizar una operación básica de cálculo de los procesos.

En los sistemas de paso de mensajes hacer uso adecuado del ancho de banda hay que repartir bien los datos sobre los procesadores con el objetivo de disminuir la granularidad, esto es, minimizar el número de mensajes y maximizar su tamaño.

La combinación de un sistema MIMD con arquitectura de paso de mensajes se conoce como ***multicomputadores***.

En los sistemas con memoria distribuida o multiprocesadores existen dos tipos:

- Una computadora con múltiples CPUs comunicadas por un bus de datos. Como ejemplo tenemos los ***procesadores masivamente paralelos MPPs***.
- Múltiples computadoras cada una con su propio procesador enlazados por una red de interconexión. Como ejemplo tenemos los ***clúster***.

Un ***clúster*** es una colección de estaciones de trabajo o PCs interconectados mediante algún sistema de red de comunicaciones.

Podemos clasificar a los *clúster* en dos tipos:

- ***Beowulf***, cada computador del *clúster* está exclusivamente dedicado a él.
- ***NOW***, caso contrario.

Características de los sistemas ***Beowulf***:

- Es un conjunto de nodos minimalistas conectados por un medio de comunicación, la topología de red usada resuelve un tipo de problema específico.
- Cada nodo se dedica exclusivamente a procesos del supercomputador.
- Las comunicaciones se realizan placa a placa por cable RJ-45 cruzado. En un ***NWO*** se usa un switch.
- La programación se realiza por paso de mensajes ya que es fuertemente dependiente de la arquitectura.
- Lo primero que se hace para programarlo es diseñar el modelo de paralelismo, se observan las comunicaciones entre los nodos y se implementan físicamente.

4.5.1. Consideraciones generales sobre los clusters.

Utilizaremos el término *clúster* para referirnos a los *clúster* y *Beowulf*.

La implementación más habitual de los clusters es como parte de una red local de alta velocidad y una máquina que actúa de servidor que además se conecta a la red exterior.

Un factor que afecta significativamente al rendimiento del clúster es el dispositivo de conexión a la red, se suelen utilizar un *hub* o un *switch*, siendo más habitual el *switch*.

Con un switch la transmisión de datos entre dos procesadores solo genera tráfico en el segmento correspondiente, el ancho de banda no se comparte entre todos los procesadores conectados al switch, así cada procesador dispone del 100% del ancho de banda. El switch permite establecer cualquier número de interconexiones simultáneas entre sus puertos siempre y cuando no coincida con el receptor.

4.5.2. ¿Por qué clusters?

El procesamiento paralelo consiste en acelerar la ejecución de un programa mediante su descomposición en fragmentos que puedan ejecutarse de forma paralela, cada uno en su propia unidad de proceso.

Los clusters constituye la alternativa de menor coste que está ampliamente utilizada y consolidada, frente al ***pipelining***, procesamiento superescalar o procesadores vectoriales.

Existe una gran disponibilidad de componentes de alto rendimiento para PCs, estaciones de trabajo y redes de interconexión que hacen posible la alternativa con clusters.

Las principales características son:

- Se pueden construir con un esfuerzo relativamente moderado.
- Son sistemas de bajo coste.
- Utilizan hardware convencional y accesible.
- Utilizan un sistema de comunicación basado en una red de área local rápida como Myrinet o Fast Ethernet.
- Utilizan software de libre distribución como Linux y algún entorno de programación paralelo como *PVM, Parallel Virtual Machine* o *MPI, Message Passing Interface.*
- Son sistemas escalables, se ajustan a las necesidades computacionales.
- Cada máquina de un cluster puede ser un sistema completo utilizable para otros propósitos.
- Reemplazar un computador defectuoso de un cluster es trivial, incluso es posible diseñar el cluster de tal forma que si un nodo falla el resto continúe trabajando.
- El rendimiento y los recursos del cluster pueden crecer con el tiempo beneficiándose de las últimas tecnologías computacionales y de redes.
- Se tiene la garantía que los programas escritos para un cluster funcionarán en cualquier otro con independencia del tipo de procesador.

Las principales inconvenientes son:

- Las redes ordinarias no están diseñadas para el procesamiento paralelo.
- En los sistemas operativos monoprocesador existe muy poco software para tratar un cluster como único sistema.

