VNIVERSITAT DE VALÈNCIA

UNIVERSITAT DE VALÈNCIA

AMPLIACIÓN DE ARQUITECTURA DE COMPUTADORES

Ingeniería Informática

Profesor: Fernando Pardo Carpio

Valencia, 8 de octubre de 2001

Prefacio

El temario recogido en estos apuntes se corresponde con la asignatura de *Ampliación de Arquitectura de Computadores* de la Ingeniería Informática de la Universidad de Valencia. Esta asignatura consta de 3 créditos teóricos y 1.5 prácticos, y se cursa en quinto. Esta asignatura es la continuación de la asignatura de *Arquitectura e Ingeniería de Computadores* de cuarto curso.

El contenido principal de la asignatura está formado por sistemas de altas prestaciones como los multicomputadores, sin olvidar otras arquitecturas de alto rendimiento como los procesadores vectoriales y matriciales. Por último se tratan algunas arquitecturas de aplicación específica.

Estos apuntes son un subconjunto de los apuntes de la asignatura de plan viejo *Arquitecturas Avanzadas*. Los contenidos de esta asignatura de plan viejo se han dividido entre dos asignaturas, por un lado la de *Ampliación de Arquitectura de Computadores*, y por el otro en parte (segundo cuatrimestre) de la asignatura de *Arquitectura e Ingeniería de los Computadores*. Por esta razón existen tres versiones de estos apuntes:

- *Arquitecturas Avanzadas*. Esta asignatura ya no se da pero son los apuntes más completos, ya que recogen todos los temas preparados hasta el momento.
- *Ampliación de Arquitectura de Computadores*. Se recogen los temas de vectoriales, matriciales, multicomputadores y otras arquitecturas avanzadas. Se incluye también una introducción al paralelismo como apéndice.
- *Arquitectura e Ingeniería de los Computadores* (segundo cuatrimestre). En esta asignatura se recogen el resto de temas de Arquitecturas Avanzadas como son el rendimiento de los sistemas paralelos, redes de interconexión y multiprocesadores.

Estos apuntes se empezaron a preparar en 1998. Posteriormente, 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

1 Procesadores vectoriales **1**
- 1.1 Procesador vectorial básico . . . 2
  - 1.1.1 Arquitectura vectorial básica . . . 2
  - 1.1.2 Instrucciones vectoriales básicas . . . 4
  - 1.1.3 Ensamblador vectorial DLXV . . . 6
  - 1.1.4 Tiempo de ejecución vectorial . . . 8
  - 1.1.5 Unidades de carga/almacenamiento vectorial . . . 10
- 1.2 Memoria entrelazada o intercalada . . . 11
  - 1.2.1 Acceso concurrente a memoria (acceso C) . . . 13
  - 1.2.2 Acceso simultáneo a memoria (acceso S) . . . 14
  - 1.2.3 Memoria de acceso C/S . . . 15
  - 1.2.4 Rendimiento de la memoria entrelazada y tolerancia a fallos . . . 15
- 1.3 Longitud del vector y separación de elementos . . . 18
  - 1.3.1 Control de la longitud del vector . . . 18
  - 1.3.2 Cálculo del tiempo de ejecución vectorial . . . 19
  - 1.3.3 Separación de elementos en el vector . . . 21
- 1.4 Mejora del rendimiento de los procesadores vectoriales . . . 23
  - 1.4.1 Encadenamiento de operaciones vectoriales . . . 23
  - 1.4.2 Sentencias condicionales . . . 23
  - 1.4.3 Matrices dispersas . . . 25
- 1.5 El rendimiento de los procesadores vectoriales . . . 27
  - 1.5.1 Rendimiento relativo entre vectorial y escalar . . . 27
  - 1.5.2 Medidas del rendimiento vectorial . . . 28
- 1.6 Historia y evolución de los procesadores vectoriales . . . 32

2 Procesadores matriciales **35**
- 2.1 Organización básica . . . 35
- 2.2 Estructura interna de un elemento de proceso . . . 36
- 2.3 Instrucciones matriciales . . . 38
- 2.4 Programación . . . 38
  - 2.4.1 Multiplicación SIMD de matrices . . . 38
- 2.5 Procesadores asociativos . . . 38
  - 2.5.1 Memorias asociativas . . . 38
  - 2.5.2 Ejemplos de procesadores asociativos . . . 38

3 Multicomputadores **39**
- 3.1 Redes de interconexión para multicomputadores . . . 41
  - 3.1.1 Topologías estrictamente ortogonales . . . 43
  - 3.1.2 Otras topologías directas . . . 45
  - 3.1.3 Conclusiones sobre las redes directas . . . 50
- 3.2 La capa de conmutación o control de flujo (*switching*) . . . 52
  - 3.2.1 Elementos básicos de la conmutación . . . 52
  - 3.2.2 Conmutación de circuitos . . . 56
  - 3.2.3 Conmutación de paquetes . . . 58

3.2.4 Conmutación de paso a través virtual, *Virtual Cut-Through* (VCT) 59
3.2.5 Conmutación de lombriz (*Wormhole*) 61
3.2.6 Conmutación cartero loco 63
3.2.7 Canales virtuales 66
3.2.8 Mecanismos híbridos de conmutación 68
3.2.9 Comparación de los mecanismos de conmutación 68
3.3 La capa de encaminamiento (*routing*) 70
3.3.1 Clasificación de los algoritmos de encaminamiento 71
3.3.2 Bloqueos 72
3.3.3 Teoría para la evitación de bloqueos mortales (*deadlocks*) 75
3.3.4 Algoritmos deterministas 81
3.3.5 Algoritmos parcialmente adaptativos 84
3.3.6 Algoritmos completamente adaptativos 87
3.3.7 Comparación de los algoritmos de encaminamiento 88

4 Otras arquitecturas avanzadas 93
4.1 Máquinas de flujo de datos 94
4.1.1 Grafo de flujo de datos 94
4.1.2 Estructura básica de un computador de flujo de datos 96
4.2 Otras arquitecturas 97

A Introducción a las arquitecturas avanzadas 99
A.1 Clasificación de Flynn 100
A.2 Otras clasificaciones 101
A.3 Introducción al paralelismo 108
A.3.1 Fuentes del paralelismo 108
A.3.2 El paralelismo de control 109
A.3.3 El paralelismo de datos 109
A.3.4 El paralelismo de flujo 111

B Comentarios sobre la bibliografía 113

Bibliografía 115

Índice de Materias 117

Índice de Figuras

1.1 La arquitectura de un supercomputador vectorial. 3
1.2 Estructura básica de una arquitectura vectorial con registros, DLXV. . 4
1.3 Características de varias arquitecturas vectoriales. 5
1.4 Instrucciones vectoriales del DLXV. 7
1.5 Penalización por el tiempo de arranque en el DLXV. 9
1.6 Tiempos de arranque y del primer y último resultados para los convoys 1-4. 10
1.7 Dos organizaciones de memoria entrelazada con $m = 2^a$ módulos y $w = 2^a$ palabras por módulo. 12
1.8 Tiempo de acceso para las primeras 64 palabras de doble precisión en una lectura. 13
1.9 Acceso segmentado a 8 palabras contiguas en una memoria de acceso C. 14
1.10 Organización de acceso S para una memoria entrelazada de m vías. . . 14
1.11 Organización de acceso C/S. 15
1.12 Dos organizaciones de memoria entrelazada usando 8 módulos: (a) 2 bancos y 4 módulos entrelazados, (b) 4 bancos y 2 módulos entrelazados. 17
1.13 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 ($n \mod MVL$). 19
1.14 Tiempo de ejecución por elemento en función de la longitud del vector. 21
1.15 Temporización para la ejecución no encadenada y encadenada. 24
1.16 Rendimiento relativo escalar/vectorial. 27
1.17 Formación de convoys en el bucle interior del código DAXPY. 28
1.18 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=$n \times n$). 33

