VNIVERSITAT DE VALÈNCIA

UNIVERSITAT DE VALÈNCIA

ARQUITECTURAS AVANZADAS

Ingeniería Informática

Profesor: Fernando Pardo Carpio

Valencia, 30 de enero de 2002

Prefacio

El temario recogido en estos apuntes se corresponde con la asignatura de *Arquitecturas Avanzadas* de la Ingeniería Informática de la Universidad de Valencia. Esta asignatura corresponde al plan de 1993 y constaba de 6 créditos teóricos y 1.5 prácticos. Se cursaba en quinto.

Esta asignatura pretendía ser la continuación de la asignatura de *Arquitectura de Computadores* de cuarto curso del plan antiguo. El contenido principal son los sistemas paralelos, es decir, multiprocesadores y multicomputadores, sin olvidar otras arquitecturas paralelas que ya tuvieron su momento en el pasado pero que todavía se siguen utilizando hoy en día.

Estos apuntes toman como base los que se vinieron preparando desde el curso de 1998. A principios del 2000 el profesor Antonio Flores Gil del Departamento de Ingeniería y Tecnología de Computadores de la Facultad de Informática de la Universidad de Murcia, aportó las fuentes de sus mucho más completos apuntes de Arquitectura de Computadores con el fin de que se pudiesen adaptar a las necesidades del temario. Con los apuntes que ya se tenían y con los del profesor Antonio Flores, se ha realizado esta nueva versión que todavía está en construcción.

Se incluyen las referencias bibliográficas utilizadas en la elaboración de los apuntes, incluyéndose un pequeño apéndice donde se comentan los diferentes libros utilizados. Con esto se ha pretendido dar una visión global a las arquitecturas avanzadas, ofreciéndole al alumno un abanico amplio de posibilidades para ampliar información.

Fernando Pardo en Valencia a 8 de octubre de 2001

Índice General

Índice de Figuras

Capítulo 1

Introducción a las arquitecturas paralelas

Hasta este momento se ha estudiado el procesamiento a nivel del procesador. Se ha visto ya que la segmentación es un primer mecanismo de paralelismo, ya que varias instrucciones consecutivas son ejecutadas de forma solapada casi en paralelo. También se vio que los procesadores superescalares realizan también procesamiento paralelo al lanzar dos o más instrucciones al mismo tiempo gracias a la presencia de varios cauces paralelos.

Sin embargo, todos estos sistemas están basados en la arquitectura Von Neumann con un procesador y memoria donde se guardan datos y programa, es decir, una máquina secuencial que procesa datos escalares. Esta arquitectura se ha ido perfeccionando incluyendo el paralelismo de las unidades de control, de cálculo, etc., pero sigue siendo una máquina de ejecución con un único flujo de instrucciones.

No hay una frontera definida entre la arquitectura monoprocesador y las masivamente paralelas. De hecho, las actuales arquitecturas monoprocesador son realmente máquinas paralelas a nivel de instrucción. La evolución de la arquitectura basada en monoprocesador ha venido ligada con la creación de más y mejores supercomputadores que tenían que librarse del concepto de monoprocesador para poder hacer frente a las demandas de computación.

El primer paso hacia la paralelización de las arquitecturas de los computadores, se da con la aparición de los procesadores o sistemas vectoriales. Los procesadores vectoriales extienden el concepto de paralelismo por segmentación al tratamiento de grandes cadenas de datos. El hecho de que los procesadores segmentados hayan venido asociados a los supercomputadores paralelos, los pone en la entrada a lo que son los sistemas paralelos, si bien siguen siendo una extensión del concepto de segmentación.

Por todo esto, el resto de capítulos van a ir dedicados de alguna u otra manera a supercomputadores paralelos. Se empieza por los procesadores vectoriales y se continúan por los sistemas basados en múltiples procesadores, o fuertemente acoplados (multiprocesadores con memoria compartida), o moderadamente acoplados (multiprocesadores con memoria local), o bien débilmente acoplados como los multicomputadores o sistemas distribuidos.

En cuanto a este capítulo se repasarán los conceptos básicos sobre sistemas paralelos, supercomputadores y su clasificación. La bibliografía para este capítulo es muy

amplia ya que en cualquier libro de arquitectura vienen los conceptos básicos sobre arquitecturas paralelas, aunque por ejemplo se recomienda la clasificación de los sistemas en [Zar96], o las introducciones a estos temas paralelos de [Hwa93] o [HP96].

1.1 Clasificación de los sistemas paralelos

1.1.1 Clasificación de Flynn

Probablemente la clasificación más popular de computadores sea la clasificación de Flynn. Esta taxonomía de las arquitecturas está basada en la clasificación atendiendo al flujo de datos e instrucciones en un sistema. Un flujo de instrucciones es el conjunto de instrucciones secuenciales que son ejecutadas por un único procesador, y un flujo de datos es el flujo secuencial de datos requeridos por el flujo de instrucciones. Con estas consideraciones, Flynn clasifica los sistemas en cuatro categorías:

SISD (*Single Instruction stream, Single Data stream*) Flujo único de instrucciones y flujo único de datos. Este el concepto de arquitectura serie de Von Neumann donde, en cualquier momento, sólo se está ejecutando una única instrucción. A menudo a los SISD se les conoce como computadores serie escalares. Todas las máquinas SISD poseen un registro simple que se llama *contador de programa* que asegura la ejecución en serie del programa. Conforme se van leyendo las instrucciones de la memoria, el contador de programa se actualiza para que apunte a la siguiente instrucción a procesar en serie. Prácticamente ningún computador puramente SISD se fabrica hoy en día ya que la mayoría de procesadores modernos incorporan algún grado de paralelización como es la segmentación de instrucciones o la posibilidad de lanzar dos instrucciones a un tiempo (superescalares).

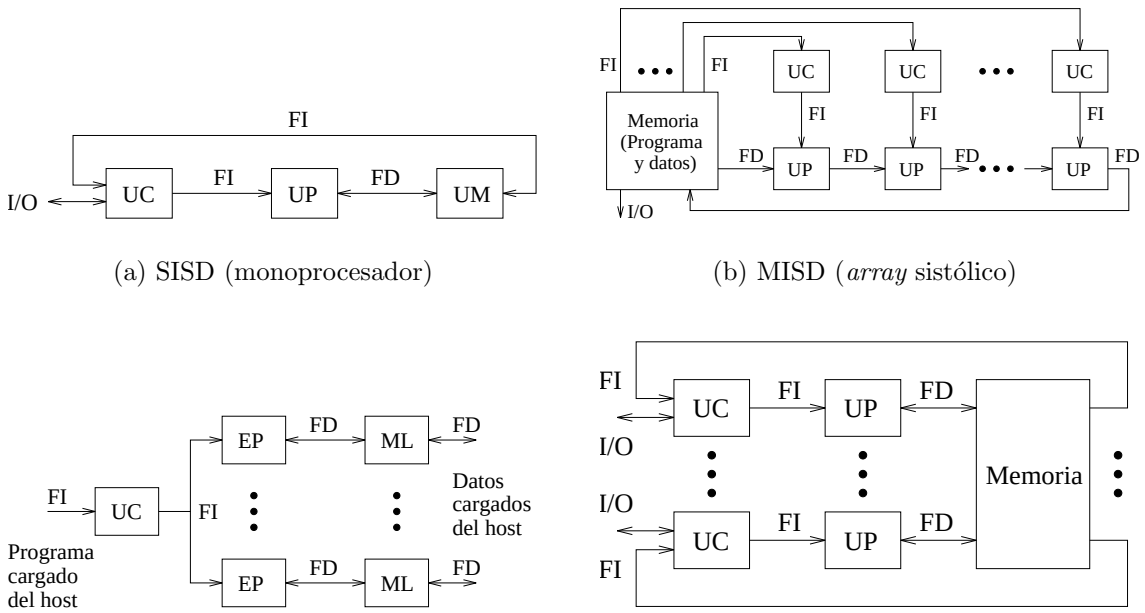
MISD (*Multiple Instruction stream, Single Data stream*) Flujo múltiple de instrucciones y único flujo de datos. Esto significa que varias instrucciones actúan sobre el mismo y único trozo de datos. Este tipo de máquinas se pueden interpretar de dos maneras. Una es considerar la clase de maquinas que requerirían que unidades de procesamiento diferentes recibieran instrucciones distintas operando sobre los mismos datos. Esta clase de arquitectura ha sido clasificada por numerosos arquitectos de computadores como impracticable o imposible, y en estos momentos no existen ejemplos que funcionen siguiendo este modelo. Otra forma de interpretar los MISD es como una clase de máquinas donde un mismo flujo de datos fluye a través de numerosas unidades procesadoras. Arquitecturas altamente segmentadas, como los *arrays sistólicos* o los *procesadores vectoriales*, son clasificados a menudo bajo este tipo de máquinas. Las arquitecturas segmentadas, o encauzadas, realizan el procesamiento vectorial a través de una serie de etapas, cada una ejecutando una función particular produciendo un resultado intermedio. La razón por la cual dichas arquitecturas son clasificadas como MISD es que los elementos de un vector pueden ser considerados como pertenecientes al mismo dato, y todas las etapas del cauce representan múltiples instrucciones que son aplicadas sobre ese vector.

SIMD *Single Instruction stream, Multiple Data stream* Flujo de instrucción simple y flujo de datos múltiple. Esto significa que una única instrucción es aplicada sobre diferentes datos al mismo tiempo. En las máquinas de este tipo, varias unidades de procesado diferentes son invocadas por una única unidad de control. Al igual que las MISD, las SIMD soportan procesamiento vectorial (matricial) asignando

cada elemento del vector a una unidad funcional diferente para procesamiento concurrente. Por ejemplo, el cálculo de la paga para cada trabajador en una empresa, es repetir la misma operación sencilla para cada trabajador; si se dispone de un arquitectura SIMD esto se puede calcular en paralelo para cada trabajador. Por esta facilidad en la paralelización de vectores de datos (los trabajadores formarían un vector) se les llama también *procesadores matriciales*.

MIMD (*Multiple Instruction stream, Multiple Data stream*) Flujo de instrucciones múltiple y flujo de datos múltiple. Son máquinas que poseen varias unidades procesadoras en las cuales se pueden realizar múltiples instrucciones sobre datos diferentes de forma simultánea. Las MIMD son las más complejas, pero son también las que potencialmente ofrecen una mayor eficiencia en la ejecución concurrente o paralela. Aquí la concurrencia implica que no sólo hay varios procesadores operando simultáneamente, sino que además hay varios programas (procesos) ejecutándose también al mismo tiempo.

La figura 1.1 muestra los esquemas de estos cuatro tipos de máquinas clasificadas por los flujos de instrucciones y datos.



(a) SISD (monoprocesador)

(b) MISD (*array* sistólico)

(c) SIMD (procesador matricial)

(d) MIMD (multiprocesador)

Figura 1.1: Clasificación de Flynn de las arquitecturas de computadores. (UC=Unidad de Control, UP=Unidad de Procesamiento, UM=Unidad de Memoria, EP=Elemento de Proceso, ML=Memoria Local, FI=Flujo de Instrucciones, FD=Flujo de datos.)

1.1.2 Otras clasificaciones

La clasificación de Flynn ha demostrado funcionar bastante bien para la tipificación de sistemas, y se ha venido usando desde décadas por la mayoría de los arquitectos de computadores. Sin embargo, los avances en tecnología y diferentes topologías, han llevado a sistemas que no son tan fáciles de clasificar dentro de los 4 tipos de Flynn. Por

ejemplo, los procesadores vectoriales no encajan adecuadamente en esta clasificación, ni tampoco las arquitecturas híbridas. Para solucionar esto se han propuesto otras clasificaciones, donde los tipos SIMD y MIMD de Flynn se suelen conservar, pero que sin duda no han tenido el éxito de la de Flynn.

La figura 1.2 muestra una taxonomía ampliada que incluye alguno de los avances en arquitecturas de computadores en los últimos años. No obstante, tampoco pretende ser una caracterización completa de todas las arquitecturas paralelas existentes.

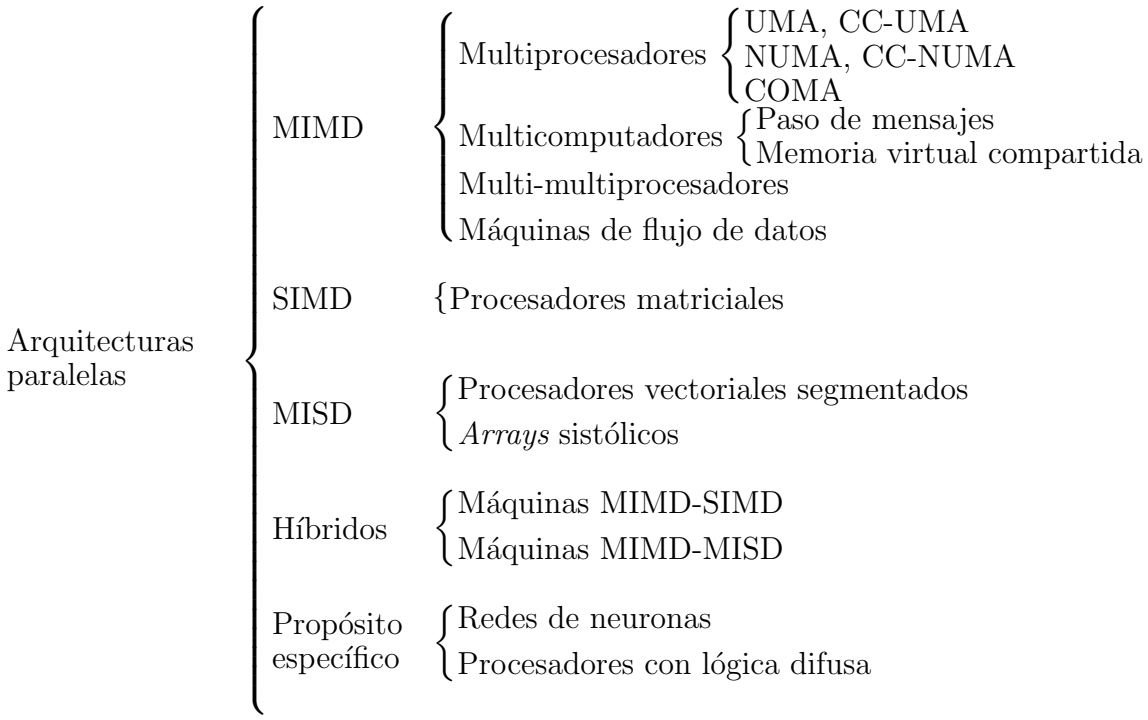


Figura 1.2: Clasificación de las arquitecturas paralelas.

Tal y como se ve en la figura, los de tipo MIMD pueden a su vez ser subdivididos en multiprocesadores, multicomputadores, multi-multiprocesadores y máquinas de flujo de datos. Incluso los multiprocesadores pueden ser subdivididos en NUMA, UMA y COMA según el modelo de memoria compartida. El tipo SIMD quedaría con los procesadores matriciales y el MISD se subdividiría en procesadores vectoriales y en *arrays* sistólicos. Se han añadido dos tipos más que son el híbrido y los de aplicación específica.

Multiprocesadores

Un *multiprocesador* se puede ver como un computador paralelo compuesto por varios procesadores interconectados que pueden compartir un mismo sistema de memoria. Los procesadores se pueden configurar para que ejecute cada uno una parte de un programa o varios programas al mismo tiempo. Un diagrama de bloques de esta arquitectura se muestra en la figura 1.3. Tal y como se muestra en la figura, que corresponde a un tipo particular de multiprocesador que se verá más adelante, un multiprocesador está generalmente formado por n procesadores y m módulos de memoria. A los procesadores los llamamos $P_1, P_2, \ldots, P_n$ y a las memorias $M_1, M_2, \ldots, M_n$. La red de interconexión conecta cada procesador a un subconjunto de los módulos de memoria.

Dado que los multiprocesadores comparten los diferentes módulos de memoria, pudiendo acceder varios procesadores a un mismo módulo, a los multiprocesadores también se les llama *sistemas de memoria compartida*. Dependiendo de la forma en que los procesadores comparten la memoria, podemos hacer una subdivisión de los multiprocesadores:

UMA (*Uniform Memory Access*) En un modelo de *Memoria de Acceso Uniforme*, la memoria física está uniformemente compartida por todos los procesadores. Esto quiere decir que todos los procesadores tienen el mismo tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener su cache privada, y los periféricos son también compartidos de alguna manera.
A estos computadores se les suele llamar *sistemas fuertemente acoplados* dado el alto grado de compartición de los recursos. La red de interconexión toma la forma de bus común, conmutador cruzado, o una red multietapa como se verá en próximos capítulos.
Cuando todos los procesadores tienen el mismo acceso a todos los periféricos, el sistema se llama multiprocesador *simétrico*. En este caso, todos los procesadores tienen la misma capacidad para ejecutar programas, tal como el Kernel o las rutinas de servicio de I/O. En un multiprocesador *asimétrico*, sólo un subconjunto de los procesadores pueden ejecutar programas. A los que pueden, o al que puede ya que muchas veces es sólo uno, se le llama *maestro*. Al resto de procesadores se les llama *procesadores adheridos* (*attached processors*). La figura 1.3 muestra el modelo UMA de un multiprocesador.
Es frecuente encontrar arquitecturas de acceso uniforme que además tienen coherencia de caché, a estos sistemas se les suele llamar **CC-UMA** (*Cache-Coherent Uniform Memory Access*).

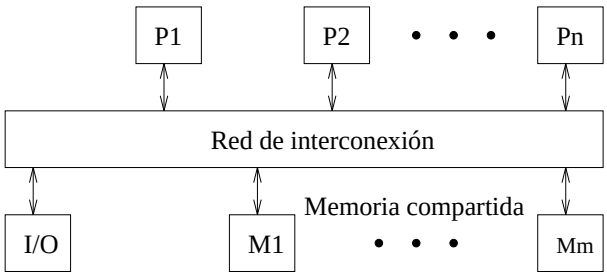


Figura 1.3: El modelo UMA de multiprocesador.

NUMA Un multiprocesador de tipo NUMA es un sistema de memoria compartida donde el tiempo de acceso varía según el lugar donde se encuentre localizado el acceso. La figura 1.4 muestra una posible configuración de tipo NUMA, donde toda la memoria es compartida pero local a cada módulo procesador. Otras posibles configuraciones incluyen los sistemas basados en agrupaciones (*clusters*) de sistemas como el de la figura que se comunican a través de otra red de comunicación que puede incluir una memoria compartida global.
La ventaja de estos sistemas es que el acceso a la memoria local es más rápido que en los UMA aunque un acceso a memoria no local es más lento. Lo que se intenta es que la memoria utilizada por los procesos que ejecuta cada procesador, se encuentre en la memoria de dicho procesador para que los accesos sean lo más locales posible.
Aparte de esto, se puede añadir al sistema una memoria de acceso global. En este caso se dan tres posibles patrones de acceso. El más rápido es el acceso a

memoria local. Le sigue el acceso a memoria global. El más lento es el acceso a la memoria del resto de módulos.
Al igual que hay sistemas de tipo CC-UMA, también existe el modelo de acceso a memoria no uniforme con coherencia de caché **CC-NUMA** (*Cache-Coherent Non-Uniform Memory Access*) que consiste en memoria compartida distribuida y directorios de cache.

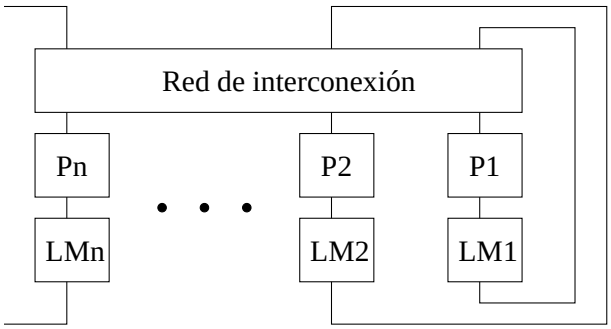


Figura 1.4: El modelo NUMA de multiprocesador.

COMA (*Cache Only Memory Access*) Un multiprocesador que sólo use caché como memoria es considerado de tipo COMA. La figura 1.5 muestra el modelo COMA de multiprocesador. En realidad, el modelo COMA es un caso especial del NUMA donde las memorias distribuidas se convierten en cachés. No hay jerarquía de memoria en cada módulo procesador. Todas las cachés forman un mismo espacio global de direcciones. El acceso a las cachés remotas se realiza a través de los directorios distribuidos de las cachés. Dependiendo de la red de interconexión empleada, se pueden utilizar jerarquías en los directorios para ayudar en la localización de copias de bloques de caché. El emplazamiento inicial de datos no es crítico puesto que el dato acabará estando en el lugar en que se use más.

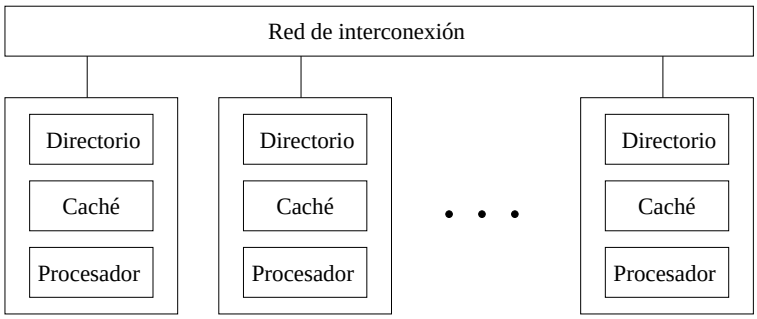


Figura 1.5: El modelo COMA de multiprocesador.

Multicomputadores

Un multicomputador se puede ver como un computador paralelo en el cual cada procesador tiene su propia memoria local. La memoria del sistema se encuentra distribuida entre todos los procesadores y cada procesador sólo puede direccionar su memoria local; para acceder a las memorias de los demás procesadores debe hacerlo por *paso de*

mensajes. Esto significa que un procesador tiene acceso directo sólo a su memoria local, siendo indirecto el acceso al resto de memorias del resto de procesadores. Este acceso local y privado a la memoria es lo que diferencia los multicomputadores de los multiprocesadores.

El diagrama de bloques de un multicomputador coincide con el visto en la figura 1.4 que corresponde a un modelo NUMA de procesador, la diferencia viene dada porque la red de interconexión no permite un acceso directo entre memorias, sino que la comunicación se realiza por paso de mensajes.

La transferencia de datos se realiza a través de la red de interconexión que conecta un subconjunto de procesadores con otro subconjunto. La transferencia de unos procesadores a otros se realiza por tanto por múltiples transferencias entre procesadores conectados dependiendo de cómo esté establecida la red.

Dado que la memoria está distribuida entre los diferentes elementos de proceso, a estos sistemas se les llama *distribuidos* aunque no hay que olvidar que pueden haber sistemas que tengan la memoria distribuida pero compartida y por lo tanto no ser multicomputadores. Además, y dado que se explota mucho la localidad, a estos sistemas se les llama *débilmente acoplados*, ya que los módulos funcionan de forma casi independiente unos de otros.

Multicomputadores con memoria virtual compartida

En un multicomputador, un proceso de usuario puede construir un espacio global de direccionamiento virtual. El acceso a dicho espacio global de direccionamiento se puede realizar por software mediante un paso de mensajes explícito. En las bibliotecas de paso de mensajes hay siempre rutinas que permiten a los procesos aceptar mensajes de otros procesos, con lo que cada proceso puede servir datos de su espacio virtual a otros procesos. Una lectura se realiza mediante el envío de una petición al proceso que contiene el objeto. La petición por medio del paso de mensajes puede quedar oculta al usuario, ya que puede haber sido generada por el compilador que tradujo el código de acceso a una variable compartida.

De esta manera el usuario se encuentra programando un sistema aparentemente basado en memoria compartida cuando en realidad se trata de un sistema basado en el paso de mensajes. A este tipo de sistemas se les llama multicomputadores con memoria virtual compartida.

Otra forma de tener un espacio de memoria virtual compartido es mediante el uso de páginas. En estos sistemas una colección de procesos tienen una región de direcciones compartidas pero, para cada proceso, sólo las páginas que son locales son accesibles de forma directa. Si se produce un acceso a una página remota, entonces se genera un fallo de página y el sistema operativo inicia una secuencia de pasos de mensaje para transferir la página y ponerla en el espacio de direcciones del usuario.

Máquinas de flujo de datos

Hay dos formas de procesar la información, una es mediante la ejecución en serie de una lista de comandos y la otra es la ejecución de un comando demandado por los datos disponibles. La primera forma empezó con la arquitectura de Von Neumann donde un programa almacenaba las órdenes a ejecutar, sucesivas modificaciones, etc., han

convertido esta sencilla arquitectura en los multiprocesadores para permitir paralelismo.

La segunda forma de ver el procesamiento de datos quizá es algo menos directa, pero desde el punto de vista de la paralelización resulta mucho más interesante puesto que las instrucciones se ejecutan en el momento tienen los datos necesarios para ello, y naturalmente se debería poder ejecutar todas las instrucciones demandadas en un mismo tiempo. Hay algunos lenguajes que se adaptan a este tipo de arquitectura comandada por datos como son el Prolog, el ADA, etc., es decir, lenguajes que exploten de una u otra manera la concurrencia de instrucciones.

En una arquitectura de flujo de datos una instrucción está lista para su ejecución cuando los datos que necesita están disponibles. La disponibilidad de los datos se consigue por la canalización de los resultados de las instrucciones ejecutadas con anterioridad a los operandos de las instrucciones que esperan. Esta canalización forma un flujo de datos que van disparando las instrucciones a ejecutar. Por esto se evita la ejecución de instrucciones basada en *contador de programa* que es la base de la arquitectura Von Neumann.

Las instrucciones en un flujo de datos son puramente autocontenidas; es decir, no direccionan variables en una memoria compartida global, sino que llevan los valores de las variables en ellas mismas. En una maquina de este tipo, la ejecución de una instrucción no afecta a otras que estén listas para su ejecución. De esta manera, varias instrucciones pueden ser ejecutadas simultáneamente lo que lleva a la posibilidad de un alto grado de concurrencia y paralelización.

La figura 1.6 muestra el diagrama de bloques de una máquina de flujo de datos. Las instrucciones, junto con sus operandos, se encuentran almacenados en la memoria de datos e instrucciones (D/I). Cuando una instrucción está lista para ser ejecutada, se envía a uno de los elementos de proceso (EP) a través de la red de arbitraje. Cada EP es un procesador simple con memoria local limitada. El EP, después de procesar la instrucción, envía el resultado a su destino a través de la red de distribución.

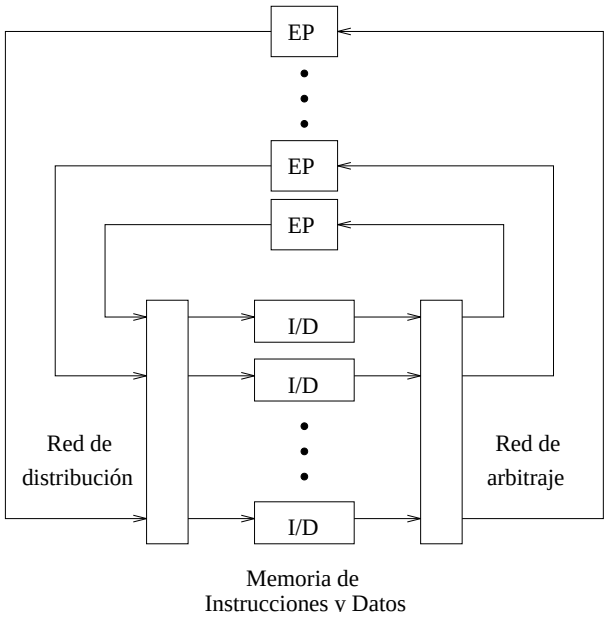


Figura 1.6: Diagrama de bloques de una máquina de flujo de datos.

Procesadores matriciales

Esta arquitectura es la representativa del tipo SIMD, es decir, hay una sola instrucción que opera concurrentemente sobre múltiples datos.

Un procesador matricial consiste en un conjunto de elementos de proceso y un procesador escalar que operan bajo una unidad de control. La unidad de control busca y decodifica las instrucciones de la memoria central y las manda bien al procesador escalar o bien a los nodos procesadores dependiendo del tipo de instrucción. La instrucción que ejecutan los nodos procesadores es la misma simultáneamente, los datos serán los de cada memoria de procesador y por tanto serán diferentes. Por todo esto, un procesador matricial sólo requiere un único programa para controlar todas las unidades de proceso.

La idea de utilización de los procesadores matriciales es explotar el paralelismo en los datos de un problema más que paralelizar la secuencia de ejecución de las instrucciones. El problema se paraleliza dividiendo los datos en particiones sobre las que se pueden realizar las mismas operaciones. Un tipo de datos altamente particionable es el formado por vectores y matrices, por eso a estos procesadores se les llama matriciales.

Procesadores vectoriales

Un procesador vectorial ejecuta de forma segmentada instrucciones sobre vectores. La diferencia con los matriciales es que mientras los matriciales son comandados por las instrucciones, los vectoriales son comandados por flujos de datos continuos. A este tipo se le considera MISD puesto que varias instrucciones son ejecutadas sobre un mismo dato (el vector), si bien es una consideración algo confusa aunque aceptada de forma mayoritaria.

Arrays sistólicos

Otro tipo de máquinas que se suelen considerar MISD son los *arrays* sistólicos. En un *array* sistólico hay un gran número de elementos de proceso (EPs) idénticos con una limitada memoria local. Los EPs están colocados en forma de matriz (*array*) de manera que sólo están permitidas las conexiones con los EPs vecinos. Por lo tanto, todos los procesadores se encuentran organizados en una estructura segmentada de forma lineal o matricial. Los datos fluyen de unos EPs a sus vecinos a cada ciclo de reloj, y durante ese ciclo de reloj, o varios, los elementos de proceso realizan una operación sencilla. El adjetivo *sistólico* viene precisamente del hecho de que todos los procesadores vienen sincronizados por un único reloj que hace de "corazón" que hace moverse a la máquina.

Arquitecturas híbridas

Hemos visto dos formas de explotar el paralelismo. Por un lado estaba la paralelización de código que se consigue con las máquinas de tipo MIMD, y por otro lado estaba la paralelización de los datos conseguida con arquitecturas SIMD y MISD. En la práctica, el mayor beneficio en paralelismo viene de la paralelización de los datos. Esto es debido a que el paralelismo de los datos explota el paralelismo en proporción a la cantidad de los datos que forman el cálculo a realizar. Sin embargo, muchas veces resulta imposible explotar el paralelismo inherente en los datos del problema y se hace necesario utilizar

tanto el paralelismo de control como el de datos. Por lo tanto, procesadores que tienen características de MIMD y SIMD (o MISD) a un tiempo, pueden resolver de forma efectiva un elevado rango de problemas.

Arquitecturas específicas

Las arquitecturas específicas son muchas veces conocidas también con el nombre de *arquitecturas VLSI* ya que muchas veces llevan consigo la elaboración de circuitos específicos con una alta escala de integración.

Un ejemplo de arquitectura de propósito específico son las redes neuronales (ANN de *Artificial Neural Network*). Las ANN consisten en un elevado número de elementos de proceso muy simples que operan en paralelo. Estas arquitecturas se pueden utilizar para resolver el tipo de problemas que a un humano le resultan fáciles y a una máquina tan difíciles, como el reconocimiento de patrones, comprensión del lenguaje, etc. La diferencia con las arquitecturas clásicas es la forma en que se programa; mientras en una arquitectura Von Neumann se aplica un programa o algoritmo para resolver un problema, una red de neuronas aprende a fuerza de aplicarle patrones de comportamiento.

La idea es la misma que en el cerebro humano. Cada elemento de proceso es como una neurona con numerosas entradas provenientes de otros elementos de proceso y una única salida que va a otras neuronas o a la salida del sistema. Dependiendo de los estímulos recibidos por las entradas a la neurona la salida se activará o no dependiendo de una función de activación. Este esquema permite dos cosas, por un lado que la red realice una determinada función según el umbral de activación interno de cada neurona, y por otro, va a permitir que pueda programarse la red mediante la técnica de ensayo-error.

Otro ejemplo de dispositivo de uso específico son los procesadores basados en *lógica difusa*. Estos procesadores tienen que ver con los principios del razonamiento aproximado. La lógica difusa intenta tratar con la complejidad de los procesos humanos eludiendo los inconvenientes asociados a lógica de dos valores clásica.

1.2 Fuentes del paralelismo

El procesamiento paralelo tiene como principal objetivo explotar el paralelismo inherente a las aplicaciones informáticas. Todas las aplicaciones no presentan el mismo perfil cara al paralelismo: unas se pueden paralelizar mucho y en cambio otras muy poco. Al lado de este factor cuantitativo evidente, es necesario considerar también un factor cualitativo: la manera a través de la cual se explota el paralelismo. Cada técnica de explotación del paralelismo se denomina fuente. Distinguiremos tres fuentes principales:

- El paralelismo de control
- El paralelismo de datos
- El paralelismo de flujo

1.2.1 El paralelismo de control

La explotación del paralelismo de control proviene de la constatación natural de que en una aplicación existen acciones que podemos "hacer al mismo tiempo". Las acciones, llamadas también tareas o procesos pueden ejecutarse de manera más o menos independiente sobre unos recursos de cálculo llamados también procesadores elementales (o PE). La figura 1.7 muestra el concepto que subyace tras el paralelismo de control

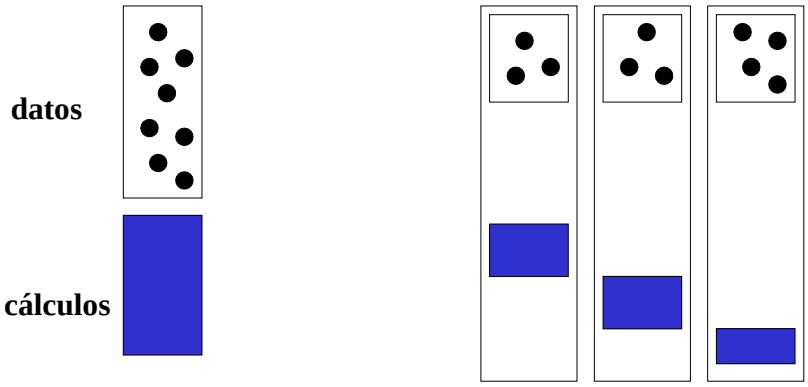


Figura 1.7: Paralelismo de control.

En el caso de que todas las acciones sean independientes es suficiente asociar un recurso de cálculo a cada una de ellas para obtener una ganancia en tiempo de ejecución que será lineal: N acciones independientes se ejecutarán N veces más rápido sobre N Elementos de Proceso (PE) que sobre uno solo. Este es el caso ideal, pero las acciones de un programa real suelen presentar dependencias entre ellas. Distinguiremos dos clases de dependencias que suponen una sobrecarga de trabajo:

- Dependencia de control de secuencia: corresponde a la secuenciación en un algoritmo clásico.
- Dependencia de control de comunicación: una acción envía informaciones a otra acción.

La explotación del paralelismo de control consiste en administrar las dependencias entre las acciones de un programa para obtener así una asignación de recursos de cálculo lo más eficaz posible, minimizando estas dependencias. La figura 1.8 muestra un ejemplo de paralelismo de control aplicado a la ejecución simultánea de instrucciones.

1.2.2 El paralelismo de datos

La explotación del paralelismo de datos proviene de la constatación natural de que ciertas aplicaciones trabajan con estructuras de datos muy regulares (vectores, matrices) repitiendo una misma acción sobre cada elemento de la estructura. Los recursos de cálculo se asocian entonces a los datos. A menudo existe un gran número (millares o incluso millones) de datos idénticos. Si el número de PE es inferior al de datos, éstos se reparten en los PE disponibles. La figura 1.9 muestra de forma gráfica el concepto de paralelismo de datos.

Como las acciones efectuadas en paralelo sobre los PE son idénticas, es posible centralizar el control. Siendo los datos similares, la acción a repetir tomará el mismo

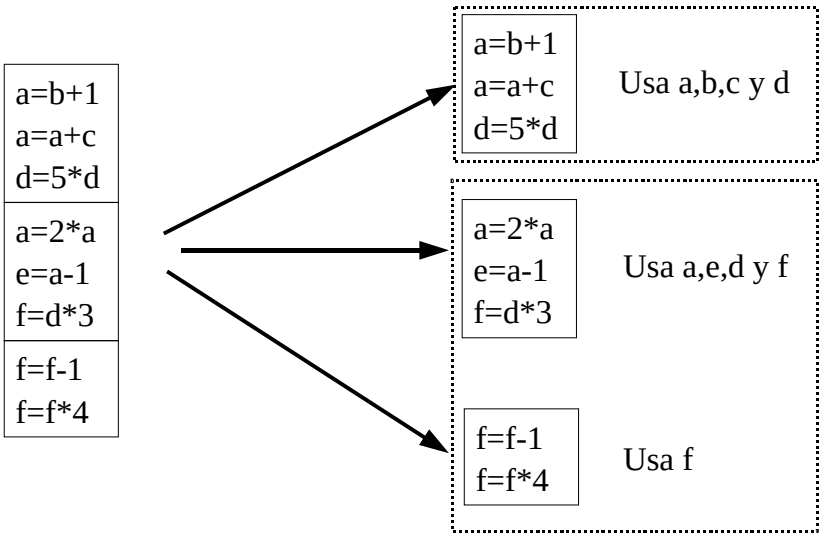


Figura 1.8: Ejemplo de paralelismo de control.

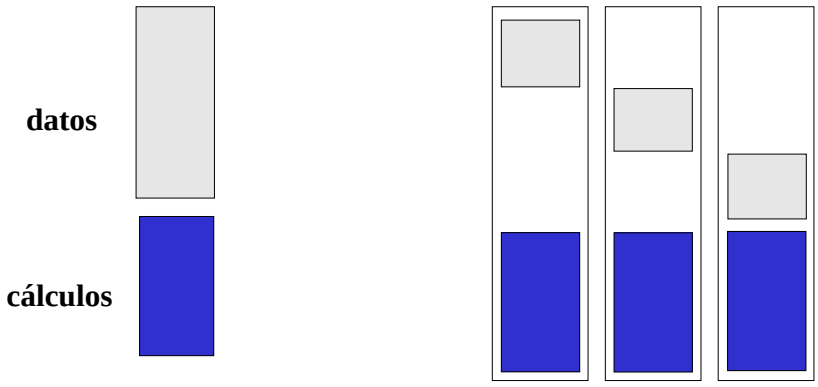


Figura 1.9: Paralelismo de datos.

tiempo sobre todos los PE y el controlador podrá enviar, de manera síncrona, la acción a ejecutar a todos los PE.

Las limitaciones de este tipo de paralelismo vienen dadas por la necesidad de dividir los datos vectoriales para adecuarlos al tamaño soportado por la máquina, la existencia de datos escalares que limitan el rendimiento y la existencia de operaciones de difusión (un escalar se reproduce varias veces convirtiéndose en un vector) y β-reducciones que no son puramente paralelas. En la figura 1.10 se muestra un ejemplo de paralelismo de datos.

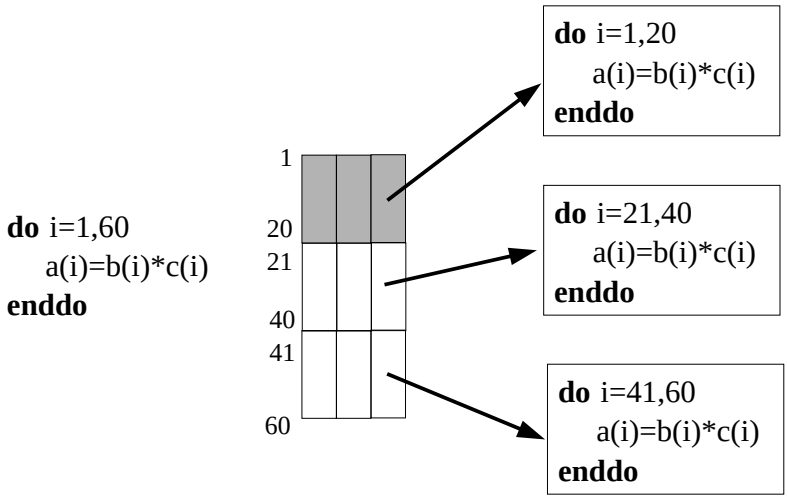


Figura 1.10: Ejemplo de la aplicación del paralelismo de datos a un bucle.

1.2.3 El paralelismo de flujo

La explotación del paralelismo de flujo proviene de la constatación natural de que ciertas aplicaciones funcionan en modo cadena: disponemos de un flujo de datos, generalmente semejantes, sobre los que debemos efectuar una sucesión de operaciones en cascada. La figura 1.11 muestra de forma gráfica el concepto de paralelismo de flujo.

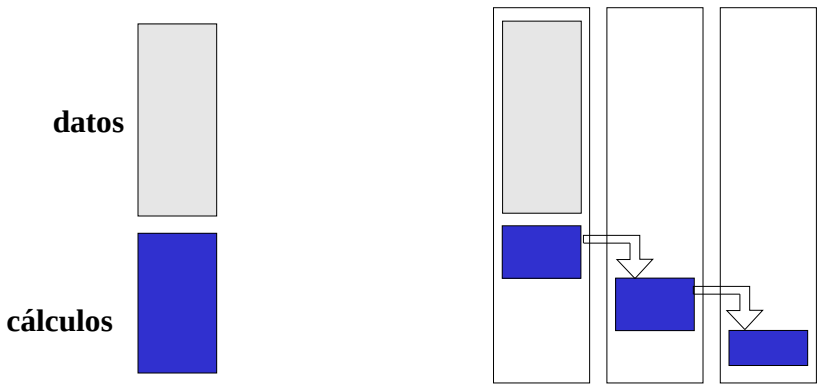


Figura 1.11: Paralelismo de flujo.

Los recursos de cálculo se asocian a las acciones y en cadena, de manera que los

resultados de las acciones efectuadas en el instante t pasen en el instante $t + 1$ al PE siguiente. Este modo de funcionamiento se llama también *segmentación* o *pipeline*.

El flujo de datos puede provenir de dos fuentes:

- Datos de tipo vectorial ubicados en memoria. Existe entonces una dualidad fuerte con el caso del paralelismo de datos.
- Datos de tipo escalar provenientes de un dispositivo de entrada. Este dispositivo se asocia a menudo a otro de captura de datos, colocado en un entorno de tiempo real.

En ambos casos, la ganancia obtenida está en relación con el número de etapas (número de PE). Todos los PEs no estarán ocupados mientras el primer dato no haya recorrido todo el cauce, lo mismo ocurrirá al final del flujo. Si el flujo presenta frecuentes discontinuidades, las fases transitorias del principio y del fin pueden degradar seriamente la ganancia. La existencia de bifurcaciones también limita la ganancia obtenida.

Capítulo 2

El rendimiento de los sistemas paralelos

2.1 Magnitudes y medidas del rendimiento

En esta sección se definirán algunas de las medidas más utilizadas a la hora de determinar el rendimiento de una arquitectura paralela. Así, se introducen los conceptos de: *speed-up*, eficiencia de un sistema, utilización, redundancia, etc. Esta sección y la siguiente se encuentran completas en [Hwa93].

2.1.1 Eficiencia, redundancia, utilización y calidad

Ruby Lee (1980) definió varios parámetros para evaluar el cálculo paralelo. A continuación se muestra la definición de dichos parámetros.

Eficiencia del sistema. Sea $O(n)$ el número total de operaciones elementales realizadas por un sistema con n elementos de proceso, y $T(n)$ el tiempo de ejecución en pasos unitarios de tiempo. En general, $T(n) < O(n)$ si los n procesadores realizan más de una operación por unidad de tiempo, donde $n \geq 2$. Supongamos que $T(1) = O(1)$ en un sistema mono-procesador. El *factor de mejora del rendimiento* (*speed-up*) se define como

$$S(n) = T(1)/T(n)$$

La *eficiencia del sistema* para un sistema con n procesadores se define como

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

La eficiencia es una comparación del grado de *speed-up* conseguido frente al valor máximo. Dado que $1 \leq S(n) \leq n$, tenemos $1/n \leq E(n) \leq 1$.

La eficiencia más baja ($E(n) \to 0$ corresponde al caso en que todo el programa se ejecuta en un único procesador de forma serie. La eficiencia máxima ($E(n) = 1$) se obtiene cuando todos los procesadores están siendo completamente utilizados durante todo el periodo de ejecución.

Escalabilidad. Un sistema se dice que es *escalable* para un determinado rango de procesadores $[1..n]$, si la eficiencia $E(n)$ del sistema se mantiene constante y cercana a la unidad en todo ese rango. Normalmente todos los sistemas tienen un determinado número de procesadores a partir del cual la eficiencia empieza a disminuir de forma más o menos brusca. Un sistema es más escalable que otro si este número de procesadores, a partir del cual la eficiencia disminuye, es menor que el otro.

No hay que confundir escalabilidad con ampliabilidad. Un sistema es ampliable si físicamente se le pueden poner más módulos (más memorias, más procesadores, más tarjetas de entrada/salida, etc). Que un sistema sea ampliable no significa que sea escalable, es decir, que un sistema sea capaz de ampliarse con muchos procesadores no significa que el rendimiento vaya a aumentar de forma proporcional, por lo que la eficiencia no tiene por qué mantenerse constante y por tanto el sistema podría no ser escalable.

Redundancia y utilización. La *redundancia* en un cálculo paralelo se define como la relación entre $O(n)$ y $O(1)$:

$$R(n) = O(n)/O(1)$$

Esta proporción indica la relación entre el paralelismo software y hardware. Obviamente, $1 \leq R(n) \leq n$. La *utilización del sistema* en un cálculo paralelo se define como

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)}$$

La utilización del sistema indica el porcentaje de recursos (procesadores, memoria, recursos, etc.) que se utilizan durante la ejecución de un programa paralelo. Es interesante observar la siguiente relación: $1/n \leq E(n) \leq U(n) \leq 1$ y $1 \leq R(n) \leq 1/E(n) \leq n$.

Calidad del paralelismo. La *calidad* de un cálculo paralelo es directamente proporcional al *speed-up* y la eficiencia, inversamente proporcional a la redundancia. Así, tenemos

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)}$$

Dado que $E(n)$ es siempre una fracción y $R(n)$ es un número entre 1 y n, la calidad $Q(n)$ está siempre limitada por el *speed-up* $S(n)$.

Para terminar con esta discusión acerca de los índices del rendimiento, usamos el *speed-up* $S(n)$ para indicar el grado de ganancia de velocidad de una computación paralela. La eficiencia $E(n)$ mide la porción útil del trabajo total realizado por n procesadores. La redundancia $R(n)$ mide el grado del incremento de la carga.

La utilización $U(n)$ indica el grado de utilización de recursos durante un cálculo paralelo. Finalmente, la calidad $Q(n)$ combina el efecto del *speed-up*, eficiencia y redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema.

2.1.2 Perfil del paralelismo en programas

El grado de paralelismo refleja cómo el paralelismo software se adapta al paralelismo hardware. En primer lugar caracterizaremos el perfil del paralelismo de un programa

para a continuación introducir los conceptos de paralelismo medio y definir el *speed-up* ideal en una máquina con recursos ilimitados.

Grado de paralelismo. *Es el número de procesos paralelos en los que se puede dividir un programa en un instante dado.* La ejecución de un programa en un ordenador paralelo puede utilizar un número diferente de procesadores en diferentes periodos de tiempo. Para cada periodo de tiempo, el número de procesadores que se puede llegar a usar para ejecutar el programa se define como el *grado de paralelismo* (GDP).

A la gráfica 2.1, que muestra el GDP en función del tiempo, se la denomina *perfil del paralelismo* de un programa dado. Por simplicidad, nos concentraremos en el análisis de los perfiles de un único programa. En la figura se muestra un ejemplo de perfil del paralelismo del algoritmo divide y vencerás.

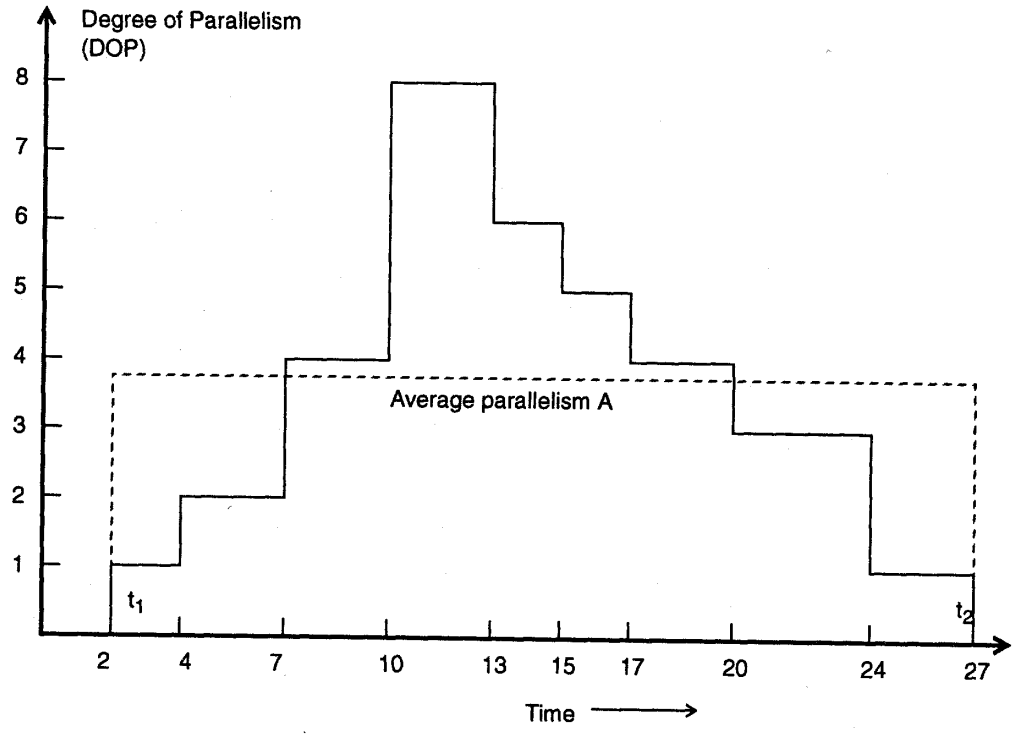


Figura 2.1: Perfil del paralelismo de un algoritmo del tipo divide y vencerás.

Las fluctuaciones en el perfil durante un periodo de observación depende de la estructura del algoritmo, la optimización del programa, la utilización de recursos, y las condiciones de ejecución del sistema donde se ejecuta el programa.

Paralelismo medio. Consideremos un procesador paralelo compuesto por n elementos de proceso homogéneos. Llamamos m al paralelismo máximo en un perfil. En el caso ideal $n \gg m$. Llamamos Δ a la *capacidad de cómputo* de un procesador, expresada en MIPS o Mflops, sin considerar las penalizaciones debidas al acceso a memoria, latencia de las comunicaciones, o sobrecarga del sistema. Cuando i procesadores están ocupados durante un periodo de tiempo, se tiene que GDP$= i$ en ese periodo.

La cantidad de trabajo realizado, a la que llamaremos W, es proporcional al área bajo la curva de perfil paralelo:

$$W = \Delta \int_{t_1}^{t_2} \mathrm{GDP}(t)\, dt$$

Esta integral se calcula frecuentemente mediante el siguiente sumatorio:

$$W = \Delta \sum_{i=1}^{m} i \cdot t_i$$

donde t_i es el tiempo que GDP$= i$ y $\sum_{i=1}^{m} t_i = t_2 - t_1$ es el tiempo total de ejecución.

El *paralelismo medio*, que llamaremos A, será por tanto

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \mathrm{GDP}(t)\, dt$$

o en su forma discreta

$$A = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} \tag{2.1}$$

***Speed-up* asintótico**. Si denotamos por $W_i = i\Delta t_i$ al trabajo realizado cuando GDP$= i$, entonces podemos escribir $W = \sum_{i=1}^{m} W_i$. Esto está suponiendo que no hay sobrecarga de ningún tipo, es decir, se trata del caso ideal de paralelización.

El tiempo de ejecución de W_i sobre un único procesador es $t_i(1) = W_i/\Delta$. El tiempo de ejecución de W_i sobre k procesadores es $t_i(k) = W_i/k\Delta$. Con un número infinito de procesadores disponibles, $t_i(\infty) = W_i/i\Delta$, para $1 \leq i \leq m$. Así, podemos escribir el *tiempo de respuesta* para un procesador e infinitos procesadores como:

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta}$$

El *speed-up asintótico* S_∞ se define como el cociente de $T(1)$ y $T(\infty)$, es decir, es un parámetro que mide la aceleración del tiempo de cálculo por el hecho de poder paralelizar al máximo la aplicación:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}} \tag{2.2}$$

Si comparamos esta fórmula (2.2) con la del paralelismo medio, A (2.1), se observa que $S_\infty = A$ en el caso ideal. En general, $S_\infty \leq A$ si se consideran las latencias debidas a las comunicaciones y otras sobrecargas del sistema. Observar que tanto S_∞ como A están definidos bajo la suposición de que $n = \infty$ o $n \gg m$.

Paralelismo disponible. Como ya vimos en las primeras secciones de este tema, existe un amplio grado de paralelismo potencial en los programas. Los códigos científicos presentan un alto grado de paralelismo, debido al paralelismo inherente de los propios

datos. Manoj Kumar (1988) indica que en códigos de cálculo intensivo es posible ejecutar de 500 a 3.500 operaciones aritméticas simultáneas. Nicolau y Fisher (1984) mostraron que un con programa Fortran estándar era posible la ejecución simultánea de 90 instrucciones para arquitecturas VLIW. Estos números muestran el lado optimista del paralelismo disponible.

David Wall (1991) indica que el límite del ILP (paralelismo a nivel de instrucción) es de alrededor de 5, raramente sobrepasando el 7. Bulter et al. (1991) indican que cuando se eliminan todas las restricciones el GDP puede exceder 17 instrucciones por ciclo. Si el hardware está perfectamente balanceado es posible conseguir de 2.0 a 5.8 instrucciones por ciclo en un procesador superescalar. Estos números muestran el lado pesimista del paralelismo disponible.

2.1.3 Rendimiento medio armónico. Ley de Amdahl

Consideremos un sistema paralelo con n procesadores ejecutando m programas en varios modos con diferentes niveles de rendimiento. Queremos definir el rendimiento medio de este tipo de multiprocesadores. Con una distribución de peso podemos definir una expresión del rendimiento.

Cada modo de ejecución puede corresponder a un tipo de ejecución como por ejemplo procesamiento escalar, vectorial, secuencial o paralela. Cada programa puede ejecutarse mediante una combinación de estos modos. El rendimiento *medio armónico* proporciona un rendimiento medio sobre la ejecución de un gran número de programas ejecutándose en varios modos.

Antes de obtener dicha expresión, estudiaremos las expresiones de la *media aritmética* y *geométrica* obtenidas por James Smith (1988). La velocidad de ejecución R_i para el programa i-ésimo se mide en MIPS o Mflops.

Media aritmética del rendimiento. Sea el conjunto $\{R_i\}$ de las velocidades de ejecución de los programas $i = 1, 2, \ldots, m$. La *media aritmética de la velocidad de ejecución* se define como

$$R_a = \sum_{i=1}^{m} \frac{R_i}{m}$$

La expresión R_a supone que los m programas tienen el mismo peso $(1/m)$. Si existe una distribución de pesos de los distintos programas $\pi = \{f_i \text{ para } i = 1, 2, \ldots, m\}$, definimos la *media aritmética ponderada de la velocidad de ejecución* como:

$$R_a^* = \sum_{i=1}^{m} (f_i R_i)$$

Esta media aritmética es proporcional a la suma de los inversos de los tiempos de ejecución; no es inversamente proporcional a la suma de los tiempos de ejecución. Por lo tanto, la media aritmética falla al representar el tiempo real consumido por los *benchmarks*.

Media geométrica del rendimiento. La *media geométrica de la velocidad de*

ejecución para m programas se define como

$$R_g = \prod_{i=1}^{m} R_i^{1/m}$$

Con una distribución de pesos $\pi = \{f_i \text{ para } i = 1, 2, \ldots, m\}$, podemos definir una *media geométrica ponderada de la velocidad de ejecución* como:

$$R_g^* = \prod_{i=1}^{m} R_i^{f_i}$$

La media geométrica tampoco capta el rendimiento real, ya que no presenta una relación inversa con el tiempo total. La media geométrica ha sido defendida para el uso con cifras de rendimiento que han sido normalizadas con respecto a una máquina de referencia con la que se está comparando.

Rendimiento medio armónico. Debido a los problemas que presentan la media aritmética y geométrica, necesitamos otra expresión del rendimiento medio basado en la media aritmética del tiempo de ejecución. De hecho, $T_i = 1/R_i$, es el tiempo medio de ejecución por instrucción para el programa i. La *media aritmética del tiempo de ejecución* por instrucción se define como

$$T_a = \frac{1}{m} \sum_{i=1}^{m} T_i = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{R_i}$$

La *media armónica de la velocidad de ejecución* sobre m programas de prueba se define por el hecho de que $R_h = \frac{1}{T_a}$:

$$R_h = \frac{m}{\sum_{i=1}^{m} \frac{1}{R_i}}$$

Con esto, el rendimiento medio armónico está de verdad relacionado con el tiempo medio de ejecución. Si consideramos una distribución de pesos, podemos definir el *rendimiento medio armónico ponderado* como:

$$R_h^* = \frac{1}{\sum_{i=1}^{m} \frac{f_i}{R_i}}$$

***Speed-up* armónico medio**. Otra forma de aplicar el concepto de media armónica es ligar los distintos modos de un programa con el número de procesadores usados. Supongamos que un programa (o una carga formada por la combinación de varios programas) se ejecutan en un sistema con n procesadores. Durante el periodo de ejecución, el programa puede usar $i = 1, 2, \ldots n$ procesadores en diferentes periodos de tiempo.

Decimos que el programa se ejecuta en *modo* i si usamos i procesadores. R_i se usa para reflejar la velocidad conjunta de i procesadores. Supongamos que $T_1 = 1/R_1 = 1$ es el tiempo de ejecución secuencial en un mono-procesador con una velocidad de ejecución $R_1 = 1$. Entonces $T_i = 1/R_i = 1/i$ es el tiempo de ejecución usando i procesadores con una velocidad de ejecución combinada de $R_i = i$ en el caso ideal.

Supongamos que un programa dado se ejecuta en n modos de ejecución con una distribución de pesos $w = \{f_i \text{ para } i = 1, 2, \ldots, n\}$. El *speed-up armónico medio ponderado* se define como:

$$S = T_1/T^* = \frac{1}{\left(\sum_{i=1}^{n} \frac{f_i}{R_i}\right)} \tag{2.3}$$

donde $T^* = 1/R_h^*$ es la *media armónica ponderada del tiempo de ejecución* para los n modos de ejecución.

La figura 2.2 muestra el comportamiento del *speed-up* para tres funciones de peso distintas.

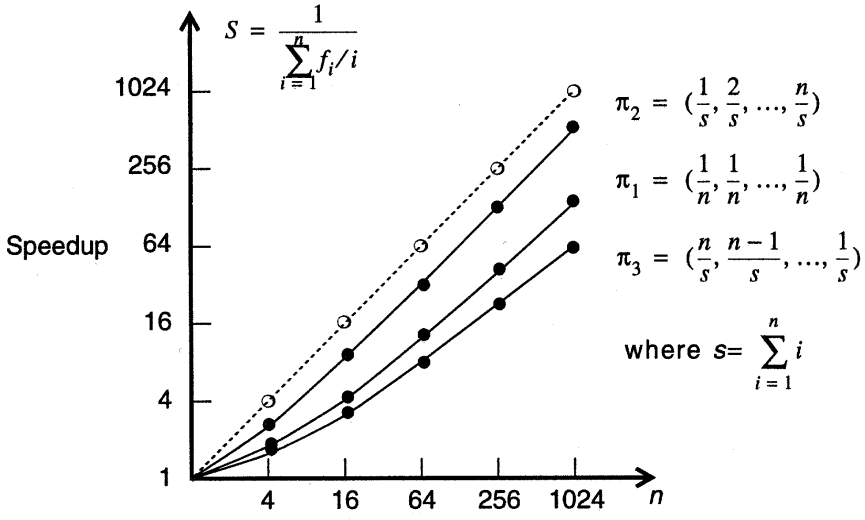


Figura 2.2: Media armónica del *speed-up* con respecto a tres distribuciones de probabilidad: π_1 para la distribución uniforme, π_2 en favor de usar más procesadores y π_3 en favor de usar menos procesadores.

Ley de Amdahl. De la expresión (2.3) de S se puede derivar la ley de Amdahl como sigue: En primer lugar supongamos que $R_i = i$ y $w = (\alpha, 0, 0, \ldots, 0, 1-\alpha)$. Esto implica que el sistema usa un modo secuencial puro con una probabilidad de α, o los n procesadores con una probabilidad de $1-\alpha$. Sustituyendo $R_1 = 1$, $R_n = n$ y w en la ecuación de S (2.3), obtenemos la siguiente expresión para el *speed-up*:

$$S_n = \frac{n}{1 + (n-1)\alpha} \tag{2.4}$$

A esta expresión se le conoce como la *ley de Amdahl*. La implicación es que $S \to 1/\alpha$ cuando $n \to \infty$. En otras palabras, independientemente del número de procesadores que se emplee, existe un límite superior del *speed-up* debido a la parte serie de todo programa.

En la figura 2.3 se han trazado las curvas correspondientes a la ecuación (2.4) para 4 valores de α. El *speed-up* ideal se obtiene para $\alpha = 0$, es decir, el caso en que no hay parte serie a ejecutar y todo el código es paralelizable. A poco que el valor de α sea no nulo, el *speed-up* máximo empieza a decaer muy deprisa.

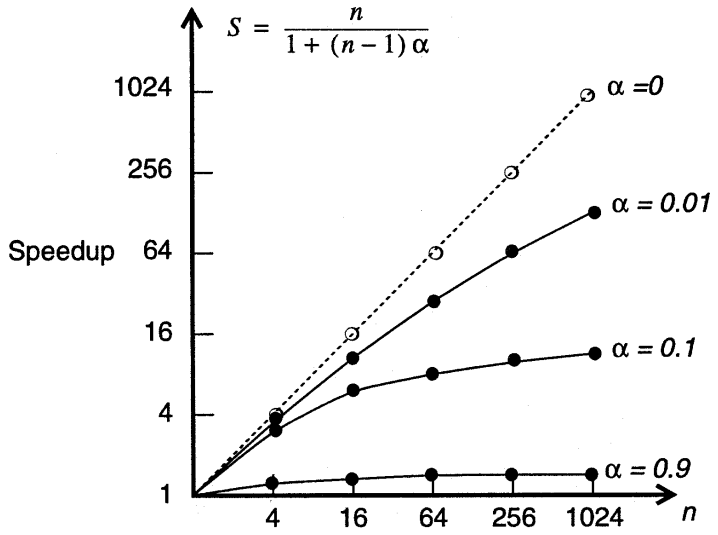


Figura 2.3: Mejora del rendimiento para diferentes valores de α, donde α es la fracción del cuello de botella secuencial.

Esta ley de Amdahl se puede generalizar y es aplicable a cualquier problema que tenga una parte *mejorable* y otra que no se pueda mejorar. Si llamamos F_m a la fracción del problema que se puede mejorar, la fracción de problema que no se puede mejorar será $(1 - F_m)$. Dado el problema se tiene una magnitud que es la que mejora con respecto a la inicial, a la magnitud inicial la podemos llamar M_{ini} y a la magnitud una vez aplicadas las mejoras M_{mej}. La mejora S_{up} es siempre el cociente entre las dos:

$$S_{up} = \frac{M_{ini}}{M_{mej}}$$

Este cociente se puede poner en función de las fracciones que son mejorables y las que no, ya que $M_{ini} = K \cdot ((1 - F_m) + F_m) = K$, y $M_{mej} = K \cdot ((1 - F_m) + F_m/S_m)$, donde la K es una proporcionalidad con las unidades de la magnitud del problema, y S_m es el factor de mejora de la parte mejorable. Con todo esto se puede reescribir la mejora total del sistema al intentar mejorar S_m veces la parte mejorable como:

$$S_{up} = \frac{1}{1 - F_m + \frac{F_m}{S_m}}$$

que es la expresión de la ley de Amdahl generalizada que se puede aplicar a cualquier objeto que se quiera mejorar y sólo una parte sea mejorable. En el caso de los multiprocesadores el factor de mejora de la parte mejorable (paralelizable) es precisamente n, es decir, el número de procesadores. Por otro lado, la fracción que no es mejorable es la parte serie no paralelizable que llamábamos α. Con todo esto, sustituyendo los diferentes valores en la expresión anterior, y multiplicando por n/n, se tiene:

$$S_{up} = \frac{1}{\alpha + \frac{1-\alpha}{n}} = \frac{n}{1 + n\alpha - \alpha} = \frac{n}{1 + (n-1)\alpha}$$

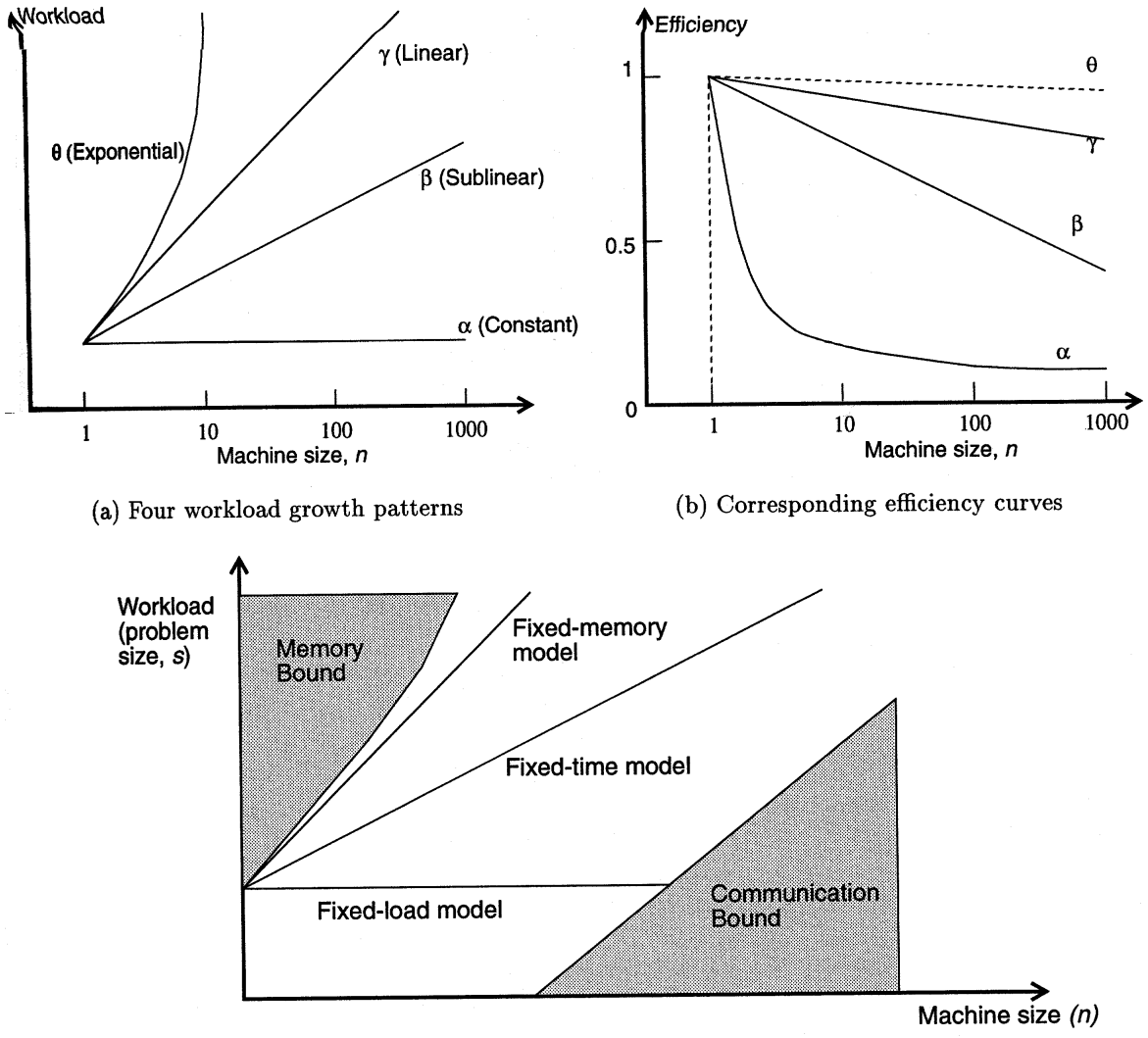
que es exactamente la misma expresión obtenida aplicando el *speed-up* medio armónico.

La expresión de la ley de Amdahl generalizada se puede aplicar a cualquier problema. Por ejemplo, el rendimiento relativo vectorial/escalar en los procesadores vectoriales,

no es más que la aplicación de esta ley al caso en que un programa tenga una parte vectorizable (parte que va a mejorar) y otra escalar, cuyo rendimiento no va a mejorar por el hecho de estar utilizando un procesador vectorial.

2.2 Modelos del rendimiento del *speed-up*

En esta sección se describen tres modelos de medición del *speed-up*. La ley de Amdahl (1967) se basa una carga de trabajo fija o en un problema de tamaño fijo. La ley de Gustafson (1987) se aplica a problemas escalables, donde el tamaño del problema se incrementa al aumentar el tamaño de la máquina o se dispone de un tiempo fijo para realizar una determinada tarea. El modelo de *speed-up* de Sun y Ni (1993) se aplica a problemas escalables limitados por la capacidad de la memoria. En la figura 2.4 se muestra un esquema de los tres modelos utilizados.



(a) Four workload growth patterns

(b) Corresponding efficiency curves

Figura 2.4: Modelos de rendimiento del *speed-up*.

2.2.1 Ley de Amdahl, limitación por carga de trabajo fija

En muchas aplicaciones prácticas, donde es importante la respuesta más rápida posible, la carga de trabajo se mantiene fija y es el tiempo de ejecución lo que se debe intentar reducir. Al incrementarse el número de procesadores en el sistema paralelo, la carga fija se distribuye entre más procesadores para la ejecución paralela. Por lo tanto, el objetivo principal es obtener los resultados lo más pronto posible. En otras palabras, disminuir el tiempo de respuesta es nuestra principal meta. A la ganancia de tiempo obtenida para este tipo de aplicaciones donde el tiempo de ejecución es crítico se le denomina *speed-up bajo carga fija*.

***Speed-up* bajo carga fija**. La fórmula vista en el apartado anterior se basa en una carga de trabajo fija, sin importar el tamaño de la máquina. Las formulaciones tradicionales del *speed-up*, incluyendo la ley de Amdahl, están basadas en un problema de tamaño fijo y por lo tanto en una carga fija. En este caso, el factor de *speed-up* está acotado superiormente por el cuello de botella secuencial.

A continuación se consideran las dos posibles situaciones: GDP$< n$ ó GDP$\geq n$. Consideremos el caso donde el GDP$= i \geq n$. Supongamos que todos los n procesadores se usan para ejecutar W_i exclusivamente. El tiempo de ejecución de W_i es

$$t_i(n) = \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil$$

De esta manera el tiempo de respuesta es

$$T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil$$

Observar que si $i < n$, entonces $t_i(n) = t_i(\infty) = W_i/i\Delta$. Ahora, definimos el *speed-up para carga fija* como :

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil}$$

Observar que $S_n \leq S_\infty \leq A$.

Existe una gran cantidad de factores que se han ignorado que pueden rebajar el *speed-up*. Estos factores incluyen latencias de comunicaciones debidas a retrasos en el acceso a la memoria, comunicaciones a través de un bus o red, o sobrecarga del sistema operativo y retrasos causados por las interrupciones. Si $Q(n)$ es la suma de todas las sobrecargas del sistema en un sistema con n procesadores, entonces:

$$S_n = \frac{T(1)}{T(n) + Q(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} \tag{2.5}$$

El retraso por sobrecarga $Q(n)$ depende siempre de la aplicación y de la máquina. Es muy difícil obtener una expresión para $Q(n)$. A no ser de que se especifique otra cosa se supondrá que $Q(n) = 0$ para simplificar la explicación.

Ley de Amdahl revisada. En 1967, Gene Amdahl derivó un *speed-up* para el caso particular donde el computador opera en modo puramente secuencial (GDP$= 1$)

o en modo totalmente paralelo (GDP= n). Es decir, $W_i = 0$ si $i \neq 1$ ó $i \neq n$. En este caso, el *speed-up* viene dado por:

$$S_n = \frac{W_1 + W_n}{W_1 + W_n/n} \tag{2.6}$$

La ley de Amdahl supone que la parte secuencia del programa W_i no cambia con respecto a tamaño n de la máquina. Sin embargo, la porción paralela se ejecuta equitativamente por los n procesadores reduciéndose el tiempo de ejecución para esta parte.

Suponiendo una situación normalizada en la cual $W_1 = \alpha$ y $W_n = 1 - \alpha$ se tiene que $W_1 + W_n = \alpha + 1 - \alpha = 1$. Con esta sustitución, la ecuación (2.4) y (2.6) son la misma. Igual que en aquella expresión α es la fracción serie del programa y $(1 - \alpha)$ la paralelizable.

La ley de Amdahl se ilustra en la figura 2.5. Cuando el número de procesadores aumenta, la carga ejecutada en cada procesador decrece. Sin embargo, la cantidad total de trabajo (carga) $W_1 + W_n$ se mantiene constante como se muestra en la figura 2.5a. En la figura 2.5b, el tiempo total de ejecución decrece porque $T_n = W_n/n$. Finalmente, el término secuencial domina el rendimiento porque $T_n \to 0$ al hacer n muy grande siendo T_1 constante.

Cuello de botella secuencial. La figura 2.5c muestra una gráfica de la ley de Amdahl para diferentes valores de $0 \leq \alpha \leq 1$. El máximo *speed-up*, $S_n = n$, se obtiene para $\alpha = 0$. El mínimo *speed-up*, $S_n = 1$, se obtiene para $\alpha = 1$. Cuando $n \to \infty$, el valor límite es $S_\infty = 1/\alpha$. Esto implica que el *speed-up* está acotado superiormente por $1/\alpha$, independientemente del tamaño de la máquina.

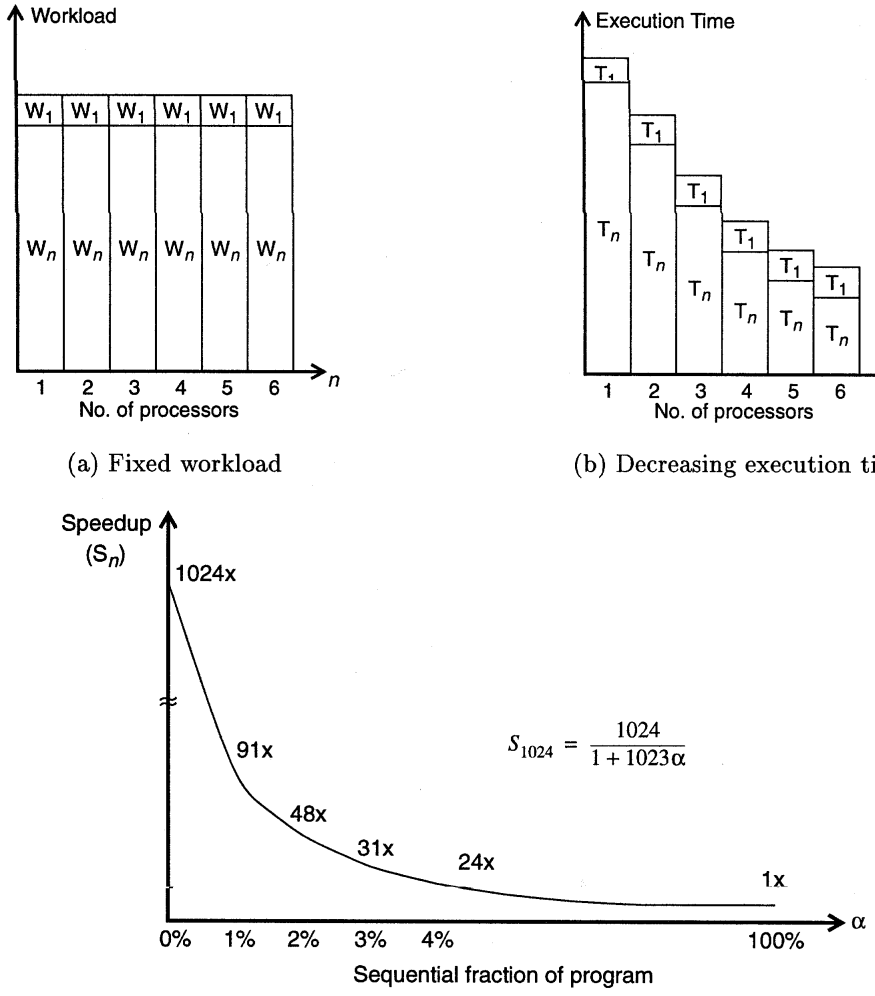
La curva del *speed-up* en la figura 2.5c cae rápidamente al aumentar α. Esto significa que con un pequeño porcentaje de código secuencial, el rendimiento total no puede ser superior a $1/\alpha$. A este α se le denomina *cuello de botella secuencial* de un programa.

El problema de un cuello de botella secuencial no puede resolverse incrementando el número de procesadores del sistema. El problema real está en la existencia de una fracción secuencial (s) del código. Esta propiedad ha impuesto una visión muy pesimista del procesamiento paralelo en las pasadas dos décadas.

De hecho, se observaron dos impactos de esta ley en la industria de los computadores paralelos. En primer lugar, los fabricantes dejaron de lado la construcción de computadores paralelos de gran escala. En segundo lugar, una parte del esfuerzo investigador se desplazó al campo de desarrollo de compiladores paralelos en un intento de reducir el valor de α y mejorar de esa forma el rendimiento.

2.2.2 Ley de Gustafson, limitación por tiempo fijo

Uno de los mayores inconvenientes de aplicar la ley de Amdahl es que el problema (la carga de trabajo) no puede aumentarse para corresponderse con el poder de cómputo al aumentar el tamaño de la máquina. En otras palabras, el tamaño fijo impide el escalado del rendimiento. Aunque el cuello de botella secuencial es un problema importante, puede aliviarse en gran medida eliminando la restricción de la carga fija (o tamaño fijo del problema). John Gustafson (1988) ha propuesto el concepto de tiempo fijo que da lugar a un modelo del *speed-up* escalado.



(a) Fixed workload

(b) Decreasing execution time

(c) Speedup with a fixed load

Figura 2.5: Modelo del *speed-up* de carga fija y la ley de Amdahl.

Escalado para conseguir una mayor precisión. Las aplicaciones de tiempo-real son la causa principal del desarrollo de un modelo de *speed-up* de carga fija y la ley de Amdahl. Existen muchas otras aplicaciones que enfatizan la precisión más que el tiempo de respuesta. Al aumentar el tamaño de la máquina para obtener mayor potencia de cálculo, queremos incrementar el tamaño del problema para obtener una mayor carga de trabajo, produciendo una solución más precisa y manteniendo el tiempo de ejecución.

Un ejemplo de este tipo de problemas es el cálculo de la predicción meteorológica. Habitualmente se tiene un tiempo fijo para calcular el tiempo que hará en unas horas, naturalmente se debe realizar el cálculo antes de que la lluvia llegue. Normalmente se suele imponer un tiempo fijo de unos 45 minutos a una hora. En ese tiempo se tiene que obtener la mayor precisión posible. Para calcular la predicción se sigue un modelo físico que divide el área terrestre a analizar en cuadrados, de manera que los cálculos

realizados en cada uno de estos cuadrados se puede hacer en paralelo. Si se disponen de muchos procesadores se podrán hacer cuadros más pequeños con lo que la precisión aumenta manteniendo el tiempo de ejecución.

***Speed-up* de tiempo fijo**. En aplicaciones de precisión crítica, se desea resolver el problema de mayor tamaño en una máquina mayor con el aproximadamente el mismo tiempo de ejecución que costaría resolver un problema menor en una máquina menor. Al aumentar el tamaño de la máquina, tendremos una nueva carga de trabajo y por lo tanto un nuevo perfil del paralelismo. Sea m' el máximo GDP con respecto al problema escalado y W_i' la carga de trabajo con GDP$= i$.

Observar que, en general, $W_i' > W_i$ para $2 \leq i \leq m'$ y $W_1' = W_1$. El *speed-up* de tiempo fijo se define bajo el supuesto de que $T(1) = T'(n)$, donde $T'(n)$ es el tiempo de ejecución del problema escalado y $T(1)$ se corresponde con el problema original sin escalar. Así, tenemos que

$$\sum_{i=1}^{m} W_i = \sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)$$

Una fórmula general para el *speed-up* de tiempo fijo se define por $S_n' = T'(1)/T'(n) = T'(1)/T(1)$. Por analogía con la ecuación (2.5) se obtiene la expresión para el *speed-up* de tiempo fijo:

$$S'_n = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} \tag{2.7}$$

Ley de Gustafson. El *speed-up* de tiempo fijo fue desarrollado por Gustafson para un perfil de paralelismo especial con $W_i = 0$ si $i \neq 1$ y $i \neq n$. De forma similar a la ley de Amdahl, podemos reescribir la ecuación anterior (2.7) como sigue (suponemos $Q(n) = 0$):

$$S'_n = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} = \frac{W_1' + W_n'}{W_1 + W_n} = \frac{W_1 + nW_n}{W_1 + W_n}$$

La figura 2.6a muestra la relación del escalado de la carga de trabajo con el *speed-up* escalado de Gustafson. De hecho, la ley de Gustafson puede reformularse en términos de $\alpha = W_1$ y $1 - \alpha = W_n$, bajo la suposición de que $W_1 + W_n = 1$, como sigue:

$$S'_n = \frac{\alpha + n(1-\alpha)}{\alpha + (1-\alpha)} = n - \alpha(n-1)$$

Obsérvese que la pendiente de la curva S_n en la figura 2.6c es mucho más plana que en la figura 2.5c. Esto implica que la ley de Gustafson soporta el rendimiento escalable al aumentar el tamaño de la máquina. La idea es mantener a todos los procesadores ocupados incrementando el tamaño del problema.

2.2.3 Modelo del *speed-up* limitado por la memoria fija

Xian-He Sun y Lionel Ni (1993) han desarrollado un modelo del *speed-up* limitado por la memoria que generaliza la ley de Amdahl y Gustafson para maximizar el uso de la CPU y la memoria. La idea es resolver el mayor problema posible, limitado por el

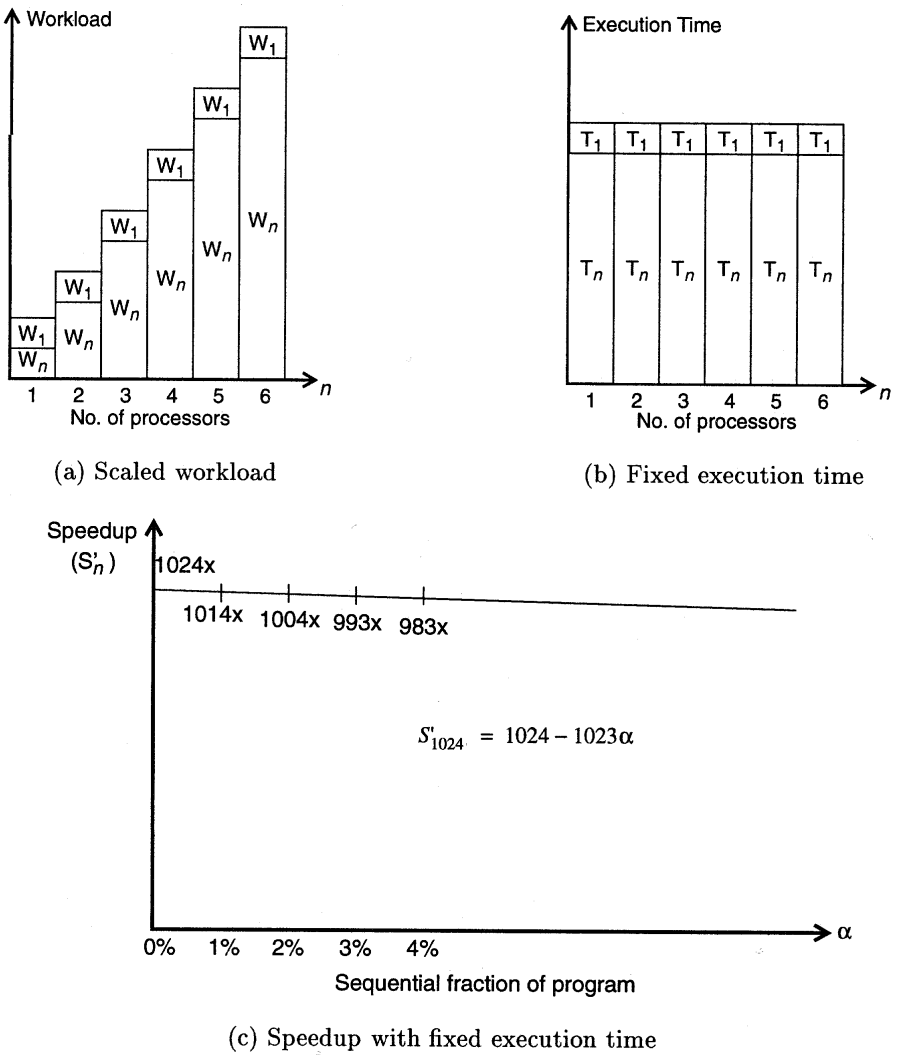


Figura 2.6: Modelo de *speed-up* de tiempo fijo y la ley de Gustafson.

espacio de memoria. En este caso también es necesario una carga de trabajo escalada, proporcionando un mayor *speed-up*, mayor precisión y mejor utilización de los recursos.

Problemas limitados por el espacio de memoria. Los cálculos científicos y las aplicaciones de ingeniería suelen necesitar una gran cantidad de memoria. De hecho, muchas aplicaciones de los ordenadores paralelos surgen de la limitación de la memoria más que de la CPU o la E/S. Esto es especialmente cierto en sistemas multicomputador con memoria distribuida. Cada elemento de proceso está limitado a usar su propia memoria local por lo que sólo puede hacer frente a un pequeño subproblema.

Cuando se utiliza un mayor número de nodos para resolver un problema grande, la capacidad de memoria total se incrementa de forma proporcional. Esto le permite al sistema resolver un problema escalado mediante el particionamiento del programa y la descomposición del conjunto de datos.

En lugar de mantener fijo el tiempo de ejecución, uno puede querer usar toda la memoria disponible para aumentar aún más el tamaño del problema. En otras palabras, si se tiene un espacio de memoria adecuado y el problema escalado cumple el límite de

tiempo impuesto por la ley de Gustafson, se puede incrementar el tamaño del problema, consiguiendo una mejor solución o una solución más precisa.

El modelo de limitación de la memoria se desarrolló bajo esta filosofía. La idea es resolver el mayor problema posible, limitado únicamente por la capacidad de memoria disponible.

***Speed-up* de memoria fija**. Sea M el requisito de memoria para un problema dado y W la carga computacional. Ambos factores están relacionados de varias formas, dependiendo del direccionamiento del espacio y las restricciones de la arquitectura. Así, podemos escribir $W = g(M)$ o $M = g^{-1}(W)$.

En un multicomputador, y en la mayoría de multiprocesadores, la capacidad total de la memoria se incrementa linealmente con el número de nodos disponibles. Sea $W = \sum_{i=1}^{m} W_i$ la carga para una ejecución secuencial del programa en un único nodo, y $W^* = \sum_{i=1}^{m^*} W_i^*$ la carga para el problema para n nodos, donde m^* es el máximo GDP del problema escalado. Los requisitos de memoria para un nodo activo está limitado por $M = g^{-1}\left(\sum_{i=1}^{m} W_i\right)$.

El *speed-up con memoria fija* se define de forma similar al caso de la ecuación (2.7):

$$S_n^* = \frac{\sum_{i=1}^{m^*} W_i^*}{\sum_{i=1}^{m^*} \frac{W_i^*}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} \tag{2.8}$$

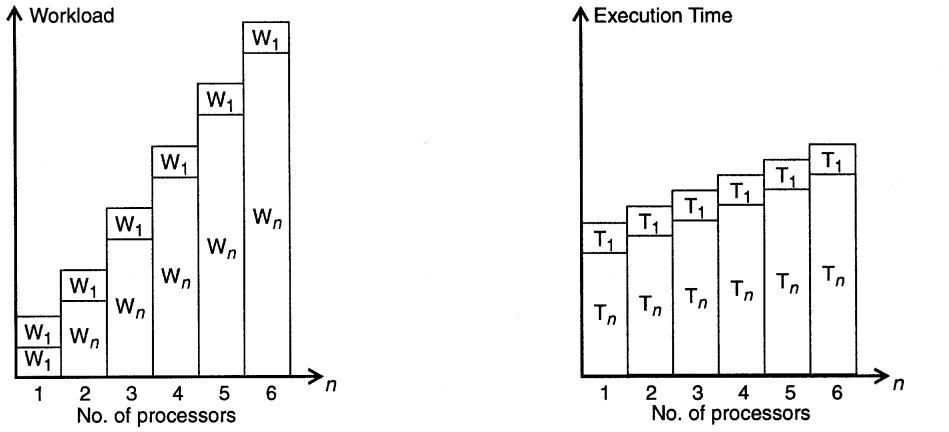
La carga de trabajo para la ejecución secuencial en un único procesador es independiente del tamaño del problema o del tamaño del sistema. Así, podemos escribir $W_1 = W'_1 = W_1^*$ para los tres modelos de *speed-up*. Consideremos el caso especial con dos modos de operación: ejecución *secuencial* frente a *perfectamente paralela*. La mejora en la memoria está relacionada con la carga escalada mediante la fórmula $W_n^* = g^*(nM)$, donde nM es el incremento en la capacidad de la memoria para un multicomputador con n nodos.

Supongamos además que $g^*(nM) = G(n)g(M) = G(n)W_n$, donde $W_n = g(M)$ y g^* es una función homogénea. El factor $G(n)$ refleja el incremento en la carga al aumentar la memoria n veces. Esto nos permite rescribir la fórmula anterior bajo la suposición de que $W_i = 0$ si $i \neq 1$ o n y $Q(n) = 0$:

$$S_n^* = \frac{W_1^* + W_n^*}{W_1^* + W_n^*/n} = \frac{W_1 + G(n)W_n}{W_1 + G(n)W_n/n} \tag{2.9}$$

Rigurosamente hablando este modelo sólo es válido bajo estas dos suposiciones: (1) El conjunto de toda la memoria forma un espacio global de direcciones (en otras palabras, suponemos un espacio de memoria compartido distribuido); (2) Todo el espacio de memoria disponible se utiliza para el problema escalado. Existen tres casos especiales donde se puede aplicar la ecuación (2.9):

1. $G(n) = 1$. Se corresponde con el caso donde el tamaño del problema es fijo, siendo equivalente a la ley de Amdahl.
2. $G(n) = n$. Se aplica al caso en el que la carga se incrementa n veces cuando la memoria se incrementa n veces. En este caso, la ecuación se corresponde con la ley de Gustafson con un tiempo de ejecución fijo.
3. $G(n) > n$. Se corresponde con la situación donde la carga computacional se incrementa más rápidamente que los requisitos de memoria. En este caso, el modelo de memoria fija da posiblemente todavía un mayor *speed-up* que el de tiempo fijo.



(a) Scaled workload

(b) Slightly increased execution time

Figura 2.7: Modelo de *speed-up* de memoria fija.

De este análisis se pueden obtener las siguientes conclusiones: la ley de Amdahl y la de Gustafson son casos particulares del modelo de tiempo fijo. Cuando la computación crece más rápidamente que los requisitos de memoria, lo que es frecuente en el caso de algunas simulaciones científicas y aplicaciones de ingeniería, el modelo de memoria fija (figura 2.7) da lugar a un mayor *speed-up* (es decir, $S_n^* \geq S'_n \geq S_n$) y una mejor utilización de los recursos.

2.3 Modelos del rendimiento según la granularidad

El contenido de esta sección y resto del capítulo se encuentran en el libro [Sto93].

Un parámetro que se suele dar para caracterizar los sistemas multiprocesadores es el **rendimiento de pico** o rendimiento máximo del sistema. Habitualmente este rendimiento de pico se suele calcular como el número de procesadores del sistema multiplicado por el rendimiento de cada uno de los procesadores.

Cuando el sistema opera al rendimiento máximo todos los procesadores están realizando un trabajo útil; ningún procesador está parado y ningún procesador ejecuta alguna instrucción extra que no estuviera en el algoritmo original. En este estado de rendimiento de pico todos los n procesadores están contribuyendo al rendimiento efectivo del sistema y la velocidad de procesamiento viene incrementada por un factor n.

El estado de rendimiento máximo o de pico es un estado raro que difícilmente se puede alcanzar. Hay varios factores que introducen ineficiencia. Algunos de estos factores son los siguientes:

- Retrasos introducidos por las comunicaciones entre procesos.
- La sobrecarga de trabajo debida a la necesidad de sincronizar el trabajo entre los distintos procesadores.
- La pérdida de eficiencia cuando algún procesador se queda sin trabajo para realizar.
- La pérdida de eficiencia cuando uno o más procesadores realizan algún esfuerzo inútil.

- El coste de procesamiento para controlar el sistema y la programación de operaciones.

Estos problemas se hacen realmente serios cuando el número de procesadores es elevado, es decir, es difícil mantener un bajo grado de ineficiencia al aumentar el número de procesadores. Normalmente se obtiene una eficiencia bastante alta con sistemas con pocos procesadores (4-16) pero esta eficiencia se ve seriamente reducida cuando el número de procesadores es alto. Dar el rendimiento de pico de un multiprocesador con pocos procesadores puede dar una idea de su rendimiento efectivo, pero el rendimiento de pico en un multiprocesador con muchos procesadores sólo debe considerarse como parte de las especificaciones, ya que no tiene por qué dar una estimación real del rendimiento del sistema.

A continuación se pretende estudiar la influencia de la sobrecarga de procesamiento por el hecho de añadir más procesadores al cálculo. Se va a comprobar que el rendimiento de un sistema multiprocesador depende fuertemente de la relación R/C, donde R es una unidad de ejecución (con unidades de tiempo o instrucciones por segundo), y C es la sobrecarga debida a las comunicaciones producidas por R. El cociente de los dos da la cantidad de sobrecarga que aparece por unidad de cómputo. Cuando la relación es pequeña no resulta provechoso paralelizar porque aparece mucha sobrecarga. Cuando la relación da un número muy alto entonces es beneficioso paralelizar puesto que la sobrecarga que aparece es pequeña. Normalmente el factor R/C da un valor alto siempre que se divida el problema en trozos grandes, ya que entonces las comunicaciones serán pequeñas comparativamente.

El factor R/C da también idea de la granularidad del sistema, es decir, de lo mucho que se ha dividido el problema en pedazos:

Grano grueso: Un sistema cuyo paralelismo es de *grano grueso* suele tener un factor R/C relativamente grande puesto que los trozos R son grandes y producen un coste de comunicaciones relativamente pequeño. Si un sistema es de grano grueso es beneficioso paralelizar puesto que R/C es grande, pero si los trozos en que se divide el problema son grandes, el problema queda dividido en pocos trozos y el rendimiento máximo no es muy alto (pocas unidades funcionando en paralelo).