4.5.3. ¿Cuándo y cómo utilizar un cluster?

La razón de ser del procesamiento paralelo es acelerar la resolución de un problema, la aceleración o *speedup* depende tanto del problema como de la arquitectura del computador paralelo

Las aplicaciones más beneficiadas del *speedup* son las que describen esencialmente procesos paralelos.

Debemos tener en cuenta el hardware de la máquina ya que tenemos que maximizar la relación entre el tiempo de cálculo útil y el perdido en el paso de mensajes, parámetros que dependen de la capacidad de las CPUs y de la velocidad de red de comunicaciones respectivamente.

La clave está en decomponer el problema para que cada procesador pueda operar el mayor tiempo posible sobre su fragmento de los datos sin tener que recurrir a los demás procesadores.

Con *n* procesadores es posible resolver el problema *n* veces más rápido que haciendo uso de uno solo, consiguiendo una aceleración lineal con el número de procesadores.

Se produce ***deceleración*** en la ejecución de un programa sobre un cluster cuando el tiempo de ejecución del programa sobre el cluster se incrementa a medida que se aumenta el número de procesadores del mismo, esto se debe a la mala paralelización del programa, por la gran cantidad de datos compartida entre procesadores.

La aceleración es prácticamente lineal cuando el número de computadores en el cluster es pequeño, pero al ir añadiendo nodos el tiempo de distribución de los datos comienza a ser significativo y la ejecución ya no acelera tanto.

Es interesante utilizar un cluster si la aplicación es suficientemente paralela, ha sido paralelizada o está en disposición de hacerlo.

Existen dos opciones para que una aplicación use los computadores de un clúster:

- Programa explícitamente el paso de mensajes.
- Utilizar las herramientas de ***paralelización automática*** y los ***lenguajes de programación paralelos***.

La mejor opción entre eficiencia, facilidad de uso y portabilidad son:

- Las ***bibliotecas de paso de mensajes*** en las que el programador construye explícitamente el mensaje entre dos procesadores. Tenemos **PVM** (*Parallel Virtual Machine*) y **MPI** (*Message Passing Interface*)
- Las ***de función colectiva*** que son un modelo esencialmente paralelo donde un grupo de procesadores se comunica a la vez. Tenemos **AFAPI** (*Aggregate Function API*)

4.5.4. Programación de clusters.

En aplicaciones de procesamiento paralelo los datos se tienen que intercambiar entre las tareas para resolver cierto problema.

Se utilizan dos paradigmas de programación:

- La ***memoria compartida***. Es similar a un tablón de anuncios, el que tiene permiso escribe y otro que tenga permiso lee. Tiene un problema de control de acceso a la información, se debe de estar seguro que la información que lee un procesador es semánticamente válido y no son datos inválidos. Esto se soluciona mediante semáforos monitores, regiones críticas. Este paradigma solo se emplea si el grado de acoplamiento es alto, quiere decir que los distintos procesos comparten mucha información, y el paso de mensajes resulta poco eficaz.
- El ***paso de mensajes***. Un proceso envía a otro proceso un mensaje que contiene la información que debe conocer, aquí no existe el problema de control de acceso a la información ya que si a un proceso le llega un mensaje es el correcto. El inconveniente es si el grado acoplamiento es alto ya que no resulta nada eficiente.

4.6. Rendimiento y costes en sistemas paralelos.

4.6.1. Factores que influyen en la velocidad computacional.

4.6.1.1. Granularidad de los procesos.

Para lograr una mejora en la velocidad haciendo uso del paralelismo es necesario dividir el cálculo en tareas o procesos que se puedan ejecutar de forma simultánea.

El tamaño de un proceso se puede describir por su granularidad.

Un proceso tiene ***granularidad gruesa*** cuando está compuesto por un gran número de instrucciones secuenciales, son instrucciones que no necesitan comunicación con otros procesos para ser ejecutadas.

Un proceso tiene ***granularidad fina*** cuando está compuesto por pocas instrucciones secuenciales.

Lo ideal es incrementar la granularidad para reducir costes pero esto implica la reducción del número de procesos concurrentes y la cantidad de paralelismo.

En el paso de mensajes se debe reducir la sobrecarga en la comunicación principalmente en los clúster donde la latencia de comunicación puede ser importante.