2.1 Computador matricial básico. 36
2.2 Componentes de un elemento de proceso (EP) en un computador matricial. 37

3.1 Arquitectura básica de un multicomputador. 40
3.2 Clasificación de las redes de interconexión 42
3.3 Variaciones de mallas y toros. 44
3.4 Topologías estrictamente ortogonales en una red directa. 45
3.5 Hipercubos, ciclo cubos y n-cubos k-arios. 46
3.6 Otras topologías directas. 47
3.7 Matriz lineal, anillos, y barril desplazado. 48
3.8 Algunas topologías árbol. 49
3.9 Árbol, estrella y árbol grueso. 50
3.10 Distintas unidades de control de flujo en un mensaje. 53
3.11 Un ejemplo de control de flujo asíncrono de un canal físico. 53
3.12 Un ejemplo de control de flujo síncrono de un canal físico. 54
3.13 Modelo de encaminador (*router*) genérico. (LC = Link controller.) . . . 55

3.14 Cálculo de la latencia en una red para el caso de ausencia de carga (R = Encaminador o *Router*). 56
3.15 Cronograma de un mensaje por conmutación de circuitos. 57
3.16 Un ejemplo del formato en una trama de creación de un circuito. (CHN = Número de canal; DEST = Dirección destino; XXX = No definido.) . 57
3.17 Cronograma de un mensaje por conmutación de paquetes. 59
3.18 Un ejemplo de formato de la cabecera del paquete. (DEST = Dirección destino; LEN = Longitud del paquete en unidades de 192 bytes; XXX = No definido.) 59
3.19 Cronograma para un mensaje conmutado por VCT. ($t_{blocking}$= Tiempo de espera en un enlace de salida.) 60
3.20 Cronograma de un mensaje conmutado mediante wormhole. 61
3.21 Un ejemplo de mensaje bloqueado con la técnica wormhole. 62
3.22 Formato de los paquetes conmutados mediante wormhole en el Cray T3D. 62
3.23 Cronograma de la transmisión de un mensaje usando la conmutación del cartero loco. 63
3.24 Un ejemplo del formato de un mensaje en la técnica de conmutación del cartero loco. 64
3.25 Ejemplo de encaminamiento con la conmutación del cartero loco y la generación de flits de dirección muertos. 65
3.26 Canales virtuales. 67
3.27 Un ejemplo de la reducción del retraso de la cabecera usando dos canales virtuales por canal físico. 68
3.28 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.) 69
3.29 Una taxonomía de los algoritmos de encaminamiento 71
3.30 Configuración de bloqueo en una malla 2-D. 73
3.31 Una clasificación de las situaciones que pueden impedir el envío de paquetes. 74
3.32 Configuración ilegal para R. 76
3.33 Redes del ejemplo anterior. 78
3.34 Red del ejemplo anterior. 79
3.35 El algoritmo de encaminamiento XY para mallas 2-D. 82
3.36 El algoritmo de encaminamiento por dimensiones para hipercubos. . . . 83
3.37 Grafo de dependencias entre canales para anillos unidireccionales. . . . 83
3.38 El algoritmo de encaminamiento por dimensiones para toros 2-D unidireccionales. 85
3.39 Caminos permitidos en el encaminamiento totalmente adaptativo y adaptativo por planos. 86
3.40 Redes crecientes y decrecientes en el plano A_i para el encaminamiento adaptativo por planos. 86
3.41 Latencia media del mensaje vs. tráfico normalizado aceptado para mallas 16×16 para una distribución uniforme del destino de los mensajes. . . 89
3.42 Latencia media del mensaje vs. tráfico normalizado aceptado para mallas $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes. . 90
3.43 Latencia media del mensaje vs. tráfico normalizado aceptado para toros 16×16 para una distribución uniforme del destino de los mensajes. . . 91
3.44 Latencia media del mensaje vs. tráfico normalizado aceptado para toros $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes. . 92

3.45 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 92

4.1 Grafo de flujo de datos para calcular $N!$ 95
4.2 La máquina MIT de flujo de datos. 96
4.3 Ejemplo de máquina de flujo de datos estática. 97
4.4 Ejemplo de máquina de flujo de datos dinámica. 98

A.1 Clasificación de Flynn de las arquitecturas de computadores. 101
A.2 Clasificación de las arquitecturas paralelas. 102
A.3 El modelo UMA de multiprocesador. 103
A.4 El modelo NUMA de multiprocesador. 104
A.5 El modelo COMA de multiprocesador. 104
A.6 Diagrama de bloques de una máquina de flujo de datos. 106
A.7 Paralelismo de control. 109
A.8 Ejemplo de paralelismo de control. 110
A.9 Paralelismo de datos. 110
A.10 Ejemplo de la aplicación del paralelismo de datos a un bucle. 111
A.11 Paralelismo de flujo. 111

Capítulo 1

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].

1.1 Procesador vectorial básico

1.1.1 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 1.1 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.

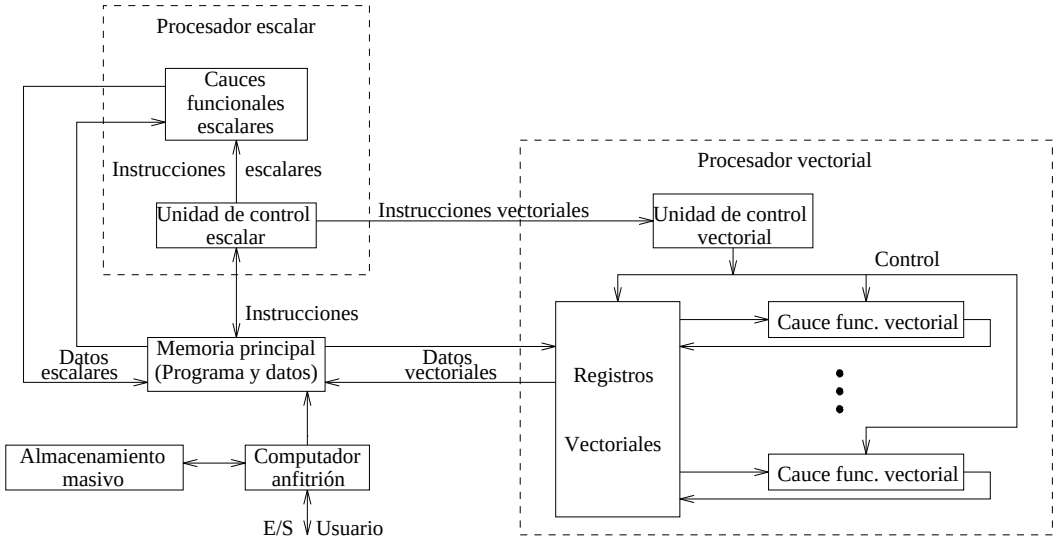


Figura 1.1: La arquitectura de un supercomputador vectorial.

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 1.2.

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.

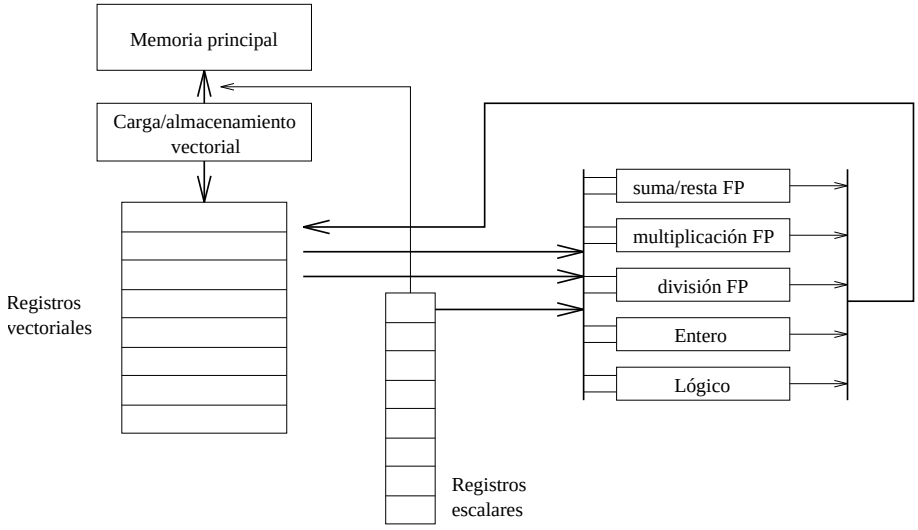


Figura 1.2: Estructura básica de una arquitectura vectorial con registros, DLXV.