Grano fino: Un sistema cuyo paralelismo es de grano fino suele tener un factor R/C pequeño puesto que los trozos R en que se ha dividido el problema son pequeños. Normalmente, si se divide el problema en trozos muy pequeños se obtiene un R/C pequeño por lo que no resulta muy beneficioso paralelizar, pero al ser los trozos pequeños el problema puede quedar muy dividido y cada unidad funcional puede realizar una tarea distinta. En estos casos se podría alcanzar un gran rendimiento por el gran paralelismo existente, pero no se alcanza puesto que el factor R/C es pequeño.

En resumen: si se tiene un problema muy paralelizable (divisible en muchos pedazos) normalmente no va a ser interesante paralelizar tanto puesto que las sobrecargas no van a permitir aprovechar todo ese rendimiento potencial. Si un problema es poco paralelizable (divisible en pocos pedazos) en rendimiento máximo alcanzable es pequeño, ya que se ha dividido el problema en pocos trozos que se ejecutarán en paralelo, pero el paralelismo obtenido será bastante eficiente puesto que las comunicaciones entre trozos grandes son escasas.

2.3.1 Modelo básico: 2 procesadores y comunicaciones no solapadas

Vamos a suponer que cierta aplicación tiene M tareas a realizar las cuales se pueden llevar a cabo de forma paralela. El objetivo está en ejecutar estas M tareas en un sistema con N procesadores en el menor tiempo posible. Para empezar el análisis se comenzará con $N = 2$ procesadores y luego se extenderá el razonamiento a cualquier número de procesadores.

Como punto de partida realizaremos las siguientes suposiciones (más tarde se pueden relajar para obtener resultados más realistas):

1. Cada tarea se ejecuta en R unidades de tiempo.
2. Cada tarea de un procesador se comunica con todas las tareas del resto de procesadores con un coste de sobrecarga de C unidades de tiempo. El coste de comunicaciones con las tareas en el mismo procesador es cero.

Si se tienen dos procesadores se pueden repartir las tareas entre ellos. En un caso extremo se le pueden dar todas las tareas a un único procesador, y en el otro caso extremo se puede realizar un reparto igualitario de tareas entre los dos procesadores. Entre estos dos extremos se tienen situaciones donde un procesador tiene k tareas mientras que el otro tiene $(M - k)$ donde k puede ser cualquier reparto entre 0 y M. En cualquier caso el tiempo de ejecución va a tener dos términos, uno debido al coste de ejecución de las tareas (función de R) y otro debido a la sobrecarga por las comunicaciones (función de C). La expresión que se obtiene para el tiempo de ejecución (T_e) es la siguiente:

$$T_e = R \max\{M - k, k\} + C(M - k)k \tag{2.10}$$

El tiempo propio de ejecución de las tareas es el término $R \max(M - k, k)$ y es lineal con k. El término debido a la sobrecarga es $C(M - k)k$ y es un término que crece cuadráticamente con k. Cuanto más repartidas se encuentran las tareas menor es el término debido a R y mayor es el debido a C. Esto significa que, o bien el tiempo mínimo se obtiene cuando las tareas están igualitariamente repartidas, o bien cuando sólo un procesador ejecuta todas las tareas, no hay término medio.

Las contribuciones de R y C al tiempo de ejecución total se pueden ver mejor en la figura 2.8. En la figura 2.8(a) se muestra la situación en la cual el coste de las comunicaciones es tan alto que resulta más provechoso ejecutar todas las tareas en un único procesador. En la figura 2.8(b) se da la situación en la cual el coste de las comunicaciones es menor que la ganancia obtenida por el hecho de paralelizar, por lo que en este caso es rentable dividir las tareas entre los dos procesadores.

Para obtener la relación R/C a partir de la cual es beneficioso paralelizar basta con igualar el tiempo de ejecución con reparto igualitario ($k = M/2$ reparto igualitario) con el tiempo de ejecución con un único procesador (RM) e igualar los términos debidos a R y C, es decir:

$$RM = R\frac{M}{2} + C\frac{M}{2}\frac{M}{2}$$

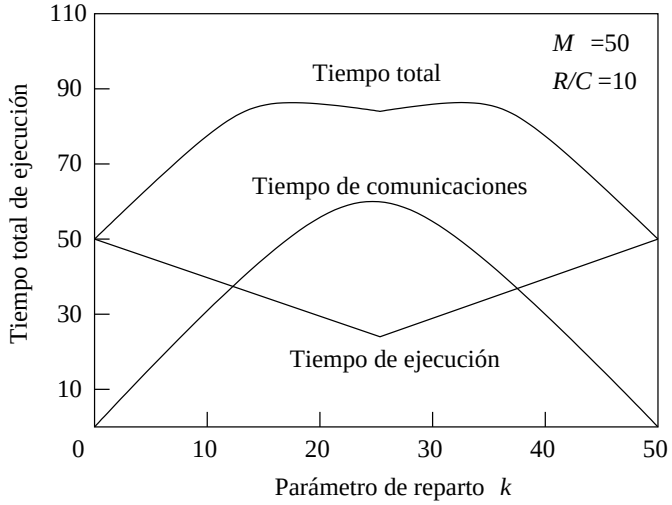
$$R\frac{M}{2} = C\frac{M}{2}\frac{M}{2}$$

realizando unas operaciones básicas de sustitución se tiene finalmente una cota para

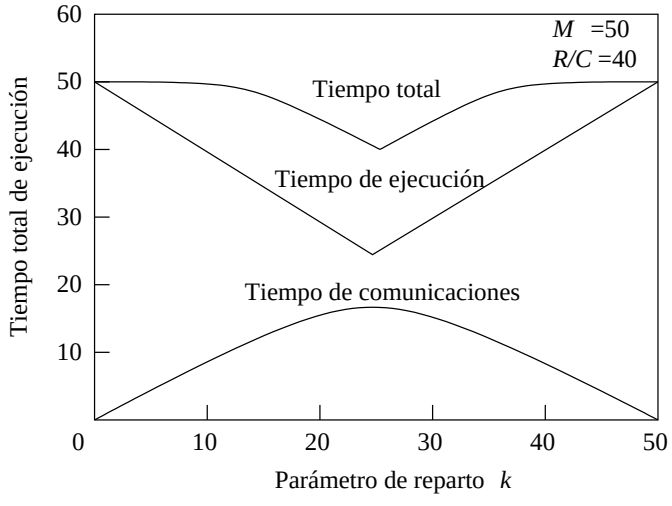
R/C:

$$\frac{R}{C} = \frac{M}{2} \tag{2.11}$$

esto quiere decir que si $R/C > M/2$ entonces resulta beneficioso paralelizar, y en caso contrario resulta mejor dejar todas las tareas en un único procesador.



(a)



(b)

Figura 2.8: Tiempo de ejecución para dos factores R/C diferentes.

2.3.2 Extensión a N procesadores

En el caso de tener N procesadores se puede suponer que cada uno ejecuta k_i tareas, de manera que $M = \sum_{i=1}^{N} k_i$. Con esto se puede generalizar la ecuación 2.10 obteniéndose

la siguiente expresión para el tiempo total de ejecución con N procesadores:

$$\begin{aligned} T_e &= R\ \max\{k_i\} + \tfrac{C}{2}\sum_{i=1}^{N} k_i(M-k_i) \\ &= R\ \max\{k_i\} + \tfrac{C}{2}\left(M^2 - \sum_{i=1}^{N} k_i^2\right) \end{aligned} \tag{2.12}$$

Al igual que ocurría en el caso anterior, el mínimo de esta función se obtiene, o bien cuando sólo un procesador tiene todas las tareas, o bien, cuando se reparten de forma igualitaria las tareas. En este último caso el reparto igualitario es un tanto especial, ya que se deben repartir dándole a cada procesador un número $\lceil \frac{M}{N} \rceil$ de tareas hasta que queden menos de $\lceil \frac{M}{N} \rceil$ tareas que se le asignan a uno de los procesadores que queden; esto significa que pueden haber algunos procesadores que no reciban ninguna tarea.

Por lo tanto, el reparto igualitario deja a p procesadores con $\lceil \frac{M}{N} \rceil$ tareas, a otro se le asignan $M - \lceil \frac{M}{N} \rceil p$ tareas, y el resto de procesadores no tiene ninguna. Se puede demostrar que este reparto, y no otro, es el que da el mínimo. La demostración es como sigue: Supongamos que $k_1 = \lceil \frac{M}{N} \rceil$ es el máximo número de tareas que tiene un procesador. Con el reparto propuesto sólo puede haber un procesador con menos de estas tareas a menos que no tenga ninguna. Vamos a suponer que hay dos procesadores que tienen menos tareas, en vez de uno, y veremos cómo al aumentar las tareas de uno de ellos se reduce el tiempo total de ejecución.

En efecto, sean k_2 y k_3 las tareas asignadas a dos procesadores que cumplen que $k_1 > k_2 \geq k_3 \geq 1$. Supongamos a continuación que pasamos una tarea del procesador que tiene k_3 al procesador que tiene k_2. El coste de ejecución debido a R no cambia, ya que el máximo de tareas sigue siendo el mismo. Sin embargo el tiempo de ejecución debido a las comunicaciones varía. En efecto, inicialmente el término exacto que varía es:

$$\frac{C}{2}\left(M^2 - \frac{C}{2}(k_1^2 + k_2^2 + k_3^2 + \ldots)\right)$$

si ahora se hace $k_2 = k_2 + 1$ y $k_3 = k_3 - 1$, el término anterior pasa a ser:

$$\begin{aligned} &= \tfrac{C}{2}\left(M^2 - (k_1^2 + (k_2+1)^2 + (k_3-1)^2 + \ldots)\right) \\ &= \tfrac{C}{2}\left(M^2 - (k_1^2 + k_2^2 + 1 + 2k_2 + k_3^2 + 1 - 2k_3 + \ldots)\right) \\ &= \tfrac{C}{2}\left(M^2 - (k_1^2 + k_2^2 + k_3^2 + \ldots)\right) - \tfrac{C}{2}(1 + 2k_2 - 2k_3) \\ &< \tfrac{C}{2}\left(M^2 - (k_1^2 + k_2^2 + k_3^2 + \ldots)\right) \end{aligned}$$

es decir, al pasar una tarea de un procesador que tiene unas tareas k_3 a otro procesador que tiene las mismas o más tareas k_2 pero sin llegar al máximo, se reduce el tiempo de ejecución en un factor $(C/2)(1 + 2k_2 - 2k_3)$, que como $k_2 \geq k_3$ siempre será mayor que cero.

El **umbral** del factor R/C a partir del cual resulta interesante el reparto de tareas coincide con el visto para los dos procesadores y es $R/C = M/2$. Para hacer la demostración para N procesadores cualesquiera basta con igualar el tiempo de ejecución con un procesador (RM) y el tiempo de ejecución con los N procesadores:

$$RM = \frac{RM}{N} + \frac{CM^2}{2} - \frac{CM^2}{2N}$$

$$R\frac{N-1}{N} = C\frac{M}{2}\left(1 - \frac{1}{N}\right)$$

$$\frac{R}{C}\left(1-\frac{1}{N}\right)=\frac{M}{2}\left(1-\frac{1}{N}\right)$$
$$\frac{R}{C}=\frac{M}{2}$$

Resulta interesante calcular el ***speed-up*** para ver cómo influye el factor R/C en la mejora del rendimiento por el hecho de añadir procesadores al sistema. El *speed-up* es siempre la relación entre el tiempo de ejecución con un procesador y con muchos:

$$\begin{aligned} Speedup &= \frac{RM}{\frac{RM}{N}+\frac{CM^2}{2}-\frac{CM^2}{2N}} \\ &= \frac{R}{\frac{R}{N}+\frac{CM(1-1/N)}{2}} \\ &= \frac{\frac{R}{C}N}{\frac{R}{C}+\frac{M(N-1)}{2}} \end{aligned}$$

Si $\frac{R}{C} \gg \frac{M(N-1)}{2}$ entonces $Speedup \approx N$, es decir, si el tiempo debido a la sobrecarga es muy pequeño comparado con el coste de ejecución entonces el sistema es bastante eficiente puesto que el *speed-up* crece con el número de procesadores. Llega un momento en que no importa lo grande que sea el factor R/C, ya que siempre hay un número de procesadores a partir del cual este factor ya no se puede considerar pequeño y el *speed-up* deja de crecer linealmente con N.

Al igual que ocurre con la ley de Amdahl llega un momento en que añadir más procesadores aumenta muy poco el rendimiento llegándose a un límite de mejora que no se puede superar. En efecto, haciendo el límite para $N \to \infty$, se llega a que la asíntota para el *speed-up* es:

$$S_{asint} = 2\frac{R}{CM}$$

Como resumen se puede decir que la sobrecarga debida a las comunicaciones juega un gran papel en la mejora del rendimiento en un sistema. No importa el rendimiento de pico que pueda tener un sistema; si la sobrecarga debida a las comunicaciones es relativamente alta, bastarán unos pocos procesadores para obtener mejor rendimiento que con muchos procesadores. En esto, la correcta división del software, es decir, la granularidad de la aplicación, juega también un papel importante en el rendimiento final del sistema.

Tareas no uniformes

Al principio de esta sección se hacía la suposición de que todas las M tareas se ejecutaban en el mismo tiempo R. El caso habitual es que cada tarea tenga su propio tiempo de ejecución lo que puede complicar bastante el análisis del reparto de tareas.

La forma de repartir las tareas cuando cada una tiene un tiempo de ejecución diferente sería la siguiente:

1. El término debido a R se minimiza siempre que se tenga un reparto igualitario, pero la igualdad se refiere al tiempo y no al número de tareas, por lo tanto se intentará repartir las tareas de manera que el tiempo de ejecución de todos los procesadores sea el mismo. Esto implicará que algunos procesadores tengan más tareas que otros.
2. El término debido a las comunicaciones se puede minimizar realizando un reparto lo más desparejo posible. Esto significa que manteniendo la regla anterior hay que intentar agrupar las tareas de manera que unos procesadores tengan muchas y otros procesadores tengan muy pocas.
3. Las dos reglas anteriores no aseguran que se vaya a obtener un tiempo de ejecución mínimo por lo que habrá que revisar el reparto obtenido.

Esta forma de repartir tareas entre procesadores no asegura la obtención del tiempo mínimo de ejecución aunque puede llegar a acercarse bastante. Existen métodos estadísticos para obtener de forma segura repartos mejores.

2.3.3 Otras suposiciones para las comunicaciones

En la sección anterior se había supuesto que unas tareas en unos procesadores se comunicaban con otras tareas en otros procesadores y viceversa, lo que provocaba la aparición de un término debido a las comunicaciones que crecía de forma cuadrática con el número de tareas. A continuación se comentan otras suposiciones algo más optimistas y que se encuentran también en sistemas procesadores reales.

Un modelo con coste de comunicaciones lineal

En este modelo se va a suponer que las tareas de un procesador se comunican con el resto de tareas del resto de procesadores, pero en vez de suponer que cada tarea de un procesador se comunica con cada tarea del resto de procesadores, lo que se va a suponer es que cada tarea de un procesador se comunica con el resto de procesadores y no con cada tarea dentro de cada procesador; el procesador ya se encarga de difundir esta comunicación entre las tareas. De esta manera el coste de comunicaciones será proporcional al coste por tarea y al número de procesadores, siendo un coste linean con el número de tareas:

$$T_e = R \max\{k_i\} + CN \tag{2.13}$$

Aunque la fórmula es diferente a la del modelo obtenido anteriormente, se puede demostrar que aplicando los mismos criterios de reparto utilizados entonces (reparto igualitario pero intentando que sea a la vez disparejo) se obtiene el tiempo de ejecución mínimo. La diferencia es que con este modelo hay un mayor *speed-up* disponible.

En una distribución equitativa el primer término de la ejecución es aproximadamente RM/N que decrece al aumentar N. Por otro lado, el término debido a las comunicaciones (CN) crece al aumentar N, por lo que llega un momento a partir del cual el tiempo deja de disminuir para hacerse más grande. Esto quiere decir que añadir más procesadores no sólo no disminuye el tiempo de ejecución sino que lo aumenta. El tiempo de ejecución a partir del cual añadir más procesadores empeora el rendimiento es un mínimo local de la expresión (2.13), por lo que es fácil calcular el número de procesadores umbral derivando la expresión anterior con respecto a N e igualándola a cero para calcular el

mínimo:

$$-\frac{RM}{N^2} + C = 0$$

$$C = \frac{RM}{N^2}$$

dando finalmente que:

$$N_{umbral} = \sqrt{\frac{RM}{C}}$$

Esta raíz cuadrada que se obtiene es un desastre. Uno espera que M tareas puedan llevarse a cabo velozmente en $N = M$ procesadores, pero este modelo dice que debido al coste de las comunicaciones, el paralelismo efectivo se reduce a la raíz cuadrada de lo que se había previsto. Estas malas noticias se pueden mitigar con un factor R/C más alto por lo que la granularidad gruesa es mejor en este caso, aunque este efecto también se encuentra dentro de la raíz.

Estos resultados son todavía más pesimistas si se considera el coste de los procesadores extra en relación con su beneficio. Dado que el tiempo de ejcución ya no disminuye una vez alcanzado N_{umbral} se puede decir que, mucho antes de que N llegue a este umbral, se habrá alcanzado el punto donde la mejora obtenida al añadir un procesador no justifica su coste. Por ejemplo, una aplicación que en principio tiene unas 10.000 tareas se podría ejecutar como mucho en 100 procesadores para obtener el tiempo mínimo, pero sólo en unos 10 si además queremos que el sistema sea económicamente aprovechable.

El modelo presentado difiere del modelo original en el segundo término. En el modelo original el coste del segundo término crecía de forma cuadrática con la constante M. Las contribuciones al tiempo de ejecución variaban inversamente con N. Para N grande, el tiempo de ejecución se hacía del orden de $CM^2/2$ que no crece por mucho que se incremente N. Como ambos miembros de la ecuación decrecían con N el tiempo siempre decrece al aumentar el número de procesadores.

En el modelo propuesto ahora el segundo término crece con N y esto es por lo que aparece el umbral a partir del cual el rendimiento decae. Los dos modelos muestran que la penalización por sobrecarga existe y que se manifiesta limitando el uso efectivo del paralelismo. En un caso el paralelismo viene limitado por el número de tareas a ejecutar y en el otro viene limitado por el número de procesadores efectivos que son interesantes utilizar.

Un modelo optimista: comunicaciones completamente solapadas

Hasta ahora se ha supuesto que el procesador se encontraba ocupado, o bien realizando un trabajo útil R, o bien, comunicándose con el resto de procesadores y tareas. Esta suposición es cierta puesto que lo normal es que haya recursos compartidos y no se puedan hacer las dos cosas al mismo tiempo. Sin embargo, hay sistemas donde se puede considerar que mientras se está realizando la comunicación también se está llevando a cabo un trabajo útil. A continuación se propone un modelo para el caso extremo en el cual todo el coste de comunicaciones se puede llevar a cabo en paralelo con el trabajo útil.

Para este nuevo modelo se supone que si el coste debido a las comunicaciones está por debajo del trabajo útil, entonces sólo se considera el trabajo útil y no hay por tanto

ningún coste adicional por el hecho de comunicar unas tareas con otras. Esto se expresa en la siguiente ecuación:

$$T_e = \max\left\{ R\max\{k_i\}, \frac{C}{2}\sum_{i=1}^{N} k_i(M-k_i) \right\} \tag{2.14}$$

Las gráficas de la figura 2.8 pueden servir para ver el resultado de esta ecuación para dos procesadores. En esta figura aparecen las dos componentes, una debida a las comunicaciones (parábola invertida), y la otra debida al trabajo útil (líneas rectas), el máximo formado por ambos términos da el tiempo de ejecución para este modelo optimista. Las intersecciones de ambas curvas dan las situaciones donde el tiempo de ejecución es mínimo. Si no hay intersecciones porque las comunicaciones se solapan completamente con el trabajo útil entonces el mínimo se encuentra en el reparto equitativo.

Los puntos de intersección de ambos términos se dan cuando:

$$R(M-k) = C(M-k)k$$

obteniéndose entonces el reparto:

$$k = \frac{R}{C}$$

siempre que $1 \leq k \leq M/2$.

Si se sustituye esta condición en la ecuación (2.14), el tiempo de ejecución será:

$$R(M - R/C)$$

y el *speed-up* queda como:

$$1/(1 - \frac{R}{CM})$$

Como k está restringido en un rango, lo mismo le ocurrirá a R/C quedando $1 \leq R/C \leq M/2$. Para R/C dentro de este rango, el *speed-up* para dos procesadores está en el rango de 1 a 2 y es máximo cuando $R/C = M/2$ que es el mismo valor obtenido para el primer modelo de todos. Si no hay solapamiento completo entonces la distribución buena ya no es la igualitaria, aunque en realidad ésta se puede obtener haciendo R/C lo suficientemente grande.

Para N procesadores este modelo es fácil de analizar debido a los resultados siguientes. Para cualquier valor k_i máximo obtenido del tiempo de ejecución (término R), el reparto equitativo da el máximo tiempo de comunicaciones. Por lo tanto, la condición a partir de la cual se da el tiempo mínimo (reparto igualitario) será cuando coincidan el tiempo mínimo de ejecución y el máximo de comunicaciones:

$$\frac{RM}{N} = \frac{CM^2}{2}\left(1 - \frac{1}{N}\right)$$

que para N grande ocurre más o menos cuando:

$$\frac{R}{C} = \frac{M}{2}N$$

En este caso, para un tiempo total mínimo, el número de procesadores en función de R/C y M viene dado por la siguiente función:

$$N = \frac{2}{M}\frac{R}{C}$$

obteniéndose que la opción óptima para el número de procesadores es inversamente proporcional al número de tareas disponibles.

Si aumenta el paralelismo disponible (M) la mejor estrategia consiste en disminuir el número de procesadores. El decrecimiento de N con M viene del hecho de que el coste de la sobrecarga crece M veces más rápido que el tiempo de ejecución.

Un modelo con varios enlaces de comunicaciones

Una suposición común al resto de modelos expuestos hasta ahora, era que el paralelismo permite que el tiempo de ejecución (R) se solape entre los procesadores, pero las operaciones de sobrecarga (C) se realizaban de forma secuencial. Si se considera que las operaciones de sobrecarga son solamente las debidas a las comunicaciones, entonces estos modelos sirven para sistemas en los cuales sólo existe un canal de comunicación común para todos los procesadores. Este es el caso en el que todos los procesadores están conectados a un bus común o red en anillo o comparten una memoria común a la que se accede de forma exclusiva.

Es posible replicar los enlaces de comunicación (redes complejas) y otras características arquitectónicas que contribuyan al término de sobrecarga del modelo. Haciendo esto se obtiene que C ya no es constante sino que se convierte en una función de N.

Supongamos que se tiene un sistema en el cual los enlaces de intercomunicación crecen con N de manera que cada procesador tiene un enlace dedicado a cada uno del resto de procesadores. Con esta suposición las comunicaciones entre procesadores quedan solapadas unas con otras. Sin embargo, incluso con $\mathrm{O}(N^2)$ enlaces, todavía no es posible establecer más de $\mathrm{O}(N)$ conversaciones concurrentes porque cada procesador puede enviar o recibir información de un único procesador a un tiempo.

En este caso se puede dividir el segundo término de la ecuación (2.12) por N obteniéndose:

$$T_e = R \max\{k_i\} + \frac{C}{2N}\sum_{i=1}^{N} k_i(M - k_i) \tag{2.15}$$

Esta ecuación supone que un procesador está o calculando o comunicando o sin hacer nada, y que el coste total debido a las comunicaciones decrece inversamente con N, ya que pueden haber N conversaciones a un tiempo. El tiempo sin hacer nada viene en parte por el tiempo que tienen que esperar los procesadores que acaban pronto a los que les cuesta más.

Los dos términos de la ecuación (2.15) tienden a decrecer con N. Esta expresión es muy similar a la del modelo inicial de la ecuación (2.12) salvo por la aparición de N en el segundo término. Una distribución igualitaria minimiza el primer término, pero no el segundo que sigue siendo mínimo para el caso más dispar posible. Suponiendo como siempre que el reparto es igualitario, el mínimo tiempo posible será:

$$T_e = \frac{RM}{N} + \frac{CM^2}{2N}\left(1 - \frac{1}{N}\right)$$

El paralelismo es útil en este caso pero sólo hasta que el tiempo de ejecución deja de decrecer cuando se añaden nuevos procesadores. Esto quiere decir que este tiempo de ejecución alcanza un mínimo. Para calcular este mínimo se deriva T_e con respecto a N e igualamos a cero:

$$-\frac{RM}{N^2} - \frac{CM^2}{2N^2} + \frac{2CM^2}{2N^3} = 0$$

$$\frac{CM}{N} = R + \frac{CM}{2}$$

obteniéndose que:

$$N = \frac{CM}{R + \frac{CM}{2}} = \frac{2}{\frac{2R}{CM} + 1} < 2$$

Esto quiere decir que el tiempo de ejecución siempre mejora con la adición de procesadores, salvo que se tenga un procesador solamente.

Para saber si N procesadores dan mejor tiempo que un único procesador hay que igualar los tiempos de un procesador con los de varios:

$$RM = \frac{RM}{N} + \frac{CM^2}{2N}\left(1 - \frac{1}{N}\right)$$

Simplificando se obtiene que el punto a partir del cual es menor el tiempo con N procesadores se da cuando se cumple:

$$\frac{R}{C} = \frac{M}{2N}$$

En este caso el factor de granularidad R/C y N están inversamente relacionados en el umbral. Por lo tanto, cuanto más grande sea N menor granularidad se puede permitir. En el umbral la máquina paralela tiene un coste realmente alto, por un lado no se gana nada en tiempo y, por otro, el número de procesadores es del orden de N y el número de enlaces es del orden de N^2.

La conclusión de este modelo es que añadiendo enlaces al sistema (aumentando el ancho de banda de las comunicaciones) se puede permitir una granularidad menor que en otros casos. Sin embargo, esta menor granularidad genera un coste que crece más rápido que sólo el incremento del coste de procesamiento. La decisión de si el aumento de velocidad obtenido por el aumento del ancho de banda vale la pena o no, depende fuertemente de la tecnología utilizada para las comunicaciones entre procesadores.

Resumen de los modelos presentados

A continuación se resumen los hallazgos encontrados a partir de los modelos presentados:

1. Las arquitecturas multiprocesador producen un coste por sobrecarga adicional que no está presente en los mono-procesadores, procesadores vectoriales, u otros tipos de procesadores donde hay un único flujo de instrucciones. El coste por sobrecarga incluye el coste de preparación de tareas, contención en los recursos compartidos, sincronización, y comunicaciones entre procesadores.
2. Aunque el tiempo de ejecución para un trozo de programa tiende a disminuir con el número de procesadores que trabajan en ese trozo de programa. El coste por sobrecarga tiende a crecer con el número de procesadores. De hecho, es posible que el coste de la sobrecarga más rápido que lineal en el número de procesadores.

3. La relación R/C es una medida de la cantidad de ejecución de programa (tiempo de ejecución útil) por unidad de sobrecarga (tiempo de comunicaciones), dentro de la implementación de un programa en una arquitectura específica. Cuando más grande sea esta relación más eficiente será la computación, ya que una porción pequeña del tiempo está dedicada a la sobrecarga. Sin embargo, si la relación R/C se hace grande al particionar el cálculo en pocos trozos grandes en vez de muchos trozos pequeños, el paralelismo disponible se reduce enormemente, lo que limita la mejora que se puede obtener de un multiprocesador.

Con esto aparece un dilema claro: por un lado R/C debe ser pequeño para poder tener un gran número de tareas potencialmente paralelas, y por otro lado, R/C debe ser grande para evitar los costes de sobrecarga. Debido a esto no se puede esperar tener un sistema de alto rendimiento sin más que construir el multiprocesador con el mayor número de procesadores posible permitido por la tecnología.

Existe algún número máximo de procesadores por debajo del cual es coste está justificado, y este número máximo depende mayormente de la arquitectura del sistema, la tecnología utilizada (especialmente comunicaciones), y de las características de la aplicación específica que se tenga que ejecutar.

Capítulo 3

Procesadores vectoriales

En el camino hacia los multiprocesadores y multicomputadores nos encontramos con los procesadores vectoriales que son una forma también de procesamiento paralelo.

Normalmente el cálculo científico y matemático precisa de la realización de un número elevado de operaciones en muy poco tiempo. La mayoría de los problemas físicos y matemáticos se pueden expresar fácilmente mediante la utilización de matrices y vectores. Aparte de que esto supone una posible claridad en el lenguaje, va a permitir explotar al máximo un tipo de arquitectura específica para este tipo de tipos de datos, y es la de los procesadores vectoriales.

El paralelismo viene de que al operar con matrices, normalmente, los elementos de las matrices son independientes entre sí, es decir, no existen dependencias de datos dentro de las propias matrices, en general. Esto permite que todas las operaciones entre elementos de unas matrices con otras puedan realizarse en paralelo, o al menos en el mismo cauce de instrucciones sin que haya un conflicto entre los datos.

Otra ventaja del cálculo matricial es que va a permitir replicar las unidades de cálculo sin necesidad de replicar las unidades de control. Se tendría en este caso una especie de multiprocesador sin necesidad de tener que replicar tanto la unidad de control como la de cálculo, eso sí, el número de tareas que un sistema de este tipo podría abordar son limitadas.

Los procesadores vectoriales se caracterizan porque van a ofrecer una serie de operaciones de alto nivel que operan sobre vectores, es decir, matrices lineales de números. Una operación típica de un procesador vectorial sería la suma de dos vectores de coma flotante de 64 elementos para obtener el vector de 64 elementos resultante. La instrucción en este caso es equivalente a un lazo software que a cada iteración opera sobre uno de los 64 elementos. Un procesador vectorial realiza este lazo por hardware aprovechando un cauce más profundo, la localidad de los datos, y una eventual repetición de las unidades de cálculo.

Las instrucciones vectoriales tienen unas propiedades importantes que se resumen a continuación aunque previamente ya se han dado unas pinceladas:

- El cálculo de cada resultado es independiente de los resultados anteriores en el mismo vector, lo que permite un cauce muy profundo sin generar *riesgos* por las dependencias de datos. La ausencia de estos riesgos viene decidida por el compilador o el programador cuando se decidió que la instrucción podía ser utilizada.
- Una sola instrucción vectorial especifica una gran cantidad de trabajo, ya que equi-

vale a ejecutar un bucle completo. Por lo tanto, el requisito de anchura de banda de las instrucciones se reduce considerablemente. En los procesadores no vectoriales, donde se precisan muchas más instrucciones, la búsqueda y decodificación de las instrucciones puede representar un cuello de botella, que fue detectado por Flynn en 1966 y por eso se le llama *cuello de botella de Flynn.*

- Las instrucciones vectoriales que acceden a memoria tienen un patrón de acceso conocido. Si los elementos de la matriz son todos adyacentes, entonces extraer el vector de un conjunto de bancos de memoria entrelazada funciona muy bien. La alta latencia de iniciar un acceso a memoria principal, en comparación con acceder a una cache, se amortiza porque se inicia un acceso para el vector completo en lugar de para un único elemento. Por ello, el coste de la latencia a memoria principal se paga una sola vez para el vector completo, en lugar de una vez por cada elemento del vector.
- Como se sustituye un bucle completo por una instrucción vectorial cuyo comportamiento está predeterminado, los riesgos de control en el cauce, que normalmente podrían surgir del salto del bucle, son inexistentes.

Por estas razones, las operaciones vectoriales pueden hacerse más rápidas que una secuencia de operaciones escalares sobre el mismo número de elementos de datos, y los diseñadores están motivados para incluir unidades vectoriales si el conjunto de las aplicaciones las puede usar frecuentemente.

El presente capítulo ha sido elaborado a partir de [HP96], [HP93] y [Hwa93]. Como lecturas adicionales se puede ampliar la información con [Sto93] y [HB87].

3.1 Procesador vectorial básico

3.1.1 Arquitectura vectorial básica

Un procesador vectorial está compuesto típicamente por una unidad escalar y una unidad vectorial. La parte vectorial permite que los vectores sean tratados como números en coma flotante, como enteros o como datos lógicos. La unidad escalar es un procesador segmentado normal y corriente.

Hay dos tipos de arquitecturas vectoriales:

Máquina vectorial con registros: en una máquina de este tipo, todas las operaciones vectoriales, excepto las de carga y almacenamiento, operan con vectores almacenados en registros. Estas máquinas son el equivalente vectorial de una arquitectura escalar de carga/almacenamiento. La mayoría de máquinas vectoriales modernas utilizan este tipo de arquitectura. Ejemplos: Cray Research (CRAY-1, CRAY-2, X-MP, Y-MP y C-90), los supercomputadores japoneses (NEC SX/2 y SX/3, las Fujitsu VP200 y VP400 y la Hitachi S820)

Máquina vectorial memoria-memoria: en estas máquinas, todas las operaciones vectoriales son de memoria a memoria. Como la complejidad interna, así como el coste, son menores, es la primera arquitectura vectorial que se empleó. Ejemplo: el CDC.

El resto del capítulo trata sobre las máquinas vectoriales con registros, ya que las de memoria han caído en desuso por su menor rendimiento.

La figura 3.1 muestra la arquitectura típica de una máquina vectorial con registros. Los registros se utilizan para almacenar los operandos. Los cauces vectoriales funcionales cogen los operandos, y dejan los resultados, en los registros vectoriales. Cada registro vectorial está equipado con un contador de componente que lleva el seguimiento del componente de los registros en ciclos sucesivos del cauce.

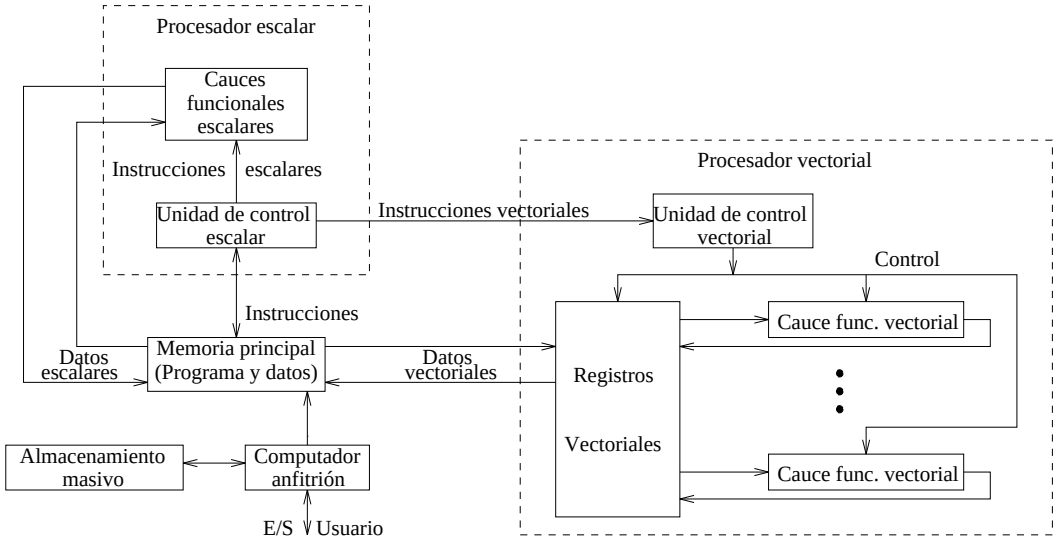


Figura 3.1: La arquitectura de un supercomputador vectorial.

La longitud de cada registro es habitualmente fija, como por ejemplo 64 componentes de 64 bits cada uno como en un Cray. Otras máquinas, como algunas de Fujitsu, utilizan registros vectoriales reconfigurables para encajar dinámicamente la longitud del registro con la longitud de los operandos.

Por lo general, el número de registros vectoriales y el de unidades funcionales es fijo en un procesador vectorial. Por lo tanto, ambos recursos deben reservarse con antelación para evitar conflictos entre diferentes operaciones vectoriales.

Los supercomputadores vectoriales empezaron con modelos uniprocesadores como el Cray 1 en 1976. Los supercomputadores vectoriales recientes ofrecen ambos modelos, el monoprocesador y el multiprocesador. La mayoría de supercomputadores de altas prestaciones modernos ofrecen multiprocesadores con hardware vectorial como una característica más de los equipos.

Resulta interesante definirse una arquitectura vectorial sobre la que explicar las nociones de arquitecturas vectoriales. Esta arquitectura tendría como parte entera la propia del DLX, y su parte vectorial sería la extensión vectorial lógica de DLX. Los componentes básicos de esta arquitectura, parecida a la de Cray 1, se muestra en la figura 3.2.

Los componentes principales del conjunto de instrucciones de la máquina DLXV son:

Registros vectoriales. Cada registro vectorial es un banco de longitud fija que contiene un solo vector. DLXV tiene ocho registros vectoriales, y cada registro vectorial contiene 64 dobles palabras. Cada registro vectorial debe tener como mínimo dos puertos de lectura y uno de escritura en DLXV. Esto permite un alto grado de solapamiento entre las operaciones vectoriales que usan diferentes registros vectoriales.

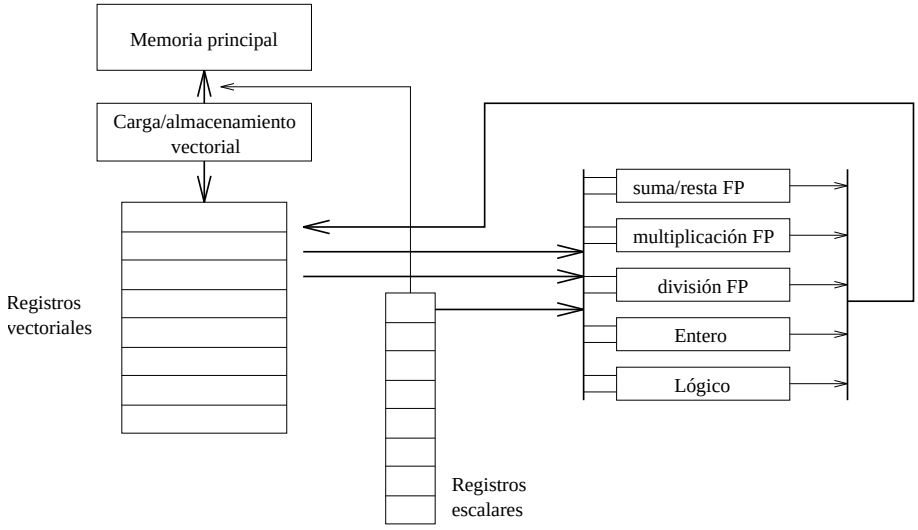


Figura 3.2: Estructura básica de una arquitectura vectorial con registros, DLXV.

Unidades funcionales vectoriales. Cada unidad se encuentra completamente segmentada y puede comenzar una nueva operación a cada ciclo de reloj. Se necesita una unidad de control para detectar conflictos en las unidades funcionales (riesgos estructurales) y conflictos en los accesos a registros (riesgos por dependencias de datos).

Unidad de carga/almacenamiento de vectores. Es una unidad que carga o almacena un vector en o desde la memoria. Las cargas y almacenamientos en DLXV están completamente segmentadas, para que las palabras puedan ser transferidas entre los registros vectoriales y memoria, con un ancho de banda de una palabra por ciclo de reloj tras una latencia inicial.

Conjunto de registros escalares. Estos también pueden proporcionar datos como entradas a las unidades funcionales vectoriales, así como calcular direcciones para pasar a la unidad de carga/almacenamiento de vectores. Estos serían los 32 registros normales de propósito general y los 32 registros de punto flotante del DLX.

La figura 3.3 muestra las características de algunos procesadores vectoriales, incluyendo el tamaño y el número de registros, el número y tipo de unidades funcionales, y el número de unidades de carga/almacenamiento.

3.1.2 Instrucciones vectoriales básicas

Principalmente las instrucciones vectoriales se pueden dividir en seis tipos diferentes. El criterio de división viene dado por el tipo de operandos y resultados de las diferentes instrucciones vectoriales:

1. **Vector-vector:** Las instrucciones *vector-vector* son aquellas cuyos operandos son vectores y el resultado también es un vector. Suponiendo que V_i, V_j y V_k son registros vectoriales, este tipo de instrucciones implementan el siguiente tipo de funciones:

$$f_1 : V_i \rightarrow V_j$$

Processor	Year announced	Clock rate (MHz)	Registers	Elements per register (64-bit elements)	Functional units	Load-store units
CRAY-1	1976	80	8	64	6: add, multiply, reciprocal, integer add, logical, shift	1
CRAY X-MP CRAY Y-MP	1983 1988	120 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store
CRAY-2	1985	166	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer (add shift, population count), logical	1
Fujitsu VP100/200	1982	133	8-256	32-1024	3: FP or integer add/logical, multiply, divide	2
Hitachi S810/820	1983	71	32	256	4: 2 integer add/logical, 1 multiply-add, and 1 multiply/divide-add unit	4
Convex C-1	1985	10	8	128	4: multiply, add, divide, integer/logical	1
NEC SX/2	1984	160	8 + 8192	256 variable	16: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, 4 shift	8
DLXV	1990	200	8	64	5: multiply, divide, add, integer add, logical	1
Cray C-90	1991	240	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	4
Convex C-4	1994	135	16	128	3: each is full integer, logical, and FP (including multiply-add)	
NEC SX/4	1995	400	8 + 8192	256 variable	16: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, 4 shift	8
Cray J-90	1995	100	8	64	4: FP add, FP multiply, FP reciprocal, integer/logical	
Cray T-90	1996	~500	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	4

Figura 3.3: Características de varias arquitecturas vectoriales.

$$f_2 : V_j \times V_k \rightarrow V_i$$

Algunos ejemplos son: $V_1 = \sin(V_2)$ o $V_3 = V_1 + V_2$.

2. **Vector-escalar:** Son instrucciones en cuyos operandos interviene algún escalar y el resultado es un vector. Si s es un escalar son las instrucciones que siguen el siguiente tipo de función:

$$f_3 : d \times V_i \rightarrow V_j$$

Un ejemplo es el producto escalar, que el resultado es un vector tras multiplicar el vector origen por el escalar elemento a elemento.

3. **Vector-memoria:** Suponiendo que M es el comienzo de un vector en memoria, se tienen las instrucciones de carga y almacenamiento de un vector:

$$f_4 : M \rightarrow V \qquad \text{Carga del vector}$$
$$f_5 : V \rightarrow M \qquad \text{Almacenamiento del vector}$$

4. **Reducción de vectores:** Son instrucciones cuyos operandos son vectores y el resultado es un escalar, por eso se llaman de reducción. Los tipos de funciones que describen estas instrucciones son los siguientes:

$$f_6 : V_i \rightarrow s_j$$

$$f_7 : V_i \times V_j \rightarrow s_k$$

El máximo, la suma, la media, etc., son ejemplos de f_6, mientras que el producto punto ($s = \sum_{i=1}^{n} a_i \times b_i$) es un ejemplo de f_7.

5. **Reunir y Esparcir:** Estas funciones sirven para almacenar/cargar vectores dispersos en memoria. Se necesitan dos vectores para reunir o esparcir el vector de/a la memoria. Estas son las funciones para reunir y esparcir:

$$\begin{array}{ll} f_8 : M \rightarrow V_1 \times V_0 & \text{Reunir} \\ f_9 : V_1 \times V_0 \rightarrow M & \text{Esparcir} \end{array}$$

La operación *reunir* toma de la memoria los elementos no nulos de un vector disperso usando unos índices. La operación *esparcir* hace lo contrario, almacena en la memoria un vector en un vector disperso cuyas entradas no nulas están indexadas. El registro vectorial V_1 contiene los datos y el V_0 los índices de los elementos no nulos.

6. **Enmascaramiento:** En estas instrucciones se utiliza un vector de *máscara* para comprimir o expandir un vector a un vector índice. La aplicación que tiene lugar es la siguiente:

$$f_{10} : V_0 \times V_m \rightarrow V_1$$

3.1.3 Ensamblador vectorial DLXV

En DLXV las operaciones vectoriales usan los mismos mnemotécnicos que las operaciones DLX, pero añadiendo la letra `V` (`ADDV`). Si una de las entradas es un escalar se indicará añadiendo el sufijo "SV" (`ADDSV`). La figura 3.4 muestra las instrucciones vectoriales del DLXV.

Ejemplo: el bucle DAXPY

Existe un bucle típico para evaluar sistemas vectoriales y multiprocesadores que consiste en realizar la operación:

$$Y = a \cdot X + Y$$

donde X e Y son vectores que residen inicialmente en memoria, mientras que a es un escalar. A este bucle, que es bastante conocido, se le llama SAXPY o DAXPY dependiendo de si la operación se realiza en simple o doble precisión. A esta operación nos referiremos a la hora de hacer cálculos de rendimiento y poner ejemplos. Estos bucles forman el bucle interno del benchmark Linpack. (SAXPY viene de *single-precision* $a \times X$ *plus* Y; DAXPY viene de *double-precision* $a \times X$ *plus* Y.) Linpack es un conjunto de rutinas de álgebra lineal, y rutinas para realizar el método de eliminación de Gauss.

Para los ejemplos que siguen vamos a suponer que el número de elementos, o longitud, de un registro vectorial coincide con la longitud de la operación vectorial en la que estamos interesados. Más adelante se estudiará el caso en que esto no sea así.

Resulta interesante, para las explicaciones que siguen, dar los programas en ensamblador para realizar el cálculo del bucle DAXPY. El siguiente programa sería el código escalar utilizando el juego de instrucciones DLX:

Instruction	Operands	Function
ADDV ADDSV	V1,V2,V3 V1,F0,V2	Add elements of V2 and V3, then put each result in V1. Add F0 to each element of V2, then put each result in V1.
SUBV SUBVS SUBSV	V1,V2,V3 V1,V2,F0 V1,F0,V2	Subtract elements of V3 from V2, then put each result in V1. Subtract F0 from elements of V2, then put each result in V1. Subtract elements of V2 from F0, then put each result in V1.
MULTV MULTSV	V1,V2,V3 V1,F0,V2	Multiply elements of V2 and V3, then put each result in V1. Multiply F0 by each element of V2, then put each result in V1.
DIVV DIVVS DIVSV	V1,V2,V3 V1,V2,F0 V1,F0,V2	Divide elements of V2 by V3, then put each result in V1. Divide elements of V2 by F0, then put each result in V1. Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1, ..., 63 × R1 into V1.
S--V S--SV	V1,V2 F0,V1	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--SV performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MOVI2S MOVS2I	VLR,R1 R1,VLR	Move contents of R1 to the vector-length register. Move the contents of the vector-length register to R1.
MOVF2S MOVS2F	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

Figura 3.4: Instrucciones vectoriales del DLXV.

```
     LD    F0,a
     ADDI  R4,Rx,#512   ; última dirección a cargar
loop:
     LD    F2,0(Rx)     ; carga X(i) en un registro
     MULTD F2,F0,F2     ; a.X(i)
     LD    F4,0(Ry)     ; carga Y(i) en un registro
     ADDD  F4,F2,F4     ; a.X(i)+Y(i)
     SD    0(Ry),F4     ; almacena resultado en Y(i)
     ADDI  Rx,Rx,#8     ; incrementa el índice de X
     ADDI  Ry,Ry,#8     ; incrementa el índice de Y
     SUB   R20,R4,Rx    ; calcula límite
     BNZ   R20,loop     ; comprueba si fin.
```

El programa correspondiente en una arquitectura vectorial, como la DLXV, sería de la siguiente forma:

```
     LD     F0,a        ; carga escalar a
     LV     V1,Rx       ; carga vector X
     MULTSV V2,F0,V1    ; a*X(i)
     LV     V3,Ry       ; carga vector Y
     ADDV   V4,V2,V3    ; suma
     SV     Ry,V4       ; almacena el resultado
```

De los códigos anteriores se desprenden dos cosas. Por un lado la máquina vectorial reduce considerablemente el número de instrucciones a ejecutar, ya que se requieren

sólo 6 frente a casi las 600 del bucle escalar. Por otro lado, en la ejecución escalar, debe bloquearse la suma, ya que comparte datos con la multiplicación previa; en la ejecución vectorial, tanto la multiplicación como la suma son independientes y, por tanto, no se bloquea el cauce durante la ejecución de cada instrucción sino entre una instrucción y la otra, es decir, una sola vez. Estos bloqueos se pueden eliminar utilizando segmentación software o desarrollando el bucle, sin embargo, el ancho de banda de las instrucciones será mucho más alto sin posibilidad de reducirlo.

3.1.4 Tiempo de ejecución vectorial

Tres son los factores que influyen en el tiempo de ejecución de una secuencia de operaciones vectoriales:

- La longitud de los vectores sobre los que se opera.
- Los riesgos estructurales entre las operaciones.
- Las dependencias de datos.

Dada la longitud del vector y la *velocidad de inicialización*, que es la velocidad a la cual una unidad vectorial consume nuevos operandos y produce nuevos resultados, podemos calcular el tiempo para una instrucción vectorial. Lo normal es que esta velocidad sea de uno por ciclo del reloj. Sin embargo, algunos supercomputadores producen 2 o más resultados por ciclo de reloj, y otros, tienen unidades que pueden no estar completamente segmentadas. Por simplicidad se supondrá que esta velocidad es efectivamente la unidad.

Para simplificar la discusión del tiempo de ejecución se introduce la noción de *convoy*, que es el conjunto de instrucciones vectoriales que podrían potencialmente iniciar su ejecución en el mismo ciclo de reloj. Las instrucciones en un convoy no deben incluir ni riesgos estructurales ni de datos (aunque esto se puede relajar más adelante); si estos riesgos estuvieran presentes, las instrucciones potenciales en el convoy habría que serializarlas e inicializarlas en convoyes diferentes. Para simplificar diremos que las instrucciones de un convoy deben terminar de ejecutarse antes que cualquier otra instrucción, vectorial o escalar, pueda empezar a ejecutarse. Esto se puede relajar utilizando un método más complejo de lanzar instrucciones.

Junto con la noción de convoy está la de *toque* o *campanada* (*chime*) que puede ser usado para evaluar el rendimiento de una secuencia de vectores formada por convoyes. Un toque o campanada es una medida aproximada del tiempo de ejecución para una secuencia de vectores; la medida de la campanada es independiente de la longitud del vector. Por tanto, para una secuencia de vectores que consiste en m convoyes se ejecuta en m campanadas, y para una longitud de vector de n, será aproximadamente $n \times m$ ciclos de reloj. Esta aproximación ignora algunas sobrecargas sobre el procesador que además dependen de la longitud del vector. Por consiguiente, la medida del tiempo en campanadas es una mejor aproximación para vectores largos. Se usará esta medida, en vez de los periodos de reloj, para indicar explícitamente que ciertas sobrecargas están siendo ignoradas.

Para poner las cosas un poco más claras, analicemos el siguiente código y extraigamos de él los convoyes:

```
LD     F0,a        ; carga el escalar en F0
LV     V1,Rx       ; carga vector X
MULTSV V2,F0,V1    ; multiplicación vector-escalar
```

```
LV     V3,Ry       ; carga vector Y
ADDV   V4,V2,V3    ; suma vectorial
SV     Ry,V4       ; almacena el resultado.
```

Dejando de lado la primera instrucción, que es puramente escalar, el primer convoy lo ocupa la primera instrucción vectorial que es `LV`. La `MULTSV` depende de la primera por eso no puede ir en el primer convoy, en cambio, la siguiente `LV` sí que puede. `ADDV` depende de esta `LV` por tanto tendrá que ir en otro convoy, y `SV` depende de esta, así que tendrá que ir en otro convoy aparte. Los convoyes serán por tanto:

1. `LV`
2. `MULTSV  LV`
3. `ADDV`
4. `SV`

Como esta secuencia está formada por 4 convoyes requerirá 4 campanadas para su ejecución.

Esta aproximación es buena para vectores largos. Por ejemplo, para vectores de 64 elementos, el tiempo en campanadas sería de 4, de manera que la secuencia tomaría unos 256 ciclos de reloj. La sobrecarga de eventualmente lanzar el convoy 2 en dos ciclos diferentes sería pequeña en comparación a 256.

Tiempo de arranque vectorial y tasa de inicialización

La fuente de sobrecarga más importante, no considerada en el modelo de campanadas, es el *tiempo de arranque* vectorial. El tiempo de arranque viene de la latencia del cauce de la operación vectorial y está determinada principalmente por la profundidad del cauce en relación con la unidad funcional empleada. El tiempo de arranque incrementa el tiempo efectivo en ejecutar un convoy en más de una campanada. Además, este tiempo de arranque retrasa la ejecución de convoyes sucesivos. Por lo tanto, el tiempo necesario para la ejecución de un convoy viene dado por el tiempo de arranque y la longitud del vector. Si la longitud del vector tendiera a infinito, entonces el tiempo de arranque sería despreciable, pero lo normal es que el tiempo de arranque sea de 6 a 12 ciclos, lo que significa un porcentaje alto en vectores típicos que como mucho rondarán los 64 elementos o ciclos.

Operation	**Start-up penalty**
Vector add	6
Vector multiply	7
Vector divide	20
Vector load	12

Figura 3.5: Penalización por el tiempo de arranque en el DLXV.

Siguiendo con el ejemplo mostrado anteriormente, supongamos que cargar y salvar tienen un tiempo de arranque de 12 ciclos, la multiplicación 7 y la suma 6 tal y como se desprende de la figura 3.5. Las sumas de los arranques de cada convoy para este ejemplo sería 12+12+6+12=42, como estamos calculando el número de campanadas *reales* para

vectores de 64 elementos, la división 42/64=0.65 da el número de campanadas totales, que será entonces 4.65, es decir, el tiempo de ejecución teniendo en cuenta la sobrecarga de arranque es 1.16 veces mayor. En la figura 3.6 se muestra el tiempo en que se inicia cada instrucción así como el tiempo total para su ejecución en función de n que es el número de elementos del vector.

Convoy	Starting time	First-result time	Last-result time
1. `LV`	0	12	$11 + n$
2. `MULTSV LV`	$12 + n$	$12 + n + 12$	$23 + 2n$
3. `ADDV`	$24 + 2n$	$24 + 2n + 6$	$29 + 3n$
4. `SV`	$30 + 3n$	$30 + 3n + 12$	$41 + 4n$

Figura 3.6: Tiempos de arranque y del primer y último resultados para los convoys 1-4.

El tiempo de arranque de una instrucción es típicamente la profundidad del cauce de la unidad funcional que realiza dicha instrucción. Si lo que se quiere es poder lanzar una instrucción por ciclo de reloj (tasa de inicialización igual a uno), entonces

$$\text{Profundidad del cauce} = \left\lceil \frac{\text{Tiempo total de ejecución de la unidad}}{\text{Periodo de reloj}} \right\rceil \tag{3.1}$$

Por ejemplo, si una operación necesita 10 ciclos de reloj para completarse, entonces hace falta un cauce con una profundidad de 10 para que se pueda inicializar una instrucción por cada ciclo de reloj. Las profundidades de las unidades funcionales varían ampliamente (no es raro ver cauces de profundidad 20) aunque lo normal es que tengan profundidades entre 4 y 8 ciclos.

3.1.5 Unidades de carga/almacenamiento vectorial

El comportamiento de la unidad de carga/almacenamiento es más complicado que el de las unidades aritméticas. El tiempo de arranque para una carga es el tiempo para coger la primera palabra de la memoria y guardarla en un registro. Si el resto del vector se puede coger sin paradas, la tasa de inicialización es la misma que la velocidad a la que las nuevas palabras son traídas y almacenadas. Al contrario que en las unidades funcionales, la tasa de inicialización puede no ser necesariamente una instrucción por ciclo.

Normalmente, el tiempo de arranque para las unidades de carga/almacenamiento es algo mayor que para las unidades funcionales, pudiendo llegar hasta los 50 ciclos. Típicamente, estos valores rondan entre los 9 y los 17 ciclos (Cray 1 y Cray X-MP)

Para conseguir una tasa (o velocidad) de inicialización de una palabra por ciclo, el sistema de memoria debe ser capaz de producir o aceptar esta cantidad de datos. Esto se puede conseguir mediante la creación de bancos de memoria múltiples como se explica en la sección 3.2. Teniendo un número significativo de bancos se puede conseguir acceder a la memoria por filas o por columnas de datos.

El número de bancos en la memoria del sistema para las unidades de carga y almacenamiento, así como la profundidad del cauce en unidades funcionales son de alguna

manera equivalentes, ya que ambas determinan las tasas de inicialización de las operaciones utilizando estas unidades. El procesador no puede acceder a un banco de memoria más deprisa que en un ciclo de reloj. Para los sistemas de memoria que soportan múltiples accesos vectoriales simultáneos o que permiten accesos no secuenciales en la carga o almacenamiento de vectores, el número de bancos de memoria debería ser más grande que el mínimo, de otra manera existirían conflictos en los bancos.

3.2 Memoria entrelazada o intercalada

La mayor parte de esta sección se encuentra en el [Hwa93], aunque se puede encontrar una pequeña parte en el [HP96] en el capítulo dedicado a los procesadores vectoriales.

Para poder salvar el salto de velocidad entre la CPU/caché y la memoria principal realizada con módulos de RAM, se presenta una técnica de entrelazado que permite el acceso segmentado a los diferentes módulos de memoria paralelos.

Vamos a suponer que la memoria principal se encuentra construida a partir de varios módulos. Estos módulos de memoria se encuentran normalmente conectados a un bus del sistema, o a una red de conmutadores, a la cual se conectan otros dispositivos del sistema como procesadores o subsistemas de entrada/salida.

Cuando se presenta una dirección en un módulo de memoria esta devuelve la palabra correspondiente. Es posible presentar diferentes direcciones a diferentes módulos de memoria de manera que se puede realizar un acceso paralelo a diferentes palabras de memoria. Ambos tipos de acceso, el paralelo y el segmentado, son formas paralelas practicadas en una organización de memoria paralela.

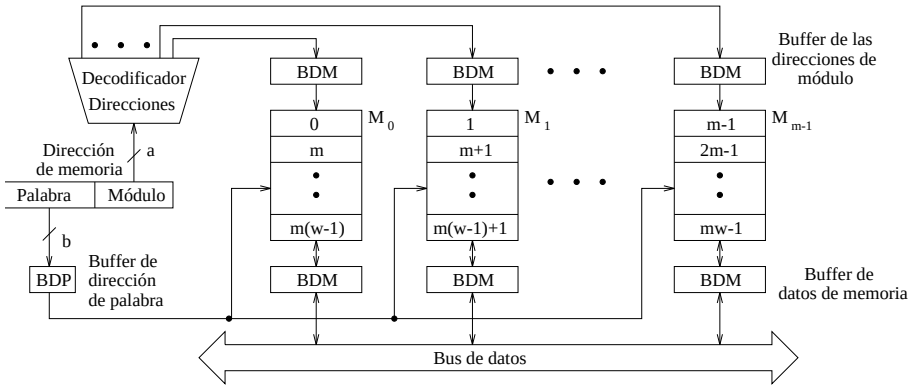
Consideremos una memoria principal formada por $m = 2^a$ módulos, cada uno con $w = 2^b$ palabras o celdas de memoria. La capacidad total de la memoria es $m\,w = 2^{a+b}$ palabras. A estas palabras se les asignan direcciones de forma lineal. Las diferentes formas en las que se asignan linealmente las direcciones producen diferentes formas de organizar la memoria.

Aparte de los accesos aleatorios, la memoria principal es accedida habitualmente mediante bloques de direcciones consecutivas. Los accesos en bloque son necesarios para traerse una secuencia de instrucciones o para acceder a una estructura lineal de datos, etc. En un sistema basado en cache la longitud del bloque suele corresponderse con la longitud de una línea en la caché, o con varias líneas de caché. También en los procesadores vectoriales el acceso a la memoria tiene habitualmente una estructura lineal, ya que los elementos de los vectores se encuentran consecutivos. Por todo esto, resulta preferible diseñar la memoria principal para facilitar el acceso en bloque a palabras contiguas.

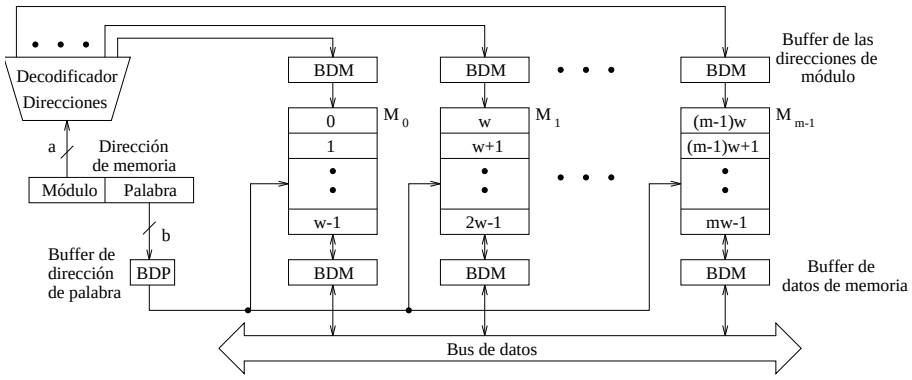
La figura 3.7 muestra dos formatos de direcciones para realizar una memoria entrelazada. El *entrelazado de orden bajo* (figura 3.7(a)) reparte las localizaciones contiguas de memoria entre los m módulos de forma horizontal. Esto implica que los a bits de orden bajo de la dirección se utilizan para identificar el módulo de memoria. Los b bits de orden más alto forman la dirección de la palabra dentro de cada módulo. Hay que hacer notar que la misma dirección de palabra está siendo aplicada a todos los módulos de forma simultánea. Un decodificador de direcciones se emplea para distribuir la selección de los módulos. Este esquema no es bueno para tolerancia a fallos, ya que en caso de fallo de un módulo toda la memoria queda inutilizable.

El *entrelazado de orden alto* (figura 3.7(b)) utiliza los a bits de orden alto como selector de módulo, y los b bits de orden bajo como la dirección de la palabra dentro de cada módulo. Localizaciones contiguas en la memoria están asignadas por tanto a un mismo módulo de memoria. En un ciclo de memoria, sólo se accede a una palabra del módulo. Por lo tanto, el entrelazado de orden alto no permite el acceso en bloque a posiciones contiguas de memoria. Este esquema viene muy bien para tolerancia a fallos.

Por otro lado, el entrelazado de orden bajo soporta el acceso de bloque de forma segmentada. A no ser que se diga otra cosa, se supondrá para el resto del capítulo que la memoria es del tipo entrelazado de orden bajo.



(a) Memoria de m vías entrelazada de orden bajo (esquema C de acceso a memoria).



(b) Memoria de m vías entrelazada de orden alto.

Figura 3.7: Dos organizaciones de memoria entrelazada con $m = 2^a$ módulos y $w = 2^a$ palabras por módulo.

Ejemplo de memoria modular en un procesador vectorial

Supongamos que queremos captar un vector de 64 elementos que empieza en la dirección 136, y que un acceso a memoria supone 6 ciclos de reloj. ¿Cuántos bancos de memoria debemos tener para acceder a cada elemento en un único ciclo de reloj? ¿Con qué dirección se accede a estos bancos? ¿Cuándo llegarán los elementos a la CPU?.

Respuesta Con seis ciclos por acceso, necesitamos al menos seis bancos de memoria, pero como queremos que el número de bancos sea potencia de dos, elegiremos ocho bancos. La figura 3.1 muestra las direcciones a las que se accede en cada banco en cada periodo de tiempo.

Beginning at clock no.	**Bank**							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	192	136	144	152	160	168	176	184
6	256	200	208	216	224	232	240	248
14	320	264	272	280	288	296	304	312
22	384	328	336	344	352	360	368	376

Tabla 3.1: Direcciones de memoria (en bytes) y momento en el cual comienza el acceso a cada banco.

La figura 3.8 muestra la temporización de los primeros accesos a un sistema de ocho bancos con una latencia de acceso de seis ciclos de reloj. Existen dos observaciones importantes con respecto a la tabla 3.1 y la figura 3.8: La primera es que la dirección exacta proporcionada por un banco está muy determinada por los bits de menor orden; sin embargo, el acceso inicial a un banco está siempre entre las ocho primeras dobles palabras de la dirección inicial. La segunda es que una vez se ha producido la latencia inicial (seis ciclos en este caso), el patrón es acceder a un banco cada n ciclos, donde n es el número total de bancos ($n = 8$ en este caso).

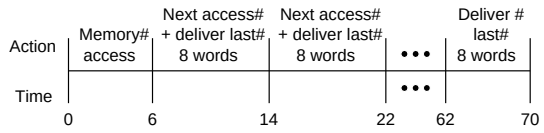


Figura 3.8: Tiempo de acceso para las primeras 64 palabras de doble precisión en una lectura.

3.2.1 Acceso concurrente a memoria (acceso C)

Los accesos a los m módulos de una memoria se pueden solapar de forma segmentada. Para esto, el ciclo de memoria (llamado *ciclo mayor de memoria* se subdivide en m *ciclos menores*.

Sea θ el tiempo para la ejecución del ciclo mayor y τ para el menor. Estos dos tiempos se relacionan de la siguiente manera:

$$\tau = \frac{\theta}{m} \tag{3.2}$$

donde m es el *grado de entrelazado*. La temporización del acceso segmentado de 8 palabras contiguas en memoria se muestra en la figura 3.9. A este tipo de *acceso concurrente* a palabras contiguas se le llama *acceso C* a memoria. El ciclo mayor θ es el tiempo total necesario para completar el acceso a una palabra simple de un módulo. El ciclo

menor τ es el tiempo necesario para producir una palabra asumiendo la superposición de accesos de módulos de memoria sucesivos separados un ciclo menor τ.

Hay que hacer notar que el acceso segmentado al bloque de 8 palabras contiguas está emparedado entre otros accesos de bloque segmentados antes y después del bloque actual. Incluso a pesar de que el tiempo total de acceso del bloque es 2θ, el *tiempo efectivo de acceso* de cada palabra es solamente τ al ser la memoria contiguamente accedida de forma segmentada.

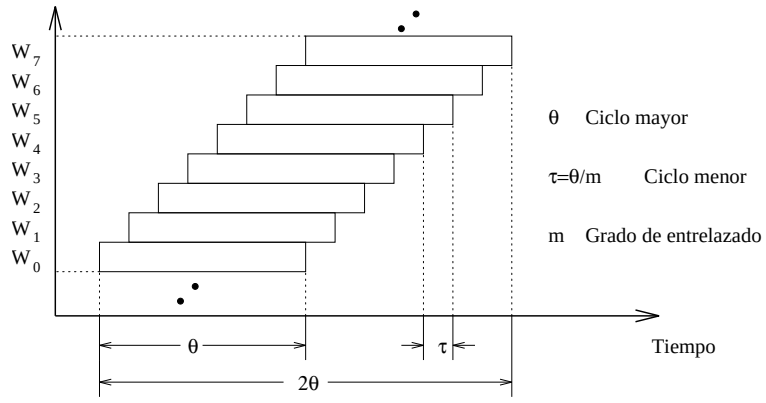


Figura 3.9: Acceso segmentado a 8 palabras contiguas en una memoria de acceso C.

3.2.2 Acceso simultáneo a memoria (acceso S)

La memoria entrelazada de orden bajo puede ser dispuesta de manera que permita *accesos simultáneos*, o *accesos S*, tal y como se muestra en la figura 3.10.

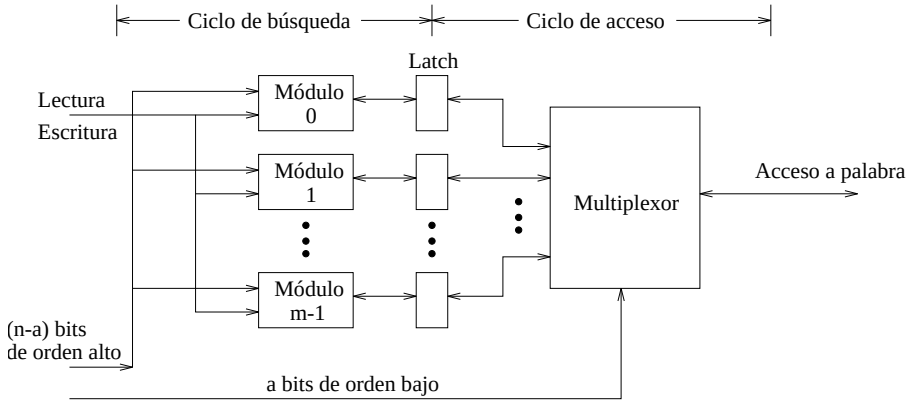


Figura 3.10: Organización de acceso S para una memoria entrelazada de m vías.

Al final de cada ciclo de memoria, $m = 2^a$ palabras consecutivas son capturadas en los buffers de datos de forma simultánea. Los a bits de orden bajo se emplean entonces para multiplexar las m palabras hacia fuera, una por cada ciclo menor. Si se elige el ciclo menor para que valga un $1/m$ de la duración del ciclo mayor (Ec. 3.2), entonces se necesitan dos ciclos de memoria para acceder a m palabras consecutivas.

Sin embargo, si la fase de acceso del último acceso, se superpone con la fase de búsqueda del acceso actual, entonces son m palabras las que pueden ser accedidas en un único ciclo de memoria.

3.2.3 Memoria de acceso C/S

Una organización de memoria que permite los accesos de tipo C y también los de tipo S se denomina *memoria de acceso C/S*. Este esquema de funcionamiento se muestra en la figura 3.11, donde n buses de acceso se utilizan junto a m módulos de memoria entrelazada conectados a cada bus. Los m módulos en cada bus son entrelazados de m vías para permitir accesos C. Los n buses operan en paralelo para permitir los accesos S. En cada ciclo de memoria, al menos $m \cdot n$ palabras son capturadas si se emplean completamente los n buses con accesos a memoria segmentados.

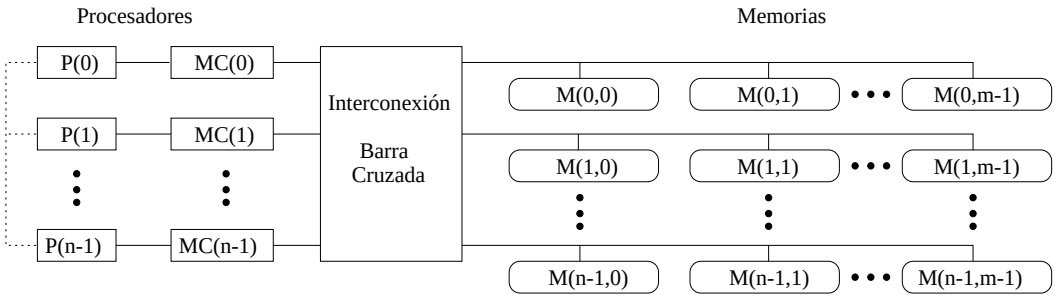


Figura 3.11: Organización de acceso C/S.

La memoria C/S está especialmente indicada para ser usada en configuraciones de multiprocesadores vectoriales, ya que provee acceso segmentado en paralelo de un conjunto vectorial de datos con un alto ancho de banda. Una *cache vectorial* especialmente diseñada es necesaria en el interior de cada módulo de proceso para poder garantizar el movimiento suave entre la memoria y varios procesadores vectoriales.

3.2.4 Rendimiento de la memoria entrelazada y tolerancia a fallos

Con la memoria pasa algo parecido que con los procesadores: no por poner m módulos paralelos en el sistema de memoria se puede acceder m veces más rápido. Existe un modelo más o menos empírico que da el aumento del ancho de banda por el hecho de aumentar el número de bancos de la memoria. Este modelo fue introducido por Hellerman y da el ancho de banda B en función del número de bancos m:

$$B = m^{0.56} \approx \sqrt{m}$$

Esta raíz cuadrada da una estimación pesimista del aumento de las prestaciones de la memoria. Si por ejemplo se ponen 16 módulos en la memoria entrelazada, sólo se obtiene un aumento de 4 veces el ancho de banda. Este resultado lejano a lo esperado viene de que en la memoria principal de los multiprocesadores los accesos entrelazados se mezclan con los accesos simples o con los accesos de bloque de longitudes dispares.

Para los procesadores vectoriales esta estimación no es realista, ya que las transacciones con la memoria suelen ser casi siempre vectoriales y, por tanto, pueden ser fácilmente entrelazadas.

En 1992 Cragon estimó el tiempo de acceso a una memoria entrelazada vectorial de la siguiente manera: Primero se supone que los n elementos de un vector se encuentran consecutivos en una memoria de m módulos. A continuación, y con ayuda de la figura 3.9, no es difícil inferir que el tiempo que tarda en accederse a un vector de n elementos es la suma de lo que tarda el primer elemento (θ), que tendrá que recorrer todo el *cauce*, y lo que tardan los $(n-1)$ elementos restantes ($\theta(n-1)/m$) que estarán completamente encauzados. El tiempo que tarda un elemento (t_1) se obtiene entonces dividiendo lo que tarda el vector completo entre n:

$$t_1 = \frac{\theta + \dfrac{\theta(n-1)}{m}}{n} = \frac{\theta}{n} + \frac{\theta(n-1)}{n\,m} = \frac{\theta}{m}\left(\frac{m}{n} + \frac{n-1}{n}\right) = \frac{\theta}{m}\left(1 + \frac{m-1}{n}\right)$$

Por lo tanto el tiempo medio t_1 requerido para acceder a un elemento en la memoria ha resultado ser:

$$t_1 = \frac{\theta}{m}\left(1 + \frac{m-1}{n}\right)$$

Cuando $n \to \infty$ (vector muy grande), $t_1 \to \theta/m = \tau$ tal y como se derivó en la ecuación (3.2). Además si $m = 1$, no hay memoria entrelazada y $t_1 = \theta$. La ecuación que se acaba de obtener anuncia que la memoria entrelazada se aprovecha del acceso segmentado de los vectores, por lo tanto, cuanto mayor es el vector más rendimiento se obtiene.

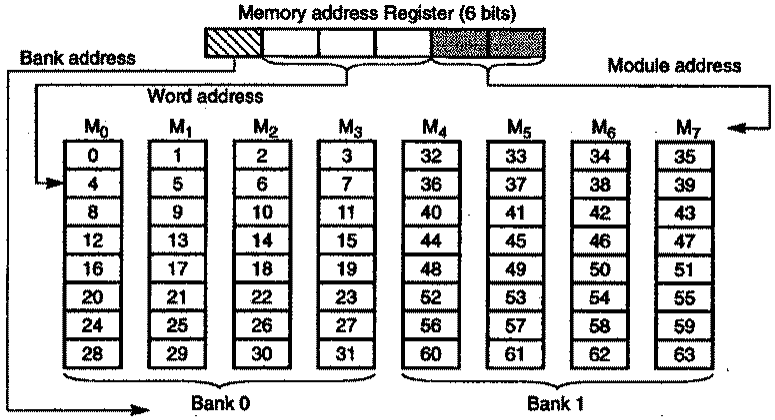
Tolerancia a fallos

La división de la memoria en bancos puede tener dos objetivos: por un lado permite un acceso concurrente lo que disminuye el acceso a la memoria (memoria entrelazada), por otro lado se pueden configurar los módulos de manera que el sistema de memoria pueda seguir funcionando en el caso de que algún módulo deje de funcionar. Si los módulos forman una memoria entrelazada el tiempo de acceso será menor pero el sistema no será tolerante a fallos, ya que al perder un módulo se pierden palabras en posiciones salteadas en toda la memoria, con lo que resulta difícil seguir trabajando. Si por el contrario los bancos se han elegido por bloques de memoria (entrelazado de orden alto) en vez de palabras sueltas, en el caso en que falle un bloque lo programas podrán seguir trabajando con los bancos restantes aislándose ese bloque de memoria erróneo del resto.

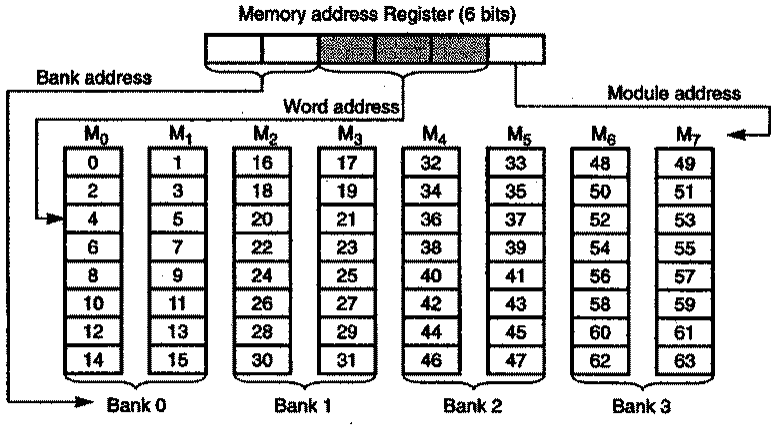
En muchas ocasiones interesa tener ambas características a un tiempo, es decir, por un lado interesa tener memoria entrelazada de orden bajo para acelerar el acceso a la memoria, pero por otro interesa también una memoria entrelazada de orden alto para tener la memoria dividida en bloques y poder seguir trabajando en caso de fallo de un módulo o banco. Para estos casos en que se requiere alto rendimiento y tolerancia a fallos se puede diseñar una memoria mixta que contenga módulos de acceso entrelazado, y bancos para tolerancia a fallos.

La figura 3.12 muestra dos alternativas que combinan el entrelazado de orden alto con el de orden bajo. Ambas alternativas ofrecen una mejora del rendimiento de la memoria junto con la posibilidad de tolerancia a fallos. En el primer ejemplo (figura 3.12a)

se muestra una memoria de cuatro módulos de orden bajo y dos bancos de memoria. En el segundo ejemplo (figura 3.12b) se cuenta con el mismo número de módulos pero dispuestos de manera que hay un entrelazado de dos módulos y cuatro bancos de memoria. El primer ejemplo presenta un mayor entrelazado por lo que tendrá un mayor rendimiento que el segundo, pero también presenta menos bancos por lo que en caso de fallo se pierde una mayor cantidad de memoria, aparte de que el daño que se puede causar al sistema es mayor.



(a) Four-way interleaving within each memory bank



(b) Two-way interleaving within each memory bank

Figura 3.12: Dos organizaciones de memoria entrelazada usando 8 módulos: (a) 2 bancos y 4 módulos entrelazados, (b) 4 bancos y 2 módulos entrelazados.

Si la tolerancia a fallos es fundamental para un sistema, entonces hay que establecer un compromiso entre el grado de entrelazado para aumentar la velocidad y el número de bancos para aumentar la tolerancia a fallos. Cada banco de memoria es independiente de las condiciones de otros bancos y por tanto ofrece un mejor aislamiento en caso de avería.

3.3 Longitud del vector y separación de elementos

Esta sección pretende dar respuesta a dos problemas que surgen en la vida real, uno es qué hacer cuando la longitud del vector es diferente a la longitud de los registros vectoriales (por ejemplo 64 elementos), y la otra es cómo acceder a la memoria si los elementos del vector no están contiguos o se encuentran dispersos.

3.3.1 Control de la longitud del vector

La longitud natural de un vector viene determinada por el número de elementos en los registros vectoriales. Esta longitud, casi siempre 64, no suele coincidir muchas veces con la longitud de los vectores reales del programa. Aun más, en un programa real se desconoce incluso la longitud de cierto vector u operación incluso en tiempo de compilación. De hecho, un mismo trozo de código puede requerir diferentes longitudes en función de parámetros que cambien durante la ejecución de un programa. El siguiente ejemplo en Fortran muestra justo este caso:

```
      do 10 i=1,n
10    Y(i)=a*X(i)+Y(i)
```

La solución de estos problemas es la creación de un *registro de longitud vectorial* VLR (*Vector-Length register*). El VLR controla la longitud de cualquier operación vectorial incluyendo las de carga y almacenamiento. De todas formas, el vector en el VLR no puede ser mayor que la longitud de los registros vectoriales. Por lo tanto, esto resuelve el problema siempre que la longitud real sea menor que la *longitud vectorial máxima* MVL (*Maximum Vector Length*) definida por el procesador.

Para el caso en el que la longitud del vector real sea mayor que el MVL se utiliza una técnica denominada *seccionamiento* (*strip mining*). El seccionamiento consiste en la generación de código de manera que cada operación vectorial se realiza con un tamaño inferior o igual que el del MVL. Esta técnica es similar a la de desenrollamiento de bucles, es decir, se crea un bucle que consiste en varias iteraciones con un tamaño como el del MVL, y luego otra iteración más que será siempre menor que el MVL. La versión seccionada del bucle DAXPY escrita en Fortran se da a continuación:

```
   low=1
   VL=(n mod MVL)             /* Para encontrar el pedazo aparte */
   do 1 j=0,(n/MVL)           /* Bucle externo                   */
       do 10 i=low,low+VL-1   /* Ejecuta VL veces                */
           Y(i)=a*X(i)+Y(i)   /* Operación principal             */
10     continue
       low=low+VL             /* Comienzo del vector siguiente   */
       VL=MVL                 /* Pone la longitud al máximo  */
1  continue
```

En este bucle primero se calcula la parte que sobra del vector (que se calcula con el modulo de n y MVL) y luego ejecuta ya las veces que sea necesario con una longitud de vector máxima. O sea, el primer vector tiene una longitud de `(n mod MVL)` y el resto tiene una longitud de `MVL` tal y como se muestra en la figura 3.13. Normalmente los compiladores hacen estas cosas de forma automática.

Junto con la sobrecarga por el tiempo de arranque, hay que considerar la sobrecarga por la introducción del bucle del seccionamiento. Esta sobrecarga por seccionamiento,

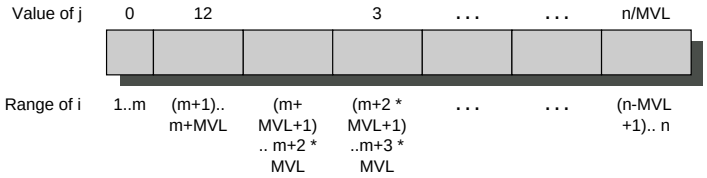


Figura 3.13: Un vector de longitud arbitraria procesado mediante seccionamiento. Todos los bloques menos el primero son de longitud MVL. En esta figura, la variable m se usa en lugar de la expresión ($n \mod MVL$).

que aparece de la necesidad de reiniciar la secuencia vectorial y asignar el VLR, efectivamente se suma al tiempo de arranque del vector, asumiendo que el convoy no se solapa con otras instrucciones. Si la sobrecarga de un convoy es de 10 ciclos, entonces la sobrecarga efectiva por cada 64 elementos se incrementa en 10, o lo que es lo mismo 0.15 ciclos por elemento del vector real.

3.3.2 Cálculo del tiempo de ejecución vectorial

Con todo lo visto hasta ahora se puede dar un modelo sencillo para el cálculo del tiempo de ejecución de las instrucciones en un procesador vectorial. Repasemos estos costes:

1. Por un lado tenemos el número de convoyes en el bucle que nos determina el número de campanadas. Usaremos la notación $T_{campanada}$ para indicar el tiempo en campanadas.
2. La sobrecarga para cada secuencia seccionada de convoyes. Esta sobrecarga consiste en el coste de ejecutar el código escalar para seccionar cada bloque, T_{bucle}, más el coste de arranque para cada convoy, $T_{arranque}$.
3. También podría haber una sobrecarga fija asociada con la preparación de la secuencia vectorial la primera vez, pero en procesadores vectoriales modernos esta sobrecarga se ha convertido en algo muy pequeño por lo que no se considerará en la expresión de carga total. En algunos libros donde todavía aparece este tiempo se le llama T_{base}.

Con todo esto se puede dar una expresión para calcular el tiempo de ejecución para una secuencia vectorial de operaciones de longitud n, que llamaremos T_n:

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{bucle} + T_{arranque}) + n \times T_{campanada} \tag{3.3}$$

Los valores para $T_{arranque}$, T_{bucle} y $T_{campanada}$ dependen del procesador y del compilador que se utilice. Un valor típico para T_{bucle} es 15 (Cray 1). Podría parecer que este tiempo debería ser mayor, pero lo cierto es que muchas de las operaciones de esta sobrecarga se solapan con las instrucciones vectoriales.

Para aclarar todo esto veamos un ejemplo. Se trata de averiguar el tiempo que tarda un procesador vectorial en realizar la operación $A = B \times s$, donde s es un escalar, A y B son vectores con una longitud de 200 elementos. Lo que se hace primero es ver el código en ensamblador que realiza esta operación. Para ello supondremos que las direcciones de A y B son inicialmente `Ra` y `Rb`, y que s se encuentra en `Fs`. Supondremos que `R0` siempre contiene 0 (DLX). Como 200 mod 64=8, la primera iteración del bucle seccionado se realizará sobre un vector de longitud 8, y el resto con una longitud de

64 elementos. La dirección del byte de comienzo del segmento siguiente de cada vector es ocho veces la longitud del vector. Como la longitud del vector es u ocho o 64, se incrementa el registro de dirección por $8 \times 8 = 64$ después del primer segmento, y por $8 \times 64 = 512$ para el resto. El número total de bytes en el vector es $8 \times 200 = 1600$, y se comprueba que ha terminado comparando la dirección del segmento vectorial siguiente con la dirección inicial más 1600. A continuación se da el código:

```
        ADDI      R2,R0,#1600   ; Bytes en el vector
        ADD       R2,R2,Ra      ; Final del vector A
        ADDI      R1,R0,#8      ; Longitud del 1er segmento
        MOVI2S    VLR,R1        ; Carga longitud del vector en VLR
        ADDI      R1,R0,#64     ; Longitud del 1er segmento
        ADDI      R3,R0,#64     ; Longitud del resto de segmentos
LOOP:   LV        V1,Rb         ; Carga B
        MULTVS    V2,V1,Fs      ; Vector * escalar
        SV        Ra,V2         ; Guarda A
        ADD       Ra,Ra,R1      ; Dirección del siguiente segmento de A
        ADD       Rb,Rb,R1      ; Dirección del siguiente segmento de B
        ADDI      R1,R0,#512    ; Byte offset del siguiente segmento
        MOVI2S    VLR,R3        ; Longitud 64 elementos
        SUB       R4,R2,Ra      ; Final de A?
        BNZ       R4,LOOP       ; sino, repite.
```

Las tres instrucciones vectoriales del bucle dependen unas de otras y deben ir en tres convoyes separados, por lo tanto $T_{campanada} = 3$. El tiempo del bucle ya habíamos dicho que ronda los 15 ciclos. El valor del tiempo de arranque será la suma de tres cosas:

- El tiempo de arranque de la instrucción de carga, que supondremos 12 ciclos.
- El tiempo de arranque de la multiplicación, 7 ciclos.
- El tiempo de arranque del almacenamiento, otros 12 ciclos.

Por lo tanto obtenemos un valor $T_{arranque} = 12 + 7 + 12 = 31$ ciclos de reloj. Con todo esto, y aplicando la ecuación (3.3), se obtiene un tiempo total de proceso de $T_{200} = 784$ ciclos de reloj. Si dividimos por el número de elementos obtendremos el tiempo de ejecución por elemento, es decir, $784/200 = 3.9$ ciclos de reloj por elemento del vector. Comparado con $T_{campanada}$, que es el tiempo sin considerar las sobrecargas, vemos que efectivamente la sobrecarga puede llegar a tener un valor significativamente alto.

Resumiendo las operaciones realizadas se tiene el siguiente proceso hasta llegar al resultado final:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{arranque}) + n \times T_{campanada}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

$$T_{start=12+7+12=31}$$

$$T_{200} = 660 + 4 \times 31 = 784$$

La figura 3.14 muestra la sobrecarga y el tiempo total de ejecución por elemento del ejemplo que estamos considerando. El modelo que sólo considera las campanadas tendría un coste de 3 ciclos, mientras que el modelo más preciso que incorpora la sobrecarga añade 0.9 a este valor.

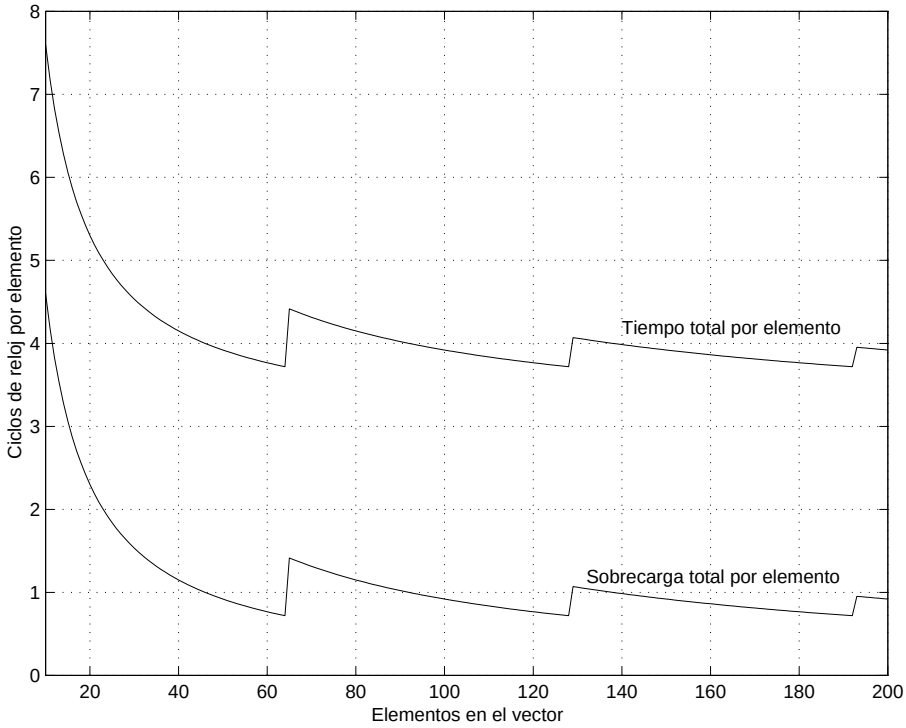


Figura 3.14: Tiempo de ejecución por elemento en función de la longitud del vector.

3.3.3 Separación de elementos en el vector

El otro problema que se presenta en la programación real es que la posición en memoria de elementos adyacentes no siempre son contiguas. Por ejemplo, consideremos el código típico para multiplicación de matrices:

```
  do 10 i=1,100
      do 10 j=1,100
          A(i,j)=0.0
          do 10 k=1,100
10            A(i,j)=A(i,j)+B(i,k)*C(k,j)
```

En la sentencia con etiqueta 10, se puede vectorizar la multiplicación de cada fila de B con cada columna de C, y seccionar el bucle interior usando k como variable índice. Cuando una matriz se ubica en memoria, se lineariza organizándola en filas o en columnas. El almacenamiento por filas, utilizado por muchos lenguajes menos el Fortran, consiste en asignar posiciones consecutivas a elementos consecutivos en la fila, haciendo adyacentes los elementos $B(i,j)$ y $B(i,j+1)$. El almacenamiento por columnas, utilizado en Fortran, hace adyacentes los elementos $B(i,j)$ y $B(i+1,j)$.

Suponiendo que utilizamos el almacenamiento por columnas de Fortran nos encontramos con que los accesos a la matriz B no son adyacentes en memoria sino que se encuentran separados por una fila completa de elementos. En este caso, los elementos de B que son accedidos en el lazo interior, están separados por el tamaño de fila multiplicado por 8 (número de bytes por elemento) lo que hace un total de 800 bytes.

A esta distancia en memoria entre elementos consecutivos se le llama *separación* (*stride*). Con el ejemplo que estamos viendo podemos decir que los elementos de C

tienen una separación de 1, (1 palabra doble, 8 bytes), mientras que la matriz B tiene una separación de 100 (100 palabras dobles, 800 bytes).

Una vez cargados estos elementos adyacentes en el registro vectorial, los elementos son lógicamente contiguos. Por todo esto, y para aumentar el rendimiento de la carga y almacenamiento de vectores con elementos separados, resulta interesante disponer de instrucciones que tengan en cuenta la separación entre elementos contiguos de un vector. La forma de introducir esto en el lenguaje ensamblador es mediante la incorporación de dos instrucciones nuevas, una de carga y otra de almacenamiento, que tengan en cuenta no sólo la dirección de comienzo del vector, como hasta ahora, sino también el paso o la separación entre elementos. En DLXV, por ejemplo, existen las instrucciones `LVWS` para carga con separación, y `SVWS` para almacenamiento con separación. Así, la instrucción `LVWS V1,(R1,R2)` carga en V1 lo que hay a partir de R1 con un paso o separación de elementos de R2, y `SVWS (R1,R2),V1` guarda los elementos del vector V1 en la posición apuntada por R1 con paso R2.

Naturalmente, el que los elementos no estén separados de forma unitaria crea complicaciones en la unidad de memoria. Se había comprobado que una operación memoria-registro vectorial podía proceder a velocidad completa si el número de bancos en memoria era al menos tan grande el tiempo de acceso a memoria en ciclos de reloj. Sin embargo, para determinadas separaciones entre elementos, puede ocurrir que accesos consecutivos se realicen al mismo banco, llegando incluso a darse el caso de que todos los elementos del vector se encuentren en el mismo banco. A esta situación se le llama *conflicto del banco de memoria* y hace que cada carga necesite un mayor tiempo de acceso a memoria. El conflicto del banco de memoria se presenta cuando se le pide al mismo banco que realice un acceso cuando el anterior aún no se había completado. Por consiguiente, la condición para que se produzca un conflicto del banco de memoria será:

$$\frac{\text{Mín. común mult.(separación, núm. módulos)}}{\text{Separación}} < \text{Latencia acceso a memoria}$$

Los conflictos en los módulos no se presentan si la separación entre los elementos y el número de bancos son relativamente primos entre sí, y además hay suficientes bancos para evitar conflictos en el caso de separación unitaria. El aumento de número de bancos de memoria a un número mayor del mínimo, para prevenir detenciones con una separación 1, disminuirá la frecuencia de detenciones para las demás separaciones. Por ejemplo, con 64 bancos, una separación de 32 parará cada dos accesos en lugar de cada acceso. Si originalmente tuviésemos una separación de 8 con 16 bancos, pararía cada dos accesos; mientras que con 64 bancos, una separación de 8 parará cada 8 accesos. Si tenemos acceso a varios vectores simultáneamente, también se necesitarán más bancos para prevenir conflictos. La mayoría de supercomputadores vectoriales actuales tienen como mínimo 64 bancos, y algunos llegan a 512.

Veamos un ejemplo. Supongamos que tenemos 16 bancos de memoria con un tiempo de acceso de 12 ciclos de reloj. Calcular el tiempo que se tarda en leer un vector de 64 elementos separados unitariamente. Repetir el cálculo suponiendo que la separación es de 32. Como el número de bancos es mayor que la latencia, la velocidad de acceso será de elemento por ciclo, por tanto 64 ciclos, pero a esto hay que añadirle el tiempo de arranque que supondremos 12, por tanto la lectura se realizará en $12+64 = 76$ ciclos de reloj. La peor separación es aquella en la que la separación sea un múltiplo del número de bancos, como en este caso que tenemos una separación de 32 y 16 bancos. En este caso siempre se accede al mismo banco con lo que cada acceso colisiona con el anterior,

esto nos lleva a un tiempo de acceso de 12 ciclos por elemento y un tiempo total de $12 \times 64 = 768$ ciclos de reloj.

3.4 Mejora del rendimiento de los procesadores vectoriales

3.4.1 Encadenamiento de operaciones vectoriales

Hasta ahora, se habían considerado separadas, y por tanto en convoyes diferentes, instrucciones sobre vectores que utilizaran el mismo o los mismos registros vectoriales. Este es el caso, por ejemplo de dos instrucciones como

```
MULTV  V1,V2,V3
ADDV   V4,V1,V5
```

Si se trata en este caso al vector `V1` no como una entidad, sino como una serie de elementos, resulta sencillo entender que la operación de suma pueda iniciarse unos ciclos después de la de multiplicación, y no después de que termine, ya que los elementos que la suma puede ir necesitando ya los ha generado la multiplicación. A esta idea, que permite solapar dos instrucciones, se le llama *encadenamiento*. El encadenamiento permite que una operación vectorial comience tan pronto como los elementos individuales de su operando vectorial fuente estén disponibles, es decir, los resultados de la primera unidad funcional de la cadena se adelantan a la segunda unidad funcional. Naturalmente deben ser unidades funcionales diferentes, de lo contrario surge un conflicto temporal.