Si vamos dividiendo en problema en partes paralelas se llega a un punto en que el tiempo de comunicación domina sobre el tiempo total de ejecución, utilizamos la siguiente razón como medida de granularidad:

$$\frac{\text{tiempo de computación}}{\text{tiempo de comunicación}} = \frac{t_{comp}}{t_{com}}$$

Es importante maximizar esta razón, siempre que se mantenga el suficiente paralelismo.

La granularidad está relacionada con el número de procesadores que se utilizan. Puede incrementarse dos formas:

- Aumentando el tamaño del bloque de datos utilizado por un procesador.
- Disminuyendo el número de procesadores si el tamaño del bloque de datos es fijo.

Lo ideal es diseñar programas paralelos que puedan variar su granularidad, es decir conseguir un diseño escalable.

4.6.1.2. Factor de aceleración (*speedup*).

Una medida del rendimiento relativo entre un sistema multiprocesador y un sistema con un único procesador es el factor de aceleración denominado ***speedup***:

$$S(M) = \frac{t_s}{t_p}$$

$\boldsymbol{t_s}$ es el tiempo de ejecución de un programa en un único procesador.

$\boldsymbol{t_p}$ es el tiempo de ejecución en un sistema paralelo con M procesadores.

S(M) da el incremento de velocidad que se obtiene cuando se utiliza un sistema con M procesadores en lugar de uno sólo.

El ***paso o la etapa computacional*** son las operaciones que en su conjunto permiten obtener un resultado parcial completo de la tarea a realizar.

$$S(M) = \frac{\text{Número de pasos computacionales utilizando un procesador}}{\text{Número de pasos computacionales paralelos con M procesadores}}$$

Todos los pasos computacionales que se realizan en paralelo contabilizan solo como uno. La máxima aceleración que se puede lograr con ***M*** procesadores es ***M***, ***aceleración lineal***.

La ***aceleración máxima absoluta*** de ***M*** se debería alcanzar cuando:

- La computación se puede dividir en procesos de igual duración.
- Cada proceso se localiza en un procesador.
- No hay sobrecarga (***overhead***).

$$S(M) = \frac{t_s}{\frac{t_s}{M}} = M$$

Una ***aceleración superlineal***, donde $S(M) > M$, puede encontrarse en alguna ocasión pero no es habitual.

Existen varios factores que aparecen como sobrecarga y que limitan la aceleración:

- Los periodos en los que no todos los procesadores están realizando un trabajo útil.
- Los cálculos adicionales que aparecen en el programa paralelo y que no aparecen en el secuencial.
- El tiempo de comunicación para enviar mensajes.

4.6.1.3. Ley de Amdahl.

La parte secuencial de un programa determina una cota inferior para el tiempo de ejecución, aún cuando se utilicen al máximo las técnicas de paralelismo.

f es la fracción de programa que no se puede dividir en tareas paralelas, $0 \le f \le 1$, y se considera que no hay sobrecarga cuando el programa se divide en tareas paralelas. El tiempo de computación necesario para ejecutar el programa en M procesadores $t_p(M)$ es:

$$t_p(M) = f \cdot t_s + \frac{(1-f) \cdot t_s}{M}$$

La ley de Amdahl expresa que el factor de aceleración viene dado por:

$$S(M) = \frac{t_s}{f \cdot t_s + (1-f) \cdot \frac{t_s}{M}} = \frac{M}{1 + (M-1) \cdot f}$$

El factor de aceleración nunca puede ser mayor que $\frac{1}{f}$.

El límite de Amdahl es $\frac{1}{f}$ siendo f la fracción de código no paralelizable del programa y se define como el mayor factor de aceleración posible cuando el número de procesadores disponibles tiende al infinito.

En los resultados de Amdahl suponemos que no hay sobrecarga, es decir que el tiempo de comunicación no lo tenemos en cuenta.

4.6.1.4. Eficiencia.

Obtenemos la ***eficiencia, E,*** de un sistema si se normaliza la aceleración de un programa paralelo dividiéndola entre el número de procesadores.

$$E = \frac{\text{Tiempo de ejecución utilizando un único procesador}}{\text{Tiempo de ejecución utilizando un multiprocesador} \cdot \text{número de procesadores}} = \frac{t_s}{t_p \cdot M}$$

Teniendo en cuenta esta fórmula podemos expresar la eficiencia E en función del factor de aceleración *S(M)* con la siguiente expresión:

$$E = \frac{S(M)}{M} \cdot 100\% = \frac{1}{1+(M-1)\cdot f} \cdot 100\%$$

La eficiencia da la fracción de tiempo que se utilizan los procesadores durante la computación, es una forma de medir el uso de los procesadores.