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 1.3 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.

1.1.2 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 1.3: 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:

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

1.1.3 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 1.4 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 1.4: 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.

1.1.4 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 1.5: 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 1.5. 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 1.6 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 1.6: 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

$$\text{Profundidad del cauce} = \left\lceil \frac{\text{Tiempo total de ejecución de la unidad}}{\text{Periodo de reloj}} \right\rceil \tag{1.1}$$

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.

1.1.5 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 1.2. 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.

1.2 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.

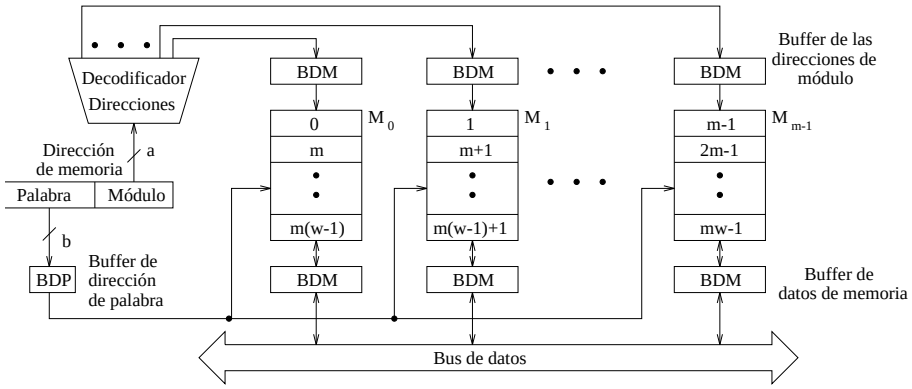
Consideremos una memoria principal formada por $m = 2^a$ módulos, cada uno con $w = 2^b$ palabras o celdas de memoria. La capacidad total de la memoria es $m\,w = 2^{a+b}$ palabras. A estas palabras se les asignan direcciones de forma lineal. Las diferentes formas en las que se asignan linealmente las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

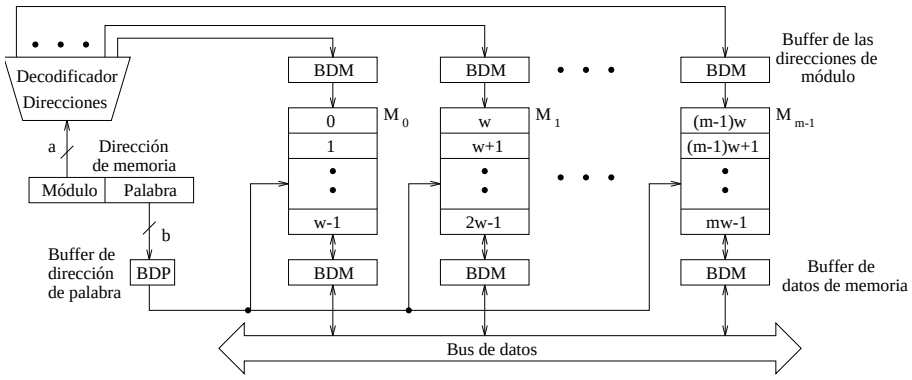
La figura 1.7 muestra dos formatos de direcciones para realizar una memoria entrelazada. El *entrelazado de orden bajo* (figura 1.7(a)) reparte las localizaciones contiguas de memoria entre los m módulos de forma horizontal. Esto implica que los a bits de orden bajo de la dirección se utilizan para identificar 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 1.7(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 1.7: 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 1.1 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 1.1: Direcciones de memoria (en bytes) y momento en el cual comienza el acceso a cada banco.

La figura 1.8 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 1.1 y la figura 1.8: 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).

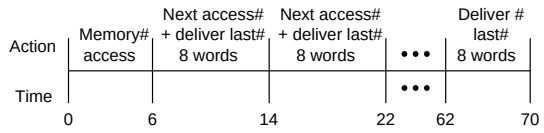


Figura 1.8: Tiempo de acceso para las primeras 64 palabras de doble precisión en una lectura.

1.2.1 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:

$$\tau = \frac{\theta}{m} \tag{1.2}$$

donde m es el *grado de entrelazado*. La temporización del acceso segmentado de 8 palabras contiguas en memoria se muestra en la figura 1.9. 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.

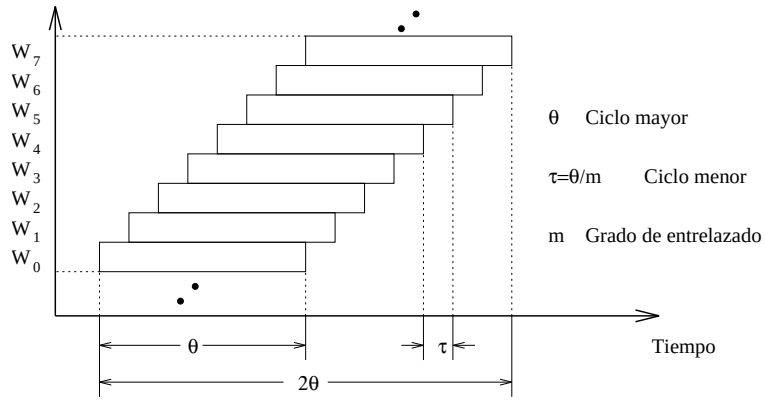


Figura 1.9: Acceso segmentado a 8 palabras contiguas en una memoria de acceso C.

1.2.2 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 1.10.

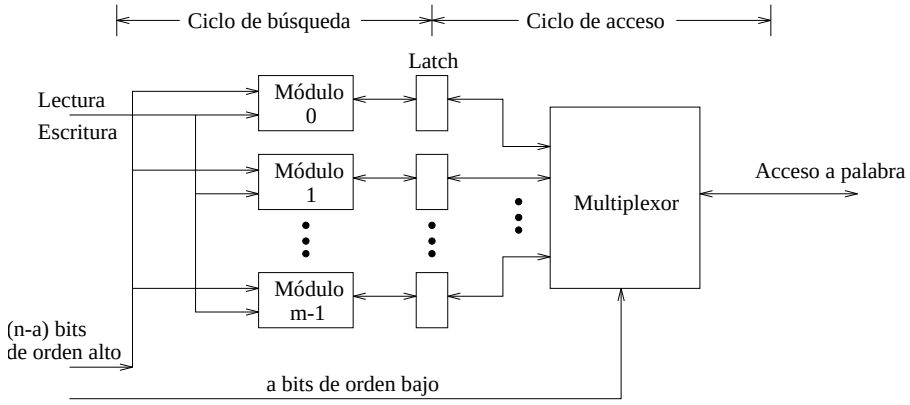


Figura 1.10: Organización de acceso S para una memoria entrelazada de m vías.

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. 1.2), 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.

1.2.3 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 1.11, 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.

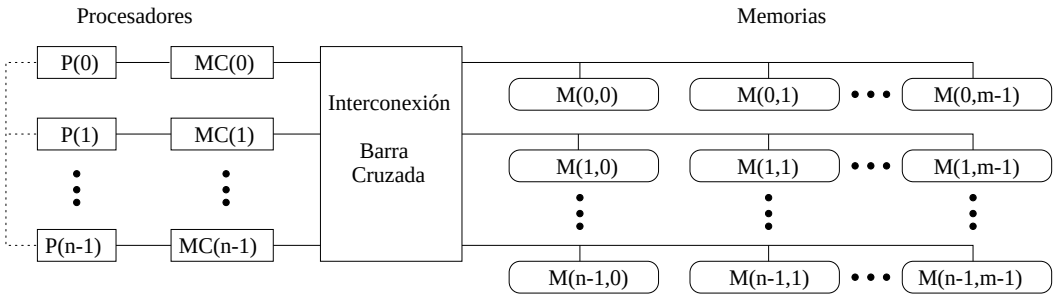


Figura 1.11: Organización de acceso C/S.