Si las unidades están completamente segmentadas, basta retrasar el comienzo de la siguiente instrucción durante el tiempo de arranque de la primera unidad. El tiempo total de ejecución para la secuencia anterior sería:

Longitud del vector + Tiempo de arranque suma + Tiempo de arranque multiplicación

La figura 3.15 muestra los tiempos de una versión de ejecución no encadenada y de otra encadenada del par de instrucciones anterior suponiendo una longitud de 64 elementos. El tiempo total de la ejecución encadenada es de 77 ciclos de reloj que es sensiblemente inferior a los 145 ciclos de la ejecución sin encadenar. Con 128 operaciones en punto flotante realizadas en ese tiempo, se obtiene 1.7 FLOP por ciclo de reloj, mientras que con la versión no encadenada la tasa sería de 0.9 FLOP por ciclo de reloj.

3.4.2 Sentencias condicionales

Se puede comprobar mediante programas de test, que los niveles de vectorización en muchas aplicaciones no son muy altos [HP96]. Debido a la ley de Amdahl el aumento de velocidad en estos programas está muy limitado. Dos razones por las que no se obtiene un alto grado de vectorización son la presencia de condicionales (sentencias *if*) dentro de los bucles y el uso de matrices dispersas. Los programas que contienen sentencias *if* en los bucles no se pueden ejecutar en modo vectorial utilizando las técnicas expuestas en este capítulo porque las sentencias condicionales introducen control de flujo en el bucle. De igual forma, las matrices dispersas no se pueden tratar eficientemente

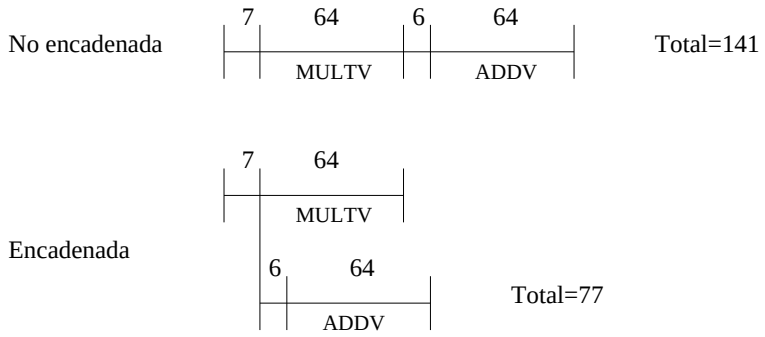


Figura 3.15: Temporización para la ejecución no encadenada y encadenada.

utilizando algunas de las capacidades que se han mostrado; esto es por ejemplo un factor importante en la falta de vectorización de Spice. Se explican a continuación algunas técnicas para poder ejecutar de forma vectorial algunas de estas estructuras.

Dado el siguiente bucle:

```
    do 100 i=1,64
        if (A(i) .ne. 0) then
            A(i)=A(i)-B(i)
        endif
100 continue
```

Este bucle no puede vectorizarse a causa de la ejecución condicional del cuerpo. Sin embargo, si el bucle anterior se pudiera ejecutar en las iteraciones para las cuales $A(i) \neq 0$ entonces se podría vectorizar la resta. Para solucionarlo se emplea una máscara sobre el vector.

El *control de máscara vectorial* es un vector booleano de longitud MVL. Cuando se carga el *registro de máscara vectorial* con el resultado de un test del vector, cualquier instrucción vectorial que se vaya a ejecutar solamente opera sobre los elementos del vector cuyas entradas correspondientes en el registro de máscara vectorial sean 1. Las entradas del registro vectorial destino que corresponden a un 0 en el registro de máscara no se modifican por la operación del vector. Para que no actúe, el registro de máscara vectorial se inicializa todo a 1, haciendo que las instrucciones posteriores al vector operen con todos los elementos del vector. Con esto podemos reescribir el código anterior para que sea vectorizable:

```
    LV      V1,Ra      ; Carga vector A en V1
    LV      V2,Rb      ; Carga vector B en V2
    LD      F0,#0      ; Carga F0 con 0 en punto flotante
    SNESV   F0,V1      ; Inicializa VM a 1 si V1(i)!=0
    SUBV    V1,V1,V2   ; Resta bajo el control de la máscara
    CVM                ; Pone la máscara todo a unos
    SV      Ra,V1      ; guarda el resultado en A.
```

El uso del vector de máscara tiene alguna desventaja. Primero, la operación se realiza para todos los elementos del vector, por lo que da lo mismo que la condición se cumpla o no, siempre consume tiempo de ejecución. De todas formas, incluso con una máscara repleta de ceros, el tiempo de ejecución del código en forma vectorial suele ser menor que la versión escalar. Segundo, en algunos procesadores lo que hace la máscara es deshabilitar el almacenamiento en el registro del resultado de la operación, pero la operación se hace en cualquier caso. Esto tiene el problema de que si por ejemplo

estamos dividiendo, y no queremos dividir por cero (para evitar la excepción) lo normal es comprobar los elementos que sean cero y no dividir, pero en un procesador cuya máscara sólo deshabilite el almacenamiento y no la operación, realizará la división por cero generando la excepción que se pretendía evitar.

3.4.3 Matrices dispersas

Las matrices dispersas son matrices que tienen una gran cantidad de elementos, siendo la mayoría de ellos cero. Este tipo de matrices, que habitualmente ocuparían mucha memoria de forma innecesaria, se encuentran almacenadas de forma compacta y son accedidas indirectamente. Para una representación típica de una matriz dispersa nos podemos encontrar con código como el siguiente:

```
        do 100 i=1,n
100         A(K(i))=A(K(i))+C(M(i))
```

Este código realiza la suma de los vectores dispersos `A` y `C`, usando como índices los vectores `K` y `M` que designan los elementos de `A` y `B` que no son cero (ambas matrices deben tener el mismo número de elementos no nulos). Otra forma común de representar las matrices dispersas utiliza un vector de bits como máscara para indicar qué elementos existen y otro vector para almacenar sus valores. A menudo ambas representaciones coexisten en el mismo programa. Este tipo de matrices se encuentran en muchos códigos, y hay muchas formas de tratar con ellas dependiendo de la estructura de datos utilizada en el programa.

Un primer mecanismo consiste en las operaciones de *dispersión* y *agrupamiento* utilizando vectores índices. El objetivo es moverse de una representación densa a la dispersa normal y viceversa. La operación de agrupamiento coge el vector índice y busca en memoria el vector cuyos elementos se encuentran en las direcciones dadas por la suma de una dirección base y los desplazamientos dados por el vector índice. El resultado es un vector no disperso (denso) en un registro vectorial. Una vez se han realizado las operaciones sobre este vector denso, se pueden almacenar de nuevo en memoria de forma expandida mediante la operación de dispersión que utilizará el mismo vector de índices. El soporte hardware para estas operaciones se denomina *dispersar-agrupar* (*scatter-gather*). En el ensamblador vienen dos instrucciones para realizar este tipo de tareas. En el caso del DLXV se tiene `LVI` (cargar vector indexado), `SVI` (almacenar vector indexado), y CVI (crear vector índice, por ejemplo `CVI V1,R1` introduce en V1 los valores `0,R1,2*R1,3*R1,...,63*R1`). Por ejemplo, suponer que `Ra`, `Rc`, `Rk` y `Rm` contienen las direcciones de comienzo de los vectores de la secuencia anterior, entonces el bucle interno de la secuencia se podría codificar como:

```
LV    Vk,Rk         ; Carga K
LVI   Va,(Ra+Vk)    ; Carga A(K(i))
LV    Vm,Rm         ; Carga M
LVI   Vc,(Rc+Vm)    ; Carga C(M(i))
ADDV  Va,Va,Vc      ; Los suma
SVI   (Ra+Vk),Va    ; Almacena A(K(i))
```

De esta manera queda vectorizada la parte de cálculo con matrices dispersas. El código en Fortran dado con anterioridad nunca se vectorizaría de forma automática puesto que el compilador no sabría si existe dependencia de datos, ya que no sabe a priori lo que contiene el vector `K`.

Algo parecido se puede realizar mediante el uso de la máscara que se vio en las sentencias condicionales. El registro de máscara se usa en este caso para indicar los elementos no nulos y así poder formar el vector denso a partir de un vector disperso.

La capacidad de dispersar/agrupar (*scatter-gather*) está incluida en muchos de los supercomputadores recientes. Estas operaciones rara vez alcanzan la velocidad de un elemento por ciclo, pero son mucho más rápidas que la alternativa de utilizar un bucle escalar. Si la propiedad de dispersión de un matriz cambia, es necesario calcular un nuevo vector índice. Muchos procesadores proporcionan soporte para un cálculo rápido de dicho vector. La instrucción `CVI` (*Create Vector Index*) del DLX crea un vector índice dado un valor de salto (m), cuyos valores son $0, m, 2 \times m, ..., 63 \times m$. Algunos procesadores proporcionan una instrucción para crear un vector índice comprimido cuyas entradas se corresponden con las posiciones a 1 en el registro máscara. En DLX, definimos la instrucción *CVI* para que cree un vector índice usando el vector máscara. Cuando el vector máscara tiene todas sus entradas a uno, se crea un vector índice estándar.

Las cargas y almacenamientos indexados y la instrucción `CVI` proporcionan un método alternativo para soportar la ejecución condicional. A continuación se muestra la secuencia de instrucciones que implementa el bucle que vimos al estudiar este problema y que corresponde con el bucle mostrado en la página 66:

```
LV      V1,Ra        ; Carga vector A en V1
LD      F0,#0        ; Carga F0 con cero en punto flotante
SNESV   F0,V1        ; Pone VM(i) a 1 si V1(i)<>F0
CVI     V2,#8        ; Genera índices en V2
POP     R1,VM        ; Calcula el número de unos en VM
MOVI2S  VLR,R1       ; Carga registro de longitud vectorial
CVM                  ; Pone a 1 los elementos de la máscara
LVI     V3,(Ra+V2)   ; Carga los elementos de A distintos de cero
LVI     V4,(Rb+V2)   ; Carga los elementos correspondientes de B
SUBV    V3,V3,V4     ; Hace la resta
SVI     (Ra+V2),V3   ; Almacena A
```

El que la implementación utilizando dispersar/agrupar (*scatter-gather*) sea mejor que la versión utilizando la ejecución condicional, depende de la frecuencia con la que se cumpla la condición y el coste de las operaciones. Ignorando el encadenamiento, el tiempo de ejecución para la primera versión es $5n + c_1$. El tiempo de ejecución de la segunda versión, utilizando cargas y almacenamiento indexados con un tiempo de ejecución de un elemento por ciclo, es $4n + 4 \times f \times n + c_2$, donde f es la fracción de elementos para la cual la condición es cierta (es decir, $A \neq 0$. Si suponemos que los valores c_1 y c_2 son comparables, y que son mucho más pequeños que n, entonces para que la segunda técnica sea mejor que la primera se tendrá que cumplir

$$5n \geq 4n + 4 \times f \times n$$

lo que ocurre cuando $\frac{1}{4} \geq f$.

Es decir, el segundo método es más rápido que el primero si menos de la cuarta parte de los elementos son no nulos. En muchos casos la frecuencia de ejecución es mucho menor. Si el mismo vector de índices puede ser usado varias veces, o si crece el número de sentencias vectoriales con la sentencia *if*, la ventaja de la aproximación de dispersar/agrupar aumentará claramente.

3.5 El rendimiento de los procesadores vectoriales

3.5.1 Rendimiento relativo entre vectorial y escalar

A partir de la ley de Amdahl es relativamente sencillo calcular el rendimiento relativo entre la ejecución vectorial y la escalar, es decir, lo que se gana al ejecutar un programa de forma vectorial frente a la escalar tradicional. Supongamos que r es la relación de velocidad entre escalar y vectorial, y que f es la relación de vectorización. Con esto, se puede definir el siguiente *rendimiento relativo*:

$$P = \frac{1}{(1-f) + f/r} = \frac{r}{(1-f)r + f} \tag{3.4}$$

Este rendimiento relativo mide el aumento de la velocidad de ejecución del procesador vectorial sobre el escalar. La relación hardware de velocidad r es decisión del diseñador. El factor de vectorización f refleja el porcentaje de código en un programa de usuario que se vectoriza. El rendimiento relativo es bastante sensible al valor de f. Este valor se puede incrementar utilizando un buen compilador vectorial o a través de transformaciones del programa.

Cuanto más grande es r tanto mayor es este rendimiento relativo, pero si f es pequeño, no importa lo grande que sea r, ya que el rendimiento relativo estará cercano a la unidad. Fabricantes como IBM tienen una r que ronda entre 3 y 5, ya que su política es la de tener cierto balance entre las aplicaciones científicas y las de negocios. Sin embargo, empresas como Cray y algunas japonesas eligen valores mucho más altos para r, ya que la principal utilización de estas máquinas es el cálculo científico. En estos casos la r ronda entre los 10 y 25. La figura 3.16 muestra el rendimiento relativo para una máquina vectorial en función de r y para varios valores de f.

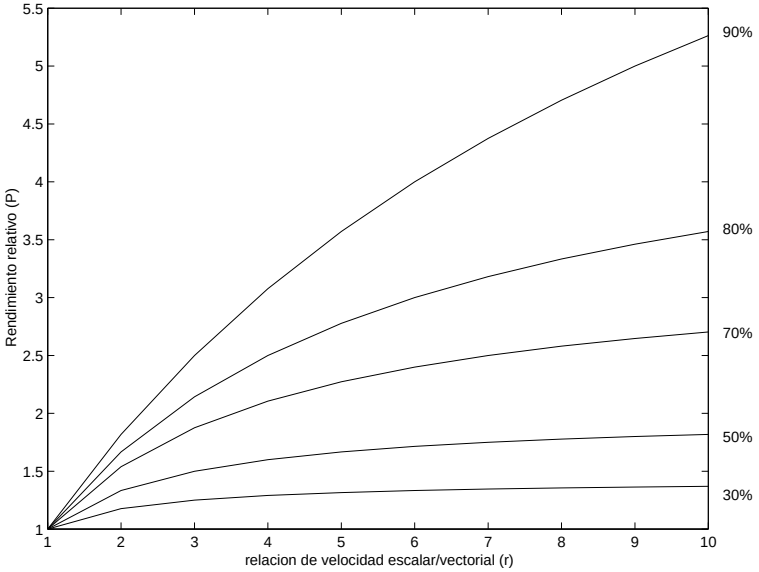


Figura 3.16: Rendimiento relativo escalar/vectorial.

3.5.2 Medidas del rendimiento vectorial

Dado que la longitud del vector es tan importante en el establecimiento del rendimiento de un procesador, veremos las medidas relacionadas con la longitud además del tiempo de ejecución y los MFLOPS obtenidos. Estas medidas relacionadas con la longitud tienden a variar de forma muy importante dependiendo del procesador y que son importantes de comparar. (Recordar, sin embargo, que el *tiempo* es siempre la medida de interés cuando se compara la velocidad relativa de dos procesadores.) Las tres medidas más importantes relacionadas con la longitud son

- R_n. Es la velocidad de ejecución, dada en MFLOPS, para un vector de longitud n.
- R_∞. Es la velocidad de ejecución, dada en MFLOPS, para un vector de longitud infinita. Aunque esta medida puede ser de utilidad para medir el rendimiento máximo, los problemas reales no manejan vectores ilimitados, y la sobrecarga existente en los problemas reales puede ser mayor.
- $N_{1/2}$. La longitud de vector necesaria para alcanzar la mitad de R_∞. Esta es una buena medida del impacto de la sobrecarga.
- N_v. La longitud de vector a partir de la cual el modo vectorial es más rápido que el modo escalar. Esta medida tiene en cuenta la sobrecarga y la velocidad relativa del modo escalar respecto al vectorial.

Veamos como se pueden determinar estas medidas en el problema DAXPY ejecutado en el DLXV. Cuando existe el encadenamiento de instrucciones, el bucle interior del código DAXPY en convoys es el que se muestra en la figura 3.17 (suponiendo que Rx y Ry contienen la dirección de inicio).

```
LV V1,Rx    MULTSV V2,F0,V1   Convoy 1: chained load and multiply
LV V3,Ry    ADDV V4,V2,V3     Convoy 2: second load and ADD, chained
SV Ry,V4                      Convoy 3: store the result
```

Figura 3.17: Formación de convoys en el bucle interior del código DAXPY.

El tiempo de ejecución de un bucle vectorial con n elementos, T_n, es:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{bucle} + T_{arranque}) + n \times T_{campanada}$$

El encadenamiento permite que el bucle se ejecute en tres campanadas y no menos, dado que existe un cauce de memoria; así $T_{campanada} = 3$. Si $T_{campanada}$ fuera una indicación completa del rendimiento, el bucle podría ejecutarse a una tasa de $2/3 \times tasa\,del\,reloj$ MFLOPS (ya que hay 2 FLOPs por iteración). Así, utilizando únicamente $T_{campanada}$, un DLXV a 200 MHz ejecutaría este bucle a 133 MFLOPS suponiendo la no existencia de seccionamiento (*strip-mining*) y el coste de inicio. Existen varias maneras de aumentar el rendimiento: añadir unidades de carga-almacenamiento adicionales, permitir el solapamiento de convoys para reducir el impacto de los costes de inicio, y decrementar el número de cargas necesarias mediante la utilización de registros vectoriales.

Rendimiento máximo del DLXV en el DAXPY

En primer lugar debemos determinar el significado real del rendimiento máximo, R_∞. Por ahora, continuaremos suponiendo que un convoy no puede comenzar hasta que todas las instrucciones del convoy anterior hayan finalizado; posteriormente eliminaremos esta restricción. Teniendo en cuenta esta restricción, la sobrecarga de inicio para la secuencia vectorial es simplemente la suma de los tiempos de inicio de las instrucciones:

$$T_{arranque} = 12 + 7 + 12 + 6 + 12 = 49$$

Usando $MVL = 64$, $T_{loop} = 15$, $T_{start} = 49$, y $T_{chime} = 3$ en la ecuación del rendimiento, y suponiendo que n no es un múltiplo exacto de 64, el tiempo para una operación de n elementos es

$$T_n = \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n = (n + 64) + 3n = 4n + 64$$

La velocidad sostenida está por encima de 4 ciclos de reloj por iteración, más que la velocidad teórica de 3 campanadas, que ignora los costes adicionales. La mayor parte de esta diferencia es el coste de inicio para cada bloque de 64 elementos (49 ciclos frente a 15 de la sobrecarga del bucle).

Podemos calcular R_∞ para una frecuencia de reloj de 200 MHz como

$$R_\infty = lim_{n\to\infty} \left(\frac{Operaciones\ por\ iteración \times frecuencia\ de\ reloj}{Ciclos\ de\ reloj\ por\ iteración} \right)$$

El numerador es independiente de n, por lo que

$$R_\infty = \frac{Operaciones\ por\ iteración \times frecuencia\ de\ reloj}{lim_{n\to\infty}\ (Ciclos\ de\ reloj\ por\ iteración)}$$

$$lim_{n\to\infty}\ (Ciclos\ de\ reloj\ por\ iteración) = lim_{n\to\infty} \left(\frac{T_n}{n} \right) = lim_{n\to\infty} \left(\frac{4n + 64}{n} \right) = 4$$

$$R_\infty = \frac{2 \times 200\ MHz}{4} = 100\ MFLOPS$$

El rendimiento sin el coste de inicio, que es el rendimiento máximo dada la estructura de la unidad funcional vectorial, es 1.33 veces superior. En realidad, la distancia entre el rendimiento de pico y el sostenido puede ser incluso mayor.

Rendimiento sostenido del DLXV en el Benchmark Linpack

El benchmark Linpack es una eliminación de Gauss sobre una matriz de 100×100. Así, la longitud de los elementos van desde 99 hasta 1. Un vector de longitud k se usa k veces. Así, la longitud media del vector viene dada por

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66.3$$

Ahora podemos determinar de forma más precisa el rendimiento del DAXPY usando una longitud de vector de 66.

$$T_n = 2 \times (15 + 49) + 3 \times 66 = 128 + 198 = 326$$

$$R_{66} = \frac{2 \times 66 \times 200\,\text{MHz}}{326} = 81\,\text{MFLOPS}$$

El rendimiento máximo, ignorando los costes de inicio, es 1.64 veces superior que el rendimiento sostenido que hemos calculado. En realidad, el benchmark Linpack contiene una fracción no trivial de código que no puede vectorizarse. Aunque este código supone menos del 20% del tiempo antes de la vectorización, se ejecuta a menos de una décima parte del rendimiento cuando se mide en FLOPs. Así, la ley de Amdahl nos dice que el rendimiento total será significativamente menor que el rendimiento estimado al analizar el bucle interno.

Dado que la longitud del vector tiene un impacto significativo en el rendimiento, las medidas $N_{1/2}$ y N_v se usan a menudo para comparar máquinas vectoriales.

Ejemplo Calcular $N_{1/2}$ para el bucle interno de DAXPY para el DLXV con un reloj de 200 MHz.

Respuesta Usando R_∞ como velocidad máxima, queremos saber para qué longitud del vector obtendremos 50 MFLOPS. Empezaremos con la fórmula para MFLOPS suponiendo que las medidas se realizan para $N_{1/2}$ elementos:

$$MFLOPS = \frac{FLOPs\ ejecutados\ en\ N_{1/2}\ iteraciones}{Ciclos\ de\ reloj\ para\ N_{1/2}\ iteraciones} \times \frac{Ciclos\ de\ reloj}{Segundos} \times 10^{-6}$$

$$50 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 200$$

Simplificando esta expresión y suponiendo que $N_{1/2} \leq 64$, tenemos que $T_{n \leq 64} = 1 \times 64 + 3 \times n$, lo que da lugar a

$$T_{N_{1/2}} = 8 \times N_{1/2}$$

$$1 \times 64 + 3 \times N_{1/2} = 8 \times N_{1/2}$$

$$5 \times N_{1/2} = 64$$

$$N_{1/2} = 12.8$$

Por lo tanto, $N_{1/2} = 13$; es decir, un vector de longitud 13 proporciona aproximadamente la mitad del rendimiento máximo del DLXV en el bucle DAXPY.

Ejemplo ¿Cuál es la longitud del vector, N_v, para que la operación vectorial se ejecute más rápidamente que la escalar?

Respuesta De nuevo, sabemos que $R_v < 64$. El tiempo de una iteración en modo escalar se puede estimar como $10 + 12 + 12 + 7 + 6 + 12 = 59$ ciclos de reloj, donde 10 es el tiempo estimado de la sobrecarga del bucle. En el ejemplo anterior se vio que $T_{n \leq 64} = 64 + 3 \times n$ ciclos de reloj. Por lo tanto,

$$64 + 3 \times N_v = 59\, N_v$$

$$N_v = \left\lceil \frac{64}{56} \right\rceil$$

$$N_v = 2$$

Rendimiento del DAXPY en un DLXV mejorado

El rendimiento del DAXPY, como en muchos problemas vectoriales, viene limitado por la memoria. Consecuentemente, éste se puede mejorar añadiendo más unidades de acceso a memoria. Esta es la principal diferencia arquitectónica entre el CRAY X-MP (y los procesadores posteriores) y el CRAY-1. El CRAY X-MP tiene tres cauces de acceso a memoria, en comparación con el único cauce a memoria del CRAY-1, permitiendo además un encadenamiento más flexible. ¿Cómo afectan estos factores al rendimiento?

Ejemplo ¿Cuál sería el valor de T_{66} para el bucle DAXPY en el DLXV si añadimos dos cauces más de acceso a memoria?.

Respuesta Con tres canales de acceso a memoria, todas las operaciones caben en un único convoy. Los tiempos de inicio son los mismos, por lo que

$$T_{66} = \left\lceil \frac{66}{64} \right\rceil \times (T_{loop} + T_{arranque}) + 66 \times T_{campanada}$$

$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

Con tres cauces de acceso a memoria, hemos reducido el tiempo para el rendimiento sostenido de 326 a 194, un factor de 1.7. Observación del efecto de la ley de Amdahl: Hemos mejorado la velocidad máxima teórica, medida en el número de *campanadas*, en un factor de 3, pero la mejora total es de 1.7 en el rendimiento sostenido.

Otra mejora se puede conseguir del solapamiento de diferentes convoys y del coste del bucle escalar con las instrucciones vectoriales. Esta mejora requiere que una operación vectorial pueda usar una unidad funcional antes de que otra operación haya finalizado, complicando la lógica de emisión de instrucciones.

Para conseguir una máxima ocultación de la sobrecarga del seccionamiento (*strip-mining*), es necesario poder solapar diferentes instancias del bucle, permitiendo la ejecución simultánea de dos instancias de un convoy y del código escalar. Esta técnica, denominada *tailgating*, se usó en el Cray-2. Alternativamente, podemos desenrollar el bucle exterior para crear varias instancias de la secuencia vectorial utilizando diferentes conjuntos de registros (suponiendo la existencia de suficientes registros). Permitiendo el máximo solapamiento entre los convoys y la sobrecarga del bucle escalar, el tiempo de inicio y de la ejecución del bucle sólo sería *observable* una única vez en cada convoy. De esta manera, un procesador con registros vectoriales puede conseguir unos costes de arranque bajos para vectores cortos y un alto rendimiento máximo para vectores muy grandes.

Ejemplo ¿Cuales serían los valores de R_∞ y T_{66} para el bucle DAXPY en el DLXV si añadimos dos cauces más de acceso a memoria y permitimos que los costes del seccionamiento (*strip-mining*) y de arranque se solapen totalmente?.

Respuesta

$$R_\infty = lim_{n\to\infty} \left(\frac{Operaciones\ por\ iteración \times frecuencia\ de\ reloj}{Ciclos\ de\ reloj\ por\ iteración} \right)$$

$$lim_{n\to\infty} (Ciclos\ de\ reloj\ por\ iteración) = lim_{n\to\infty} \left(\frac{T_n}{n} \right)$$

Dado que la sobrecarga sólo se observa una vez, $T_n = n + 49 + 15 = n + 64$. Así,

$$lim_{n\to\infty}\left(\frac{T_n}{n}\right) = lim_{n\to\infty}\left(\frac{n+64}{n}\right) = 1$$

$$R_\infty = \frac{2 \times 200\,\text{MHz}}{1} = 400\,\text{MFLOPS}$$

Añadir unidades adicionales de acceso a memoria y una lógica de emisión más flexible da lugar a una mejora en el rendimiento máximo de un factor de 4. Sin embargo, $T_{66} = 130$, por lo que para vectores cortos, la mejora en el rendimiento sostenido es de $\frac{326}{100} = 2.5$ veces.

3.6 Historia y evolución de los procesadores vectoriales

Para finalizar, la figura 3.18 muestra una comparación de la diferencia de rendimiento entre los procesadores vectoriales y los procesadores superescalares de última generación. En esta figura podemos comprobar cómo en los últimos años se ha ido reduciendo la diferencia en rendimiento de ambas arquitecturas.

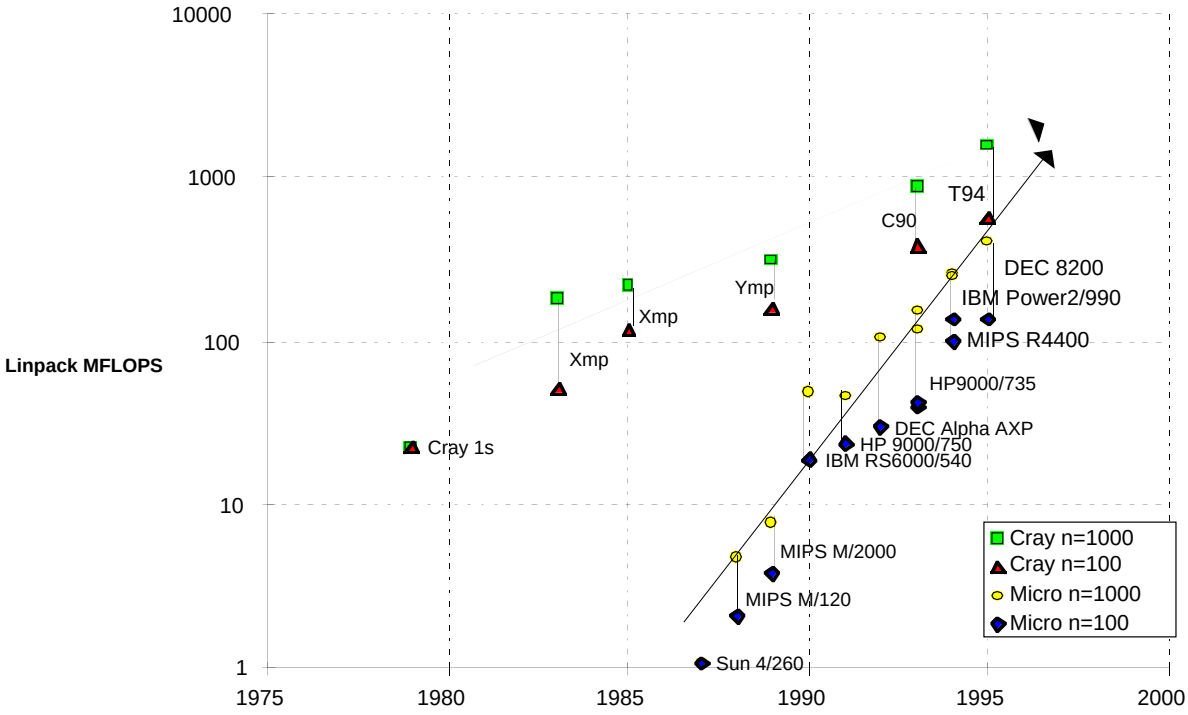


Figura 3.18: Comparación del rendimiento de los procesadores vectoriales y los microprocesadores escalares para la resolución de un sistema de ecuaciones lineales denso (tamaño de la matriz=$n \times n$).

Capítulo 4

Procesadores matriciales

Según la clasificación de Flynn uno de los cuatro tipos de sistemas es el SIMD (*Single Instruction stream Multiple Data stream*) Este tipo de sistemas explotan el paralelismo inherente en los datos más que en las instrucciones. El computador de tipo SIMD clásico es el computador matricial.

La bibliografía para este tema se encuentra en [HB87] y [Hwa93].

4.1 Organización básica

La configuración básica de un procesador matricial se muestra en la figura 4.1. Como vemos se trata de N elementos de proceso (EP) sincronizados y bajo el control de una única unidad de control (UC). Cada elemento de proceso está formado básicamente por una unidad aritmético lógica, asociada a unos registros de trabajo, y una memoria local para el almacenamiento de datos distribuidos. La unidad de control, que muchas veces es un procesador escalar, tiene su propia memoria para almacenar el programa y datos. Las instrucciones escalares y de control como saltos, etc. se ejecutan directamente en la unidad de control. Las instrucciones vectoriales son transmitidas a los EPs para su ejecución. De esta manera se alcanza un alto grado de paralelismo gracias a la multiplicidad de los elementos procesadores.

Este esquema que se acaba de comentar y que corresponde al de la figura 4.1, es el modelo de computador matricial con *memoria distribuida*. Otra posibilidad consiste en tener la *memoria compartida* intercalando la red de interconexión entre los elementos de proceso y las memorias. Las diferencias con el modelo anterior son que las memorias ligadas a los EPs son sustituidas por módulos en paralelo que son compartidos por todos los EPs mediante la red de interconexión. La otra diferencia es que la red de interconexión del modelo de la figura se intercambia por la red de interconexión o *alineamiento* entre los elementos de proceso y las memorias.

Un modelo operacional para los computadores matriciales viene especificado por la siguiente quíntupla:

$$M = \langle N, C, I, M, R \rangle \tag{4.1}$$

donde:

1. N es el número de elementos de proceso en la máquina. Por ejemplo, la Illiac IV tiene 64 EPs, mientras que la Connection Machine CM-2 tiene 65.536 EPs.

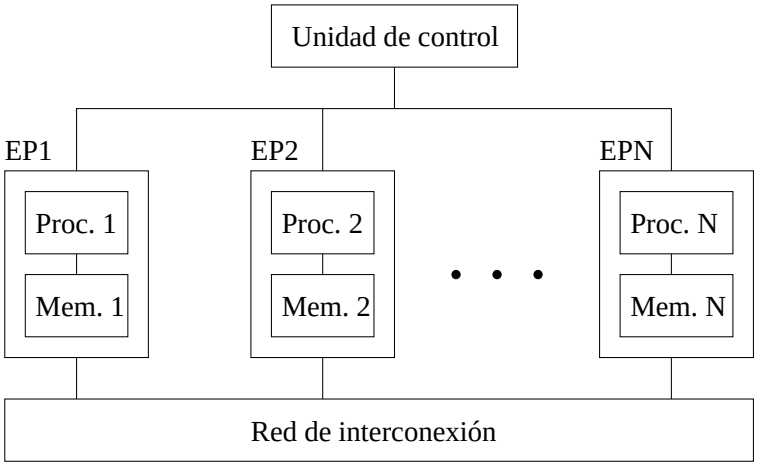


Figura 4.1: Computador matricial básico.

2. C es el conjunto de instrucciones ejecutadas directamente por la unidad de control incluyendo las instrucciones escalares y las de control del flujo de ejecución.
3. I es el conjunto de instrucciones que la unidad de control envía a todos los EPs para su ejecución en paralelo. Estas instrucciones son aritméticas, lógicas, rutado de datos, enmascaramiento, y otras operaciones locales que son ejecutadas por los EPs sobre la memoria local.
4. M es el conjunto de posibilidades de enmascaramiento donde cada máscara se encarga de dividir el conjunto de EPs en subconjuntos de EPs habilitados o deshabilitados.
5. R es el conjunto de funciones de rutado que especifican varios patrones para ser establecidos en la red de interconexión para intercomunicación entre EPs.

4.2 Estructura interna de un elemento de proceso

Aunque las características de un EP en un procesador matricial pueden variar de unas máquinas a otras, se van a dar aquí unas nociones generales de los elementos básicos que forman los EPs en un computador matricial. La figura 4.2 muestra un procesador ejemplo para ilustrar las explicaciones.

Vamos a suponer que cada elemento de proceso EP_i tiene su memoria local MEP_i. Internamente al EP habrá un conjunto de registros e indicadores formado por A_i, B_i,C_i y S_i, una unidad aritmético-lógica, un registro índice local I_i, un registro de direcciones D_i, y un registro de encaminamiento de datos R_i. El R_i de cada EP_i está conectado al R_j de otros EPs vecinos mediante la red de interconexión. Cuando se producen transferencias de datos entre los los EPs, son los contenidos de R_i los que se transfieren. Representamos los N EPs como EP_i para $i = 0, 1, \ldots, N-1$, donde el índice i es la dirección del EP_i. Definimos una constante m que será el número de bits necesarios para codificar el número N. El registro D_i se utiliza entonces para contener los m bits de la dirección del EP_i. Esta estructura que se cuenta aquí está basada en el diseño del Illiac-IV.

Algunos procesadores matriciales pueden usar dos registros de encaminamiento, uno para la entrada y otro para la salida. Se considerará en nuestro caso un único registro

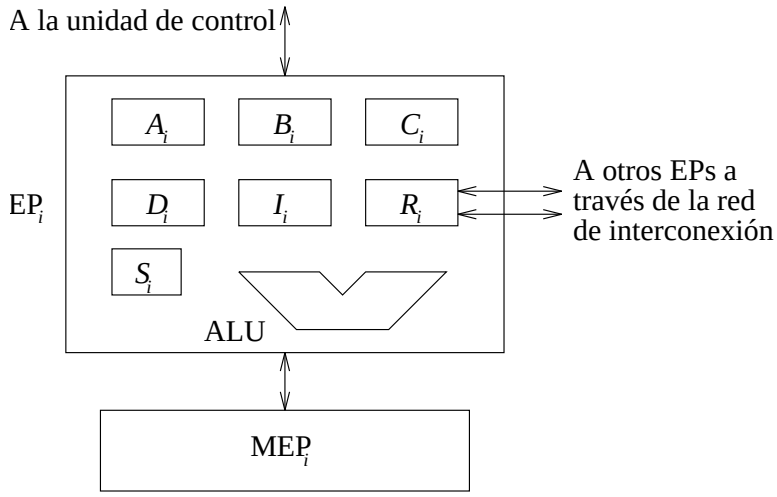


Figura 4.2: Componentes de un elemento de proceso (EP) en un computador matricial.

(R_i) en el cual las entradas y salidas están totalmente aisladas gracias al uso de biestables maestro-esclavo. Cada EP$_i$ está o bien en modo activo o bien en modo inactivo durante cada ciclo de instrucción. Si un EP$_i$ está activo, ejecuta la instrucción que le ha enviado la unidad de control (UC). Si está inactivo no ejecuta la instrucción que reciba. Los esquemas de enmascaramiento se utilizan para modificar los indicadores de estado S_i, supondremos que si $S_i = 1$ entonces el EP$_i$ está activo, y sino, está inactivo.

El conjunto de indicadores S_i para $i = 0, 1, \ldots, N-1$, forma un registro de estado S para todos los EPs. En la unidad de control hay un registro que se llama M que sirve de máscara para establecer el estado de los EPs, por lo tanto M tiene N bits. Obsérvese que las configuraciones de bits de los registros M y S son intercambiables bajo el control de la UC cuando se va a establecer un enmascaramiento.

Desde el punto de vista hardware, la longitud física de un vector está determinada por el número de EPs. La UC realiza la partición de un vector largo en bucles vectoriales, el establecimiento de una dirección base global y el incremento de desplazamiento respecto de esa dirección base. La distribución de los elementos vectoriales sobre las diferentes MEP es crucial para la utilización eficaz de una colección de EPs. Idealmente se deberían poder obtener N elementos simultáneamente procedentes de las diferentes MEP. En el peor caso, todos los elementos del vector estarían alojados en una sola MEP. En tal situación, tendrían que ser accedidos de forma secuencial, uno tras otro.

Un vector lineal unidimensional de n elementos puede ser almacenado en todas las MEP si $n \leq N$. Los vectores largos ($n > N$) pueden ser almacenados distribuyendo los n elementos cíclicamente entre los N EPs. El cálculo de matrices bidimensionales puede causar problemas, ya que pueden ser necesarios cálculos intermedios entre filas y columnas. La matriz debería almacenarse de modo que fuera posible el acceso en paralelo a una fila, una columna, o una diagonal en un ciclo de memoria.

En un procesador matricial, los operandos vectoriales pueden ser especificados por los registros a utilizar o por las direcciones de memoria a referenciar. Para instrucciones de referencia a memoria, cada EP$_i$ accede a la MEP$_i$ local, con el desplazamiento indicado por su propio registro índice I_i. El registro I_i modifica la dirección de memoria global difundida desde la UC. Así, diferentes posiciones pueden ser accedidas en diferentes MEP$_i$ simultáneamente con la misma dirección global especificada por la UC.

La utilización del registro índice local (I_i) es evidente cuando se quiere acceder a los elementos de la diagonal de una matriz. Si la matriz se encuentra colocada de forma consecutiva a partir de la dirección 100, habrá que cargar cada registro índice con el desplazamiento correspondiente a cada elemento de la diagonal. Una vez inicializados los índices, se pasará como operando la dirección 100 que es la del comienzo de la matriz, y cada procesador leerá un elemento distinto indicado por el índice, en este caso la diagonal de la matriz.

Aparte del propio paralelismo obtenido por el cálculo en paralelo sobre los elementos de un vector, el propio encaminamiento de los datos reduce el tiempo de ejecución de determinadas tareas. Por ejemplo, en el tratamiento de imágenes, donde muchas operaciones son cálculos entre píxels vecinos, una arquitectura matricial paraleliza con facilidad los cálculos. También en operaciones donde unos datos en un vector dependen de resultados anteriores sobre ese vector puede beneficiarse de la flexibilidad de interconexión entre los EPs.

Los procesadores matriciales son computadores de propósito específico destinados a limitadas aplicaciones científicas donde pueden alcanzar un rendimiento elevado. Sin embargo, los procesadores matriciales tienen algunos problemas de vectorización y programación difíciles de resolver. Lo cierto es que los computadores matriciales no son populares entre los fabricantes de supercomputadores comerciales.

4.3 Instrucciones matriciales

4.4 Programación

4.4.1 Multiplicación SIMD de matrices

4.5 Procesadores asociativos

4.5.1 Memorias asociativas

4.5.2 Ejemplos de procesadores asociativos

Capítulo 5

Generalidades sobre las redes de interconexión

La red de interconexión es uno de los elementos más importantes de cualquier arquitectura paralela puesto que va a modificar el rendimiento global del sistema y la topología de la arquitectura. La red de interconexión va a ser el vehículo a través del cual se van a comunicar los diferentes elementos del sistema, memoria con procesador, elementos periféricos, unos procesadores con otros en un computador matricial, etc. Las redes de interconexión se utilizan en computadores matriciales así como en multiprocesadores y multicomputadores, dependiendo del tipo de sistema los elementos que se interconectan pueden variar, pero la topología de la red, así como los protocolos, suelen ser comunes e independientes del tipo de sistema. En este capítulo se estudiarán precisamente las diferentes topologías de redes de interconexión sean cuales fueren los elementos a interconectar.

En cuanto a la bibliografía de este tema, conviene consultar [DYN97] que es uno de los libros de redes para multicomputadores más completos y modernos. También en [CSG99] se encuentra una buena sección dedicada a las redes. Por último, algunas de las definiciones y parámetros se han tomado de [Wil96].

5.1 Definiciones básicas y parámetros que caracterizan las redes de interconexión

En esta parte dedicada a las redes se verán diferentes topologías para la construcción de redes de interconexión. La eficiencia en la comunicación en la capa de interconexión es crítica en el rendimiento de un computador paralelo. Lo que se espera es conseguir una red con latencia baja y una alta tasa de transferencia de datos y por tanto una ancho de banda amplio. Las propiedades de red que veremos a continuación van a ayudar a la hora de elegir el tipo de diseño para una arquitectura. Veamos por tanto las definiciones que nos van a caracterizar una red:

Tamaño de la red

El *tamaño de la red* es el número de nodos que contiene, es decir, el número de elementos interconectados entre si. Estos nodos pueden ser procesadores, memorias, computadores, etc.

Grado del nodo

El número de canales que entran y salen de un nodo es el *grado del nodo* y lo representaremos por *d degree*. En el caso de canales unidireccionales, el número de canales que entran en el nodo es el *grado de entrada* y los que salen el *grado de salida*. El grado del nodo representa el número de puertos de E/S y por lo tanto el coste del nodo. Esto quiere decir que el grado de los nodos debería ser lo más reducido posible para reducir costes. Si además el grado del nodo es constante se consigue que el sistema pueda ser modular y fácilmente escalable con la adición de nuevos módulos.

Diámetro de la red

El *diámetro D* de una red es el máximo de los caminos más cortos entre dos nodos cualquiera de una red.

$$D = max_{i,j \in N}(min_{p \in P_{ij}} length(p))$$

donde P_{ij} es el conjunto de caminos de i a j. La longitud del camino se mide por el número de enlaces por los que pasa el camino. El diámetro de la red nos da el número máximo de saltos entre dos nodos, de manera que de esta forma tenemos una medida de lo buena que es la comunicación en la red. Por todo esto, el diámetro de la red debería ser lo más pequeño posible desde el punto de vista de la comunicación. El diámetro se utilizó hasta finales de los 80 como la principal figura de mérito de una red, de ahí la popularidad que tuvieron en esa época redes de bajo diámetro como los hipercubos.

Anchura de la bisección

El ancho de la bisección, B, es el mínimo número de canales que, al cortar, separa la red en dos partes iguales. La bisección del cableado, B_W, es el número de cables que cruzan esta división de la red. $B_W = BW$, donde W es el ancho de un canal en bits. Este parámetro nos da una cota inferior de la densidad del cableado. Desde el punto de vista del diseñador, B_W es un factor fijo, y el ancho de la bisección restringe la anchura de los canales a $W = \frac{B_W}{B}$. La figura 5.1 muestra un ejemplo del cálculo del ancho de la bisección.

Cuando una red dada se divide en dos mitades iguales, al número mínimo de canales que atraviesa el corte se le llama *anchura de canal biseccional*. En el caso de una red de comunicaciones, caso común, cada canal estará compuesto por un número w de bits, hilos o cables. De esta manera podemos definir la *anchura de cable biseccional* como $B = bw$. Este parámetro B refleja la densidad de cables en una red. Esta anchura de la bisección nos da una buena indicación del ancho de banda máximo en una red a través de un corte transversal realizado en la red. El resto de cortes estarían acotados por esta anchura de la bisección.

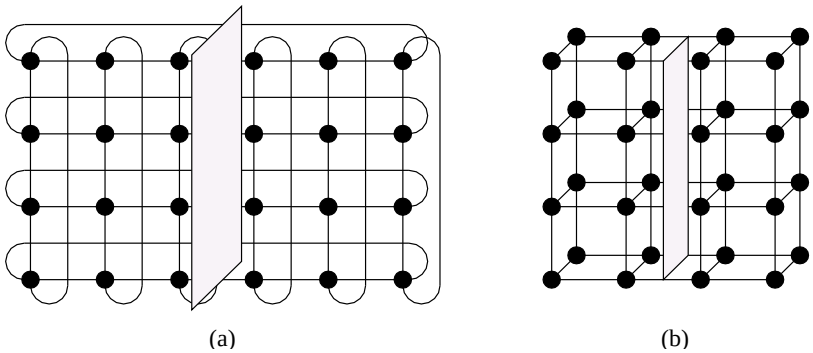


Figura 5.1: Ejemplos del cálculo del ancho de la bisección: toro 2-D. (b) toro 3-D (no se muestran los enlaces de cierre).

Longitud del cable

La *longitud del cable* entre nodos es también importante puesto que tiene efectos directos sobre la latencia de la señal, desfase del reloj, o incluso los requerimientos de potencia.

Redes simétricas

Una red es *simétrica* si es isomorfa a ella misma independientemente del nodo que consideremos como origen, es decir, una red es *simétrica* si la topología de esta se ve igual desde cualquier nodo. Este tipo de redes tiene la ventaja de que simplifica mucho de los problemas de manejo de recursos. Por ejemplo, dado un patrón de tráfico uniforme, en las redes simétricas se produce una carga uniforme de los canales de la red, cosa que no ocurre en las redes asimétricas.

Otras propiedades como que si los nodos son homogéneos, o si los canales tienen memoria o son sólo conmutadores, son otras propiedades útiles para caracterizar la red.

Rendimiento de la red

Para resumir los puntos anteriormente expuestos, veamos cuales son los factores que intervienen en el rendimiento global de una red de interconexión:

Funcionalidad Esto indica cómo la red soporta el encaminamiento de datos, tratamiento de las interrupciones, sincronización, combinación petición/mensaje, y la coherencia.

Latencia de la red Indica el retraso de un mensaje, en el peor caso, a través de la red.

Ancho de banda Indica la velocidad máxima de transmisión de datos, en Mbytes/s, transmitidos a través de la red.

Complejidad hardware Indica el coste de implementación como el coste de los cables, conmutadores, conectores, arbitraje, y lógica de interfaz.

Escalabilidad Indica la capacidad de una red para expandirse de forma modular con nuevos recursos en la máquina y sin mucho detrimento en el rendimiento global.

Capacidad de transmisión de la red

La *capacidad de transmisión de una red* se define como el número total de datos que pueden ser transmitidos a través de la red por unidad de tiempo. Una forma de estimar la capacidad de transmisión es calcular la capacidad de la red, es decir, el número total de mensajes que pueden haber en la red a la vez. Normalmente, la máxima capacidad de transmisión es una fracción de su capacidad.

Un *punto caliente* suele estar formado por un par de nodos que recogen una porción demasiado grande el tráfico total de la red. El tráfico en estos puntos calientes puede degradar el rendimiento de toda la red a causa de la congestión que producen. La *capacidad de transmisión en puntos calientes* se define como la máxima velocidad a la que se pueden enviar mensajes de un nodo específico P_i a otro nodo específico P_j.

Las redes de dimensión pequeña (2D,3D) operan mejor bajo cargas no uniformes ya que se comparten más recursos. En una red de dimensiones más elevadas, los cables se asignan a una dimensión particular y no pueden ser compartidas entre dimensiones. Por ejemplo, en un n-cubo binario es posible que una línea se sature mientras otra línea físicamente adyacente, pero asignada a una dimensión diferente, permanece inactiva. En un toro, todas las líneas físicamente adyacentes se combinan en un único canal que es compartido por todos los mensajes.

La latencia mínima de la red se alcanza cuando el parámetro k y la dimensión n se eligen de manera que las componentes de la latencia debida a la distancia D (enlaces entre nodos) y a la relación L/W (longitud L del mensaje normalizada por la anchura W) quedan aproximadamente iguales. La menor latencia se obtiene con dimensiones muy bajas, 2 para hasta 10245 nodos.

Las redes de dimensión baja reducen la contención porque con pocos canales de ancho de banda amplio se consigue que se compartan mejor los recursos y, por tanto, un mejor rendimiento de las colas de espera al contrario que con muchos canales de banda estrecha. Mientras que la capacidad de la red y la latencia peor de bloque son independientes de la dimensión, las redes de dimensión pequeña tienen una mayor capacidad de transmisión máxima y una latencia de bloque media menor que las redes de dimensiones altas.

5.1.1 Topología, control de flujo y encaminamiento

El diseño de una red se realiza sobre tres capas independientes: *topología*, *encaminamiento* y *control de flujo*.

- La **topología** hace referencia al grafo de interconexión de la red $I = G(N, C)$ donde N son los nodos del grafo y C es el conjunto de enlaces unidireccionales o bidireccionales que los conectan. Si pensamos en un multiprocesador como en un problema de asignación de recursos, la topología es la primera forma de asignación de los mismos.
- El **control de flujo** hace referencia al método utilizado para regular el tráfico en la red. Es el encargado de evitar que los mensajes se entremezclen y controlar su avance para asegurar una progresión ordenada de los mismos a través de la red. Si dos mensajes quieren usar el mismo canal al mismo tiempo, el control de flujo determina (1) qué mensaje obtiene el canal y (2) qué pasa con el otro mensaje.
- El **encaminamiento** hace referencia al método que se usa para determinar el cami-

no que sigue un mensaje desde el nodo origen al nodo destino. El encaminamiento se puede ver como una relación, $C \times N \times C$, que asigna al canal ocupado por la cabecera del mensaje y en función del nodo destino un conjunto de canales que pueden utilizarse a continuación para que el mensaje llegue a su destino. El encaminamiento es una forma dinámica de asignación de recursos. Dada una topología, un nodo actual, y un destino, la relación de encaminamiento determina como llevar un mensaje desde el nodo actual al nodo destino.

Topología

Una topología se evalúa en términos de los siguientes cinco parámetros: Ancho de la bisección, grado del nodo, diámetro de la red, longitud de la red y su simetría. Estos parámetros ya fueron expuestos al principio de este capítulo.

La topología de la red también guarda una gran relación con la construcción física de la red, especialmente con el empaquetamiento, que consiste en poner juntos todos los nodos procesadores de la red y sus interconexiones. En la figura 5.2 se muestra un ejemplo de empaquetamiento.

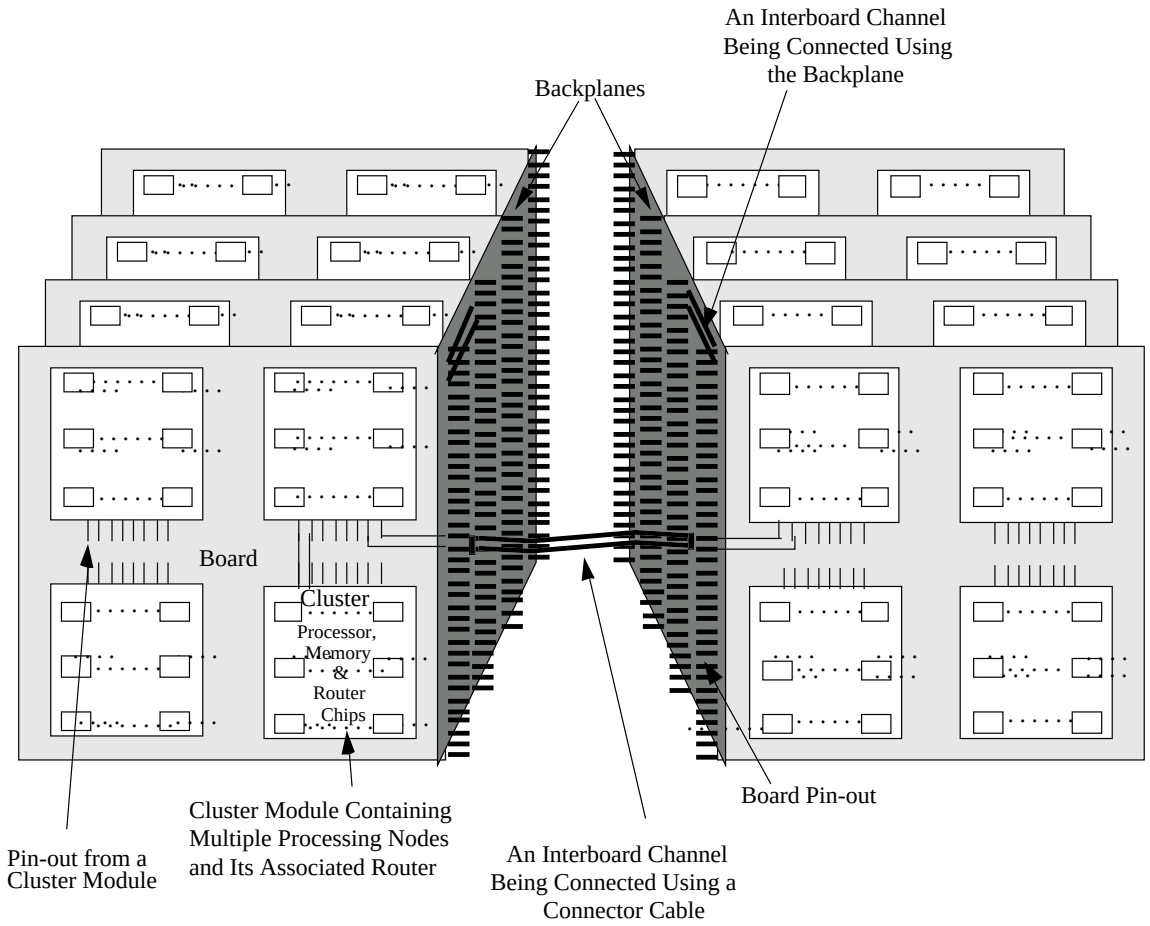


Figura 5.2: Ejemplo de empaquetamiento de un multicomputador.

Control de flujo

El control de flujo se refiere al método usado para regular el tráfico en una red. Determina cuando un mensaje o parte de un mensaje puede avanzar. También se hace cargo de la política de asignación de los recursos: buffers y canales; a las unidades de información: mensajes, paquetes y flits.

- **Mensaje**. Es la unidad lógica de comunicación. Dos objetos se comunican mediante el envío de un mensaje. Es la única unidad vista por los clientes de un servicio de red.
- **Paquete**. Un mensaje se divide en uno o más paquetes. Un paquete es la menor unidad que contiene información de encaminamiento. Si existe más de un paquete en un mensaje, cada paquete contiene además un número de secuencia que permite su reensamblaje.
- **Flit**. Un paquete se puede dividir a su vez en dígitos de control de flujo o *flits*, la menor unidad sobre la cual se puede realizar el control de flujo. Es decir, la comunicación de recursos, cables y buffers, se realizan en función de flits. En general, un flit no contiene información de encaminamiento. Únicamente el primer flit de un paquete sabe a donde se dirige. El resto de flits debe seguir al flit cabecera para determinar su encaminamiento.

Una simple analogía ilustra la diferencia entre los paquetes y los flits. Los paquetes son como los automóviles. Dado que conocen a donde van, pueden entrelazarse libremente. Los flits, por otro lado, son como los vagones de un tren. deben seguir al flit cabecera para encontrar su destino. No se pueden mezclar con los flits de otros paquetes, o perderían su único contacto con su destino.

Otro aspecto importante que tiene que solucionar el control de flujo es el de bloqueo de un paquete. En este caso será necesario disponer de algún espacio buffer para su almacenamiento temporal. Cuando ya no existe más espacio, el mecanismo de control de flujo detiene la transmisión de información. Cuando el paquete avanza y aparece más espacio buffer disponible, la transmisión comienza de nuevo. Existe también la alternativa de eliminar el paquete cuando deja de existir espacio disponible o desviarse a través de otro canal.

Encaminamiento

El encaminamiento es el método usado por un mensaje para elegir un camino entre los canales de la red. El encaminamiento puede ser visto como un par (R, ρ), donde

$$R \subset C \times N \times C,$$

$$\rho : P(C) \times \alpha \longmapsto C.$$

La relación de encaminamiento R identifica los caminos permitidos que pueden ser usados por un mensaje para alcanzar su destino. Dada la posición actual del mensaje, C, y su nodo destino, N, R identifica un conjunto de canales permitidos, C, que pueden ser usados como siguiente paso en el camino.

La función ρ selecciona uno de los caminos de entre los permitidos. En cada paso del encaminamiento, ρ toma el conjunto de posibles canales siguientes, $P(C)$, alguna

información adicional acerca del estado de la red, α, y elige en canal en concreto, C. La información adicional, α, puede ser constante, aleatoria, o basada en información sobre el tráfico de la red.

Los métodos de encaminamiento pueden clasificarse en *deterministas*, *inconscientes* (*oblivious*), o adaptativos. Con el encaminamiento determinista, el camino que sigue un mensaje depende únicamente de los nodos origen y destino. En este caso R es una función y α es constante (no se proporciona información adicional).

En un encaminamiento inconsciente se puede elegir varios caminos a través de la red, pero no se utiliza información acerca del estado de la red para elegir un camino. El mensaje no es consciente del resto de tráfico en la red. Con un encaminamiento de este tipo, R es una relación (pueden existir varios caminos permisibles). Para que el encaminamiento sea inconsciente, α no puede contener información acerca del estado de la red. Puede ser aleatoria, una función de tiempo, o una función del contenido del mensaje.

El caso más general es el encaminamiento adaptativo, donde el router puede usar información acerca del estado de la red. Es este caso, α puede ser cualquier función.

5.2 Clasificación de las redes de interconexión según su topología

Entre otros criterios, las redes de interconexión de han clasificado en función de modo de funcionamiento (síncrono o asíncrono), y control de la red (centralizado, descentralizado, o distribuido). Hoy en día, los multicomputadores, multiprocesadores, y NOWs dominan el mercado de los computadores paralelos. Todas estas arquitecturas utilizan redes asíncronas con control distribuido. Por lo tanto, nos centraremos en otros criterios que son más significativos en la actualidad.

La figura 5.3 muestra una clasificación de las redes de interconexión conocidas en cuatro grupos en función principalmente de la topología de la red: redes de medio común, redes directas, redes indirectas, y redes híbridas. Para cada grupo, la figura muestra una jerarquía de clases, indicando alguna implementación real que utiliza dicha topología.

En las *redes de medio compartido*, el medio de transmisión está compartido por todos los dispositivos que tienen posibilidad de comunicación. Un enfoque alternativo consiste en tener enlaces punto a punto que conecten de forma directa cada elemento de comunicación con un subconjunto (normalmente reducido) de otros los dispositivos existentes en la red. En este caso, la comunicación entre elementos de comunicación no vecinos requiere la transmisión de la información a través de varios dispositivos intermedios. A estas redes se les denominan *redes directas*. En vez de conectar de forma directa los elementos de comunicación, las *redes indirectas* los conectan mediante uno o más conmutadores. Si existen varios conmutadores, estos suelen estar conectados entre ellos mediante enlaces punto a punto. En este caso, cualquier comunicación entre distintos dispositivos requiere transmitir la información a través de uno o más conmutadores. Finalmente, es posible una aproximación *híbrida*.

- Redes de medio compartido
  - Redes de área local
    - Bus de contención (Ethernet)
    - Bus de tokens (Arenet)
    - Anillo de tokens (FDDI Ring, IBM Token Ring)
  - Bus de sistema (Sun Gigaplane, DEC AlphaServer8X00, SGI PoweerPath-2)
- Redes directas (Redes estáticas basadas en encaminador)
  - Topologías estrictamente ortogonales
    - Malla
      - Malla 2-D (Intel Paragon)
      - Malla 3-D (MIT J-Machine)
    - Toros (n-cubo k-arios)
      - Toro 1-D unidireccional o anillo (KSR forst-level ring)
      - Toro 2-D bidireccional (Intel/CMU iWarp)
      - Toro 2-D bidireccional (Cray T3D, Cray T3E)
    - Hipercubo (Intel iPSC, nCUBE)
  - Otras topologías directas: Árboles, Ciclos cubo-conectados, Red de Bruijn, Grafos en Estrella, etc.
- Redes Indirectas (Redes dinámicas basadas en conmutadores)
  - Topologías Regulares
    - Barra cruzada (Cray X/Y-MP, DEC GIGAswitch, Myrinet)
    - Redes de Interconexión Multietapa (MIN)
      - Redes con bloqueos
        MIN Unidireccionales (NEC Cenju-3, IBM RP3)
        MIN Bidireccional (IBM SP, TMC CM-5, Meiko CS-2)
      - Redes sin bloqueos: Red de Clos
  - Topologías Irregulares (DEC Autonet, Myrinet, ServerNet)
- Redes Híbridas
  - Buses de sistema múltiples (Sun XDBus)
  - Redes jerárquicas (Bridged LANs, KSR)
    - Redes basadas en Agrupaciones (Stanford DASH, HP/Convex Exemplar)
  - Otras Topologías Hipergrafo: Hiperbuses, Hipermallas, etc.

Figura 5.3: Clasificación de las redes de interconexión. (1-D = unidimensional; 2-D = bidimensional; 3-D = tridimensional; CMU = Carnegie Mellon University; DASH = Directory Architecture for Shared-Memory; DEC = Digital Equipment Corp.; FDDI = Fiber Distributed Data Interface; HP = Hewlett-Packard; KSR = Kendall Square Research; MIN = Multistage Interconnection Network; MIT = Massachusetts Institute of Technology; SGI = Silicon Graphics Inc.; TMC = Thinking Machines Corp.)

5.2.1 Redes de medio compartido

La estructura de interconexión menos compleja es aquella en la que el medio de transmisión está compartido por todos los elementos de comunicación. En estas *redes de medio compartido*, sólo un dispositivo puede utilizar la red en un momento dado. Cada elemento conectado a la red tiene circuitos para manejar el paso de direcciones y datos. La red en sí misma actúa, normalmente, como un elemento pasivo ya que no genera mensajes.

Un concepto importante es el de *estrategia de arbitraje* que determina cómo se resuelven los conflictos de acceso al medio. Una característica de un medio compartido es la posibilidad de soportar un *broadcast* atómico en donde todos los dispositivos conectados al medio pueden monitorizar actividades y recibir la información que se está transmitiendo en el medio compartido. Esta propiedad es importante para un soporte eficiente de muchas aplicaciones que necesitan comunicaciones del tipo "uno a todos" o "uno a muchos", como las barreras de sincronización y protocolos de coherencia de caché basados en *snoopy*. Debido al limitado ancho de banda, un medio compartido únicamente puede soportar un número limitado de dispositivos antes de convertirse en un cuello de botella.

Las redes de medio compartido pueden dividirse en dos grandes grupos: las redes de área local, usadas principalmente en la construcción de redes de ordenadores cuya distancia máxima no supera unos pocos kilómetros, y los buses usados en la comunicación interna de los uniprocesadores y multiprocesadores.

Bus del sistema (*Backplane bus*)

Un *bus del sistema* es la estructura de interconexión más simple para los multiprocesadores basados en bus. Se usa normalmente para interconectar procesadores y módulos de memoria para proporcionar una arquitectura UMA. La figura 5.4 muestra una red con un bus único. Un bus de este tipo consta usualmente de 50 a 300 hilos físicamente realizados mediante una placa base.

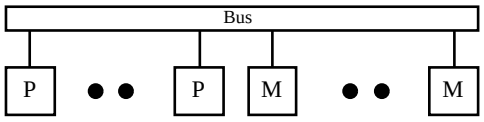


Figura 5.4: Una red con bus único. (M = memoria; P = procesador.)

Existen tres tipos de información en un bus de este tipo: datos, direcciones y señales de control. Las señales de control incluyen la señal de petición del bus y la señal de permiso de acceso al bus, entre muchas otras. Además del ancho de las líneas de datos, el máximo ancho de banda depende de la tecnología. El número de procesadores que pueden conectarse al bus depende de varios factores, como la velocidad del procesador, el ancho de banda del bus, la arquitectura caché y el comportamiento del programa.

LANs de medio compartido

Es posible utilizar una LAN de alta velocidad como una columna vertebral que permita interconectar ordenadores para proporcionar un entorno de computación distribuido.

Físicamente, una LAN usa hilos de cobre o fibra óptica como medio de transmisión. La topología utilizada puede ser un bus o un anillo. Debido a razones de implementación y rendimiento, no es práctico tener un control centralizado o algún mecanismo fijo de asignación de acceso que determine quién puede acceder al bus. Las tres alternativas principales de LANs basadas en un control distribuido se describen a continuación.

Bus de contención: El mecanismo de arbitraje de bus más popular es que todos los dispositivos compitan para tener el acceso exclusivo al bus. Debido a la compartición del medio todos los dispositivos pueden monitorizar el estado del bus y detectar colisiones. Aquí, el término "colisión" significa que dos o más dispositivos están usando el bus al mismo tiempo y sus datos colisionan. Cuando se detecta una colisión, los dispositivos causantes de la misma abortan la transmisión para intentarlo posteriormente. Entre las LANs que utilizan este mecanismo está la Ethernet que adopta el protocolo CSMA/CD (*Carrier-Sense Multiple Access with Collision Detection*). El ancho de banda de la Ethernet es 10 Mbps y la distancia máxima es de 250 metros (cable coaxial). Para romper la barrera de 10 Mbps ha aparecido Fast Ethernet que puede proporcionar un ancho de banda de 100 Mbps.

Token Bus: Una desventaja del bus de contención es su naturaleza no determinista, ya que no puede garantizar cuánto se debe esperar hasta ganar el acceso al bus. Por lo tanto, el bus de contención no es el idóneo para soportar aplicaciones de tiempo real. Para eliminar el comportamiento no determinista, aparece un enfoque alternativo que implica pasar un testigo entre los dispositivos conectados a la red. El dispositivo que tiene el testigo tiene el acceso al bus. Cuando termina de transmitir sus datos, el testigo se pasa al siguiente dispositivo según algún esquema prefijado. Restringiendo el tiempo máximo que se puede estar en posesión del testigo se puede garantizar una cota superior al tiempo que un dispositivo debe esperar. Arcnet soporta token bus con un ancho de banda de 2.4 Mbps.

Token Ring: Una extensión natural al token bus es la de utilizar una estructura de anillo. El token ring de IBM proporciona anchos de banda de 4 y 16 Mbps sobre cable coaxial. El protocolo FDDI (*Fiber Distributed Data Interface*) es capaz de proporcionar un ancho de banda de 100 Mbps usando fibra óptica.

5.2.2 Redes Directas

La escalabilidad es una característica importante en el diseño de sistemas multiprocesador. Los sistemas basados en buses no son escalables al convertirse el bus en un cuello de botella cuando se añaden más procesadores. Las *redes directas* o *redes punto a punto* son una arquitectura red popular y que escalan bien incluso con un número elevado de procesadores. Las redes directas consisten en un conjunto de *nodos*, cada uno directamente conectado a un subconjunto (usualmente pequeño) de otros nodos en la red. En la figura 5.6 se muestra varios tipos de redes directas. Los correspondientes patrones de interconexión se estudiarán posteriormente. Cada nodo es un ordenador programable con su propio procesador, memoria local, y otros dispositivos. Estos nodos pueden tener diferentes capacidades funcionales. Por ejemplo, el conjunto de nodos puede contener procesadores vectoriales, procesadores gráficos, y procesadores de E/S. La figura 5.5 muestra la arquitectura de un nodo genérico. Un componente común en estos nodos es un *encaminador* (router) que se encarga de manejar la comunicación entre los nodos a través del envío y recepción de mensajes. Por esta razón, las redes directas también son conocidas como redes basadas en routers. Cada router tiene conexión directa con

el router de sus vecinos. Normalmente, dos nodos vecinos están conectados por un par de canales unidireccionales en direcciones opuestas. También se puede utilizar un canal bidireccional para conectar dos nodos. Aunque la función del encaminador puede ser realizada por el procesador local, los encaminadores dedicados se han usado de forma habitual en multicomputadores de altas prestaciones, permitiendo el solapamiento de la computación y las comunicaciones dentro de cada nodo. Al aumentar el número de nodos en el sistema, el ancho de banda total de comunicaciones, memoria y capacidad de procesamiento del sistema también aumenta. Es por esto que las redes directas son una arquitectura popular para construir computadores paralelos de gran escala.

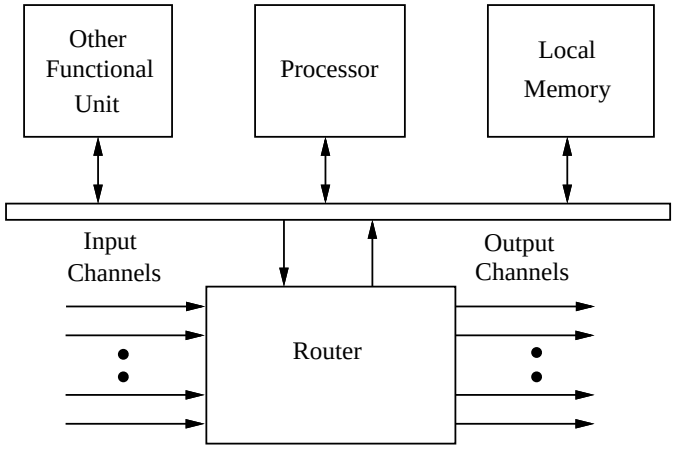


Figura 5.5: Arquitectura de un nodo genérico.

Cada encaminador soporta un número de canales de entrada y salida. Los canales *internos* o *puertos* conectan el procesador/memoria local al encaminador. Aunque normalmente existe únicamente un par de canales internos, algunos sistemas usan más canales internos para evitar el cuello de botella entre el procesador/memoria local y el router. Los canales *externos* se usan para comunicaciones entre los routers. Conectando los canales de entrada de un nodo a los canales de salida del otros nodos podemos definir la red directa. A no ser que se diga lo contrario, utilizaremos el término "canal" para referirnos a un canal externo. A dos nodos conectados directamente se les llaman *vecinos* o nodos *adyacentes*. Normalmente, cada nodo tiene un número fijo de canales de entrada y salida, cada canal de entrada está emparejado con el correspondiente canal de salida. A través de todas las conexiones entre estos canales, existen varias maneras de conectar los diferentes nodos. Obviamente, cada nodo de la red debe poder alcanzar cualquier otro nodo.

5.2.3 Redes Indirectas

Las *redes indirectas* o *basadas en conmutadores* (*switch*) forman el tercer grupo de redes de interconexión. En lugar de proporcionar una conexión directa entre algunos nodos, la comunicación entre cualquier pareja de nodos se realiza a través de *conmutadores*. Cada nodo tiene un adaptador de red que se conecta a un conmutador. Cada conmutador consta de un conjunto de *puertos*. Cada puerto consta de un enlace de entrada y otro de salida. Un conjunto (posiblemente vacío) de puertos en cada conmutador están conectados a los procesadores o permanecen abiertos, mientras que el resto de puertos

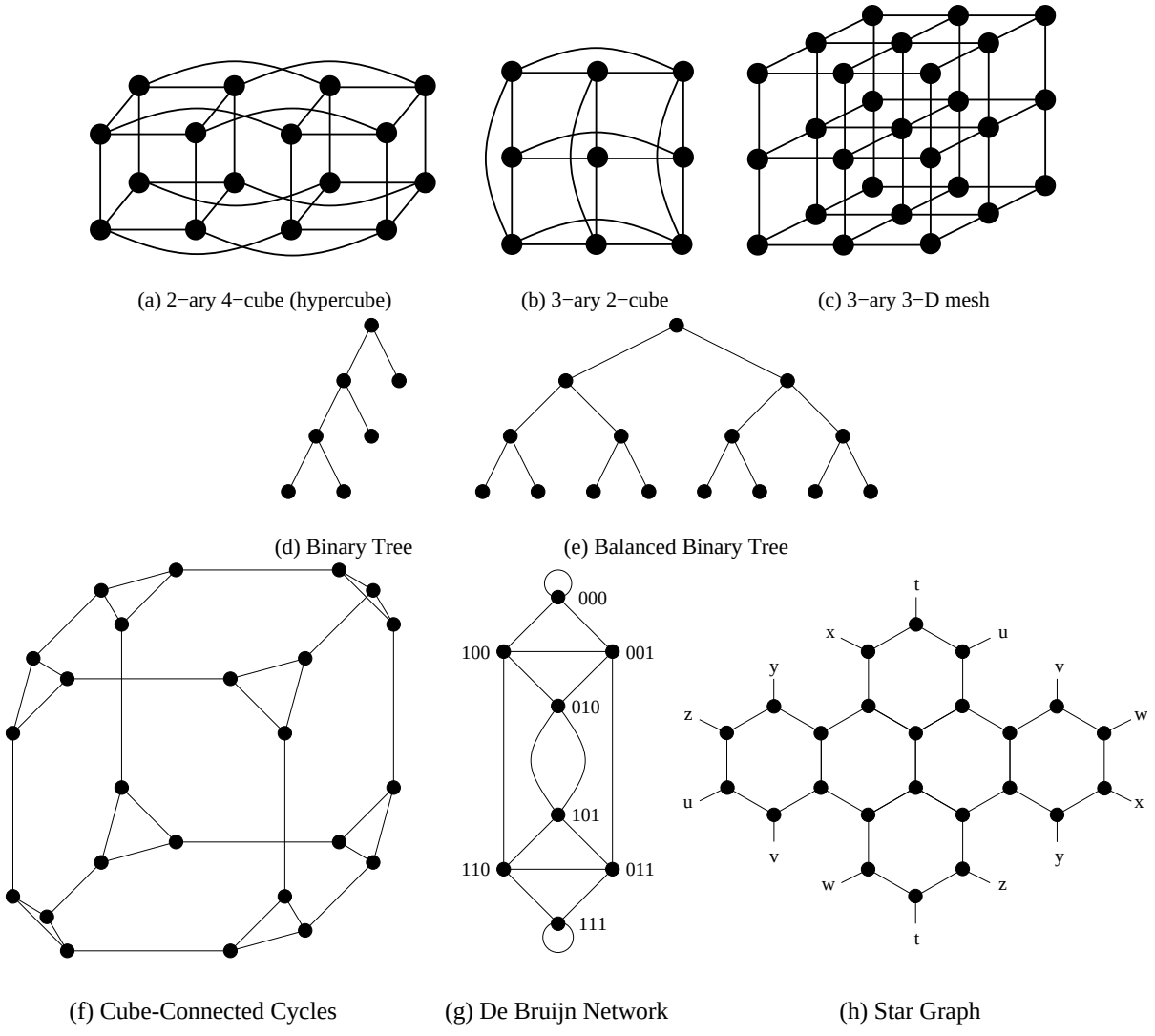


Figura 5.6: Algunas topologías propuestas para redes directas.

están conectados a puertos de otros conmutadores para proporcionar conectividad entre los procesadores. La interconexión de estos conmutadores define la topología de la red.

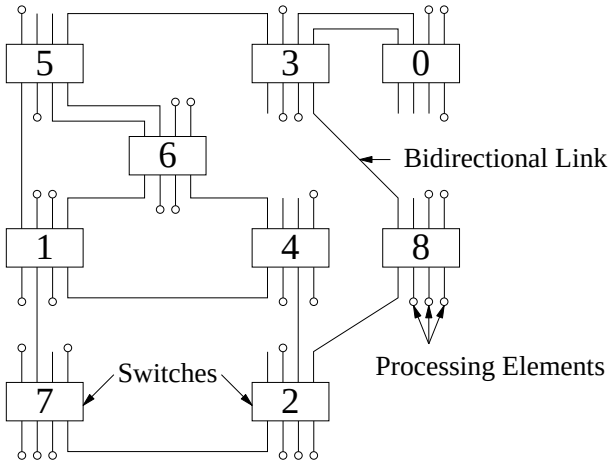


Figura 5.7: Un red conmutada con topología irregular.

Las redes conmutadas han evolucionado considerablemente con el tiempo. Se han propuesto un amplio rango de topologías, desde topologías regulares usadas en los procesadores array (matriciales) y multiprocesadores de memoria compartida UMA a las topologías irregulares utilizadas en la actualidad en los NOWs. Las topologías regulares tienen patrones de conexión entre los conmutadores regulares mientras que las topologías irregulares no siguen ningún patrón predefinido. En el tema siguiente se estudiarán más detenidamente las topologías regulares. La figura 5.7 muestra una red conmutada típica con topología irregular. Ambos tipos de redes pueden clasificarse además según el número de conmutadores que tiene que atravesar un mensaje para llegar a su destino. Aunque esta clasificación no es importante en el caso de topologías irregulares, puede significar una gran diferencia en el caso de redes regulares ya que algunas propiedades específicas se pueden derivar en función de este dato.

5.2.4 Redes Híbridas

En esta sección describiremos brevemente algunas topologías que no se encuadran en las vistas hasta ahora. En general, las redes híbridas combinan mecanismos de redes de medio compartido y redes directas o indirectas. Por tanto, incrementan el ancho de banda con respecto a las redes de medio compartido, y reducen la distancia entre nodos con respecto a las redes directas e indirectas. Existen varias aplicaciones concretas de las redes híbridas. Este es el caso de las *bridged LANs.* Sin embargo, para sistemas que necesitan muy alto rendimiento, las redes directas e indirectas consiguen una mejor escalabilidad que las redes híbridas ya que los enlaces punto a punto son más sencillos y más rápidos que los buses de medio compartido. La mayoría de los computadores paralelos de alto rendimiento usan redes directas o indirectas. Recientemente las redes híbridas han ganado aceptación de nuevo. El uso de tecnología óptica permite la implementación de redes de alto rendimiento.

Se han propuesto redes híbridas para diferentes propósitos. En general, las redes híbridas pueden modelarse mediante hipergrafos, donde los vértices del hipergrafo representan un conjunto de nodos de procesamiento, y las aristas representan el conjunto

de canales de comunicación y/o buses. Obsérvese que una arista en un hipergrafo puede conectar un número arbitrario de nodos. Cuando una arista conecta exactamente dos nodos entonces representa un canal punto a punto. En caso contrario representa un bus. En algunos diseños de redes, cada bus tiene un único nodo conductor. Ningún otro dispositivo puede utilizar ese bus. En ese caso, no existe necesidad de arbitraje. Sin embargo, todavía es posible la existencia de varios receptores en un tiempo dado, manteniéndose la capacidad de *broadcast* de los buses. Obviamente, cada nodo de la red debe poder controlar al menos un bus, por lo que el número de buses necesarios no puede ser menor que el número de nodos. En este caso, la topología de la red se puede modelar mediante un hipergrafo directo.

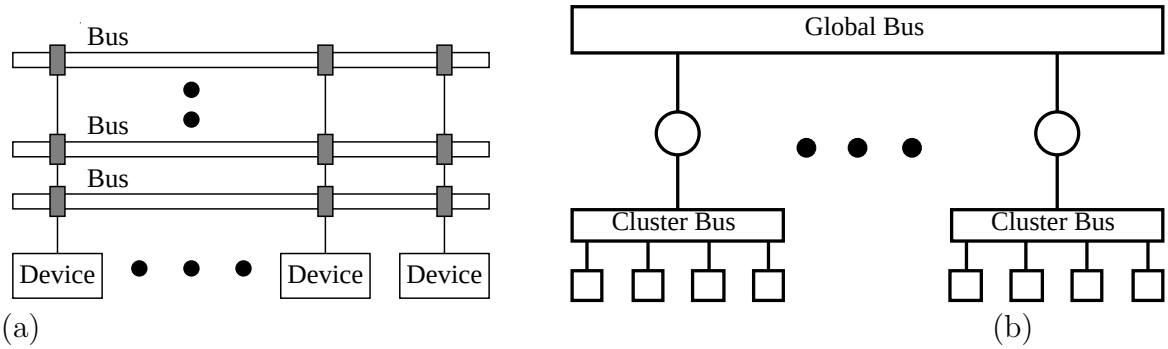


Figura 5.8: Redes Híbridas. (a) Una red multibus. (b) Una jerarquía de dos niveles de buses.

Redes multibus

Debido al ancho de banda limitado que proporcionan las redes de medio compartido, éstas sólo son capaz de soportar un número pequeño de dispositivos, tiene una distancia limitada, y no es escalable. Algunos investigadores han estudiado cómo podría eliminarse ese cuello de botella. Una aproximación para incrementar el ancho de banda de la red se muestra en la figura 5.8a. Sin embargo, los problemas de cableado y el coste del interface hacen que sea de poca utilidad para el diseño de multiprocesadores. Debido a las limitaciones eléctricas de la tecnología del encapsulado, es poco probable tener una red multibus con más de cuatro buses. Sin embargo, utilizando otras tecnologías de empaquetado como la multiplexación por división de la longitud de onda en fibra óptica hace posible integración de múltiples buses.

Redes jerárquicas

Otra aproximación para incrementar el ancho de banda de la red es la que se muestra en la figura 5.8b. Diferentes buses se interconectan mediante routers o puentes para transferir información de un lado a otro de la red. Estos routers o puentes pueden filtrar el tráfico de la red examinando la dirección destino de cada mensaje que le llegue. La red jerárquica permite expandir el área de la red y manejar más dispositivos, pero deja de ser una red de medio compartido. Esta aproximación se usa para interconectar varias LANs. Normalmente, el bus global tiene un mayor ancho de banda. En caso contrario, se convertiría en un cuello de botella. Esto se consigue utilizando una tecnología más rápida. Las redes jerárquicas han sido también propuestas como esquema

de interconexión para los multiprocesadores de memoria compartida. También en este caso, el bus global puede convertirse en un cuello de botella.

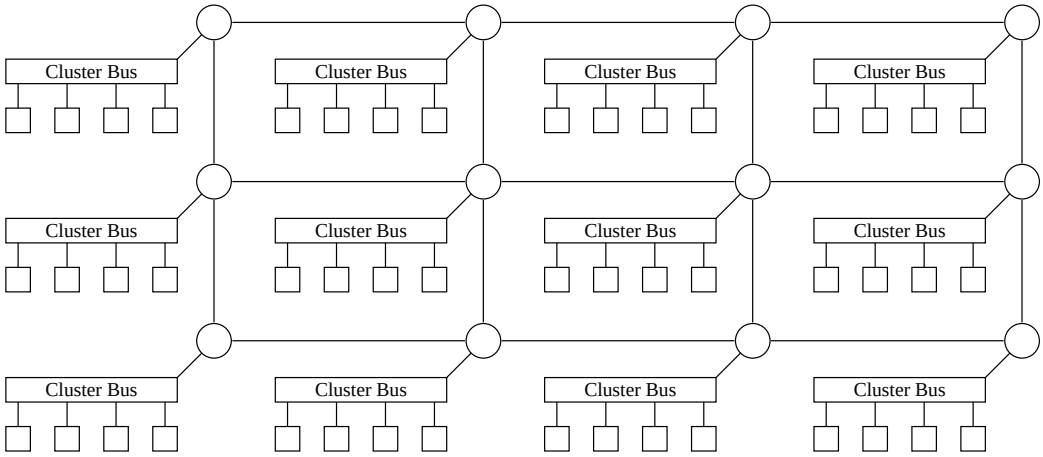


Figura 5.9: Malla bidimensional basada en clusters.

Redes basadas en clusters

Las redes basadas en clusters también tienen una estructura jerárquica. Incluso pueden considerarse como una subclase dentro de las redes jerárquicas. Estas redes combinan las ventajas de dos o más clases de redes a diferentes niveles en la jerarquía. Por ejemplo, es posible combinar las ventajas de los buses y los enlaces punto a punto usando buses en el nivel más bajo de la jerarquía para formar clusters, y una topología directa para conectar los clusters en el nivel superior. Este es el caso del computador paralelo DASH (*Stanford Directory Architecture for Shared-Memory*). La figura 5.9 muestra la arquitectura básica del este computador paralelo. En el nivel inferior cada cluster consta de cuatro procesadores conectados mediante un bus. En el nivel superior, una malla 2-D conecta los clusters. La capacidad *broadcast* del bus se usa a nivel de cluster para implementar un protocolo *snoopy* para mantener la coherencia de la caché. La red directa en el nivel superior elimina la limitación del ancho de banda de un bus, incrementando considerablemente la escalabilidad de la máquina.

Es posible realizar otras combinaciones. En lugar de combinar buses y redes directas, el multiprocesador HP/Convex Exemplar combinan redes directas e indirectas. Este multiprocesador consta de crossbar no bloqueantes de tamaño 5×5 en en nivel inferior de la jerarquía, conectando cuatro bloques funcionales y un interface de E/S para forma un cluster o *hipernodo.* Cada bloque funcional consta de dos procesadores, dos bancos de memoria e interfaces. Estos hipernodos se conectan a un segundo nivel denominado *interconexión toroidal coherente* formada por múltiples anillos usando el Interface de Coherencia Escalable (SCI). Cada anillo conecta un bloque funcional de todos hipernodos. En el nivel inferior de la jerarquía, el crossbar permite a todos los procesadores dentro del hipernodo acceder a los módulos de memoria entrelazada de ese hipernodo. En el nivel superior, los anillos implementan un protocolo de coherencia de la caché.

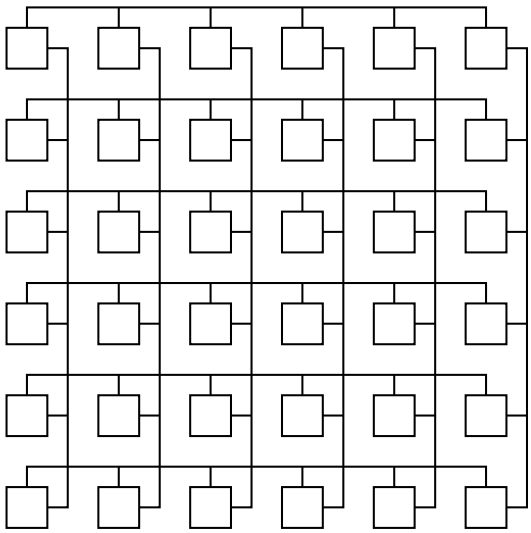


Figura 5.10: Una hipermalla bidimensional.

Otras topologías hipergrafo

Se han propuesto muchas otras topologías híbridas. Entre ellas, una clase particularmente interesante en la de *hipermallas*. Una hipermalla es una topología regular basada en un conjunto de nodos dispuestos en varias dimensiones. En lugar de existir conexiones directas entre los vecinos de cada dimensión, cada nodo está conectado a todos lo nodos de la dimensión a través de un bus. Existen varias formas de implementar una hipermalla. La más directa consiste en conectar todos los nodo de cada dimensión a través de un bus compartido. La figura 5.10 muestra una hipermalla 2-D. En esta red, los buses están dispuestos en dos dimensiones. Cada nodo se conecta a un bus en cada dimensión. Esta topología fue propuesta por Wittie[1], y se le denomina *spanning-bus hypercube*. La misma tiene un diámetro muy pequeño, y la distancia media entre nodos escala muy bien en función del tamaño de la red. Sin embargo, el ancho de banda total no escala bien. Además, los frecuentes cambios en el maestro del bus causan una sobrecarga significativa.

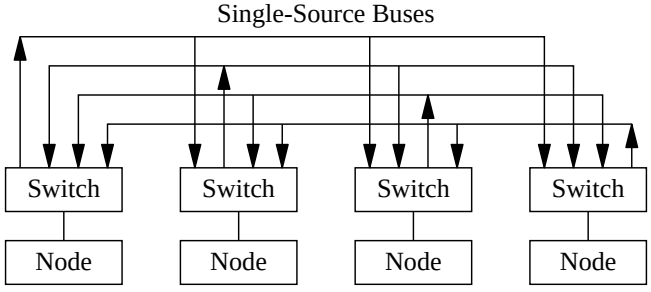


Figura 5.11: Una hipermalla unidimensional con conmutador crossbar distribuido.

Una implementación alternativa que elimina las restricciones señaladas arriba consiste en reemplazar el bus compartido que conecta los nodos a lo largo de una dimensión

[1]L. D. Wittie. *Communications structures for large networks of microcomputers.* IEEE Transactions on Computers, vol C-29, pp. 694-702, August 1980.

dada por un conjunto de tantos buses como nodos existan en esa dimensión. Esta es la aproximación propuesta en la *Distributed Crossbar Switch Hypermesh* (DCSH). La figura 5.11 muestra una dimensión de la red. Cada bus es controlado por un único nodo. Por tanto, no hay cambios en la pertenencia del bus. Además, el ancho de banda escala con el número de nodos. Los dos principales problemas, sin embargo, son el alto número de buses necesarios y el alto número de puertos de entrada y salida que necesita cada nodo.

Capítulo 6

Multiprocesadores

Los multiprocesadores son sistemas MIMD basados en varios procesadores funcionando de forma paralela e independiente. La principal característica de los multiprocesadores es que la memoria está compartida, es decir, todos los procesadores comparten el mismo espacio de direccionamiento.

El libro [CSG99] contiene la mayor parte de los temas tratados en este capítulo, además se trata de un libro muy completo sobre multiprocesadores.

6.1 Redes de interconexión para multiprocesadores

6.1.1 Redes de medio compartido. Buses

Un *bus* de sistema está formado por un conjunto de conductores para la transacción de datos entre procesadores, módulos de memoria, y dispositivos periféricos conectados al bus. En el bus sólo puede haber una transacción a un tiempo entre una fuente (maestro) y uno o varios destinos (esclavos). En el caso de que varios maestros quieran realizar transacciones, la lógica de arbitraje del bus debe decidir quien será el siguiente que consiga el bus y realice la transacción.

Por esta razón a los buses digitales se les llama *buses de contención* o *buses de tiempo compartido*. Un sistema basado en bus tiene un coste bajo comparado con otros sistemas de conexión. Su uso está muy extendido en la industria y existen varios estándares del IEEE disponibles.

El bus es un camino de comunicaciones común entre procesadores, memoria y los subsistemas de entrada/salida. El bus se realiza en la mayoría de los casos sobre una placa de circuito impreso. Las tarjetas con los procesadores, memorias, etc. se conectan a este circuito impreso o placa madre a través de conectores o cables.

El multiprocesador basado en bus es uno de los sistemas multiprocesador más utilizados en computadores de prestaciones medias. Ello es debido a su bajo coste, facilidad de diseño, etc. El principal problema que tienen es su baja escalabilidad lo que no permite tener sistemas con muchos procesadores de forma eficiente. Dependiendo del ancho de banda del bus y de los requisitos de los procesadores que incorpora, un bus puede albergar entre 4 y 16 procesadores de forma eficiente. Por encima de estos números máximos, que dependen del procesador y el bus, el canal de conexión, en este caso el

bus, se convierte en el cuello de botella del sistema.

A pesar de esto, y dado que los requisitos de la mayoría de los sistemas informáticos no necesitan de muchos procesadores, se suele utilizar el sistema basado en bus por su bajo coste y facilidad de diseño. Incluso en sistemas más complejos se sigue utilizando el bus, de manera que para aumentar la escalabilidad del sistema se disponen varios buses formando una jerarquía, o bien, se interconectan varios buses entre sí a través de una red.

Hay varios estándares de buses disponibles para el diseñador. Muchas veces estos buses han surgido a partir de un sistema concreto y se ha intentado luego que fuera lo más estándar posible con los problemas que este tipo de política suele acarrear. Ejemplos de esto los encontramos en el bus del PC, que actualmente es obsoleto, o el popular VME con sus sucesivas extensiones que sigue siendo, a pesar de todo, el bus más utilizado en sistemas empotrados a medida.

Para evitar los problemas del paso del tiempo, y garantizar la portabilidad del bus independientemente del procesador, han aparecido en los últimos tiempos buses que son independientes de una arquitectura específica y que además han sido ideados para hacer frente a los sucesivos avances tecnológicos. Un ejemplo de bus de este tipo es el bus de altas prestaciones Futurebus+, que es el estándar 896 del IEEE. Este bus fue creado por un grupo de trabajo a partir de una hoja en blanco, sin tener en cuenta ningún procesador en particular, y con la intención de definir un bus de muy altas prestaciones que hiciera frente a sucesivos avances tecnológicos. Esta labor no ha sido sencilla y de hecho se ha tardado entre 10 y 15 años, desde que surgió la idea, en tener una definición completa del bus.

6.1.2 Redes indirectas

Para aplicaciones de propósito general es necesario el uso de conexiones dinámicas que puedan soportar todos los patrones de comunicación dependiendo de las demandas del programa. En vez de usar conexiones fijas, se utilizan conmutadores y árbitros en los caminos de conexión para conseguir la conectividad dinámica. Ordenados por coste y rendimiento, las redes dinámicas principales son los buses, las redes de conexión multietapa (MIN *Multistage Interconnection Network*), y las redes barras de conmutadores.

El precio de estas redes es debido al coste de los cables, conmutadores, árbitros, y conectores. El rendimiento viene dado por el ancho de banda de la red, la tasa de transmisión de datos, la latencia de la red, y los patrones de comunicación soportados.

Las redes indirectas se pueden modelar mediante un grafo $G(N, C)$ donde N es el conjunto de conmutadores, y C es el conjunto de enlaces unidireccionales o bidireccionales entre conmutadores. Para el análisis de la mayoría de las propiedades, no es necesario incluir los nodos de procesamiento en el grafo. Este modelo nos permite estudiar algunas propiedades interesantes de la red. Dependiendo de las propiedades que se estén estudiando, un canal bidireccional podrá ser modelado como un línea o como dos arcos en direcciones opuestas (dos canales unidireccionales).

Cada conmutador en una red indirecta puede estar conectado a cero, uno, o más procesadores. Únicamente los conmutadores conectados a algún procesador pueden ser el origen o destino de un mensaje. Además, la transmisión de datos de un nodo a otro requiere atravesar el enlace que une el nodo origen al conmutador, y el enlace entre el último conmutador del camino recorrido por el mensaje y el nodo destino. Por lo tanto,

la *distancia entre dos nodos* es la distancia entre los conmutadores que conectan esos nodos más dos. De manera similar, el *diámetro* de la red, definido como la máxima distancia entre dos nodos de la red, es la máxima distancia entre dos conmutadores conectados a algún nodo más dos. Obsérvese que la distancia entre dos nodos conectados a través de un único conmutador es dos.

Como ya se vio en el capítulo dedicado a las redes, las redes de interconexión, y en el caso que nos ocupa ahora, las redes indirectas, pueden caracterizarse por tres factores: topología, encaminamiento y conmutación. La topología define cómo los conmutadores están interconectados a través de los canales, y puede modelarse con un grafo como el indicado anteriormente. Para una red indirecta con N nodos, la topología ideal conectaría esos nodos a través de un único conmutador de $N \times N$. A dicho conmutador se le conoce con el nombre de *crossbar*. Sin embargo, el número de conexiones físicas de un conmutador está limitado por factores hardware tales como el número de pins disponibles y el la densidad máxima del cableado. Estas dificultades imposibilitan el uso de crossbar en redes de gran tamaño. Como consecuencia, se ha propuesto un gran número de topologías alternativas.