Para un mismo programa el aumento de los procesadores del sistema supone un decremento en la eficiencia, ya que se reparte el trabajo realizado por cada uno y el tiempo de uso de cada procesador disminuye.

4.6.1.5. Coste.

El ***coste, C,*** o trabajo de una computación, se define como:

C = Tiempo de ejecución $\cdot$ número de procesadores utilizados

El ***coste de una ejecución paralela*** lo expresamos como:

$$C = \frac{t_s \cdot M}{S(M)} = \frac{t_s}{E}$$

4.6.1.6. Escalabilidad.

La ***escalabilidad hardware*** indica que un diseño hardware permite ampliar su tamaño para obtener una mejora en el rendimiento.

La ***escalabilidad algorítmica*** indica que un algoritmo paralelo puede soportar un incremento grande de datos con un incremento bajo y acotado de pasos computacionales.

La definición más simple de ***escalabilidad*** es que un sistema es escalable si el rendimiento del mismo se incrementa linealmente con relación al número de procesadores usados para una aplicación.

Principales parámetros que afectan a la escalabilidad:

- ***Tamaño del sistema***, número de procesadores utilizados en el sistema.
- ***Frecuencia de reloj***, determina el ciclo de máquina básico.
- ***Tamaño del problema***, cantidad de trabajo computacional necesaria para resolver un determinado problema.
- ***Tiempo de CPU***, tiempo usado en la ejecución de un determinado programa en un sistema con *M* procesadores.
- ***Capacidad de memoria***, cantidad de memoria principal utilizada para la ejecución del programa.
- ***Pérdidas de comunicación (overhead)***, cantidad de tiempo gastada en la comunicación entre procesadores, sincronización, etc.
- ***Coste del sistema***, coste total económico de los recursos hardware y software necesarios para ejecutar un programa.

4.6.1.7. Balance de carga.

Consiste en distribuir de forma equitativa la carga computacional entre todos los procesadores disponibles y con ello conseguir la máxima velocidad de ejecución.

Se pretende evitar que algunos procesadores finalicen antes que otros sus tareas y queden libres debido al reparto no equitativo de trabajo, puede ocurrir que algún procesador sea más rápido que otro o que se den ambas situaciones.

Al ***balance de carga estático*** se le trata antes de la ejecución de cualquier proceso, presenta los siguientes inconvenientes:

- Es difícil hacer una estimación precisa del tiempo de ejecución de todas las partes en las que se divide un programa sin ejecutarlas.
- Algunos sistemas pueden tener retardos en las comunicaciones que pueden variar bajo diferentes circunstancias, lo que dificulta incorporar la variable retardo de comunicación en el balance de carga estático.
- Normalmente los problemas necesitan una serie de pasos computacionales para alcanzar la solución.

El ***balance de carga dinámico*** es el que se puede tratar durante la ejecución, aunque lleva consigo una sobrecarga durante la ejecución del programa es mucho más eficiente que el balance de carga estático.

Dependiendo de dónde y cómo se almacenen y reparten las tareas el balance de carga dinámico se divide en:

- Balance de carga ***dinámico centralizado***, es una estructura Maestro/Esclavo el proceso maestro tiene todas las tareas a realizar que son enviadas a los procesos esclavos, cuando un esclavo finaliza su tarea solicita otra nueva al maestro, esta técnica se denomina también *programación por demanda o bolsa de trabajo* en problemas de distintos tamaños es mejor comenzar por las que tengan mayor tamaño, si todas las tareas son del mismo tamaño e importancia se puede usar una cola FIFO.
- Balance de carga ***dinámico distribuido o descentralizado***, se utilizan varios maestros y cada uno controla a un grupo de esclavos. Tiene como inconveniente que solo puede repartir una tarea cada vez y solo puede responder a nuevas peticiones de una en una. Se pueden producir colisiones si varios esclavos solicitan peticiones a la vez. Esta estructura se recomienda si no hay muchos esclavos y las tareas son intensivas computacionalmente, para tareas de *grano fino o poca carga computacional* y muchos esclavos es mejor distribuir las tareas en más de un sitio.