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.

1.2.4 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 1.9, 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 (1.2). 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.

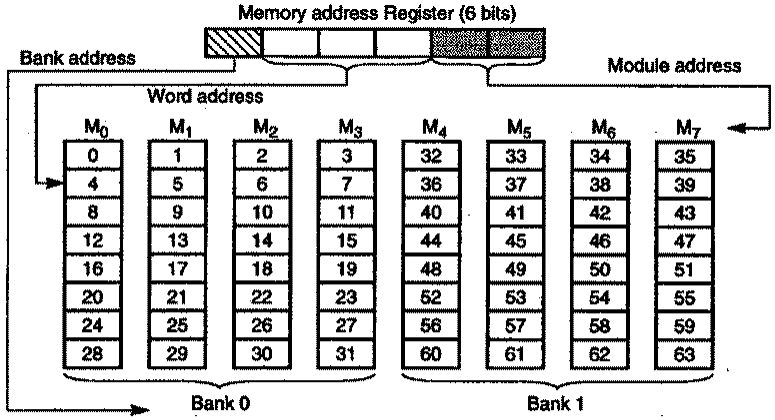
Tolerancia a fallos

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.

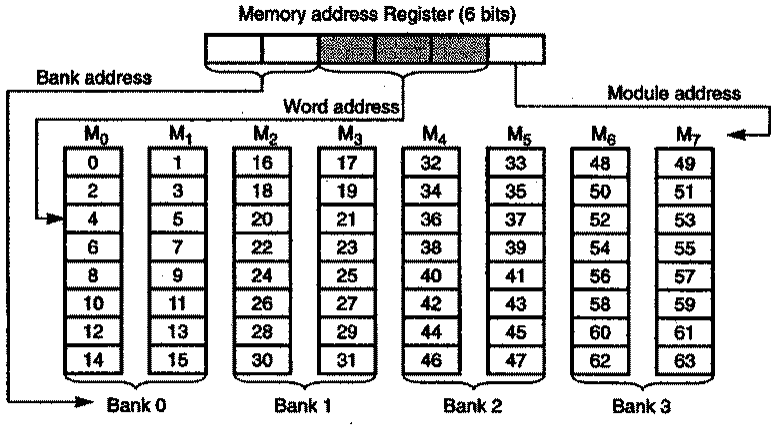
En muchas ocasiones interesa tener ambas características a un tiempo, es decir, por un lado interesa tener memoria entrelazada de orden bajo para acelerar el acceso a la memoria, pero por otro interesa también una memoria entrelazada de orden alto para tener la memoria dividida en bloques y poder seguir 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 1.12 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 1.12a)

se muestra una memoria de cuatro módulos de orden bajo y dos bancos de memoria. En el segundo ejemplo (figura 1.12b) 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 1.12: 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.

1.3 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.

1.3.1 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 1.13. 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,

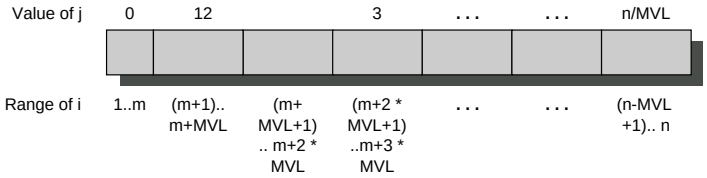


Figura 1.13: Un vector de longitud arbitraria procesado mediante seccionamiento. Todos los bloques menos el primero son de longitud MVL. En esta figura, la variable m se usa en lugar de la expresión ($n \mod MVL$).

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.

1.3.2 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:

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{bucle} + T_{arranque}) + n \times T_{campanada} \tag{1.3}$$

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 (1.3), 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 1.14 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.

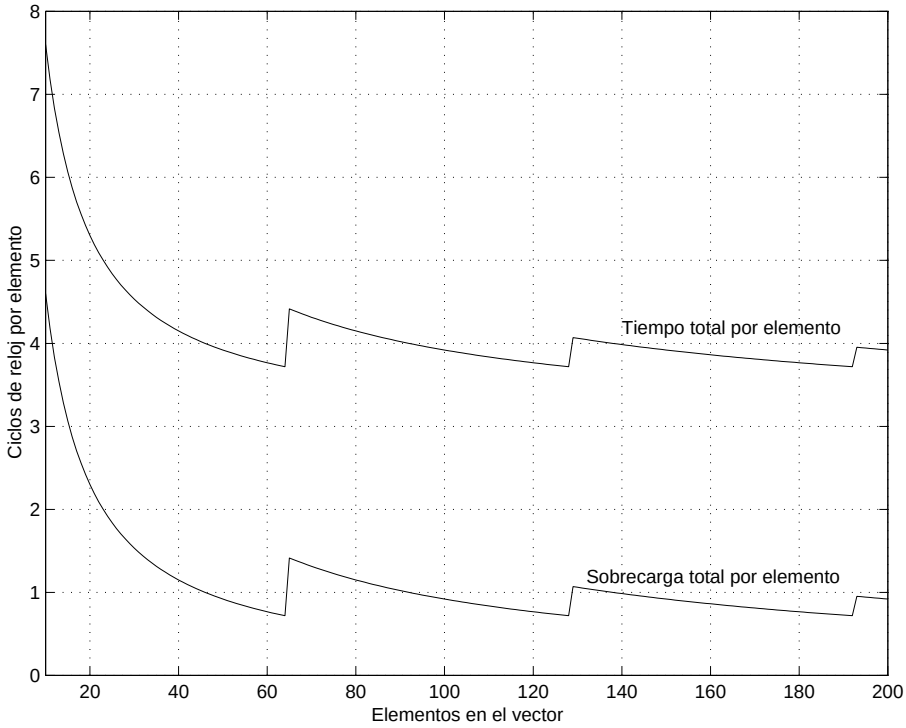


Figura 1.14: Tiempo de ejecución por elemento en función de la longitud del vector.

1.3.3 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.

1.4 Mejora del rendimiento de los procesadores vectoriales

1.4.1 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 1.15 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.

1.4.2 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

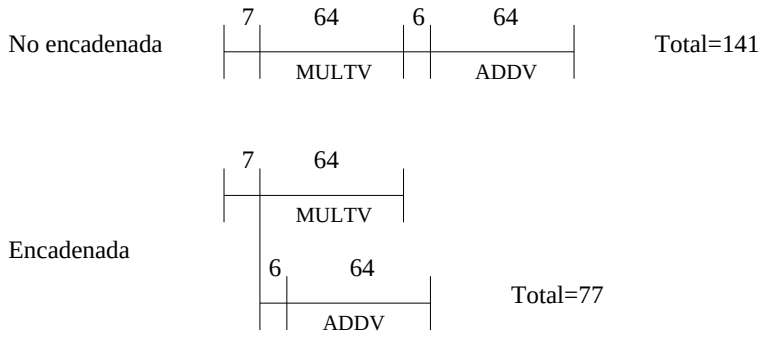


Figura 1.15: Temporización para la ejecución no encadenada y encadenada.

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.