En esta sección nos centraremos en las diferentes topologías existentes de redes indirectas, así como los algoritmos de encaminamiento utilizados en estas redes.

6.1.3 Red de barra cruzada

El mayor ancho de banda y capacidad de interconexión se consigue con la red de barra cruzada. Una red de barra cruzada se puede visualizar como una red de una sola etapa de conmutación. Los conmutadores de cada cruce dan las conexiones dinámicas entre cada par destino-fuente, es decir, cada conmutador de cruce puede ofrecer un camino de conexión dedicado entre un par. Los conmutadores se pueden encender o apagar (*on/off*) desde el programa. Una barra cruzada genérica de conmutadores se muestra en la figura 6.1, donde los elementos V (vertical) y H (horizontal) pueden ser indistintamente procesadores, memorias, etc. Por ejemplo son típicas las configuraciones de procesador con memoria compartida, donde los módulos verticales son todo procesadores y los horizontales memorias, o al revés.

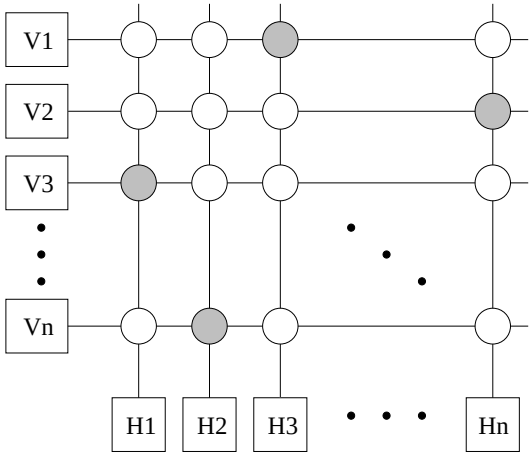


Figura 6.1: La red de conmutación en barra cruzada.

Hay que hace notar, para el caso de multiprocesadores con memoria compartida,

que un módulo de memoria sólo puede satisfacer una petición del procesador cada vez. Cuando varias peticiones llegan al mismo módulo de memoria, la propia red debe resolver el conflicto; el comportamiento de cada barra en el conmutador de barra cruzada es muy parecido al de un bus. Sin embargo, cada procesador puede generar una secuencia de direcciones para acceder a varios módulos de memoria de forma simultánea. Por lo tanto, tal como se muestra en la figura 6.1 suponiendo que los V son procesadores y los H memorias, sólo un conmutador puede activarse en cada columna. Sin embargo, varios conmutadores en diferentes columnas se pueden activar simultáneamente para soportar accesos a memoria paralelos o entrelazados.

Otra aplicación de la red de barra cruzada es la comunicación entre procesadores. La barra cruzada entre procesadores permite conexiones por permutación entre procesadores. Sólo conexiones uno a uno son permitidas, por lo tanto, una barra cruzada $n \times n$ conecta como mucho n pares a un tiempo. Esta es la diferencia con la barra cruzada entre procesadores y memoria. Ambos tipos de barras cruzadas tienen operaciones y propósitos diferentes.

Las redes de barras cruzadas (*crossbar*) permiten que cualquier procesador del sistema se conecte con cualquier otro procesador o unidad de memoria de tal manera que muchos procesadores pueden comunicarse simultáneamente sin contención. Es posible establecer una nueva conexión en cualquier momento siempre que los puertos de entrada y salida solicitados estén libres. Las redes de crossbar se usan en el diseño de multiprocesadores de pequeña escala pero alto rendimiento, en el diseño de *routers* para redes directas, y como componentes básicos en el diseño de redes indirectas de gran escala. Un crossbar se puede definir como una redes conmutada con N entradas y Msalidas, que permite hasta $\min\{N, M\}$ interconexiones punto a punto sin contención. La figura 6.1 muestra un red crossbar de $N \times N$. Aunque lo normal es que N y M sean iguales, no es extraño encontrar redes en las que N y M difieren, especialmente en los crossbar que conectan procesadores y módulos de memoria.

El coste de una red de este tipo es $O(NM)$, lo que hace que sea prohibitiva para valores grandes de N y M. Los crossbars han sido utilizados tradicionalmente en multiprocesadores de memoria compartida de pequeña escala, donde todos los procesadores pueden acceder a la memoria de forma simultánea siempre que cada procesador lea de, o escriba en, un modulo de memoria diferente. Cuando dos o más procesadores compiten por el mismo módulo de memoria, el arbitraje deja proceder a un procesador mientras que el otro espera. El árbitro en un crossbar se distribuye entre todos los puntos de conmutación que conectan a la misma salida. Sin embargo, el esquema de arbitración puede ser menos complejo que en el caso de un bus ya que los conflictos en un crossbar son la excepción más que la regla, y por tanto más fáciles de resolver.

Para una red de barras cruzadas con control distribuido, cada punto de conmutación puede estar en uno de los cuatro estados que se muestran en la figura 6.2. En la figura 6.2a, la entrada de la fila en la que se encuentra el punto de conmutación tiene acceso a la correspondiente salida mientras que las entradas de las filas superiores que solicitaban la misma salida están bloqueadas. En la figura 6.2b, a una entrada de una fila superior se le ha permitido el acceso a la salida. La entrada de la fila en la que se encuentra el punto de conmutación no ha solicitado dicha salida, y puede ser propagada a otros conmutadores. En la figura 6.2c, una entrada de una fila superior tiene acceso a la salida. Sin embargo, la entrada de la columna en la que se encuentra en punto de conmutación también ha solicitado esa salida y está bloqueada. La configuración de la figura 6.2(d) sólo es necesaria si el crossbar tiene soporte para comunicaciones *multicast* (uno a muchos).

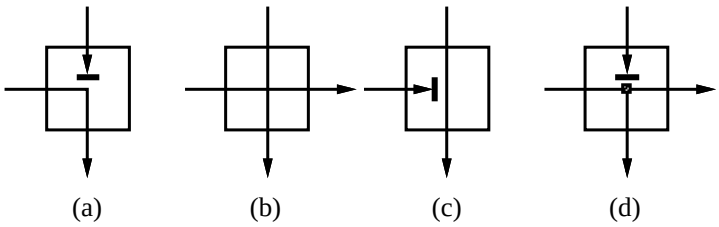


Figura 6.2: Estados de un punto de conmutación en una red de barras cruzadas.

Los avances en VLSI permiten la integración del hardware para miles de conmutadores en un único chip. Sin embargo, el número de pines de un chip VLSI no puede excederse de algunos centenares, lo que restringe el tamaño del mayor crossbar que puede integrarse en un único chip VLSI. Crossbars de mayor tamaño se pueden conseguir mediante la partición del mismo en otros de menor tamaño, cada uno de ellos implementado usando un único chip. Por lo tanto, un crossbar de $N \times N$ se puede implementar con $(N/n)(N/n)$crossbar de tamaño $n \times n$.

6.1.4 Redes de interconexión multietapa (MIN)

Los MIN (*Multistage Interconnection Networks* se han utilizado tanto en máquinas MIMD como SIMD. La figura 6.3 muestra una red multietapa generalizada. Un número de $a \times b$ conmutadores se usa en cada etapa. Entre etapas adyacentes se utiliza una red de interconexión fija. Los conmutadores pueden ser programados dinámicamente para establecer las conexiones deseadas entre las entradas y salidas.

Las *redes de interconexión multietapa* (MINs) conectan dispositivos de entrada a dispositivos de salida a través de un conjunto de etapas de conmutadores, donde cada conmutador es una red de barra cruzada. El número de etapas y los patrones de conexión entre etapas determinan la capacidad de encaminamiento de las redes.

Las MINs fueron inicialmente propuestas por las compañías de teléfonos y posteriormente para los procesadores matriciales. En estos casos, un controlador central establece el camino entre la entrada y la salida. En casos en donde el número de entradas es igual al número de salidas, cada entrada puede transmitir de manera síncrona un mensaje a una salida, y cada salida recibir un mensaje de exactamente una entrada. Este patrón de comunicación *unicast* puede representarse mediante una permutación de la dirección asociada a la entrada. Debido a esta aplicación, las MINs se han popularizado como redes de alineamiento que permiten acceder en paralelo a arrays almacenados en bancos de memoria. El almacenamiento del array se divide de tal manera que se permita un acceso sin conflictos, y la red se utiliza para reordenar el array durante el acceso. Estas redes pueden configurarse con un número de entradas mayor que el número de salidas (concentradores) y viceversa (expansores). Por otra parte, en multiprocesadores asíncronos, el control centralizado y el encaminamiento basado en la permutación es inflexible. En este caso, se requiere un algoritmo de encaminamiento que establezca un camino a través de los diferentes estados de la MIN.

Dependiendo del esquema de interconexión empleado entre dos estados adyacentes y el número de estados, se han propuesto varias MINs. Las MINs permiten la construcción de multiprocesadores con centenares de procesadores y han sido utilizadas en algunas máquinas comerciales.

Un modelo MIN generalizado

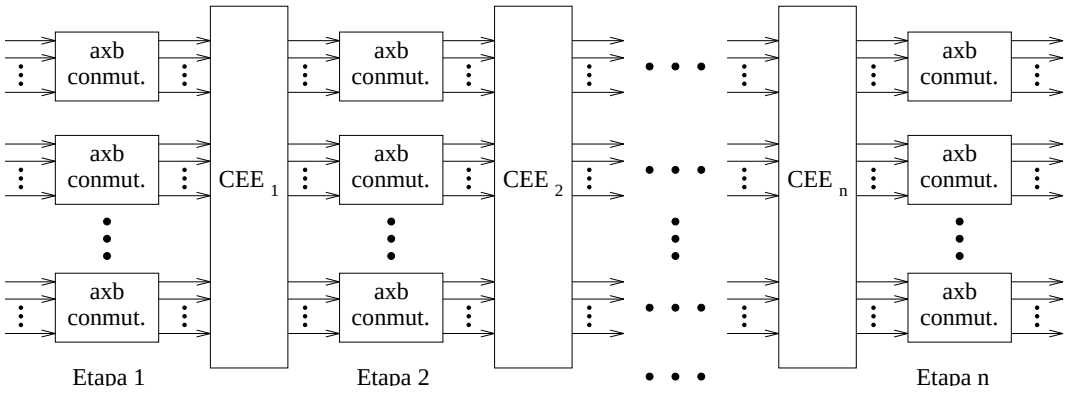


Figura 6.3: Estructura generalizada de una interconexión multietapa (MIN).

Existen muchas formas de interconectar etapas adyacentes. La figura 6.3 muestra una red de interconexión general con N entradas y M salidas. La red consta de g *etapas*, G_0 a G_{g-1}. Como se muestra en la figura 6.3, cada etapa, G_i, tiene w_i conmutadores de tamaño $a_{i,j} \times b_{i,j}$, donde $1 \leq j \leq w_i$. Así, la etapa $G_i$consta de p_i entradas y q_i salidas, donde

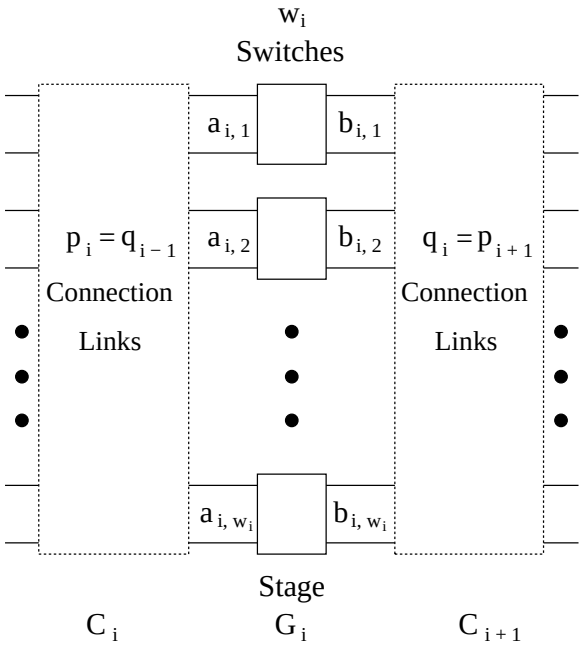


Figura 6.4: Visión detallada de una etapa G_i.

$$p_i = \sum_{j=1}^{w_i} a_{i,j} \quad \text{y} \quad q_i = \sum_{j=1}^{w_i} b_{i,j}$$

La conexión entre dos etapas adyacentes G_{i-1} y G_i, denotada por C_i, define el *patrón de conexión* para $p_i = q_{i-1}$ enlaces, donde $p_0 = N$ y $q_{g-1} = M$. Por lo tanto, una MIN

puede representarse como

$$C_0(N)G_0(w_0)C_1(p_1)G_1(w_1)\ldots G_{g-1}(w_{g-1})C_g(M)$$

Un patrón de conexión $C_i(p_i)$ define cómo deben de estar conectados los enlaces p_i entre las $q_{i-1} = p_i$ salidas de la etapa G_{i-1}y las p_i entradas de la etapa G_i. Patrones diferentes de conexión dan lugar a diferentes características y propiedades topológicas de las MINs. Los enlaces de C_i están numerados de 0 a $p_i - 1$.

Desde un punto de vista práctico, es interesante que todos los conmutadores sean idénticos, para amortizar el coste de diseño. Las redes Banyan son una clase de MINs con la propiedad de que existe un único camino entre cada origen y destino. Una red Delta N-nodo es una subclase de red Banyan, que se construye a partir de $k \times k$ conmutadores en n etapas. donde cada etapa contiene $\frac{N}{k}$ conmutadores. Muchas de las redes multietapa más conocidas, como la Omega, Inversa, Cubo, Mariposa, y de Línea base , pertenecen a la clase de redes Delta, habiéndose demostrado ser topológica y funcionalmente equivalentes.

En el caso de que los conmutadores tengan el mismo número de puertos de entrada y salida, las MINs también tendrán el mismo número de puertos de entrada y salida. Dado que existe una correspondencia uno a uno entre las entradas y las salidas, a estas conexiones se les denominan *permutaciones*. A continuación definiremos cinco permutaciones básicas. Aunque estas permutaciones fueron originariamente definidas para redes con conmutadores 2×2, la mayoría de las definiciones se pueden extender pare redes con conmutadores $k \times k$ y que tienen $N = k^n$ entradas y salidas, donde n es un entero. Sin embargo, algunas de estas permutaciones sólo están definidas para el caso de que N sea potencia de dos. Con $N = k^n$ puertos, sea $X = x_{n-1}x_{n-2}\ldots x_0$ la codificación de un puerto arbitrario, $0 \leq X \leq N - 1$, donde $0 \leq x_i \leq k - 1$, $0 \leq i \leq n - 1$.

6.1.5 Tipos de etapas de permutación para MIN

Las diferentes clases de redes multietapa se diferencian en los módulos conmutadores empleados y en los patrones de la *conexión entre etapas* (CEE). El módulo conmutador más simple es el 2×2, y los patrones para la CEE más usados suelen ser el barajado perfecto, mariposa, barajado multivía, barra cruzada, conexión cubo, etc. Veamos a continuación algunos de estos patrones fijos de interconexión.

Conexión de barajado perfecto

El patrón de barajado perfecto tiene un amplio campo de aplicación en las interconexiones multietapa. Fue originalmente propuesto para calcular la transformada rápida de Fourier. La permutación entre los elementos de entrada y salida de la red está basada en la mezcla perfecta de dos montones de cartas iguales que consiste en intercalar una a una las cartas de un montón con las del otro montón. La red de barajado perfecto toma la primera mitad de las entradas y la entremezcla con la segunda mitad, de manera que la primera mitad pasa a las posiciones pares de las salidas, y la segunda mitad a las impares.

La permutación *k-baraje perfecto*, σ^k, se define por

$$\sigma^k(X) = (kX + \left\lfloor \frac{kX}{N} \right\rfloor) \bmod N$$

Un modo más sólido de describir dicha conexión es

$$\sigma^k(x_{n-1}x_{n-2}\ldots x_1x_0) = x_{n-2}\ldots x_1x_0x_{n-1}$$

La conexión *k-baraje perfecto* realiza un desplazamiento cíclico hacia la izquierda de los dígitos de X en una posición. Para $k = 2$, esta acción se corresponde con el barajado perfecto de una baraja de N cartas, como se demuestra en la figura 6.5a para el caso de $N = 8$. El baraje perfecto corta la baraja en dos partes y las entremezcla empezando con la segunda parte. El *baraje perfecto inverso* realiza la acción contraria como se define a continuación:

$$\sigma^{k^{-1}}(x_{n-1}x_{n-2}\ldots x_1x_0) = x_0x_{n-1}\ldots x_2x_1$$

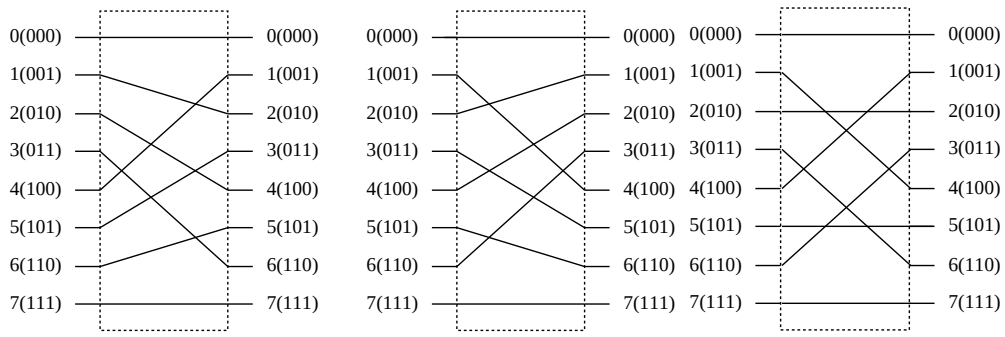


Figura 6.5: Barajado perfecto, barajado perfecto inverso, y bit reversal para $N = 8$.

Suponiendo que las entradas tienen la forma $a_{n-1}a_{n-2}\ldots a_1a_0$, donde a_i es el bit i-ésimo de la dirección de cada entrada, el barajado perfecto realiza la siguiente transformación en la dirección:

$$\text{Baraja}(a_{n-1}a_{n-2}\ldots a_1a_0) = a_{n-2}a_{n-3}\ldots a_1a_0a_{n-1} \tag{6.1}$$

es decir, los bits de la dirección son desplazados cíclicamente una posición hacia la izquierda. La inversa del barajado perfecto los mueve cíclicamente hacia la derecha.

Conexión de dígito inverso (*Digit Reversal Connection*)

La permutación de *dígito inverso* ρ_k está definida por

$$\rho^k(x_{n-1}x_{n-2}\ldots x_1x_0) = x_0x_1\ldots x_{n-2}x_{n-1}$$

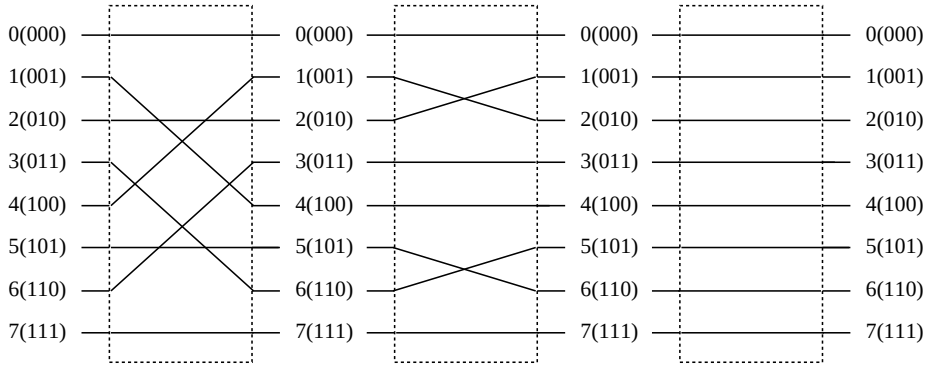
A esta permutación se le suele denominar *bit inverso*, indicando claramente que fue propuesta para $k = 2$. Sin embargo, la definición es también válida para $k > 2$. La figura 6.5c muestra una conexión de bit inverso para el caso $k = 2$ y $N = 8$.

Conexión mariposa

La i-ésima k-aria permutación mariposa β_i^k, para $0 \leq i \leq n-1$, se define como

$$\beta_i^k(x_{n-1} \ldots x_{i+1} x_i x_{i-1} \ldots x_1 x_0) = x_{n-1} \ldots x_{i+1} x_0 x_{i-1} \ldots x_1 x_i$$

La i-ésima permutación mariposa intercambia los dígitos 0 e i-ésimo del índice. La figura 6.6 muestra la conexión mariposa para $k = 2$, $i = 0, 1, 2$ y $N = 8$. Observar que β_0^k define una conexión uno a uno denominada conexión identidad, I.



(a) Second Butterfly (b) First Butterfly (c) Zeroth Butterfly

Figura 6.6: La conexión mariposa para $N = 8$.

Conexión Cubo

La i-ésima conexión *cubo* E_i, para $0 \leq i \leq n-1$, se define únicamente para $k = 2$, por

$$E_i(x_{n-1} \ldots x_{i+1} x_i x_{i-1} \ldots x_0) = x_{n-1} \ldots x_{i+1} \overline{x}_i x_{i-1} \ldots x_0$$

La i-ésima conexión cubo complementa el i-ésimo bit del índice. La figura 6.7 muestra la conexión cubo para $i = 0, 1, 2$ y $N = 8$. A E_0 también se le denomina conexión *intercambio*.

Conexión en Línea Base

La i-ésima k-aria permutación en línea base δ_i^k, para $0 \leq i \leq n-1$, se define por

$$\delta_i^k(x_{n-1} \ldots x_{i+1} x_i x_{i-1} \ldots x_1 x_0) = x_{n-1} \ldots x_{i+1} x_0 x_i x_{i-1} \ldots x_1$$

La i-ésima conexión de línea base realiza un desplazamiento cíclico del los $i+1$ dígitos menos significativos del índice una posición a la derecha. La figura 6.8 muestra una conexión en línea base para $k = 2$, $i = 0, 1, 2$ y $N = 8$. Observar que δ_0^k define la conexión identidad I.

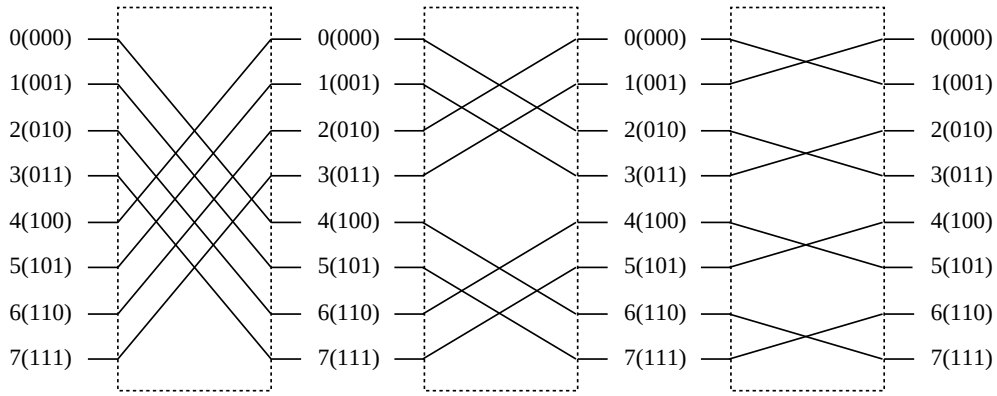


Figura 6.7: La conexión cubo para $N = 8$.

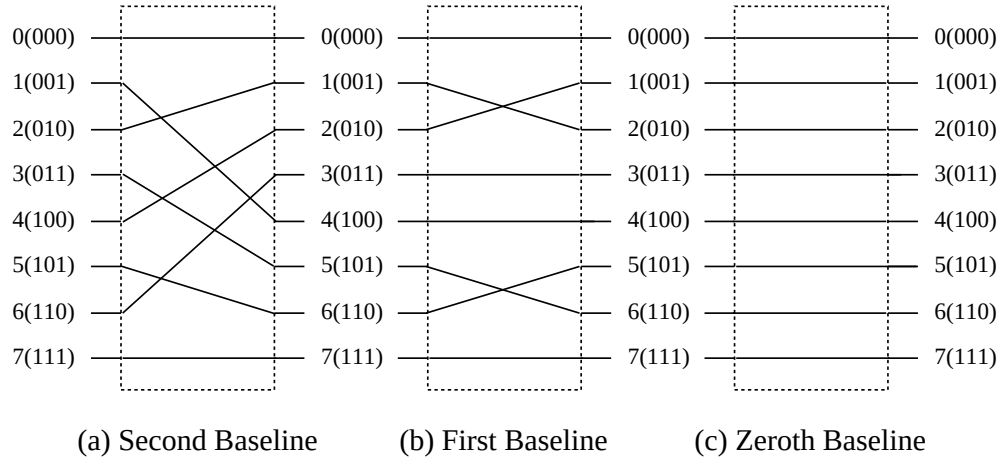


Figura 6.8: La conexión en línea base para $N = 8$.

6.1.6 Clasificación de las redes MIN

Dependiendo de la disponibilidad de caminos para establecer nuevas conexiones, las redes multietapa se han dividido tradicionalmente en tres clases:

1. *Bloqueantes.* La conexión entre un par entrada/salida libre no siempre es posible debido a conflictos entre las conexiones existentes. Típicamente, existe un único camino entre cada par entrada/salida, lo que minimiza el numero de conmutadores y los estados. Sin embargo, es posible proporcionar múltiples caminos para reducir los conflictos e incrementar la tolerancia a fallos. A estas redes bloqueantes también se les denominan *multicamino.*
2. *No bloqueantes.* Cualquier puerto de entrada puede conectarse a cualquier puerto de salida libre sin afectar a las conexiones ya existentes. Las redes no bloqueantes tienen la misma funcionalidad que un crossbar. Estar redes requieren que existan múltiples caminos entre cada entrada y salida, lo que lleva a etapas adicionales.
3. *Reconfigurables.* Cada puerto de entrada puede ser conectado a cada puerto de salida. Sin embargo, las conexiones existentes pueden requerir de un reajuste en sus caminos. Estas redes también necesitan de la existencia de múltiples caminos entre cada entrada y salida pero el numero de caminos y el coste es menor que en el caso de redes no bloqueantes.

La redes no bloqueantes son caras. Aunque son más baratas que un crossbar del mismo tamaño, su costo es prohibitivo para tamaños grandes. El ejemplo más conocido de red no bloqueante multietapa es la red Clos, inicialmente propuesta para redes telefónicas. Las redes reconfigurables requieren menos estados o conmutadores más sencillos que una red no bloqueante. El mejor ejemplo de red reconfigurable es la red Beneš. La figura 6.9 muestra de una Beneš de 8×8. Para 2^n entradas, esta red necesita $2n - 1$ etapas, y proporciona 2^{n-1} caminos alternativos. Las redes reconfigurables requieren un controlador central para reconfigurar las conexiones, y fueron propuestas para los procesadores matriciales (*array processors*). Sin embargo, las conexiones no se pueden reconfigurar fácilmente en multiprocesadores ya que el acceso de los procesadores a la red es asíncrono. Así, las redes reconfigurables se comportan como redes bloqueantes cuando los accesos son asíncronos. Nos centraremos en la redes bloqueantes.

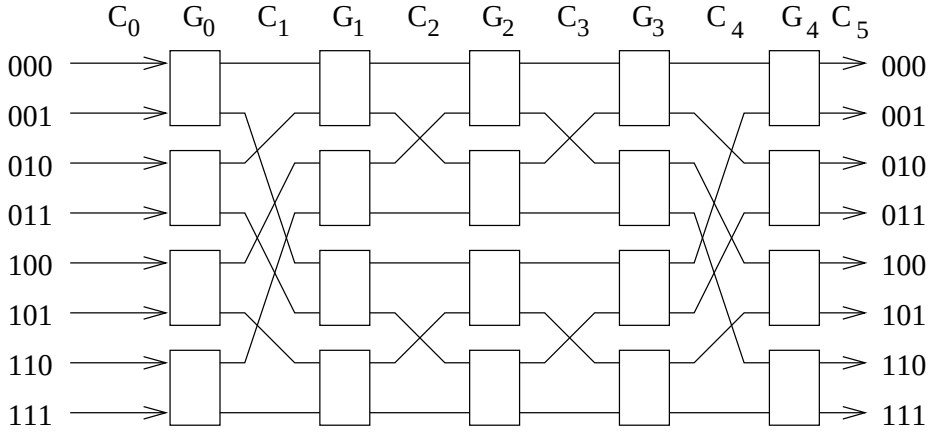


Figura 6.9: Una red Beneš 8×8.

Dependiendo del tipo de canales y conmutadores, las redes multietapa pueden dividirse en dos clases:

1. *MINs unidireccionales.* Los canales y conmutadores son unidireccionales.
2. *MINs bidireccionales.* Los canales y conmutadores son bidireccionales. Esto implica que la información se puede transmitir simultáneamente en direcciones opuestas entre conmutadores vecinos.

Redes de Interconexión Multietapa Unidireccionales

Los bloques básicos de construcción de una red multietapa unidireccional son los conmutadores unidireccionales. Un conmutador $a \times b$ es un *crossbar* con a entradas y b salidas. Si cada puerto de entrada puede conectarse a exactamente un puerto de salida, a lo más existirán $\min\{a, b\}$ conexiones simultáneas. Si cada puerto de entrada puede conectarse a varios puertos de salida, se necesita un diseño más complicado para soportar la comunicación *multicast (*uno a varios). En el modo de comunicación *broadcast* (uno a todos), cada puerto puede conectarse a todos los puertos de salida. La figura 6.10 muestra los cuatro posibles estados de un conmutador 2×2. Los últimos dos estados se usan para soportar las comunicaciones uno a muchos y uno a todos.

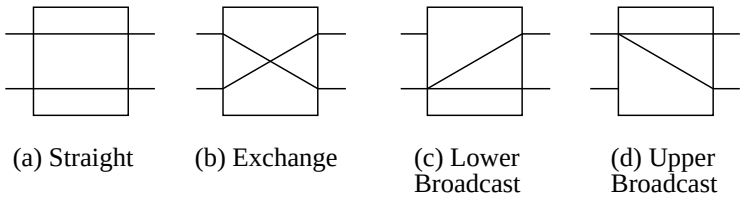


Figura 6.10: Cuatro posibles estados de un conmutador 2×2.

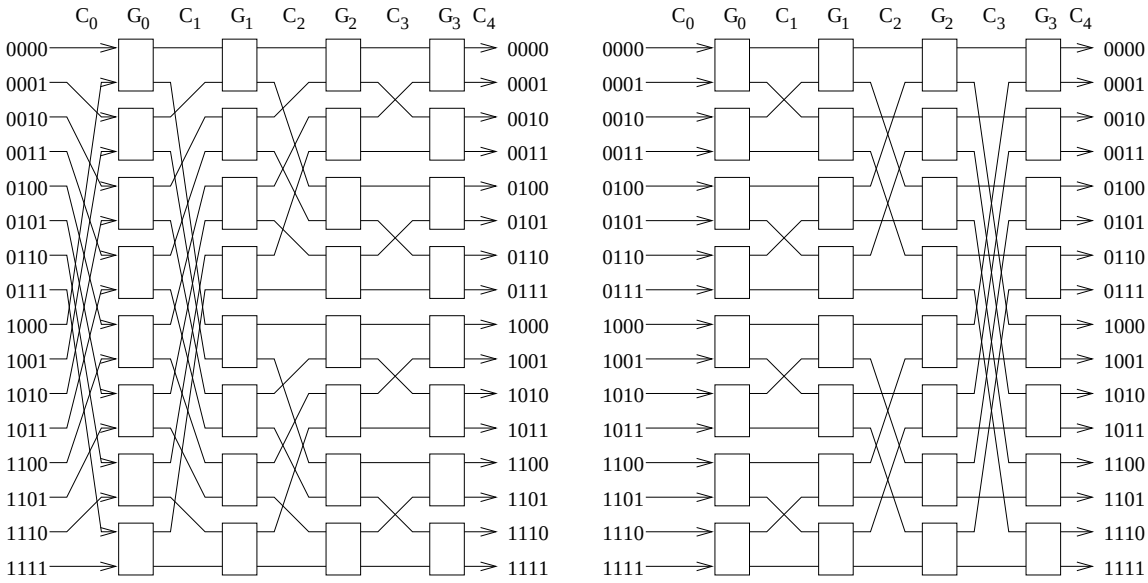
En las redes multietapa con $N = M$, es habitual usar conmutadores con el mismo número de puertos de entrada y de salida ($a = b$). Si $N > M$, se usarán conmutadores con $a > b$. A estos conmutadores también se les denominan *conmutadores de concentración.* En el caso de $N < M$ se usarán *conmutadores de distribución* con $a < b$.

Se puede demostrar que con N puertos de entrada y salida, una red multietapa unidireccional con conmutadores $k \times k$ necesita al menos $\lceil \log_k N \rceil$ etapas para que exista un camino entre cualquier puerto de entrada y de salida. Añadiendo etapas adicionales es posible utilizar caminos alternativos para enviar un mensaje desde un puerto de entrada a un puerto de salida. Cualquier camino a través de la MIN cruza todos las etapas. Por tanto, todos los caminos tienen la misma longitud.

A continuación se consideran cuatro topologías equivalentes en redes multietapa unidireccionales. Estas redes pertenecen a la familia de redes Delta.

- **Redes multietapa de línea base** (*Baseline MINs*). En una red multietapa de línea base, el patrón de conexión C_i viene descrito por la $(n-i)$-ésima permutación en línea base δ^k_{n-i} para $1 \leq i \leq n$. El patrón de conexión para C_o viene dado por σ^k.

  La *red de línea base* es un ejemplo de red con una forma muy conveniente de algoritmo de autoencaminamiento, en el que los bits sucesivos de la dirección de destino controlan las sucesivas etapas de la red. Cada etapa de la red divide el rango de encaminamiento en dos. La figura 6.12 muestra este tipo de red suponiendo 4 entradas y 4 salidas. El funcionamiento consiste en que la primera etapa divide el camino en dos vías, uno hacia la parte baja de la red, y el otro hacia la parte alta. Por tanto, el bit más significativo de la dirección de destino puede usarse para dirigir la



(a) Multistage Baseline Network

(b) Multistage Butterfly Network

(c) Multistage Cube Network

(d) Omega Network

Figura 6.11: Cuatro redes de interconexión unidireccionales multietapa de 16 × 16.

entrada hacia la mitad alta de la segunda etapa si el bit es 0, o hacia la mitad baja si el bit es 1. La segunda etapa divide la ruta entre el cuarto más alto o el segundo cuarto si la mitad más alta se había seleccionado, o el cuarto más bajo o el primero si la mitad seleccionada era la baja. El segundo bit más significativo es usado para decidir qué cuarto elegir. Este proceso se repite para todas las etapas que haya. Es evidente que el número de bits necesarios para la dirección de las entradas dará el número de etapas necesario para conectar completamente la entrada con la salida. Por último, el bit menos significativo controla la última etapa. Este tipo de redes autorutadas sugiere la transmisión por conmutación de paquetes.

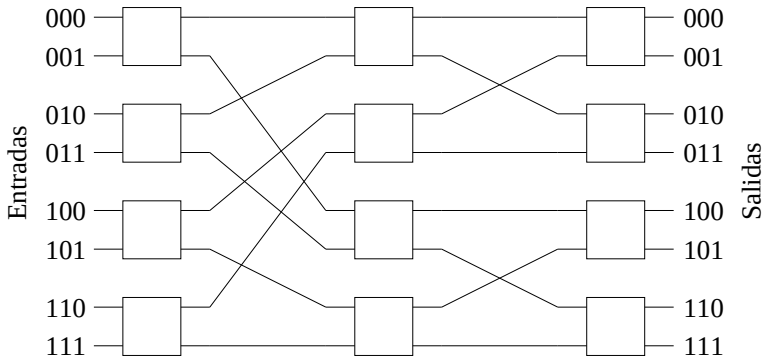
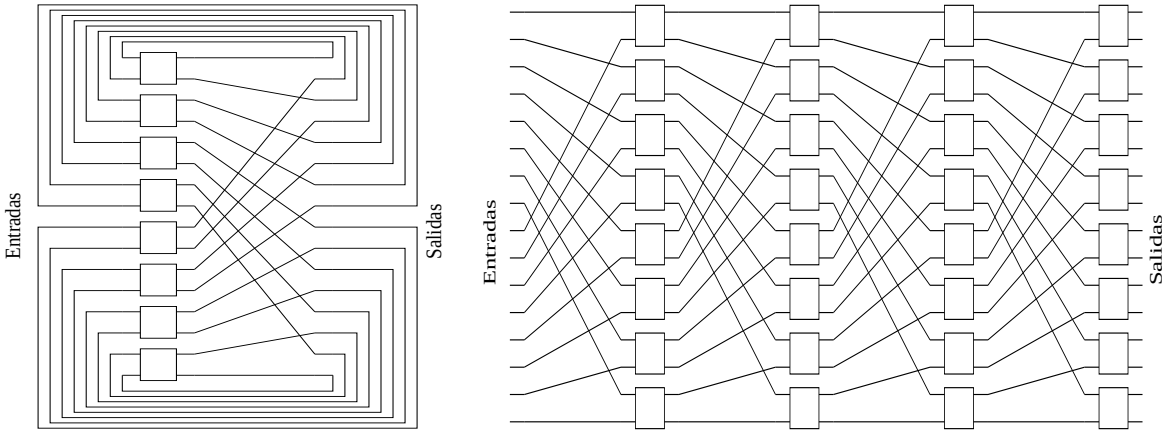


Figura 6.12: Red de línea base 8×8.

Es fácil ver que una red de este tipo se genera de forma recursiva, ya que se van dividiendo las posibles rutas en mitades donde cada mitad es a su vez una red de línea base y así sucesivamente hasta la última etapa. Los conmutadores son 2×2 con dos posibles estados, directo y cruzado, entre las entradas y salidas.

- **Redes multietapa mariposa** (*Butterfly MINs*). En una red multietapa mariposa, el patrón de conexión $C_i$viene descrito por la i-ésima permutación mariposa β_i^k para $0 \leq i \leq n-1$. El patrón de conexión de C_n es β_0^k.
- **Redes multietapa cubo** (*Cube MINs*). En una red multietapa cubo, el patrón de conexión C_i viene descrito por la $(n-i)$-ésima permutación mariposa β_{n-i}^k para $1 \leq i \leq n$. El patrón de conexión de C_o es σ^k.
- **Red Omega**. En una red omega, el patrón de conexión C_i viene descrito por la permutación k-baraje perfecto σ^k para $0 \leq i \leq n-1$. El patrón de conexión de C_n es es β_0^k. Así, todos los patrones de conexión menos el último son idénticos. El último patrón de conexión no produce una permutación.
  La figura 6.13(b) muestra la red Omega para 16 elementos de entrada y 16 de salida. Tal y como se comentó anteriormente, son necesarios $\log_2 a$ etapas siendo a el número de entradas a la red. Como cada etapa necesita $n/2$ conmutadores, la red total requiere $n(\log_2 n)/2$ conmutadores, cada uno controlado de forma individual. La red Omega se propuso para el procesamiento de matrices utilizando conmutadores de cuatro estados (directo, intercambio, el de arriba a todos, y el de abajo a todos) y actualmente se utiliza en sistemas multiprocesadores.
- **Red de barajado/intercambio**. Es una red un tanto peculiar basada en la red omega. Para hacer todas las posibles interconexiones con el patrón de barajado, se puede utilizar una red de recirculación que devuelve las salidas de nuevo a las entradas hasta que se consigue la interconexión deseada. Para poder hacer esto se añaden *cajas de intercambio* que no son más que conmutadores 2×2. La figura 6.13(a) muestra una red de este tipo para 16 elementos de entrada y 16 de salida. Si hay 2^n

entradas es necesario un máximo de n ciclos para realizar cualquier permutación entre los elementos de entrada y salida.



(a) Red de barajado-intercambio (b) Red Omega

Figura 6.13: Redes basadas en el barajado perfecto.

La equivalencia topológica entre estas redes multietapa puede ser vista como sigue: Consideremos que cada enlace de entrada del primer estado se numera usando una cadena de n dígitos $s_{n-1} \ldots s_0$, donde $0 \leq s_i \leq k-1$ para $0 \leq i \leq n-1$. El dígito menos significativo s_0 da la dirección del puerto de entrada en el correspondiente conmutador y la dirección del conmutador viene dada por $s_{n-1}s_{n-2} \ldots s_1$. En cada etapa, un conmutador dado es capaz de conectar cualquier puerto de entrada con cualquier puerto de salida. Esto puede interpretarse como el cambio del valor del dígito menos significativo de la dirección. Para conseguir conectar cualquier entrada con cualquier salida de la red, deberíamos de ser capaces de cambiar el valor de todos los dígitos. Como cada conmutador sólo es capaz de cambiar el dígito menos significativo de la dirección, los patrones de conexión entre las etapas se definen de tal forma que se permute la posición de los dígitos, y después de n etapas todos los dígitos habrán ocupado la posición menos significativa. Por lo tanto, la MINs definidas arriba difieren en el orden en el que los dígitos de la dirección ocupan la posición menos significativa.

La figura 6.11 muestra la topología de cuatro redes de interconexión multietapa: (a) red *baseline*, (b) red *butterfly*, (c) red *cube*, y (d) red *omega*.

Redes de interconexión multietapa bidireccionales

La figura 6.14 ilustra un conmutador bidireccional en donde cada puerto está asociado a un par de canales unidireccionales en direcciones opuestas. Esto implica que la información entre conmutadores vecinos puede transmitirse simultáneamente en direcciones opuestas. Para facilitar la explicación, supondremos que los nodos procesadores están en la parte izquierda de la red, como se muestra en la figura 6.15. Un conmutador bidireccional soporta tres tipos de conexiones: *hacia delante*, *hacia atrás* y *de vuelta* (ver figura 6.14). Dado que son posibles la conexiones de vuelta entre puertos del mismo lado del conmutador, los caminos tienen diferentes longitudes. En la figura 6.15 se

muestra una MIN mariposa bidireccional de ocho nodos.

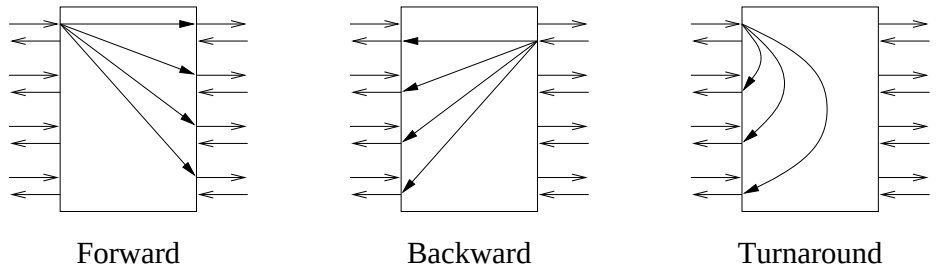


Figura 6.14: Conexiones en un conmutador bidireccional.

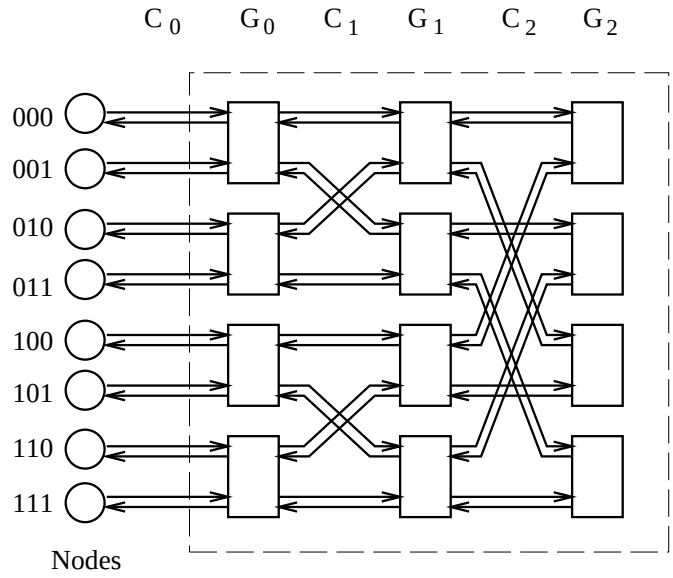


Figura 6.15: Una MIN mariposa bidireccional de ocho nodos.

En las BMINs, los caminos se establecen cruzando etapas hacia delante, después estableciendo una conexión de vuelta, y finalmente cruzando los estados en dirección hacia atrás. A este método se le denomina *encaminamiento de vuelta* (*turnaround routing*). La figura 6.16 muestra dos caminos alternativos desde el nodo S al nodo D en una BMIN mariposa de ocho nodos. Cuando se cruzan las etapas hacia delante, existen varios caminos. En cada conmutador se puede seleccionar cualquiera de sus puertos de salida. Sin embargo, una vez cruzada la conexión de vuelta, existe un único camino hasta el nodo destino. En el peor caso, el establecimiento de un camino en una BMIN de n etapas requiere cruzar $2n-1$ etapas. Este comportamiento es muy parecido al de las redes Beneš. En efecto, la BMIN línea base puede considerarse como una red Beneš plegada.

En la figura 6.17, se muestra un BMIN mariposa con encaminamiento de vuelta. En un *fat tree*, los procesadores se localizan en las hojas, y los vértices internos son conmutadores. El ancho de banda de la transmisión entre los conmutadores se incrementa añadiendo más enlaces en paralelo en los conmutadores cercanos al conmutador raíz. Cuando se encamina un mensaje desde un procesador a otro, se envía hacia arriba en el árbol hasta alcanzar el menor antecesor común de ambos procesadores, y después se envía hacia abajo hacia su destino.

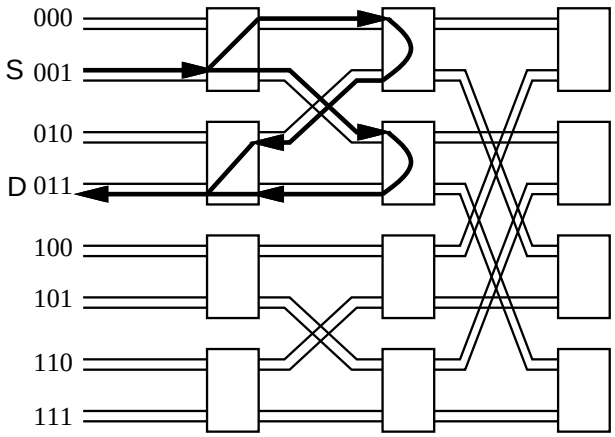


Figura 6.16: Caminos alternativos para una MIN mariposa bidireccional de ocho nodos.

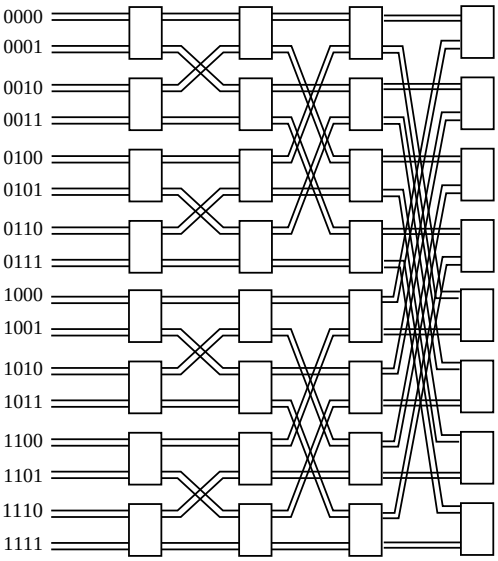
6.1.7 Encaminamiento en redes MIN

En esta sección describiremos diversos aspectos del encaminamiento en redes MINs. Para procesadores matriciales, un controlador central establece el camino entre la entrada y la salida. En casos donde el número de entradas es igual al número de salidas, cada entrada transmite de forma síncrona un mensaje a una salida, y cada salida recibe exactamente un mensaje de una entrada. Calcular las conexiones del conmutador para realizar dicha permutación es una tarea compleja. Además, algunas permutaciones pueden no ser posibles en un paso a través de la red. En este caso, serán necesarios múltiples pasos de los datos a través de la red en cuyo caso el objetivo pasa a ser el minimizar el número de pasadas. La complejidad del cálculo de las conexiones del conmutador es proporcional al número de conmutadores.

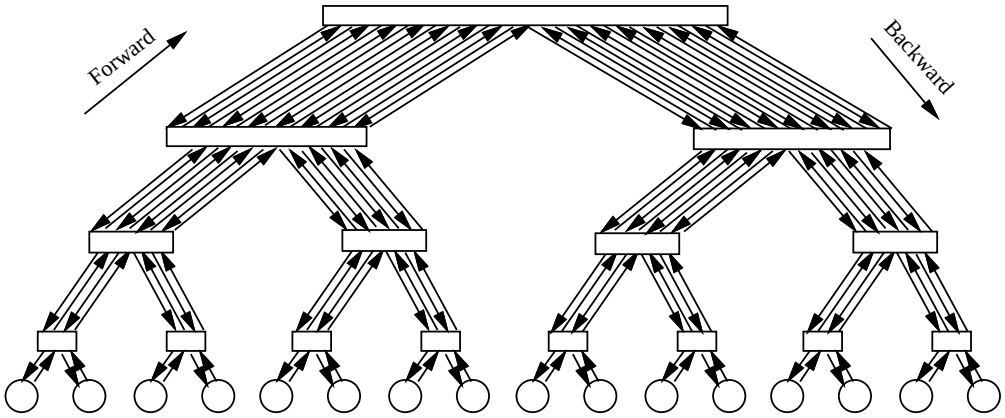
Por otra parte, en multiprocesadores asíncronos, el control centralizado y el encaminamiento por permutación es poco flexible. En este caso es necesario un algoritmo de encaminamiento para establecer el camino por las diferentes etapas de la MIN. La solución más simple consiste en utilizar un encaminamiento en origen. En este caso, el nodo origen especifica el camino completo. Dado que esta solución produce una considerable sobrecarga, nos centraremos en encaminamientos distribuidos. Los algoritmos de encaminamiento para MINs se describirán posteriormente.

Condición de bloqueo en MINs

En este apartado obtendremos condiciones necesarias y suficientes para que se bloqueen dos circuitos, es decir, necesiten el mismo enlace intermedio. En un sistema con N procesadores, existen exactamente N enlaces entre cada etapa de los $k \times k$ conmutadores. La red consta de $n = log_k N$, donde cada etapa está compuesta de $\frac{N}{k}$ conmutadores. Los patrones de enlaces intermedios conectan una salida de un conmutador en la etapa i con un entrada de un conmutador de la etapa $i+1$. El bloqueo ocurre cuando dos paquetes deben atravesar la misma salida en un conmutador de la etapa i, $i = 0, 1, \ldots n-1$.En cada etapa i, podemos numerar las salidas de dicha etapa desde 0 hasta $N-1$. A estas salidas se les denominan salidas intermedias en la etapa i. Si pensamos en la red como una caja negra, podemos representarla como se muestra en la figura 6.18 para el caso de una red Omega con conmutadores 2×2. Cada etapa de conmutadores y cada



(a) A 16−Node Butterfly BMIN Built with 2 x 2 Switches



(b) A 16−Node Fat Tree

Figura 6.17: Árbol grueso y BMIN mariposa.

etapa de enlaces puede representarse como una función que permuta las entradas. En la figura se muestra un ejemplo de un camino desde la entrada 6 a la salida 1. Las salidas intermedias para este camino también se han marcado. Estas son las salidas 4, 0 y 1 para los conmutadores de las etapas 0, 1 y 2, respectivamente. Nuestra meta inicial es la siguiente: Dado un par entrada/salida, generar las direcciones de todas las salidas intermedias de ese camino, estas direcciones serán únicas. Dos caminos de entrada/salida presentan un conflicto si atraviesan el mismo enlace intermedio o comparten una salida común en una etapa intermedia.

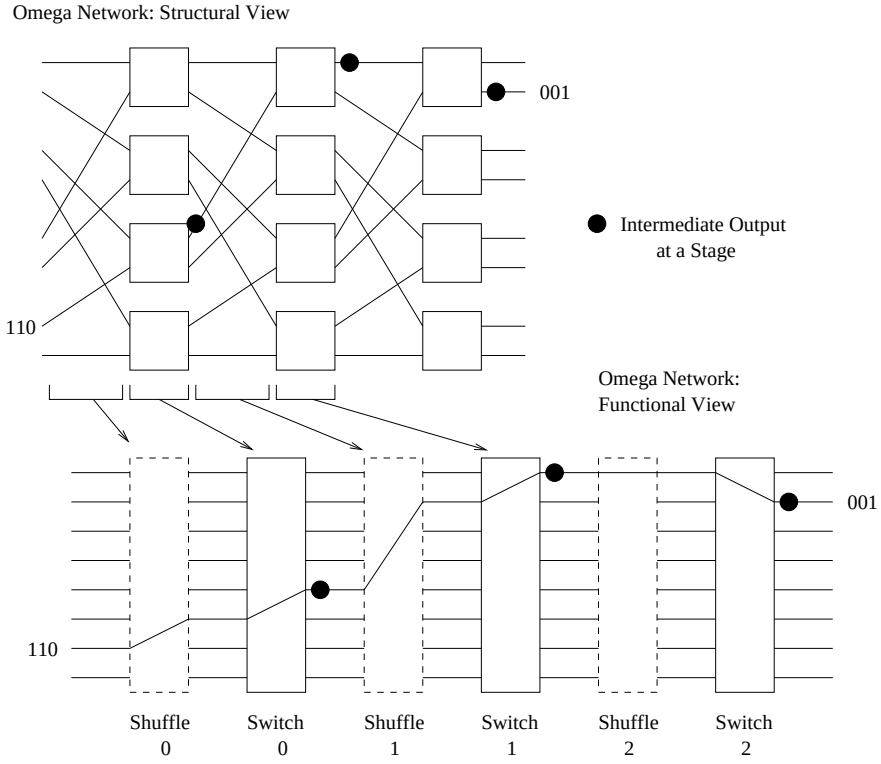


Figura 6.18: Vista funcional de la estructura de una red Omega multietapa.

A continuación obtendremos la condición de bloqueo para la red Omega. Para otras redes es posible establecer condiciones equivalentes de manera similar. A partir de ahora todas las numeraciones de las entradas/salidas se representarán en base k. Para facilitar el análisis supondremos una red con conmutación de circuitos. Consideremos que se ha establecido un circuito desde $s_{n-1}s_{n-2}\cdots s_1s_0$ a $d_{n-1}d_{n-2}\cdots d_1d_0$. Consideremos la primera etapa de enlaces en la figura 6.18. Esta parte de la red establece la siguiente conexión entre sus entradas y salidas.

$$s_{n-1}s_{n-2}\ldots s_1s_0 \longrightarrow s_{n-2}s_{n-3}\ldots s_1s_0s_{n-1} \tag{6.2}$$

La parte derecha de la ecuación anterior es la codificación de la salida en el patrón k-desplazamiento y la codificación de la entrada al primer nivel de conmutadores. Cada conmutador sólo puede cambiar el dígito menos significativo de la codificación actual. Después de la permutación desplazamiento, éste es s_{n-1}. Así que la salida de la primera etapa será aquella que haga que el dígito menos significativo de la codificación sea igual a d_{n-1}. Por lo tanto, la conexión entrada/salida establecida en la primera etapa de conmutadores deberá ser aquella que conecta la entrada a la salida como sigue:

$$s_{n-2}s_{n-3}\ldots s_1s_0s_{n-1} \rightarrow s_{n-2}s_{n-3}\ldots s_1s_0d_{n-1} \tag{6.3}$$

La parte derecha de esta expresión es la codificación de la entrada en la segunda etapa de enlaces. Es también la salida de la primera etapa de conmutadores y por tanto la primera salida intermedia. En la figura 6.18 se muestra un camino desde la entrada 6 a la salida 1 que tiene un total de tres salidas intermedias. De forma similar, podemos escribir la expresión de la salida intermedia en la etapa i como

$$s_{n-i-2}s_{n-i-3}\ldots s_0d_{n-1}d_{n-2}\ldots d_{n-i-1} \tag{6.4}$$

Esto es suponiendo que las etapas están numeradas desde 0 hasta $n-1$. Ahora podemos escribir la condición de bloqueo. Para cualquier entrada/salida (S, D) y (R, T), los dos caminos pueden establecerse sin conflictos si y sólo si, $\forall i,\ 0 \leq i \leq n-1$

$$s_{n-i-2}s_{n-i-3}\ldots s_0d_{n-1}d_{n-2}\ldots d_{n-i-1} \neq r_{n-i-2}r_{n-i-3}\ldots r_0t_{n-1}t_{n-2}\ldots t_{n-i-1} \tag{6.5}$$

La detección de bloqueos en dos pares de entrada/salida no es tan complicada como podría parecer a primera vista. Observando la estructura de la condición de bloqueo podemos observar que si los circuitos se bloquean, entonces los dígitos $n-i-1$ menos significativos de las direcciones origen son iguales y los $i+1$ dígitos más significativos de las direcciones destino son iguales. Supongamos que tenemos una función $\phi(S, R)$ que devuelve el mayor entero l tal que los l dígitos menos significativos de S y R son iguales. De forma similar, supongamos que tenemos una función $\psi(D, T)$ que devuelve el mayor entero m tal que los m dígitos más significativos de D y T son iguales. Entonces dos caminos (S, D) y (R, T) pueden establecerse sin conflictos si y sólo si

$$\phi(S, R) + \psi(D, T) < n \tag{6.6}$$

donde $N = k^n$. Desde un punto de vista práctico, el bloqueo se puede calcular mediante secuencias de operaciones de desplazamiento y comparación, como se indica a continuación:

$$\begin{array}{l} s_{n-1}s_{n-2}\ldots \boxed{s_2s_1s_0d_{n-1}d_{n-2}\ldots d_3}\ d_2d_1d_0 \\ r_{n-1}r_{n-2}\ldots \boxed{r_2r_1r_0\ t_{n-1}t_{n-2}\ldots t_3}\ t_2t_1t_0 \end{array}$$

Las direcciones de los dos pares entrada/salida se concatenan. Figurativamente, una ventana de tamaño n se desplaza sobre ambos pares y se compara el contenido de ambas ventanas. Si son iguales en algún punto, entonces existe un conflicto en alguna etapa. El orden de ejecución es de $O(log_kN)$. Para determinar si todos los caminos pueden establecerse sin conflictos, deberíamos realizar $O(N^2)$ comparaciones cada una de las cuales tarda $O(log_kN)$ pasos en realizarse dando lugar a un algoritmo de orden $O(N^2log_kN)$. En comparación, el mejor algoritmo de configuración de todos los conmutadores mediante un control centralizado necesita un tiempo de $O(Nlog_kN)$. Sin embargo, cuando se utiliza la formulación arriba indicada, muy a menudo no es necesario tener en cuenta el número de comparaciones en el peor de los casos. A menudo, la propia estructura del patrón de comunicaciones puede utilizarse para determinar si existe conflicto entre ellos.

Algoritmos de autoencaminamiento para MINs.

Una propiedad única de las redes Delta es la propiedad de autoencaminamiento (*selft-routing*). Esta propiedad permite que en estas MINs las decisiones de encaminamiento se puedan realizar en función de la dirección destino, sin importar la dirección origen. El autoencaminamiento se realiza usando etiquetas de encaminamiento. Para un conmutador $k \times k$, existen k puertos de salida. Si el valor de la etiqueta de encaminamiento correspondiente es i $(0 \leq i \leq k-1)$, el paquete correspondiente será enviado a través del puerto i. Para una MIN de n estados, la etiqueta de encaminamiento es $T = t_{n-1} \ldots t_1 t_0$, donde t_i controla el conmutador de la etapa G_i.

Cada conmutador sólo es capaz de cambiar el dígito menos significativo de la dirección actual, Por lo tanto, las etiquetas de encaminamiento tendrán en cuenta qué dígito es el menos significativo en cada etapa, reemplazándolo por el correspondiente dígito de la dirección destino. Para un destino dado $d_{n-1}d_{n-2} \ldots d_0$, la etiqueta de encaminamiento en una red mariposa se forma haciendo $t_i = d_{i+1}$ para $0 \leq i \leq n-2$ y $t_{n-1} = d_0$. En una MIN cubo, la etiqueta está formada por $t_i = d_{n-i-1}$ para $0 \leq i \leq n-1$. Finalmente, en una red Omega, la etiqueta de encaminamiento se forma haciendo $t_i = d_{n-i-1}$ para $0 \leq i \leq n-1$. El siguiente ejemplo muestra los caminos seleccionados por el algoritmo de encaminamiento basado en etiquetas para una MIN mariposa de 16 nodos.

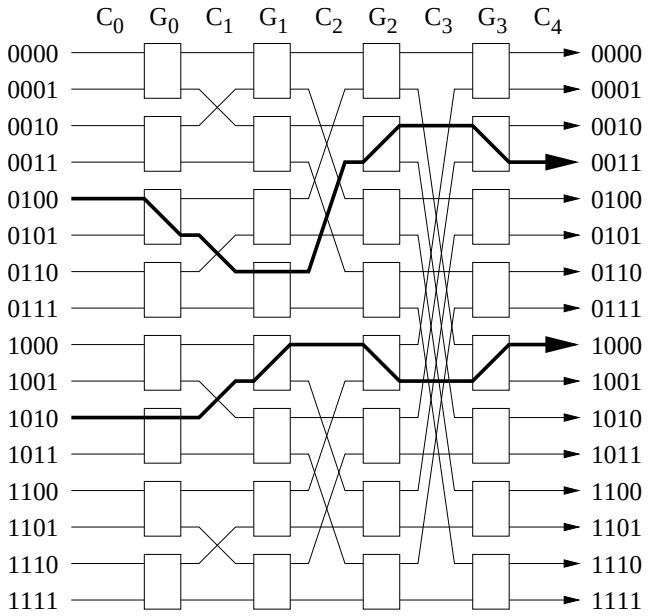


Figura 6.19: Selección del camino por el algoritmo de encaminamiento basado en etiquetas en una MIN mariposa de 16 nodos.

Ejemplo

La figura 6.19 muestra una red MIN de 16 nodos usando conmutadores 2×2, y los caminos seguidos por los paquetes del nodo 0100 al nodo 0011 y del nodo 1010 al nodo 1000. Como se indicó anteriormente, la etiqueta de encaminamiento para un destino dado $d_{n-1}d_{n-2} \ldots d_0$ se forma haciendo $t_i = d_{i+1}$ para $0 \leq i \leq n-2$ y $t_{n-1} = d_0$. Así, la etiqueta para el destino 0011 es 1001. Esta etiqueta indica que el paquete debe elegir la salida superior del

conmutador (puerto 0) en los estados G_2 y G_1, y la salida inferior (puerto 1) en los estados G_3 y G_0. La etiqueta para el destino 1000 es 0100. Esta etiqueta indica que el paquete debe coger la salida superior de conmutador en los estados G_3, G_1y G_0, y la salida inferior del conmutador en el estado G_2.

Una de las características interesantes de las MINs tradicionales vistas anteriormente (TMINs) es que existe un algoritmo sencillo para encontrar un camino de longitud $log_k N$ entre cualquier par entrada/salida. Sin embargo, si un enlace se congestiona o falla, la propiedad de camino único puede fácilmente interrumpir la comunicación entre algunos pares entrada/salida. La congestión de paquetes sobre algunos canales causa el problema conocido como punto caliente. Se han propuesto muchas soluciones para resolver este problema. Una aproximación popular es proporcionar múltiples caminos alternativos entre cualquier par origen y destino con lo que se consigue reducir la congestión de la red al mismo tiempo que se consigue tolerancia a fallos. Estos métodos requieren normalmente hardware adicional, como el uso de etapas extras o canales adicionales.

El uso de canales adiciones da lugar a las MINs dilatadas. En una MIN d-dilatada (DMIN), cada conmutador se reemplaza por un conmutador d-dilatado. En este conmutador, cada puerto tiene d canales. Usando canales replicados, las DMINs ofrecen una mejora sustancial del *throughput* de la red. La etiqueta de encaminamiento en una DMIN puede determinarse mediante la dirección destino al igual que se mencionó para las TMINs. Dentro de cada conmutador, los paquetes se envían a un puerto de salida particular de forma aleatoria entre los canales libres. Si todos los canales están ocupados, el paquete se bloquea.

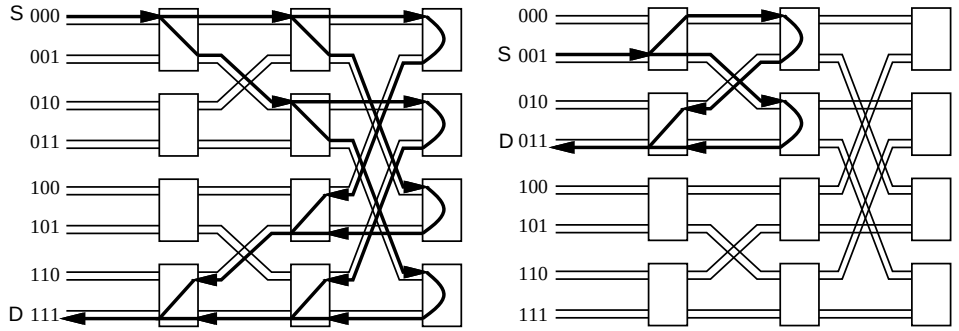
Otra aproximación al diseño de MINs consiste en permitir comunicaciones bidireccionales. En este caso, cada puerto del conmutador tiene canales duales. En una red mariposa (BMIN) construida con conmutadores $k \times k$, la dirección origen S y la dirección destino D se representan mediante números k-arios $s_{n-1} \ldots s_1 s_0$ y $d_{n-1} \ldots d_1 d_0$ respectivamente. La función $FirstDifference\,(S, D)$ devuelve t, la posición en donde existe la primera diferencia (más a la izquierda) entre $s_{n-1} \ldots s_1 s_0$ y $d_{n-1} \ldots d_1 d_0$.

Un camino de encaminamiento con vuelta atrás entre cualquier origen y destino debe cumplir las siguientes restricciones:

- El camino consiste en algunos canales hacia delante, algunos canales hacia atrás, y exactamente una conexión de vuelta.
- El número de canales hacia adelante es igual al número de canales hacia atrás.
- Ningún canal hacia adelante y hacia atrás a lo largo del camino son el canal pareja del mismo puerto.

Obsérvese que la última condición se utiliza para prevenir comunicaciones redundantes. Para encaminar un paquete desde el origen al destino, el paquete se envía en primer lugar a la etapa G_t. No importa a que conmutador llegue (en la etapa G_t). Después, el paquete gira y se envía de vuelta a su destino. Cuando se mueve hacia el estado G_t, un paquete tiene varias opciones sobre qué canal de salida elegir. La decisión puede resolverse seleccionando aleatoriamente entre aquellos canales de salida que no están bloqueados por otros paquetes. Después de que el paquete ha llegado a un conmutador en el estado G_t, elige el único camino desde ese conmutador hacia su destino. Este camino de vuelta puede determinarse mediante un algoritmo de encaminamiento basado en etiquetas para TMINs.

Observar que los algoritmos de encaminamiento descritos arriba son algoritmos distribuidos, en donde cada conmutador determina el canal de salida basándose en la información que lleva el propio paquete. El siguiente ejemplo muestra los caminos disponibles en una red BMIN mariposa de ocho nodos.



(a) FirstDifference(S, D) = 2 (b) FirstDifference(S, D) = 1

Figura 6.20: Caminos disponibles en una BMIN mariposa de ocho nodos.

Ejemplo

La figura 6.20 muestra los caminos disponibles para una red BMIN mariposa de ocho nodos usando conmutadores 2×2. La figura 6.20a muestra el caso en donde los nodos origen y destino son 000 y 111, respectivamente. En este caso $FirstDifference(S, D) = 2$. Así, los paquetes giran en la etapa G_2 y existen cuatro caminos diferentes disponibles. En la figura 6.20b, los nodos origen y destino son 001y 011, respectivamente. $FirstDifference(S, D) = 1$, y el paquete gira en la etapa G_1.

6.1.8 Resumen de las redes indirectas y equivalencias

En la tabla 6.1 se resumen las características más importantes de los buses, redes multietapa, y conmutadores de barra cruzada, para la construcción de redes dinámicas. Evidentemente el bus es la opción más económica pero tiene el problema del bajo ancho de banda disponible para cada procesador, especialmente si hay muchos. Otro problema con el bus es que es sensible a los fallos, de manera que es necesario su duplicación para obtener sistemas tolerantes a fallos.

El conmutador de barra cruzada es el más caro de hacer, especialmente teniendo en cuenta que su complejidad hardware crece con n^2. Sin embargo, la barra cruzada tiene el mayor ancho de banda y las mayor capacidad de rutado. Para una red con no demasiados elementos es la mejor opción.

Las redes multietapa suponen un compromiso intermedio entre los dos extremos. La mayor ventaja de las MINs es su escalabilidad gracias a la construcción modular. El problema de esta red es que la latencia crece con $\log n$, es decir, con el número de etapas en la red. También, el coste asociado con los cables y los conmutadores puede ser importante.

Para la construcción de futuros MPP, algunas de las topologías estáticas pueden resultar más interesantes por su alta escalabilidad en determinadas aplicaciones. Si las

Características de la red	Bus	Multietapa	Barra cruzada
Latencia mínima por unidad de transmisión	Constante	$O(\log_k n)$	Constante
Ancho de banda por procesador	de $O(w/n)$ a $O(w)$	de $O(w)$ a $O(nw)$	de $O(w)$ a $O(nw)$
Complejidad cableado	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Complejidad conmutadores	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Capacidad de rutado y conectado	Sólo uno a uno cada vez.	Algunas permutaciones y broadcast si la red no está bloqueada.	Todas las permutaciones a la vez.
Computadores representativos	Symmetry S-1 Encore Multimax	BBN TC-2000 IBM RP3	Cray Y-MP/816 Fujitsu VPP500
Notas	Suponemos n procesadores; la anchura de banda es w	MIN $n \times n$ con conmutadores $k \times k$ y líneas de w bits	Barra cruzada $n \times n$ con líneas de w bits

Tabla 6.1: Resumen de las características de las redes dinámicas.

tecnologías ópticas y microelectrónicas avanzan rápidamente, los MINs a escala alta o las redes de barras cruzadas podrían llegar a ser realizables o económicamente rentables para la realización de conexiones dinámicas en computadores de propósito general.

6.2 Modelos de consistencia de memoria

Este apartado se encuentra principalmente en [Tan95] y [CSG99], pero se puede ampliar el tema en [CDK96], [Hwa93] y [HP96].

La coherencia es esencial si se quiere transmitir información entre procesadores mediante la escritura de uno en una localización que leerá el segundo. Sin embargo, no se dice nada acerca de cuando el valor escrito debe hacerse visible. A menudo, en la escritura de un programa paralelo queremos asegurarnos que una lectura devuelve el valor de una determinada escritura. Es decir, queremos establecer un orden entre una escritura y una lectura. Normalmente, usamos alguna forma de evento de sincronización para asegurarnos esta dependencia.

Consideremos, por ejemplo, los fragmentos de código que se muestran en la figura 6.21 y que son ejecutados por los procesadores P_1 y P_2. De este código se deduce que el programador quiere que el proceso que se ejecuta en P_2 realice una espera activa hasta que el valor de la variable compartida *flag* cambie a 1, y después imprimir el valor de la variable A que debería ser 1, dado que dicho valor ha sido actualizado antes del valor de *flag* por el procesador P_1. En este caso, estamos accediendo a otra localización

(*flag*) para preservar el orden entre los diferentes accesos a la misma localización (A). En particular, estamos suponiendo que la escritura de A es visible para el procesador P_2 antes que la escritura a *flag*, y que la lectura de flag por P_2 que hace que salgamos del bucle *while* se ha completado antes que la lectura de A. Esta ordenación entre los accesos de P_1 y P_2 no está garantizada por la coherencia, que, por ejemplo, únicamente requiere que el nuevo valor de A se haga visible al procesador P_2, no necesariamente antes de que el nuevo valor de *flag* sea observado.

```
        /* El valor inicial de A y flag es 0 */
  P1            P2
A=1         while (flag == 0); /* bucle vacío */
flag=1      print A
```

Figura 6.21: Requisitos de un evento de sincronización mediante flags.

Claramente, estamos esperando más del sistema de memoria que devolver el "último valor escrito" para cada localización: Establecer un orden entre los accesos realizados por diferentes procesos a la localización (digamos A), en algunos casos esperamos que un sistema de memoria respete el orden de las lecturas y escrituras a *diferentes* localizaciones (A y *flag*) realizadas por un proceso dado. La coherencia no dice nada acerca del orden en el cual las escrituras realizadas por P_1 deben hacerse visibles a P_2, dado que estas operaciones se realizan sobre diferentes localizaciones. De manera similar, no dice nada acerca del orden en el cual las lecturas realizadas a diferentes localizaciones por P_2 se realizan de forma relativa a P_1. Por lo tanto, la coherencia por si sola no evita un resultado de 0 en el ejemplo. Claramente, necesitamos algo más que la coherencia para dar a un espacio de direcciones compartido una semántica clara, es decir, un modelo de ordenación que los programadores puedan usar para razonar acerca de los posibles resultados y por tanto de la corrección de sus programas.

Un *modelo de consistencia de la memoria* para un espacio de direcciones compartido especifica las restricciones en el orden en el cual las operaciones de memoria deben parecer haberse realizado (es decir, hacerse visibles a los procesadores) entre ellas. Esto incluye operaciones a las mismas localizaciones o diferentes localizaciones, y por el mismo proceso o por diferentes procesos. Por lo tanto, la consistencia de la memoria incluye la coherencia.

6.2.1 Consistencia secuencial

Antes de ver el modelo de consistencia secuencial, que es sin duda uno de los más implementados, se introduce el problema de la consistencia de la memoria con un modelo ideal que no es implementable en la práctica, pero que es lo que probablemente uno esperaría del comportamiento de un sistema multiprocesador cualquiera.

Modelo de consistencia estricta

Este modelo es el caso ideal en el cual cualquier escritura se ve instantáneamente por cualquier lectura posterior. Dicho de otra manera más formal: "Cualquier lectura de

cierta posición de memoria X devuelve el valor almacenado por la escritura más reciente realizada sobre X".

Hay que hacer notar que **este modelo de consistencia no se puede implementar en ningún sistema real**. Efectivamente, como la distancia entre los procesadores y la memoria no es nula, la señal que viaja entre el procesador que escribe y la memoria sufre un cierto retraso. Si durante este retraso, en el cual la memoria no tiene aún el nuevo dato escrito, otro procesador decide leer, no va a leer el valor que se acaba de mandar a la memoria, que aún no ha llegado, sino el que hubiera anteriormente.

Por lo tanto, la única forma en la que se podría implementar este modelo es suponiendo que los datos viajan de forma instantánea (velocidad infinita), lo cual va en contra de la teoría general de relatividad.

El modelo de consistencia secuencial

Cuando se discutieron las características de diseño fundamentales de una arquitectura de comunicación, se describió de forma informal un modelo de ordenación deseable para un espacio de direcciones compartido: el que nos asegura que el resultado de un programa multihilo (*multithread*) ejecutándose con cualquier solapamiento de los distintos hilos sobre un sistema monoprocesador sigue siendo el mismo en el caso de que algunos de los hilos se ejecuten en paralelo sobre diferentes procesadores. La ordenación de los accesos a los datos dentro de un proceso es la del programa, y entre los procesos viene dada por alguno de los posibles solapamientos entre los ordenes de los diferentes programas. Así, en el caso del multiprocesador, éste no debe ser capaz de hacer que los resultados sean visibles en el espacio de direcciones compartido en un forma diferente a la que se pueda generar por los accesos solapados de diferentes procesos. Este modelo intuitivo fue formalizado por Lamport en 1979 como *consistencia secuencial*, que se define como sigue:

> *Consistencia secuencial.* Un multiprocesador es secuencialmente consistente si el resultado de cualquier ejecución es la misma que si las operaciones de todos los procesadores se realizaran en algún orden secuencial, y las operaciones de cada procesador individual ocurren en esta secuencia en el orden especificado por el programa.

La figura 6.22 representa el modelo de abstracción proporcionado a los programadores por un sistema de consistencia secuencial. Varios procesadores *parecen* compartir una *única* memoria lógica, incluso aunque en la máquina real la memoria principal pueda estar distribuida entre varios procesadores, incluso con sus cachés y buffers de escritura privados. Cada procesador parece emitir y completar las operaciones a la memoria de una vez y de forma atómica en el orden del programa —es decir, una operación a la memoria no parece haber sido emitida hasta que la previa ha sido completada— y la memoria común parece servir a estas peticiones una cada vez, de manera entrelazada de acuerdo con un orden arbitrario. Las operaciones a la memoria aparecen como *atómicas* dentro de este orden; es decir, deben aparecer globalmente (a todos los procesadores) como si una operación en este orden entrelazado consistente se ejecuta y se completa antes de que empiece la siguiente.

Al igual que en el caso de la coherencia, no es importante en qué orden se emitan realmente las operaciones a la memoria o incluso cuándo se completan. Lo que im-

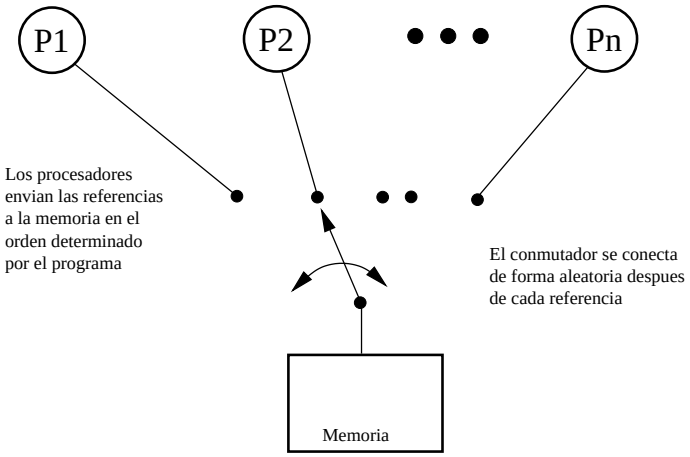


Figura 6.22: Abstracción desde el punto de vista del programador del subsistema de memoria bajo el modelo de consistencia secuencial.

porta es que parezcan completarse en un orden que no viole la consistencia secuencial. Veámoslo mediante el ejemplo que se muestra en la figura 6.23.

P_1	P_2
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

Figura 6.23: Órdenes entre los accesos sin sincronización.

En este ejemplo, el resultado (0,2) para (A,B) no estaría permitido bajo la consistencia secuencial (preservando nuestra intuición) dado que parecería que las escrituras de A y B en el proceso P_1 se han ejecutado fuera de orden. Sin embargo, las operaciones de memoria podrían ejecutarse y completarse en el orden 1b, 1a, 2b, 2a. No importa que se hayan completado fuera de orden, dado que el resultado (1,2) de la ejecución es la misma que si las operaciones se hubiesen ejecutado y completado en el orden del programa. Por otra parte, la ejecución 1b, 2a, 2b, 1a no cumpliría la consistencia secuencial, ya que produciría el resultado (0,2) que no está permitido bajo este paradigma.

La implementación de la consistencia secuencial hace necesario que el sistema (software y hardware) preserve las restricciones intuitivas definidas arriba. Existen dos restricciones. La primera es que las operaciones de memoria de un proceso se hagan visibles en el *orden del programa*. El segundo requisito, necesario para garantizar que el orden total sea consistente para todos los procesos, es que las operaciones aparezcan atómicas; es decir, que parezca que una operación se ha completado con respecto a todos los procesos antes que la siguiente operación en el orden total haya sido emitida. El truco de la segunda restricción es hacer que las escrituras parezcan atómicas, especialmente en un sistema con varias copias de un bloque que necesita ser informado ante una escritura. La *atomicidad de las escrituras*, incluidas en la definición de consistencia secuencial, implica que la posición en la cual parece realizarse una escritura en el orden total debe ser la misma con respecto a todos los procesadores. Esto asegura que nada de lo que un procesador haga *después* de que haya visto el nuevo valor producido por una

escritura se hace visible a los otros procesadores antes de que ellos también hayan visto el nuevo valor para esa escritura. De hecho, mientras que la coherencia (serialización de las escrituras) dice que las escrituras a la misma localización deben parecerles a todos los procesadores que han ocurrido en el mismo orden, la consistencia secuencial dice que todas las escrituras (a cualquier localización) deben parecerles a todos los procesadores que han ocurrido en el mismo orden. El siguiente ejemplo muestra por qué es importante la escritura atómica.

Ejemplo Consideremos los tres procesos que se muestran en la figura 6.24. Mostrar cómo la no preservación de la atomicidad en la escritura viola la consistencia secuencial.

Respuesta Dado que P_2 espera hasta que A sea 1 y después hace que B valga 1, y dado que P_3 espera hasta que B vale 1 y únicamente después lee el valor de A, por transitividad podríamos deducir que P_3 encontrará que A vale 1. Si P_2 puede continuar después de la lectura de A y escribir B antes de tener garantizado que P_3 ha visto el nuevo valor de A, entonces P_3 puede leer el nuevo valor de B y el viejo valor de A de su caché, violando nuestra intuición de la consistencia secuencial.

P_1		P_2		P_3
A=1;	$\longrightarrow$	while (A==0);		
		B=1;	$\longrightarrow$	while (B==0);
				print A

Figura 6.24: Ejemplo que ilustra la importancia de la atomicidad de las escrituras para la consistencia secuencial.

El orden del programa de cada proceso impone un orden parcial sobre el conjunto de todas las operaciones; es decir, impone una ordenación sobre el conjunto de operaciones que emite dicho proceso. Un intercalamiento de las operaciones de diferentes procesos definen un orden total en el conjunto de todos las operaciones. En realidad, para cada posible intercalamiento se define un orden total diferente dando lugar a un gran número de posibles órdenes totales. Este hecho da lugar a las siguientes definiciones:

> *Ejecución secuencialmente consistente.* Una ejecución de un programa se dice que es secuencialmente consistente si los resultados que produce son los mismos que los que produce uno de los posibles órdenes totales (intercalados como se ha definido anteriormente). Es decir, debe existir un orden total o intercalación de órdenes de los programas de los procesos que den lugar al mismo resultado.
>
> *Sistema secuencialmente consistente.* Un sistema es secuencialmente consistente si cualquier posible ejecución de ese sistema se corresponde con (produce el mismo resultado que) alguno de los posibles ordenes totales definidos arriba.

Condiciones suficientes para preservar la consistencia secuencial

Es posible definir un conjunto de condiciones suficientes que garanticen la consistencia secuencial en un multiprocesador. El siguiente conjunto se usa de forma habitual debido a su relativa simplicidad sin ser excesivamente restrictivas:

1. Todo proceso emite las peticiones a la memoria en el orden especificado por el programa.
2. Después de emitir una operación de escritura, el proceso que la emitió espera ha que se haya realizado la escritura antes de emitir su siguiente operación.
3. Después de emitir una operación de lectura, el proceso que la emitió espera a que la lectura se haya completado, *y* también espera a que se complete la escritura que generó el valor devuelto por la lectura, antes de emitir su siguiente operación. Es decir, si la escritura que generó el valor que se obtiene mediante la lectura ha sido realizada con respecto a este procesador (y debe haberlo hecho si se ha devuelto ese valor) entonces el procesador debe esperar hasta que la escritura se haya realizado con respecto a todos los procesadores.

La tercera condición es la que asegura la atomicidad de la escritura, y es bastante exigente. No es simplemente una restricción local, ya que la lectura debe esperar hasta que escritura que le precede de forma lógica sea visible de forma global. Obsérvese que estas son condiciones suficientes, más que condiciones necesarias. La consistencia secuencial puede preservarse con menos serialización en muchas situaciones.

6.2.2 Otros modelos de consistencia

Con el orden del programa definido en términos del programa fuente, está claro que para que se cumplan estas condiciones el compilador no debe cambiar el orden de las operaciones a la memoria en su presentación al procesador. Desafortunadamente, muchas de las optimizaciones empleadas en los compiladores y procesadores violan esta condición. Por ejemplo, los compiladores reordenan rutinariamente los accesos a diferentes localizaciones dentro de un proceso, así el procesador puede emitir accesos a memoria sin cumplir el orden del programa visto por el programador. Explicitamente, los programas paralelos usan compiladores uniprocesador, que sólo se preocupan de preservar las dependencias a la misma localización.

Incluso en el caso de que el compilador preserve el orden del programa, los modernos procesadores usan mecanismos sofisticados como buffers de escritura, memoria entrelazada, encauzamiento y técnicas de ejecución desordenada. Éstas permiten que las operaciones a la memoria de un proceso sean emitidas, ejecutadas y/o completadas sin seguir el orden del programa. Estas optimizaciones en la arquitectura y el compilador funcionan en los programas secuenciales ya que en ellos sólo es necesario preservar las dependencias entre los accesos a la misma localización de la memoria.

Debido a las fuertes restricciones que impone la consistencia secuencial, se han propuesto otros modelos más relajados que permiten implementaciones de más rendimiento y que todavía preservan un modelo de programación simple. De hecho, existen ciertos modelos relajados que mantienen la propiedad de una semántica de ejecución idéntica a la que sería bajo un modelo de consistencia secuencial. En la figura 6.25 se muestran algunos de los modelos propuestos. Una flecha de un modelo a otro indica que el segundo permite más optimizaciones que el primero y, por tanto, proporciona mayores prestaciones a costa, posiblemente, de un interfaz más complejo. Dos modelos no conectados con flechas no pueden compararse en términos de rendimiento o programabilidad. Las flechas discontinuas aparecen en el caso de considerar un revisión particular de *processor consistency* y *release consistency* propuesta por J. Hennessy, K. Gharachorloo y A.

Gupta[1].

--

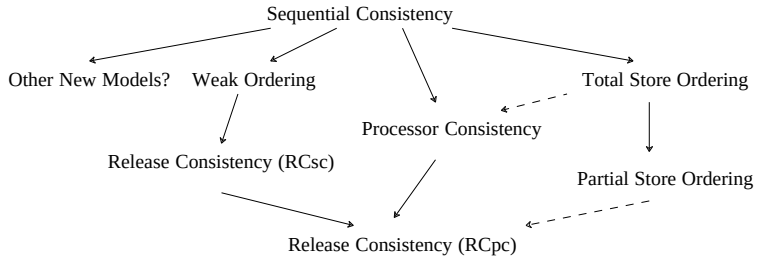


Figura 6.25: Modelos de consistencia de memoria.

Para entender las diferencias entre los diferentes modelos relajados y las posibles implicaciones para una implementación de los mismos, la forma más sencilla es definir los modelos en términos de las ordenaciones entre lecturas y escrituras *realizadas por un único procesador* que preserva cada modelo. Es posible establecer cuatro ordenaciones en función de las cuatro posibles dependencias existentes entre dos instrucciones:

1. $R \rightarrow R$: una lectura seguida de una lectura.
2. $R \rightarrow W$: una lectura seguida de una escritura, que siempre se preserva si las operaciones son sobre la misma dirección, dado que se trata de una antidependencia.
3. $W \rightarrow W$: una escritura seguida de otra escritura, que siempre se preserva si las operaciones son sobre la misma dirección, dado que se trata de una dependencia de salida.
4. $W \rightarrow R$: una escritura seguida de una lectura, que siempre se preserva si son sobre la misma dirección, dado que se trata de una dependencia verdadera.

Si existe una dependencia entre una lectura y una escritura, la semántica de un programa ejecutado sobre un uniprocesador exige que las operaciones estén ordenadas. Si no existen dependencias, el modelo de consistencia de la memoria determina qué órdenes deben preservarse. Un modelo de consistencia secuencial exige que se preserven los cuatro órdenes, lo que equivale a considerar un único módulo de memoria centralizado que serializa las operaciones de todos los procesadores, o asumir que todas las lecturas y escrituras a la memoria se comportan como barreras.

Cuando se relaja una ordenación, simplemente queremos decir que permitimos la finalización anticipada de una operación por parte de un procesador. Por ejemplo, relajar la ordenación $W \rightarrow R$ significa que permitimos que una lectura que era posterior a una escritura se complete antes. Hay que recordar que un escritura no se completa (realiza) hasta que se han completado todas las invalidaciones, así, permitiendo que ocurra una lectura después de un fallo en una operación de escritura anterior pero antes de que las invalidaciones se hayan realizado ya no se preserva este orden.

En realidad, un modelo de consistencia no restringe el orden de los eventos, sino que dice qué ordenaciones pueden ser *observadas*. Por ejemplo, en la consistencia secuencial,

[1]Revision to "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", Technical Repor CSL-TR-93-568, Stanford University, April 1993.

el sistema debe parecer que preserva los cuatro ordenes descritos anteriormente, aunque en la práctica se permite la reordenación. Este sutileza permite implementaciones que usan trucos que reordenan eventos sin permitir que dichas reordenaciones sean observadas. En la consistencia secuencial una implementación puede, por ejemplo, permitir que un procesador, P, inicie otra escritura antes de que la escritura anterior haya sido completada, siempre que P no permita que el valor de la última escritura sea visible antes de que la escritura anterior se complete.

El modelo de consistencia también debe definir la ordenación existente entre los accesos a las variables de sincronización, que actúan como barreras, y el resto de los accesos. Cuando una máquina implementa la consistencia secuencial, todas las lecturas y escrituras, incluyendo los accesos de sincronización, son barreras y por tanto deben mantenerse ordenadas. Para modelos más débiles, debemos especificar las restricciones en la ordenación impuestas por los accesos de sincronización, al igual que las restricciones sobre las variables ordinarias. La restricción en la ordenación más simple es que que todo acceso de sincronización se comporta como una barrera. Si utilizamos S para indicar un acceso a una variable de sincronización, también podremos indicar estas restricciones con la notación $S \rightarrow W$, $S \rightarrow R$, $W \rightarrow S$ y $R \rightarrow S$. Hay que recordar que un acceso de sincronización es también una lectura o una escritura y que está afectado por la ordenación respecto a otros accesos de sincronización, lo que significa que existe una ordenación implícita $S \rightarrow S$.

Modelo de consistencia PRAM o de procesador

El primer modelo que examinaremos relaja la ordenación entre una escritura y una lectura (a direcciones diferentes), eliminando la ordenación $W \rightarrow R$; este modelo fue usado por primera vez en la arquitectura IBM 370. Estos modelos permiten almacenar las escrituras en buffers que pueden ser sobrepasadas por las lecturas, lo que ocurre siempre que el procesador permita que una lectura proceda antes de garantizar que una escritura anterior de ese procesador ha sido vista por todos los procesadores. Este modelo permite a la máquina ocultar algo de la latencia de una operación de escritura. Además, relajando únicamente una ordenación, muchas aplicaciones, incluso aquellas que no están sincronizadas, funcionan correctamente, aunque será necesaria una operación de sincronización para asegurar que se complete una escritura antes de que se realice una lectura. Si se ejecuta una operación de sincronización antes de una lectura (por ejemplo, un patrón $W...S...R$), entonces las ordenaciones $W \rightarrow S$ y $S \rightarrow R$ nos aseguran que la escritura será realizada antes que la lectura. A este modelo se le ha denominado *Processor consistency* o *total store ordering* (TSO), y muchas máquinas han utilizado de forma implícita este método. Este modelo es equivalente a realizar las escrituras con barreras. En la figura 6.26 resumiremos todos los modelos, mostrando los ordenes impuestos, y mostraremos un ejemplo en la figura 6.27.

Ordenamiento por almacenamiento parcial

Si permitimos que las escrituras no conflictivas se completen de forma desordenada, relajando la ordenación $W \rightarrow W$, llegamos a un modelo que ha sido denominado como *partial store ordering* (PSO). Desde el punto de vista de la implementación, esto permite el encauzamiento o solapamiento de las operaciones de escritura.

Consistencia débil

Una tercera clase de modelos relajados eliminan las ordenaciones $R \rightarrow R$ y $R \rightarrow W$, además de las otras dos. Estos modelos, denominados *weak ordering*, no preservan el orden entre las referencias, a excepción de las siguientes:

- Una lectura o escritura se completará antes que cualquier operación de sincronización ejecutada por el procesador después de la lectura o escritura según el orden del programa.
- Una operación de sincronización se completará antes que cualquier lectura o escritura que ocurra según el orden del programa después de la operación

Como se muestra en la figura 6.26, las únicas ordenaciones impuestas por la *ordenación débil* son las creadas por las operaciones de sincronización. Aunque hemos eliminado los ordenes $R \rightarrow R$ y $R \rightarrow W$, el procesador sólo puede aprovecharse de este hecho si dispone de lecturas no bloqueantes. En caso contrario, el procesador implementa implícitamente estos dos órdenes, dado que no se pueden ejecutar más instrucciones hasta que se complete la lectura. Incluso con lecturas no bloqueantes, la ventaja del procesador puede verse limitada dado que la principal ventaja ocurre cuando una lectura produce un fallo de caché y es improbable que el procesador pueda mantenerse ocupado durante los centenares de ciclos que se necesita para manejar un fallo de caché. En general, la principal ventaja de todos los modelos de consistencia débil viene del ocultamiento de la latencia de escritura más que de las latencias de lectura.

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	R →R, R →W, W→R, W→W	S→W, S→R, R→S, W→S, S→S
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	R →R, R →W, W→W	S→W, S→R, R→S, W→S, S→S
Partial store order	SPARC	R →R, R →W	S→W, S→R, R→S, W→S, S→S
Weak ordering	PowerPC		S→W, S→R, R→S, W→S, S→S
Release consistency	Alpha, MIPS		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_A$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

Figura 6.26: Ordenaciones impuestas por varios modelos de consistencia. Se muestran tanto los accesos ordinarios como los accesos a variables de sincronización.

Modelo de consistencia de liberación (*release*)

Es posible obtener un modelo más relajado todavía extendiendo la ordenación débil. Este modelo, llamado *release consistency* (consistencia en la liberación), distingue entre operaciones de sincronización que se usan para *adquirir* el acceso a una variable compartida (denotadas por S_A) y aquellas que *liberan* un objeto para permitir que otro procesador puede adquirir el acceso (denotadas por S_R). La consistencia en la liberación se basa en la observación de que en los programas sincronizados es necesaria una operación de adquisición antes de usar un dato compartido, y que una operación de liberación deber seguir a todas las actualizaciones a los datos compartidos y preceder en

el tiempo a la siguiente operación de adquisición. Esto nos permite relajar ligeramente la ordenación observando que una lectura o escritura que precede a una adquisición no necesitan completarse antes de la adquisición, y que una lectura o escritura que sigue a una liberación no necesitan esperar a dicha liberación. Por tanto, las ordenaciones que se preservan son las que implican únicamente a S_A y S_R, como se muestra en la figura 6.26; como muestra en el ejemplo de la figura 6.27, este modelo impone el menor número de ordenes de los cinco modelos.