4.6.2. Costes de la comunicación mediante paso de mensajes.

La ***latencia de comunicación***, es el tiempo empleado en transmitir un mensaje entre dos componentes de un sistema paralelo y es igual a la suma del tiempo necesario para preparar el mensaje y el tiempo que éste emplea en atravesar la red hasta su destino.

Tiene los siguientes parámetros:

- ***Tiempo de inicialización (t_s):*** Incluye el tiempo en preparar el mensaje, el tiempo de ejecución del algoritmo de enrutamiento y el tiempo de conexión entre el emisor y el enrutador.
- ***Tiempo de salto (t_h):*** es el tiempo que tarda la cabecera de un mensaje en viajar entre dos nodos directamente conectados en la red, *latencia de nodo*.
- ***Tiempo de transferencia por palabra (t_w):*** si el ancho de banda de un canal es ***r*** palabras por segundo, su tiempo de transferencia por palabra es *1/r*. Es el tiempo que tarda una palabra en ser transmitida por el canal.

El coste de la comunicación varía dependiendo del tiempo del algoritmo de enrutamiento utilizado:

Almacenamiento y reenvío (store and forward).

Cada nodo intermedio entre el emisor y el receptor reenvía el mensaje únicamente cuando lo ha recibido y almacenado completamente.

$$t_s + \left(m \cdot t_w + t_h\right) \cdot l$$

m es el tamaño del mensaje.
l son los enlaces.
$\boldsymbol{t_h}$ es el tiempo que tarda el mensaje en cada enlace para la cabecera.
$\boldsymbol{m \cdot t_w}$ es el tiempo que tarda para el resto del mensaje.

El tiempo de salto en los sistemas paralelos es mucho menor que el tiempo de transferencia del mensaje $\boldsymbol{m \cdot t_w}$, tendríamos que:

$$t_{com} = t_s + m \cdot t_w \cdot l$$

Corte y continuación (cut through).

Este algoritmo se basa en los sistemas de enrutamiento de paquetes para mejorar los costes de comunicación en sistemas paralelos. Requiere menos memoria y es más rápido que el algoritmo de almacenamiento y reenvío.

Este algoritmo divide cada mensaje en unidades de tamaño fijo llamados ***dígitos de control de flujo (flow control digits o flits).***

Los ***flits*** son más pequeños que los paquetes porque no incluyen información de enrutamiento y la información para la corrección de errores es más sencilla.

Antes de enviar el primer *flit* se envía un paquete llamado *tracer*, cuando se establece la conexión se envían los *flits* uno tras otro siguiendo la misma ruta.

$$t_{com} = t_s + l \cdot t_h + m \cdot t_w$$

m es el tamaño del mensaje.
l son los enlaces.
$\boldsymbol{t_h}$ es el tiempo de salto.

$l \cdot t_h$ es el tiempo que tarda la cabecera del mensaje en llegar al receptor.
$m \cdot t_w$ es el tiempo que tarda el contenido del mensaje en ser transmitido.

Para la comunicación entre nodos cercanos o tamaños de mensaje pequeños los dos algoritmos se comportan de manera aproximada.

4.6.3. Costes de la comunicación mediante memoria compartida.

Para la memoria compartida resulta más complejo encontrar una unidad de medida por los siguientes motivos:

- La disposición de los datos en memoria está determinada por el sistema.
- La limitación en el tamaño de las cachés puede provocar un problema de hiperpaginación (thrashing). Cada fallo de la caché puede provocar operaciones de coherencia de caché y comunicaciones entre procesadores.
- La sobrecarga (*overhead*) incluida por las operaciones de coherencia de la caché (*invalidar y actualizar*) es difícil de cuantificar.
- La localidad espacial es un fenómeno difícil de modelar.
- La lectura anticipada de palabras juega un papel importante en la reducción de los tiempos de acceso a los datos.
- El fenómeno *false sharing* puede incluir una sobrecarga (*overhead*) importante en muchos programas. Se produce este fenómeno cuando diferentes procesadores acceden a diferentes datos que están almacenados en el mismo bloque de caché.
- La competencia por los recursos del sistema normalmente supone una de las mayores sobrecargas en sistemas de memoria compartida.

ÍNDICE

Índice Analítico