1.4.3 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 24:

```
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.

1.5 El rendimiento de los procesadores vectoriales

1.5.1 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*:

$$P = \frac{1}{(1-f) + f/r} = \frac{r}{(1-f)r + f} \tag{1.4}$$

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 1.16 muestra el rendimiento relativo para una máquina vectorial en función de r y para varios valores de f.

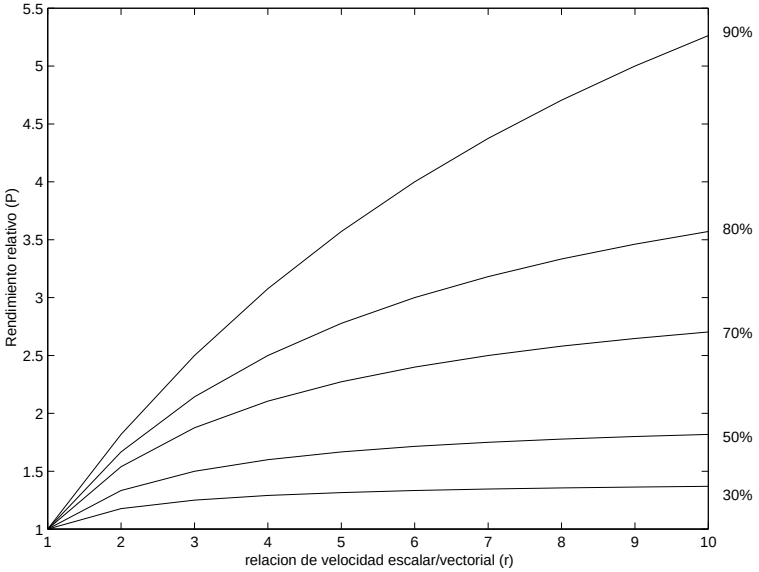


Figura 1.16: Rendimiento relativo escalar/vectorial.

1.5.2 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 1.17 (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 1.17: 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.

1.6 Historia y evolución de los procesadores vectoriales

Para finalizar, la figura 1.18 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.

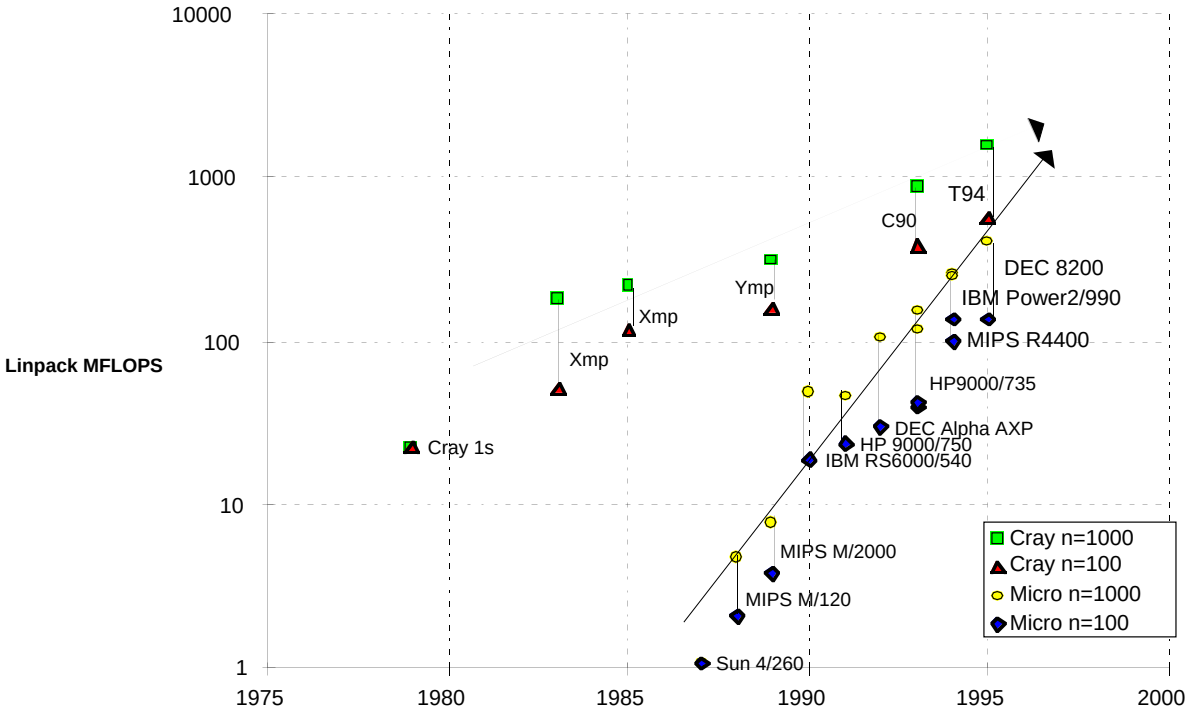


Figura 1.18: Comparación del rendimiento de los procesadores vectoriales y los microprocesadores escalares para la resolución de un sistema de ecuaciones lineales denso (tamaño de la matriz=$n \times n$).

Capítulo 2

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].

2.1 Organización básica

La configuración básica de un procesador matricial se muestra en la figura 2.1. 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 2.1, 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:

$$M = \langle N, C, I, M, R \rangle \tag{2.1}$$

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.

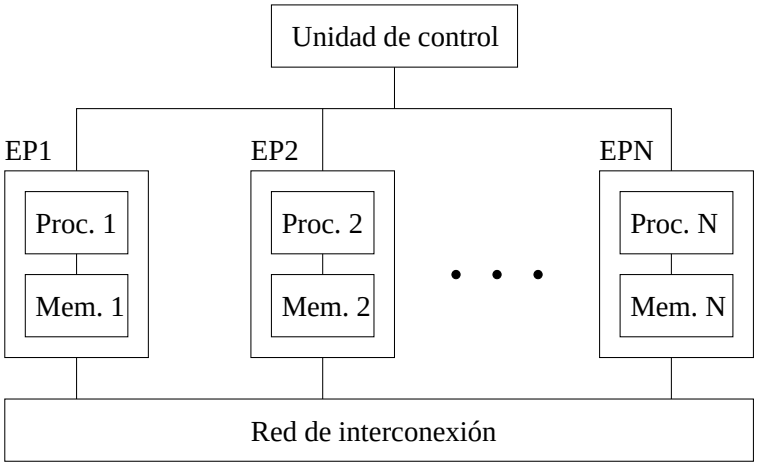


Figura 2.1: Computador matricial básico.

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.

2.2 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 2.2 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

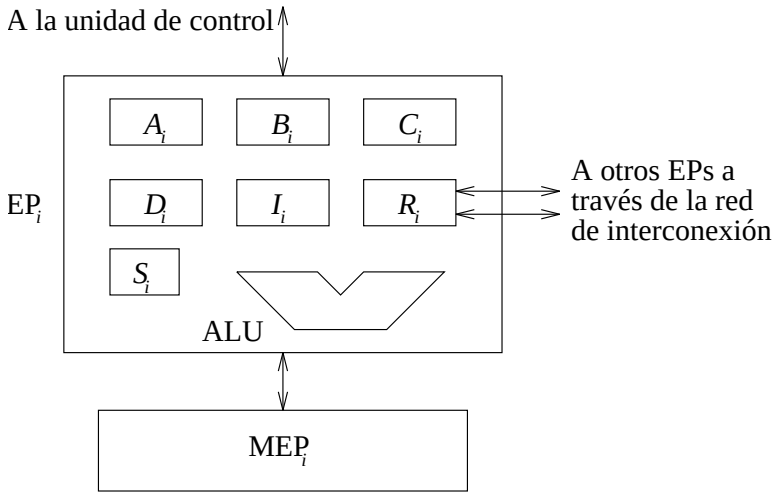


Figura 2.2: Componentes de un elemento de proceso (EP) en un computador matricial.

(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.

2.3 Instrucciones matriciales

2.4 Programación

2.4.1 Multiplicación SIMD de matrices

2.5 Procesadores asociativos

2.5.1 Memorias asociativas