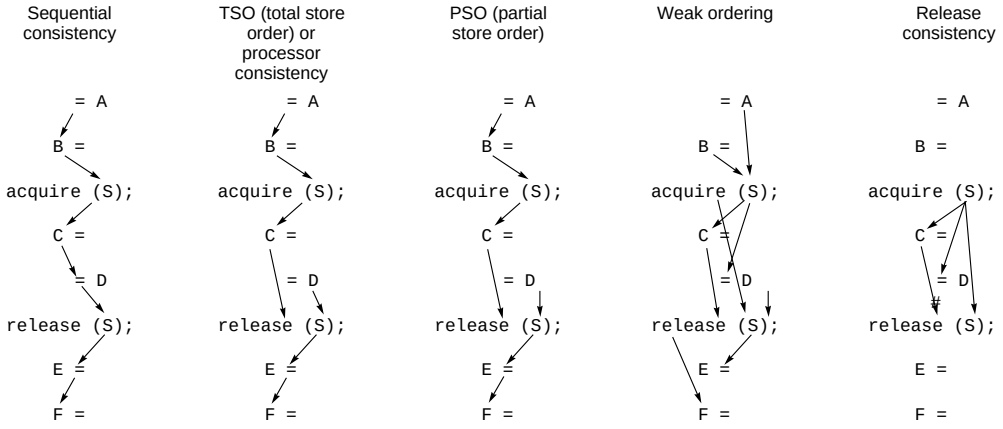


Figura 6.27: Ejemplos de los cinco modelos de consistencia vistos en esta sección que muestran la reducción en el número de órdenes impuestos conforme los modelos se hacen más relajados.

Para comparar los modelos de *release consistency* y *weak ordering*, consideremos qué ordenaciones son necesarias para este último modelo, para ello descompondremos cada S en S_A y S_R. Esto nos llevaría a ocho ordenaciones en relación con los accesos sincronizados y los accesos ordinarios y cuatro ordenaciones en relación únicamente con los accesos sincronizados. Con esta descripción, podemos ver que cuatro de las ordenaciones impuesta en *weak ordering* no aparecen en el modelo de *release consistency*: $W \rightarrow S_A$, $R \rightarrow S_A$, $S_R \rightarrow R$ y $S_R \rightarrow W$.

El modelo de *release consistency* es uno de los menos restrictivos, permitiendo un fácil chequeo y asegurando el funcionamiento de los programas sincronizados. Mientras que la mayoría de las operaciones son una adquisición o una liberación (una adquisición normalmente lee una variable de sincronización y la actualiza de forma atómica, una liberación únicamente escribe en ella), algunas operaciones, como una barrera, actúa de adquisición y liberación a la vez en cuyo caso el modelo pasa a ser equivalente al modelo de *weak ordering*.

También es posible considerar ordenaciones más débiles. Por ejemplo, en el modelo de *release consistency* no asociamos localizaciones de memoria a variables de sincronización específicas. Si obligamos a la adquisición de la misma variable de sincronización, V, antes de acceder a una dirección de memoria particular, M, podemos relajar el orden de accesos a M y la adquisición y relajación de las variables de sincronización distintas de V.

Consistencia de liberación perezosa

Consistencia de ingreso

6.2.3 Implementación de los modelos relajados

Los modelos de consistencia relajados pueden implementarse mediante la utilización de un poco de hardware adicional. La mayor parte de la complejidad recae sobre la implementación de memorias o de los sistemas de interconexión que saquen provecho de un modelo relajado. Por ejemplo, si la memoria o la interconexión no permite la múltiples peticiones por parte de un procesador, los beneficios de los modelos relajados más ambiciosos serán pequeños. Afortunadamente, la mayoría de los beneficios pueden conseguirse permitiendo un pequeño número de peticiones en curso de escritura y una de lectura por procesador. En esta sección describiremos implementaciones directas de los modelos *processor consistency* y *release consistency*.

El modelo TSO (*processor consistency*) se implementa permitiendo que un fallo de lectura sobrepase a las escrituras pendientes. Un buffer de escritura que soporte un chequeo para determinar si existen operaciones de escritura pendientes en el buffer a la misma dirección que una operación de lectura, junto con una memoria y un sistema de interconexión que soporte dos referencias pendientes por nodo, es suficiente para implementar este esquema. Cualitativamente, la ventaja de la consistencia a nivel de procesador sobre la consistencia secuencial es que permite ocultar la latencia de los fallos en las escrituras.

El modelo de *release consistency* permite una ocultación adicional de la latencia de escritura, y si el procesador soporta lecturas no bloqueantes, permite ocultar la latencia de las lecturas. Para permitir el ocultamiento de la latencia de las escrituras tanto como sea posible, el procesador debe permitir varias escrituras pendientes y que los fallos en las lecturas sobrepasen a las escrituras pendientes. Para maximizar el rendimiento, las escrituras deben completarse y eliminarse el buffer de escritura tan pronto como sea posible, lo que permite el avance de las lecturas pendientes. El soporte de la finalización temprana de las escrituras requiere que una escritura se complete tan pronto como los datos estén disponibles y antes de que se completen las invalidaciones pendientes (dado que nuestro modelo de consistencia lo permite). Para implementar este esquema, el procesador debe ser capaz de llevar la cuenta de las invalidaciones de cada escritura emitida. Después de que cada invalidación sea confirmada, el contador de invalidaciones pendientes para esa escritura se decrementa. Debemos asegurarnos que todas las invalidaciones pendientes de todas las escrituras emitidas se han completado antes de permitir que se complete una operación de liberación, de tal manera que sólo es necesario chequear los contadores de invalidaciones pendientes sobre cualquier escritura emitida cuando se ejecuta una liberación. La liberación se retiene hasta que todas las invalidaciones de todas las escrituras se han completado. En la práctica, se limita el número de escrituras pendientes en un momento dado, con lo que se facilita el seguimiento de las mismas y de las invalidaciones pendientes.

Para ocultar la latencia de las escrituras debemos disponer de una máquina que permita lecturas no bloqueantes; en caso contrario, cuando el procesador se bloquea, el progreso sería mínimo. Si las lecturas son no bloqueantes podemos simplemente permitir que se ejecuten, sabiendo que las dependencias entre los datos preservarán una ejecución correcta. Es poco probable, sin embargo, que la adición de lecturas no bloqueantes a un modelo de consistencia relajado mejore el rendimiento sustancialmente. Esta ganancia

limitada se debe al hecho de que los tiempo de respuesta en un multiprocesador en el caso de fallo en una lectura sea grande y la habilidad de ocultar esta latencia por parte del procesador es limitada. Por ejemplo, si las lecturas son no bloqueantes pero se ejecutan en orden por parte del procesador, entonces éste se bloqueará después de algunos ciclos de forma casi segura. Si el procesador soporta lecturas no bloqueantes y ejecución desordenada, se bloqueará tan pronto como el buffer de reordenación o las estaciones de reserva se llenen. Es muy probable que este hecho ocurra en un centenar de ciclos, mientras que un fallo puede costar un millar de ciclos.

6.2.4 Rendimiento de los modelos relajados

El rendimiento potencial de un modelo de consistencia más relajado depende de las capacidades de la máquina y de la aplicación que se ejecute. Para examinar el rendimiento de un modelo de consistencia de la memoria, debemos definir en primer lugar las características del hardware:

- El cauce emite una instrucción por ciclo de reloj y el manejo puede ser estático o dinámico. La latencia de todas las unidades funcionales es de un ciclo.
- Un fallo de caché supone 50 ciclos de reloj.
- La CPU incluye un buffer de escritura con capacidad para 16 escrituras.
- Las cachés son de 64KB y el tamaño de las líneas es de 16 bytes.

Para dar una visión de los aspectos que afectan al rendimiento potencial con diferentes capacidades hardware, consideraremos cuatro modelos hardware:

1. SSBR (*statically scheduled with blocking reads*) El procesador realiza una ejecución estática y las lecturas que incurren en fallo en la caché bloquean al procesador.
2. SS (*statically scheduled*) El procesador realiza una ejecución estática pero las lecturas no causan el bloqueo del procesador hasta que se usa el resultado.
3. DS16 (*dynamically scheduled with a 16-entry reorder buffer*) El procesador permite una ejecución dinámica y tiene un buffer de reordenación que permite tener hasta 16 instrucciones pendientes de cualquier tipo, incluyendo 16 instrucciones de acceso a memoria.
4. DS64 (*dynamically scheduled with a 64-entry reorder buffer*) El procesador permite una ejecución dinámica y tiene un buffer de reordenación que permite hasta 64 instrucciones pendientes de cualquier tipo. Este buffer es, potencialmente, lo suficientemente grande para ocultar la totalidad de la latencia ante un fallo de lectura en la caché.

La figura 6.28 muestra el rendimiento relativo para dos programas paralelos, LU y Ocean, para estos cuatro modelos hardware y para dos modelos de consistencia diferentes: TSO y *release consistency*. El rendimiento mostrado es relativo al rendimiento bajo una implementación directa de la consistencia secuencial. El aumento de rendimiento observado es mucho mayor en el caso de Ocean, ya que este programa presenta un ratio de fallos de caché mucho mayor y tiene una fracción significativa de fallos de escritura. Al interpretar los datos de la figura 6.28 hay que recordar que las cachés son bastantes pequeñas. Muchos diseñadores habrían incrementado el tamaño de la caché antes de incluir lecturas no bloqueantes o empezar a pensar en ejecución dinámica de las instrucciones. Esto hubiera hecho descender el ratio de fallos y las ventajas de los modelos relajados, al menos sobre estas aplicaciones.

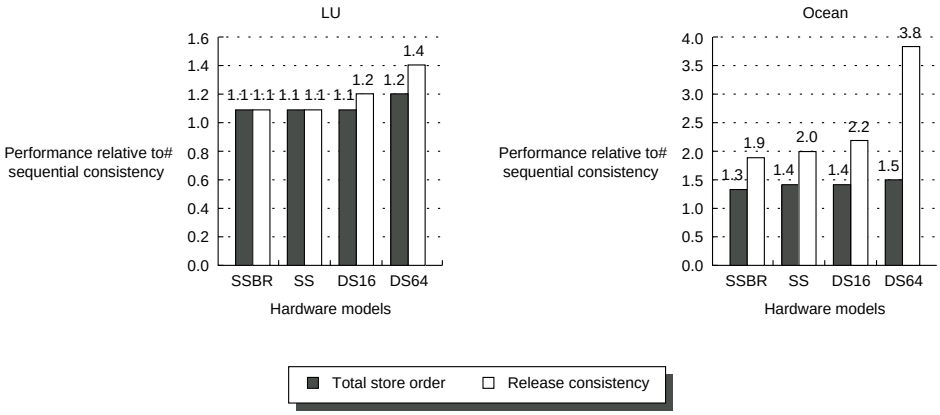


Figura 6.28: Rendimiento de los modelos de consistencia relajados sobre una variedad de mecanismos hardware.

Para finalizar, decir que, en la actualidad, la mayoría de las máquinas que soportan alguna clase de modelo de consistencia relajado, variando desde la *consistencia a nivel de procesador* a la *consistencia en la liberación*, y casi todos soportan la consistencia secuencial como una opción. Dado que la sincronización depende mucho de la máquina y es causa de errores, las expectativas nos dicen que la mayoría de los programadores usarán librerías estándar de sincronización para escribir sus programas sincronizados, haciendo la elección de un modelo de consistencia relajado invisible al programador y permitiendo mayor rendimiento.

6.3 Coherencia de las cachés

La bibliografía sobre la coherencia de caché para multiprocesadores es bastante amplia, casi cualquier libro de arquitectura de computadores incluye alguna cosa. Los más completos sin embargo son [Hwa93], [HP96], [Zar96] y [Fly95]. Los apuntes se han realizado sobre todo a partir del [Hwa93] y del [Zar96]. Del [Fly95] se han extraído algunos de los diagramas de estados y constituye una buena lectura adicional. Del [HP96] se han extraído algunas ideas y definiciones y vienen algunos experimentos sobre rendimiento que no se han incluido en los apuntes.

La memoria es uno de los componentes principales de cualquier sistema. La estructura de la memoria va a tener un importante impacto en el rendimiento general del sistema. Se analizan en esta parte los diferentes protocolos de coherencia fundamentales en los actuales sistemas multiprocesadores con memoria compartida.

En cursos anteriores el alumno se ha encontrado ya con el tema de las cachés y su coherencia en sistemas con un único procesador y una sola caché. No se explicarán aquí los diversos tipos de caché ni su realización, ya que se supone que es un tema conocido. Lo que se expone a continuación es el problema de mantener la coherencia del sistema de memoria en sistemas multiprocesadores con varias cachés y memoria compartida.

6.3.1 El problema de la coherencia de las cachés

Pensemos por un momento cuál es el modelo intuitivo que tenemos de lo que debe ser una memoria. La memoria debe proporcionar un conjunto de direcciones para almacenar valores, y cuando se lea una de estas direcciones debe devolver el último valor escrito en ella. Es en esta propiedad fundamental de las memorias en la que descansan los programas secuenciales cuando usamos la memoria para comunicar un valor desde un punto del programa donde se calcula a otros puntos donde es usado. También nos basamos en esta propiedad cuando el sistema usa un espacio de direcciones compartido para comunicar datos entre hebras o procesos que se están ejecutando en un procesador. Una lectura devuelve el último valor escrito en esa dirección, sin importar el proceso que escribió dicho valor. Las cachés (antememorias) no interfieren con el uso de múltiples procesos en un procesador, ya que todos ellos ven la memoria a través de la misma jerarquía de cachés. En el caso de usar varios procesadores, nos gustaría poder basarnos en la misma propiedad cuando dos procesos se ejecuten sobre diferentes procesadores de tal forma que el resultado de ejecutar un programa que usa varios procesos sea el mismo independientemente de si los procesos se ejecutan o no en diferentes procesadores físicos. Sin embargo, cuando dos procesos ven la memoria compartida a través de diferentes cachés, existe el peligro de que uno vea el nuevo valor en su caché mientras que el otro todavía vea el antiguo.

El problema de la coherencia en sistemas multiprocesadores se puede ver claramente mediante un ejemplo. Supongamos dos procesadores, cada uno con su caché, conectados a una memoria compartida. Supongamos que ambos procesadores acceden a una misma posición X en memoria principal. La figura 6.29 muestra los contenidos de las cachés y memoria principal asumiendo que ninguna caché contiene inicialmente la variable que vale inicialmente 1. También se asume una caché *write-through*. Después de que el valor en X ha sido escrito por A, la caché de A y la memoria contienen el nuevo valor, pero la cache de B no, y si B lee el valor de X, ¡leerá 1 y no 0 que es el valor de la memoria! Este es el problema de la coherencia de cachés en multiprocesadores.

Time	Acción	Caché A	Caché B	Memoria (X)
1	CPU A lee X	1		1
2	CPU B lee X	1	1	1
3	CPU A escribe 0 en X	0	1	0

Figura 6.29: El problema de la coherencia de cachés con dos procesadores.

Informalmente se puede decir que un sistema de memoria es *coherente* si cualquier lectura de un dato devuelve el valor más recientemente escrito de ese dato. Esta definición, aunque intuitivamente correcta, es vaga y simple; la realidad es bastante más compleja. Esta definición simple contiene dos aspectos diferentes del comportamiento del sistema de memoria, siendo los dos críticos a la hora de escribir programas en memoria compartida. El primer aspecto, llamado *coherencia* definen los datos devueltos por una lectura. El segundo aspecto, llamado *consistencia*, determina cuando un valor escrito será devuelto por una lectura.

Demos primero una definición formal para la coherencia. Se dice que un sistema de memoria es coherente si se cumple:

1. Una lectura por un procesador P de una posición X, que sigue de una escritura de

P a X, sin que ningún otro procesador haya escrito nada en X entre la escritura y la lectura de P, siempre devuelve el valor escrito por P.

2. Una lectura por un procesador de la posición X, que sigue una escritura por otro procesador a X, devuelve el valor escrito si la lectura y escritura están suficientemente separados y no hay otras escrituras sobre X entre los dos accesos.
3. Las escrituras a la misma posición están serializadas, es decir, dos escrituras a la misma posición por cualquiera dos procesadores se ven en el mismo orden por todos los procesadores. Por ejemplo, si se escriben los valores 1 y 2 en una posición, los procesadores nunca pueden leer el valor 2 y luego el 1.

La primera condición preserva el orden del programa y debe ser así incluso para monoprocesadores. La segunda condición define lo que se entiende por tener una visión *coherente* de la memoria. Con estas condiciones el problema para tener un sistema coherente queda resuelto, sin embargo, todavía queda el problema de la consistencia.

Para ver que el problema de la consistencia es en realidad complejo, basta con observar que en realidad no es necesario que una lectura sobre X recoja el valor escrito de X por otro procesador. Si por ejemplo, una escritura sobre X precede una lectura sobre X por otro procesador en un tiempo muy pequeño, puede ser imposible asegurar que la lectura devuelva el valor tan recientemente escrito, ya que el valor a escribir puede incluso no haber abandonado el procesador que escribe. La cuestión sobre cuándo exactamente un valor escrito debe ser visto por una lectura se define por el *modelo de consistenciade la memoria*.

Otro ejemplo que ilustra el problema de incoherencia de la memoria se puede ver en la figura 6.30. En esta figura se muestran tres procesadores con cachés propias y conectados a través de un bus común a una memoria compartida. La secuencia de accesos a la localización u hecha por cada procesador es la que se indica mediante los números que aparecen al lado de cada arco. Veamos cuáles son los valores leídos por P_1 y P_2 dependiendo del tipo de caché utilizada.

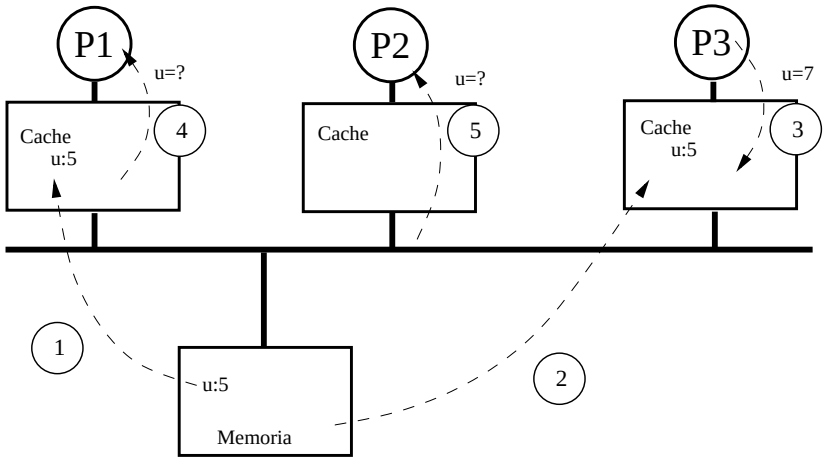


Figura 6.30: Ejemplo del problema de coherencia de las cachés.

- Si consideramos una caché *write-through* la modificación realizada por el procesador P_3 hará que el valor de u en la memoria principal sea 7. Sin embargo, el procesador P_1 leerá el valor de u de su caché en vez de leer el valor correcto de la memoria principal.

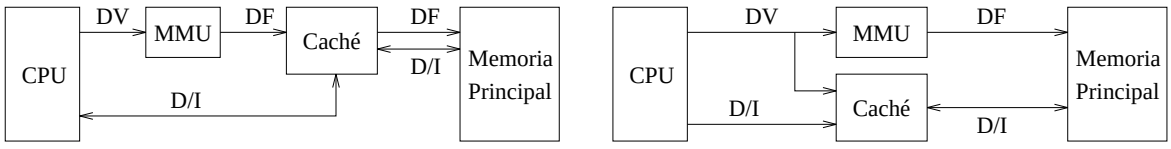
- Si considera una caché *writeback* la situación empeora. En este caso el procesador P_3 únicamente activará el bit de modificado en el bloque de la caché en donde tiene almacenado u y no actualizará la memoria principal. Únicamente cuando ese bloque de la caché sea reemplazado su contenido se volcará a la memoria principal. Ya no será únicamente P_1 el que leerá el valor antiguo, sino que cuando P_2 intente leer u y se produzca un fallo en la caché, también leerá el valor antiguo de la memoria principal. Por último, si varios procesadores escriben distintos valores en la localización u con este tipo de cachés, el valor final en la memoria principal dependerá del orden en que los bloques de caché que contienen u se reemplazan, y no tendrá nada que ver con el orden en que las escrituras a u ocurrieron.

Los problemas de coherencia en la caché también ocurren cuando utilizamos un único procesador en el caso de las operaciones de E/S. La mayoría de estas operaciones se realizan a través de dispositivos DMA con lo que es posible que que el contenido de la memoria principal y la memoria caché dejen de ser coherentes. Sin embargo, dado que las operaciones de E/S son mucho menos frecuentes que las operaciones de acceso a memoria, se han adoptado soluciones sencillas como usar direcciones de memoria que se marcan como no almacenables en la memoria caché o eliminar todos los bloques existentes en las cachés de las páginas que se van a utilizar en la operación de E/S antes de proceder a realizar dicha operación. En la actualidad, casi todos los microprocesadores proporcionan mecanismos para soportar la coherencia de cachés.

6.3.2 Direcciones físicas y virtuales, problema del *aliasing*

Hay dos formas de conectar la caché al procesador. Una manera consiste en colocar la caché después de los bloques TLB (*Transaction Lookaside Buffer*) o MMU (*Memory Management Unit*) que realizan la transformación de dirección virtual a dirección física. A estas cachés conectadas de esta forma se les llama *cachés con direcciones físicas,* y un esquema de su conexionado se muestra en la figura 6.31(a).

La otra posibilidad consiste en conectar la caché directamente a las direcciones del procesador, es decir, a las direcciones virtuales tal y como se muestra en la figura 6.31(b). En este caso tenemos las *cachés con direcciones virtuales.* Ambas tienen sus ventajas e inconvenientes como veremos a continuación.



(a) Caché con direcciones físicas.

(b) Caché con direcciones virtuales.

Figura 6.31: Direcciones físicas y virtuales en la caché. (DV=Dirección Virtual, DF=Dirección Física, D/I=Datos o Instrucciones).

Las cachés con dirección física son sencillas de realizar puesto que la dirección que se especifica en la etiqueta de la caché (*tag*) es única y no existe el problema del *aliasing* que tienen las cachés con dirección virtual. Como no tiene problemas de aliasing no

es necesario vaciar (*flush* la caché, y además el sistema operativo tiene menos *bugs* de caché en su núcleo.

El problema de las cachés con direcciones físicas es que cuesta más tiempo acceder a la caché puesto que hay que esperar a que la unidad MMU/TLB acabe de traducir la dirección. La integración de la MMU/TLB y caché en el mismo VLSI chip, muchas veces incluso con el procesador, alivia este problema.

Lo cierto es que la mayoría de sistemas convencionales utilizan la caché con direcciones físicas puesto que son muy simples y requieren una casi nula intervención del núcleo del sistema operativo.

En la caché con direcciones virtuales, el acceso a la caché se realiza al mismo tiempo y en paralelo con la traducción de dirección por parte de la MMU. Esto permite un acceso a la caché más rápido que con las direcciones físicas.

Pero con las direcciones virtuales surge el problema del **aliasing**. Es a veces frecuente que dos procesos diferentes utilicen las mismas direcciones virtuales cuando en realidad las direcciones físicas van a ser diferentes puesto que nada tiene que ver un proceso con el otro. Cuando esto ocurre, resulta que el índice/etiqueta en la caché coincide en ambos casos cuando en realidad se trata de direcciones reales diferentes. La forma de solucionar esto consiste en vaciar completamente la caché cada vez que hay un problema de *aliasing*, lo cual ocurre siempre que se cambia de contexto en Unix por ejemplo.

Estas operaciones de vaciado continuo de la caché produce un rendimiento muy bajo en la caché y del sistema en general al ocupar el bus en vaciar una y otra vez la caché. En Unix incluso el vaciado de la caché puede no ser suficiente para realizar ciertas operaciones del núcleo, llamadas al sistema y operaciones de I/O. También el depurado de programas resulta prácticamente imposible con este tipo de caché.

Una solución consiste en añadir información sobre el proceso, o contexto, a la etiqueta de la caché, o también, añadir la dirección física, pero estos métodos pueden eventualmente también degradar el rendimiento del sistema.

6.3.3 El problema de la falsa compartición

6.3.4 Soluciones a los problemas de coherencia

Antes de estudiar las diferentes técnicas para solucionar el problema de coherencia, sería interesante realizar una definición más formal de la propiedad de coherencia.

En primer lugar empezaremos con la definición de algunos términos en el contexto de los sistemas de memoria uniprocesador, para después extender dicha definición para multiprocesadores.

Operación de Memoria Un acceso a una dirección de la memoria para realizar una lectura, escritura u operación atómica de lectura-modificación-escritura. Las instrucciones que realizan múltiples lecturas y escrituras, tales como las que aparecen en muchos de los conjuntos de instrucciones complejos, pueden ser vistas como un conjunto de operaciones de memoria, y el orden en que dichas operaciones deben ejecutarse se especifica en la instrucción.

Ejecución atómica Las operaciones de memoria de una instrucción se ejecutan de forma atómica una respecto a la otra según el orden especificado. Es decir, to-

dos los aspectos de una operación deben parecer que se han ejecutado antes de cualquier aspecto de la operación siguiente.

Emisión Una operación de memoria es emitida cuando deja el entorno interno del procesador y se presenta en el sistema de memoria, que incluye las cachés, los buffers de escritura, el bus, y los módulos de memoria. Un punto muy importante es que el procesador sólo observa el estado del sistema de memoria mediante la emisión de operaciones de memoria; es decir, nuestra noción de lo que significa que una operación de memoria haya sido *realizada* es que parezca que ha tenido lugar desde la perspectiva del procesador.

Realizada con respecto a un procesador Una operación de escritura se dice que ha sido realizada con respecto al procesador cuando una lectura posterior por el procesador devuelve el valor producido por esa escritura o una posterior. Una operación de lectura se dice que ha sido realizada con respecto al procesador cuando escrituras posteriores emitidas por el procesador no pueden afectar al valor devuelto por la lectura.

Las mismas operaciones realizadas con respecto a un procesador se pueden aplicar al caso paralelo; basta con reemplazar en la definición "el procesador" por "un procesador". El problema que aparece con el concepto de orden y de las nociones intuitivas de 'posterior' y 'última', es que ahora ya no tenemos un orden definido por el programa sino órdenes de programas separados para cada proceso y esos órdenes de programa interactúan cuando acceden al sistema de memoria. Una manera agudizar nuestra noción intuitiva de sistema de memoria coherente es determinar qué pasaría en el caso de que no existieran cachés. Cada escritura y lectura a una localización de memoria accedería a la memoria física principal y se realizaría con respecto a todos los procesadores en dicho punto, así la memoria impondría un orden secuencial entre todas las operaciones de lectura y escritura realizadas en dicha localización. Además, las lecturas y escrituras a una localización desde cualquier procesador individual deberán mantener el orden del programa dentro de esta ordenación secuencial. No hay razón para creer que el sistema de memoria deba intercalar los accesos independientes desde distintos procesadores en una forma predeterminada, así cualquier intercalado que preserve la ordenación de cada programa individual es razonable. Podemos asumir algunos hechos básicos; las operaciones de cada procesador será realizadas en algún momento dado. Además, nuestra noción intuitiva de "último" puede ser vista como el más reciente en alguno de las hipotéticas ordenaciones secuenciales que mantienen las propiedades discutidas en este párrafo.

Dado que este orden secuencial debe ser consistente, es importante que todos los procesadores vean las escrituras a una localización en el mismo orden.

Por supuesto, no es necesario construir una ordenación total en cualquier punto de la máquina mientras que se ejecuta el programa. Particularmente, en un sistema con cachés no queremos que la memoria principal vea todas las operaciones de memoria, y queremos evitar la serialización siempre que sea posible. Sólo necesitamos estar seguros de que el programa se comporta como si se estuviese forzando un determinado orden secuencial.

Más formalmente, decimos que un sistema de memoria multiprocesador es *coherente* si el resultado de cualquier ejecución de un programa es tal que, para cada localización, es posible construir una hipotética ordenación secuencial de todas las operaciones realizadas sobre dicha localización (es decir, ordenar totalmente todas las lecturas/escrituras emitidas por todos los procesadores) que sea consistente con los resultados de la ejecu-

ción y en el cual:

1. las operaciones emitidas por un procesador particular ocurre en la secuencia arriba indicada en el orden en las cuales se emiten al sistema de memoria por ese procesador, y
2. el valor devuelto por cada operación de lectura es el valor escrito por la última escritura en esa localización en la secuencia arriba indicada.

Está implícita en la definición de coherencia la propiedad de que todas las escrituras a una localización (desde el mismo o diferente procesador) son vistas en el mismo orden por todos los procesadores. A esta propiedad se le denomina *serialización de las escrituras*. Esto significa que si las operaciones de lectura del procesador P_1 a una localización ven el valor producido por la escritura w_1 (de, por ejemplo, P_2) antes que el valor producido por la escritura w_2 (de, por ejemplo, P_3), entonces las lecturas de otro procesador P_4 (o P_2 o P_3) no deben poder ver w_2 antes que w_1. No hay necesidad de un concepto análogo para la serialización de las lecturas, dado que los efectos de las lecturas no son visibles para cualquier procesador distinto de aquel que la realiza.

El resultado de un programa puede ser visto como los valores devueltos por las operaciones de lectura que realiza, quizás aumentado con un conjunto implícito de lecturas a todas las localizaciones al final del programa. De los resultados no podemos determinar el orden en el cual las operaciones se realizaron realmente por la máquina, sino el orden en el que parece que se ejecutaron. De hecho, no es importante en qué orden las cosas ocurren realmente en la máquina o cuándo cambió cada bit, dado que no es detectable; todo lo que importa es el orden en el cual las cosas parecen haber ocurrido, como detectable a partir de los resultados de una ejecución. Este concepto será más importante cuando discutamos los modelos de consistencia de la memoria. Para finalizar, una definición adicional que necesitamos en el caso de los multiprocesadores es la de completitud de una operación: Una operación de lectura o escritura se dice que *se ha completado* cuando se ha realizada con respecto a todos los procesadores.

6.3.5 Esquemas de coherencia de las cachés

Existen numerosos protocolos para mantener la coherencia de las cachés en un sistema multiprocesador con memoria compartida. En primer lugar hay dos formas de abordar el problema. Una forma es resolver el problema de la coherencia por software, lo que implica la realización de compiladores que eviten la incoherencia entre cachés de datos compartidos. La otra aproximación es proveer mecanismos hardware que mantengan de forma continua la coherencia en el sistema, siendo además transparente al programador. Como esta segunda forma es la más utilizada, nos centraremos en ella de manera que todos los protocolos que siguen se refieren a soluciones hardware al problema de la coherencia.

Podemos distinguir también dos tipos de sistemas multiprocesadores; por un lado están los sistemas basados en un único bus, con un número no demasiado grande de procesadores, cada uno con su caché, y por otro lado están los sistemas más complejos con varios buses o varias subredes con un gran número de nodos procesadores. En el primero parece más adecuado un tipo de protocolo que esté continuamente sondeando o *fisgando* el bus común para ver qué transacciones podrían introducir incoherencia y actuar en consecuencia. A estos protocolos se les llama de sondeo o *snoopy* puesto que fisgan el bus para detectar incoherencia. Básicamente cada nodo procesador tendrá

los bits necesarios para indicar el estado de cada línea de su caché y así realizar las transacciones de coherencia necesarias según lo que ocurra en el bus en cada momento.

En el segundo tipo de sistemas, con varias subredes locales y un amplio número de procesadores, un protocolo de sondeo es complicado de realizar puesto que las actividades en los sub-buses son difíciles de fisgar, ya que las transacciones van a ser locales a estos. Para estos sistemas más complejos se utiliza un tipo de protocolo basado en directorio, que consiste en la existencia de un directorio común donde se guardan el estado de validez de las líneas de las cachés, de manera que cualquier nodo puede acceder a este directorio común.

Dado que los sistemas multiprocesadores basados en memoria compartida y caché suelen estar basados en bus, o una jerarquía de buses con no muchos niveles, los protocolos de coherencia suelen ser de sondeo, por esto se le dedica una atención especial a este tipo de protocolos.

Entre los protocolos de coherencia, tanto si son de sondeo como si no, existen en general dos políticas para mantener la coherencia: *invalidación en escritura* (*write invalidate*) y *actualización en escritura* (*write update*). En la política de invalidación en escritura (también llamada política de coherencia dinámica), siempre que un procesador modifica un dato de un bloque en la caché, invalida todas las demás copias de ese bloque guardadas en las otras cachés. Por contra, la política de actualización en escritura actualiza las copias existentes en las otras cachés en vez de invalidarlas.

En estas políticas, los protocolos usados para la invalidación o actualización de las otras copias dependen de la red de interconexión empleada. Cuando la red de interconexión permite el *broadcast* (como en un bus), los comandos de invalidación y actualización pueden ser enviados a todas las cachés de forma simultánea. A los protocolos que utilizan esta técnica se les denominan, tal y como se ha comentado antes, *protocolos de de sondeo o* snoopy, dado que cada caché *monitoriza* las transacciones de las demás cachés. En otras redes de interconexión, donde el *broadcast* no es posible o causa degradación (como en las redes multietapa), el comando de invalidación/actualización se envía únicamente a aquellas cachés que tienen una copia del bloque. Para hacer esto, se usa frecuentemente un directorio centralizado o distribuido. Este directorio tiene una entrada para cada bloque. La entrada por cada bloque contiene un puntero a cada caché que tiene una copia del bloque. También contiene un bit que especifica si el permiso de actualizar el bloque sólo lo tiene una determinada caché. A los protocolos que usan este esquema se les denominan *protocolos de caché basados en directorio*.

6.4 Protocolos de sondeo o *snoopy* (medio compartido)

Tal y como se ha comentado antes, hay dos tipos de protocolos atendiendo al mecanismo que utilizan. Así se tienen el *write-update* o *write-broadcast* o *actualizar en escritura*, y el *write-invalidate* o *invalidar en escritura*.

El primer método, el de **invalidar en escritura**, se basa en asegurar que un procesador tiene acceso exclusivo a un dato antes de que acceda a él. Esto se consigue invalidando todas las líneas de todas las cachés que contengan un dato que está siendo escrito en ese momento. Este protocolo es con diferencia el más usado tanto en protocolos de sondeo como en los de directorio. El acceso exclusivo asegura que no hay

otras copias leíbles o escribibles del dato cuando se realiza la escritura, ya que todas las copias del dato se invalidan en el momento de la escritura. Esto asegura que cuando otro procesador quiera leer el dato falle su caché y tenga que ir a memoria principal a buscarlo.

Si dos procesadores intentan escribir a la vez el mismo dato, hay uno que ganará necesariamente, mientras que el otro deberá esperar a que este escriba momento en el cual deberá volver a obtener la copia buena para poder volver a escribir. Por lo tanto, este protocolo asegura la escritura en serie.

El segundo método, alternativa al anterior, es el de actualizar todas las copias de las cachés cuando se escribe un determinado dato. Este protocolo se llama *actualizar en escritura*. Para mantener los requisitos de ancho de banda de este protocolo bajo control, es interesante realizar un seguimiento de si una palabra en la caché está compartida o no, o sea, que se encuentra también en otras cachés. Si no está compartida entonces no es necesario actualizar el resto de cachés.

La belleza de la coherencia de caché basada en *snooping* reside en que toda la maquinaria para resolver un problema complicado se reduce a una pequeña cantidad de interpretación extra de eventos que ocurren de forma natural en el sistema. El procesador permanece inalterado. No existen operaciones explícitas de coherencia insertadas en el programa. Extendiendo los requisitos del controlador de la caché y explotando las propiedades del bus, las lecturas y escrituras que son inherentes al programa se usan de forma implícita para mantener la coherencia de las cachés mientras que la serialización del bus mantiene la consistencia. Cada controlador de caché observa e interpreta las transacciones de memoria de los otros controladores para mantener su propio estado interno. Para conseguir un uso eficiente de ancho de banda limitado que proporciona un bus compartido

nos centraremos en protocolos que usan cachés del tipo post-escritura (*write-back*) lo que permite que varios procesadores escriban a diferentes bloques en sus cachés locales de forma simultánea sin ninguna transacción en el bus. En este caso es necesario tener especial cuidado para asegurarnos de que se transmite la suficiente información sobre el bus para poder mantener la coherencia. También comprobaremos que los protocolos proporcionan restricciones suficientes en la ordenación para mantener la serialización de las escrituras y un modelo de consistencia de memoria secuencial.

Recordemos que en un uniprocesador con una caché con post-escritura. Un fallo de escritura en la caché hace que ésta lea todo el bloque de la memoria, actualice

una palabra, y retenga el bloque como *modificado* de tal forma que la actualización en memoria principal ocurre cuando el bloque es reemplazado. En un multiprocesador, este estado de modificado también se usa por parte del protocolo para indicar la pertenencia de forma exclusiva del bloque por parte de la caché. En general, se dice que una caché es *propietaria* de un bloque si debe suministrar los datos ante una petición de ese bloque. Se dice que una caché tiene una copia *exclusiva* de un bloque si es la única caché con una copia válida del bloque (la memoria principal puede o no tener una copia válida). La exclusividad implica que la caché puede modificar el bloque sin notificárselo a nadie. Si una caché no tiene la exclusividad, entonces no puede escribir un

nuevo valor en el bloque antes de poner un transacción en el bus para comunicarse con las otras cachés. Si una caché tiene el bloque en estado modificado, entonces dicho nodo es el propietario y tiene la exclusividad. (La necesidad de distinguir entre propiedad y exclusividad se aclarará muy pronto.)

En un fallo de escritura, una forma especial de transacción llamada *lectura exclusiva* se usa para impedir a las otras cachés la escritura o adquirir una copia del bloque con propiedad exclusiva. Esto hace que el bloque de la caché pase a estado modificado permitiendo la escritura. Varios procesadores no pueden escribir el mismo bloque de forma concurrente, lo que llevaría a valores inconsistentes: Las transacciones de lectura exclusiva generadas por sus escrituras aparecerán de forma secuencial en el bus, así que sólo una de ellas puede obtener la propiedad exclusiva del bloque en un momento dado. Las acciones de coherencia de caché se consiguen mediante estos dos tipos de transacciones: lectura y lectura exclusiva. Eventualmente, cuando un bloque modificado se reemplaza en la caché, los datos se actualizan en la memoria, pero este evento no está producido por una operación de memoria sobre ese bloque y su importancia en el protocolo es incidental. Un bloque que no ha sido modificado no tiene que se escrito en la memoria y puede ser descartado.

También debemos considerar los protocolos basados en la actualización. En este caso, cuando se escribe en una localización compartida, el valor actualizado se transfiere a todos los procesadores que tienen ese bloque en la caché. De esta forma, cuando los procesadores acceden a ese bloque posteriormente, pueden hacerlo desde sus cachés con baja latencia. Las cachés de todos los demás procesadores se actualizan con una única transacción del bus, conservando el ancho de banda. Por contra, con los protocolos basados en la invalidación, en una operación de escritura el estado de la caché de ese bloque de memoria en todos los demás procesadores son declarados inválidos. Las ventajas e inconvenientes de ambos protocolos son complejos y dependen de la carga de trabajo. En general, las estrategias basadas en la invalidación se han mostrado más robustas y se proporcionan como el protocolo básico por la mayoría de los vendedores. Algunos vendedores proporcionan un protocolo de actualización como una opción a ser utilizada para bloques correspondientes a estructuras de datos específicas o páginas.

La tabla 6.2 resume los principales protocolos e implementaciones para mantener la coherencia en sistemas basados en bus. El número de protocolos del tipo de invalidar en escritura es mayor puesto que son los protocolos que se utilizan más.

A continuación veremos los protocolos MSI, MESI (conocido también como Illinois), Write once y Berkeley como ejemplos de protocolos basados en invalidación en escritura, y Dragon y Firefly que se incluyen como ejemplos de protocolos de actualización en escritura.

6.4.1 Protocolo de invalidación de 3 estados (MSI)

El primer protocolo que consideraremos es un protocolo de invalidación básico para cachés post-escritura. Es muy similar al protocolo que fue usado en la serie de multiprocesadores Silicon Graphics 4D. El protocolo usa los tres estados necesarios en cualquier caché post-escritura para distinguir bloques válidos que no han sido modificados (*clean*) de aquellos que han sido modificados (*dirty*). Específicamente, los estados son *inválido* (I), *compartido* (S) y *modificado* (M). El estado de inválido tiene un significado claro. Compartido significa que el bloque está presente en la caché y no ha sido modificado, la memoria principal está actualizada y cero o más cachés adicionales pueden tener también una copia actualizada (compartida). Modificado significa que únicamente este procesador tiene una copia válida del bloque en su caché, la copia en la memoria principal está anticuada y ninguna otra caché puede tener una copia válida del bloque (ni en estado modificado ni compartido). Antes de que un bloque compartido pueda ser

Nombre	Tipo de protocolo	Escritura a memoria	Características	Máquinas
Write once	Write invalidate	Write back después de la primera escritura	Primer protocolo de sondeo	Sequent Symmetry
Synapse N+1	Write invalidate	Write back	Estado explícito donde la memoria es propietaria	Máquinas Synapse; primeras disponibles
Berkeley	Write invalidate	Write back	Estado propietario compartido	Berkeley SPUR
Illinois MESI	Write invalidate	Write back	Estado privado limpio; puede entregar datos desde cualquier caché con copias limpias	SGI Power y series Challenge
Firefly	Write update	Write back si privado, write through si compartido	Memoria actualizada para todos	Actualmente en desuso; SPARCCenter 2000
Dragon	Write update	Write back si privado, write through si compartido		

Tabla 6.2: Protocolos de sondeo para la coherencia de cachés.

escrito y pasar al estado modificado, todas las demás copias potenciales deben de ser invalidadas vía una transacción de bus de lectura exclusiva. Esta transacción sirve para ordenar la escritura al mismo tiempo que causa la invalidaciones y por tanto asegura que la escritura se hace visible a los demás.

El procesador emite dos tipos de peticiones: lecturas (PrRd) y escrituras (PrWr). Las lecturas o escrituras pueden ser a un bloque de memoria que existe en la caché o a uno que no existe. En el último caso, el bloque que esté en la caché en la actualidad será reemplazado por el nuevo bloque, y si el bloque actual está en el estado modificado su contenido será volcado a la memoria principal.

Supongamos que el bus permite las siguientes transacciones:

- Lectura del bus (BusRd): El controlador de la caché pone la dirección en el bus y pide una copia que no piensa modificar. El sistema de memoria (posiblemente otra caché) proporciona el dato. Esta transacción se genera por un PrRd que falla en la caché y el procesador espera como resultado de esta transacción el dato solicitado.
- Lectura exclusiva del bus (BusRdX): El controlador de la caché pone la dirección en el bus y pide una copia exclusiva que piensa modificar. El sistema de memoria (posiblemente otra caché) proporciona el dato. Todas las demás cachés necesitan ser invalidadas. Esta transacción se genera por una PrWr a un bloque que, o no está en la caché o está en la caché en un estado distinto al modificado. Una vez la caché obtiene la copia exclusiva, la escritura puede realizarse en la caché. El procesador

puede solicitar una confirmación como resultado de esta transacción.

- Escritura (BusWB): El controlador de la caché pone la dirección y el contenido para el bloque de la memoria en el bus. La memoria principal se actualiza con el último contenido. Esta transacción la genera el controlador de la caché en una post-escritura; el procesador no conoce este hecho y no espera una respuesta.

La lectura exclusiva del bus es una nueva transacción que puede no existir excepto para la coherencia de la caché. Otro nuevo concepto necesario para soportar los protocolos de post-escritura es que junto con el cambio de estado de cada bloque en la caché, el controlador de la caché puede intervenir en las transacciones del bus y poner el contenido del bloque referenciado en el bus, en vez de permitir que la memoria proporcione el dato. Por supuesto, el controlador de la caché también puede iniciar nuevas transacciones, proporcionar datos para las post-escrituras, o coger los datos proporcionados por el sistema de memoria.

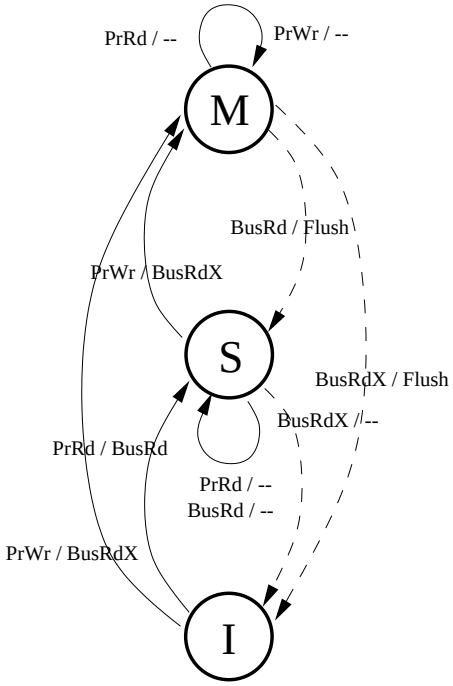


Figura 6.32: Protocolo básico de invalidación con tres estados.

El diagrama de estados que gobierna un bloque en cada caché en este protocolo se muestra en la figura 6.32. Los estados están organizados de tal manera que conforme un estado está más cerca de la parte superior más ligado está el bloque a ese procesador. La lectura de un bloque inválido por parte de un procesador (incluyendo los no presentes) produce una transacción BusRd. El bloque recientemente cargado se promociona desde el estado inválido al estado compartido dependiendo de si otra caché tiene una copia del bloque. Cualquier otra caché con el bloque en el estado compartido observa el BusRd, pero no toma ninguna acción especial, permitiendo que la memoria responda con el dato. Sin embargo, si una caché tiene el bloque en el estado modificado (y únicamente puede haber una) y observa una transacción BusRd en el bus, debe participar en la transacción ya que la copia existente en memoria no es válida. Esta caché envía los datos al bus en lugar de la memoria y degrada su copia del bloque al estado compartido. La memoria

y la caché que pedía el dato son actualizadas. Esto puede conseguirse mediante una transferencia directa entre las cachés a través del bus durante esta transacción BusRd o indicando un error en la transacción BusRd y generando una transacción de escritura para actualizar la memoria. En el último caso, la caché original volvería a realizar su solicitud para obtener el bloque de la memoria. (También es posible hacer que el dato puesto en el bus sea tomado por la caché que realizó la petición pero no por la memoria, dejando la memoria sin actualizar, pero este caso requiere más estados).

Una escritura en un bloque inválido implica cargar el bloque entero y después modificar los bytes que se deseen. En este caso se genera una transacción de lectura exclusiva en el bus, que causa que todas las otras copias del bloque sean invalidadas, asegurando a la caché que realiza la petición la propiedad exclusiva del bloque. El bloque asciende al estado de modificado y después se escribe. Si otra caché realiza una petición posterior de acceso exclusivo, en respuesta a su transacción BusRdX el bloque se degradará al estado de invalido después de poner la copia exclusiva en el bus.

La transacción más interesante ocurre cuando se escribe en un bloque compartido. El tratamiento en este caso es igual al de una escritura en un bloque inválido, usando una transacción de lectura exclusiva en el bus para adquirir la propiedad exclusiva del bloque. Los datos que nos devuelve el bus pueden ser ignorados en este caso ya que ya existen en la caché. De hecho, una optimización común para reducir el tráfico en el bus es introducir una nueva transacción, llamada *actualización del bus* o BusUpgr, para esta situación. Un BusUpgr obtiene la propiedad exclusiva al igual que un BusRdX, invalidando las demás copias, pero no devuelve el bloque al que realizó la petición. Independientemente de la transacción utilizada, el bloque pasa al estado modificado. Posteriores escrituras en el bloque mientras permanece en ese estado no generan nuevas transacciones.

El reemplazamiento de un bloque de la caché lleva a dicho bloque al estado de inválido (no presente) eliminándolo de la caché. Por lo tanto, un reemplazamiento supone que dos bloques cambien de estado en la caché: el que es reemplazado pasa de su estado actual al estado de inválido, el que ocupa su lugar de inválido a su nuevo estado. El último cambio no puede tener lugar antes del primero. Si el bloque a ser reemplazado estaba en el estado de modificado, la transición de reemplazamiento de M a I genera una transacción de post-escritura. Si estaba en el estado inválido o compartido, no es necesario hacer nada.

El resultado de aplicar el protocolo MSI al ejemplo mostrado en la figura 6.30 se puede ver en la figura 6.33.

Acción Proces.	Est. P_1	Est. P_2	Est. P_3	Acc. bus	Datos sumin. por
1. P_1 lee u	S	–	–	BusRd	Memoria
2. P_3 lee u	S	–	S	BusRd	Memoria
3. P_3 escribe u	I	–	M	BusRdX	Memoria
4. P_1 lee u	S	–	S	BusRd	Caché de P_3
5. P_2 lee u	S	S	S	BusRd	Memoria

Figura 6.33: Protocolo de invalidación de 3 estados en acción para las transacciones mostradas en la figura 6.30.

Satisfacción de la coherencia

Dado que pueden existir lecturas y escrituras a cachés locales de forma simultánea, no es obvio que se satisfaga la condición de coherencia y mucho menos la de consistencia secuencial. Examinemos en primer lugar la coherencia. Las transacciones de lectura exclusiva aseguran que la caché en donde se realiza la escritura tiene la única copia válida cuando el bloque se modifica en la caché. El único problema que podría existir es que no todas las escrituras generan transacciones en el bus. Sin embargo, la clave aquí es que entre dos transacciones de bus para un bloque, únicamente un procesador puede realizar las escrituras; el que realizó la transacción de lectura exclusiva más reciente. En la serialización, esta secuencia de escrituras aparecerá en el orden indicado por el programa entre esa transacción y la siguiente transacción para ese bloque. Las lecturas que realice dicho procesador también aparecerán en orden con respecto a las otras escrituras. Para una lectura de otro procesador, existirá al menos una transacción del bus que separará la finalización de la lectura de la finalización de esas escrituras. Esa transacción en el bus asegura que la lectura verá todas

las escrituras en un orden secuencial consistente. Por lo tanto, las lecturas de todos los procesadores ven todas las escrituras en el mismo orden.

Satisfacción de la consistencia secuencial

Para comprobar que se cumple la consistencia secuencial, veamos en primer lugar cuál es la propia definición y cómo puede construirse una consistencia global entrelazando todas las operaciones de memoria. La serialización del bus, de hecho, define un orden total sobre las transacciones de todos los bloques, no sólo aquellas que afectan a un único bloque. Todos los controladores de cachés observan las transacciones de lectura y lectura exclusiva en el mismo orden y realizan las invalidaciones en ese orden. Entre transacciones consecutivas del bus, cada procesador realiza una secuencia de operaciones a la memoria (lecturas y escrituras) en el orden del programa. Por lo tanto, cualquier ejecución de un programa define un orden parcial natural:

> Una operación de memoria M_j es posterior a la operación M_i si (i) las operaciones son emitidas por el mismo procesador y M_j aparece después que M_i en el orden del programa, o (ii) M_j genera una transacción del bus posterior a la operación de memoria M_i.

Este orden parcial puede expresarse gráficamente como se muestra en la figura . Entre transacciones del bus, cualquiera de las secuencias que se pueden formar entrelazando las distintas operaciones da lugar a un orden total consistente. En un segmento entre transacciones del bus, un procesador puede observar las escrituras de otros procesadores, ordenadas por transacciones de bus previas que genera, al mismo tiempo que sus propias escrituras están ordenadas por el propio programa.

También podemos ver cómo se cumple la CS en términos de las condiciones suficientes. La detección de la finalización de la escritura se realiza a través la aparición de una transacción de lectura exclusiva en el bus y la realización de la escritura en la caché. La tercera condición, que proporciona la escritura atómica, se consigue bien porque una lectura (i) causa una transacción en el bus que es posterior a aquella transacción de escritura cuyo valor desea obtener, en cuyo caso la escritura debe haberse completado

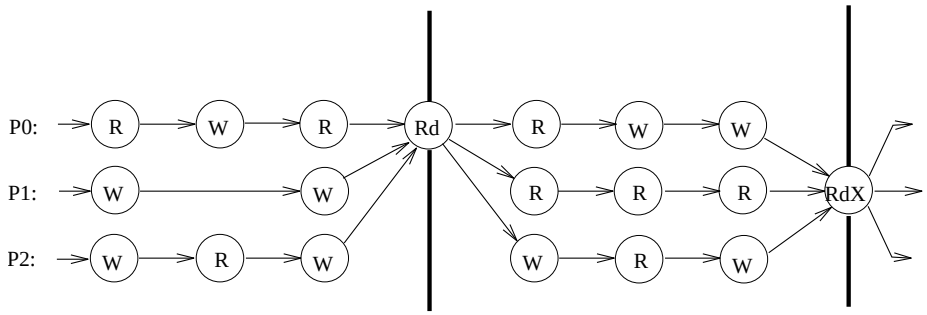


Figura 6.34: Orden parcial en las operaciones de memoria para una ejecución con el protocolo MSI.

de forma global antes de la lectura, o (ii) porque es posterior en el orden del programa como es el caso de una lectura dentro del mismo procesador, o (iii) porque es posterior en el orden del programa del mismo procesador que realizó la escritura, en cuyo caso el procesador ya ha esperado el resultado de una transacción de lectura exclusiva y la escritura se ha completado de forma global. Así, se cumplen todas las condiciones suficientes.

6.4.2 Protocolo de invalidación de 4 estados (MESI)

El protocolo MSI presenta un problema si consideramos una aplicación secuencial en un multiprocesador; este uso de la multiprogramación es de hecho la carga más usual de multiprocesadores de pequeña escala. Cuando el programa lee y modifica un dato, el protocolo MSI debe generar dos transacciones incluso en el caso de que no exista compartición del dato. La primera es una transacción BusRd que obtiene el bloque de la memoria en estado S, y la segunda en una transacción BusRdX (o BusUpgr) que convierte el bloque del estado S al M. Añadiendo un estado que indique que el bloque es la única copia (exclusiva) pero que no está modificado y cargando el bloque en ese estado, podríamos evitarnos la última transacción ya que el estado indica que ningún otro procesador tiene el bloque en caché. Este nuevo estado indica un nivel intermedio entre compartido y modificado. Al ser exclusivo es posible realizar una escritura o pasar al estado modificado sin ninguna transacción del bus, al contrario que en el caso de estar en el estado compartido; pero no implica pertenencia, así que al contrario que en el estado modificado la caché no necesita responder al observar una petición de dicho bloque (la memoria tiene una copia válida). Variantes de este protocolo MESI se usan en muchos de los microprocesadores modernos, incluyendo el Pentium, PowerPC 601 y el MIPS R4400 usado en los multiprocesadores Silicon Graphics Challenge. A este protocolo también se le conoce como protocolo Illinois por haber sido publicado por investigadores de la Universidad de Illinois en 1984.

El protocolo MESI consiste en cuatro estados: *modificado* (M), *exclusivo* (E), *compartido* (S) e *inválido* (I). I y M tienen la misma semántica que antes. El estado exclusivo, E, significa que únicamente una caché (esta caché) tiene una copia del bloque, y no ha sido modificado (es decir, la memoria principal está actualizada). El estado S significa que potencialmente dos o más procesadores tienen este bloque en su caché en un estado no modificado.

Cuando un procesador lee por primera vez un bloque, si existe una copia válida

en otra caché entra en la caché del procesador con el estado S. Sin embargo, si no existe otra copia en ese instante (por ejemplo, en una aplicación secuencial), entra con el estado E. Cuando ese bloque se actualiza por medio de una escritura puede pasar directamente del estado E a M sin generar otra transacción en el bus, dado que no existe otra copia. Si mientras tanto otra caché ha obtenido una copia del bloque, el estado del bloque deberá pasar de E a S. Este protocolo necesita que el bus proporcione una señal adicional (S) que esté disponible para que los controladores puedan determinar en una transacción BusRd si existe otra caché que tenga el mismo bloque. Durante la fase en la que se indica en el bus la dirección del bloque al que se quiere acceder, todas las cachés determinan si tienen el bloque que se solicita y, si eso ocurre, activan la señal compartida. Esta es una línea cableada como un OR, con lo que el controlador que realiza la petición puede observar si existe otro procesador que tenga en su caché el bloque de memoria referenciado y decidir si la carga del bloque se debe realizar en el estado S o E.

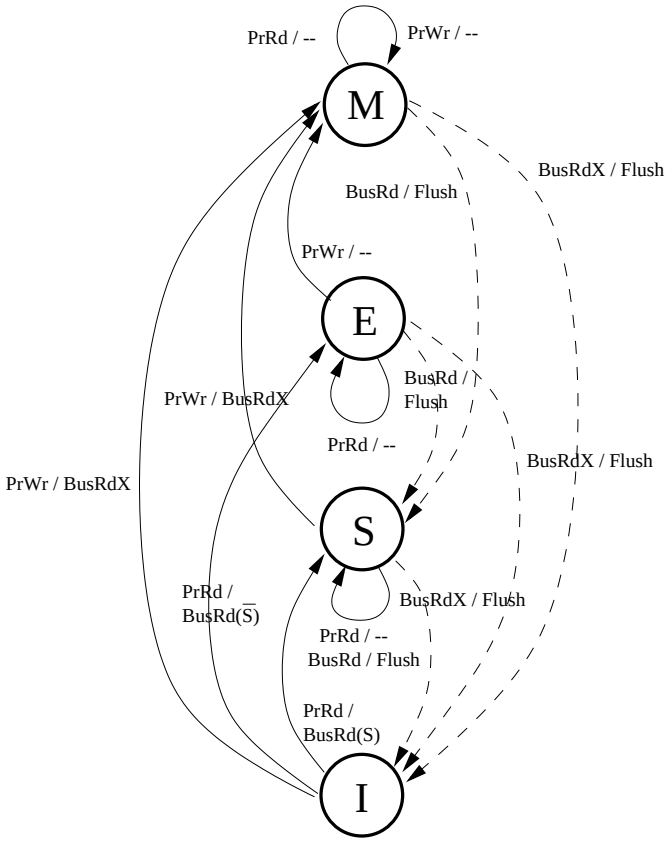


Figura 6.35: Diagrama de transición de estados para el protocolo Illinois MESI.

El diagrama de transición de estados para este protocolo se muestra en la figura 6.35. La notación BusRd(S) significa que cuando la transacción de lectura en el bus ocurre, se activa la señal S, y BusRd($\overline{S}$) significa que S no está activada. Una transacción BusRd significa que no nos importa el valor de S en esa transacción. Una escritura a un bloque en cualquier estado hace que este pase al estado M, pero si estaba en el estado E no se necesita una transacción del bus. La aparición de un BusRd hace que el bloque pase del estado E al S, dado que existe otra copia del bloque. Como ocurría anteriormente, la aparición de una transacción BusRd hace que un bloque que

estuviese en el estado M pase al estado S y dicho bloque sea puesto en el bus. Tal como ocurría en el protocolo MSI es posible hacer que dicho bloque no sea actualizado en la memoria principal, pero serían necesarios nuevos estados. Obsérvese que es posible que un bloque esté en el estado S incluso sin existir otras copias, dado que estas copias pueden ser reemplazadas sin que las demás cachés sean notificadas. Los argumentos para comprobar que el protocolo mantiene la consistencia secuencial y la coherencia de las cachés es el mismo que en el caso del protocolo MSI.

El nombre del protocolo (MESI) viene de los nombres de los estados que puede tomar una línea: *modificada*, *exclusiva*, *compartida* e *inválida* que en inglés significan *Modified*, *Exclusive*, *Shared* y *Invalid*, y lo cierto es que este nombre se usa más que el de Illinois.

Una línea que es propiedad de cualquier caché puede estar como *exclusiva* o *modificada*. Las líneas que están en el estado *compartida* no son propiedad de nadie en concreto. La aproximación MESI permite la determinación de si una línea es compartida o no cuando se carga. Una línea modificada es enviada directamente al peticionario desde la caché que la contiene. Esto evita la necesidad de enviar comandos de invalidación para todas aquellas líneas que no estaban compartidas.

Veamos las transiciones según las operaciones realizadas sobre las cachés:

Fallo de lectura: La línea se coge de la caché que la tenga. Si ninguna la tiene, se coge de la memoria principal. Si la línea que se coge está *modificada* entonces se escribe esta línea también a la memoria al mismo tiempo. Si es compartida, la caché con mayor prioridad es la que entrega la línea. La caché que entrega la línea, y aquellas que la contienen, pasan al estado de *compartida*. Sin embargo, si quien da la línea es la memoria, entonces la caché que lee pone su línea en *exclusiva* y se hace propietaria de ella.

Fallo de escritura: En este caso la línea se coge de la caché que la tenga. Si ninguna la tiene, se coge de la memoria principal. Si la línea que se coge está *modificada* entonces se escribe esta línea también a la memoria al mismo tiempo. Si es compartida, la caché con mayor prioridad es la que entrega la línea. La línea leída se actualiza y se pone en estado *modificada*. El resto de líneas pasan al estado *invalidas*. Hay que hacer notar que este protocolo sólo permite un único escribiente, es decir, una línea modificada sólo está presente en una única caché.

Acierto de escritura: Si la caché peticionaria es la propietaria (línea *modificada* o *exclusiva*), se actualiza la línea en la caché sin más. Si la línea está compartida por otras cachés (*compartida*), se actualiza la línea después de haber invalidado el resto de líneas. Una vez actualizada la línea pasa a ser *modificada* en cualquiera de los casos anteriores.

6.4.3 Write once

A veces se le conoce también como *write invalidate* directamente, ya que es el primero que se pensó de este tipo y también es el primero de los protocolos de sondeo. En ocasiones se le conoce como el protocolo de Goodman puesto que fue propuesto por James Goodman en 1983.

Este protocolo de coherencia se puede describir mediante un grafo de transiciones de 4 estados tal y como se muestra en la figura 6.36. Los 4 estados son los siguientes:

Válida: La línea de caché, que es consistente con la copia en memoria, ha sido leída

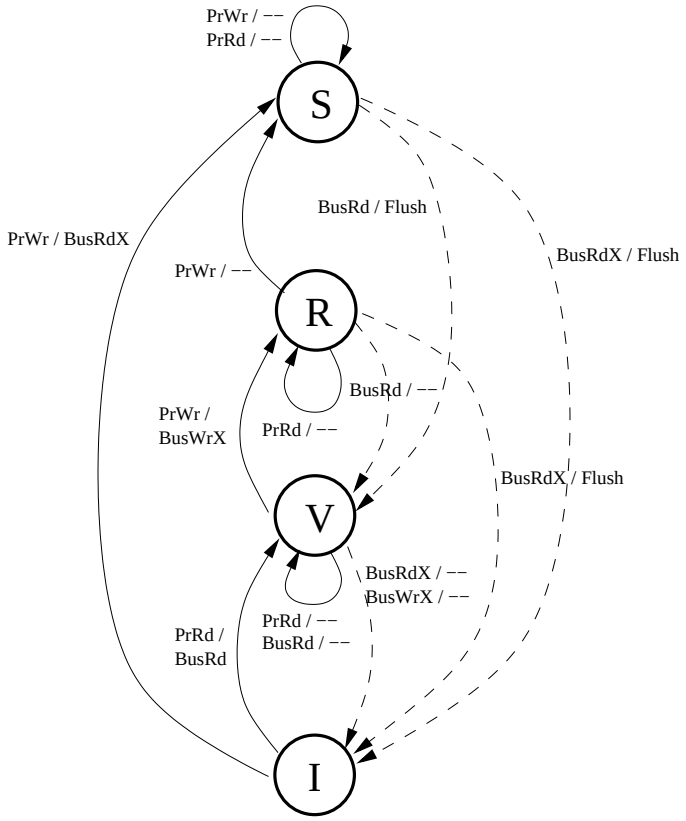


Figura 6.36: Diagrama de transición del protocolo Write once.

de la memoria principal y no ha sido modificada.

Inválida: La línea no se encuentra en la caché o no es consistente con la copia en memoria.

Reservada: Los datos han sido escritos una única vez desde que se leyó de la memoria compartida. La línea de caché es consistente con la copia en memoria que es la única otra copia.

Sucia: La línea de caché ha sido escrita más de una vez, y la copia de la caché es la única en el sistema (por lo tanto inconsistente con el resto de copias).

Cada línea de caché tiene dos bits extra donde se guarda el estado de esa línea. Dependiendo de la bibliografía cambia el nombre de los estados. Así a la línea *válida* se la conoce también como *consistente múltiple*, a la *reservada* como *consistente única*, y la *sucia* como *inconsistente única*, la *válida* se queda con el mismo nombre.

En realidad este protocolo de *escribir una vez* está basado en un protocolo más sencillo con sólo tres estados para las copias que son *válida*, *inválida* y *sucia*. El estado *reservada* se ha añadido para el caso en que se modifique una línea una sola vez, de manera que se aprovecha mejor el ancho de banda del bus.

Para mantener la consistencia el protocolo requiere dos conjuntos de comandos. Las líneas sólidas de la figura 6.36 se corresponden con los comandos emitidos por el procesador local y son *fallo de lectura*, *acierto de escritura* y *fallo de escritura*.

Siempre que se produce un fallo de lectura la línea entra en el estado de *válida*. El primer acierto de escritura lleva a la línea a su estado de *reservada*. Los aciertos de

escritura posteriores cambian el estado a *sucia*.

Las líneas discontinuas se corresponden con los comandos de invalidación emitidos por el resto de procesadores a través del bus de sondeo compartido. El comando *invalidar en lectura* lee una línea o bloque e invalida el resto de copias. El comando *invalidar en escritura* invalida todas las copias de una línea. El comando *lectura de bus* se corresponde con una lectura normal a memoria por otro procesador.

Con estas consideraciones, los eventos y acciones que tienen lugar en el sistema de memoria son:

Fallo de lectura: Cuando un procesador quiere leer un bloque que no está en la caché se produce un *fallo de lectura*. Entonces se inicia una operación de *lectura en el bus*. Si no hay copias de esta línea en otras cachés entonces se trae la línea de la memoria principal, pasando el estado a *exclusiva* puesto que es la única caché con esa línea. Si hay alguna copia *exclusiva* en alguna otra caché, entonces se coge la copia de la memoria o de esa caché, pasando ambas copias al estado de *válidas*. Si existe alguna copia *sucia* en otra caché, esa caché inhibirá la memoria principal y mandará esa copia a la caché que hizo la petición así como a la memoria principal, ambas copias resultantes pasarán al estado de *válidas*.

Acierto de escritura: Si la copia está *sucia* o *reservada*, la escritura se puede llevar a
cabo localmente y el nuevo estado es *sucia*. Si el estado es *válida* entonces se manda un comando de *invalidar en escritura* a todas las cachés invalidando sus copias. A la memoria compartida se escribe al tiempo que a la propia caché, pasando el estado a *reservada*. De esta manera nos aseguramos de que sólo hay una copia que es *reservada* o *sucia* de una determinada línea en todas las cachés.

Fallo en escritura: Cuando un procesador falla al escribir en la caché local, la copia tiene que venir de la memoria principal o de la caché que esté sucia. Esto se realiza enviando un *invalidar en lectura* a todas las cachés lo que invalidará todas las cachés. La copia local es actualizada terminando en un estado de *sucia* puesto que no se ha actualizado la memoria después de la escritura.

Acierto de lectura: Nunca hay una transición en este caso, se lee de la caché local y ya está.

Cambio de línea: Si una copia está *sucia* debe escribirse en algún momento a la memoria principal mediante un cambio de línea (lo cual ocurrirá cuando alguna otra caché lea de esa línea). Si la copia esta *limpia*, es decir, es *válida*, *exclusiva* o *inválida*, no se realiza ningún cambio.

Para poder implementar este protocolo en un bus, son necesarias unas líneas adicionales para inhibir la memoria principal cuando se produce un fallo de lectura y hay copias sucias. Muchos de los buses estándar no incluyen estas líneas por lo que resulta complicado implantar protocolos de coherencia de caché, a pesar de que estos buses incluyen multiprocesadores.

6.4.4 Berkeley

El protocolo Berkeley, también llamado Berkeley-SPUR, usa la idea de propietario de la línea de caché. En cualquier instante una línea de caché sólo puede ser propiedad de una sola de las cachés, y si ninguna tiene esa línea entonces se encuentra en memoria principal. Hay cuatro estados para realizar esto: *invalida*, *sólo lectura*, *sucia compartida*

y *sucia privada*. Cuando una línea está compartida, sólo el propietario tiene esa línea en el estado *sucia compartida*; todos los demás deberán tener esa línea como *sólo lectura*. Por lo tanto, una línea de caché sólo puede estar en el estado *sucia compartida* o *sucia privada* en una única caché, que será la propietaria de esa línea.

La figura 6.37 muestra el diagrama de estados correspondiente a este protocolo; a la izquierda se muestra el diagrama para las operaciones que realiza el procesador y a la derecha el diagrama para las órdenes que vienen a través del bus realizadas por el resto de procesadores.

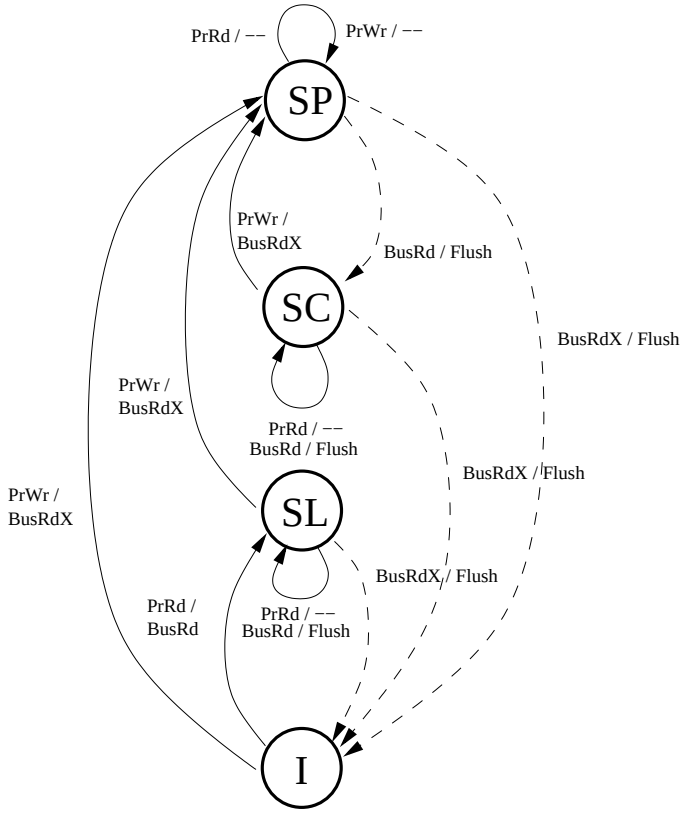


Figura 6.37: Diagramas de transición del protocolo Berkeley.

Veamos a continuación las acciones a realizar dependiendo de las operaciones de la CPU:

Fallo de lectura: Cuando ocurre un *fallo de lectura*, la caché que falló pide los datos al propietario de la línea, que puede ser la memoria (si la línea es *inválida* o de *sólo lectura* en el resto de cachés) o la caché que sea propietaria de la línea (que tenga cualquiera de los estados sucios *compartida* o *privada*), pasando el estado de la línea a *sólo lectura*. Si la línea que se pide se encuentra en el estado de *sólo lectura* en el resto de las cachés, entonces el dato lo coge de la memoria y el estado que se le asigna es el de *sólo lectura*. Si el propietario de la línea tenía el estado de *sucia privada* tendrá que cambiarlo al de *sucia compartida* puesto que a partir de ese momento existen más copias de esa línea en otras cachés.

Fallo de escritura: En un *fallo de escritura*, la línea viene directamente del propietario (memoria u otra caché). El resto de cachés con copias de la línea, incluida la inicialmente propietaria, deben invalidar sus copias. La caché que quería escribir pone el estado de esta línea a *sucia privada* y pasa a ser el propietario.

Acierto de escritura: La caché que escribe invalida cualquier otra copia de esa línea y actualiza la memoria. El nuevo estado de la línea se pone a *sucia privada* da igual el estado que tuviera esa línea. Sólo en el caso en que la línea fuera *sucia privada* no es necesaria ni la invalidación ni la actualización a memoria principal, con lo que la escritura se realiza local a la caché.

6.4.5 Protocolo de actualización de 4 estados (Dragon)

A continuación estudiaremos un protocolo de actualización básico para cachés de post-escritura. Esta es una versión mejorada del protocolo usado por los multiprocesadores SUN SparcServer. Este protocolo fue inicialmente propuesto por los investigadores de Xerox PARC para su sistema multiprocesador Dragon.

El protocolo Dragon consta de 4 estados: *exclusivo* (E), *compartido* (SC), *compartido modificado* (SM) y *modificado* (M). El estado exclusivo significa que sólo existe una caché (esta caché) con una copia del bloque y que dicho bloque no ha sido modificado. La causa de añadir el estado E en Dragon es la misma que en el protocolo MESI. SC significa que potencialmente dos o más procesadores (incluyendo esta caché) tienen este bloque en su caché y que la memoria principal puede estar o no actualizada. SM significa que potencialmente dos o más procesadores tienen este bloque en su caché, la memoria principal no está actualizada y este procesador es el responsable de actualizar la memoria principal cuando este bloque sea reemplazado en la caché. Un bloque sólo puede estar en el estado SM en una única caché en un momento dado. Sin embargo, es posible que una caché tenga el bloque en estado SM mientras que en las otras aparezca en el estado SC. M significa, como anteriormente, que el bloque ha sido modificado únicamente en esta memoria, la memoria principal tiene un valor anticuado, y este procesador es el responsable de actualizar la memoria principal

cuando el bloque sea reemplazado. Obsérvese que no existe un estado explícito de bloque inválido (I) como en los protocolos anteriores. Esto es debido a que el protocolo Dragon es un protocolo basado en la actualización; el protocolo siempre mantiene los bloque de la caché actualizados, así que siempre es posible utilizar los datos existentes en la caché.

Las peticiones del procesador, las transacciones del bus y las acciones a realizar por el protocolo son similares a las vistas en el protocolo Illinois MESI. El procesador todavía envía peticiones de lectura (PrRd) y escritura (PrWr). Sin embargo, dado que no tenemos un estado de invalidación en el protocolo, para especificar las acciones a realizar cuando un bloque de memoria se pide por primera vez, se han añadido dos tipos más de peticiones: *lectura con fallo de caché* (PrRdMiss) y *escritura con fallo de caché* (PrWrMiss). Para las transacciones de bus, tenemos *lectura en bus* (BusRd), *actualización en bus (BusUpd)* y *escritura en bus (*Bus*WB*). Las transacciones BusRd y BusWB tienen la semántica usual vista en los protocolos anteriores. BusUpd es una nueva transacción que toma como parámetro la palabra modificada por el procesador y la envía a todos los procesadores a través del bus de tal manera que se actualicen las cachés de los demás procesadores. Enviando únicamente la palabra modificada en lugar de todo el bloque, se espera una utilización más eficiente del ancho del banda del bus. Al igual que en el caso del protocolo MESI aparece la señal S para soportar el estado E.

La figura 6.38 muestra el diagrama de transición de estados para este protocolo. A continuación se exponen las acciones a realizar cuando un procesador incurre en un

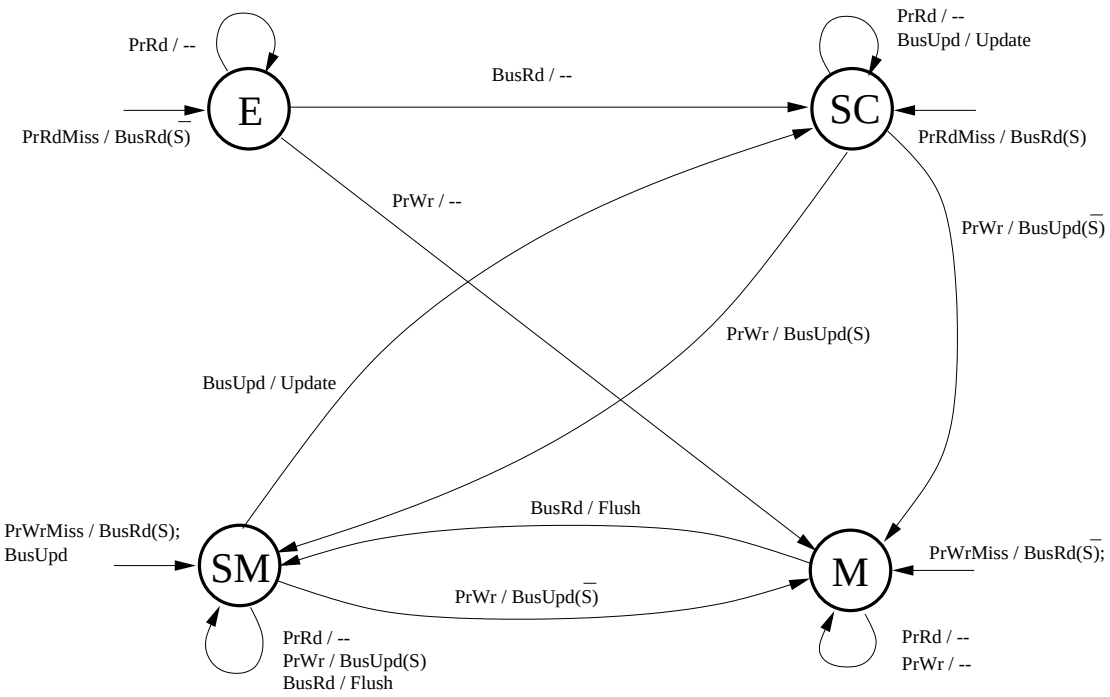


Figura 6.38: Diagrama de transición de estados del protocolo de actualización Dragon.

fallo de caché en una lectura, un acierto de caché en una escritura o un fallo de caché en una escritura (en el caso de un acierto de caché en una lectura no hay que realizar ninguna acción).

Fallo en una lectura: Se genera una transacción BusRd. Dependiendo del estado de la señal compartida (S), el bloque se carga en el estado E o SC en la caché local. Más concretamente, si el bloque está en el estado M o SM en una de las otras cachés, esa caché activará la línea S y proporcionará el último valor para ese bloque en el bus, y el bloque se cargará en la caché local en el estado SC (también se actualiza la memoria principal; si no quisiéramos hacerlo necesitaríamos estados adicionales). Si la otra caché lo tiene en estado M, cambiará al estado SM. Si ninguna otra caché tiene una copia, la línea S no se activará y el dato será proporcionado por la memoria principal y cargado en la caché local en el estado E.

Escritura: Si el bloque está en el estado SM o M en la caché local, no es necesario realizar ninguna acción. Si el bloque está en el estado E en la caché local, el bloque cambia internamente al estado M, no siendo necesaria ninguna acción adicional. Sin embargo, si el bloque está en el estado SC, se genera una transacción BusUpd. Si alguna otra caché tiene una copia del dato, actualizan sus copias y cambian su estado a SC. La caché local también actualiza su copia del bloque y cambia su estado a SM. Si no existe otra copia del dato, la señal S permanecerá inactiva y la copia local se actualizará cambiando al estado M. Finalmente, si en una escritura el bloque no está presente en la caché, la escritura se trata como una transacción del tipo fallo en la lectura seguida de una transacción de escritura. Por lo tanto, primero se genera un BusRd y si el bloque se también se encuentra en otras cachés un BusUpd.

Reemplazo: En el caso de reemplazo de un bloque (los arcos no se muestran en la figura), el bloque se escribe en la memoria principal utilizando una transacción del bus únicamente si se encuentra en el estado M o SM. Si está en el estado SC, existirá otra caché con ese bloque en estado SM, o nadie lo tiene en cuyo caso el bloque es válido en memoria principal.

6.4.6 Firefly

Este es el único protocolo del tipo *actualizar en escritura* que se va a ver puesto que prácticamente no se usan ninguno de este tipo.

El Firefly tiene 4 estados aunque en realidad sólo utiliza tres que son: *lectura privada*, *lectura compartida* y *sucia privada*. El estado *inválida* se utiliza sólo para definir la condición inicial de una línea de caché. El Firefly utiliza el esquema de actualización en vez del de invalidación, es decir, las escrituras a la caché son vistas por todos y se escribe a la memoria principal. El resto de cachés que comparten la línea, fisgan el bus actualizando sus copias. Por lo tanto, ninguna línea de caché será inválida después de ser cargada. Hay una línea especial del bus, la *LíneaCompartida*, que se activa para indicar que al menos otra caché comparte la línea.

Veamos las transiciones a realizar según los casos:

Fallo de lectura: Si otra caché tiene una copia de la línea, ésta la entrega a la caché peticionaria, activando la línea *LíneaCompartida*. Si el estado de la línea era *sucia privada* previamente, entonces debe actualizarse a la memoria principal. Todas las cachés que comparten dicha línea ponen su estado a *lectura compartida*. Si no hay cachés con esta línea, entonces se coge de la memoria y se pone el estado de *lectura privada*.

Fallo de escritura: La línea se coge de otra caché o de la memoria. Cuando la línea viene de otra caché, ésta activa la línea *LíneaCompartida* que es reconocida por la caché peticionaria. Por lo tanto, la línea se pone en estado de *lectura compartida* y la escritura se propaga a todas las cachés y a memoria, que actualizan sus copias. Si la línea viene de la memoria, se carga la línea como *sucia privada* y se escribe sin propagar la línea con los nuevos datos.

Acierto en escritura: Si la línea está *privada sucia* o *escritura privada*, se escribe a la caché sin más. En el último caso se cambia el estado a *privada sucia*. Si el estado es el de *lectura compartida* entonces además de a la caché debe escribir a la memoria. El resto de cachés que comparten la línea capturan los datos del bus actualizando así sus líneas. Además, estas cachés activan la línea *LíneaCompartida*. Esto es necesario para que así la caché que está escribiendo pueda saber si la línea está compartida o no. Si la línea no está compartida, se puede evitar la escritura "a todos" (*broadcast*), en cuyo caso se actualiza como *lectura privada*, de otra manera se carga como *lectura compartida*.

6.4.7 Rendimiento de los protocolos de sondeo

A la hora de determinar el rendimiento de un multiprocesador con arquitecturas como las vistas hasta ahora, existen varios fenómenos que se combinan. En particular, el rendimiento total de la caché es una combinación del comportamiento de tráfico causado por fallos de caché en un uniprocesador y el trafico causado por la comunicación entre

cachés, lo que puede dar lugar a invalidaciones y posteriores fallos de caché. El cambio del número de procesadores, tamaño de la caché y tamaño del bloque pueden afectar a estos dos componentes de diferente manera, llevando al sistema a un comportamiento que es una combinación de los dos efectos.

Las diferencias de rendimiento entre los protocolos de invalidación y actualización se basan en las tres características siguientes:

1. Varias escrituras a la misma palabra sin lecturas intermedias precisan varias escrituras a todos en el protocolo de actualización, pero sólo una escritura de invalidación en el de invalidación.
2. Con líneas de caché de varias palabras, cada palabra escrita en la línea precisa de una escritura a todos en el protocolo de actualización, mientras que sólo la primera escritura de cualquier palabra en la línea es necesaria en el de invalidación. Los protocolos de invalidación trabajan sobre bloques de caché, mientras que en el de actualización deben trabajar sobre palabras para aumentar la eficiencia.
3. El retraso entre la escritura de una palabra en un procesador y la lectura de ese valor escrito por otro procesador es habitualmente menor en un multiprocesador de actualización, debido a que los datos escritos son actualizados de forma inmediata en la caché del que lee (si es que el procesador que lee ya tenía copia de esa línea). Por comparación, en un protocolo de invalidación, el que va a leer se invalida primero, luego cuando llega la operación de lectura debe esperar a que esté disponible.

Como el ancho de banda del bus y la memoria es lo que más se demanda en un multiprocesador basado en bus, los protocolos de invalidación son los que más se han utilizado en estos casos. Los protocolos de actualización también causan problemas según los modelos de consistencia, reduciendo las ganancias de rendimiento potenciales mostradas en el punto 3 anterior. En diseños con pocos procesadores (2–4) donde los procesadores están fuertemente acoplados, la mayor demanda de ancho de banda de los protocolos de actualización puede ser aceptable. Sin embargo, dadas las tendencias de mayor rendimiento de los procesadores y la demanda consecuente en ancho de banda, hacen que los protocolos de actualización no se usen.

En un sistema que requiera mucha migración de procesos o sincronización, cualquier protocolo de invalidación va a funcionar mejor que los de actualización. Sin embargo, un fallo en la caché puede producirse por la invalidación iniciada por otro procesador previamente al acceso a la caché. Estos fallos por invalidación pueden incrementar el tráfico en el bus y deben ser reducidas.

Varias simulaciones han mostrado que el tráfico en el bus en un multiprocesador puede incrementarse cuando el tamaño de la línea se hace grande. El protocolo de invalidación facilita la implementación de primitivas de sincronización. Típicamente, la media de copias inválidas es bastante pequeña, como 2 o 3 en pequeños multiprocesadores.

El protocolo de actualización requiere la capacidad de realizar transacciones de escritura "a todos" (*broadcast*). Este protocolo puede evitar el efecto "ping-pong" de los datos compartidos entre varias cachés. Reduciendo la cantidad de líneas compartidas se consigue una reducción del tráfico en el bus en un multiprocesador de este tipo. Sin embargo, este protocolo no puede ser usado con varias escrituras seguidas largas.

6.5 Esquemas de coherencia basados en directorio

Los protocolos de sondeo vistos con anterioridad precisan de arquitecturas que tengan la facilidad de acceso "a todos" a un tiempo; una arquitectura basada en bus es un buen ejemplo. El problema del bus es que no es una arquitectura que pueda acomodar un número elevado de procesadores debido a la limitación del ancho de banda. Para multiprocesadores con un número elevado de procesadores se utilizan otro tipo de interconexiones como las vistas en en el capítulo 5 como las mallas, hipercubos, etc. Sin embargo, estas redes más complejas no poseen la capacidad de que cualquier nodo pueda espiar lo que hace el resto. Para solucionar esto lo que se hace es introducir protocolos de coherencia basados en un directorio al cual todos los procesadores y cachés tienen acceso.

6.5.1 Protocolos basados en directorio

Veremos a continuación que los protocolos basados en directorio se pueden dividir en los que tienen el directorio centralizado y los que lo tienen distribuido. En ambos grupos se permite que existan varias copias compartidas de la misma línea de caché para mejorar el rendimiento del multiprocesador sin incrementar demasiado el tráfico en la red.

Directorio centralizado: Fue el primer esquema propuesto (1976). Consiste en un único directorio o tabla centralizada donde se guarda información sobre el lugar donde se encuentra cada copia de la caché. Este directorio centralizado es normalmente bastante grande por lo que la búsqueda se realiza de forma asociativa. La competencia por el acceso al directorio (contención), así como los largos tiempos de búsqueda, son alguno de los inconvenientes de este esquema. Los protocolos de directorio centralizado suelen ser o de *mapeado completo* o con *mapeado limitado* que son dos protocolos que se explican más adelante.

Directorio distribuido: Dos años después se propuso otro protocolo basado en la distribución del directorio entre las diferentes cachés. En el directorio se guarda el estado de la caché así como su presencia. El estado es local, pero la presencia indica qué cachés tienen una copia del bloque. También hay dos protocolos típicos de mapeado distribuido que son los protocolos con *directorios jerárquicos* y los protocolos con *directorios encadenados*.

En ocasiones la noción de directorio centralizado o distribuido crea confusión; si nuestro modelo de sistema tiene la memoria dividida en trozos, cada uno asociado a un nodo, entonces el directorio centralizado está en realidad distribuido entre las memorias. En este caso el directorio se encuentra tan distribuido como en un protocolo de directorio distribuido. La diferencia es que en el directorio centralizado, se accede a una porción del directorio basándonos en su localización por su dirección física.

En el caso del directorio distribuido, se accede a un directorio particular porque ese nodo es propietario de la línea de caché.

Veamos a continuación los diferentes protocolos, de uno y otro tipo, más frecuentemente utilizados.

6.5.2 Protocolo de mapeado completo

El protocolo de mapeado completo mantiene un directorio en el cual cada entrada tiene un bit, llamado *bit de presencia*, por cada una de las cachés del sistema. El bit de presencia se utiliza para especificar la presencia en las cachés de copias del bloque de memoria. Cada bit determina si la copia del bloque está presente en la correspondiente caché. Por ejemplo, en la figura 6.39 las cachés de los procesadores P_a y P_c contienen una copia del bloque de datos x, pero la caché del procesador P_b no. Aparte, cada entrada del directorio tiene un bit llamado *bit de inconsistencia única*. Cuando este bit está activado, sólo uno de los bits de presencia está a uno, es decir, sólo existe una caché con la copia de ese bloque o línea, con lo que sólo esa caché tiene permiso para actualizar la línea.

Cada caché por su parte tiene dos bits en cada entrada de la caché. El primero es el *bit de validación* (v en la figura 6.39), e indica si la copia es válida o no. Si el bit es cero, indica que la copia no es válida, es decir, la copia se puede quitar de la caché. El otro bit, llamado *bit de privacidad* (p en la figura 6.39), sirve para indicar si la copia tiene permiso de escritura, es decir, cuando este bit es uno entonces es la única copia que existe de esta línea en las cachés y por tanto tiene permiso para escribir.

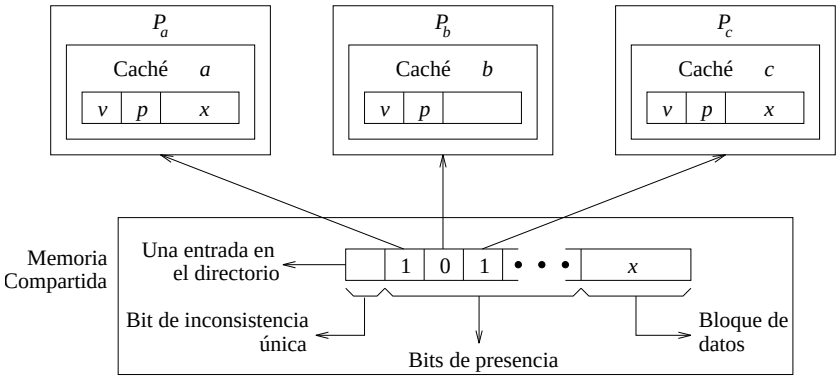


Figura 6.39: Directorio en el protocolo de mapeado completo.

Veamos qué acciones hay que realizar con este protocolo antes los fallos de escritura y lectura, así como los aciertos de escritura. Naturalmente no hay que hacer nada en especial frente a un acierto de lectura.

Fallo de lectura: Supongamos que la caché c envía una petición de fallo de lectura a la memoria. Si el bit de inconsistencia simple está activado, la memoria envía una petición de actualización a la caché que tenga el bit privado activo, o sea, a la única caché que tiene esa línea. La caché devuelve el último contenido del bloque a la memoria y desactiva su bit de privado. El bit de inconsistencia simple del bloque en el directorio central es también desactivado. La memoria activa el bit de presencia correspondiente a la caché c y envía una copia del bloque a c. Una vez la caché c recibe la copia, activa el bit de valido y desactiva el de privado.
Si el bit de inconsistencia simple estaba desactivado, trae el bloque de la memoria activando el de válido y desactivando el de privado como en el caso anterior.

Fallo de escritura: Supongamos que la caché c envía una petición de fallo de escritura a la memoria. La memoria envía peticiones de invalidación a todas las cachés que tienen copias de ese bloque poniendo a cero sus bits de presencia. Las cachés aludidas invalidan la línea poniendo a cero el bit de línea válida enviado mensajes

de reconocimiento a la memoria principal. Durante este proceso, si hay alguna caché (excepto la c) con una copia del bloque y con el bit de privado activado, la memoria se actualiza a sí misma con el contenido del bloque de esta caché. Una vez la memoria recibe todos los reconocimientos, activa el bit de presencia de c y manda la copia a la caché c. El bit de inconsistencia simple se pone a uno, puesto que sólo hay una copia. Una vez la caché recibe la copia, se modifica, y los bits de válido y privado se ponen a uno.

Acierto de escritura: Si el bit de privado es 0, c envía una petición de privacidad a la memoria. La memoria invalida todas las cachés que tienen copia del bloque (similar al caso de fallo de escritura), y luego pone el bit de inconsistencia simple a uno enviando un mensaje de reconocimiento a c. En el momento c recibe el reconocimiento de que las copias han sido invalidadas, modifica el bloque de caché y pone el bit de privado a uno.
Si el bit de privado era 1, entonces escribe sin más puesto que es la única caché con copia de esa línea.

El principal problema del mapeado completo es que el tamaño del directorio es muy grande. Tal y como hemos visto, este tamaño se puede calcular como:

$$\text{Tamaño del directorio} = (1 + \text{Número de cachés}) \times \text{Número de líneas de memoria}$$

Es decir, el tamaño del directorio es proporcional al número de procesadores multiplicado por el tamaño de la memoria. Como la memoria viene a ser habitualmente proporcional al número de procesadores (lo cual es estrictamente cierto en multiprocesadores con memoria compartida distribuida), resulta que el tamaño del directorio es directamente proporcional al cuadrado del número de procesadores, disparándose por tanto cuando el número de procesadores es elevado. Si además las líneas de caché son propiedad de sólo unos pocos procesadores, se ve que este esquema es muy ineficiente.

A continuación veremos una forma de aliviar estos problemas, y veremos también más adelante, en los protocolos de directorio distribuido, otras formas de reducir el directorio para que sea fácilmente escalable.

6.5.3 Protocolo de directorio limitado

Para evitar el crecimiento cuadrático del mapeado completo es posible restringir el número de copias de caché activas de forma simultánea de un bloque. De esta forma se limita el crecimiento del directorio hacia un valor constante.

Un protocolo general basado en directorio se puede clasificar usando la notación Dir_iX, donde Dir indica directorio, i es el número de punteros

en el directorio, y X indica el número de copias que puede haber en el sistema que necesitan tener una referencia centralizada. En el caso de redes sin posibilidad de establecer protocolos de sondeo entre procesadores o grupos de procesadores, la X vale NB donde N es el número de procesadores y B el número de bloques de la memoria de cada procesador.

Con esta nomenclatura, el protocolo basado en directorio de mapeado completo se representaría como Dir_NNB. En cambio, un protocolo basado en directorio limitado, donde $i < N$, se representaría como Dir_iNB.

El protocolo de coherencia para el protocolo de directorio limitado es exactamente igual que el visto para el mapeado completo, con la excepción del caso en que el número de peticiones de una línea en particular sea mayor que i tal y como se explica a continuación.

Supongamos que tenemos el directorio limitado a tan solo 2 punteros, que es lo que vendría a sustituir a los bits de presencia. Supongamos que ambos punteros están ocupados apuntando a copias válidas en dos cachés diferentes. Supongamos ahora que una tercera caché quiere leer en esa misma línea. Como no hay más punteros disponibles se produce lo que se llama un *desalojo* (*eviction*), que consiste en que la memoria invalida una de las cachés que utilizaban un puntero asignándole ese puntero que queda libre a la nueva caché.

Normalmente los procesadores suelen desarrollar su actividad local en zonas de memoria separadas de otros procesadores, eso quiere decir que con unos pocos punteros, incluso uno o dos, se puede mantener la actividad en el sistema sin que se produzcan *desalojos* y sin aumentar por tanto las latencias.

Los punteros son codificaciones binarias para identificar a cada procesador, por lo que en un sistema con N procesadores, son necesarios $\log_2 N$ bits para cada puntero. Con esto, el tamaño del directorio será:

$$\text{Tamaño del directorio} = (1 + \text{Punteros} \times \log_2 N) \times \text{Bloques de memoria}$$

Con las mismas suposiciones que para el mapeado completo, es fácil comprobar que el crecimiento del directorio es proporcional a $N \log_2 N$ que es sensiblemente inferior al de N^2 del protocolo con mapeado completo.

Este protocolo se considera *escalable*, con respecto a la sobrecarga de memoria, puesto que los recursos necesarios para su implementación crecen aproximadamente de forma lineal con el número de procesadores en el sistema.

6.5.4 Protocolo de directorio encadenado

Los tipos vistos anteriormente se aplican sobre todo a sistemas con directorio centralizado. Si el directorio se encuentra distribuido, lo que significa que la información del directorio se encuentra distribuida entre las memorias y/o cachés, se reduce el tamaño del directorio así como el cuello de botella que supone un directorio centralizado en multiprocesadores grandes. Directorios distribuidos hay de dos tipos, *protocolos de directorio jerárquico*, donde dividen el directorio entre varios *grupos* de procesadores, y *protocolos de directorio encadenado* basados en listas encadenadas de cachés.