2.5.2 Ejemplos de procesadores asociativos

Capítulo 3

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 3.1. 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

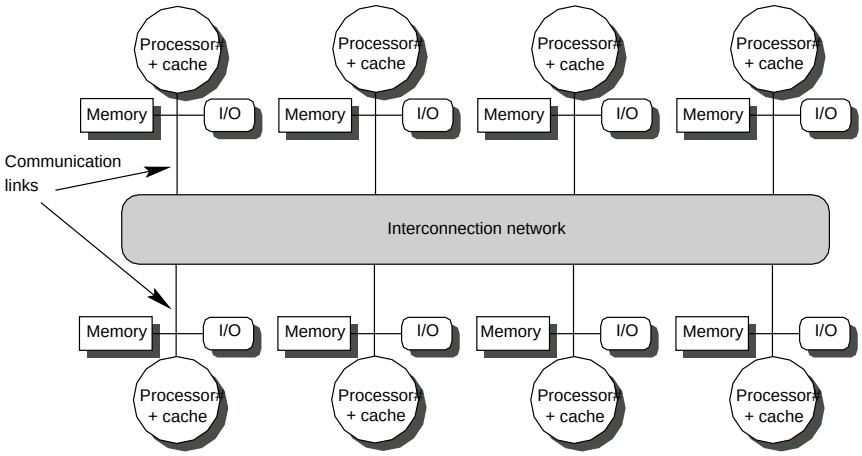


Figura 3.1: Arquitectura básica de un multicomputador.

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.

3.1 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 3.2 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 3.2: Clasificación de las redes de interconexión

3.1.1 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

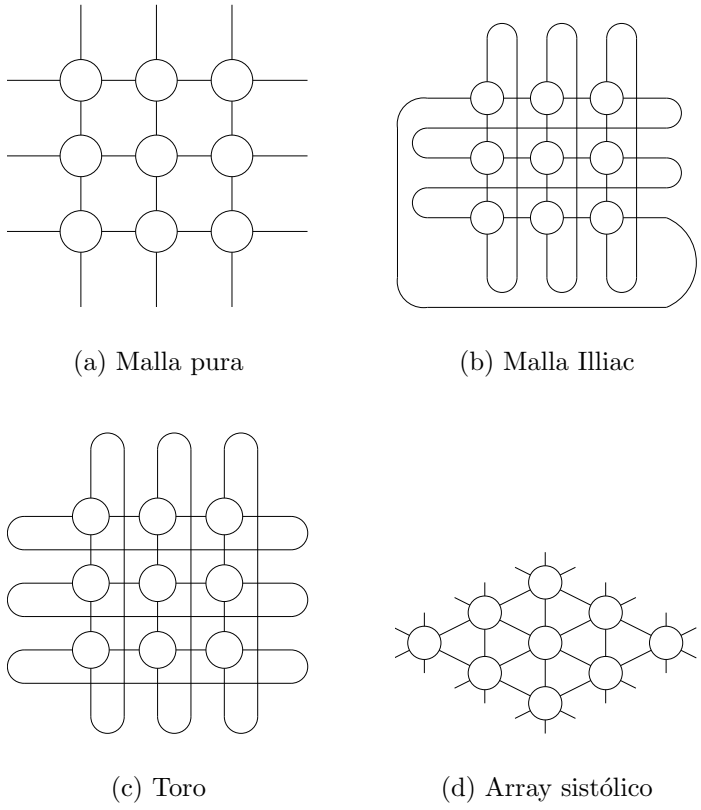
Formalmente, un malla n-dimensional tiene $k_0 \times k_1 \times \cdots \times k_{n-2} \times k_{n-1}$ nodos, k_i nodos en cada dimensión i, donde $k_i \geq 2$ y $0 \leq i \leq n-1$. Cada nodo X está definido por sus n coordenadas, $(x_{n-1}, x_{n-2}, \ldots, x_1, x_o)$, donde $0 \leq x_i \leq k_i - 1$ 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 3.3(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 3.3(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 3.3(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 3.7(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 3.3(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 3.3: 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.

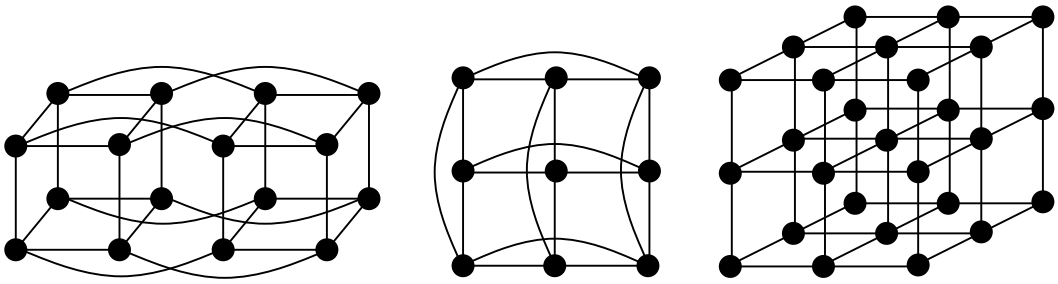
A las redes n-cubo k-aria también se les suele llamar toros. En general, un toro $n \times n$ binario tiene un grado nodal de 4 y un diámetro $d = 2\lfloor n/2 \rfloor$. El toro es una topología simétrica. Todas las conexiones de contorno añadidas ayudan a reducir el diámetro de la malla pura a la mitad.

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 3.4(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 3.4(c) muestra una

malla tridimensional.



(a) 2−ary 4−cube (hypercube) (b) 3−ary 2−cube (c) 3−ary 3−D mesh

Figura 3.4: Topologías estrictamente ortogonales en una red directa.

En la figura 3.5 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.

3.1.2 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.

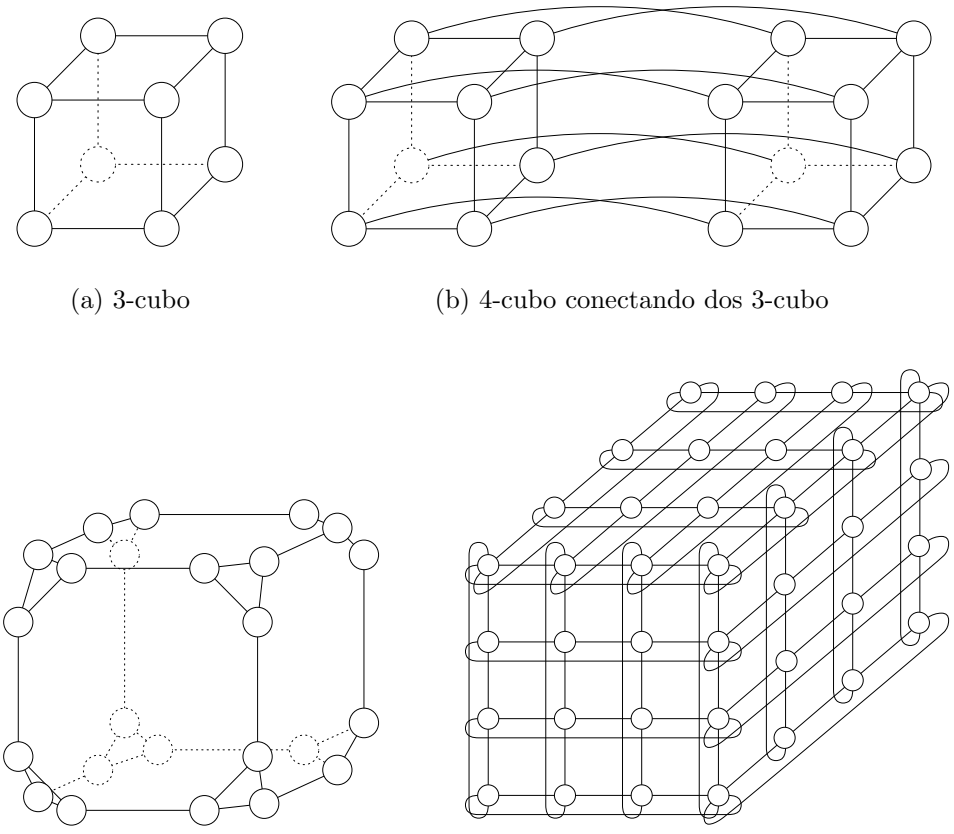
Las topologías que se presentan a continuación tienen diferentes objetivos: algunas intentan simplificar la red a costa de un mayor diámetro, otras pretenden disminuir el diámetro y aumentando los nodos manteniendo un compromiso de complejidad no muy elevada.

La figura 3.6 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 3.7(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 3.5: Hipercubos, ciclo cubos y n-cubos k-arios.

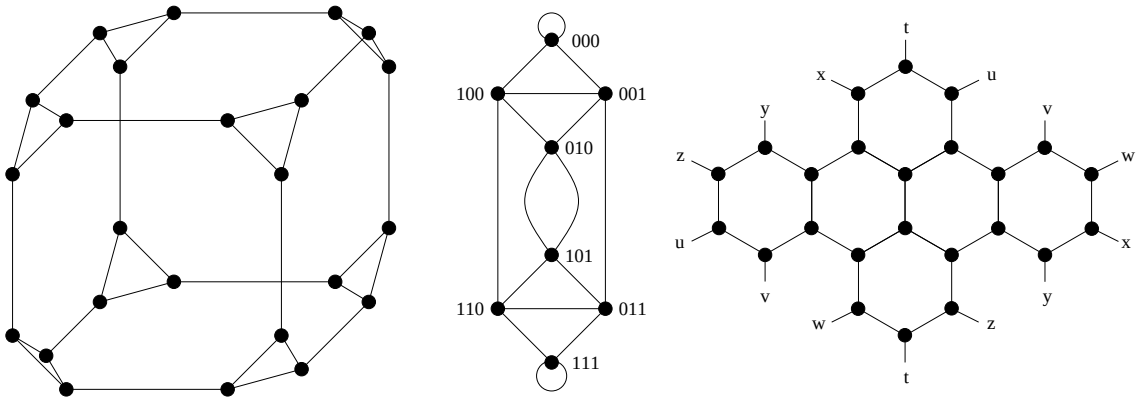
simultánea de diferentes secciones (canales) de la matriz con diferentes orígenes y destinos.

Anillo

Un *anillo* se obtiene conectando los dos nodos terminales de una matriz lineal mediante un enlace más como se muestra en la figura 3.7(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 3.7(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 3.6: 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 3.7(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

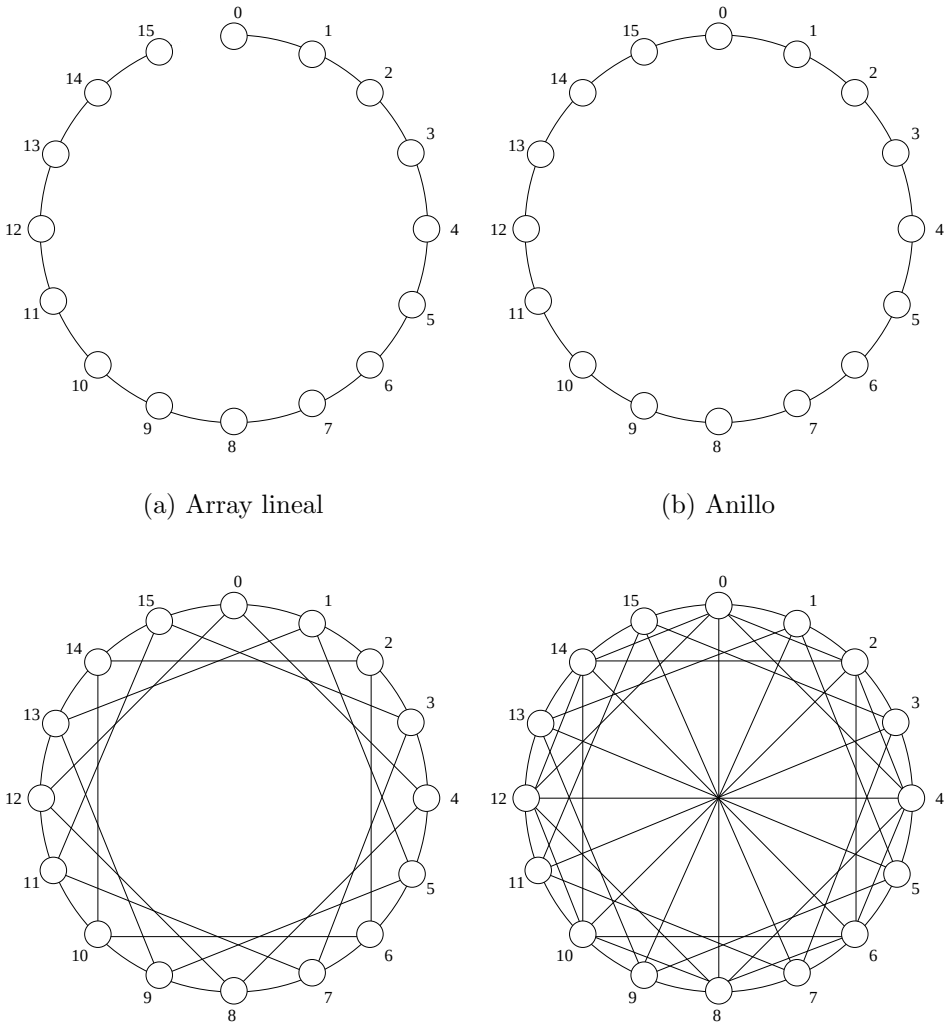
Esta arquitectura se realiza a partir del hipercubo y consiste en sustituir cada vértice del cubo por un anillo (ciclo) normalmente con un número igual de nodos que de dimensiones del cubo. Las figuras 3.5(c) y 3.6(a) muestran un 3-ciclo cubo conectado (un 3-CCC).

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

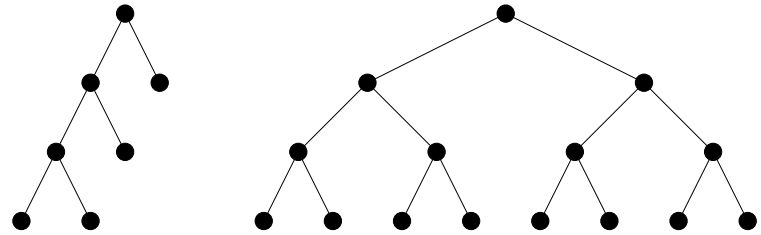
Figura 3.7: Matriz lineal, anillos, y barril desplazado.

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 3.8(a) y 3.8(b) muestran un árbol binario no balanceado y balanceado, respectivamente.



(a) Binary Tree (b) Balanced Binary Tree

Figura 3.8: 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*).

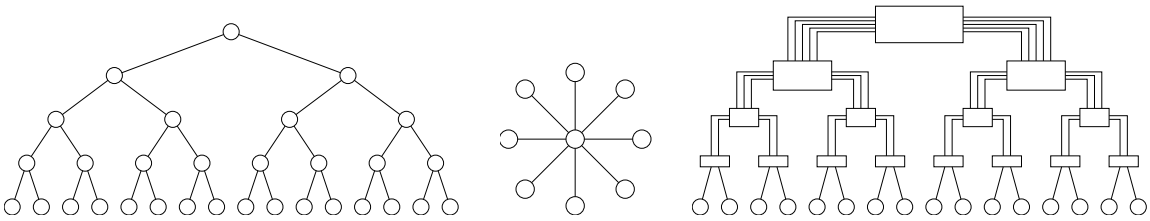
La estructura convencional de árbol utilizada en ciencias de la computación, puede ser modificada para conseguir un *árbol grueso*. Un árbol grueso binario se muestra en la figura 3.9(c). La anchura de canal de un árbol grueso se incrementa conforme se sube de las hojas a la raíz. El árbol grueso es 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 3.9(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 3.9(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 3.9: 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 3.3(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.