Los protocolos de directorio jerárquico se utilizan a menudo en arquitecturas que consisten en un conjunto de grupos de procesadores *clusters* conectados por algún tipo de red. Cada cluster contiene un conjunto de procesadores y un directorio conectado a ellos. Una petición que no puede ser servida por las cachés en un directorio, es enviada a otros clusters tal y como determina el directorio.

Los protocolos de directorio encadenado mantienen una lista enlazada por punteros entre las cachés que tienen una copia de un bloque. La lista se puede recorrer en un sentido o en los dos dependiendo de que tenga uno o dos punteros cada nodo. Una entrada en el directorio apunta a una caché con una copia del bloque correspondiente a esa entrada; esta caché a su vez tiene un puntero que apunta a otra caché que tiene

también la copia, y así sucesivamente. Por lo tanto, una entrada en el directorio contiene únicamente un único puntero que apunta a la cabeza de la lista. Con esto, el tamaño del directorio es proporcional al número de líneas de memoria y al logaritmo en base 2 del número de procesadores tal y como ocurría en el protocolo de directorio limitado. Naturalmente las cachés deben incluir también uno o dos punteros, dependiendo del tipo de lista, además de los bits propios de estado.

Un protocolo interesante, basado en directorio encadenado, es el que presenta el estándar *IEEE Scalable Coherent Interface* o *SCI*. Este interface estándar es escalable permitiendo la conexión de hasta 64K procesadores, memorias, o nodos de entrada/salida. Hay un puntero, llamado *puntero de cabeza*, asociado a cada bloque de memoria. El puntero de cabeza apunta a la primera caché en la lista. Además, se asignan a cada caché dos punteros, uno con el predecesor en la lista y otro con el sucesor. Se trata por tanto de una lista con doble enlace. La figura 6.40 muestra la estructura de memoria de un sistema SCI con los punteros de cabeza y los asociados a cada caché.

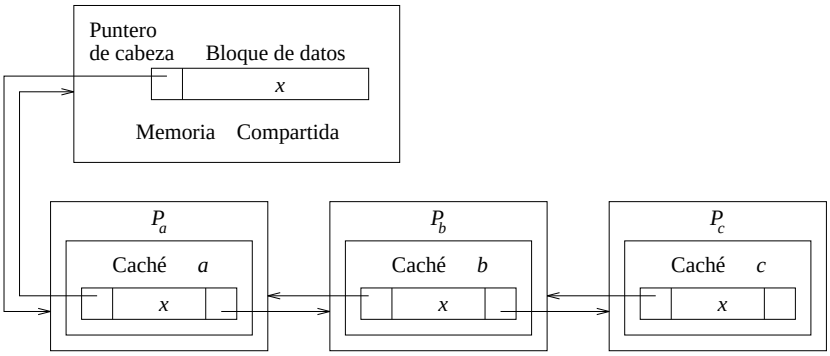


Figura 6.40: Protocolo de directorio encadenado en el estándar SCI.

Veamos las acciones a realizar en caso de fallos, aciertos, etc.

Fallo de lectura: Supongamos que la caché c envía una petición de fallo de lectura a la memoria. Si el bloque está en un estado *no cacheado*, es decir, no hay ningún puntero a ninguna caché, entonces la memoria manda el bloque a la caché. El estado del bloque se cambia a *cacheado* y el puntero de cabeza apunta a la caché. Si el estado del bloque es *cacheado*, es decir, el puntero de cabeza apunta a una caché con una copia válida del bloque, entonces la memoria envía el contenido del puntero, por ejemplo a, a c y actualiza el puntero de cabeza para que ahora apunte a c. La caché c entonces pone el puntero predecesor para que apunte al bloque de memoria. El siguiente paso consiste en que la caché c manda una petición a la caché a. Cuando a recibe la petición pone su puntero predecesor para que apunte a c y manda la línea de caché pedida a c.

Fallo de escritura: Supongamos que la caché c manda una petición de fallo de escritura a la memoria. La memoria entonces manda el puntero de cabeza (por ejemplo a) y actualiza el puntero de cabeza para apuntar a c. En este punto, la caché c que es el cabeza de lista, tiene autoridad para invalidar el resto de copias de caché para mantener sólo la copia que va a modificar. Esta invalidación se realiza enviando la caché c una petición de invalidación a a; entonces a invalida su copia y manda su puntero sucesor (que por ejemplo apunta a b) a c. En este momento la caché c usa este puntero sucesor para enviar una petición de invalidación de línea a b. Estas operaciones continúan hasta que se invalidan todas las copias. Por último se realiza la al escritura a la copia que es la única en la lista.

Acierto de escritura: Si la caché que escribe, c, es la única en la lista, simplemente actualiza la copia y no hace más.
Si la caché c es la cabeza de lista, entonces invalida todas las copias de caché para ser la única caché con copia. El proceso de invalidación es igual que el visto en el punto anterior. Por último escribe en la copia suya.
Si la caché c no está en cabeza entonces se quita a sí misma de la lista. Posteriormente interroga a la memoria para ver quién es la cabeza de lista y pone su puntero sucesor apuntando a la caché cabeza de lista en ese momento. La memoria pone su puntero para apuntar a c que será la nueva cabeza de lista. Ahora tenemos el caso anterior, así que se hace lo mismo, es decir, se invalidan el resto de copias y se escribe en la caché. Hay que hacer notar que en cualquier caso la lista se queda con un único elemento.

6.5.5 Rendimiento de los protocolos basados en directorio

Comparando los protocolos de directorio vistos hasta ahora se puede decir que los protocolos de mapeado completo ofrecen una mayor utilización del procesador que los encadenados y estos más que los de directorio limitado. Sin embargo, un directorio completamente mapeado requiere mucha más memoria que los otros dos. Por otro lado, la complejidad del directorio encadenado es mayor que la del limitado.

Comparando los protocolos basados en directorio con los de sondeo, podemos decir que los protocolos de directorio tienen la ventaja de ser capaces de restringir las peticiones de lectura/escritura sólo a aquellas cachés con una copia válida del bloque. Sin embargo, estos protocolos aumentan el tamaño de la memoria y las cachés debido a los bits y punteros extra. Los protocolos de sondeo tienen la ventaja de que su implementación es menos compleja. Sin embargo, estos protocolos de sondeo no se pueden escalar a un número elevado de procesadores y requieren cachés de doble puerto de alto rendimiento para permitir que el procesador ejecute instrucciones mientras sondea o fisga en el bus para ver las transacciones de las otras cachés.

El rendimiento de una máquina basada en directorio depende de muchos de los mismos factores que influían en el rendimiento de las máquinas basadas en bus (tamaño de la caché, número de procesadores y tamaño del bloque), como de la distribución de los fallos en la jerarquía de la memoria. La posición de los datos demandados depende tanto de la localización inicial como de los patrones de compartición. Empezaremos examinando el rendimiento básico de la caché en la carga de programas paralelos que utilizamos en la sección 6.4.7, para después determinar los efectos de los diferentes tipos de fallos.

Dado que las máquinas son más grandes y las latencias mayores que en el caso de multiprocesadores basados en snooping, empezaremos con una caché mayor (128 KB) y un bloque de 64 bytes. En las arquitecturas de memoria distribuida, la distribución de las peticiones entre locales y remotas es el factor clave para el rendimiento, ya que ésta afecta tanto al consumo del ancho de banda global como a la latencia vista por el nodo que realiza la petición. Por tanto, separaremos en las figuras de esta sección los fallos de caché en locales y remotos. Al mirar estas figuras hay que tener en cuenta que, para estas aplicaciones, la mayoría de los fallos remotos se producen debido a fallos de coherencia, aunque algunos fallos de capacidad también puede ser remotos, y en algunas aplicaciones con una distribución de datos pobre, estos fallos pueden ser significativos.

Como muestra la figura 6.41, la tasa de fallos con estas cachés no se ve afectada por

el cambio en el número de procesadores, con la excepción de Ocean, donde la tasa de fallos aumenta con 64 procesadores. Este aumento se debe al aumento de conflictos en el mapeo de la caché que ocurren cuando se reduce el tamaño de la malla, dando lugar a un aumento de los fallos locales, y al aumento de los fallos de coherencia, que son todos remotos.

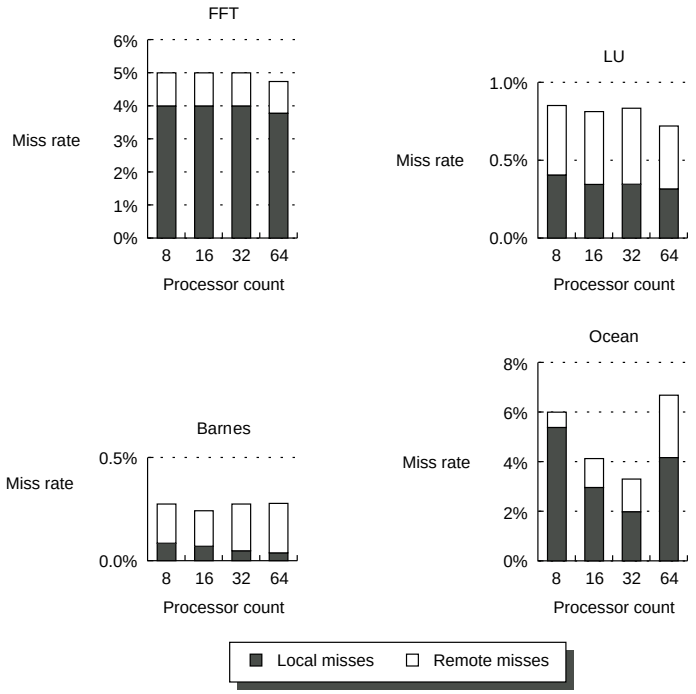


Figura 6.41: Tasa de fallo en función del número de procesadores.

La figura 6.42 muestra cómo cambia la tasa de fallos al incrementarse el tamaño de la caché, suponiendo una ejecución con 64 procesadores y bloques de 64 bytes. Estas tasas de fallo descienden tal como cabría esperar, aunque el efecto amortiguador causado por la casi nula reducción de los fallos de coherencia dan lugar a un menor decrecimiento de los fallos remotos que de los fallos locales. Para una caché de 512 KB, la tasa de fallo remota es igual o mayor que la tasa de fallo local. Tamaños de caché mayores sólo darán lugar a amplificar este fenómeno.

Finalmente, la figura 6.43 muestra el efecto de cambiar el tamaño del bloque. Dado que estas aplicaciones tienen buena localidad espacial, el incrementar el tamaño del bloque reduce la latencia de fallo, incluso para bloques grandes, aunque los beneficios en el rendimiento de utilizar los bloques de mayor tamaño son pequeños. Además, la mayoría parte de la mejora en la tasa de fallos se debe a los fallos locales.

Más que mostrar el tráfico de la memoria, la figura 6.44 muestra el número de bytes necesarios por referencia de datos en función del tamaño del bloque, dividiendo los requisitos de ancho de banda en local y global. En el caso de un bus, podemos unir ambos factores para determinar la demanda total para el bus y el ancho de banda de la memoria. Para un red de interconexión escalable, podemos usar los datos de la figura 6.44 para determinar el ancho de banda global por nodo y estimar ancho de banda de la bisección, como muestra el siguiente ejemplo.

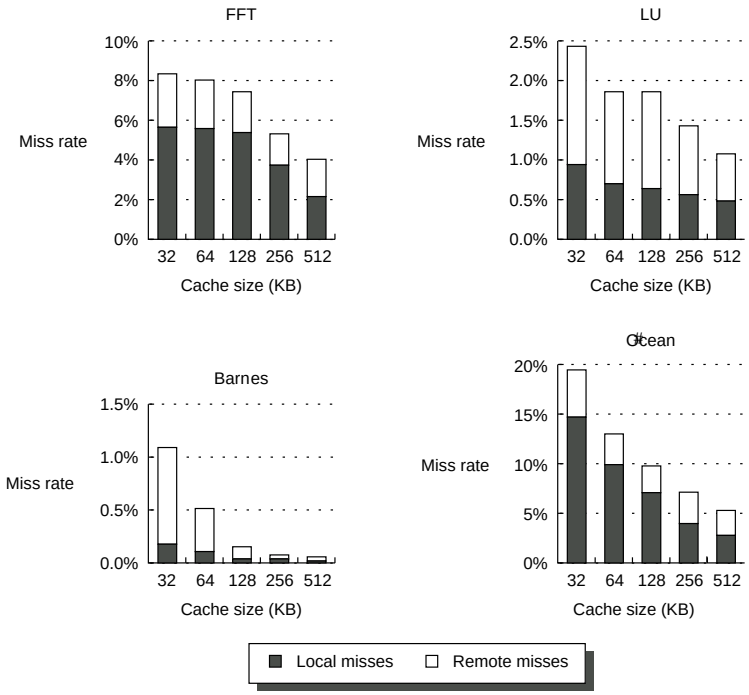


Figura 6.42: Tasa de fallo en función del tamaño de la caché.

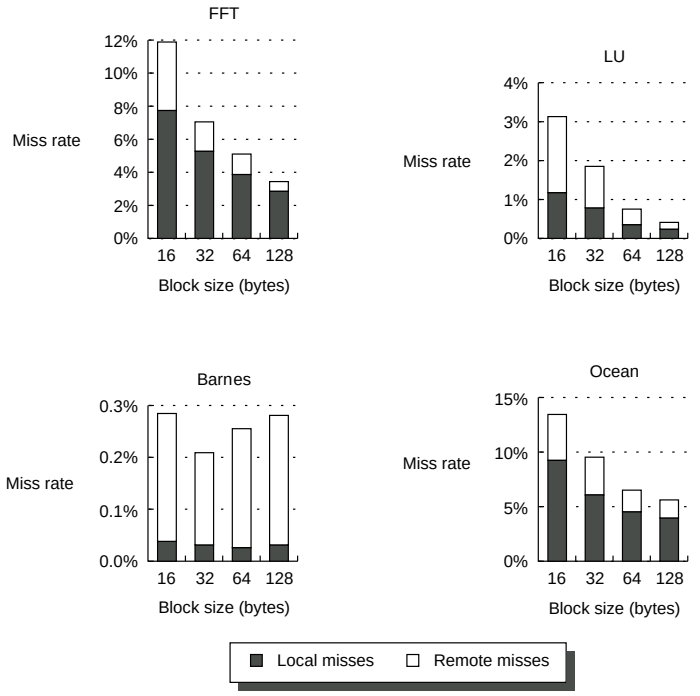


Figura 6.43: Tasa de fallo en función del tamaño del bloque suponiendo una caché de 128 KB y 64 procesadores.

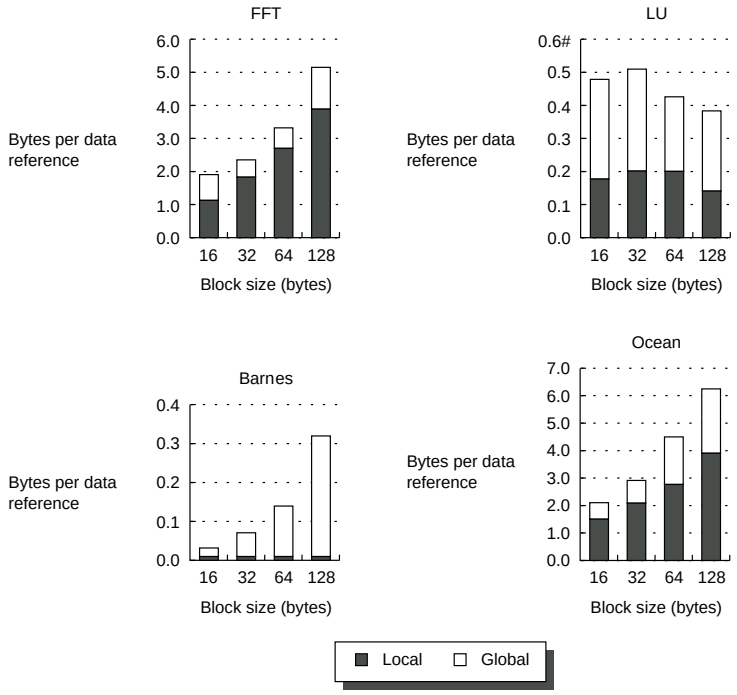


Figura 6.44: Número de bytes por referencia en función del tamaño del bloque.

Ejemplo Supongamos un multiprocesador de 64 procesadores a 200 MHz con una referencia a memoria por ciclo de reloj. Para un tamaño de bloque de 64 bytes, la tasa de fallo remota es del 0.7%. Determinar el ancho de banda por nodo y el ancho de banda estimado para la bisección para FFT. Suponer que el procesador no se para ante peticiones a memoria remota; esto puede considerarse cierto si, por ejemplo, todos los datos remotos son precargados.

Respuesta El ancho de banda por nodo es simplemente el número de bytes por referencia multiplicado por la tasa de referencias: $0.007 \times 200 \times 64 = 90\ MB/sec$. Esta tasa es aproximadamente la mitad del ancho de banda del más rápido MPP escalable disponible en 1995. La demanda de ancho de banda por nodo excede a las redes de interconexión ATM más rápida existente en 1995 por un factor de 5, y excederá ligeramente a la siguiente generación de ATM
FFT realiza comunicaciones del tipo todos a todos, así que el ancho de banda de la bisección será igual a 32 veces el ancho de banda por nodo, o 2880 MB/sec. Para una máquina de 64 procesadores conectada mediante una malla 2D, el ancho de banda de la bisección crece proporcionalmente a la raíz cuadrada del número de procesadores. Así, para 64 procesadores el ancho de banda de la bisección es 8 veces el ancho de banda del nodo. En 1995, las interconexiones del tipo MPP ofrecen cerca de 200 MB/sec por nodo para un total de 1600 MB/sec. Para 64 nodos una conexión del tipo malla 3D dobla este ancho de banda de la bisección (3200 MB/sec), que excede el ancho de banda necesario. La siguiente generación de mallas 2D también conseguirá alcanzar el ancho de banda necesario.

El ejemplo previo muestra la demanda del ancho de banda. Otro factor clave para los programas paralelos es el tiempo de acceso a memoria remota, o latencia. Para examinar este factor usaremos un ejemplo sobre una máquina basada en directorio. La figura 6.45

muestra los parámetros que supondremos para nuestra máquina. Supondremos que el tiempo de acceso para la primera palabra es de 25 ciclos y que la conexión a la memoria local tiene un ancho de 8 bytes, mientras que la red de interconexión tiene una anchura de 2 bytes. Este modelo ignora el efecto de la contención, que probablemente no es muy serio en los programas paralelos examinados, con la posible excepción de FFT, que usa comunicaciones del tipo todos a todos. La contención podría tener un serio impacto en el rendimiento para otro tipo de cargas de trabajo.

Characteristic	**Number of processor clock cycles**
Cache hit	1
Cache miss to local memory	$25 + \frac{\text{block size in bytes}}{8}$
Cache miss to remote home directory	$75 + \frac{\text{block size in bytes}}{2}$
Cache miss to remotely cached data (3-hop miss)	$100 + \frac{\text{block size in bytes}}{2}$

Figura 6.45: Características de la máquina basada en directorio del ejemplo.

La figura 6.46 muestra el coste en ciclos para una referencia a memoria, suponiendo los parámetros de la figura 6.45. Sólo se contabilizan las latencias de cada tipo de referencia. Cada barra indica la contribución de los éxitos de caché, fallos locales, fallos remotos y fallos remotos a una distancia de tres saltos. El coste está influenciado por la frecuencia total de los fallos de caché y actualizaciones, al igual que por la distribución de las localizaciones donde se satisface el fallo. El coste de una referencia a memoria remota permanece estable al aumentar el número de procesadores, excepto para Ocean. El incremento en la tasa de fallo en Ocean para 64 procesadores está claro a partir de la figura 6.41. Al aumentar la tasa de fallo, cabría esperar que el tiempo gastado en las referencias a memoria se incremente también.

Aunque la figura 6.46 muestra el coste de acceso a la memoria, que es el coste dominante en estos programas, un modelo completo del rendimiento debería considerar el efecto de la contención en el sistema de memoria, al igual que las pérdidas ocasionadas por los retrasos en la sincronización.

6.6 Sincronización

Los mecanismos de sincronización suelen implementarse mediante rutinas software que descansan en las instrucciones de sincronización proporcionadas por el hardware. Sin esta capacidad, el coste de construir las primitivas básicas de sincronización sería demasiado alto y se incrementaría al incrementarse el número de procesadores. Estas primitivas forman los bloques de construcción básicos para implementar una amplia variedad de operaciones de sincronización a nivel de usuario, incluyendo elementos tales como los cerrojos y las barreras. Durante años ha habido un considerable debate sobre qué primitivas hardware deben proporcionar las máquinas multiprocesador para implementar estas operaciones de sincronización. Las conclusiones han cambiado a lo largo del tiempo, con los cambios de la tecnología y el estilo de diseño de las máquinas.

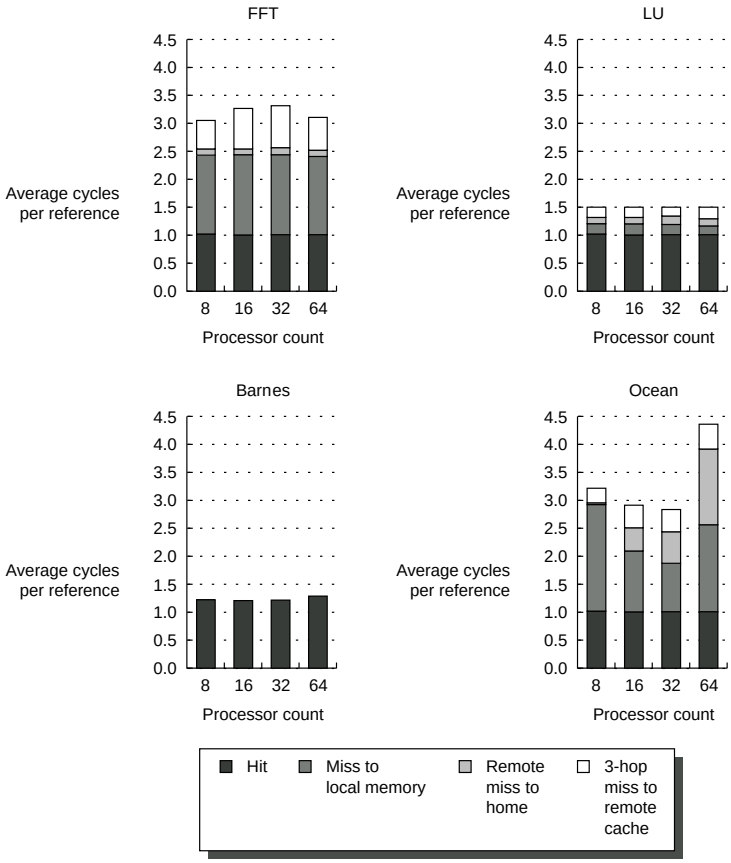


Figura 6.46: La latencia efectiva de las referencias a memoria en una máquina DSM depende de la frecuencia relativa de los fallos de caché y de la localización de la memoria desde donde se producen los accesos.

El soporte hardware tiene la ventaja de la velocidad, pero mover funcionalidad al software tiene la ventaja de la flexibilidad y la adaptabilidad a diferentes situaciones. Los trabajos de Dijkstra y Knuth mostraron que es posible proporcionar exclusión mutua únicamente con operaciones de lectura y escritura exclusiva (suponiendo una memoria con consistencia secuencial). Sin embargo, todas las operaciones de sincronización prácticas descansan en primitivas atómicas del tipo *leer-modificar-escribir*, en donde el valor de una posición de la memoria se lee, modifica y se vuelve a escribir de manera atómica.

La historia del diseño del conjunto de instrucciones nos permite observar la evolución de soporte hardware para la sincronización. Una de las instrucciones clave fue la inclusión en el IBM 370 de una instrucción atómica sofisticada, la instrucción *compare-and-swap*, para soportar la sincronización en multiprogramación sobre sistemas uniprocesador o multiprocesador. Esta instrucción compara el valor de una posición de la memoria con el valor de un registro específico, y si son iguales se intercambia el valor de la posición con el valor de un segundo registro. El Intel x86 permite añadir a cualquier instrucción un prefijo que la hace atómica, con lo que haciendo que los operandos fuente y destino sean posiciones de la memoria es posible utilizar la mayor parte del conjunto de instrucciones para implementar operaciones atómicas. También se ha propuesto que las operaciones de sincronización a nivel de usuario, como barreras y candados, deben ser soportadas a nivel de máquina, y no sólo las primitivas *leer-modificar-escribir*; es decir, el propio algoritmo de sincronización debe implementarse en hardware. Esta cuestión pasó a ser muy activa a raíz del debato sobre la reducción del conjunto de instrucciones, ya que las operaciones que acceden a memoria fueron reducidas a simple instrucciones de lectura y escritura con un único operando. La solución de SPARC fue proporcionar operaciones atómicas de intercambio entre un registro y una dirección de memoria (intercambio atómico entre el contenido de la posición de memoria y el registro) y de comparación e intercambio, mientras que MIPS renunció a las primitivas atómicas utilizadas en sus anteriores conjuntos de instrucciones, al igual que hizo la arquitectura IBM Power usada en el RS6000. La primitiva que se introdujo fue la combinación de un par de instrucciones que incluye una instrucción de lectura especial denominada *load linked* o *load locked* y una instrucción de escritura especial denominada *store conditional*. Estas instrucciones se usan como una secuencia: Si el contenido de la dirección de memoria especificada por la carga se modifica antes de que ocurra el almacenamiento condicionado, entonces este último no se realiza. Si el procesador realiza un cambio de contexto entre las dos instrucciones, tampoco se realiza el almacenamiento condicionado. El almacenamiento condicionado se define de tal manera que devuelva un valor para indicar si dicho la operación se puedo realizar con éxito. Esta aproximación ha sido posteriormente incorporada por las arquitectura PowerPC y DEC Alpha, siendo en la actualidad muy popular. En la figura 6.47 se muestra un ejemplo de cómo se pueden implementar algunas operaciones atómicas con la utilización de estas dos operaciones.

Componentes de un evento de sincronización

Consideremos los componentes de un evento de sincronización, lo que nos permitirá ver claramente por qué el soporte directo en hardware de la exclusión mutua y otras operaciones similares es difícil y da lugar a implementaciones demasiado rígidas. Existen tres componentes principales para un tipo dado de evento de sincronización:

```
try:  mov   R3, R4      ; mov exchange value
      ll    R2, 0(R1)   ; load linked
      sc    R3, 0(R1)   ; store conditional
      beqz  R3, try     ; branch store fails
      mov   R4, R2      ; put load value in R4
```

```
try:  ll    R2, 0(R1)   ; load linked
      addi  R2, R2, #1  ; increment
      sc    R2, 0(R1)   ; store conditional
      beqz  R2, try     ; branch store fails
```

Figura 6.47: Implementación de (a) un intercambio atómico y (b) una operación de lectura e incremento (*fetch-and-increment*) utilizando las operaciones *load linked* y *store conditional*

> un *método de adquisición*: un método mediante el cual un proceso intenta adquirir el derecho de la sincronización (para entrar a la sección crítica o para proceder a pasar dicho evento)
>
> un *algoritmo de espera*: un método mediante el cual un proceso espera a que una sincronización esté disponible cuando ésta no lo está. Por ejemplo, si un proceso intenta adquirir un cerrojo pero éste no está libre (o quiere proceder a pasar un evento que todavía no ha ocurrido), éste debe esperar de alguna manera hasta que el cerrojo esté libre.
>
> un *método de liberación*: un método que permita a un proceso habilitar el paso de un evento de sincronización a otros procesos; por ejemplo, una implementación de la operación UNLOCK, un método para que el último proceso que llega a una barrera libere a los procesos que están esperando, o un método para notificar a un proceso que está esperando en un evento punto a punto que el evento ha ocurrido.

La elección del algoritmo de espera es bastante independiente del tipo de sincronización. ¿Qué debe hacer un proceso que llega al punto de adquisición mientras que espera a que ocurra una liberación? Existen dos alternativas: *espera activa* y *bloqueo*. La espera activa significa que el proceso se queda ejecutando un bucle que testea de forma repetitiva si una variable a cambiado de valor. Un liberación del evento de sincronización por parte de otro proceso cambia el valor de la variable, permitiendo proceder al proceso que estaba esperando. En el caso del bloqueo, el proceso simplemente se bloquea (suspende) indicando al procesador el evento que está esperando que ocurra. El proceso será despertado y puesto en un estado de listo para su ejecución cuando la liberación que estaba esperando ocurra. Las diferencias entre la espera activa y el bloqueo son claras. La espera activa se comporta mejor cuando el periodo de espera es pequeño, mientras que el bloqueo es preferible en el caso de que el periodo de espera sea grande y si hay otros procesos en ejecución. La dificultad de implementar operaciones de sincronización de alto nivel en hardware no reside en los componentes de adquisición y liberación, sino en el algoritmo de espera. Por esta razón, es conveniente proporcionar soporte hardware para los aspectos críticos de la adquisición y liberación y permitir que los tres componentes se unan en software.

El papel del usuario, del sistema software y del hardware

¿Quién debe ser el responsable de implementar las operaciones de sincronización de alto nivel tales como los candados y las barreras?. Normalmente, un programador quiere usar candados, eventos, o incluso operaciones de más alto nivel sin tenerse que preocupar de su implementación interna. La implementación se deja en manos del sistema,

que debe decidir cuánto soporte hardware debe proporcionar y cuánta funcionalidad implementar en software. Se han desarrollado algoritmos de sincronización software que utilizan simples primitivas atómicas de intercambio y que se aproximan a la velocidad proporcionada por las implementaciones basadas totalmente en hardware, lo que los hace muy atractivos. Como en otros aspectos de diseño de un sistema, la utilización de operaciones más rápidas depende de la frecuencia de utilización de estas operaciones en las aplicaciones. Así, una vez más, la respuesta estará determinada por un mejor entendimiento del comportamiento de la aplicación.

Las implementaciones software de las operaciones de sincronización suelen incluirse en librerías del sistema. Muchos sistemas comerciales proporcionan subrutinas o llamadas al sistema que implementan cerrojos, barreras e incluso algún otro tipo de evento de sincronización. El diseño de una buena librería de sincronización

puede suponer un desafío. Una complicación potencial es que el mismo tipo de sincronización, e incluso la misma variable de sincronización, puede ejecutarse en diferentes ocasiones bajo condiciones de ejecución muy diferentes. Estos escenarios diferentes imponen requisitos en el rendimiento muy diferentes. Bajo una situación de alta contención, la mayoría de los procesos gastarán su tiempo esperando por lo que el requisito clave será que el algoritmo proporcione un gran ancho de banda para las operaciones de lock-unlock, mientras que en condiciones de baja carga la meta es proporcionar baja latencia para la adquisición del cerrojo. Una segunda complicación es que los multiprocesadores se usan normalmente con cargas de multiprogramación donde la planificación de los procesos y otras interacciones entre los recursos pueden cambiar el comportamiento de los procesos de un programa paralelo en cuanto a la sincronización. Todos estos aspectos hacen de la sincronización un punto crítico en la interacción hardware/software.

6.6.1 Cerrojos (exclusión mutua)

La operaciones de exclusión mutua (lock/unlock) se implementan utilizando un amplio rango de algoritmos. Los algoritmos simples tienden a ser más rápidos cuando existen poca contención, pero son ineficientes ante altas contenciones, mientras que los algoritmos sofisticados que se comportan bien en caso de contención tienen un mayor coste en el caso de baja contención.

Implementación de cerrojos usando coherencia

Antes de mostrar los distintos mecanismos de implementación de la exclusión mutua es conveniente articular algunas metas de rendimiento que perseguimos en el caso de los cerrojos. Estas metas incluyen:

> *Baja latencia.* Si un cerrojo está libre y ningún otro procesador está tratando de adquirirlo al mismo tiempo, un procesador debe ser capaz de adquirirlo con baja latencia.
>
> *Bajo tráfico.* Supongamos que varios o todos los procesadores intentan adquirir un cerrojo al mismo tiempo. Debería ser posible la adquisición del cerrojo de forma consecutiva con tan poca generación de tráfico o transacciones de bus como sea posible. Un alto tráfico puede ralentizar las adquisi-

ciones debido a la contención, y también puede ralentizar transacciones no relacionadas que compiten por el bus.

Escalabilidad. En relación con el punto anterior, ni la latencia ni el tráfico deben variar más rápidamente que el número de procesadores usados. Hay que tener en cuenta que dado que el número de procesadores en un SMP basado en bus no es probable que sea grande, no es importante la escalabilidad asintótica.

Bajo coste de almacenamiento. La información que es necesaria debe ser pequeña y no debe escalar más rápidamente que el número de procesadores.

Imparcialidad. Idealmente, los procesadores deben adquirir un cerrojo en el mismo orden que sus peticiones fueron emitidas. Al menos se debe evitar la muerte por inanición o una imparcialidad excesiva.

La primera implementación que veremos es la implementación de los *spin locks*: cerrojos que el procesador intenta adquirir continuamente ejecuanto un bucle. Como ya se comentó, este tipo de cerrojos se usan cuando se espera que el tiempo que se va a estar esperando es pequeño.

Una primera implementación de un cerrojo de este tipo sería el que se presenta en el siguiente código:

```
        li    R2, #1
lockit: exch  R2, 0(R1)
        bnez  R2, lockit
```

Una de las ventajas de esta implementación se produce en el caso de que exista una localidad en el acceso al cerrojo: es decir, el procesador que ha usado el cerrojo en último lugar es probable que lo use de nuevo en un futuro cercano. En estos casos, el valor del cerrojo ya reside en la caché del procesador, reduciendo el tiempo necesario para adquirir el cerrojo.

Otra ventaja que nos permite la coherencia es que el test y la adquisición del cerrojo se pueda realizar sobre la copia local sin necesidad de transacciones de bus, reduciendo el tráfico del bus y los problemas de congestión. Para conseguir esta ventaja es necesario realizar un cambio en la implementación. Con la implementación actual, cada intento de intercambio requiere una operación de escritura. Si

varios procesadores intentan obtener el cerrojo, cada uno generará una escritura. La mayoría de estas escrituras darán lugar a fallos de escritura, dado que cada procesador está intentando obtener la variable cerrojo en un estado exclusivo.

Así, podemos modificar la implementación de tal manera que el bucle dé lugar a lecturas sobre una copia local del cerrojo hasta que observe que está disponible. El procesador primero lee el valor de la variable para testear su estado. Un procesador sigue leyendo y testeando hasta que el valor de la lectura indica que el cerrojo está libre. El procesador después compite con el resto de procesadores que también pudiesen estar esperando. Todos los procesos utilizan una instrucción de intercambio que lee el valor antiguo y almacena un 1 en la variable cerrojo. El ganador verá un 0, y los perdedores un 1 en la variable cerrojo. El procesador ganador ejecuta el código existente después de la adquisición del cerrojo y, cuando termina, almacena un 0 en la variable cerrojo

para liberarlo, con lo que comienza de nuevo la carrera para adquirirlo. A continuación se muestra el código del cerrojo mejorado:

```
lockit: lw    R2, 0(R1)
        bnez  R2, lockit
        li    R2, #1
        exch  R2, 0(R1)
        bnez  R2, lockit
```

Examinemos cómo este esquema usa el mecanismo de coherencia de la caché. La figura 6.48 muestra el procesador y las operaciones de bus cuando varios procesos intentan adquirir la variable cerrojo usando un intercambio atómico. Una vez que el procesador que tiene el cerrojo lo libera (almacena un 0), las demás cachés son invalidadas y deben obtener el nuevo valor para actualizar sus copias de la variable. Una de esas cachés obtiene la copia con el nuevo valor del cerrojo (0) y realiza el intercambio. Cuando se satisface el fallo de caché para los otros procesadores, encuentran que la variable ya está adquirida, así que vuelven al bucle de testeo.

Step	Processor P0	Processor P1	Processor P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive	Write invalidate of lock variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back from P0
4		(Waits while bus/ directory busy)	Lock = 0	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets Lock =1	Exclusive	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1	Enter critical section	Shared	Bus/directory services P2 cache miss; generates write back
8		Spins, testing if lock = 0			None

Figura 6.48: Pasos en la coherencia de la caché y tráfico en el bus para tres procesadores, P0, P1 y P2 que intentan adquirir un cerrojo en protocolo de coherencia por invalidación.

Este ejemplo muestra otra ventaja de las primitivas *load-linked/store-conditional*: las operaciones de lectura y escritura están explícitamente separadas. La lectura no necesita producir ningún tráfico en el bus. Esto permite la siguiente secuencia de código que tiene las mismas características que la versión de intercambio optimizada

(R1 contiene la dirección del cerrojo):

```
lockit: ll    R2, 0(R1)
        bnez  R2, lockit
        li    R2, #1
        sc    R2, 0(R1)
        bnez  R2, lockit
```

El primer bucle comprueba el cerrojo, mientras que el segundo resuelve las carreras cuando dos procesadores ven el cerrojo disponible de forma simultánea.

Aunque nuestro esquema de cerrojo es simple y convincente, tiene el problema a la hora de escalar debido al tráfico generado cuando se libera el cerrojo. Para entender por qué no escala bien, imaginemos una máquina con todos los procesadores compitiendo por el mismo cerrojo. El bus actúa como un punto de serialización para todos los procesadores, dando lugar a mucha contención. El siguiente ejemplo muestra cómo pueden ir de mal las cosas.

Ejemplo Supongamos 20 procesadores en un bus, cada uno de ellos intentando obtener el acceso en exclusiva sobre una variable. Supongamos que cada transacción del bus (fallo de lectura o de escritura) tarda 50 ciclos. Vamos a ignorar el tiempo de acceso a la caché, así como el tiempo que el cerrojo permanece cerrado. Para determinar el número de transacciones de bus necesarias para que los 20 procesadores adquieran el cerrojo, suponiendo que todos están en el bucle principal cuando se libera el cerrojo en el tiempo 0. ¿Cuánto se tardará en procesar las 20 peticiones? Supongamos que el bus es totalmente justo de tal manera que cualquier petición pendiente se sirve antes de cualquier nueva petición y que los procesadores son iguales de rápidos.

Respuesta La figura 6.49 muestra la secuencia de eventos desde que se libera el cerrojo hasta la siguiente vez que se libera. Por supuesto, el número de procesadores compitiendo por el cerrojo desciende cada vez que se adquiere el mismo, lo que reduce el coste medio a 1525 ciclos. Por lo tanto, para 20 pares de eventos lock-unlock será necesario más de 30.000 ciclos para que todos los procesadores adquieran el cerrojo. Además, un procesador estará esperando, en promedio, la mitad de este tiempo sin hacer nada, simplemente intentando obtener la variable. El número de transacciones de bus generadas supera las 400.

La raíz del problema está en la contención y en el hecho de que los accesos se serialicen. Para solucionar estos problemas vamos a estudiar una serie de algoritmos avanzados que intentarán tener baja latencia en el caso de poca o nula contención y que minimice la serialización en el caso de que la contención sea significativa.

Algoritmos avanzados para cerrojos

Como ya se ha comentado es deseable tener un único proceso intentando obtener un cerrojo cuando este se libera (más que permitir que todos los procesadores intenten adquirirlo a la vez). Sería incluso más deseable que únicamente un procesador incurriera en un fallo de lectura cuando se produce una liberación. En primer lugar veremos una modificación del cerrojo optimizado que reduce la contención retrasando artificialmente

Event	Duration
Read miss by all waiting processors to fetch lock (20×50)	1000
Write miss by releasing processor and invalidates	50
Read miss by all waiting processors (20×50)	1000
Write miss by all waiting processors, one successful lock (50), and invalidation of all lock copies (19×50)	1000
Total time for one processor to acquire and release lock	3050 clocks

Figura 6.49: Tiempo para adquirir y liberar un cerrojo cuando 20 procesadores están compitiendo por el mismo.

los procesos cuando fallan en la adquisición de un cerrojo. Posteriormente veremos dos algoritmos: *ticket lock* (que garantiza la primera afirmación) y *array-based lock* (que garantiza ambas) y que aseguran que el acceso al cerrojo se realizará en una ordenación FIFO.

Spin lock con *exponential back-offs*. Este método opera retrasando artificialmente los procesos cuando no obtienen el cerrojo. El mejor rendimiento se obtiene cuando este retraso crece de forma exponencial en función del número de intentos para adquirir el cerrojo. La figura 6.50 muestra una implementación de este algoritmo. Esta implementación intenta preservar una baja latencia cuando la contención es pequeña no retrasando la primera iteración.

```
        li    R3,1        ;R3 = initial delay
lockit: ll    R2,0(R1)    ;load linked
        bnez  R2,lockit   ;not available-spin
        addi  R2,R2,1     ;get locked value
        sc    R2,0(R1)    ;store conditional
        bnez  R2,gotit    ;branch if store succeeds
        sll   R3,R3,1     ;increase delay by 2
        pause R3          ;delays by value in R3
        j     lockit
goit:   use data protected by lock
```

Figura 6.50: Un cerrojo con *exponential back-off*.

Ticket Lock. Este algoritmo actúa como un sistema de ticket. Cada proceso que espera la adquisición de un cerrojo toma un número, y realiza una espera activa sobre un número global que indica a quién se está sirviendo en la actualidad —como el número que muestra la pantalla luminosa en un número de espera— hasta que el número global es igual al número que ha obtenido. Para liberar el cerrojo, un proceso simplemente incrementa el número global. La primitiva atómica necesaria es del tipo *fetch&increment*, que utiliza un proceso para obtener su ticket del contador compartido. Esta primitiva puede implementarse mediante una instrucción atómica o usando LL-SC. Este cerrojo presenta una sobrecarga en el caso de ausencia de contención similar a la del

cerrojo LL-SC. Sin embargo, al igual que el cerrojo LL-SC presenta todavía un problema de tráfico. La razón es que todos los procesadores iteran sobre la misma variable. Cuando la variable se modifica en la liberación, todas las copias en las cachés se invalidan e incurren en un fallo de lectura. (El cerrojo LL-SC presentaba un comportamiento un poco peor, ya que en ese caso ocurría otra invalidación y un conjunto de lecturas cuando un procesador tenía éxito en su SC). Una manera de reducir este tráfico es introducir algún tipo de espera. No queremos usar una espera exponencial ya que no queremos que todos los procesos esperen cuando se libere el cerrojo y así ninguno intente adquirirlo durante algún tiempo. Un técnica de compromiso es que cada procesador retrase la lectura del contador una cantidad proporcional al tiempo que estiman que tardará en llegarle su turno; es decir, una cantidad proporcional a la diferencia entre su ticket y el número del contador en su última lectura.

Array-based Lock.La idea aquí es utilizar una instrucción *fetch&increment* para obtener no un valor sino una localización única sobre la que realizar la espera activa. Si existen p procesadores que pueden completar un cerrojo, entonces el cerrojo contiene un array con p localizaciones sobre las cuales los procesos pueden iterar, idealmente cada una en un bloque de memoria separado para evitar la falsa compartición. El método de adquisición es usar una operación *fetch&increment* para obtener la siguiente dirección disponible en el array sobre la cual iterar, el método de espera es iterar sobre esta localización, y el método de liberación consiste en escribir un valor que indica "liberado" sobre la siguiente localización en el array. Sólo el procesador que estaba iterando sobre esa localización tiene en su caché un bloque invalidado, y un posterior fallo de lectura le indica que ha obtenido el cerrojo. Como en el *ticket lock*, no es necesario realizar una operación de testeo después del fallo ya que únicamente se notifica a un proceso la liberación del cerrojo. Este cerrojo es claramente FIFO y por lo tanto justo. Su latencia en el caso de ausencia de contención es similar a de los cerrojos vistos hasta ahora y es más escalable que el algoritmo anterior ya que solamente un proceso incurre en un fallo de lectura. El único inconveniente es que el espacio usado es $O(p)$ en lugar de $O(1)$, pero dado que p y la constante de proporcionalidad son pequeños el problema no es significativo. También presenta un inconveniente potencial en el caso de máquinas de memoria distribuida.

Algoritmos para cerrojos en sistemas escalables

En la sección 6.6 se discutieron los distintos tipos de cerrojos existentes, cada uno de los cuales daba un paso más en la reducción del tráfico del bus y de la equidad, pero a menudo a costa de una mayor sobrecarga. Por ejemplo, el *ticket lock* permite que únicamente un procesador pida el cerrojo cuando este se libera, pero la notificación de la liberación llega a todos los procesadores a través de una invalidación y el posterior fallo de lectura. El cerrojo basado en array soluciona este problema haciendo que cada proceso espere sobre diferentes localizaciones, y el procesador que libera el cerrojo únicamente notifica la liberación a un procesador escribiendo en la localización correspondiente.

Sin embargo, las cerrojos basados en array tienen dos problemas potenciales en máquinas escalables con memoria física distribuida. En primer lugar, cada cerrojo requiere un espacio proporcional al número de procesadores. En segundo lugar, y más importante para las máquinas que no permiten tener en caché datos remotos, no hay forma de saber por anticipado sobre qué localización iterará un proceso concreto, dado que dicha localización se determina en tiempo de ejecución a través de una operación

fetch&increment. Esto hace imposible almacenar las variables de sincronización de manera que la variable sobre la cual itera un proceso esté siempre en su memoria local (de hecho, todos los cerrojos del capítulo anterior tienen este problema). En máquinas de memoria distribuida sin coherencia de cachés, como la Cray T3D y T3E, éste es un gran problema ya que los procesos iterarían sobre localizaciones remotas, dando lugar a una gran cantidad de tráfico y contención. Afortunadamente, existe un algoritmo software que reduce la necesidad de espacio al tiempo que asegura que todas las iteraciones se realizarán sobre variables almacenadas localmente.

Software Queuing Lock. Este cerrojo es una implementación software de un cerrojo totalmente hardware propuesto inicialmente en el proyecto Wisconsin Multicube. La idea es tener una lista distribuida o cola de procesos en espera de la liberación del cerrojo. El nodo cabecera de la lista representa el proceso que tiene el cerrojo. Los restantes nodos son procesos que están a la espera, y están la memoria local del proceso. Un nodo apunta al proceso (nodo) que ha intentado adquirir el cerrojo justo después de él. También existe un puntero cola que apunta al último nodo de la cola, es decir, el último nodo que ha intentado adquirir el cerrojo. Veamos de una manera gráfica como cambia la cola cuando los procesos adquieren y liberan el cerrojo.

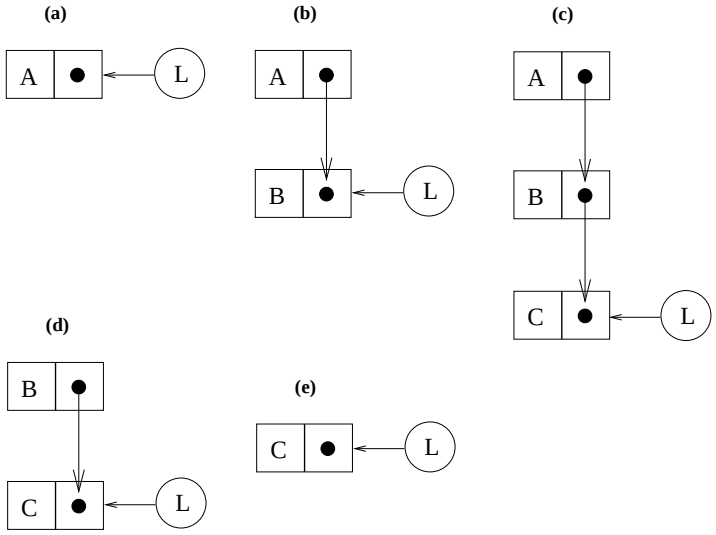


Figura 6.51: Estados de la lista para un cerrojo cuando los procesos intentan adquirirlo y cuando lo liberan.

Supongamos que el cerrojo de la figura 6.51 está inicialmente libre. Cuando un proceso A intenta adquirir el cerrojo, lo obtiene y la cola quedaría como se muestra en la figura 6.51(a). En el paso (b), el proceso B intenta adquirir el cerrojo, por lo que se pone a la cola y el puntero cola ahora apunta a él. El proceso C es tratado de forma similar cuando intenta adquirir el cerrojo en el paso (c). B y C están ahora iterando sobre las variables locales asociadas con sus nodos en la cola mientras que A mantiene el cerrojo. En el paso (d), el proceso A libera el cerrojo. Después "despierta" al siguiente proceso, B, de la cola escribiendo en la variable asociada al nodo B, y dejando la cola. B tiene ahora el cerrojo, y es la cabeza de la cola. El puntero cola no cambia. En el paso (e), B libera el cerrojo de manera similar, pasándoselo a C. Si C libera el cerrojo antes de que otro proceso intente adquirirlo, el puntero al cerrojo será NULL y el cerrojo estará otra vez libre. De esta manera, los procesos tienen garantizado un acceso al cerrojo en una ordenación FIFO con respecto al orden en el que intentaron adquirirlo.

Debe quedar claro que en el cerrojo por cola software el espacio necesario es proporcional al número de procesos que están esperando o participando en el cerrojo, no al número de procesos en el programa. Este sería el cerrojo que se elegiría para máquinas que soportan un espacio de memoria compartido con memoria distribuida pero sin coherencia de cachés.

En el caso de máquinas con coherencia de caché, al igual que con el resto de algoritmos que veamos, se pueden aplicar los mismos algoritmos teniendo en cuenta dos diferencias. Por una parte, las implicaciones de rendimiento de iterar sobre variables remotas son menos significativas, dada la posibilidad de tener una copia de la variable en la caché local. El que cada procesador itere sobre una variable distinta es, por supuesto, útil ya que de esta forma no todos los procesadores deben ir al mismo nodo origen para solicitar el nuevo valor de la variable cuando esta se invalida, reduciendo la contención. Sin embargo, existe una razón (muy poco probable) que puede ser muy importante para el rendimiento en el caso de que la variable sobre la cual itera un procesador esté almacenada localmente: si la caché está unificada, es de asignación directa y las instrucciones de bucle de iteración entran en conflicto con la propia variable, en cuyo caso los fallos debido a conflicto se satisfacen localmente. Un beneficio menor de una buena ubicación es convertir los fallos que ocurren después de una invalidación en fallos de dos saltos en lugar de tres.

Rendimiento de los cerrojos

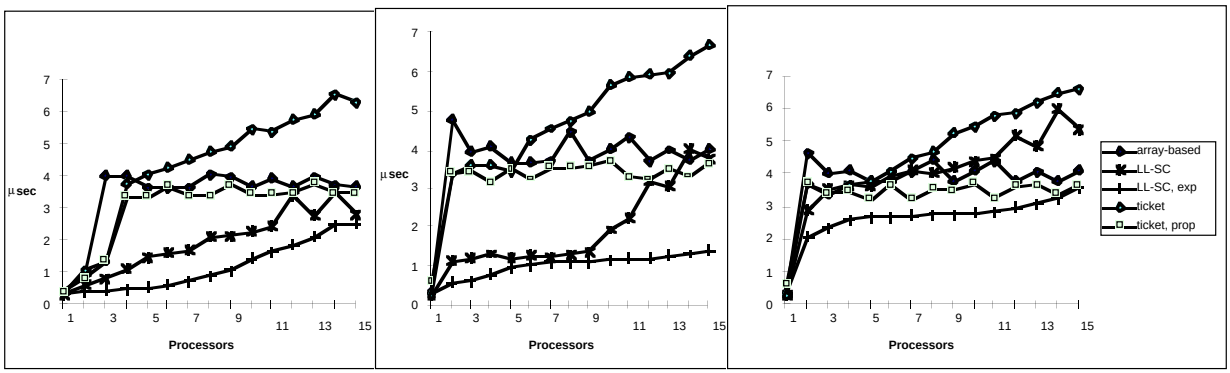
A continuación examinaremos el rendimiento de diferentes cerrojos en el SGI Challenge, como se muestra en las figura 6.52. Todos los cerrojos se han implementado usando LL-SC, ya que Challenge únicamente proporciona estas instrucciones. El cerrojos *test&set* se han implementado simulando la instrucción test&set mediante instrucciones LL-SC. En particular, cada vez que una SC falla en una escritura realizada sobre otra variable en el mismo bloque de caché, causa una invalidación al igual que haría la instrucción test&set. Los resultados se muestran en función de dos parámetros: (c) duración de la sección crítica y (d) tiempo entre la liberación del cerrojo y la siguiente vez que se intenta adquirirlo. Es decir, el código es un bucle sobre el siguiente cuerpo:

lock(L); critical_section(c); unlock(L); delay(d)

Vamos a considerar tres casos —(i) $c = 0$, $d = 0$, (ii) $c = 3.64\mu s$, $d = 0$, y (iii) $c = 3.64\mu s$, $d = 1.29\mu s$— denominados *nulo, sección crítica* y *retraso*, respectivamente. Recordar que en todos los casos c y d (multiplicado por el número de adquisiciones de cerrojo por cada procesador) se substrae del tiempo total de ejecución.

Consideremos el caso de sección crítica nula. En este caso se observa la implementación LL-SC básica del cerrojo se comporta mejor que las más sofisticadas *ticket lock* y *array-based lock*. La razón del mejor comportamiento de los cerrojos LL-SC, particularmente para pocos procesadores, es que no son equitativos, y esta falta de equidad se explota en las interacciones a nivel de arquitectura. En particular, cuando un procesador que realiza una escritura para liberar un

cerrojo realiza inmediatamente una lectura (LL) para su siguiente adquisición, es muy probable que la lectura y la SC tengan éxito en su caché antes de que otro procesador pueda leer el bloque en el bus.(En el caso del Challenge la diferencia es mucho



(a) null (c=0, d=0) (b) critical-section (c=3.64 µs, d=0) (c) delay (c=3.64 µs, d=1.29 µs)

Figura 6.52: Rendimiento de los cerrojos en el SGI Challenge para tres posibles escenarios.

más grande ya que el procesador que ha liberado el cerrojo puede satisfacer su siguiente lectura a través de su buffer de escritura antes incluso de que la correspondiente lectura exclusiva llegue al bus). La transferencia del cerrojo es muy rápida, y el rendimiento es bueno. Al incrementarse el número de procesadores, la probabilidad de autotransferencia desciende y el tráfico en el bus debido a las invalidaciones y a fallos de lectura se incrementa, por lo que también se incrementa el tiempo entre trasferencias del cerrojo. El retraso exponencial ayuda a reducir el tráfico en ráfagas y por tanto disminuye el ratio de escalada.

Al otro extremo ($c = 3.64\mu s$, $d = 1.29\mu s$), observamos que el cerrojo LL-SC no tiene un buen comportamiento, incluso para pocos procesadores. Esto es debido a que el procesador realiza una espera después de liberar el cerrojo y antes de intentar adquirirlo de nuevo, haciendo mucho más probable que otros procesadores en espera adquieran el cerrojo antes. Las autotransferencias son escasas, de tal manera que la transferencia del cerrojo es más lenta incluso en el caso de dos procesadores. Es interesante observar que el rendimiento para el caso de retraso exponencial es particularmente malo cuando el retraso d entre la liberación y la siguiente adquisición es distinto de cero y el número de procesadores es pequeño. Esto se debe a que para pocos procesadores, es muy probable que mientras que un procesador que acaba de liberar el candado está esperando a que expire d antes de intentar su siguiente adquisición, los otros procesadores están en un periodo de espera y ni siquiera intentan obtener el cerrojo.

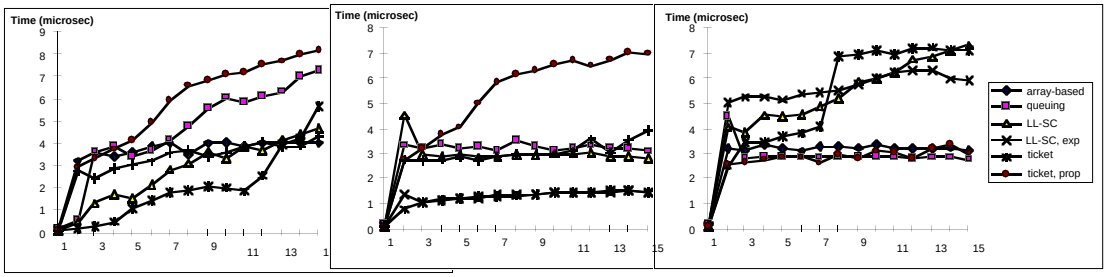
Consideremos el resto de implementaciones. Estos son equitativos, de tal manera que cada transferencia de adquisición es a un procesador diferente e implica transacciones de bus en el camino crítico de la transferencia. De aquí que todas impliquen cerca de tres transacciones de bus en el camino crítico por transferencia de cerrojo incluso cuando se usan dos procesadores. Las diferencias reales en el tiempo se deben a las transacciones de bus exactas y a la latencia que pueda ser ocultada por parte del procesador. El *ticket lock* sin periodos de espera escala pobremente: Con todos los procesos intentando leer el contador de servicio, el número esperado de transacciones entre la liberación y la lectura por parte del procesador correcto es $p/2$, dando lugar a una degradación lineal en la transferencia del cerrojo en el camino crítico. Con una espera proporcional, es probable que el procesador correcto sea el que realice la lectura en primer lugar después de la liberación, de tal manera que el tiempo por transferencia no aumenta con p. El *array-based lock* también presenta una propiedad similar, dado que únicamente el procesador correcto realiza la lectura, de tal manera que su rendimiento no se degrada

con el incremento de procesadores.

Los resultados ilustran la importancia de las interacciones de la arquitectura en la determinación del rendimiento de los cerrojos, y que el cerrojo LL-SC se comporta bastante bien en buses que tienen suficiente ancho de banda (y número de procesadores por buses reales). El rendimiento para los cerrojos LL-SC no equitativos aumenta hasta ser tan mala o un poco peor que el más sofisticado de los cerrojos para más de 16 procesadores, debido al mayor tráfico, pero no demasiado debido al gran ancho de banda. Cuando se utiliza la espera exponencial para reducir el tráfico, el cerrojo LL-SC muestra el mejor tiempo de transferencia para todos los casos. Los resultados ilustran la dificultad y la importancia de la metodología experimental en la evolución de los algoritmos de sincronización. Las secciones críticas nulas muestran algunos efectos interesantes, pero una comparación significativa depende de los patrones de sincronización en la aplicaciones reales. Un experimento que utiliza LL-SC pero garantiza una adquisición en *round-robin* entre los procesos (equidad) usando una variable adicional muestra un rendimiento muy similar al *ticket lock*, confirmando que la falta de equidad y las autotransferencias son la razón del mejor rendimiento para pocos procesadores.

Rendimiento de los cerrojos para sistemas escalables

Los experimentos realizados para ilustrar el rendimiento de la sincronización son los mismos que se usaron en la sección 6.6 para el SGI Challenge, usando de nuevo LL-SC como primitiva para construir operaciones atómicas. El resultado para los algoritmos que hemos discutido se muestran en la figura 6.53 para ejecuciones con 16 procesadores sobre la máquina SGI Origin2000. De nuevo se usan tres conjuntos diferentes de valores para los retrasos dentro y después de la sección crítica.



(a) null (c=0, d=0) (b) critical-section (c=3.64 µs, d=0) (c) delay (c=3.64 µs, d=1.29 µs)

Figura 6.53: Rendimiento de los cerrojos son la máquina SGI Origin2000, para tres escenarios diferentes.

Al igual que para el caso de memoria compartida, hasta que no se introducen retrasos entre secciones críticas el comportamiento de los cerrojos no equitativos presenta un mayor rendimiento. El retraso exponencial ayuda al LL-SC cuando existe una sección crítica nula, dado que en este caso es posible reducir la alta contención que se produce. En caso contrario, con secciones criticas nulas

el cerrojo basado en ticket escala pobremente, aunque en el caso de usar retraso exponencial escala muy bien. Los cerrojos basados en arrays también escalan muy bien. En el caso de existir coherencia de cachés, el mejor emplazamiento de las variables cerrojo en la memoria principal proporcionada por el cerrojo por colas software no es particularmente útil, y de hecho incurre en contención al realizar las operaciones de

comparación e intercambio (implementadas con LL-SC) y escala peor que el cerrojo basado en array. Si forzamos a los cerrojos a que tengan un comportamiento equitativo, su comportamiento es muy parecido al cerrojo basado en ticket sin retraso exponencial.

Si usamos una sección crítica no nula y un retraso entre los accesos al cerrojo (Figura 6.53(c)), suponiendo que todos los cerrojos se comportan equitativamente, el cerrojo LL-SC pierde su ventaja, mostrando su peor escalamiento. El cerrojo basado en array, el cerrojo por colas, y el cerrojo basado en ticket con retraso exponencial escalan muy bien. La mejor colocación de los datos del cerrojo por colas no importa, pero su contención no es peor que la del resto. En resumen, podemos concluir que el cerrojo basado en array y en ticket presentan un buen comportamiento y robustez para máquinas escalables con coherencia de caché, al menos al ser implementadas con LL-SC, y el cerrojo LL-SC con retraso exponencial se comporta mejor cuando no existe retraso entre la liberación y la siguiente adquisición, debido a su repetido comportamiento no equitativo. El cerrojo más sofisticado, el basado en cola, es innecesario pero también se comporta bien cuando existen retrasos entre la liberación y la siguiente adquisición.

6.6.2 Eventos de sincronización punto a punto (banderas)

La sincronización punto a punto dentro de un programa paralelo se implementa a menudo usando una espera activa sobre variables ordinarias que actúan como banderas (*flags*). Si queremos utilizar bloqueo en lugar de espera activa, podemos utilizar semáforos de una manera similar a la utilizada en la programación concurrente y en los sistemas operativos.

Las banderas son variables de control, usadas típicamente para comunicar que ha ocurrido un evento de sincronización más que para transferir valores. Si dos procesos tienen una relación productor/consumidor en la variable compartida a, entonces se puede utilizar una bandera para manejar la sincronización como se muestra a continuación:

```
P1                                  P2
a=f(x);        /*set a */          while (flag is 0) do nothing;
flag=1;                            b=g(a);              /* use a */
```

Si sabemos que la variable a se ha inicializado a un cierto valor, digamos 0, la cambiaremos a un nuevo valor si estamos interesados en este evento de producción, después podemos usar el propio a como variable de sincronización, como sigue:

```
P1                                  P2
a=f(x);        /*set a */          while (a is 0) do nothing;
                                   b=g(a);              /* use a */
```

Esto elimina la necesidad de una variable de bandera (*flag*) separado, y evita la escritura y la lectura de la variable.

6.6.3 Eventos globales de sincronización (barreras)

Una operación de sincronización común en los programas con bucles paralelos es la *barrera*. Una barrera fuerza a todos los procesos a esperar hasta que todos ellos alcanzan la barrera y entonces se permite la continuación de los mismos.

Algoritmos software

Los algoritmos software para las barreras se implementan usando cerrojos, contadores compartidos y banderas. Una implementación típica de la misma se puede conseguir con dos cerrojos: uno usado para proteger un contador que lleva la cuenta de los procesos que llegan a a la barrera y otro usado para bloquear a los procesos hasta que el último de ellos llega a la barrera. La figura muestra una implementación típica de este tipo de barrera, denominado *barrera centralizada*.

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count = count +1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) { /* all arrived */
       count=0;/* reset counter */
       release=1;/* release processes */
}
else { /* more to come */

       spin (release=1);/* wait for arrivals */
}
```

Figura 6.54: Código de una barrera centralizada.

En la práctica, se realiza una implementación de la barrera un poco más compleja. Frecuentemente una barrera se usa dentro de un bucle, de tal manera que los procesos salen de la barrera para realizar algún trabajo para después alcanzar la barrera de nuevo. Supongamos que uno de los procesos nunca deja la barrera (quedándose en la operación spin), lo que puede ocurrir si el SO decide en ese momento ejecutar otro proceso. Sería posible que uno de los procesos siguiera adelante y llegara a la barrera antes de que el último proceso la hubiera dejado. El proceso rápido atrapa al proceso lento en la barrera al resetear la bandera *release*. Ahora todos los procesos esperarán indefinidamente ya que el número de procesos nunca alcanzará el valor total. La observación importante es que el programador no hizo nada mal. Es la implementación de la barrera la que hizo suposiciones que no eran correctas. Una solución obvia es contar los procesos cuando salen de la barrera (al igual que hacemos en la entrada) y no permitir la reentrada de un proceso hasta que todos los procesos hayan dejado la instancia anterior de la barrera. Esto incrementaría significativamente la latencia de la barrera y la contención, que como veremos posteriormente ya es bastante grande. Una solución alternativa es usar una *sense-reversing barrier*, que hace uso de una variable privada por proceso, *local_sense*, que se inicializa a 1 para cada proceso. La figura 6.55 muestra el código para este tipo de barrera.

```
local_sense = ! local_sense; /*toggle local_sense*/
lock (counterlock);/* ensure update atomic */
count++;/* count arrivals */
unlock (counterlock);/* unlock */
if (count==total) { /* all arrived */
       count=0;/* reset counter */
       release=local_sense;/* release processes */
}
else { /* more to come */
       spin (release=local_sense);/*wait for signal*/
}
```

Figura 6.55: Código para una barrera sense-reversing.

Algoritmos para barreras en sistemas escalables

El problema de eventos globales como las barreras es un preocupación clave en las máquinas escalables. Una cuestión de considerable debate es si es necesario un soporte hardware especial para las operaciones globales o si son suficientes los algoritmos software. El CM-5 representa una parte del espectro, con un red de "control" especial que proporciona barreras, reducciones, broadcasts y otras operaciones globales sobre un subárbol de la máquina. Cray T3D proporciona soporte hardware para las barreras, mientras que T3E no lo hace. Dado que es sencillo construir barreras que iteren sobre variables locales, la mayoría de las máquinas escalables no proporcionan un soporte hardware especial para las barreras sino que se construyen en librerías software usando primitivas de intercambio o LL-SC.

En la barrera centralizada usada en la máquinas basadas en bus, todos los procesadores usan el mismo cerrojo para incrementar el mismo contador cuando indican su llegada a la barrera, y esperan sobre la misma variable de flag hasta que son liberados. En una máquina más grande, la necesidad de que todos los procesadores accedan al mismo cerrojo y lean y escriban de las mismas variables puede dar lugar a mucho tráfico y contención. De nuevo, esto es particularmente cierto en el caso de máquinas sin coherencia de cachés, donde la variable se puede convertir en un punto caliente de la red cuando varios procesadores iteran sobre ella.

Es posible implementar la llegada y la salida de la barrera de una manera más distribuida, en donde no todos los procesadores tienen que acceder a la misma variable o cerrojo. La coordinación de la llegada o liberación puede realizarse en fases o rondas, con subconjuntos de procesos que se coordinan entre ellos, de tal manera que después de algunas fases todos los procesos están sincronizados. La coordinación de diferentes subconjuntos puede realizarse en paralelo ya que no es necesaria mantener ninguna serialización entre ellos. En una máquina basada en bus, distribuir las acciones de coordinación no importa mucho ya que el bus serializa todas las acciones que requieren comunicación; sin embargo, puede ser muy importante en máquinas con memoria distribuida e interconectada donde diferentes subconjuntos pueden coordinarse en diferentes partes de la red. Veamos algunos de estos algoritmos distribuidos.

Árboles de combinación software. Una manera sencilla de coordinar la llegada o partida es a través de una estructura de árbol. Un árbol de llegada es un árbol que usan los procesadores para indicar su llegada a la barrera. Se reemplaza un único

cerrojo y contador por un árbol de contadores. El árbol puede ser de cualquier tipo, por ejemplo k-ario. En el caso más simple, cada hoja del árbol es un proceso que participa en la barrera. Cuando un proceso llega a la barrera, indica su llegada realizando una operación *fetch&increment* sobre el contador asociado a su padre. Después chequea que el valor devuelto por la operación para comprobar si es el último de los hermanos en llegar. Si no, simplemente espera a la liberación. En caso contrario, el proceso pasa a ser el representante de sus hermanos en el siguiente nivel del árbol y realiza una operación *fetch&increment* sobre el contador de ese nivel. De esta manera, cada nodo del árbol envía un único representante al primer nivel del árbol donde todos los procesos están representados por ese nodo hijo que ha llegado. Para un árbol de grado k, será necesario pasar por $log_k p$ niveles para completar notificación de llegada de los p procesos. Si los subárboles de procesadores están repartidos en diferentes partes de la red, y si las variables de los nodos del árbol están distribuidas de manera apropiada en la memoria, las operaciones *fetch&increment* sobre nodos que no tiene relación ascendiente-descendiente no necesitan serialización.

Se puede utilizar una estructura de árbol similar para la liberación, de tal manera que los procesadores no realicen una espera activa sobre la misma variable. Es decir, el último proceso en llegar a la barrera pone a uno la variable de liberación asociada con la raíz del árbol, sobre la cual están realizando una espera activa $k-1$ procesos. Cada uno de los k procesos pone a uno la variable de liberación del siguiente nivel del árbol sobre la que están realizando espera activa $k-1$ procesos, y así bajando en el árbol hasta que todos los procesos se liberan. La longitud del camino crítico de la barrera en términos del número de operaciones dependientes o serializadas (es decir, transacciones de red) es por tanto $O(log_k p)$, frente al $O(p)$ para una la barrera centralizada u $O(p)$ para cualquier barrera sobre un bus centralizado.

Barreras en árbol con iteración local. Existen dos formas de asegurarse de que un procesador itere sobre una variable local en el caso de que. Una es predeterminar qué procesador se moverá desde un nodo a su padre en el árbol basándose en el identificador del proceso y el número de procesadores que participan en la barrera. En este caso, un árbol binario facilita la implementación de la iteración local, ya que la variable sobre la que se itera puede almacenarse en la memoria local del proceso en espera en lugar de aquel que pasa al nivel del padre. De hecho, en este caso es posible implementar la barrera sin operaciones atómicas, usando únicamente operaciones de lectura y escritura como sigue. En la llegada, un proceso que llega a cada nodo simplemente itera sobre la variable de llegada asociada a ese nodo. El otro proceso asociado con el nodo simplemente escribe en la variable al llegar. El proceso cuyo papel sea iterar ahora simplemente itera sobre la variable liberada asociada a ese nodo, mientras que el otro proceso procede a ir al nodo padre. A este tipo de barrera de árbol binario se le denomina "barrera por torneo", dado que uno de los procesos puede pensarse como el perdedor del torneo en cada etapa del árbol de llegada.

Otra forma de asegurar la iteración local es usar árboles p-nodo para implementar una barrera entre los p procesos, donde cada nodo del árbol (hoja o interno) se asigna a un único proceso. Los árboles de llegada y la activación pueden ser los mismos, o pueden mantenerse como árboles con diferentes factores de ramaje. Cada nodo interno (proceso) en el árbol mantiene un array de variables de llegada, con una entrada por hijo, almacenadas en la memoria local del nodo. Cuando un proceso llega a la barrera, si su nodo en el árbol no es una hoja entonces comprueba en primer lugar su array de flags de llegada y espera hasta que todos sus hijos hayan indicado su llegada poniendo a uno su correspondiente entrada en el array. A continuación, pone a uno su entrada

en el array de llegada de su padre (remoto) y realiza una espera activa sobre el flag de liberación asociado a su nodo en el árbol de liberación. Cuando llega el proceso raíz y todas los flags de llegada de su array están activos, entonces todos los procesos han llegado. La raíz pone a uno los flags (remotos) de liberación de todos sus hijos en el árbol de liberación; estos procesos salen de su espera activa y ponen a uno los flags de sus hijos, y así hasta que todos los procesos se liberan.

Rendimiento

Las metas a alcanzar para una barrera son las misma que las que vimos para los cerrojos:

Baja latencia (*longitud del camino crítico pequeña*). Ignorando la contención, nos gustaría que el número de operaciones en la cadena de operaciones dependientes necesarias para p procesadores para pasar la barrera sea pequeña.

Bajo tráfico. Dado que las barreras son operaciones globales, es muy probable que muchos procesadores intenten ejecutar la barrera al mismo tiempo. Nos gustaría que el algoritmo reduzca el número de transacciones y por tanto una posible contención.

Escalabilidad. En relación con el punto anterior, nos gustaría que una barrera escale bien en función del número de procesadores que vamos a usar.

Bajo coste de almacenamiento. La información que es necesaria debe ser pequeña y no debe aumentar más rápidamente que el número de procesadores.

Imparcialidad. Esta meta no es importante en este caso ya que todos los procesadores se liberan al mismo tiempo, pero nos gustaría asegurar que el mismo procesador no sea el último en salir de la barrera, o preservar una ordenación FIFO.

En una barrera centralizada, cada procesador accede a la barrera una vez, de tal manera que el camino crítico tiene longitud de al menos p. Consideremos el tráfico del bus. Para completar sus operaciones, una barrera centralizada de p procesadores realiza $2p$ transacciones de bus para obtener el cerrojo e incrementar el contador, dos transacciones de bus para que el último procesador ponga a cero el contador y libere la bandera, y otras $p - 1$ transacciones para leer la bandera después de que su valor haya sido invalidado. Obsérvese que éste es mejor que el tráfico para que un cerrojo sea adquirido por p procesadores, dado que en ese caso cada una de las p liberaciones dan lugar a una invalidación dando lugar a $O(p^2)$ transacciones de bus. Sin embargo, la contención resultante puede ser sustancial si muchos procesadores llegan a la barrera de forma simultánea. En la figura 6.56 se muestra el rendimiento de algunos algoritmos de barrera.

Mejora de los algoritmos de barrera en un bus

Veamos si es posible obtener una barrera con un mejor comportamiento para un bus. Una parte del problema de la barrera centralizada es que todos los procesos compiten por el mismo cerrojo y variables de bandera. Para manejar esta situación, podemos

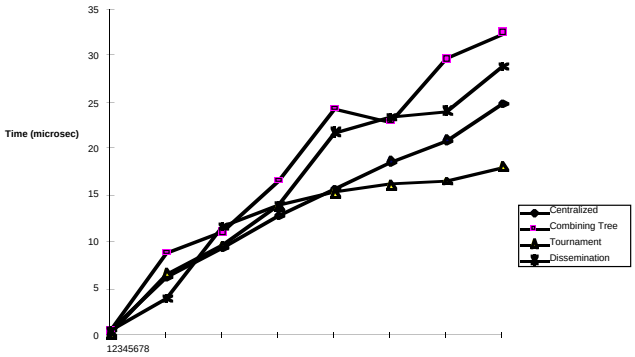


Figura 6.56: Rendimiento de algunas barreras en el SGI Challenge.

construir barreras que hagan que menos procesadores compitan por la misma variable. Por ejemplo, los procesadores pueden indicar su llegada a la barrera a través de un árbol de combinación software. En un árbol de combinación binario, únicamente dos procesadores se intercomunican su llegada a cada nodo del árbol, y únicamente uno de los dos se mueve hacia arriba para participar en el siguiente nivel del árbol. De esta forma, únicamente dos procesadores acceden a una variable dada. En una red con múltiples caminos paralelos, como los existentes en la máquinas con gran número de procesadores, un árbol de combinación se comporta mucho mejor que una barrera centralizada ya que los diferentes pares de procesadores se pueden comunicar entre ellos en paralelo por diferentes partes de la red. Sin embargo, en una interconexión centralizada como un bus, aunque cada par de procesadores se comuniquen a través de variables diferentes todos ellos generan transacciones y por tanto contención sobre el mismo bus. Dado que un árbol binario con p hojas tiene aproximadamente $2p$ nodos, un árbol de combinación necesita un número similar de transacciones de bus que la barrera centralizada. Además presenta mayor latencia ya que necesita $log\, p$ pasos para llegar de las hojas a la raíz del árbol, y de hecho del orden de p transacciones de bus serializadas. La ventaja de un árbol de combinación es que no usa cerrojos sino operaciones de lectura y escritura, que puede compensar si el número de procesadores en grande. El resto de los algoritmos de barrera escalables los estudiaremos, junto con las barreras en árbol, en el contexto de las máquinas escalables.

6.7 Conclusiones

Los procesadores simétricos de memoria compartida son una extensión natural de las estaciones de trabajo uniprocesador y de los PCs. Las aplicaciones secuenciales pueden ejecutarse sin necesidad de modificación, y beneficiándose incluso en el rendimiento al tener el procesador por una fracción de tiempo mayor y de

una mayor cantidad de memoria principal compartida y capacidades de E/S típicamente disponibles en estas máquinas. Las aplicaciones paralelas son también fácilmente portables a este entorno ya que todos los datos locales y compartidos son directamente accesibles por todos los procesadores utilizando operaciones de lectura y escritura ordinarias. Las porciones de cálculo intensivo de una aplicación secuencial pueden ser paralelizadas de forma selectiva. Una ventaja clave de las cargas de multiprogramación es la granularidad fina con la que se pueden compartir los recursos entre los distintos procesos. Esto es cierto tanto en lo temporal, en donde procesadores y/o páginas de la

memoria principal pueden ser reasignadas frecuentemente entre los diferentes procesos, como en lo físico, en donde la memoria principal puede dividirse entre las aplicaciones a nivel de página. Debido a estas características, los principales vendedores de ordenadores, desde las estaciones de trabajo suministradas por SUN, Silicon Graphics, Hewlett Packard, Digital e IBM hasta los suministradores de PCs como Intel y Compaq están produciendo y vendiendo este tipo de máquinas. De hecho, para algunos de los vendedores de estaciones de trabajo este mercado constituye una fracción substancial de sus ventas, y una mayor fracción de sus beneficios netos debido a los mayores márgenes existentes en estas máquinas.

El reto clave en el diseño de estas máquinas es la organización y la implementación del subsistema de memoria compartida usado como comunicación entre los procesos además de manejar el resto de accesos a memoria. La mayoría de las máquinas de pequeña escala que se encuentran en la actualidad usan un bus como interconexión. En este caso el reto está en mantener coherentes en las cachés privadas de los procesadores los datos compartidos. Existen una gran cantidad de opciones disponibles, como hemos discutido, incluyendo el conjunto de estados asociados a los bloques de la caché, la elección del tamaño del bloque, y si usar invalidación o actualización. La tarea clave de la arquitectura del sistema es realizar las elecciones oportunas en función de los patrones de compartición de datos predominantes en las aplicaciones para las que se utilizará la máquina y de su facilidad de implementación.

Conforme progresan los procesadores, el sistema de memoria, los circuitos integrados y la tecnología de empaquetamiento, existen tres importantes razones que nos hace pensar que los multiprocesadores de pequeña escala continuarán siendo importantes en un futuro. La primera es que ofrecen un buen coste/rendimiento que hará que individuos o pequeños grupos de personas pueden utilizarlos como un recurso compartido o un servidor de cómputo o ficheros. La segunda es que los microprocesadores actuales están diseñados para poder ser utilizados como multiprocesadores por lo que ya no existe un tiempo de espera significativo entre el último microprocesador y su incorporación a un multiprocesador. La tercera razón es que la tecnología software para máquinas paralelas (compiladores, sistemas operativos, lenguajes de programación) está madurando rápidamente para máquinas de memoria compartida de pequeña escala, aunque todavía le queda un largo camino en el caso de máquinas de gran escala. Por ejemplo, la mayoría de los vendedores de sistemas de ordenadores tienen una versión paralela de sus sistemas operativos para sus multiprocesadores basados en bus.

Capítulo 7

Multicomputadores

Los multiprocesadores de memoria compartida presentan algunas desventajas como por ejemplo:

1. Son necesarias técnicas de sincronización para controlar el acceso a las variables compartidas
2. La contención en la memoria puede reducir significativamente la velocidad del sistema.
3. No son fácilmente ampliables para acomodar un gran número de procesadores.

Un sistema multiprocesador alternativo al sistema de memoria compartida que elimina los problemas arriba indicados es tener únicamente una memoria local por procesador eliminando toda la memoria compartida del sistema. El código para cada procesador se carga en la memoria local al igual que cualquier dato que sea necesario. Los programas todavía están divididos en diferentes partes, como en el sistema de memoria compartida, y dichas partes todavía se ejecutan concurrentemente por procesadores individuales. Cuando los procesadores necesitan acceder a la información de otro procesador, o enviar información a otro procesador, se comunican enviando mensajes. Los datos no están almacenados globalmente en el sistema; si más de un proceso necesita un dato, éste debe duplicarse y ser enviado a todos los procesadores peticionarios. A estos sistemas se les suele denominar multicomputadores.

La arquitectura básica de un sistema multiprocesador de paso de mensajes se muestra en la figura 7.1. Un multiprocesador de paso de mensajes consta de nodos, que normalmente están conectados mediante enlaces directos a otros pocos nodos. Cada nodo está compuesto por un procesador junto con una memoria local y canales de comunicación de entrada/salida. No existen localizaciones de memoria global. La memoria local de cada nodo sólo puede ser accedida por el procesador de dicho nodo. Cada memoria local puede usar las mismas direcciones. Dado que cada nodo es un ordenador autocontenido, a los multiprocesadores de paso de mensajes se les suelen denominar multicomputadores.

El número de nodos puede ser tan pequeño como 16 (o menos), o tan grande como varios millares (o más). Sin embargo, la arquitectura de paso de mensajes muestra sus ventajas sobre los sistemas de memoria compartida cuando el número de procesadores es grande. Para sistemas multiprocesadores pequeños, los sistemas de memoria compartida presentarán probablemente un mejor rendimiento y mayor flexibilidad. El número de canales físicos entre nodos suele oscilar entre cuatro y ocho. La principal ventaja de

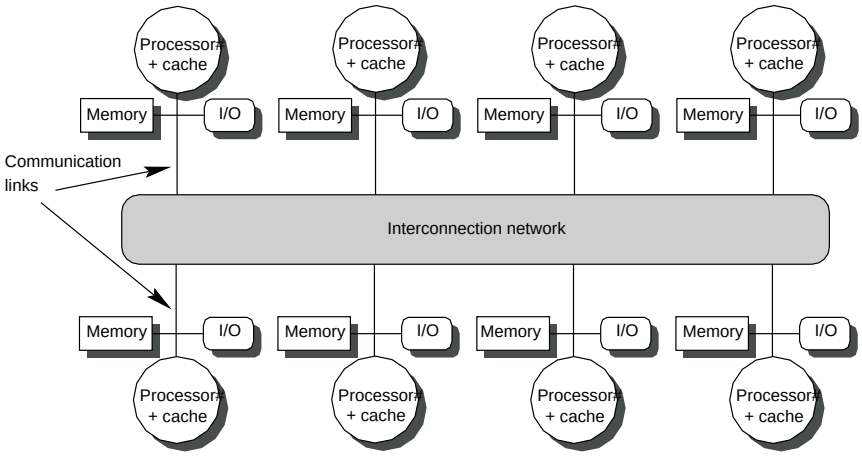


Figura 7.1: Arquitectura básica de un multicomputador.

esta arquitectura es que es directamente escalable y presenta un bajo coste para sistemas grandes. Encaja en un diseño VLSI, con uno o más nodos fabricados en un chip, o en pocos chips, dependiendo de la cantidad de memoria local que se proporcione.

Cada nodo ejecuta uno o más *procesos*. Un proceso consiste a menudo en un código secuencial, como el que se encontraría en un ordenador von Neumann. Si existe más de un proceso en un procesador, éste puede eliminarse del planificador cuando está a la espera de enviar o recibir un mensaje, permitiendo el inicio de otro proceso. Se permite el paso de mensajes entre los distintos procesos de un procesador mediante el uso de canales internos. Los mensajes entre procesos pertenecientes a diferentes procesadores se pasan a través de canales externos usando los canales físicos de comunicación existentes entre los procesadores.

Idealmente, los procesos y los procesadores que ejecutan dichos procesos pueden ser vistos como entidades completamente separadas. Un problema se describe como un conjunto de procesos que se comunican entre sí y que se hacen encajar sobre una estructura física de procesadores. El conocimiento de la estructura física y de la composición de los nodos es necesaria únicamente a la hora de planificar una ejecución eficiente.

El tamaño de un proceso viene determinado por el programador y puede describirse por su *granularidad*, que puede expresarse de manera informal como la razón:

$$Granularidad = \frac{Tiempo\ de\ cálculo}{Tiempo\ de\ comunicación}$$

o mediante los términos:

1. Granularidad gruesa.
2. Granularidad media.
3. Granularidad fina.

En la granularidad gruesa, cada proceso contiene un gran número de instrucciones secuenciales que tardan un tiempo sustancial en ejecutarse. En la granularidad fina, un proceso puede tener unas pocas instrucciones, incluso una instrucción; la granularidad media describe el terreno intermedio entre ambos extremos. Al reducirse la granulari-

dad, la sobrecarga de comunicación de los procesos suele aumentar. Es particularmente deseable reducir la sobrecarga de comunicación debido a que el coste de la misma suele ser bastante alto. Por ello, la granularidad empleada en este tipo de máquinas suele ser media o gruesa.

Cada nodo del sistema suele tener una copia del núcleo de un sistema operativo. Este sistema operativo se encarga de la planificación de procesos y de realizar las operaciones de paso de mensajes en tiempo de ejecución. Las operaciones de encaminamiento de los mensajes suelen estar soportadas por hardware, lo que reduce la latencia de las comunicaciones. Todo el sistema suele estar controlado por un ordenador anfitrión.

Existen desventajas en los sistemas multicomputador. El código y los datos deben de transferirse físicamente a la memoria local de cada nodo antes de su ejecución, y esta acción puede suponer una sobrecarga considerable. De forma similar, los resultados deben de transferirse de los nodos al sistema anfitrión. Claramente los cálculos a realizar deben ser lo suficientemente largos para compensar este inconveniente. Además, el programa a ejecutar debe ser intensivo en cálculo, no intensivo en operaciones de entrada/salida o de paso de mensajes. El código no puede compartirse. Si los procesos van a ejecutar el mismo código, lo que sucede frecuentemente, el código debe duplicarse en cada nodo. Los datos deben pasarse también a todos los nodos que los necesiten, lo que puede dar lugar a problemas de incoherencia. Estas arquitecturas son generalmente menos flexibles que las arquitecturas de memoria compartida. Por ejemplo, los multiprocesadores de memoria compartida pueden emular el paso de mensajes utilizando zonas compartidas para almacenar los mensajes, mientras que los multicomputadores son muy ineficientes a la hora de emular operaciones de memoria compartida.

7.1 Redes de interconexión para multicomputadores

Antes de ver las redes más comunes en sistemas multicomputadores, conviene repasar los distintos tipos de redes utilizadas en sistemas paralelos. La figura 7.2 muestra una clasificación que integra la mayoría de las redes comúnmente utilizadas en sistemas de altas prestaciones.

Las redes directas, también llamadas estáticas, son las más extendidas para la comunicación entre los elementos de un multicomputador. Las redes estáticas utilizan enlaces directos que son fijos una vez construida la red. Este tipo de red viene mejor para construir ordenadores donde los patrones de comunicación son predecibles o realizables mediante conexiones estáticas. Se describen a continuación las principales topologías estáticas en términos de los parámetros de red comentando sus ventajas en relación a las comunicaciones y la escalabilidad.

La mayoría de las redes directas implementadas en la práctica tienen una topología ortogonal. Una topología se dice *ortogonal* si y sólo si los nodos pueden colocarse en un espacio ortogonal n-dimensional, y cada enlace puede ponerse de tal manera que produce un desplazamiento en una única dimensión. Las topologías ortogonales pueden clasificarse a su vez en estrictamente ortogonales y débilmente ortogonales. En una topología *estrictamente ortogonal*, cada nodo tiene al menos un enlace cruzando cada dimensión. En una topología *débilmente ortogonal*, algunos de los nodos pueden no tener enlaces en alguna dimensión.

- Redes de medio compartido
  - Redes de área local
    - Bus de contención (Ethernet)
    - Bus de tokens (Arenet)
    - Anillo de tokens (FDDI Ring, IBM Token Ring)
  - Bus de sistema (Sun Gigaplane, DEC AlphaServer8X00, SGI PoweerPath-2)
- Redes directas (Redes estáticas basadas en encaminador)
  - Topologías estrictamente ortogonales
    - Malla
      - Malla 2-D (Intel Paragon)
      - Malla 3-D (MIT J-Machine)
    - Toros (n-cubo k-arios)
      - Toro 1-D unidireccional o anillo (KSR forst-level ring)
      - Toro 2-D bidireccional (Intel/CMU iWarp)
      - Toro 2-D bidireccional (Cray T3D, Cray T3E)
    - Hipercubo (Intel iPSC, nCUBE)
  - Otras topologías directas: Árboles, Ciclos cubo-conectados, Red de Bruijn, Grafos en Estrella, etc.
- Redes Indirectas (Redes dinámicas basadas en conmutadores)
  - Topologías Regulares
    - Barra cruzada (Cray X/Y-MP, DEC GIGAswitch, Myrinet)
    - Redes de Interconexión Multietapa (MIN)
      - Redes con bloqueos
        MIN Unidireccionales (NEC Cenju-3, IBM RP3)
        MIN Bidireccional (IBM SP, TMC CM-5, Meiko CS-2)
      - Redes sin bloqueos: Red de Clos
  - Topologías Irregulares (DEC Autonet, Myrinet, ServerNet)
- Redes Híbridas
  - Buses de sistema múltiples (Sun XDBus)
  - Redes jerárquicas (Bridged LANs, KSR)
    - Redes basadas en Agrupaciones (Stanford DASH, HP/Convex Exemplar)
  - Otras Topologías Hipergrafo: Hiperbuses, Hipermallas, etc.

Figura 7.2: Clasificación de las redes de interconexión

7.1.1 Topologías estrictamente ortogonales

La característica más importante de las topologías estrictamente ortogonales es que el encaminamiento es muy simple. En efecto, en una topología estrictamente ortogonal los nodos pueden enumerarse usando sus coordenadas en el espacio n-dimensional. Dado que cada enlace atraviesa una única dimensión y que cada nodo tiene al menos un enlace atravesando cada dimensión, la distancia entre dos nodos puede calcularse como la suma de la diferencia en las dimensiones. Además, el desplazamiento a lo largo de un enlace dado sólo modifica la diferencia dentro de la dimensión correspondiente. Teniendo en cuenta que es posible cruzar cualquier dimensión desde cualquier nodo en la red, el encaminamiento se puede implementar fácilmente seleccionando un enlace que decremente el valor absoluto de la diferencia de alguna de las dimensiones. El conjunto de dichas diferencias pueden almacenarse en la cabecera del paquete, y ser actualizada (añadiendo o substrayendo una unidad) cada vez que el paquete se encamina en un nodo intermedio. Si la topología no es estrictamente ortogonal, el encaminamiento se complica.

Las redes directas más populares son las *mallas n-dimensionales*, los *n-cubos k-arios* o *toros*, y los *hipercubos*. Todas ellas son estrictamente ortogonales en su definición aunque hay ciertas variantes que no son estrictamente ortogonales pero se las engloba en el mismo grupo.

Mallas

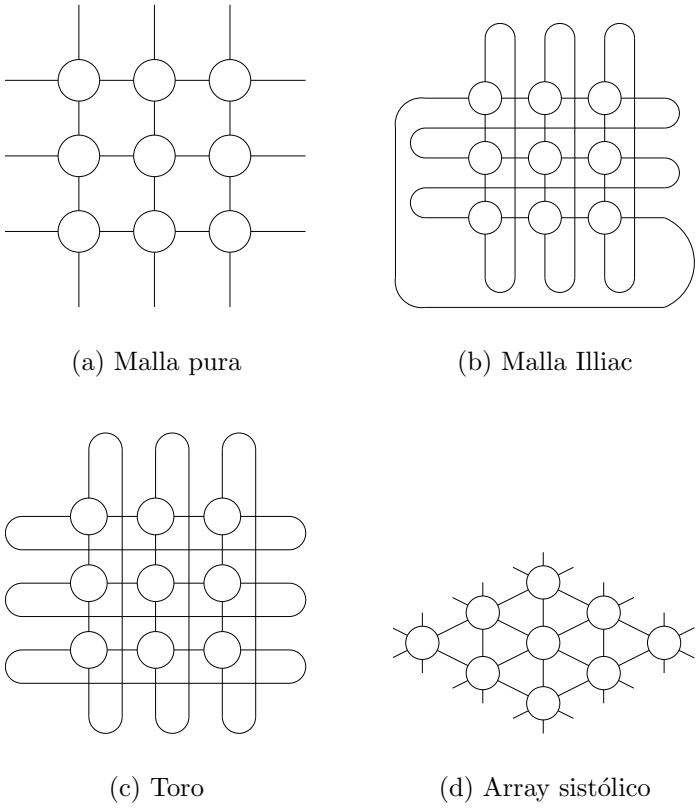
Formalmente, un malla n-dimensional tiene $k_0 \times k_1 \times \cdots \times k_{n-2} \times k_{n-1}$ nodos, k_i nodos en cada dimensión i, donde $k_i \geq 2$ y $0 \leq i \leq n-1$. Cada nodo X está definido por sus n coordenadas, $(x_{n-1}, x_{n-2}, \ldots, x_1, x_o)$, donde $0 \leq x_i \leq k_i - 1$ para $0 \leq i \leq n-1$. Dos nodos X e Y son vecinos si y sólo si $y_i = x_i$ para todo i, $0 \leq i \leq n-1$, excepto uno, j, donde $y_j = x_j \pm 1$. Así, los nodos pueden tener de n a $2n$ vecinos, dependiendo de su localización en la red. Por tanto, esta topología no es regular.

La figura 7.3(a) muestra una malla de 3×3. La estructura de malla, con algunas variaciones, se ha venido empleando en varios computadores comerciales, especialmente para la interconexión de procesadores en SIMD o masivamente paralelos.

En general, una malla k-dimensional, con $N = n^k$ nodos, tiene un grado de nodo interior de $2k$ y el diámetro de la red es $d = k(n-1)$. Hay que hacer notar que la malla pura, mostrada en la figura 7.3(a) no es simétrica y que los nodos en los bordes pueden ser 2 y 3 mientras que en el interior son siempre 4.

La figura 7.3(b) muestra una variación de una malla o toro en la que se permiten conexiones largas entre los nodos en los bordes. El Illiac IV tiene una malla con este principio ya que se trata de una malla de 8×8 con un grado nodal constante de 4 y un diámetro de 7. La malla Illiac es topológicamente equivalente a un anillo acorde de grado 4 como se muestra en la figura 7.7(c) para una configuración de $N = 9 = 3 \times 3$.

En general, una malla Illiac $n \times n$ debería tener un diámetro $d = n - 1$, que es sólo la mitad del diámetro de una malla pura. El *toro* de la figura 7.3(c) se puede ver como otra variante de la malla aunque se trata en realidad de un 2-cubo ternario, es decir, pertenece al grupo de las redes n-cubo k-arias que se muestran a continuación.



(a) Malla pura (b) Malla Illiac

(c) Toro (d) Array sistólico

Figura 7.3: Variaciones de mallas y toros.

Redes n-cubo k-arias

En un n-cubo k-ario, todos los nodos tienen el mismo número de vecinos. La definición del n-cubo k-ario difiere de la de malla n-dimensional en que todos los k_i son iguales a k y dos nodos X e Y son vecinos si y sólo si $y_i = x_i$ para todo i, $0 \leq i \leq n-1$, excepto uno, j, donde $y_j = (x_j \pm 1)\ mod\ k$. El cambio a aritmética modular en la dirección añade el canal entre el nodo $k-1$ y el 0 en cada dimensión en los n-cubos k-arios, dándole regularidad y simetría. Cada nodo tiene n vecinos si $k = 2$ y $2n$ vecinos si $k > 2$. Cuando $n = 1$ el n-cubo k-ario se colapsa en un anillo bidireccional con k nodos.

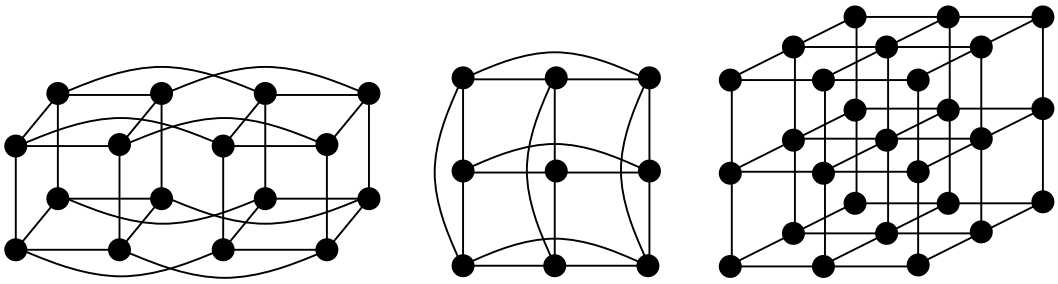
A las redes n-cubo k-aria también se les suele llamar toros. En general, un toro $n \times n$ binario tiene un grado nodal de 4 y un diámetro $d = 2\lfloor n/2 \rfloor$. El toro es una topología simétrica. Todas las conexiones de contorno añadidas ayudan a reducir el diámetro de la malla pura a la mitad.

Hipercubos

Otra topología con regularidad y simetría es el hipercubo, que es un caso particular de mallas n-dimensionales o n-cubo k-ario. Un hipercubo es una malla n-dimensional con $k_i = 2$ para $0 \leq i \leq n-1$, o un n-cubo binario.

La figura 7.4(a) muestra un hipercubo con 16 nodos. La parte (b) de dicha figura ilustra un 2-cubo ternario o toro bidimensional (2-D). La figura 7.4(c) muestra una

malla tridimensional.



(a) 2−ary 4−cube (hypercube) (b) 3−ary 2−cube (c) 3−ary 3−D mesh

Figura 7.4: Topologías estrictamente ortogonales en una red directa.

En la figura 7.5 se muestran diversas topologías del tipo n-cubo k-aria, incluyendo hipercubos de 3 y 4 dimensiones. También en esta figura se ha incluido el ciclo cubo-conectado que es un tipo especial de red directa no estrictamente ortogonal pero basada en el hipercubo de tres dimensiones.

7.1.2 Otras topologías directas

Aparte de las topologías definidas arriba, se han propuesto muchas otras topologías en la literatura. La mayoría de ellas fueron propuestas con la meta de minimizar el diámetro de la red para un número dado de nodos y grado del nodo. Como veremos en secciones posteriores, para estrategias de conmutación encauzadas, la latencia es casi insensible al diámetro de la red, especialmente cuando los mensajes son largos. Así que es poco probable que esas topologías lleguen a ser implementadas. En los siguientes párrafos, presentaremos una descripción informal de algunas topologías de red directas relevantes.

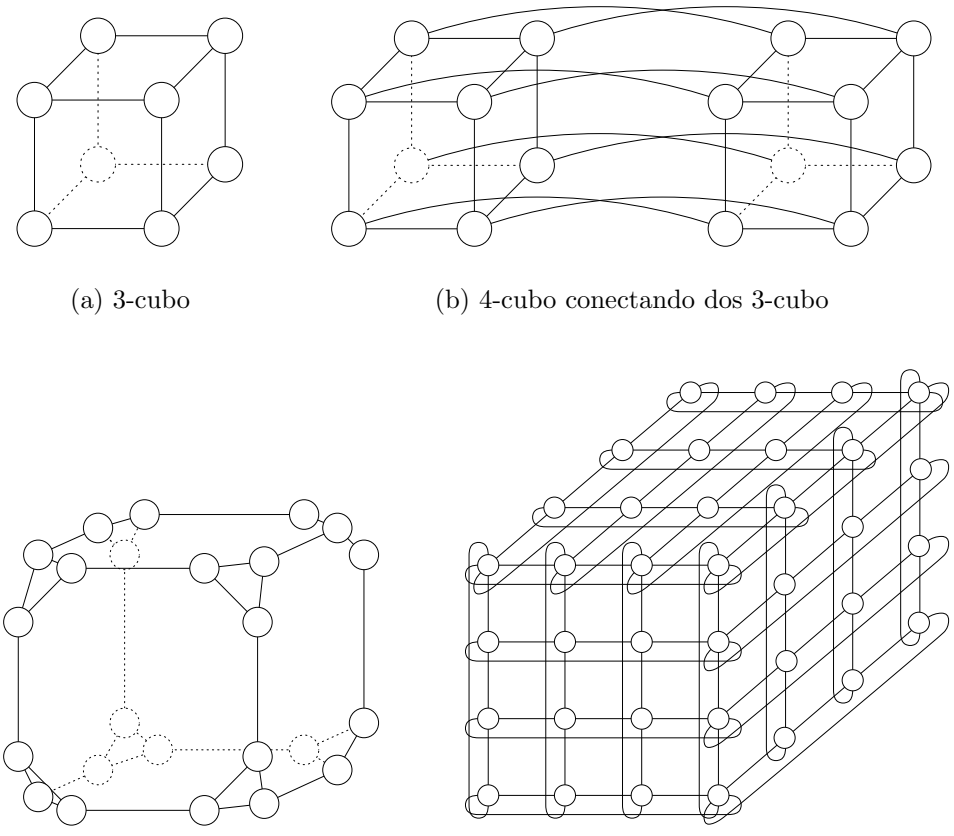
Las topologías que se presentan a continuación tienen diferentes objetivos: algunas intentan simplificar la red a costa de un mayor diámetro, otras pretenden disminuir el diámetro y aumentando los nodos manteniendo un compromiso de complejidad no muy elevada.

La figura 7.6 muestra otras topologías propuestas para reducir el grado del nodo a la vez que se mantiene un diámetro bajo.

Matriz lineal

Esta es una red monodimensional en la cual N nodos están conectados mediante $N-1$ enlaces tal y como se muestra en la figura 7.7(a). Los nodos internos son de grado 2 mientras que los terminales son de grado 1. El diámetro es $N-1$ que es excesivo si N es grande. La anchura biseccional b es 1. Estas matrices lineales son los más simples de implementar. La desventaja es que no es simétrica y que la comunicación es bastante ineficiente a poco que N sea algo grande, con $N=2$ el sistema está bien, pero para N más grande hay otras topologías mejores.

No hay que confundir la matriz lineal con el *bus* que es un sistema de tiempo compartido entre todos los nodos pegados a el. Una matriz lineal permite la utilización



(a) 3-cubo

(b) 4-cubo conectando dos 3-cubo

(c) 3 ciclo cubo conectado

(d) n-cubo k-ario con $k = 4$ y $n = 3$

Figura 7.5: Hipercubos, ciclo cubos y n-cubos k-arios.

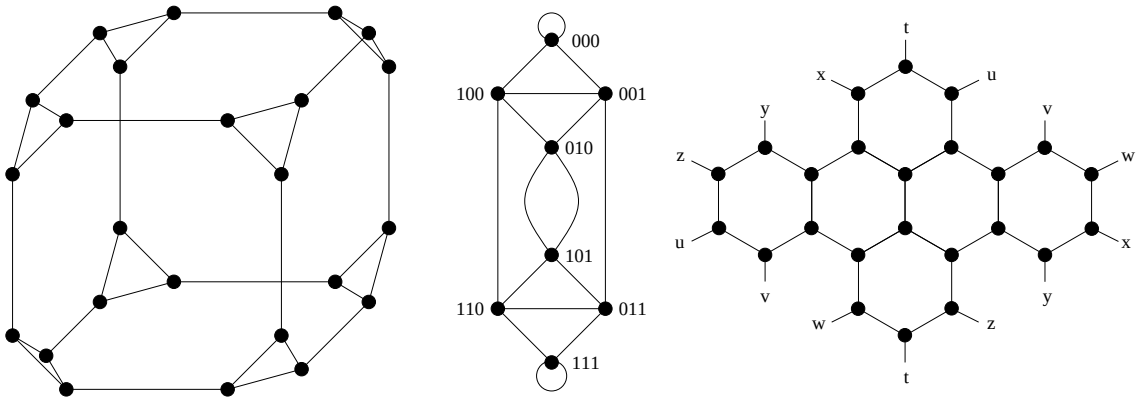
simultánea de diferentes secciones (canales) de la matriz con diferentes orígenes y destinos.

Anillo

Un *anillo* se obtiene conectando los dos nodos terminales de una matriz lineal mediante un enlace más como se muestra en la figura 7.7(b). El anillo puede ser unidireccional (diámetro $N - 1$) o bidireccional (diámetro $N/2$). El anillo es simétrico con todos los nodos de grado 2 constante.

Este tipo de topologías se han utilizado en el pasado como en el IBM *token ring* donde los mensajes circulaban hasta encontrar un nodo con un *token* que coincida con el mensaje. También en otros sistemas se ha utilizado para la comunicación entre procesadores por el paso de paquetes.

Si añadimos más enlaces a cada nodo (aumentamos el grado del nodo), entonces obtenemos *anillos acordes* como se muestra en la figura 7.7(c). Siguiendo esto se pueden ir añadiendo enlaces a los nodos aumentando la conectividad y reduciendo el diámetro de la red. En el extremo podemos llegar a la *red completamente conectada* en la cual los nodos son de grado $N - 1$ y el diámetro es 1.



(a) Cube-Connected Cycles (b) De Bruijn Network (c) Star Graph

Figura 7.6: Otras topologías directas.

Desplazador de barril

El desplazador de barril *barrel shifter* se obtiene a partir de la configuración en anillo, añadiendo a cada nodo enlaces con los nodos que tengan una distancia potencia de 2 como se muestra en la figura 7.7(d). Esto implica que cada nodo i está conectado con un nodo j si se cumple que $|j - i| = 2^r$ para algún $r = 0, 1, \ldots, n-1$ siendo el tamaño de la red $N = 2^n$. El grado de cada nodo es $d = 2n - 1$ y un diámetro $D = n/2$.

Naturalmente, la conectividad del desplazador de barril es mayor que la de cualquier anillo acorde con un grado de nodo menor. Suponiendo $N = 16$, el desplazador de barril tiene un grado de nodo igual a 7 con un diámetro de 2. A pesar de todo la complejidad del desplazador de barril es menor que la del anillo completamente conectado.

Ciclo Cubo Conectado

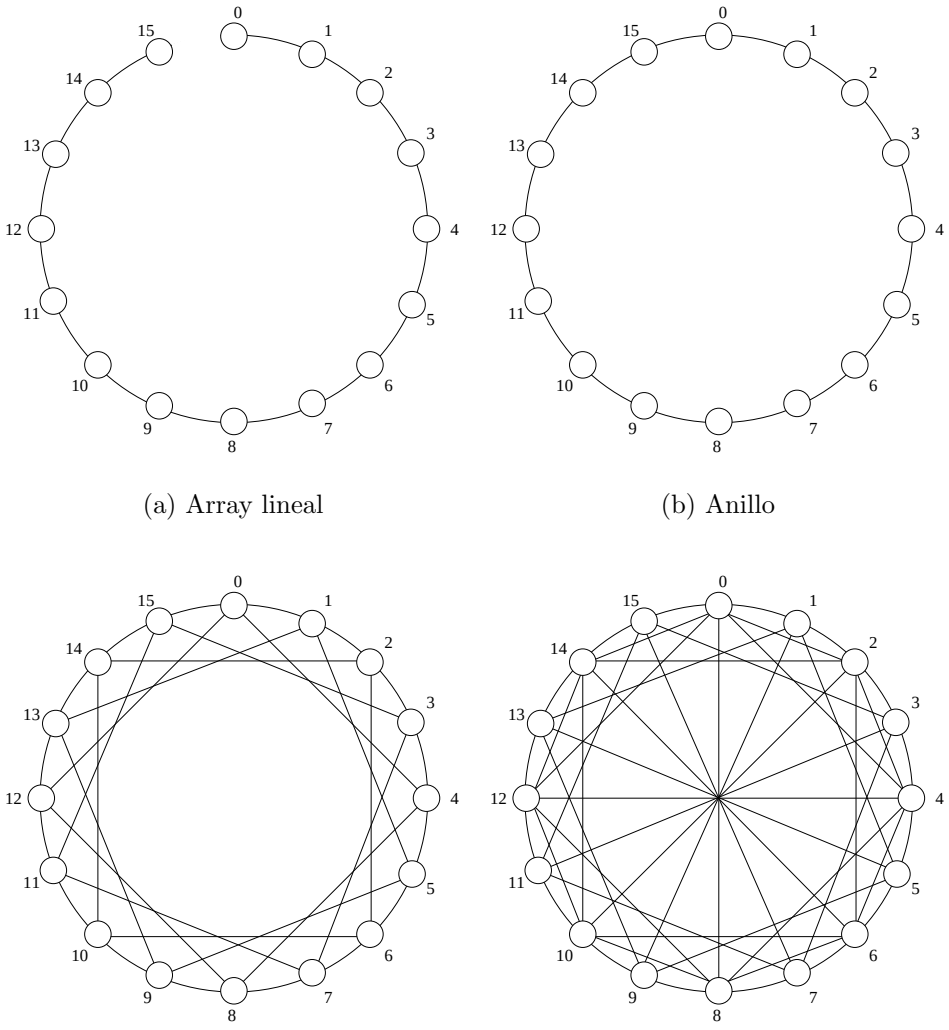
Esta arquitectura se realiza a partir del hipercubo y consiste en sustituir cada vértice del cubo por un anillo (ciclo) normalmente con un número igual de nodos que de dimensiones del cubo. Las figuras 7.5(c) y 7.6(a) muestran un 3-ciclo cubo conectado (un 3-CCC).

En general uno puede construir un k-ciclo cubo conectado a partir de un k-cubo con $n = 2^k$ ciclos. La idea es reemplazar cada vértice del hipercubo k-dimensional por un anillo de k nodos. Un k-cubo puede ser transformado entonces en un k-CCC con $k \times 2^k$ nodos.

El diámetro de un k-CCC es $2k$, es decir, el doble que el del hipercubo. La mejora de esta arquitectura está en que el grado de nodo es siempre 3 independientemente de la dimensión del hipercubo, por lo que resulta una arquitectura escalable.

Supongamos un hipercubo con $N = 2^n$ nodos. Un CCC con el mismo número N de nodos se tendría que construir a partir de un hipercubo de menor dimensión tal que $2^n = k \cdot 2^k$ para algún $k < n$.

Por ejemplo, un CCC de 64 nodos se puede realizar con un 4-cubo reemplazando los vértices por ciclos de 4 nodos, que corresponde al caso $n = 6$ y $k = 4$. El CCC tendría un diámetro de $2k = 8$ mayor que 6 en el 6-cubo. Pero el CCC tendría un grado



(a) Array lineal

(b) Anillo

(c) Anillo acorde de grado 4 (coincide con la malla del Illiac)

(d) Desplazador de barril

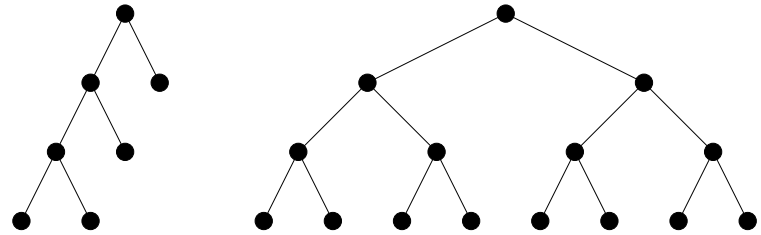
Figura 7.7: Matriz lineal, anillos, y barril desplazado.

nodal de 3, menor que 6 en el 6-cubo. En este sentido, el CCC es una arquitectura más conveniente para fabricar sistemas escalables siempre que la mayor latencia pueda ser tolerada de alguna manera.

Árbol, árbol grueso y estrella

Una topología popular es el *árbol.* Esta topología tiene un nodo *raíz* conectado a un cierto número de nodos descendientes. Cada uno de estos nodos se conecta a la vez a un conjunto disjunto (posiblemente vacío) de descendientes. Un nodo sin descendientes es un nodo *hoja.* Una propiedad característica de los árboles es que cada nodo tiene un único padre. Por tanto, los árboles no tienen ciclos. Un árbol en el cual cada nodo menos las hojas tiene un número k fijo de descendientes es un árbol k-ario. Cuando la distancia entre cada nodo hoja y la raíz es la misma, es decir, todas las ramas del árbol

tienen la misma longitud, el árbol está *balanceado*. Las figura 7.8(a) y 7.8(b) muestran un árbol binario no balanceado y balanceado, respectivamente.



(a) Binary Tree (b) Balanced Binary Tree

Figura 7.8: Algunas topologías árbol.

La desventaja más importante de los árboles como redes de interconexión en general es que el nodo raíz y los nodos cerca de él se convierten en cuellos de botella. Además, no existen caminos alternativos entre cualquier par de nodos. El cuello de botella puede eliminarse utilizando canales con mayor ancho de banda en los canales cercanos al nodo raíz. Sin embargo, el uso de canales de diferente ancho de banda no es práctico, especialmente para la transmisión de mensajes encauzada. Una forma práctica de implementar árboles con canales de mayor ancho de banda en la vecindad del nodo raíz son los árboles gruesos (*fat trees*).

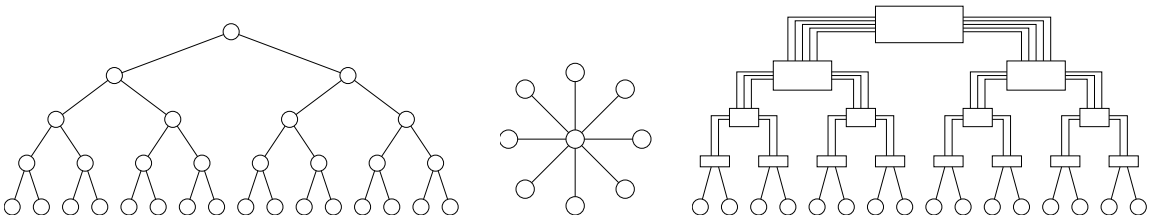
La estructura convencional de árbol utilizada en ciencias de la computación, puede ser modificada para conseguir un *árbol grueso*. Un árbol grueso binario se muestra en la figura 7.9(c). La anchura de canal de un árbol grueso se incrementa conforme se sube de las hojas a la raíz. El árbol grueso es más parecido a un árbol real donde las ramas son más gruesas conforme nos acercamos a la raíz.

El árbol grueso se introdujo para aliviar un problema grave de los árboles binarios convencionales, que consiste en el cuello de botella que se produce cerca de la raíz ya que el tráfico en esta zona es más intenso. El árbol grueso se ha utilizado, y se utiliza, en algunos supercomputadores. La idea del árbol grueso binario puede extenderse a árboles gruesos multivía.

Una de las propiedades más interesantes de los árboles es que, para cualquier grafo conectado, es posible definir un árbol dentro del grafo. Como consecuencia, para cualquier red conectada, es posible construir una red acíclica conectando todos los nodos eliminando algunos enlaces. Esta propiedad puede usarse para definir un algoritmo de encaminamiento para redes irregulares. Sin embargo, este algoritmo de encaminamiento puede ser ineficiente debido a la concentración del tráfico alrededor del nodo raíz.

Un *árbol binario* con 31 nodos y 5 niveles se muestra en la figura 7.9(a). En general, un árbol completamente balanceado de k niveles tiene $N = 2^k - 1$ nodos. El grado máximo de nodo en un árbol binario es 3 y el diámetro es $2(k-1)$. Si el grado de los nodos es constante, entonces el árbol es fácilmente escalable. Un defecto del árbol es que el diámetro puede resultar demasiado grande si hay muchos nodos.

La *estrella* es un árbol de dos niveles con un alto grado de nodo que es igual a $d = N - 1$ como se muestra en la figura 7.9(b). El diámetro resultante es pequeño, contante e igual a 2. La estructura en estrella se suele utilizar en sistemas con un supervisor que hace de nodo central.



(a) Árbol binario (b) Estrella (c) Árbol grueso binario

Figura 7.9: Árbol, estrella y árbol grueso.

Matrices sistólicas

Este es un tipo de arquitectura segmentada multidimensional diseñada para la realización de algoritmos específicos fijos. Por ejemplo, la red de la figura 7.3(d) corresponde a una matriz sistólica especialmente diseñada para la multiplicación de dos matrices. En este caso el grado de los nodos interiores es 6.

Con la interconexión fija y el funcionamiento síncrono de los nodos, la matriz sistólica encaja con la estructura de comunicación del algoritmo a realizar. Para aplicaciones específicas como el tratamiento de imágenes o señales, las matrices sistólicas pueden ofrecer una mejor relación entre rendimiento y coste. Sin embargo, la estructura puede tiene una aplicabilidad limitada y puede resultar muy difícil de programar.

7.1.3 Conclusiones sobre las redes directas

En la tabla 7.1 se resumen las características más importantes de las redes estáticas vistas hasta ahora. El grado nodal de la mayoría de redes es menor que 4, que está bastante bien. Por ejemplo, el Transputer chip de INMOS en un microprocesador que lleva incorporado la lógica de intercomunicación con un total de 4 puertos. Con un grado nodal constante igual a 4, un Transputer se puede utilizar como un bloque de construcción.

Los grados nodales de la estrella y de la red completamente conectada son ambos malos. El grado del nodo del hipercubo crece con $\log_2 N$ siendo también poco recomendable cuando N se hace grande.

Los diámetros de la red varían en un margen amplio. Con la invención del encaminamiento hardware (encaminado de agujero de gusano o *wormhole routing*), el diámetro ya no es un parámetro tan crítico ya que el retraso en la comunicación entre dos nodos cualquiera se ha convertido en algo casi constante con un alto nivel de segmentación. El número de enlaces afecta el coste de la red. La anchura biseccional afecta al ancho de banda.

La propiedad de simetría afecta a la escalabilidad y a la eficiencia en el rutado. Es justo decir que el coste total de la red crece con d (diámetro) y l (número de enlaces), por tanto, un diámetro pequeño es aun una virtud, pero la distancia media entre nodos puede ser una mejor medida. La anchura de banda biseccional puede mejorarse con un canal más ancho. Tomando el análisis anterior, el anillo, la malla, el toro, k-aria n-cubo, y el CCC todos tienen características propicias para construir los futuros sistemas MPP

Tipo de red	Grado nodal (d)	Diámetro de la red (D)	Número Enlaces (l)	Anchura bisección (B)	Simetría	Notas sobre el tamaño
Array lineal	2	$N-1$	$N-1$	1	No	N nodos
Anillo	2	$\lfloor N/2 \rfloor$	N	2	Sí	N nodos
Conectado completo	$N-1$	1	$N(N-1)/2$	$(N/2)^2$	Sí	N nodos
Árbol binario	3	$2(h-1)$	$N-1$	1	No	Altura árbol $h = \lceil \log_2 N \rceil$
Estrella	$N-1$	2	$N-1$	$\lfloor N/2 \rfloor$	No	N nodos
Malla 2D	4	$2(r-1)$	$2N-2r$	r	No	Malla $r \times r$ con $r = \sqrt{N}$
Malla Illiac	4	$r-1$	$2N$	$2r$	No	Equivalente al acorde con $r = \sqrt{N}$
Toro 2D	4	$2\lfloor r/2 \rfloor$	$2N$	$2r$	Sí	Toro $r \times r$ con $r = \sqrt{N}$
Hipercubo	n	n	$nN/2$	$N/2$	Sí	N nodos, $n = \log_2 N$ (dimensión)
CCC	3	$2k-1+\lfloor k/2 \rfloor$	$3N/2$	$N/(2k)$	Sí	$N = k \times 2^k$ nodos con longitud de ciclo $k \geq 3$
k-aria n-cubo	$2n$	$n\lfloor k/2 \rfloor$	nN	$2k^{n-1}$	Sí	$N = k^n$ nodos

Tabla 7.1: Resumen de las características de las redes estáticas.

(Procesadores Masivamente Paralelos).

En definitiva, con la utilización de técnicas segmentadas en el encaminamiento, la reducción del diámetro de la red ya no es un objetivo primordial. Cuestiones como la facilidad de encaminamiento, la escalabilidad y la facilidad para ampliar el sistema pueden ser actualmente temas más importantes, por lo que las redes estrictamente ortogonales se imponen en los multicomputadores actuales.

7.2 La capa de conmutación o control de flujo (*switching*)

En esta sección nos centraremos en las técnicas que se implementan dentro de los encaminadores (*routers*) para realizar el mecanismo por el cual los mensajes pasan a través de la red. Estas técnicas difieren en varios aspectos. Las *técnicas de conmutación* determinan cuándo y cómo se conectan los conmutadores internos del encaminador para conectar las entradas y salidas del mismo, así como cuándo los componentes del mensaje pueden transferirse a través de esos caminos. Estás técnicas están ligadas a los mecanismos de *control de flujo* que sincronizan la transferencia de unidades de información entre encaminadores y a través de los mismos durante el proceso de envío de los mensajes a través de la red. El control de flujo está a su vez fuertemente acoplado a los algoritmos de *manejo de buffers* que determinan cómo se asignan y liberan los buffers, determinando como resultado cómo se manejan los mensajes cuando se bloquean en la red.

Las implementaciones del nivel de conmutación difieren en las decisiones que se realizan en cada una de estas áreas, y su temporización relativa, es decir, cuándo una operación puede comenzar en relación con la ocurrencia de otra. Las elecciones específicas interactúan con la arquitectura de los encaminadores y los patrones de tráfico impuestos por los programas paralelos para determinar las características de latencia y el rendimiento de la red de interconexión.

7.2.1 Elementos básicos de la conmutación

El control de flujo es un protocolo asíncrono para transmitir y recibir una unidad de información. La *unidad de control de flujo* (flit) se refiere a aquella porción del mensaje cuya transmisión debe sincronizarse. Esta unidad se define como la menor unidad de información cuya transferencia es solicitada por el emisor y notificada por el receptor. Esta señalización *request/acknowledgment* se usa para asegurar una transferencia exitosa y la disponibilidad de espacio de buffer en el receptor. Obsérvese que estas transferencias son atómicas en el sentido de que debe asegurarse un espacio suficiente para asegurar que el paquete se transfiere en su totalidad.

El control de flujo ocurre a dos niveles. El *control de flujo del mensaje* ocurre a nivel de paquete. Sin embargo, la transferencia del paquete por el canal físico que une dos encaminadores se realiza en varios pasos, por ejemplo, la transferencia de un paquete de 128-bytes a través de una canal de 16-bits. La transferencia resultante multiciclo usa un *control del flujo del canal* para enviar un flit a través de la conexión física.

Las técnicas de conmutación suelen diferenciarse en la relación entre el tamaño de la unidad de control física y del paquete. En general, cada mensaje puede dividirse en *paquetes* de tamaño fijo. Estos paquetes son a su vez divididos en unidades de control de flujo o flits. Debido a las restricciones de anchura del canal, puede ser necesario varios ciclos de canal para transferir un único flit. Un *phit* es la unidad de información que puede transferirse a través de un canal físico en un único paso o ciclo. Los flits representan unidades lógicas de información en contraposición con los phits que se corresponden a cantidades físicas, es decir, número de bits que pueden transferirse en paralelo en un único ciclo. La figura 7.10 muestra N paquetes, 6 flits/paquete y 2 phits/flit.

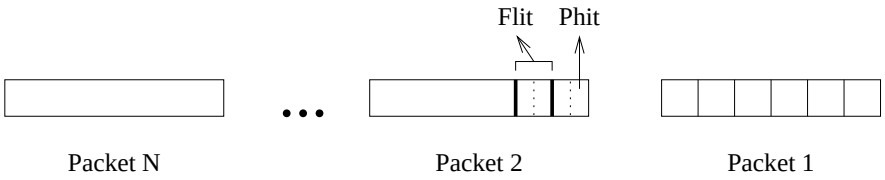


Figura 7.10: Distintas unidades de control de flujo en un mensaje.

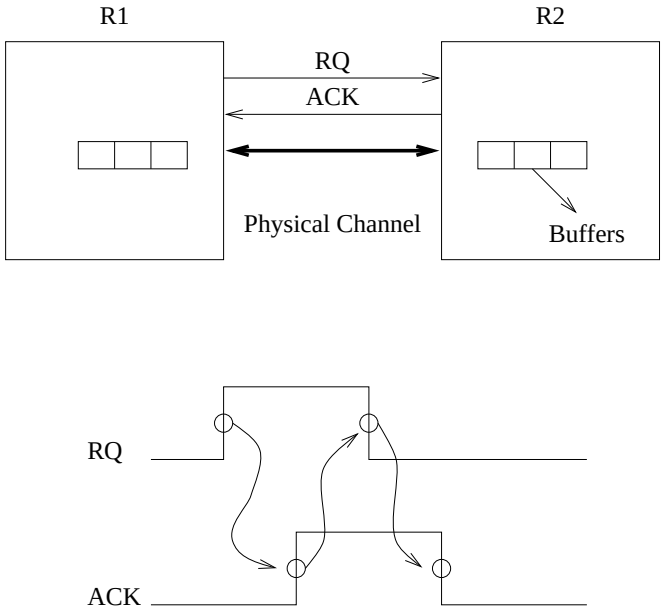


Figura 7.11: Un ejemplo de control de flujo asíncrono de un canal físico.

Existen muchos candidatos a protocolo de sincronización para coordinar la transferencia de bits a través de un canal. La figura 7.11 ilustra un ejemplo de protocolo asíncrono de cuatro fases. El encaminador $R1$ pone a uno la señal RQ antes de comenzar la transferencia de información. El encaminador $R2$ responde leyendo los datos y activando la señal ACK. Esto da lugar a la desactivación de RQ por parte de $R1$ que causa la desactivación de ACK por $R2$. La señal ACK se utiliza tanto para confirmar la recepción (flanco ascendente) como para indicar la existencia de espacio de buffer (flanco descendente) para la siguiente transferencia.

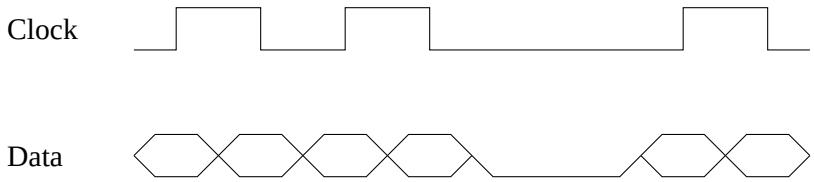


Figura 7.12: Un ejemplo de control de flujo síncrono de un canal físico.

El control de flujo del canal también puede ser síncrono, como se muestra en la figura 7.12. La señal de reloj se transmite por el canal y ambos flancos de la señal de reloj se utilizan para validar las líneas de datos en el receptor. La figura 7.12 no muestra las señales ACK utilizadas para indicar la existencia de espacio en el nodo receptor.

Mientras que las transferencias inter-encaminador deben realizarse necesariamente en términos de phits, las técnicas de conmutación manejan flits (que pueden definirse hasta llegar a tener el tamaño del paquete completo). Las técnicas de conmutación manejan el conmutador interno para conectar buffers de entrada con buffers de salida, y enviar los flits a través de este camino. Estas técnicas se distinguen por el instante en que ocurren en relación con la operación de control de flujo y la operación de encaminamiento. Por ejemplo, la conmutación puede tener lugar después de que un flit haya sido recibido completamente. Alternativamente, la transferencia de un flit a través del conmutador puede empezar en cuanto finaliza la operación de encaminamiento, pero antes de recibir el resto del flit desde el encaminador anterior. En este caso la conmutación se solapa con el control de flujo a nivel de mensaje. En una última técnica de conmutación propuesta, la conmutación comienza después de recibir el primer phit, antes incluso de que haya finalizado la operación de encaminamiento.

Modelo de encaminador

A la hora de comparar y contrastar diferentes alternativas de conmutación, estamos interesados en cómo las mismas influyen en el funcionamiento del encaminador y, por tanto, la latencia y ancho de banda resultante. La arquitectura de un encaminador genérico se muestra en la figura 7.13 y está compuesto de los siguientes elementos principales:

- *Buffers*. Son buffers FIFO que permiten almacenar mensajes en tránsito. En el modelo de la figura 7.13, un buffer está asociado con cada canal físico de entrada y de salida. En diseños alternativos, los buffers pueden asociarse únicamente a las entradas o a las salidas. El tamaño del buffer es un número entero de unidades de control de flujo (*flits*).
- *Conmutador* . Este componente es el responsable de conectar los buffers de entrada del encaminador (*router*) con los buffers de salida. Los encaminadores de alta

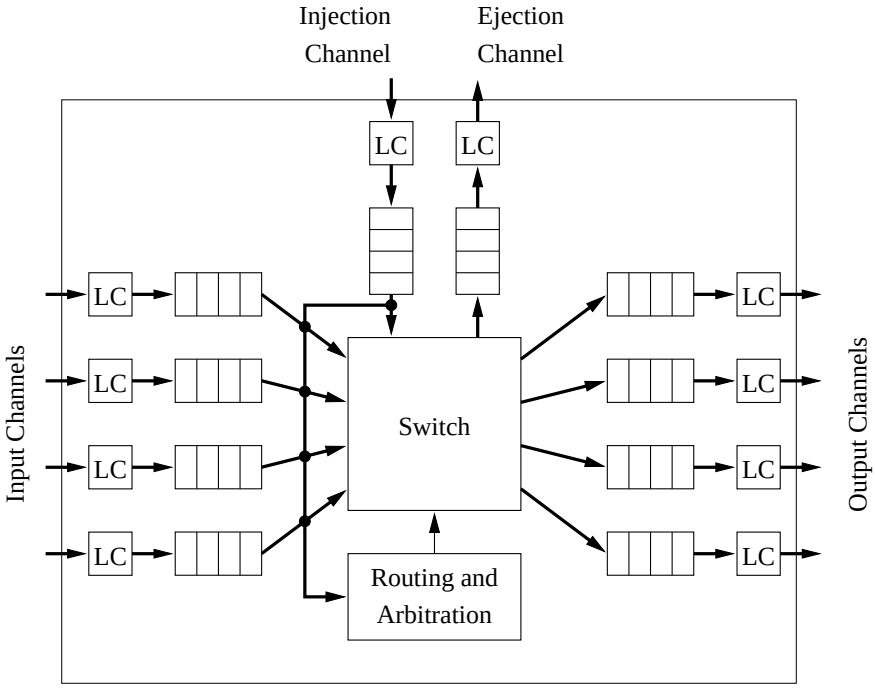


Figura 7.13: Modelo de encaminador (*router*) genérico. (LC = Link controller.)

velocidad utilizan redes de barra cruzada (*crossbar*) con conectividad total, mientras que implementaciones de más baja velocidad utilizan redes que no proporcionan una conectividad total entre los buffers de entrada y los de salida.

- *Unidad de encaminamiento y arbitraje.* Este componente implementa los algoritmos de encaminamiento, selecciona el enlace de salida para un mensaje entrante, programando el conmutador en función de la elección. Si varios mensajes piden de forma simultánea el mismo enlace de salida este componente debe proporcionar un arbitraje entre ellos. Si el enlace pedido está ocupado, el mensaje debe permanecer en el buffer de entrada hasta que éste quede libre.
- *Controladores de enlace (LC).* El flujo de mensajes a través de los canales físicos entre encaminadores adyacentes se implementa mediante el LC. Los controladores de enlace de cada lado del canal deben coordinarse para transferir flits.
- *Interfaz del procesador.* Este componente simplemente implementa un canal físico con el procesador en lugar de con un encaminador adyacente. Consiste en uno o más canales de inyección desde el procesador y uno o más canales de eyección hacia el procesador. A los canales de eyección también se les denominan canales de reparto o canales de consumición.

Desde el punto de vista del rendimiento del encaminador (*router*) estamos interesados en dos parámetros. Cuando un mensaje llega a un encaminador, éste debe ser examinado para determinar el canal de salida por el cual se debe enviar el mensaje. A esto se le denomina *retraso de encaminamiento* (*routing delay*), y suele incluir el tiempo para configurar el conmutador. Una vez que se ha establecido un camino a través del encaminador, estamos interesados en la velocidad a la cual podemos enviar mensajes a través del conmutador. Esta velocidad viene determinado por el retraso de propagación a través de conmutador (retraso intra-encaminadores), y el retraso que permite la sincronización de la transferencia de datos entre los buffers de entrada y de

salida. A este retraso se le denomina latencia de *control de flujo interno*. De manera similar, al retraso a través de los enlaces físicos (retraso inter-encaminadores) se le denomina latencia del *control de flujo externo*. El retraso debido al encaminamiento y los retrasos de control de flujo determinan la latencia disponible a través del conmutador y, junto con la contención de los mensajes en los enlaces, determina el rendimiento o productividad de la red (*throughput*).

Técnicas de conmutación

Antes de comenzar a estudiar las distintas técnicas de conmutación, es necesario tener en cuenta algunas definiciones. Para cada técnica de conmutación se considerará el cálculo de la latencia base de un mensaje de L bits en ausencia de tráfico. El tamaño del *phit* y del *flit* se supondrán equivalentes e iguales al ancho de un canal físico (W bits). La longitud de la cabecera se supondrá que es de 1 flit, así el tamaño el mensaje será de $L+W$. Un encaminador (*router*) puede realizar una decisión de encaminamiento en t_r segundos. El canal físico entre 2 encaminadores opera a B Hz, es decir, el ancho de banda del canal físico es de BW bits por segundo. Al retraso de propagación a través de este canal se denota por $t_w = \frac{1}{B}$. Una vez que sea establecido un camino a través de un encaminador, el retraso intra-encaminador o retraso de conmutación se denota por t_s. El camino establecido dentro del encaminador se supone que coincide con el ancho del canal de W bits. Así, en t_s segundos puede transferirse un flit de W bits desde la entrada a la salida del encaminador. Los procesadores origen y destino constan de D enlaces. La relación entre estos componentes y su uso en el cálculo de la latencia de un mensaje bajo condiciones de no carga se muestra en la figura 7.14.

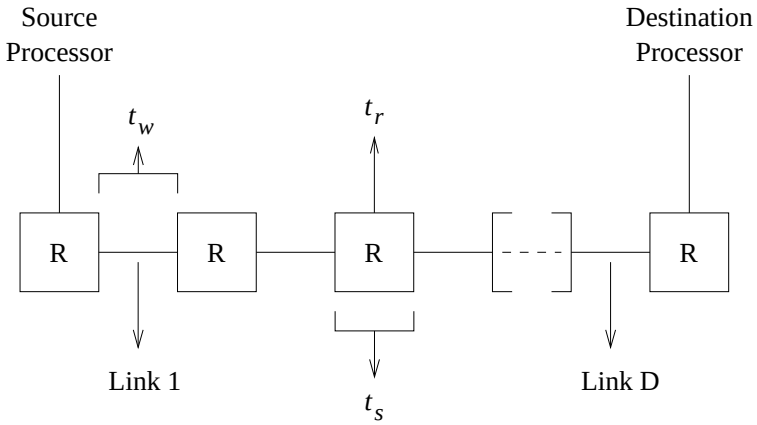


Figura 7.14: Cálculo de la latencia en una red para el caso de ausencia de carga (R = Encaminador o *Router*).

7.2.2 Conmutación de circuitos

En la conmutación de circuitos, se reserva un camino físico desde el origen hasta el destino antes de producirse la transmisión de los datos. Esto se consigue mediante la inyección de un flit cabecera en la red. Esta trama de sondeo del encaminamiento contiene la dirección del destino e información de control adicional. La misma progresa hacia el destino reservando los enlaces físicos conforme se van transmitiendo a través

de los encaminadores intermedios. Cuando alcanza el destino, se habrá establecido un camino, enviándose una señal de asentimiento de vuelta al origen. El contenido del mensaje puede ahora ser emitido a través del camino hardware. El circuito puede liberarse por el destino o por los últimos bits del mensaje. En la figura 7.15 se muestra un cronograma de la transmisión de un mensaje a través de tres enlaces. La cabecera se envía a través de los tres enlaces seguida de la señal de asentimiento. En la figura 7.16 se muestra un ejemplo del formato de la trama de creación de un circuito.

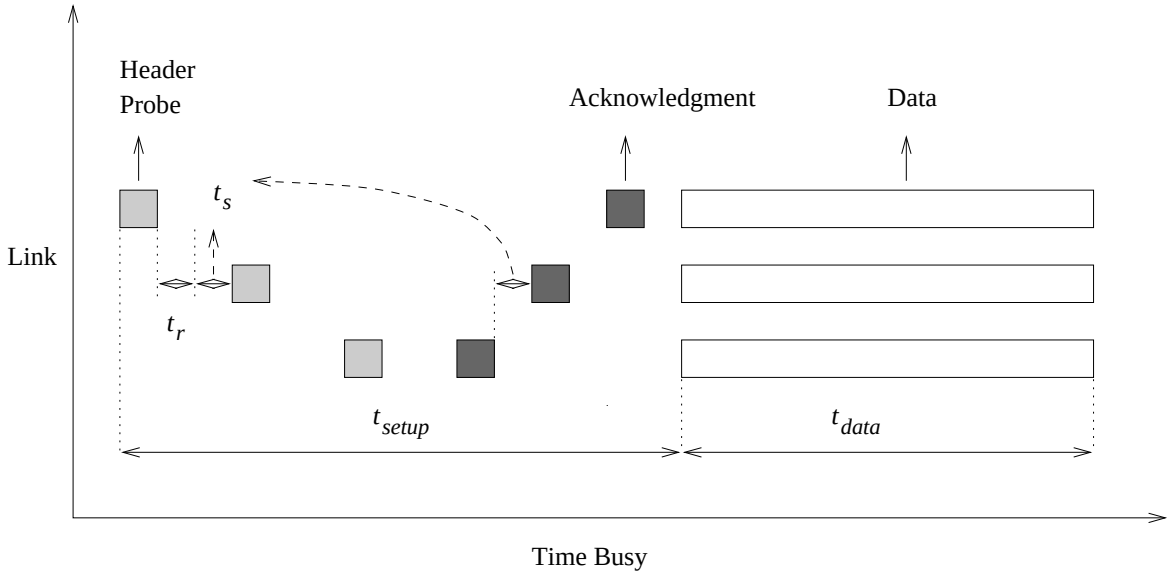


Figura 7.15: Cronograma de un mensaje por conmutación de circuitos.

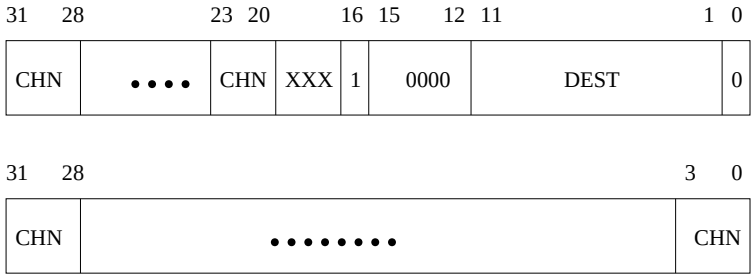


Figura 7.16: Un ejemplo del formato en una trama de creación de un circuito. (CHN = Número de canal; DEST = Dirección destino; XXX = No definido.)

El mejor comportamiento de esta técnica de conmutación ocurre cuando los mensajes son infrecuentes y largos, es decir, cuando el tiempo de transmisión del mensaje es largo en comparación con el tiempo de establecimiento del circuito. La desventaja es que el camino físico se reserva para toda la duración del mensaje y puede bloquear a otros mensajes. Por ejemplo, consideremos el caso en donde la trama de sondeo está esperando a que un enlace físico esté libre. Todos los enlaces reservados por la trama hasta ese punto permanecen reservados, no pueden ser utilizados por otros mensajes, y pueden bloquear el establecimiento de otros circuitos. Así, si el tamaño del mensaje no es mucho más grande que el tamaño de la trama de sondeo, sería ventajoso transmitir el mensaje junto con la cabecera y almacenar el mensaje dentro de los encaminadores

mientras que se espera a que se libere un enlace. A esta técnica alternativa se le denomina conmutación de paquetes.

La latencia base un mensaje en esta técnica de conmutación está determinado por el tiempo de establecimiento de un camino, y el tiempo posterior en el que el camino está ocupado transmitiendo los datos. El modo de funcionamiento de encaminador difiere un poco el mostrado en la figura 7.13. Mientras que la trama de sondeo se almacena en cada encaminador, los datos no son almacenados. No hay buffers y el circuito se comporta como si se tratase de un único enlace entre el origen y destino.

La expresión de la latencia base para un mensaje es la siguiente:

$$\begin{aligned} t_{circuit} &= t_{setup} + t_{data} \\ t_{setup} &= D[t_r + 2(t_s + t_w)] \\ t_{data} &= \frac{1}{B}\left\lceil \frac{L}{W} \right\rceil \end{aligned} \tag{7.1}$$

7.2.3 Conmutación de paquetes

En la conmutación de circuitos, la totalidad del mensaje se transmite después de que se haya restablecido el circuito. Una alternativa sería que el mensaje pudiese dividirse y transmitirse en paquetes de longitud fija, por ejemplo, 128 bytes. Los primeros bytes del paquete contendrían la información de encaminamiento y constituirían lo que se denomina cabecera paquete. Cada paquete se encamina de forma individual desde el origen al destino. Un paquete se almacena completamente en cada nodo intermedio antes de ser enviado al siguiente nodo. Esta es la razón por la que a este método de conmutación también se le denomina conmutación de *almacenamiento y reenvío* (SAF). La información de la cabecera se extrae en los encaminadores intermedios y se usa para determinar el enlace de salida por el que se enviará el paquete. En la figura 7.17 se muestra un cronograma del progreso de un paquete a lo largo de tres enlaces. En esta figura podemos observar que la latencia experimentada por un paquete es proporcional a la distancia entre el nodo origen y destino. Observar que en la figura se ha omitido la latencia del paquete a través del encaminador.

Esta técnica es ventajosa cuando los mensajes son cortos y frecuentes. Al contrario que en la conmutación de circuitos, donde un segmento de un camino reservado puede estar sin ser usado durante un período de tiempo significativo, un enlace de comunicación se usa completamente cuando existen datos que transmitir. Varios paquetes pertenecientes a un mensaje pueden estar en la red simultáneamente incluso si el primer paquete no ha llegado todavía al destino. Sin embargo, dividir un mensaje en paquetes produce una cierta sobrecarga. Además del tiempo necesario en los nodos origen y destino, cada paquete debe ser encaminado en cada nodo intermedio. En la figura 7.18 se muestra un ejemplo del formato de la cabecera del paquete.

La latencia base de un mensaje en conmutación de paquetes se puede calcular como:

$$t_{packet} = D\left\{ t_r + (t_s + t_w)\left\lceil \frac{L+W}{W} \right\rceil \right\} \tag{7.2}$$

Esta expresión sigue el modelo de encaminador (*router*) mostrado en la figura 7.13,

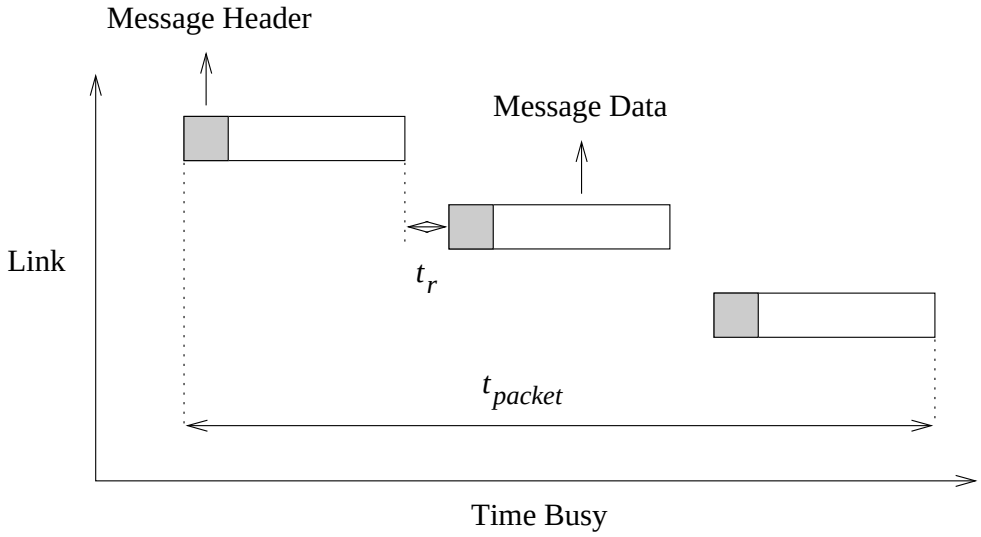


Figura 7.17: Cronograma de un mensaje por conmutación de paquetes.

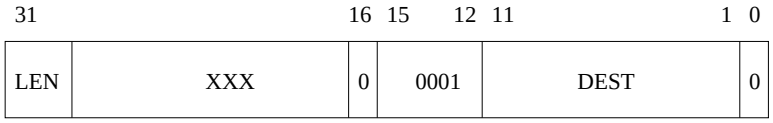


Figura 7.18: Un ejemplo de formato de la cabecera del paquete. (DEST = Dirección destino; LEN = Longitud del paquete en unidades de 192 bytes; XXX = No definido.)

y es el resultado de incluir los factores que representan el tiempo de transferencia del paquete de longitud $L + W$ a través del canal (t_w) así como del almacenamiento en la entrada y la transferencia desde los buffers de entrada a los de salida del encaminador.

7.2.4 Conmutación de paso a través virtual, *Virtual Cut-Through* (VCT)

La conmutación de paquete se basa en suponer que un paquete debe recibirse en su globalidad antes de poder realizar cualquier decisión de encaminamiento y antes de continuar el envío de paquete hacia su destino. En general, esto no es cierto. Consideremos un paquete de 128 bytes y el modelo de encaminador mostrado en la figura 7.13. En ausencia de canales físicos de ciento veintiocho bytes de ancho, la transferencia del paquete a través del canal físico llevará varios ciclos. Sin embargo, los primeros bytes contienen la información de encaminamiento que estará disponible después de los primeros ciclos. En lugar de esperar a recibir el paquete en su totalidad, la cabecera del paquete puede ser examinada tan pronto como llega. El encaminador puede comenzar a enviar la cabecera y los datos que le siguen tan pronto como se realice una decisión de encaminamiento y el buffer de salida esté libre. De hecho, el mensaje no tiene ni siquiera que ser almacenado en la salida y puede ir directamente a la entrada del siguiente encaminador antes de que el paquete completo se haya recibido en el encaminador actual. A esta técnica de conmutación se le denomina conmutación *virtual cut-through* (VCT) o paso a través virtual. En ausencia de bloqueo, la latencia experimentada por la cabecera en cada nodo es la latencia de encaminamiento y el retraso de propagación a

través del encaminador y a lo largo de los canales físicos. El mensaje puede segmentarse a través de comunicaciones sucesivas. Si la cabecera se bloquea en un canal de salida ocupado, la totalidad del mensaje se almacena en el nodo. Así, para altas cargas de la red, la conmutación VCT se comporta como la conmutación de paquetes.

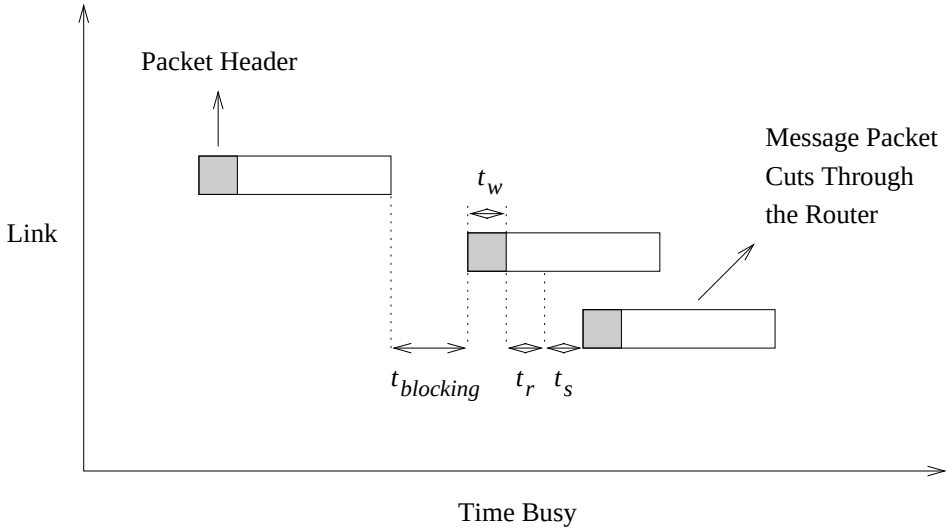


Figura 7.19: Cronograma para un mensaje conmutado por VCT. ($t_{blocking}$= Tiempo de espera en un enlace de salida.)

La figura 7.19 ilustra un cronograma de la transferencia de un mensaje usando conmutación VCT. En esta figura el mensaje se bloquea después de pasar el primer enlace a la espera de que se libere un canal de salida. En este caso observamos cómo la totalidad del paquete tiene que ser transferida al primer encaminador mientras éste permanece bloqueado esperando a la liberación de un puerto de salida. Sin embargo, en la misma figura podemos observar que el mensaje atraviesa el segundo encaminador sin detenerse cruzando al tercer enlace.

La latencia base para un mensaje que no se bloquea en alguno de los encaminadores puede calcularse como sigue:

$$t_{vct} = D(t_r + t_s + t_w) + \max(t_s, t_w) \left\lceil \frac{L}{W} \right\rceil \tag{7.3}$$

En el cálculo de dicha latencia se ha supuesto que el encaminamiento ocurre a nivel de flit con la información del encaminamiento contenida en un flit. Este modelo supone que no existe penalización de tiempo por atravesar un encaminador si el buffer y el canal de salida están libres. Dependiendo de la velocidad de operación de los encaminadores este hecho puede no ser realista. Otro hecho a destacar es que únicamente la cabecera experimenta el retraso del encaminamiento, al igual que el retraso en la conmutación y en los enlaces en cada encaminador. Esto es debido a que la transmisión está segmentada y a la existencia de buffers a la entrada y salida del conmutador. Una vez que el flit cabecera alcanza al destino, el resto del tiempo viene determinado por el máximo entre el retraso del conmutador y el retraso en el enlace entre los encaminadores. Si el conmutador sólo tuviera buffers en la entrada, entonces en un ciclo, un flit atravesaría el conmutador y el canal entre los encaminadores, en este caso el coeficiente del segundo

término sería $t_s + t_w$. Obsérvese que la unidad de control de flujo del mensaje es un paquete. Por lo tanto, incluso en el caso de que el mensaje pueda atravesar el encaminador, debe existir suficiente espacio buffer para permitir el almacenamiento de un paquete completo en el caso de que la cabecera se bloquee.

7.2.5 Conmutación de lombriz (*Wormhole*)

La necesidad de almacenar completamente los paquetes dentro del encaminador (*router*) puede complicar el diseño de encaminadores compactos, rápidos y de pequeño tamaño. En la conmutación segmentada, los paquetes del mensaje también son segmentados a través de la red. Sin embargo, las necesidades de almacenamiento dentro de los encaminadores pueden reducirse sustancialmente en comparación con la conmutación VCT. Un paquete se divide en flits. El flit es la unidad de control de flujo de los mensajes, siendo los buffers de entrada y salida de un encaminador lo suficientemente grandes para almacenar algunos flits. El mensaje se segmenta dentro de la red a nivel de flits y normalmente es demasiado grande para que pueda ser totalmente almacenado dentro de un buffer. Así, en un instante dado, un mensaje bloqueado ocupa buffers en varios encaminadores. En la figura 7.20 se muestra el diagrama temporal de la conmutación segmentada. Los rectángulos en blanco y ilustran la propagación de los flits al largo del canal físico. Los rectángulos sombreados muestran la propagación de los flits cabecera a lo largo de los canales físicos. También se muestran en la figura los retrasos debidos al encaminamiento y a la propagación dentro del encaminador de los flits cabecera. La principal diferencia entre la conmutación segmentada y la conmutación VCT es que la unidad de control del mensaje es el flit y, como consecuencia, el uso de buffers más pequeños. Un mensaje completo no puede ser almacenado en un buffer.

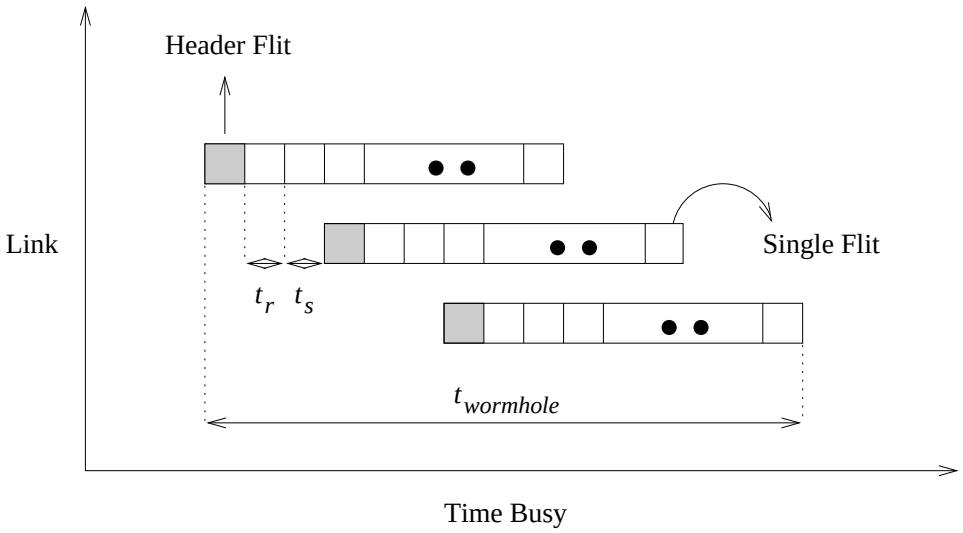


Figura 7.20: Cronograma de un mensaje conmutado mediante wormhole.

En ausencia de bloqueos, los paquetes de mensaje se segmenta a largo de la red. Sin embargo, las características de bloqueo son muy diferentes de las del VCT. Si el canal de salida demandado está ocupado, el mensaje se bloquea in situ. Por ejemplo, la figura 7.21 ilustra una instantánea de un mensaje siendo transmitido a través de

los encaminadores $R1$, $R2$ y $R3$. Los buffers de entrada y salida son de dos flits y las cabeceras tienen esa misma longitud. En el encaminador $R3$, el mensaje A necesita un canal de salida que está siendo utilizado por el mensaje B. Por lo tanto, el mensaje A queda bloqueado. El menor tamaño de los buffers en cada nodo hace que el mensaje ocupe buffers en varios encaminadores, pudiendo dar lugar al bloqueo de otros mensajes. De hecho las dependencias entre los buffers abarcan varios encaminadores. esta propiedad complicará el diseño de algoritmos libres de bloqueos para esta técnica de conmutación, como veremos en la siguiente sección. Sin embargo, ya no es necesario el uso de la memoria del procesador local para almacenar mensaje, reduciendo significativamente la latencia media del mensaje. Las bajas necesidades de almacenamiento y la segmentación de los mensajes permite la construcción de encaminadores que son pequeños, compactos, y rápidos. La figura 7.22 muestra el formato de los paquetes en el Cray T3D.

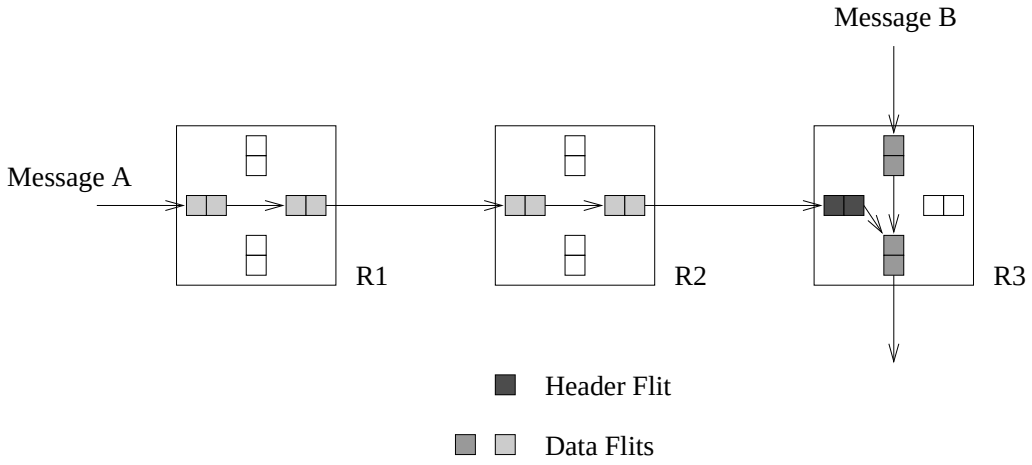


Figura 7.21: Un ejemplo de mensaje bloqueado con la técnica wormhole.

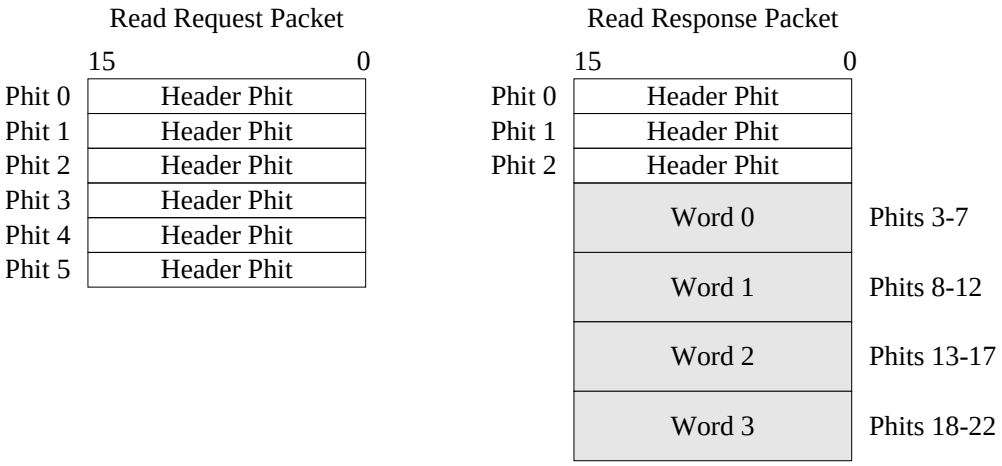


Figura 7.22: Formato de los paquetes conmutados mediante wormhole en el Cray T3D.

La latencia base para un mensaje conmutado mediante wormhole puede calcularse como sigue:

$$t_{wormhole} = D(t_r + t_s + t_w) + \max(t_s, t_w) \left\lceil \frac{L}{W} \right\rceil \tag{7.4}$$

Esta expresión supone que la capacidad de los buffers se mide en flits tanto en las entradas como en las salidas del encaminador. Obsérvese que, en ausencia de contención, VCT y la conmutación segmentada tienen la misma latencia. Una vez que el flit cabecera llega al destino, el tiempo de llegada del resto del mensaje está determinado por el máximo entre el retraso enn el conmutador y el retraso en el enlace.

7.2.6 Conmutación cartero loco

La conmutación VCT mejora el rendimiento de la conmutación de paquetes permitiendo la segmentación de los mensajes a la vez que retiene la posibilidad de almacenar completamente los paquetes de un mensaje. La conmutación segmentada proporciona una mayor reducción de la latencia al permitir menor espacio de almacenamiento que el VCT de tal manera que el encaminamiento puede realizarse utilizando encaminadores implementados en un único chip, proporcionando la menor latencia necesaria para el procesamiento paralelo fuertemente acoplado. Esta tendencia a aumentar la segmentación del mensaje continua con el desarrollo del mecanismo de conmutación del cartero loco en un intento de obtener la menor latencia de encaminamiento posible por nodo.

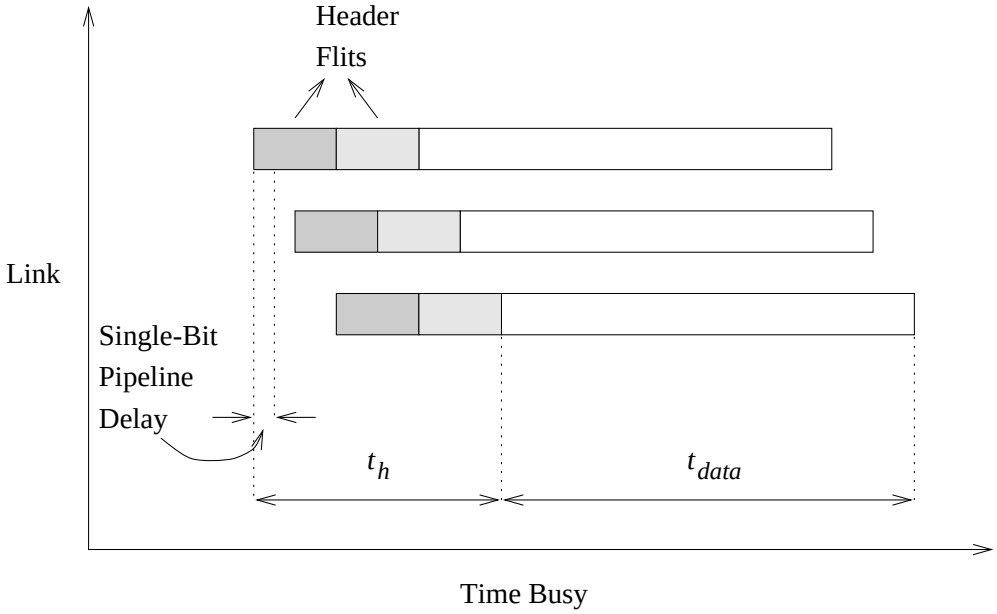


Figura 7.23: Cronograma de la transmisión de un mensaje usando la conmutación del cartero loco.

Esta técnica puede entenderse más fácilmente en el contexto de canales físicos serie. Consideremos una malla 2-D con paquetes cuyas cabeceras tienen dos flits. El encaminamiento se realiza por dimensiones: los mensajes se encaminan en primer lugar a lo largo de la dimensión 0 y después a lo largo de la dimensión 1. El primer flit cabecera contiene la dirección destino de un nodo en la dimensión 0. Cuando el mensaje llega a ese nodo, se envía a través de la dimensión 1. El segundo flit cabecera

contiene el destino del mensaje en esta dimensión. En VCT y *wormhole* los flits no pueden continuar hasta que los flits cabecera han sido recibidos completamente en el encaminador. Si tenemos un flit de 8 bits, la transmisión de los flits cabecera a través de un canal serie de un bit tardará 16 ciclos. Suponiendo un retraso de 1 ciclo para seleccionar el canal de salida en cada encaminador intermedio, la latencia mínima para que la cabecera alcance el encaminador destino a una distancia de tres enlaces es de 51 ciclos. El cartero loco intenta reducir aún más la latencia por nodo mediante una segmentación a nivel de bit. Cuando el flit cabecera empieza a llegar a un encaminador, se supone que el mensaje continuará a lo largo de la misma dimensión. Por lo tanto los bits cabecera se envían hacia el enlace de salida de la misma dimensión tan pronto como se reciben (suponiendo que el canal de salida está libre). Cada bit de la cabecera también se almacena localmente. Una vez que se recibe el último bit del primer flit de la cabecera, el encaminador puede examinar este flit y determinar si el mensaje debe continuar a lo largo de esta dimensión. Si el mensaje debe ser enviado por la segunda dimensión, el resto del mensaje que empieza con el segundo flit de la cabecera se transmite por la salida asociada a la segunda dimensión. Si el mensaje ha llegado a su destino, se envía al procesador local. En esencia, el mensaje primero se envía a un canal de salida y posteriormente se comprueba la dirección, de ahí el nombre de la técnica de conmutación. Esta estrategia puede funcionar muy bien en redes 2-D ya que un mensaje realizará a lo más un giro de una dimensión a otra y podemos codificar la diferencia en cada dimensión un 1 flit cabecera. El caso común de los mensajes que continúan en la misma dimensión se realiza muy rápidamente. La figura 7.23 muestra un cronograma de la transmisión de un mensaje que se transmite sobre tres enlaces usando esta técnica de conmutación.

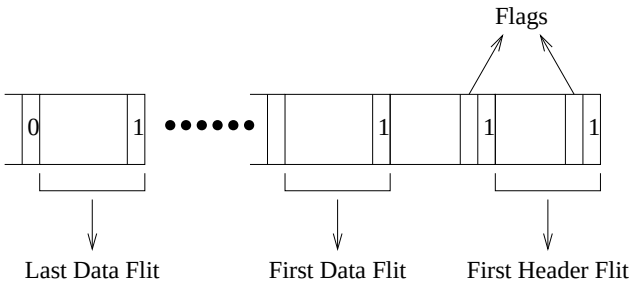


Figura 7.24: Un ejemplo del formato de un mensaje en la técnica de conmutación del cartero loco.

Es necesario considerar algunas restricciones en la organización de la cabecera. la figura 7.24 muestra un ejemplo, donde la diferencia en cada dimensión se codifica en un flit cabecera, y estos flits se ordenan de acuerdo con el orden en que se atraviesa cada dimensión. Por ejemplo, cuando el paquete ha atravesado completamente la primera dimensión el encaminador puede empezar a transmitir en la segunda dimensión con el comienzo del primer bit del segundo flit cabecera. El primer flit se elimina del mensaje, pero continúa atravesando la primera dimensión. A este flit se le denomina *flit de dirección muerto*. En una red multidimensional, cada vez que un mensaje cambia a una nueva dirección, se genera un flit muerto y el mensaje se hace más pequeño. En cualquier punto si se almacena un flit muerto, por ejemplo, al bloquearse por otro paquete, el encaminador local lo detecta y es eliminado.

Consideremos un ejemplo de encaminamiento en una malla 2-D 4×4. En este ejemplo la cabecera de encaminamiento se encuentra comprimida en 2 flits. Cada flit

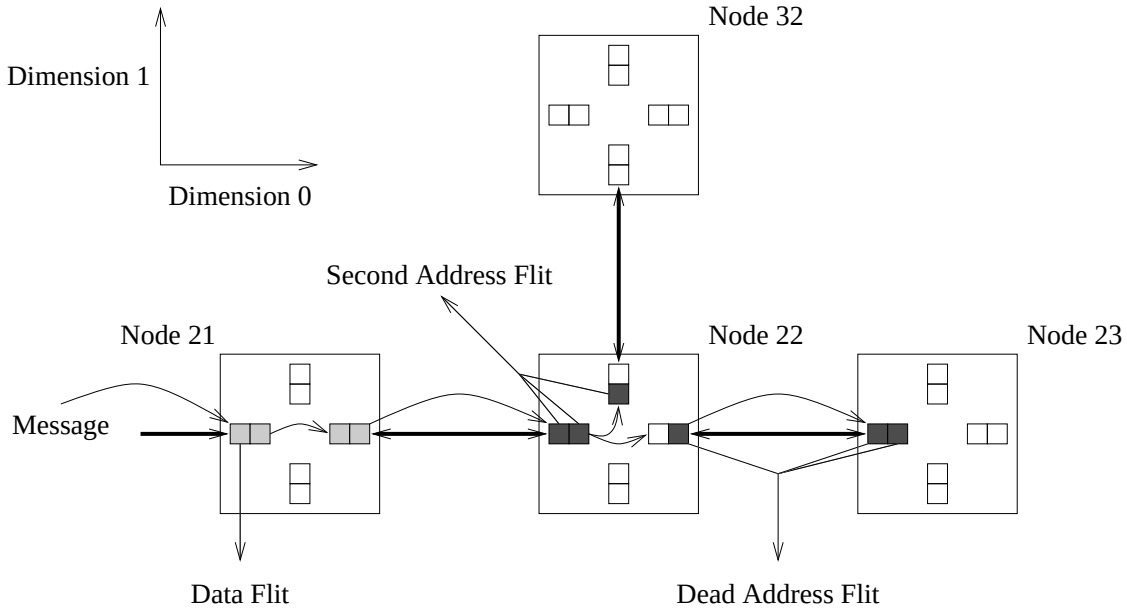


Figura 7.25: Ejemplo de encaminamiento con la conmutación del cartero loco y la generación de flits de dirección muertos.

consta de 3 bits: un bit especial de inicio y dos bits para identificar el destino en cada dimensión. El mensaje se segmenta a través de la red a nivel de bit. El buffer de entrada y salida es de 2 bits. Consideremos el caso en donde el nodo se transmite desde el nodo 20 al nodo 32. la figura 7.25 ilustra el progreso y la localización de los flits cabecera. El mensaje se transmite a lo largo de la dimensión 0 al nodo 22 en donde se transmite al nodo 32 a lo largo de la dimensión 1. En el nodo 22, el primer flit se envía a través del la salida cuando se recibe. Después de recibir el tercer bit, se determina que el mensaje debe continuar a lo largo de la dimensión 1. El primer bit del segundo flit cabecera se envía a través de la salida de la dimensión 1 tal como se muestra la figura. Observar que los flits cabecera se eliminan del mensaje cuando éste cambia de dimensión. Los flits muertos continúan a lo largo de la dimensión 0 hasta que se detectan y son eliminados.

Para un número dado de procesadores el tamaño de los flits de dirección muertos está determinado por el número de procesadores en una dimensión. Por lo tanto, para un número dado de procesadores, las redes de baja dimensión producirán un menor número de flits muertos de mayor tamaño mientras que las redes de mayor dimensión introducirán una mayor cantidad de flits muertos de menor tamaño. Inicialmente podría parecer que los flits de dirección muertos podrían afectar negativamente al rendimiento hasta que son eliminados de la red ya que están consumiendo ancho de banda de los canales físicos. Dado que los paquetes de un mensaje son generalmente más grandes que los flits muertos, la probabilidad de que un paquete se bloquee a causa de un flit de dirección muerto es muy pequeña. Es más probable que el flit muerto se bloquee por un paquete. En este caso el encaminador local tiene la oportunidad de detectar el flit muerto y eliminarlo de la red. A altas cargas, podríamos estar preocupados por los flits de dirección muertos que consumen un ancho de banda escaso. En este caso, es interesante destacar que un incremento del bloqueo en la red proporcionará más oportunidades para que los encaminadores eliminen los flits muertos. A mayor congestión, menor será la probabilidad de que un paquete encuentre un flit de dirección muerto.

Mediante el encaminamiento optimista del flujo de bits que componen un mensaje hacia un canal de salida, la latencia de encaminamiento en un nodo se minimiza consiguiéndose una segmentación a nivel de bit. Consideremos de nuevo una malla 2-D con canales físicos con una anchura de 1 bit y con paquetes con dos flits cabecera de 8-bits, atravesando tres enlaces, la latencia mínima para que la cabecera alcance el destino es de 18 en lugar de 51 ciclos. En general, la estrategia del cartero loco es útil cuando son necesarios varios ciclos para que la cabecera atraviese un canal físico. En este caso la latencia puede reducirse mediante el envío optimista de porciones de la cabecera antes de que pueda determinarse el enlace de salida real. Sin embargo, los modernos encaminadores permiten la transmisión de flits más anchos a lo largo de un canal en un único ciclo. Si la cabecera puede transmitirse en un ciclo, las ventajas que se pueden obtener son pequeñas.

La latencia base de un mensaje encaminado usando la técnica de conmutación del cartero loco puede calcularse como sigue:

$$\begin{aligned} t_{madpostman} &= t_h + t_{data} \\ t_h &= (t_s + t_w)D + \max(t_s, t_w)W \\ t_{data} &= \max(t_s, t_w)L \end{aligned} \tag{7.5}$$

La expresión anterior realiza varias suposiciones. La primera es que se usan canales serie de 1 bit que son los más favorables a la estrategia del cartero loco. El tiempo de encaminamiento t_r se supone que es equivalente al retraso en la conmutación y ocurre de forma concurrente con la transmisión del bit, y por lo tanto no aparece en la expresión. El término t_h se corresponde con el tiempo necesario para enviar la cabecera.

Consideremos el caso general en donde no tenemos canales serie, sino canales de C bits, donde $1 < C < W$. En este caso se necesitan varios ciclos para transferir la cabecera. En este caso la estrategia de conmutación del cartero loco tendría una latencia base de

$$t_{madpostman} = D(t_s + t_w) + \max(t_s, t_w) \left\lceil \frac{W}{C} \right\rceil + \max(t_s, t_w) \left\lceil \frac{L}{C} \right\rceil \tag{7.6}$$

Por razones de comparación, en este caso la expresión de la conmutación segmentada habría sido

$$t_{wormhole} = D \left\{ t_r + (t_s + t_w) \left\lceil \frac{W}{C} \right\rceil \right\} + \max(t_s, t_w) \left\lceil \frac{L}{C} \right\rceil \tag{7.7}$$

Supongamos que los canales internos y externos son de C bits, y un flit cabecera (cuya anchura es de W bits) requiere $\lceil \frac{W}{C} \rceil$ ciclos para cruzar el canal y el encaminador. Este coste se realiza en cada encaminador intermedio. Cuando $C = W$, la expresión anterior se reduce a la expresión de la conmutación segmentada con cabeceras de un único flit.

7.2.7 Canales virtuales

Las técnicas de comunicación anteriores fueron descritas suponiendo que los mensajes o parte de los mensajes se almacenan a la entrada y salida de cada canal físico. Por

lo tanto, una vez que un mensaje ocupa el buffer asociado a un canal, ningún otro mensaje pueda acceder al canal físico, incluso en el caso de que el mensaje este bloqueado. Sin embargo, un canal físico puede soportar varios canales *virtuales* o *lógicos* multiplexados sobre el mismo canal físico. Cada canal virtual unidireccional se obtiene mediante el manejo independiente de un par de buffers de mensajes como se muestra en la figura 7.26. Esta figura muestra dos canales virtuales unidireccionales en cada dirección sobre un canal físico. Consideremos la conmutación segmentada con mensaje en cada canal virtual. Cada mensaje puede comprobar si el canal físico a nivel de flit. El protocolo del canal físico debe ser capaz de distinguir entre los canales virtuales que usan el canal físico. Lógicamente, cada canal virtual funciona como si estuviera utilizando un canal físico distinto operando a mitad de velocidad. Los canales virtuales fueron originariamente introducidos para resolver el problema de bloqueo en las redes con conmutación segmentada. El bloqueo en una red ocurre cuando los mensajes no pueden avanzar debido a que cada mensaje necesita un canal ocupado por otro mensaje. Discutiremos este problema en detalle en la siguiente sección.

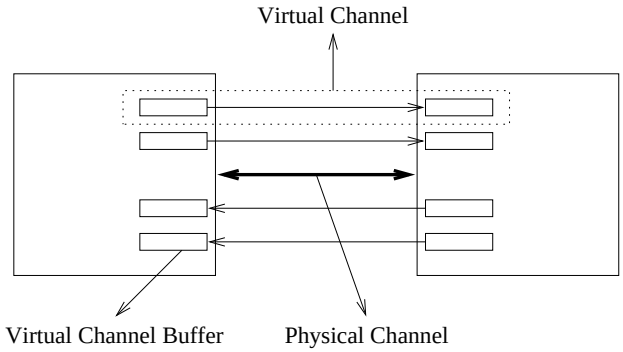


Figura 7.26: Canales virtuales.

Los canales virtuales también se pueden utilizar para mejorar la latencia de los mensajes y el rendimiento de la red. Al permitir que los mensajes compartan un canal físico, los mensajes pueden progresar en lugar de quedarse bloqueados. Por ejemplo, la figura 7.27 muestra a los mensajes cruzando el canal físico entre los encaminadores $R1$ y $R2$. Sin canales virtuales, el mensaje A impide al mensaje B avanzar hasta que la transmisión del mensaje A si hubiera completado. Sin embargo, en la figura existen dos canales virtuales multiplexados sobre cada canal físico. De esta forma, ambos mensajes pueden continuar avanzando. La velocidad a la cual cada mensaje avanza es nominalmente la mitad de la velocidad conseguida cuando el canal no está compartido. De hecho, el uso de canales virtuales desacopla los canales físicos de los buffers de mensajes permitiendo que varios mensajes compartan un canal físico de la misma manera que varios programas pueden compartir un procesador. El tiempo total que un mensaje permanece bloqueado en un encaminador a la espera de un canal libre se reduce, dando lugar a una reducción total en la latencia de los mensajes. Existen dos casos específicos donde la compartición del ancho de banda de un canal físico es particularmente beneficiosa. Consideremos el caso donde el mensaje A está temporalmente bloqueado en el nodo actual. Con un protocolo de control de flujo de los canales físicos apropiado, el mensaje B puede hacer uso de la totalidad del ancho de banda del canal físico entre los encaminadores. Sin canales virtuales, ambos mensajes estarían bloqueados. Consideremos ahora el caso en donde mensaje A es mucho más grande, en comparación, que el mensaje B. El mensaje B puede continuar a mitad de la velocidad del enlace, y a continuación el mensaje A puede continuar la transmisión utilizando todo el ancho de

banda del enlace.

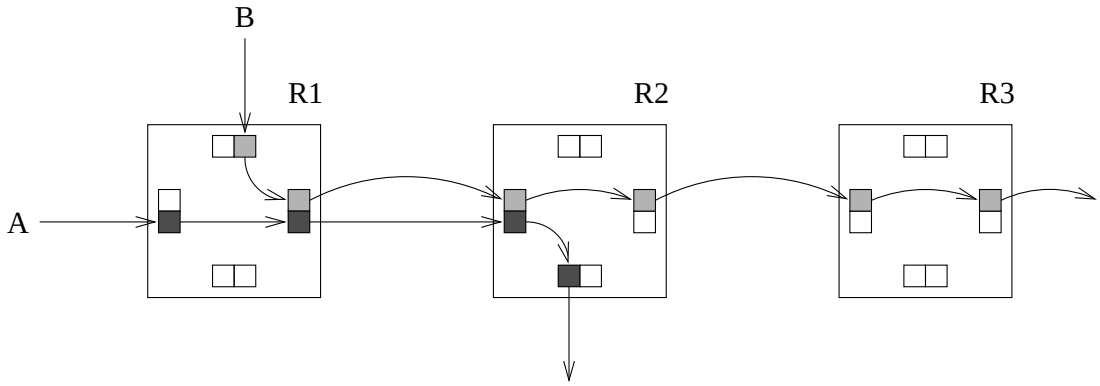


Figura 7.27: Un ejemplo de la reducción del retraso de la cabecera usando dos canales virtuales por canal físico.

Podríamos imaginarnos añadiendo más canales virtuales para conseguir una mayor reducción al bloqueo experimentado por cada mensaje. El resultado sería un incremento del rendimiento de la red medido en flits/s, debido al incremento de la utilización de los canales físicos. Sin embargo, cada canal virtual adicional mejora el rendimiento en una cantidad menor, y el incremento de la multiplexación de canales reduce la velocidad de transferencia de los mensajes individuales, incrementando la latencia del mensaje. Este incremento en la latencia debido a la multiplexación sobrepasará eventualmente a la reducción de la latencia debido al bloqueo dando lugar a un incremento global de la latencia media de los mensajes.

El incremento en el número de canales virtuales tiene un impacto directo en el rendimiento del encaminador debido a su impacto en el ciclo de reloj hardware. El controlador de los enlaces se hace cada vez más complejo dado que debe soportar el arbitraje entre varios canales virtuales. El número de entradas y salidas que deben de conmutarse en cada nodo aumenta, incrementando sustancialmente la complejidad del conmutador. Para una cantidad fija de espacio buffer en nodo, ¿cómo se asigna dicho espacio entre los canales?. Además, el flujo de mensajes a través del encaminador debe coordinarse con la asignación del ancho de banda del canal físico. El incremento de la complejidad de estas funciones puede dar lugar a incremento en las latencias de control de flujo interno y externo. Este incremento afecta a todos los mensajes que pasan a través de los encaminadores.

7.2.8 Mecanismos híbridos de conmutación

Conmutación encauzada de circuitos

Conmutación de exploración

7.2.9 Comparación de los mecanismos de conmutación

En esta sección, comparamos el rendimiento de varias técnicas de conmutación. En particular, analizamos el rendimiento de redes que utilizan conmutación de paquetes, VCT,

y conmutación segmentada (*wormhole switching*). VCT y la conmutación segmentada tienen un comportamiento similar a baja carga.

Para conmutación de paquetes, consideraremos buffers laterales con capacidad para cuatro paquetes. Para VCT, consideraremos buffers con capacidades para uno, dos y cuatro paquetes. Para la conmutación segmentada, se mostrará el efecto de añadir canales virtuales. El número de canales virtuales varía de uno a cuatro. En comparación, la capacidad de los buffers asociados a cada canal virtual se mantiene constante (4 flits) sin importar el número de canales virtuales. Por lo tanto, al aumentar los canales virtuales se aumenta también la capacidad total de almacenamiento asociada con cada canal físico. El efecto de añadir canales virtuales a la vez que se mantiene la capacidad total de almacenamiento constante se estudiará posteriormente.

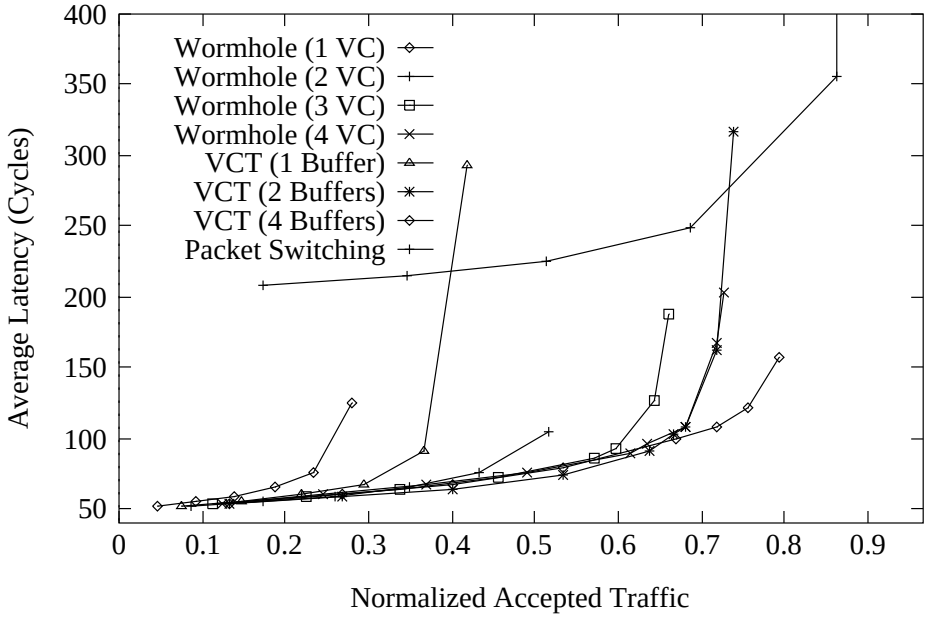


Figura 7.28: Latencia media del paquete vs. tráfico aceptado normalizado en una malla 16×16 para diferentes técnicas de conmutación y capacidades de buffer. (VC = Virtual channel; VCT = Virtual cut-through.)

La figura muestra la latencia media de un paquete en función del tráfico aceptado normalizado para diferentes técnicas de conmutación en un malla de 16×16 usando un encaminamiento por dimensiones, paquetes de 16 flits y una distribución uniforme del destino de los mensajes. Como era de esperar de la expresión de la latencia que obtuvimos para las diferentes técnicas de conmutación, VCT y la conmutación segmentada presentan la misma latencia cuando el tráfico es bajo. Esta latencia es mucho menor que la obtenida mediante la conmutación de paquetes. Sin embargo, al incrementarse el tráfico la conmutación segmentada sin canales virtuales satura rápidamente la red, dando lugar a una baja utilización de los canales.

Esta baja utilización de canales puede mejorarse añadiendo canales virtuales. Al añadir canales virtuales, el rendimiento de la red (*throughput*) también aumenta. Como puede observarse, el añadir nuevos canales virtuales da lugar a una mejora en el rendimiento cada vez menor. De forma similar, al incrementar el tamaño de la cola en la técnica de VCT da lugar a un aumento considerable del rendimiento de la red. Una observación interesante es que la latencia media para VCT y la conmutación segmentada

con canales virtuales es casi idéntica para todo el rango de carga aplicada hasta que las curvas alcanza el punto de saturación. Además, cuando la capacidad de almacenamiento por canal físico es la misma que en el VCT, la conmutación segmentada con canales virtuales consigue un mayor rendimiento de la red. Este es el caso del rendimiento de la conmutación segmentada con cuatro canales virtuales frente a la técnica de VCT con capacidad de un único paquete por canal físico.

Cuando la conmutación VCT se implementa utilizando colas laterales con capacidad para varios paquetes, los canales se liberan después de transmitir cada paquete. Por lo tanto, un paquete bloqueado no impide el uso del canal por parte de otros paquetes. Como una consecuencia de este hecho, el rendimiento de la red es mayor que en el caso de la conmutación segmentada con cuatro canales virtuales. Obsérvese, sin embargo, que la mejora es pequeña, a pesar del hecho de que la capacidad de almacenamiento para la conmutación VCT es dos o cuatro veces la capacidad existente en la conmutación segmentada. En particular, obtenemos el mismo resultado para la conmutación segmentada con cuatro canales virtuales y la conmutación VCT con dos buffers por canal. En el caso de paquetes largos, y manteniendo constante el tamaño de los buffers para la conmutación segmentada, los resultados son más favorables para la conmutación VCT pero las necesidades de almacenamiento se incrementan de forma proporcional.

Finalmente, cuando la red alcanza el punto de saturación, la conmutación VCT tiene que almacenar los paquetes muy frecuentemente, con lo que desaparece la segmentación. Como consecuencia, las técnicas de VCT y conmutación de paquetes con el mismo número de buffers consiguen un rendimiento similar cuando la red alcanza el punto de saturación.

La conclusión más importante es que la conmutación segmentada (*wormhole routing*) es capaz de conseguir latencias y rendimiento comparables a los del VCT, en el caso de existir suficientes canales virtuales y teniendo una capacidad de almacenamiento similar. Si la capacidad de almacenamiento es mayor para la técnica de VCT entonces esta técnica de conmutación consigue un mejor rendimiento pero la diferencia es pequeña si se utilizan suficientes canales virtuales en la conmutación segmentada. Una ventaja adicional de la conmutación segmentada es que es capaz de manejar mensajes de cualquier tamaño sin dividirlos en paquetes lo que no ocurre en el caso de VCT, especialmente cuando los buffers se implementan en hardware.

Aunque el añadir nuevos canales virtuales incrementa el tiempo de encaminamiento, haciendo disminuir la frecuencia del reloj, también se pueden realizar consideraciones similares al añadir espacio de almacenamiento en la conmutación VCT. A partir de ahora nos centraremos en redes que utilizan conmutación segmentada a no ser que se indique lo contrario.

7.3 La capa de encaminamiento (*routing*)

En esta sección estudiaremos los algoritmos de encaminamiento. Estos algoritmos establecen el camino que sigue cada mensaje o paquete. La lista de algoritmos propuestos en la literatura es casi interminable. Nosotros nos centraremos en aquellos que han sido usados o propuestos en los multiprocesadores actuales o futuros.

Muchas de las propiedades de la red de interconexión son una consecuencia directa del algoritmo de encaminamiento usado. Entre estas propiedades podemos citar las siguientes:

- *Conectividad*. Habilidad de encaminar paquetes desde cualquier nodo origen a cualquier nodo de destino.
- *Adaptabilidad*. Habilidad de encaminar los paquetes a través de caminos alternativos en presencia de contención o componentes defectuosos.
- *Libre de bloqueos (deadlock y livelock)*. Habilidad de garantizar que los paquetes no se bloquearán o se quedarán esperando en la red para siempre.
- *Tolerancia fallos*. Habilidad de encaminar paquetes en presencia de componentes defectuosos. Aunque podría parecer que la tolerancia fallos implica a la adaptabilidad, esto no es necesariamente cierto. La tolerancia a fallos puede conseguirse sin adaptabilidad mediante en encaminamiento de un paquete en dos o más fases, almacenándolo en algún nodo intermedio.

7.3.1 Clasificación de los algoritmos de encaminamiento

La figura 7.29 muestra una taxonomía de los algoritmos encaminamiento. Los algoritmos de encaminamiento se pueden clasificar de acuerdo a varios criterios. Estos criterios se indican en la columna izquierda en itálica. Cada fila contiene aproximaciones alternativas que pueden usarse para cada criterio. Las flechas indican las relaciones existentes entre las diferentes aproximaciones. Los algoritmos de encaminamiento pueden clasificarse en primer lugar en función del número de destinos. Los paquetes pueden tener un único destino (unicast routing) o varios destinos (multicast routing).

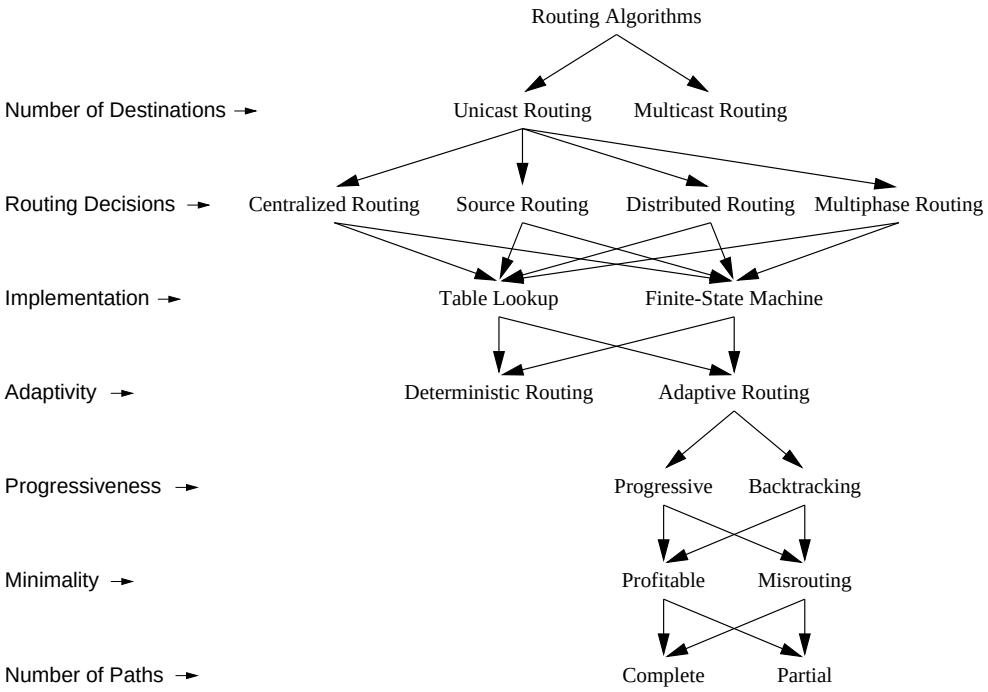


Figura 7.29: Una taxonomía de los algoritmos de encaminamiento

Los algoritmos de encaminamiento también pueden clasificarse de acuerdo con el lugar en donde se toman las decisiones de encaminamiento. Básicamente, el camino puede establecerse o bien por un controlador centralizado (encaminamiento centralizado), en el nodo origen antes de la inyección del paquete (encaminamiento de origen) o

ser determinado de una manera distribuida mientras el paquete atraviesa la red (encaminamiento distribuido). También son posibles esquemas híbridos. A estos esquemas híbridos se les denominan encaminamiento multifase. En el encaminamiento multifase, el nodo origen determina algunos de los nodos destinos. El camino entre estos se establece de una manera distribuida y el paquete puede ser enviado a todos los nodos destinos calculados (*multicast routing*) o únicamente al último de los nodos destino (*unicast routing*). En este caso, los nodos intermedios se usan para evitar la congestión o los fallos.

Los algoritmos de encaminamiento pueden implementarse de varias maneras. Entre ellas, las más interesantes son el uso de una tabla de encaminamiento (*table-lookup*) o la ejecución de un algoritmo de encaminamiento de acuerdo con una máquina de estados finita. En este último caso, los algoritmos puede ser deterministas o adaptativos. Los algoritmos de encaminamiento deterministas siempre suministran el mismo camino entre un nodo origen y un nodo destino. Los algoritmos adaptativos usan información sobre el tráfico de la red y/o el estado de los canales para evitar la congestión o las regiones con fallos de la red.

Los algoritmos de encaminamiento pueden clasificarse de acuerdo con su progresividad como *progresivos* o con *vuelta atrás*. En los algoritmos de encaminamiento progresivos la cabecera siempre sigue hacia delante reservando un nuevo canal en cada operación de encaminamiento. En los algoritmos con vuelta atrás la cabecera puede retroceder hacia atrás, liberando canales reservados previamente.

A un nivel más bajo, los algoritmos de encaminamiento pueden clasificarse dependiendo de su minimalidad como aprovechables (*profitables*) o con desencaminamientos (*misrouting*). Los algoritmos de encaminamiento aprovechables sólo proporcionan canales que acercan al paquete a su destino. También se les denominan *mínimos*. Los algoritmos con desencaminamientos pueden proporcionar además algunos canales que alejen al paquete su destino. A estos últimos también se les denominan *no mínimos*. Al nivel más bajo, los algoritmos de encaminamiento se pueden clasificar dependiendo del número de caminos alternativos como completamente adaptativos (también conocidos como *totalmente adaptativos*) o *parcialmente adaptativos*.

7.3.2 Bloqueos

En las redes directas, los paquetes deben atravesar varios nodos intermedios antes de alcanzar su destino. En las redes basadas en conmutadores, los paquetes atraviesan varios conmutadores antes de alcanzar su destino. Sin embargo, puede ocurrir que algunos paquetes no puedan alcanzar su destino, incluso existiendo un camino libre de fallos entre los nodos origen y destino para cada paquete. Suponiendo que existe un algoritmo de encaminamiento capaz de utilizar esos caminos, existen varias situaciones que pueden impedir la recepción del paquete. En este apartado estudiaremos este problema y las soluciones existentes.

Como se vio en la sección anterior, se necesita espacio buffer para permitir el almacenamiento de fragmentos de un paquete, o incluso la totalidad del mismo, en cada nodo intermedio o conmutador. Sin embargo, dado que existe un coste asociado a la existencia de dichos buffers, la capacidad de los mismos es finita. Al permitir que un paquete cuya cabecera no ha llegado a su destino solicite buffers adicionales al mismo tiempo que mantiene los buffers que almacenan en la actualidad el paquete, puede surgir una situación de bloqueo. Un *bloqueo mortal* (deadlock) ocurre cuando algu-

nos paquetes no pueden avanzar hacia su destino ya que los buffers que solicitan están ocupados. Todos los paquetes involucrados en una configuración de bloqueo mortal están bloqueados para siempre. Obsérvese que un paquete puede bloquearse en la red porque el nodo destino no lo consuma. Esta clase de bloqueo se produce a causa de la aplicación, estando fuera del ámbito que a nosotros nos interesa. En lo que queda de sección supondremos que los paquetes siempre se consumen cuando llegan a su nodo destino en un tiempo finito. La figura 7.30 muestra un ejemplo de esta situación.

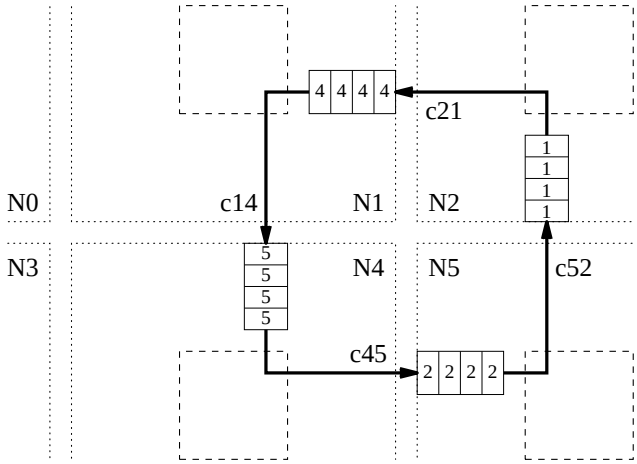


Figura 7.30: Configuración de bloqueo en una malla 2-D.

Una situación diferente surge cuando algunos paquetes no pueden llegar a su destino, incluso aunque no estén bloqueados permanentemente. Un paquete puede estar viajando alrededor del nodo destino sin llegar nunca a alcanzarlo porque los canales que necesitan están ocupados por otros paquetes. A esta situación se le conoce con el nombre de *bloqueo activo* (livelock) y sólo puede ocurrir en el caso de que los paquetes puedan seguir caminos no mínimos.

Finalmente, un paquete puede permanecer completamente parado si el tráfico es intenso y los recursos solicitados son asignados siempre a otros paquetes que lo solicitan. Esta situación, conocida como *muerte por* inanición (starvation), suele deberse a una asignación incorrecta de los recursos en el caso de conflicto entre dos o más paquetes.

Es muy importante eliminar los diferentes tipos de bloqueos (*deadlock*, *livelock* y *starvation*) al implementar un red de interconexión. En caso contrario, algunos paquetes nunca llegarían a su destino. Dado que estas situaciones surgen debido a la limitación en los recursos de almacenamiento, la probabilidad de llegar a una de estas situaciones aumenta al aumentar el tráfico de la red y decrece con la cantidad de espacio de almacenamiento. Por ejemplo, una red que utilice una conmutación segmentada es mucho más sensible a los bloqueos que la misma red utilizando el mecanismo de conmutación de almacenamiento y reenvío (SAF) en el caso de que el algoritmo de encaminamiento no sea libre de bloqueos.

En la figura 7.31 se muestra una clasificación de las situaciones que pueden impedir la llegada de un paquete y las técnicas existentes para resolver estas situaciones. La muerte por inanición (*starvation*) es relativamente sencilla de resolver. Solamente es necesario usar un esquema de asignación de recursos correcto. Por ejemplo, un esquema de asignación de recursos basado en una cola circular. Si permitimos la existencia de distintas prioridades, será necesario reservar parte del ancho de banda para paquetes

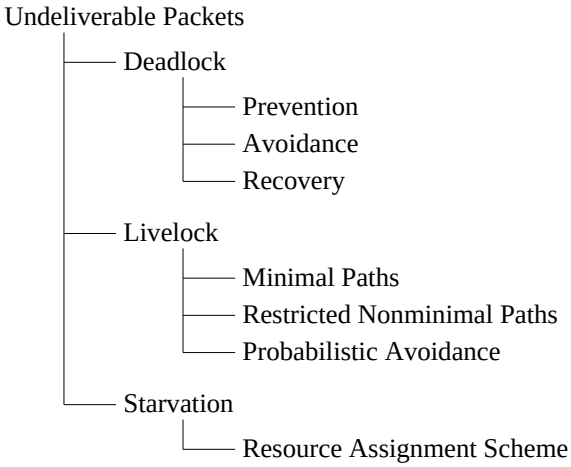


Figura 7.31: Una clasificación de las situaciones que pueden impedir el envío de paquetes.

de baja prioridad para de esta forma evitar la muerte por inanición. Esto se puede conseguir limitando el número de paquetes de mayor prioridad, o reservando algunos canales virtuales o buffers para los paquetes de baja prioridad.

El bloqueo activo (*livelock*) también es fácil de evitar. El modo más simple consiste en usar únicamente caminos mínimos. Esta restricción incrementa el rendimiento en las redes que usan conmutación segmentada ya que los paquetes no ocupan más canales que los estrictamente necesarios para alcanzar su destino. La principal motivación de usar caminos no mínimos es la tolerancia a fallos. Incluso en el caso de usar caminos no mínimos, el bloqueo se puede prevenir limitando el número de encaminamientos que no nos acercan al destino. Otro motivo para usar caminos no mínimos es la evitación del *deadlock* mediante el encaminamiento mediante desviación. En este caso, el encaminamiento es probabilísticamente libre de bloqueos.

El bloqueo mortal (*deadlock*) es de lejos el problema más difícil de resolver. Existen tres estrategias para tratar este problema: *prevención* del bloqueo, *evitación* del bloqueo y *recuperación* del bloqueo[1]. En la prevención del *deadlock*, los recursos (canales o buffers) son asignados a un paquete de tal manera que una petición nunca da lugar a una situación de bloqueo. Esto se puede conseguir reservando todos los recursos necesarios antes de empezar la transmisión del paquete. Este es el caso de todas las variantes de la técnica de conmutación de circuitos en el caso de permitir la vuelta hacia atrás (*backtracking*). En la evitación del *deadlock*, los recursos son asignados a un paquete al tiempo que este avanza a través de la red. Sin embargo, un recurso se le asigna a un paquete únicamente si el estado global resultante es seguro. Este estrategia evita el envío de nuevos paquetes para actualizar el estado global por dos razones: porque consumen ancho de banda y porque pueden contribuir a su vez a producir una situación de bloqueo. Conseguir que el estado global sea seguro de forma distribuida no es una tarea sencilla. La técnica comúnmente utilizada consiste en establecer un orden entre los recursos y garantizar que los recursos son asignados a los paquetes en orden decreciente. En las estrategias de recuperación, los recursos se asignan a paquetes sin ningún chequeo adicional. En este caso son posibles las situaciones de bloqueo, y se hace necesario un mecanismo de detección de las mismas. Si se detecta un *deadlock*, se

[1] En el artículo *Deadlock detection in distributed systems* de M. Singhal se utiliza el término detección en lugar de recuperación.

liberan algunos de los recursos que son reasignados a otros paquetes. Para la liberación de estos recursos, se suele proceder a la eliminación de los paquetes que tenían dichos recursos.

Las estrategias de prevención de bloqueos mortales son muy conservadoras. Sin embargo, la reserva de todos los recursos antes de empezar la transmisión del paquete puede dar lugar a una menor utilización de recursos. Las estrategias de evitación de bloqueos mortales son menos conservadoras, solicitando los recursos cuando son realmente necesarios para enviar el paquete. Finalmente, las estrategias de recuperación son optimistas. Únicamente pueden ser usadas si los bloqueos son raros y el resultado de los mismos puede ser tolerado.

7.3.3 Teoría para la evitación de bloqueos mortales (*deadlocks*)

El objetivo de este apartado es llegar a una condición necesaria y suficiente para conseguir un encaminamiento libre de bloqueos en redes directas utilizando SAF, VCT o conmutación segmentada. El resultado para la conmutación segmentada también puede aplicarse a la conmutación mediante la técnica del cartero loco (*mad postman*) realizando la suposición de que los flits muertos son eliminados de la red tan pronto como se bloquean. Las condiciones necesarias y suficientes para la conmutación segmentada se convierte en una condición suficiente en el caso de método del explorador (*scouting switching*).

Definiciones básicas

La red de interconexión I se modela usando un grafo conexo con múltiples arcos, $I = G(N, C)$. Los vértices del grafo, N, representan el conjunto de nodos de procesamiento. Los arcos del grafo, C, representa el conjunto de canales de comunicación. Se permite la existencia de más de un canal entre cada par de nodos. Los canales bidireccionales se consideran como dos canales unidireccionales. Nos referiremos a un canal y al buffer asociado al mismo de forma indistinta. Los nodos origen y destino de un canal c_i vienen denotados por s_i y d_i, respectivamente.

Un algoritmo de encaminamiento se modela mediante dos funciones: encaminamiento y selección. La *función de encaminamiento* devuelve un conjunto de canales de salida basándose en el nodo actual y el nodo destino. La selección del canal de salida, basándose en el estado de los canales de salida y del nodo actual, a partir de este conjunto se realiza mediante la *función de selección*. Si todos todos los canales de salida están ocupados, el paquete se encaminará de nuevo hasta que consiga reservar un canal. Como veremos, la función de encaminamiento determina si el algoritmo de encaminamiento es o no libre de bloqueos. La función de selección únicamente afecta al rendimiento. Obsérvese que en nuestro modelo el dominio de la función de selección es $N \times N$ ya que sólo tiene en consideración el nodo actual y el nodo destino. Por lo tanto, no se considera el camino seguido por el paquete a la hora de calcular el siguiente canal a utilizar.

Para conseguir unos resultados teóricos tan generales como sean posibles, no supondremos ninguna restricción acerca de la tasa de generación de paquetes, destino de los mismos, y longitud de los mismos. Tampoco supondremos ninguna restricción en

los caminos suministrados por el algoritmo de encaminamiento. Se permiten tanto los caminos mínimos como los no mínimos. Sin embargo, y por razones de rendimiento, un algoritmo de encaminamiento debe proporcional al menos un canal perteneciente a un camino mínimo en cada nodo intermedio. Además, nos centraremos en los bloqueos producidos por la red de interconexión. Por lo tanto, supondremos que los paquetes se consumen en los nodos destinos en un tiempo finito.

Consideraremos varias técnicas de conmutación. Cada una de ellas pueden verse como un caso particular de la teoría general. Sin embargo, serán necesario realizar una serie de suposiciones para algunas de estas técnicas. Para la conmutación segmentada supondremos que una cola no puede contener flits pertenecientes a paquetes diferentes. Después de aceptar el último flit, una cola debe vaciarse antes de aceptar otro flit cabecera. Cuando un canal virtual tiene colas a ambos lados, ambas colas deben vaciarse antes de aceptar otro flit cabecera. Así, si un paquete se bloquea, su flit cabecera siempre ocupará la cabeza de una cola. Además, para cada camino P que pueda establecerse mediante una función de encaminamiento R, todos los subcaminos de P deben de ser también caminos de R. A las funciones de encaminamiento que satisfacen esta última propiedad se les denominan *coherentes*. Para la técnica del cartero loco supondremos las mismas restricciones que en la conmutación segmentada. Además, los flits muertos se eliminan de la red tan pronto como se bloquean.

Una *configuración* es una asignación de un conjunto de paquetes o flits a cada cola. La configuración que se mostró en la figura 7.30 es un ejemplo de *configuración bloqueada*. Esta configuración también estaría bloqueada si existieran paquetes adicionales viajando por la red y que no estén bloqueados. Una configuración bloqueada en donde todos los paquetes están bloqueados se denomina *canónica*. Dada una configuración bloqueada, la correspondiente configuración canónica puede obtenerse deteniendo la inyección de paquetes en todos los nodos, y esperando a que lleguen todos los paquetes que no están bloqueados. Desde un punto de vista práctico, sólo es necesario considerar configuraciones bloqueadas canónicas.

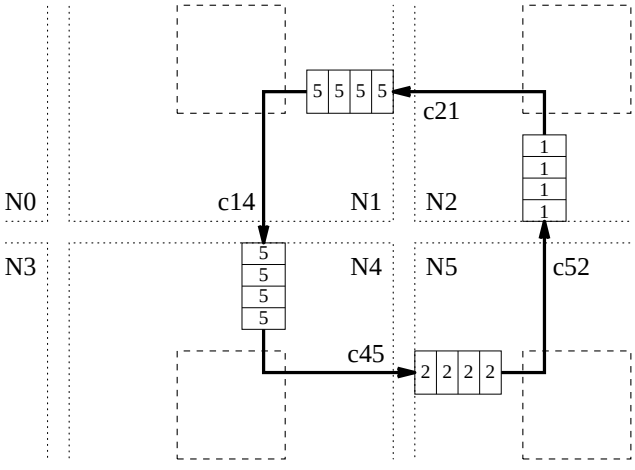


Figura 7.32: Configuración ilegal para R.

Observar que una configuración describe el estado de una red de interconexión en un instante dado. Así, no es necesario considerar configuraciones que describen situaciones imposibles. Una configuración es *legal* si describe una situación posible. En particular, no consideraremos configuraciones en donde se exceda la capacidad del buffer. Así, un paquete no puede ocupar un canal dado al menos que la función de encaminamiento lo

proporcione para el destino de ese paquete. La figura 7.32 muestra una configuración ilegal para una función de encaminamiento R en una malla bidireccional que envía los paquetes por cualquier camino mínimo.

En resumen, una *configuración de bloqueo mortal canónica* es una configuración legal en donde ningún paquete puede avanzar. Si se utilizan las técnicas de conmutación SAF ó VCT:

- Ningún paquete ha llegado a su nodo destino.
- Los paquetes no pueden avanzar debido a que las colas de todos los canales de salida alternativos suministrados por la función de encaminamiento están llenos.

Si se está utilizando el encaminamiento segmentado:

- No hay ningún paquete cuyo flit cabecera haya llegado a su destino.
- Los flits cabecera no pueden avanzar porque las colas de todos los canales de salida alternativos suministrados por la función de encaminamiento no están vacíos (recordar que hemos realizado la suposición de que una cola no puede contener flits pertenecientes a diferentes paquetes).
- Los flits de datos no pueden avanzar porque el siguiente canal reservado por sus paquetes cabecera tiene su cola llena. Observar que un flit de datos puede estar bloqueado en un nodo incluso si hay canales de salida libres para alcanzar su destino ya que los flits de datos deben seguir el camino reservado por su cabecera.

En algunos casos, una configuración no puede alcanzarse mediante el encaminamiento de paquetes a partir de una red vacía. Esta situación surge cuando dos o más paquetes precisan del uso del mismo canal al mismo tiempo para alcanzar dicha configuración. Una configuración que puede alcanzarse mediante el encaminamiento de paquetes a partir de una red vacía se dice que es *alcanzable* o *encaminable*. Hacer notar que de la definición del dominio de la función de encaminamiento como $N \times N$, toda configuración legal es también alcanzable. Efectivamente, dado que la función de encaminamiento no tiene memoria del camino seguido por cada paquete, podemos considerar que, para cualquier configuración legal, un paquete almacenado en la cola de un canal fue generado por el nodo origen de ese canal. En la conmutación segmentada, podemos considerar que el paquete fue generado por el nodo origen del canal que contiene el último flit del paquete. Esto es importante ya que cuando todas las configuraciones legales son alcanzables, no necesitamos considerar la evolución dinámica de la red que dio lugar a dicha configuración. Podemos considerar simplemente las configuraciones legales, sin tener en cuenta de la secuencia de inyección de paquetes necesaria para alcanzar dicha configuración. Cuando todas las configuraciones legales son alcanzables, una función de encaminamiento se dice libre de bloqueos mortales (*deadlock-free*) si, y sólo si, no existe una configuración de bloqueo mortal para dicha función de encaminamiento.

Condición necesaria y suficiente

El modelo teórico para la evitación de bloqueos que vamos a estudiar a continuación se basa en el concepto de *dependencia entre canales*. Cuando un paquete está ocupando un canal y a continuación solicita el uso de otro canal, existe una dependencia entre dichos canales. Ambos canales están en uno de los caminos que puede seguir el paquete. Si se usa conmutación segmentada (*wormhole switching*), dichos canales no tienen por qué ser adyacentes ya que un paquete puede estar ocupando varios ca-

nales simultáneamente. Además, en un nodo dado, un paquete puede solicitar el uso de varios canales para después seleccionar uno de ellos (encaminamiento adaptativo). Todos los canales solicitados son candidatos a la selección. Así, todos los canales solicitados producen dependencias, incluso si no son seleccionados en una operación de encaminamiento determinada. Veámoslo con un ejemplo:

Consideremos un anillo unidireccional con cuatro nodos denotados por n_i, $i = \{0, 1, 2, 3\}$ y un canal unidireccional conectando cada par de nodos adyacentes. Sea c_i, $i = \{0, 1, 2, 3\}$, el canal de salida del nodo n_i. En este caso, es sencillo definir una función de encaminamiento conexa. Se puede enunciar como sigue: Si el nodo actual n_i es igual al nodo destino n_j, almacenar el paquete. En caso contrario, usar c_i, $\forall j \neq i$. La figura 7.33(a) muestra la red. Existe una dependencia cíclica entre los canales c_i. En efecto, un paquete en el nodo n_0 destinado al nodo n_2 puede reservar c_0 y después solicitar c_1. Un paquete en el nodo n_1 destinado al nodo n_3 puede reservar c_1 y después solicitar c_2. Un paquete en el nodo n_2 destinado al nodo n_0 puede reservar c_2 y después solicitar c_3. Finalmente, un paquete en el nodo n_3 destinado al nodo n_1 puede reservar c_3 y después solicitar c_0. Es fácil de ver que una configuración conteniendo los paquetes arriba mencionados no es libre de bloqueos ya que cada paquete tiene reservado un canal y está esperando un canal ocupado por otro paquete.

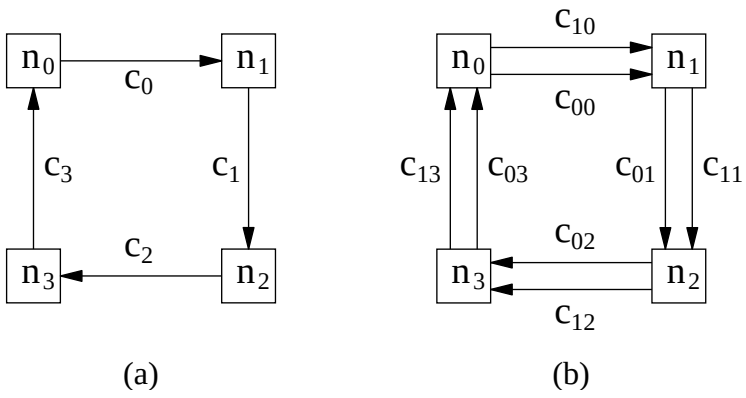


Figura 7.33: Redes del ejemplo anterior.

Consideremos ahora que cada canal físico c_i se divide en dos canales virtuales, c_{0i} y c_{1i}, como se muestra en la figura 7.33(b). La nueva función de encaminamiento puede enunciarse como sigue: Si el nodo actual n_i es igual al nodo destino n_j, almacenar el paquete. En caso contrario, usar c_{0i}, si $j < i$ o c_{1i}, si $j > i$. Como se puede ver, la dependencia cíclica ha sido eliminada ya que después de usar el canal c_{03}, se alcanza el nodo n_0. Así, como todos los destinos tienen un índice mayor que n_0, no es posible pedir el canal c_{00}. Observar que los canales c_{00} y c_{13} nunca son usados. Además, la nueva función de encaminamiento está libre de bloqueos mortales. Veamos que no existe ninguna configuración de bloqueo mortal intentando construir una. Si hubiese un paquete almacenado en la cola del canal c_{12}, estaría destinado a n_3 y los flits podrían avanzar. Así que c_{12} debe estar vacío. Además, si

hubiese un paquete almacenado en la cola de c_{11}, estaría destinado a n_2 ó n_3. Como c_{12} está vacío, los flits pueden avanzar y c_{11} debe estar vacío. Si hubiese un paquete almacenado en la cola de c_{10}, estaría destinado a n_1, n_2 o n_3. Como c_{11} y c_{12} están vacíos, los flits pueden avanzar y c_{10} debe estar vacío. De forma similar, se puede demostrar que el resto de canales pueden vaciarse.

Cuando se considera encaminamiento adaptativo, los paquetes tienen normalmente varias opciones en cada nodo. Incluso si una de estas elecciones es un canal usado por otro paquete, pueden estar disponibles otras elecciones de encaminamiento. Así, no es necesario eliminar todas las dependencias cíclicas, supuesto que cada paquete puede encontrar siempre un camino hacia su destino utilizando canales que no estén involucrados en dependencias cíclicas. Esto se muestra con el siguiente ejemplo:

Consideremos un anillo unidireccional con cuatro nodos denotados por n_i, $i = \{0, 1, 2, 3\}$ y dos canales unidireccionales conectando cada par de nodos adyacentes, excepto los nodos n_3 y n_0 que están enlazados con un único canal. Sea c_{Ai}, $i = \{0, 1, 2, 3\}$ y c_{Hi}, $i = \{0, 1, 2\}$ los canales de salida del nodo n_i. La función de encaminamiento puede formularse como sigue: Si el nodo actual n_i es igual al nodo destino n_j, almacenar el paquete. En caso contrario, usar o bien c_{Ai}, $\forall j \neq i$ o bien c_{Hi}, $\forall j > i$. La figura 7.34 muestra la red.

Existen dependencias cíclicas entre los canales c_{Ai}. Efectivamente, un paquete en el nodo n_0 destinado al nodo n_2 puede reservar c_{A1} y después solicitar c_{A2} y c_{H2}. Un paquete en el nodo n_2 destinado a n_0 puede reservar c_{A2} y después solicitar c_{A3}. Finalmente, un paquete en el nodo n_3 destinado al nodo n_1 puede reservar c_{A3} y después solicitar c_{A0} y c_{H0}.

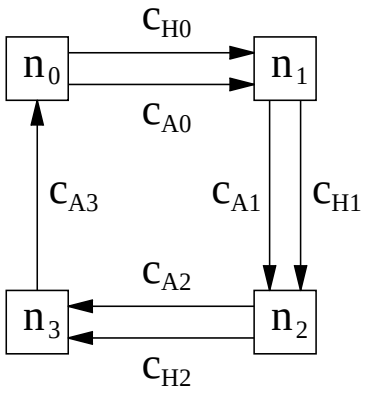


Figura 7.34: Red del ejemplo anterior.

Sin embargo, la función de encaminamiento está libre de bloqueos mortales. Aunque nos centraremos en la conmutación segmentada, el siguiente análisis también es válido para otras técnicas de conmutación. Veamos que no existe ninguna configuración de bloqueo intentando construir una. Si hubiese un paquete almacenado en la cola del canal c_{H2}, estaría destinado a n_3 y los flits podrían avanzar. Así, c_{H2} debe estar vacía. Además, si hubiese un paquete almacenado en la cola de c_{H1}, estaría destinado a n_2 ó n_3. Como c_{H2} está vacío, los flits pueden avanzar y c_{H1} se vaciará. Si hubiesen flits

almacenados en la cola de c_{H0}, estarán destinados a n_1, n_2 ó n_3. Incluso si su flit cabecera estuviese almacenado en c_{A1} ó c_{A2}, como c_{H1} y c_{H2} están vacíos, los flits pueden avanzar y c_{H0} se vaciará.

Así, no es posible establecer una configuración bloqueada utilizando únicamente canales c_{Ai}. Aunque existe una dependencia cíclica entre ellos, c_{A0} no puede contener flits destinados a n_0. Esta configuración no sería legal ya que n_0 no puede enviar paquetes hacia si mismo a través de la red. Para cualquier otro destino, estos flits pueden avanzar ya que c_{H1} y c_{H2} están vacíos. De nuevo, c_{A0} puede vaciarse, rompiendo la dependencia cíclica. Por lo tanto, la función de encaminamiento es libre de bloqueos mortales.

Este ejemplo muestra que los bloqueos pueden evitarse incluso cuando existen dependencias cíclicas entre algunos canales. Obviamente, si existieran dependencias cíclicas entre todos los canales de la red, no habrían caminos de escape para los ciclos. Así, la idea clave consiste en proporcionar un camino libre de dependencias cíclicas para escapar de los ciclos. Este camino puede considerarse como un camino de escape. Observar que al menos un paquete en cada ciclo debe poder seleccionar el camino de escape en el nodo actual, cualquiera que sea su destino. En nuestro caso, para cada configuración legal, el paquete cuyo flit cabecera se encuentra almacenado en el canal c_{A0} puede destinarse a n_1, n_2 o n_3. En el primer caso, este puede ser emitido de forma inmediata. En los otros dos casos, el paquete puede usar el canal c_{H1}.

Las dependencias entre los canales pueden agruparse para simplificar el análisis de los *deadlocks*. Una forma conveniente es el *grafo de dependencias entre canales*[2] que denotaremos por: $D = (G, E)$. Los vértices de D son los canales de la red de interconexión I. Los arcos de D son los pares de canales (c_i, c_j) tales que existe una dependencia de canales de c_i a c_j.

Teorema. *Una función de encaminamiento determinista, R, para una red de interconexión, I, está libre de bloqueos si, y sólo si, no existen ciclos en su grafo de dependencias entre canales, D.*

Prueba: $\Rightarrow$ Supongamos que una red tiene un ciclo en D. Dado que no existen ciclos de longitud 1, este ciclo debe tener una longitud de dos o más. Así, podemos construir una configuración de bloqueo llenando las colas de cada canal en el ciclo con flits destinados a un nodo a una distancia de dos canales, donde el primer canal en el encaminamiento pertenece al ciclo.

$\Leftarrow$ Supongamos que una red no tiene ciclos en D. Dado que D es acíclico, podemos asignar un orden total entre los canales de C de tal manera que si c_j puede ser utilizado inmediatamente después de c_i entonces $c_i > c_j$. Consideremos el menor canal en esta ordenación con una cola llena c_l. Cada canal c_n a los que c_l alimenta es menor que c_l y por lo tanto no tiene una cola llena. Así, ningún flit en la cola de c_l está bloqueado, y no tenemos una situación de bloqueo mortal.

[2]Este grafo fue propuesto por los investigadores Dally y Seitz en su artículo *Deadlock-free message routing in multiprocessor interconnection networks*. IEEE Transactions on Computers, vol. C-36, no. 5, pp. 547-553, May 1987.

7.3.4 Algoritmos deterministas

Los algoritmos de encaminamiento deterministas establecen el camino como una función de la dirección destino, proporcionando siempre el mismo camino entre cada par de nodos. Debemos diferenciar el encaminamiento determinista del encaminamiento inconsciente (*oblivious*). Aunque ambos conceptos han sido considerados a veces como idénticos, en el último la decisión de encaminamiento es independiente del (es decir, inconsciente del) estado de la red. Sin embargo, la elección no es necesariamente determinista. Por ejemplo, una tabla de encaminamiento puede incluir varias opciones como canal de salida dependiendo de la dirección destino. Una opción específica puede seleccionarse aleatoriamente, cíclicamente o de alguna otra manera que sea independiente del estado de la red. Un algoritmo de encaminamiento determinista siempre proporcionaría el mismo canal de salida para el mismo destino. Mientras que un algoritmo determinista es inconsciente, lo contrario no es necesariamente cierto.

El encaminamiento determinista se hizo muy popular cuando Dally propuso el mecanismo de conmutación segmentada. La conmutación segmentada requiere muy poco espacio de buffer. Los encaminadores en este caso son compactos y rápidos. Sin embargo, la segmentación no funciona eficientemente si una de las etapas es mucho más lenta que el resto de etapas. Así, estos encaminadores tienen el algoritmo de encaminamiento implementado en hardware. No cabe, por tanto, sorprenderse que los diseñadores eligieran los algoritmos de encaminamiento más sencillos para de esta forma conseguir un encaminamiento hardware tan rápido y eficiente como fuera posible. La mayoría de los multicomputadores comerciales (Intel Paragon, Cray T3D, nCUBE-2/3) y experimentales (Stanford DASH, MIT J-Machine) usan un encaminamiento determinista.

En este apartado presentaremos los algoritmos de encaminamiento deterministas más populares. Obviamente, los algoritmos más populares son también los más simples.

Encaminamiento por orden de la dimensión

Como ya vimos, algunas topologías pueden descomponerse en varias dimensiones ortogonales. Este es el caso de los hipercubos, mallas, y toros. En estas topologías, es fácil calcular la distancia entre el nodo actual y el nodo destino como suma de las diferencias de posiciones en todas las dimensiones. Los algoritmos de encaminamiento progresivos reducirán una de estas diferencias en cada operación de encaminamiento. El algoritmo de encaminamiento progresivo más simple consiste en reducir una de estas diferencias a cero antes de considerar la siguiente dimensión. A este algoritmo de encaminamiento se le denomina encaminamiento por dimensiones. Este algoritmo envía los paquetes cruzando las dimensiones en un orden estrictamente ascendente (o descendente), reduciendo a cero la diferencia en una dimensión antes de encaminar el paquete por la siguiente.

Para redes n-dimensionales e hipercubos, el encaminamiento por dimensiones da lugar a algoritmos de encaminamiento libres de bloqueos mortales. Estos algoritmos son muy conocidos y han recibido varios nombres, como encaminamiento XY (para mallas 2-D) o *e-cubo* (para hipercubos). Estos algoritmos se describen en las figuras 7.35 y 7.36, respectivamente, donde *FirstOne()* es una función que devuelve la posición del primer bit puesto a uno, e *Internal* es el canal de conexión al nodo local. Aunque estos algoritmos asumen que la cabecera del paquete lleva la dirección absoluta del nodo destino, las primeras sentencias de cada algoritmo calculan la distancia del nodo

actual al nodo destino en cada dimensión. Este valor es el que llevaría la cabecera del paquete si se utilizase un direccionamiento relativo. Es fácil de demostrar que el grafo de dependencias entre canales para el encaminamiento por dimensiones en mallas n-dimensionales e hipercubos es acíclico.

Aunque el encaminamiento por orden de dimensión se suele implementar de manera distribuida usando una máquina de estados finita, también puede implementarse usando en encaminamiento fuente (*street-sign routing*) o una tabla de búsqueda distribuida (*interval routing*).

Bloqueos en toros

El grafo de dependencias entre canales para los toros tiene ciclos, como ya vimos para el caso de anillos unidireccionales en la figura 7.37. Esta topología fue analizada por Dally y Seitz, proponiendo una metodología para el diseño de algoritmos de encaminamiento deterministas a partir del teorema visto en el apartado 7.3.3.

Algorithm: XY Routing for 2-D Meshes
Inputs: Coordinates of current node $(Xcurrent, Ycurrent)$
and destination node $(Xdest, Ydest)$
Output: Selected output $Channel$
Procedure:
$Xoffset := Xdest - Xcurrent$;
$Yoffset := Ydest - Ycurrent$;
if $Xoffset < 0$ **then**
$Channel := X-$;
endif
if $Xoffset > 0$ **then**
$Channel := X+$;
endif
if $Xoffset = 0$ **and** $Yoffset < 0$ **then**
$Channel := Y-$;
endif
if $Xoffset = 0$ **and** $Yoffset > 0$ **then**
$Channel := Y+$;
endif
if $Xoffset = 0$ **and** $Yoffset = 0$ **then**
$Channel := Internal$;
endif

Figura 7.35: El algoritmo de encaminamiento XY para mallas 2-D.

La metodología comienza considerando una función de encaminamiento conexa y su grafo de dependencias entre canales D. Si éste no es acíclico, se restringe el encaminamiento eliminando arcos del grafo D para hacerlo acíclico. Si no es posible conseguir un grafo acíclico sin que la función de encaminamiento deje de ser conexa, se procede a añadir arcos a D asignando a cada canal físico un conjunto de canales virtuales. Utilizando esta metodología se establece una ordenación total entre los canales virtuales, que se etiquetan a continuación según esta ordenación. Cada vez que se rompe un ciclo dividiendo un canal físico en dos canales virtuales, se introduce un nuevo índice para

Algorithm: Dimension-Order Routing for Hypercubes
Inputs: Addresses of current node *Current*
and destination node *Dest*
Output: Selected output *Channel*
Procedure:
$offset := Current \oplus Dest$;
if $offset = 0$ **then**
$Channel := Internal$;
else
$Channel :=$ FirstOne($offset$);
endif

Figura 7.36: El algoritmo de encaminamiento por dimensiones para hipercubos.

establecer el orden entre los canales virtuales. Además, al eliminar un ciclo mediante la utilización de un nuevo canal virtual a cada canal físico, al nuevo conjunto de canales virtuales se le asigna un valor diferente del índice correspondiente. Veamos la aplicación de esta metodología a anillos unidireccionales y a n-cubos k-arios.

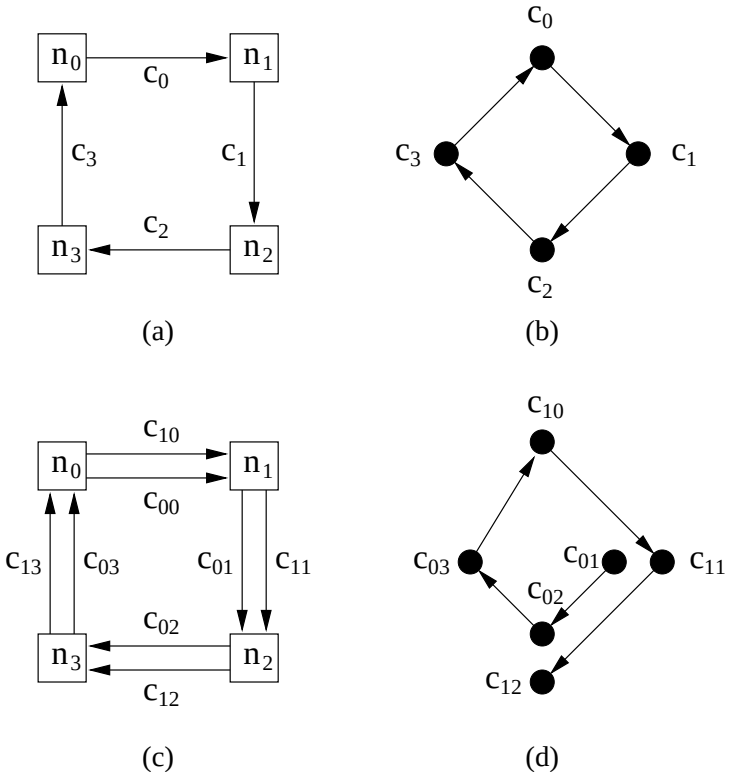


Figura 7.37: Grafo de dependencias entre canales para anillos unidireccionales.

Consideremos un anillo unidireccional con cuatro nodos denotados por n_i, $i = \{0, 1, 2, 3\}$ y un canal unidireccional conectando cada par de nodos adyacentes. Sea c_i, $i = \{0, 1, 2, 3\}$, el canal de salida del nodo n_i. En este caso, es sencillo definir una función de encaminamiento conexa. Se puede enunciar como sigue: Si el nodo actual n_i es igual al nodo destino n_j, almacenar

el paquete. En caso contrario, usar c_i, $\forall j \neq i$. La figura 7.37(a) muestra la red. La figura 7.37(b) muestra que el grafo de dependencias entre canales para esta función de encaminamiento presenta un ciclo. Así, siguiendo la metodología propuesta, cada canal físico c_i se divide en dos canales virtuales, c_{0i} y c_{1i}, como muestra la figura 7.37(c). Los canales virtuales se ordenan de acuerdo con sus índices. La función de encaminamiento se redefine para que utilice los canales virtuales en orden estrictamente decreciente. La nueva función de encaminamiento puede formularse como sigue: Si el nodo actual n_i es igual al nodo destino n_j, almacenar el paquete. En caso contrario, usar c_{0i}, si $j < i$ o c_{1i}, si $j > i$. La figura 7.37(d) muestra el grado de dependencias entre canales para esta función de encaminamiento. Como puede observarse, el ciclo se ha eliminado ya que después de usar el canal c_{03}, se alcanza el nodo n_0. De este modo, como todos los nodos destino tienen un índice mayor que n_0, no es posible pedir el canal c_{00}. Obsérvese que los canales c_{00} y c_{13} no están presentes en el grafo ya que nunca se usan.

Es posible extender la función de encaminamiento de anillos unidireccionales para su uso en n-cubos k-arios unidireccionales. Al igual que en el caso anterior, cada canal físico se divide en dos canales virtuales. Además, se añade un nuevo índice a cada canal virtual. Cada canal virtual vendrá etiquetado por c_{dvi}, donde $d, d = \{0, \ldots, n-1\}$ es la dimensión que atraviesa el canal, $v, v = \{0, 1\}$ indica el canal virtual, e $i, i = \{0, \ldots, k-1\}$ indica la posición dentro del anillo correspondiente. La función de encaminamiento envía los paquetes siguiendo un orden ascendente en las dimensiones. Dentro de cada dimensión, se utiliza la función de encaminamiento para los anillos. Es fácil comprobar que esta función encamina los paquetes en orden estrictamente decreciente de los índices de los canales. La figura 7.38 muestra el algoritmo de encaminamiento por dimensiones para los 2-cubos k-arios (toros bidimensionales).

7.3.5 Algoritmos parcialmente adaptativos

En esta sección nos centraremos en el estudio de algoritmos que incrementan el número de caminos alternativos entre dos nodos dados. Dependiendo de si podemos utilizar cualquier camino mínimo entre un nodo origen y un nodo destino, o únicamente algunos de ellos, los algoritmos pueden clasificarse en totalmente adaptativos o parcialmente adaptativos, respectivamente.

Los algoritmos parcialmente adaptativos representan un compromiso entre la flexibilidad y el coste. Intentan aproximarse a la flexibilidad del encaminamiento totalmente adaptativo a expensas de un moderado incremento en la complejidad con respecto al encaminamiento determinista. La mayoría de los algoritmos parcialmente adaptativos propuestos se basan en la ausencia de dependencias cíclicas entre canales para evitar los bloqueos. Algunas propuestas intentan maximizar la adaptabilidad sin incrementar los recursos necesarios para evitar los bloqueos. Otras propuestas intentan minimizar los recursos necesarios para obtener un cierto nivel de adaptabilidad.

Los algoritmos totalmente adaptativos permiten la utilización de cualquier camino mínimo entre el nodo origen y el nodo destino, maximizando el rendimiento de la red (*throughput*). La mayoría de los algoritmos propuestos se basan en el teorema

Algorithm: Dimension-Order Routing for Unidirectional 2-D Tori
Inputs: Coordinates of current node $(Xcurrent, Ycurrent)$
and destination node $(Xdest, Ydest)$
Output: Selected output $Channel$
Procedure:
$Xoffset := Xdest - Xcurrent$;
$Yoffset := Ydest - Ycurrent$;
if $Xoffset < 0$ **then**
$Channel := c_{00}$;
endif
if $Xoffset > 0$ **then**
$Channel := c_{01}$;
endif
if $Xoffset = 0$ **and** $Yoffset < 0$ **then**
$Channel := c_{10}$;
endif
if $Xoffset = 0$ **and** $Yoffset > 0$ **then**
$Channel := c_{11}$;
endif
if $Xoffset = 0$ **and** $Yoffset = 0$ **then**
$Channel := Internal$;
endif

Figura 7.38: El algoritmo de encaminamiento por dimensiones para toros 2-D unidireccionales.

presentado por Dally y Seitz, requiriendo la ausencia de dependencias cíclicas entre canales, lo que da lugar a un gran número de canales virtuales. Utilizando la técnica de canales de escape propuesta por Duato, es posible conseguir un encaminamiento totalmente adaptativo minimizando el número de recursos.

En esta sección nos centraremos en el estudio de dos algoritmos de encaminamiento, uno parcialmente adaptativo, basado en restringir la adaptabilidad en dos dimensiones, y otro totalmente adaptativo, basado en la idea de caminos de escape que se esbozó en el apartado 7.3.3.

Encaminamiento adaptativo por planos

El objetivo del encaminamiento adaptativo por planos es minimizar los recursos necesarios para conseguir un cierto nivel de adaptatividad. Fue propuesto por Chien y Kim para mallas n-dimensionales e hipercubos. La idea del encaminamiento por planos es proporcionar adaptabilidad en dos dimensiones en un momento dado. Así, un paquete se encamina adaptativamente en una serie de planos 2-D.

La figura 7.39 muestra como funciona el encaminamiento adaptativos por planos. Un encaminamiento totalmente adaptativo permite que un paquete pueda encaminarse en el subcubo m-dimensional definido por el nodo actual y destino, como muestra la figura 7.39(a) para tres dimensiones. El encaminamiento por planos restringe el encaminamiento a utilizar en primer lugar el plano A_0, después moverse al plano A_1, y así sucesivamente, tal como muestran las figuras 7.39(b) y 7.39(c) para tres y cuatro

dimensiones, respectivamente. Dentro de cada plano están permitidos todos los caminos. El número de caminos en un plano dependerá de la distancia en las dimensiones correspondientes.

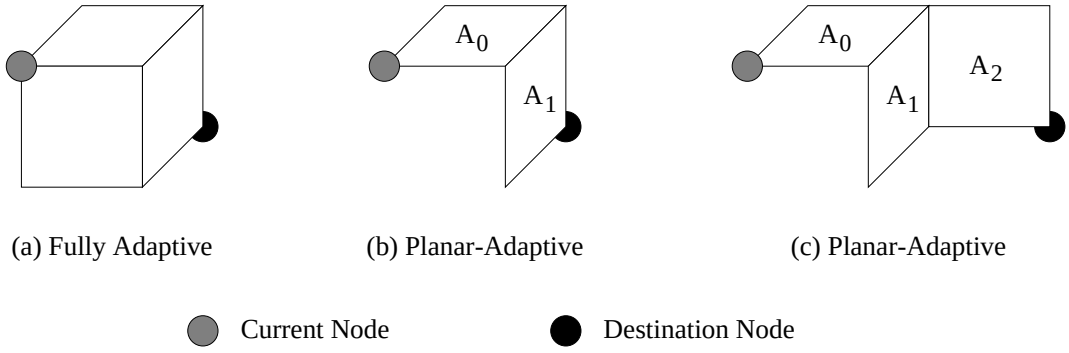


Figura 7.39: Caminos permitidos en el encaminamiento totalmente adaptativo y adaptativo por planos.

Cada plano A_i está formado por dos dimensiones, d_i y d_{i+1}. Existen un total de $(n-1)$ planos adaptativos. El orden de las dimensiones es arbitrario. Sin embargo, es importante observar que los planos A_i y A_{i+1} comparten la dimensión d_{i+1}. Si la diferencia en la dimensión d_i se reduce a cero, entonces el paquete puede pasar al plano A_{i+1}. Si la diferencia en la dimensión d_{i+1} se reduce a cero mientras que el paquete se encuentra en el plano A_i, no existirán caminos alternativos al encaminar el paquete a través del plano A_{i+1}. En este caso, el plano A_{i+1} puede ser omitido. Además, si en el plano A_i, la diferencia en la dimensión d_{i+1} se reduce a cero en primer lugar, el encaminamiento continuará exclusivamente en la dimensión d_i hasta que la diferencia en esta dimensión se reduzca a cero. Por lo tanto, con el fin de ofrecer tantas alternativas de encaminamiento como sea posible, se dará una mayor prioridad a los canales de la dimensión d_i cuando estemos atravesando el plano A_i.

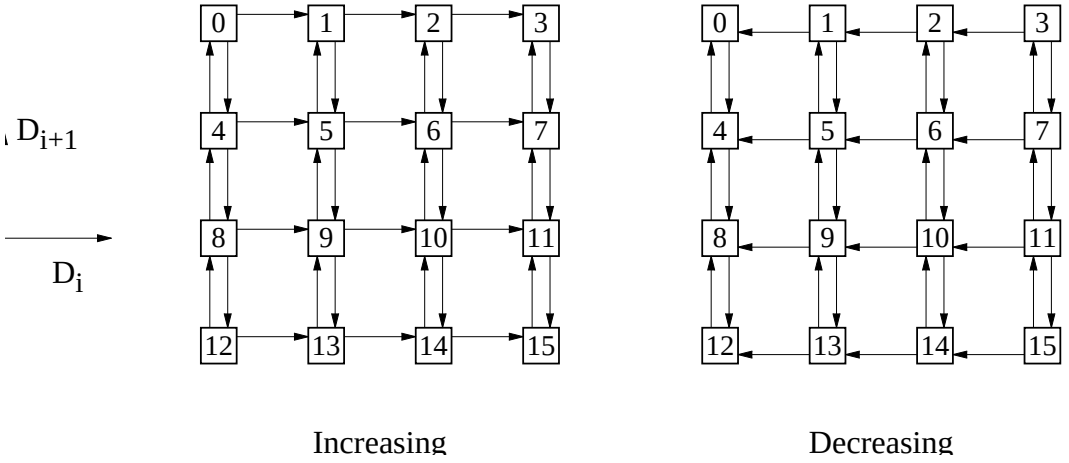


Figura 7.40: Redes crecientes y decrecientes en el plano A_i para el encaminamiento adaptativo por planos.

Tal como hemos definido el encaminamiento por planos, son necesarios tres canales virtuales por canal físico para evitar bloqueos en mallas y seis canales virtuales para

evitar bloqueos en toros. A continuación analizaremos las mallas más detenidamente. Los canales en la primera y última dimensión necesitan únicamente uno y dos canales virtuales, respectivamente. Sea $d_{i,j}$ el conjunto de j canales virtuales que cruzan la dimensión i de la red. Este conjunto puede descomponerse en dos subconjuntos, uno en la dirección positiva y otro en la dirección negativa. Sean $d_{i,j}+$ y $d_{i,j}-$ dichos conjuntos.

Cada plano A_i se define como la combinación de varios conjuntos de canales virtuales:

$$A_i = d_{i,2} + d_{i+1,0} + d_{i+1,1}$$

Con el fin de evitar los bloqueos, el conjunto de canales virtuales en A_i se divide en dos clases: redes *crecientes* y *decrecientes*. Las redes crecientes están formadas por los canales $d_{i,2}+$ y $d_{i+1,0}$. La red decreciente está formada por $d_{i,2}-$ y $d_{i+1,1}$ (Figura 7.40). Los paquetes que cruzan la dimensión d_i en dirección positiva utilizan la red creciente de A_i. Al no existir relación entre la redes crecientes y decrecientes de A_i, y al cruzarse los planos secuencialmente, es fácil de comprobar que no existen dependencias cíclicas entre los canales. Por lo tanto, el algoritmo de encaminamiento adaptativo por planos es libre de bloqueos.

Modelo de giro

7.3.6 Algoritmos completamente adaptativos

Salto negativo

Redes virtuales

Redes deterministas y adaptativas

Algoritmo de Duato

Es posible implementar un algoritmo totalmente adaptativo para n-cubos k-arios utilizando únicamente tres canales virtuales por canal físico. Basándonos en la idea de utilizar un algoritmo libre de bloqueo que actúe como vías de escape y añadiendo canales adicionales a cada dimensión que puedan utilizarse sin restricciones, y que nos proporcionan la adaptabilidad, es posible obtener un algoritmo totalmente adaptativo.

Como algoritmo de base usaremos una extensión del algoritmo adaptativo que vimos para anillos unidireccionales. La extensión para anillos bidireccionales es directa. Simplemente, se usa un algoritmo de encaminamiento similar para ambas direcciones de cada anillo. Dado que los paquetes utilizan únicamente caminos mínimos, no existen dependencias entre los canales de una dirección y los canales de la dirección opuesta. Por lo tanto, el algoritmo de encaminamiento para anillos bidireccionales está libre de bloqueos. La extensión a n-cubos k-arios bidireccionales se consigue utilizando el encaminamiento por dimensiones. Para conseguir un encaminamiento mínimo totalmente adaptativo se añade un nuevo canal virtual a cada canal físico (uno en cada dirección). Estos canales virtuales pueden recorrerse en cualquier sentido, permitiendo cualquier camino mínimo entre dos nodos. Se puede demostrar que el algoritmo resultante es libre de bloqueos.

7.3.7 Comparación de los algoritmos de encaminamiento

En esta sección analizaremos el rendimiento de los algoritmos de encaminamiento deterministas y adaptativos sobre varias topologías y bajo diferentes condiciones de tráfico. Dado el gran número de topologías y algoritmos de encaminamiento existentes, una evaluación exhaustiva sería imposible. En lugar de esto, nos centraremos en unas cuantas topologías y algoritmos de encaminamiento, mostrando la metodología usada para obtener unos resultados que nos permitan una evaluación preliminar. Estos resultados se pueden obtener simulando el comportamiento de la red bajo una carga sintética. Una evaluación detallada requiere usar trazas que sean representativas de las aplicaciones bajo estudio.

La mayoría de los multicomputadores y multiprocesadores actuales usan redes de baja dimensión (2-D ó 3-D) o toros. Por lo tanto, usaremos mallas 2-D y 3-D así como toros para evaluar los distintos algoritmos de encaminamiento. Además, la mayoría de estas máquinas utilizan encaminamiento por dimensiones, aunque el encaminamiento totalmente adaptativo ha empezado a ser introducido tanto en máquinas experimentales como comerciales. Por tanto, analizaremos el comportamiento de los algoritmos de encaminamiento por dimensiones y totalmente adaptativos (estos últimos requieren dos conjuntos de canales virtuales: un conjunto para el encaminamiento determinista y el otro para el encaminamiento totalmente adaptativo).

A continuación realizaremos una breve descripción de los algoritmos de encaminamiento que utilizaremos. El algoritmo determinista para mallas cruza las dimensiones en orden creciente. En principio, no requiere canales virtuales, aunque pueden utilizarse para aumentar el rendimiento. En este último caso, se seleccionará el primer canal virtual libre. El algoritmo totalmente adaptativo para mallas consiste en dos canales virtuales, uno que se utiliza como vía de escape y que permite el encaminamiento siguiendo el algoritmo X-Y y el otro encargado de hacer posibles todos los caminos mínimos entre el nodo actual y el nodo destino. Cuando existen varios canales de salida libres, se da preferencia al canal totalmente adaptativo en la menor dimensión útil, seguido por los canales adaptativos según el orden creciente de dimensiones útiles. Si se usan más de dos canales virtuales, el resto de canales virtuales permiten un encaminamiento totalmente adaptativo. En este caso, los canales virtuales se seleccionan de tal manera que se minimice la multiplexación de canales. El algoritmo determinista para toros requiere dos canales virtuales por canal físico, como vimos en la sección anterior. Cuando se utilizan más de dos canales virtuales, cada par de canales adicionales tienen la misma funcionalidad de encaminamiento que el primer par. También evaluaremos un algoritmo parcialmente adaptativo para toros, basado en la extensión del algoritmo parcialmente adaptativo que vimos para anillos unidireccionales. El algoritmo extendido usa canales bidireccionales siguiendo caminos mínimos. Además, las dimensiones se cruzan en orden ascendente. El algoritmo totalmente adaptativo para toros requiere un tercer canal virtual, el resto de canales se usan como en el caso del algoritmo parcialmente adaptativo. De nuevo, en el caso de existir varios canales de salida libres, se dará preferencia a los canales totalmente adaptativos en la menor dimensión útil, seguido de los canales adaptativos según el orden de dimensiones útiles.

A no ser que se diga lo contrario, los parámetros de simulación son los siguientes: 1 ciclo de reloj para calcular el algoritmo de encaminamiento, para transferir un flit de un buffer de entrada a un buffer de salida, o para transferir un flit a través de un canal físico. Los buffers de entrada y salida tienen una capacidad variable de tal manera que la capacidad de almacenamiento por canal físico se mantiene constante. Cada nodo

tiene cuatro canales de inyección y recepción de paquetes. Además, la longitud de los mensajes se mantiene constante a 16 flits (más 1 flit de cabecera). También se supondrá que el destino de los mensajes sigue una distribución uniforme.

Encaminamiento determinista vs. adaptativo

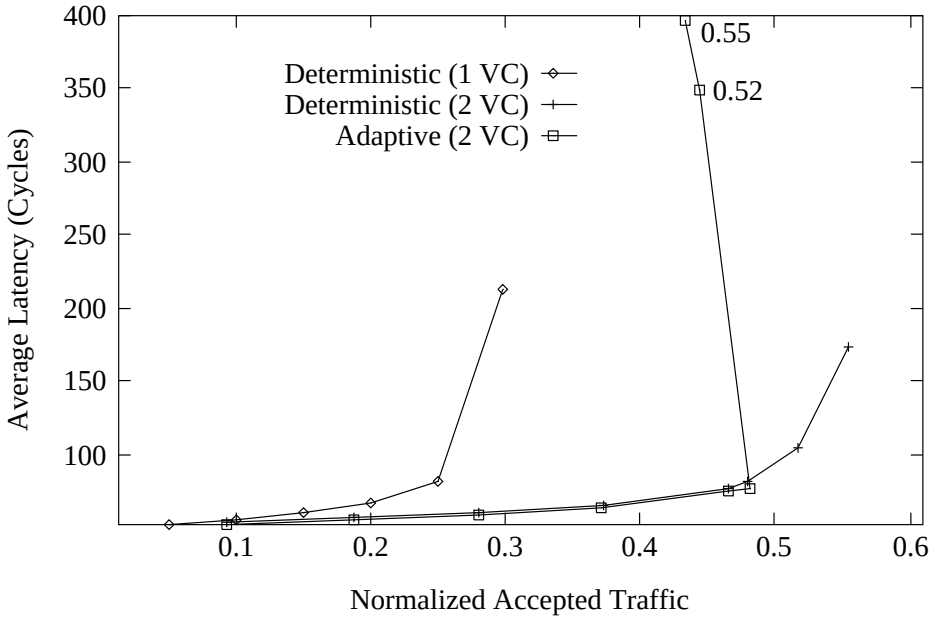


Figura 7.41: Latencia media del mensaje vs. tráfico normalizado aceptado para mallas 16×16 para una distribución uniforme del destino de los mensajes.

La figura 7.41 muestra la latencia media de un mensaje en función del tráfico normalizado aceptado en una malla 2-D. La gráfica muestra el rendimiento del encaminamiento determinista con uno y dos canales virtuales, y totalmente adaptativo (con dos canales virtuales). Como se puede apreciar, el uso de dos canales virtuales casi dobla el rendimiento del algoritmo determinista. La principal razón es que cuando se bloquean los mensajes, el ancho de banda del canal no se malgasta ya que otros mensajes pueden utilizarlo. Por lo tanto, añadiendo unos pocos canales virtuales se consigue reducir la contención e incrementar la utilización del canal. El algoritmo adaptativo consigue el 88% del rendimiento conseguido por el algoritmo determinista con el mismo número de canales. Sin embargo, la latencia es casi idéntica, siendo ligeramente inferior para el algoritmo adaptativo. Así, la flexibilidad adicional del encaminamiento totalmente adaptativo no consigue mejorar el rendimiento cuando el tráfico presenta una distribución uniforme. La razón es que la red está cargada de forma uniforme. Además, las mallas no son regulares, y los algoritmos adaptativos tienden a concentrar tráfico en la parte central de la bisección de la red, reduciendo la utilización de los canales existentes en el borde de la malla.

Obsérvese la existencia de una pequeña degradación del rendimiento cuando el algoritmo adaptativo alcanza el punto de saturación. Si la tasa de inyección se mantiene constante en este punto, la latencia se incrementa considerablemente mientras que el tráfico aceptado decrece. Este comportamiento es típico de los algoritmos de encaminamiento que permiten dependencias cíclicas entre canales.

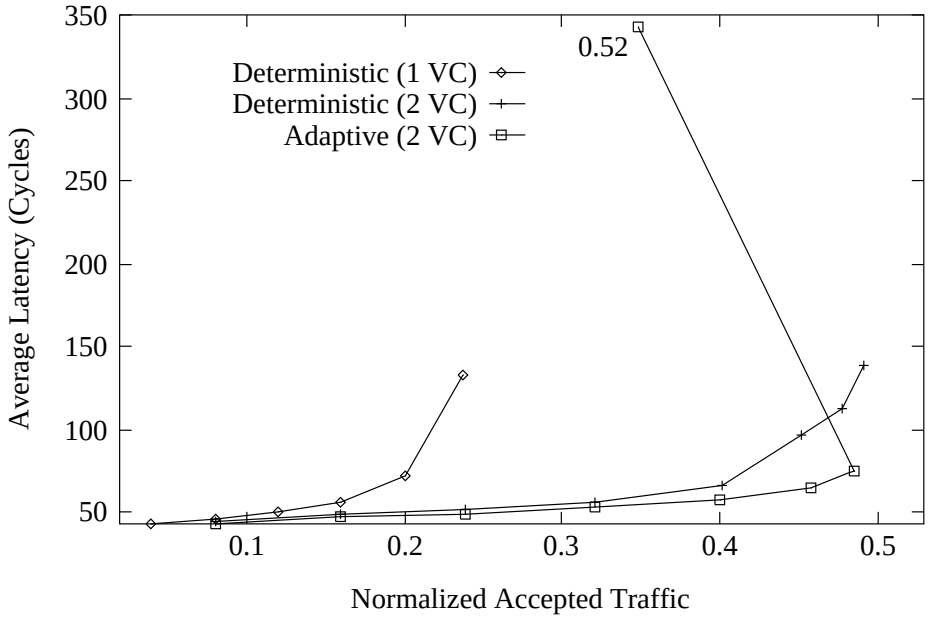


Figura 7.42: Latencia media del mensaje vs. tráfico normalizado aceptado para mallas $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes.

La figura 7.42 muestra la latencia media de los mensajes en función del tráfico normalizado aceptado en una malla 3-D. Esta gráfica es bastante similar a la de las mallas 2-D. Sin embargo, existen algunas diferencias significativas. Las ventajas de usar dos canales virtuales en el algoritmo determinista son más evidentes en el caso de mallas 3-D. En este caso, el rendimiento es el doble. Además, el algoritmo totalmente adaptativo alcanza el mismo rendimiento que el algoritmo determinista con el mismo número de canales virtuales. La reducción de la latencia conseguida por el algoritmo totalmente adaptativo es más evidente en las mallas 3-D. La razón es que los mensajes tienen un canal adicional para elegir en la mayoría de los nodos intermedios. De nuevo, existe una degradación del rendimiento cuando el algoritmo adaptativo alcanza el punto de saturación. Esta degradación es más pronunciada que en las mallas 2-D.

La figura 7.43 muestra la latencia media de los mensajes en función del tráfico aceptado en un toro 2-D. La gráfica muestra el rendimiento del encaminamiento determinista con dos canales virtuales, parcialmente adaptativo con dos canales virtuales, y totalmente adaptativo con tres canales virtuales. Tanto el algoritmo parcial como totalmente adaptativo incrementan el rendimiento considerablemente en comparación con el algoritmo determinista. El algoritmo parcialmente adaptativo incrementa el rendimiento en un 56%. La razón es que la utilización de los canales no está balanceada en el algoritmo de encaminamiento determinista. Sin embargo, el algoritmo parcialmente adaptativo permite a la mayoría de los mensajes elegir entre dos canales virtuales en lugar de uno, reduciendo la contención e incrementando la utilización de los canales. Obsérvese que esta flexibilidad adicional se consigue sin incrementar el número de canales virtuales. El algoritmo totalmente adaptativo incrementa el rendimiento de la red en un factor de 2.5 en comparación con el algoritmo determinista. Esta mejora considerable se debe principalmente a la posibilidad de cruzar las dimensiones en cualquier orden. Al contrario que en las mallas, los toros son topologías regulares. Así, los algoritmos adaptativos son capaces de mejorar la utilización de los canales distribuyendo el tráfico de forma uniforme a través de la red. Los algoritmos parcial y totalmente adaptativos

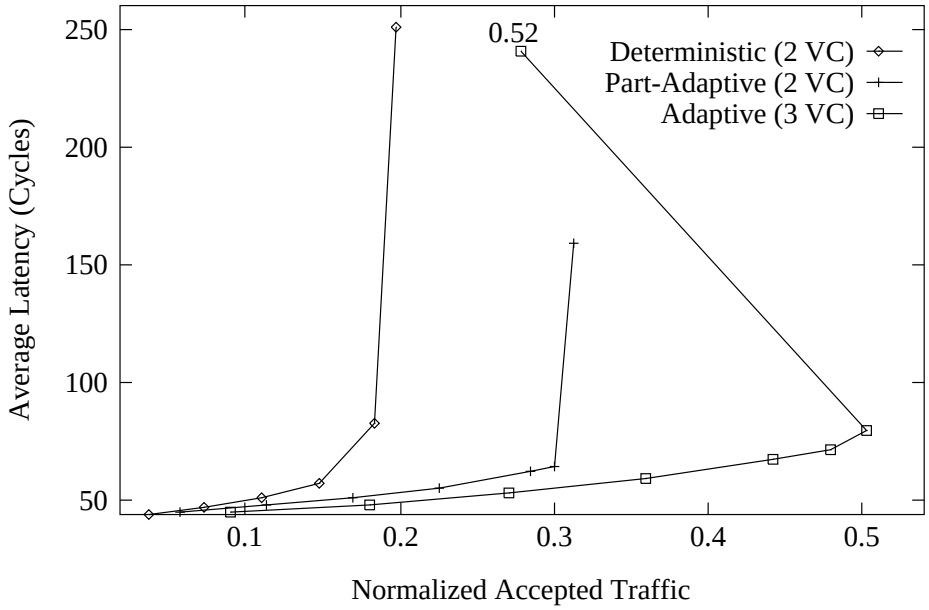


Figura 7.43: Latencia media del mensaje vs. tráfico normalizado aceptado para toros 16×16 para una distribución uniforme del destino de los mensajes.

también consiguen una reducción de la latencia de los mensajes con respecto al algoritmo determinista para todo el rango de carga de la red. De manera similar, el algoritmo totalmente adaptativo reduce la latencia con respecto al parcialmente adaptativo. Sin embargo, la degradación en el rendimiento a partir del punto de saturación reduce el tráfico aceptado por la red al 55% de su valor máximo.

La figura 7.44 muestra la latencia media de los mensajes en función del tráfico aceptado en un toro 3-D. Además de los algoritmos analizados en la figura 7.43, esta gráfica muestra también el rendimiento del algoritmo parcialmente adaptativo con tres canales virtuales. Al igual que en el caso de las mallas, los algoritmos de encaminamiento adaptativos se comportan comparativamente mejor en un toro 3-D que un toro 2-D. En este caso, los algoritmos parcial y totalmente adaptativos incrementan el rendimiento de la red por un factor de 1.7 y 2.6, respectivamente, sobre el rendimiento obtenido con el algoritmo determinista. La reducción en la latencia también es más acusada que en el caso de toros 2-D. La gráfica también muestra que añadiendo un canal virtual al algoritmo parcialmente adaptativo el rendimiento de la red no mejora significativamente. Aunque aumenta el rendimiento en un 18%, también se incrementa la latencia. La razón es que el algoritmo parcialmente adaptativo con dos canales virtuales ya permite el uso de dos canales virtuales en la mayoría de los mensajes, permitiendo la compartición del ancho de banda. Así, añadir otro canal virtual tiene poco impacto en el rendimiento. Este efecto es similar si se consideran otras distribuciones del tráfico en la red. Este resultado también confirma que la mejora conseguida por el algoritmo totalmente adaptativo se debe principalmente a la posibilidad de cruzar las dimensiones en cualquier orden.

La figura 7.45 muestra la desviación estándar de la latencia en función del tráfico aceptado por la red en un toro 2-D. Como se puede observar, un mayor grado de adaptabilidad supone además una reducción de la desviación con respecto al valor medio. La razón es que el encaminamiento adaptativo reduce considerablemente la contención en los nodos intermedios, haciendo que la latencia sea más predecible.

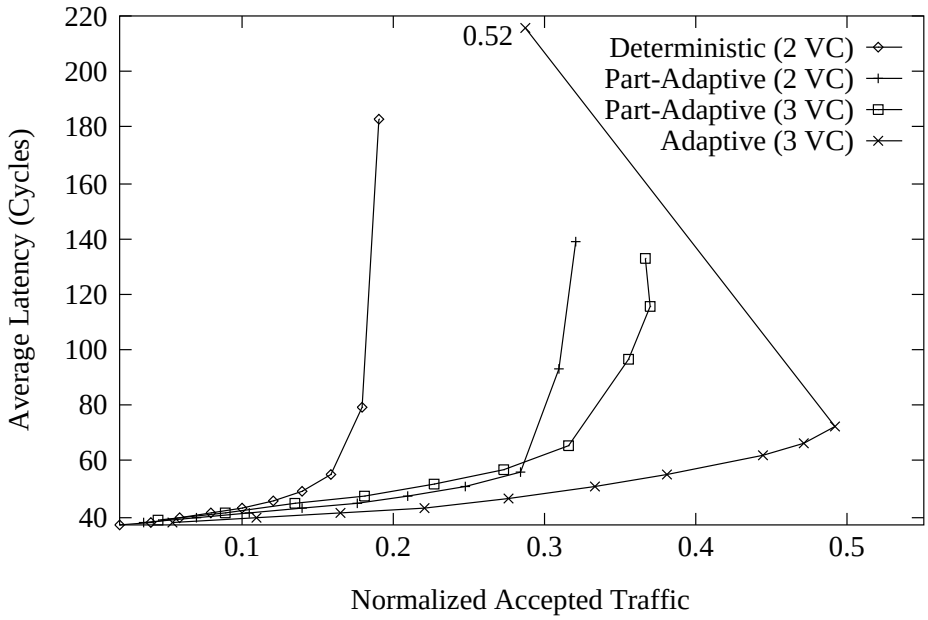


Figura 7.44: Latencia media del mensaje vs. tráfico normalizado aceptado para toros $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes.

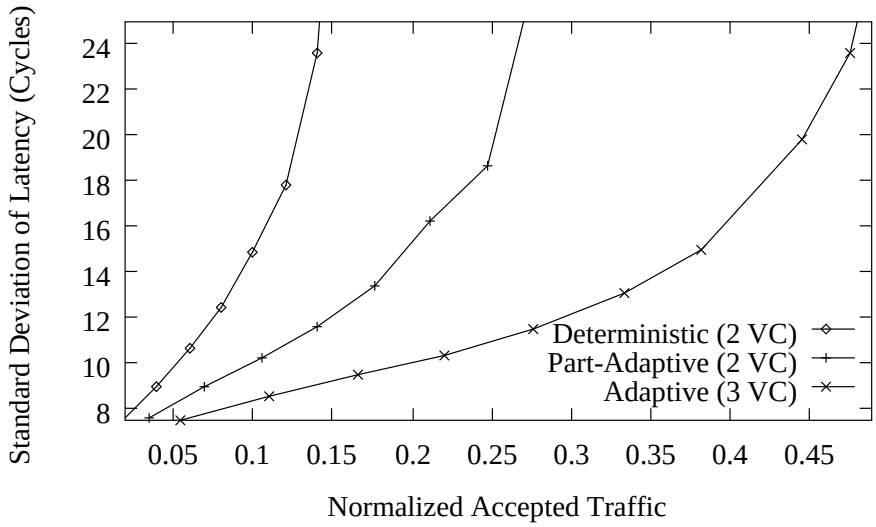


Figura 7.45: Desviación estándar de la latencia vs. tráfico normalizado aceptado en un toro $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes

Capítulo 8

Otras arquitecturas avanzadas

Hasta ahora, la mayoría de sistemas que se han explicado pretendían resolver problemas generales de procesamiento, es decir, se trataba de arquitecturas que de alguna u otra manera permitían resolver de una forma eficiente una amplia gama de problemas de computación. Esto es especialmente interesante puesto que estas arquitecturas son las que se utilizan en los sistemas de propósito general que son los que ocupan la mayor parte del mercado de computadores. Hay diferencias entre unas arquitecturas y otras, por ejemplo es evidente que una máquina vectorial va a ser ideal para el cálculo científico, mientras que un multicomputador es más interesante para la realización de tareas sencillas por muchos usuarios conectados a el, pero en ambos casos el sistema es fácil de programar y resuelve prácticamente el mismo tipo de problemas.

Existen, sin embargo, otras arquitecturas que para un caso particular de problemas funcionan mejor que ninguna otra, pero no resuelven bien el problema de una programación sencilla o el que se pueda utilizar en cualquier problema. Estas arquitecturas son por tanto específicas para determinados problemas y no es fácil encontrarlas en sistemas de propósito general. Sin embargo, son necesarias para su utilización en sistemas *empotrados*, es decir, sistemas que realizan una tarea específica, que requieran una especial capacidad en una determinada tarea que un sistema de propósito general no puede tener. Los procesadores vectoriales y matriciales son ejemplos de arquitecturas que estarían en la frontera entre el propósito general y los sistemas empotrados. Ya como sistemas empotrados, o formando parte como un módulo en un sistema de propósito general, tenemos arquitecturas como los arrays sistólicos, los DSPs, las redes de neuronas, los procesadores difusos, etc.

Como en la mayoría de los casos estas soluciones requieren la realización de un chip a medida, a estas arquitecturas específicas se les llama también *arquitecturas VLSI*. La mejora y abaratamiento del diseño de circuitos integrados ha permitido que se pudieran implementar todas estas arquitecturas específicas en chips simples, lo que ha permitido el desarrollo y uso de estas arquitecturas.

Hemos visto entonces que hay problemas específicos que inspiran determinadas arquitecturas como acabamos de ver. Por otro lado, existen diferentes formas de enfocar la computación incluso de problemas generales, pero que por su todavía no probada eficiencia, o porque simplemente no han conseguido subirse al carro de los sistemas comerciales, no se utilizan en la práctica. Una arquitectura que estaría dentro de este ámbito sería la arquitectura de flujo de datos, donde el control del flujo del programa la realizan los datos y no las instrucciones. Hay otras arquitecturas como las basadas

en *transputers* que resultan también interesantes y de hecho se utilizan comercialmente, aunque en realidad no suponen una nueva filosofía de arquitectura sino más bien una realización práctica de conceptos conocidos.

En este capítulo veremos estas arquitecturas cuyo conocimiento va a resultar interesante para poder resolver problemas específicos que se pueden presentar y que quizá una arquitectura de propósito general no resuelve de forma eficaz. Se empezará por la máquina de flujo de datos y se seguirá con las arquitecturas VLSI.

8.1 Máquinas de flujo de datos

Hay dos formas de procesar la información, una es mediante la ejecución en serie de una lista de instrucciones y la otra es la ejecución de las instrucciones según las pidan los datos disponibles. La primera forma empezó con la arquitectura de Von Neumann donde un programa almacenaba las órdenes a ejecutar, sucesivas modificaciones, etc., han convertido esta sencilla arquitectura en los multiprocesadores para permitir paralelismo.

La segunda forma de ver el procesamiento de datos quizá es algo menos directa, pero desde el punto de vista de la paralelización resulta mucho más interesante puesto que las instrucciones se ejecutan en el momento tienen los datos necesarios para ello, y naturalmente se deberían poder ejecutar todas las instrucciones demandadas en un mismo tiempo. Hay algunos lenguajes que se adaptan a este tipo de arquitectura comandada por datos como son el Prolog, el ADA, etc. es decir, lenguajes que explotan de una u otra manera la concurrencia de instrucciones.

En una arquitectura de flujo de datos un instrucción está lista para ejecución cuando los datos que necesita están disponibles. La disponibilidad de los datos se consigue por la canalización de los resultados de las instrucciones ejecutadas con anterioridad a los operandos de las instrucciones que esperan. Esta canalización forma un flujo de datos que van disparando las instrucciones a ejecutar. Por esto se evita la ejecución de instrucciones basada en *contador de programa* que es la base de la arquitectura Von Neumann.

Las instrucciones en un flujo de datos son puramente autocontenidas; es decir, no utilizan variables en una memoria compartida global, sino que llevan los valores de las variables en ellas mismas. En una maquina de este tipo, la ejecución de una instrucción no afecta a otras que estén listas para su ejecución. De esta manera, varias instrucciones pueden ser ejecutadas simultáneamente lo que lleva a la posibilidad de un alto grado de concurrencia y paralelización.

8.1.1 Grafo de flujo de datos

Para mostrar el comportamiento de las máquinas de flujo de datos se utiliza un grafo llamado *grafo de flujo de datos*. Este grafo de flujo de datos muestra las dependencias de datos entre las instrucciones. El grafo representa los pasos de ejecución de un programa y sirve como interfaz entre la arquitectura del sistema y el lenguaje de programación.

Los nodos en el grafo de flujo de datos, también llamados *actores*, representan los operadores y están interconectados mediante arcos de entrada y salida que llevan etiquetas conteniendo algún valor. Estas etiquetas se ponen y quitan de los arcos de acuerdo con unas reglas de disparo. Cada actor necesita que unos determinados arcos

de entrada tengan etiquetas para que pueda ser disparado (o ejecutado). Cuando están presentes las etiquetas en los arcos necesarios, el actor se habilita y entonces se dispara. Al dispararse, quita una etiqueta de cada uno de los arcos de entrada requeridos para el disparo, realiza la función requerida según las etiquetas que hubiera, y pone las etiquetas de resultado en los arcos de salida correspondientes.

Cada nodo de un grafo de flujo de datos puede representarse (o guardarse) como una *copia de actividad*. Una copia de actividad consiste en unos campos para el tipo de operación, el almacenamiento de las etiquetas de entrada, y para las direcciones de destino. Como en un grafo de flujo de datos, una colección de copias de actividad se puede utilizar para representar un programa.

Hay dos tipos de etiquetas: etiquetas de datos y etiquetas booleanas. Para distinguir el tipo de entradas a un actor se utilizan flechas simples para llevar etiquetas de datos y flechas dobles para las etiquetas booleanas. El *actor de conmutación* pone la etiqueta de entrada en uno de los arcos de salida dependiendo de la etiqueta que llega por la entrada booleana. El *actor de combinación* pone una de las etiquetas de entrada en el arco de salida dependiendo de la etiqueta booleana. La *puerta T* pasa la etiqueta de la entrada a la salida cuando la etiqueta booleana es *True*, y la *puerta F* cuando es *False*.

La figura 8.1 muestra el grafo de flujo de datos para el cálculo de $N!$ cuyo código secuencial se da a continuación. Hay que hacer notar que los arcos booleanos de entrada de los dos actores que combinan están inicializados con etiquetas *False*. Esto causa que en el inicio del programa, la etiqueta de datos N vaya a la salida de los dos combinadores.

```
Input N;
  i=N;
  While i>1
  {   N=N(i-1);
      i=i-1;
  }
Output N;
```

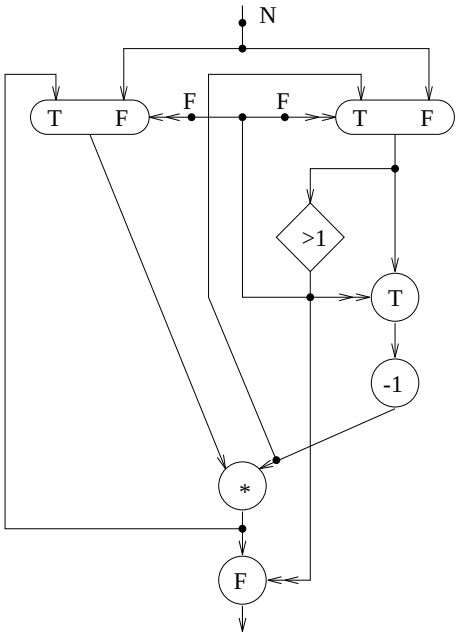


Figura 8.1: Grafo de flujo de datos para calcular $N!$

8.1.2 Estructura básica de un computador de flujo de datos

Se han realizado algunas máquinas para probar el funcionamiento de la arquitectura de flujo de datos. Una de ellas fue desarrollada en el MIT y su estructura se muestra en la figura 8.2. El computador MIT está formado por cinco unidades:

Unidad de proceso, formada por un conjunto de elementos de proceso especializados.

Unidad de memoria, formada por células de instrucción para guardar la instrucción y sus operandos, es decir, cada célula guarda una copia de actividad.

Red de arbitraje, que envía instrucciones a la unidad de procesamiento para su ejecución.

Red de distribución, que transfiere los resultados de las operaciones a la memoria.

Unidad de control, que administra todas las unidades.

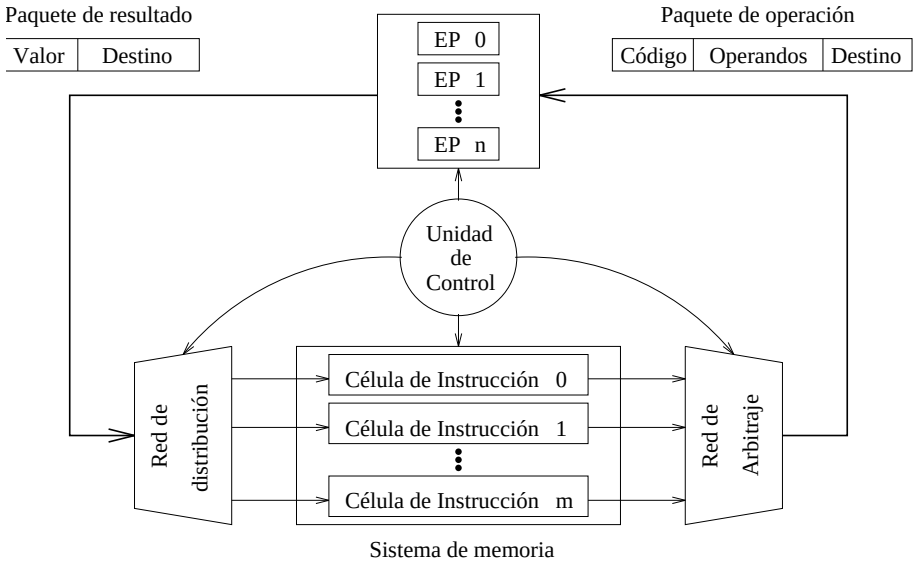


Figura 8.2: La máquina MIT de flujo de datos.

Cada célula de instrucción guarda una instrucción que consiste en un código de operación, unos operandos, y una dirección de destino. La instrucción se activa cuando se reciben todos los operandos y señales de control requeridas. La red de arbitraje manda la instrucción activa como un paquete de operación a unos de los elementos de proceso. Una vez que la instrucción es ejecutada, el resultado se devuelve a través de la red de distribución al destino en memoria. Cada resultado se envía como un paquete que consiste en un resultado mas una dirección de destino.

Las máquinas de flujo de datos se pueden clasificar en dos grupos:

Máquinas estáticas: En una máquina de flujo de datos estática, una instrucción se activa siempre que se reciban todos los operandos requeridos y haya alguna instrucción esperando recibir el resultado de esta instrucción; si no es así, la instrucción permanece inactiva. Es decir, cada arco en el grafo del flujo de datos puede tener una etiqueta como mucho en cualquier instante. Un ejemplo de este tipo de flujo de datos se muestra en la figura 8.3. La instrucción de multiplicación no debe ser activada hasta que su resultado previo ha sido utilizado por la suma.

A menudo, esta restricción se controla con el uso de señales de reconocimiento.

Máquinas dinámicas: En una máquina de flujo de datos dinámica, una instrucción se activa cuando se reciben todos los operandos requeridos. En este caso, varios conjuntos de operandos pueden estar listos para una instrucción al mismo tiempo. En otras palabras, un arco puede contener más de una etiqueta.

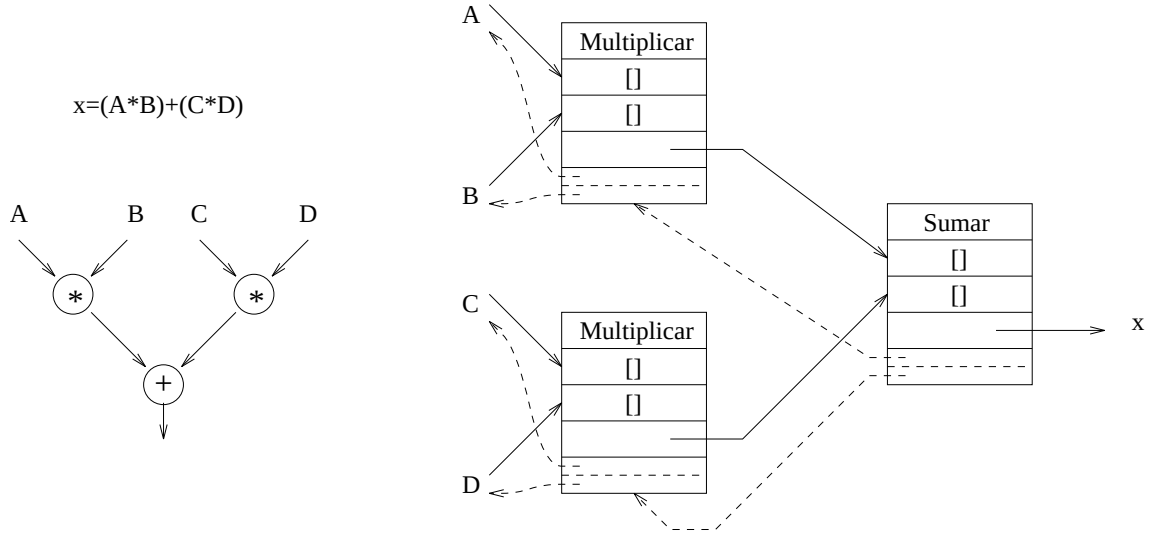


Figura 8.3: Ejemplo de máquina de flujo de datos estática (las flechas discontinuas son las señales de reconocimiento).

Comparado con el flujo de datos estático, el dinámico permite mayor paralelismo ya que una instrucción no necesita esperar a que se produzca un espacio libre en otra instrucción antes de guardar su resultado. Sin embargo, en el tipo dinámico es necesario establecer un mecanismo que permita distinguir los valores de diferentes conjuntos de operandos para una instrucción.

Uno de estos mecanismos consiste en formar una cola con los valores de cada operando en orden de llegada. Sin embargo, el mantenimiento de muchas largas colas resulta muy costoso. Para evitar las colas, el formato del paquete de resultado a menudo incluye un campo de etiqueta; con esto, los operandos que coinciden se pueden localizar comparando sus etiquetas. Una memoria asociativa, llamada *memoria de coincidencias*, puede ser usada para buscar las coincidencias entre operandos. Cada vez que un paquete de resultados llega a la unidad de memoria, sus campos de dirección y de etiqueta son guardados en la memoria de coincidencias. Tanto la dirección como la etiqueta serán utilizadas como clave para determinar qué instrucción está activa.

A pesar de que las máquinas convencionales, basadas en el modelo Von Neumann, tienen muchas desventajas, la industria todavía sigue fabricando de forma mayoritaria este tipo de computadores basados en flujo de control. Esta elección está basada en la efectividad/coste, lanzamiento al mercado, etc. Aunque las arquitecturas de flujo de datos tienen un mayor potencial de paralelización, todavía se encuentran en su etapa de investigación. Las máquinas de flujo de control todavía dominan el mercado.

8.2 Otras arquitecturas

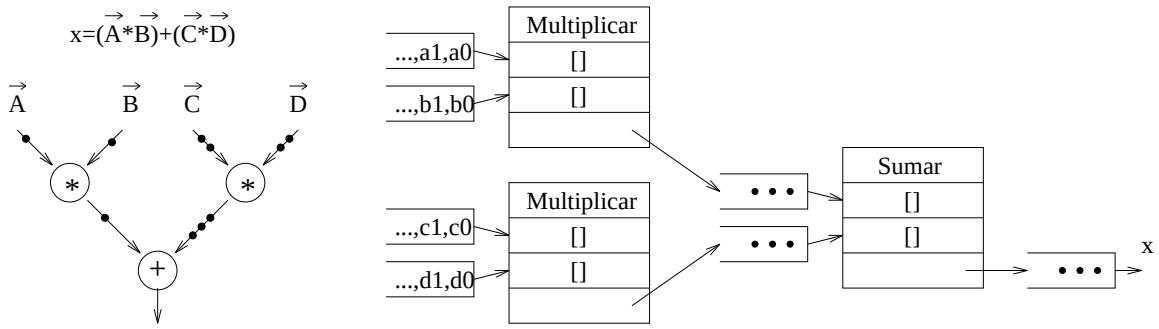


Figura 8.4: Ejemplo de máquina de flujo de datos dinámica. Un arco en grafo de flujo de datos puede llevar más de una etiqueta a un tiempo.

Apéndice A

Comentarios sobre la bibliografía

La siguiente lista de libros es una lista priorizada atendiendo a la idoneidad de los contenidos para la asignatura de å, no se está por tanto diciendo que un libro sea mejor que otro, sino que unos cubren mejor que otros determinados contenidos de la asignatura. No hay un libro que se pueda decir que cubre todos los aspectos de la asignatura o que es el libro de texto básico, pero prácticamente los 3 o 4 primeros son suficientes para preparar la mayoría de temas que se explican durante la asignatura.

[CSG99] *Parallel Computer Architecture: A Hardware/Software approach.* Uno de los libros sobre arquitecturas paralelas más completos y actuales. Presenta una visión bastante general de las arquitecturas paralelas pero eso no impide entrar con gran detalle en todos los temas que trata. Aunque el libro es sobre arquitecturas paralelas en general se centra sobre todo en los multiprocesadores y multicomputadores. La estructura de este libro se asemeja bastante a la adoptada para el temario.

[Hwa93] *Advanced computer architecture: Parallelism, scalability, programmability.* De este libro se han extraído muchos de los contenidos de la asignatura y aunque trata de casi todos los temas hay algunos que están mejor descritos en otros textos. Es el libro sobre arquitectura, escrito por el mismo autor, que viene a sustituir al [HB87] que en algunos temas ya se estaba quedando algo anticuado. Comparado con otros no sigue exactamente la misma estructura y a veces los temas están repartidos en varios capítulos, pero todo el contenido es bastante interesante y actual. Vienen muchos ejemplos de máquinas comerciales y de investigación.

[DYN97] *Interconnection Networks; An Engineering Approach.* Toda la parte de redes del temario del curso de å se ha extraído de este libro. Probablemente es uno de los libros sobre redes de interconexión para multicomputadores más completos que existen actualmente.

[HP96] *Computer Architecture, a Quantitative Approach.* Esta última edición de Hennessy y Patterson es una mejora sobre la versión editada en español, ya que incluye muchos aspectos nuevos del procesamiento paralelo, multiprocesadores y multicomputadores. Buena parte del libro está dedicada a la segmentación de instrucciones y procesadores RISC, pero hay capítulos, como el de los vectoriales, que han sido utilizados íntegramente para el temario del curso. Sirve de apoyo para casi el resto de temas del curso entre los que destacan la clasificación de los computadores, las redes, memoria entrelazada, caché y consistencia de la memoria. Es un libro de lectura amena con ejemplos y gráficos cuantitativos.

[Sto93] *High-Performance Computer Architecture.* Es un libro con algunos aspectos interesantes de las arquitecturas avanzadas. Presenta unos modelos útiles para la extracción de conclusiones sobre los sistemas paralelos. Estos modelos son los utilizados en el tema dedicado al estudio del rendimiento en el curso. Sirve de complemento a otros temas siendo interesante el capítulo dedicado a los procesadores vectoriales.

[HB87] *Arquitectura de Computadoras y Procesamiento Paralelo.* Libro clásico de arquitectura de computadores, el problema es que parte de sus contenidos se han quedado algo obsoletos. En cualquier caso sigue siendo una referencia válida para muchos temas y para tener una referencia en español sobre segmentación, vectoriales, matriciales y máquinas de flujo.

[HP93] *Arquitectura de Computadoras, un Enfoque Cuantitativo.* Se trata de una versión en español del conocido Hennessy y Patterson, pero una edición anterior que el comentado anteriormente en inglés. Esto significa que hay ciertas carencias en cuanto a multiprocesadores. En cambio el tema de vectoriales está bien tratado, y es casi la única referencia en español para este curso.

[Zar96] *Computer Architecture, single and parallel systems.* Aunque no es un libro muy extenso trata muy bien los temas que incluye. Para la asignatura destaca el tema sobre segmentación, la clasificación de los sistemas paralelos, la caché, y es de los pocos libros de arquitectura en general que trata con cierta extensión las arquitecturas específicas como las máquinas de flujo de datos, las matrices sistólicas, las redes neuronales, los sistemas difusos, etc.

[Tan95] *Distributed Operating Systems.* Este libro trata sobre todo los sistemas operativos, pero el capítulo dedicado a la memoria compartida distribuida es muy útil, especialmente para el tema de modelos de consistencia de memoria.

[CDK96] *Distributed systems: concepts and design.* Es una referencia adicional al [Tan95] sobre el tema de modelos de consistencia de memoria. El resto del libro trata de los sistemas distribuidos, pero no tanto desde el punto de vista de la arquitectura.

[Fly95] *Computer architecture: pipelined and parallel processor design.* Libro muy completo que enfoca los mismos temas desde una óptica diferente. Resulta interesante como segunda lectura, no porque en el resto de libros estén mejor, sino porque el nivel es algo más elevado. Destaca el tema de coherencia de cachés y para ampliar un poco el tema de redes.

[Wil96] *Computer Architecture, design and performance.* Otro libro con contenidos interesantes. Destacan sobre todo la parte de segmentación, la de redes, y la de flujo de datos.

[Kai96] *Advanced Computer Architecture: a systems design approach.* Este libro, a pesar de su nombre, poco tiene que ver con los contenidos de los libros clásicos de arquitectura de computadores. No obstante, los temas dedicados a las matrices sistólicas y a las máquinas de flujo de datos son interesantes.

[Sta96] *Organización y Arquitectura de Computadores, diseño para optimizar prestaciones.* No es realmente un libro de arquitecturas avanzadas, pero como trata algunos temas a bajo nivel, puede ser interesante para completar algunos aspectos. Destaca la descripción que hace de algunos buses y en especial del Futurebus+.

[Sta93] *Computer organization and architecture: principles of structure and function.* Versión en inglés de su homólogo en castellano.

Bibliografía

[CDK96] George Coulouris, Jean Dollimore, y Tim Kindberg. *Distributed systems: concepts and design*. Addison-Wesley, 1996. BIBLIOTECA: CI 681.3 COU (2 copias), CI-Informática (1 copia).

[CSG99] David Culler, Jaswinder Pal Singh, y Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software approach*. Morgan Kaufmann, 1999.

[DYN97] José Duato, Sudhakar Yalamanchili, y Lionel Ni. *Interconnection Networks; An Engineering Approach*. IEEE Computer Society, 1997. BIBLIOTECA: CI 681.3 DUA (2 copias), CI-Informática (1 copia).

[Fly95] Michael J. Flynn. *Computer architecture: pipelined and parallel processor design*. Jones and Bartlett, 1995. BIBLIOTECA: CI 681.3.06 FLY (1 copia), CI-Informática (1 copia).

[HB87] Kai Hwang y Fayé A. Briggs. *Arquitectura de Computadoras y Procesamiento Paralelo*. McGraw-Hill, 1987. BIBLIOTECA: CI 681.3 HWA (2 copias).

[HP93] John L. Hennessy y David A. Patterson. *Arquitectura de Computadoras, un Enfoque Cuantitativo*. Morgan Kaufmann, segunda edición, 1993. BIBLIOTECA: CI 681.3 HEN (1 copia), CI-Informática (2 copias), Aulas Informáticas (2 copias), fice FE.L/03728.

[HP96] John L. Hennessy y David A. Patterson. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann, primera edición, 1996. BIBLIOTECA: CI 681.3 HEN (2 copias), CI-Informática (1 copia).

[Hwa93] Kai Hwang. *Advanced computer architecture: Parallelism, scalability, programmability*. McGraw-Hill, 1993. BIBLIOTECA: CI 681.3 HWA (3 copias), CI-IFIC (1 copia), CI-Informática (1 copia).

[Kai96] Richard Y. Kain. *Advanced Computer Architecture: a systems design approach*. Prentice-Hall, 1996. BIBLIOTECA: CI 681.3 KAI (1 copia), CI-Informática (1 copia).

[Sta93] William Stallings. *Computer organization and architecture: principles of structure and function*. Prentice Hall, tercera edición, 1993. BIBLIOTECA: CI 681.3 STA (2 copias).

[Sta96] William Stallings. *Organización y Arquitectura de Computadores, diseño para optimizar prestaciones*. Prentice Hall, cuarta edición, 1996. BIBLIOTECA: CI 681.3 STA (4 copias), CI-Informática (1 copia).

[Sto93] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, primera y tercera edición, 1987 y 1993. BIBLIOTECA: CI 681.3 STO (2 copias), CI-Informática (2 copias).

[Tan95] Andrew S. Tanenbaum. *Distributed operating systems*. Prentice-Hall, 1995. BIBLIOTECA: CI 681.3.06 TAN (1 copia), CI-Informática (1 copia).

[Wil96] Barry Wilkinson. *Computer Architecture, design and performance*. Prentice-Hall, segunda edición, 1996. BIBLIOTECA: CI 681.3 WIL (3 copias), CI-Informática (1 copia).

[Zar96] Mehdi R. Zargham. *Computer Architecture, single and parallel systems*. Prentice-Hall, 1996. BIBLIOTECA: CI-Informática (2 copias).

ÍNDICE DE MATERIAS