3.1.3 Conclusiones sobre las redes directas

En la tabla 3.1 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 3.1: 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.

3.2 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.

3.2.1 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 3.10 muestra N paquetes, 6 flits/paquete y 2 phits/flit.

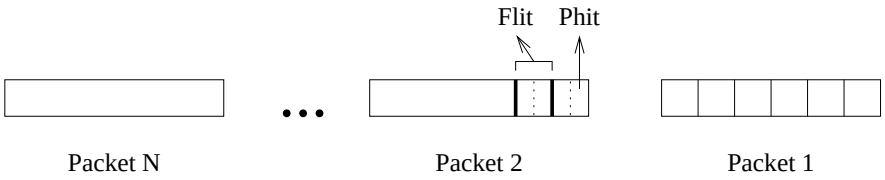


Figura 3.10: Distintas unidades de control de flujo en un mensaje.

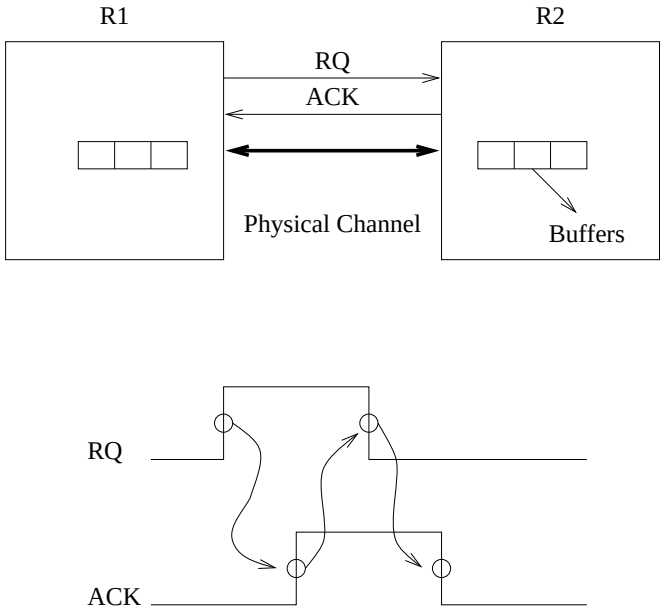


Figura 3.11: Un ejemplo de control de flujo asíncrono de un canal físico.

Existen muchos candidatos a protocolo de sincronización para coordinar la transferencia de bits a través de un canal. La figura 3.11 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.

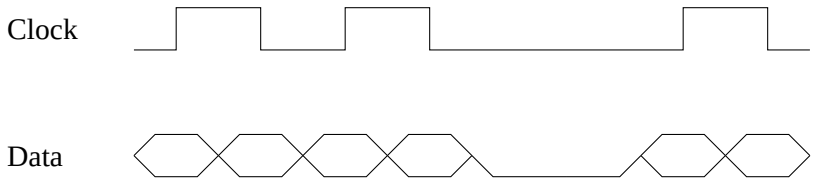


Figura 3.12: Un ejemplo de control de flujo síncrono de un canal físico.

El control de flujo del canal también puede ser síncrono, como se muestra en la figura 3.12. 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 3.12 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 3.13 y está compuesto de los siguientes elementos principales:

- *Buffers*. Son buffers FIFO que permiten almacenar mensajes en tránsito. En el modelo de la figura 3.13, 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

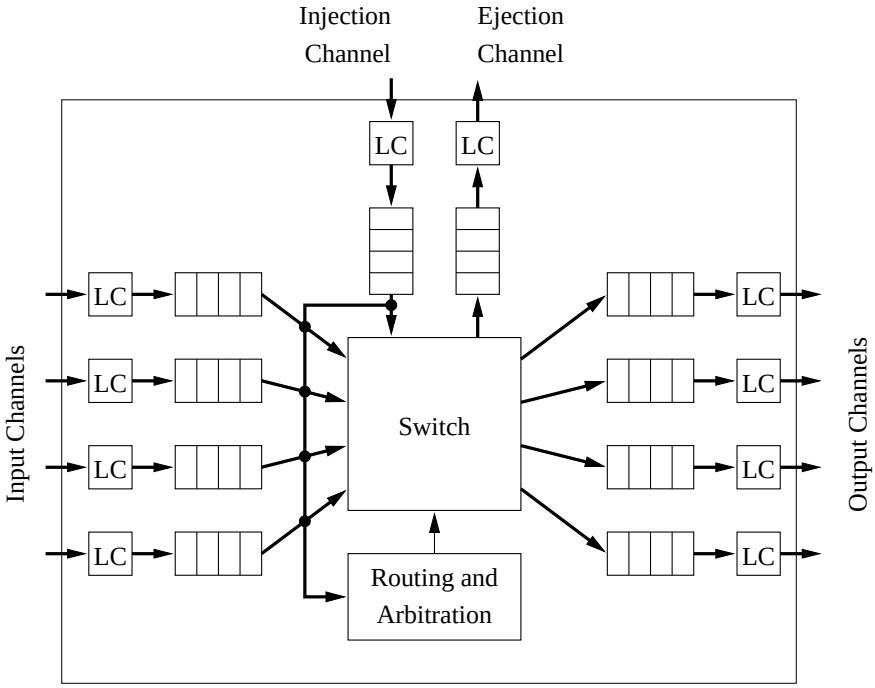


Figura 3.13: Modelo de encaminador (*router*) genérico. (LC = Link controller.)

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 3.14.

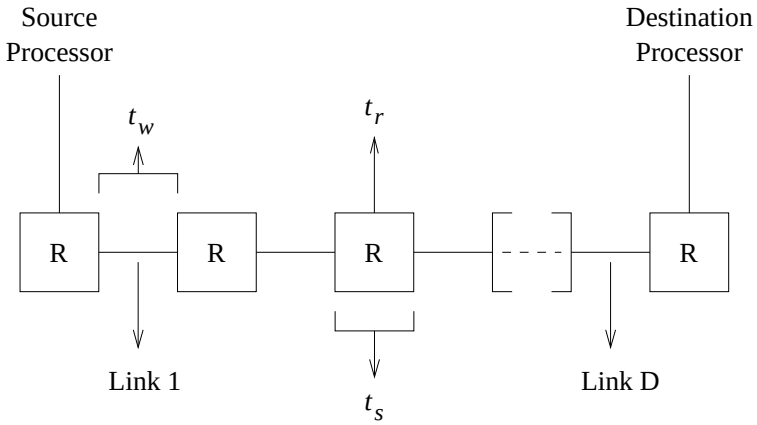


Figura 3.14: Cálculo de la latencia en una red para el caso de ausencia de carga (R = Encaminador o *Router*).

3.2.2 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 3.15 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 3.16 se muestra un ejemplo del formato de la trama de creación de un circuito.

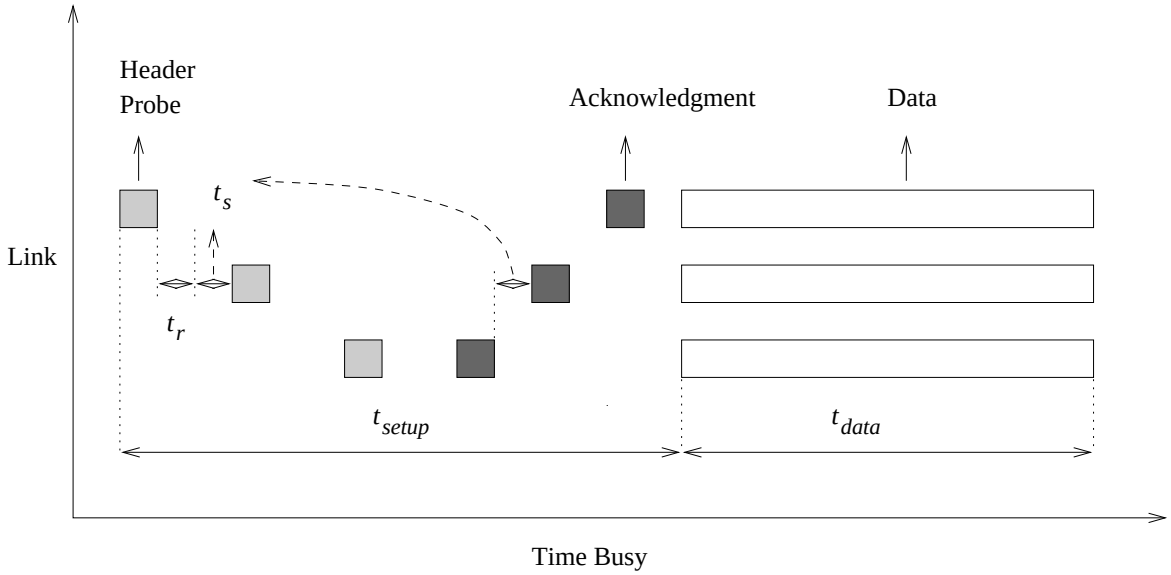


Figura 3.15: Cronograma de un mensaje por conmutación de circuitos.

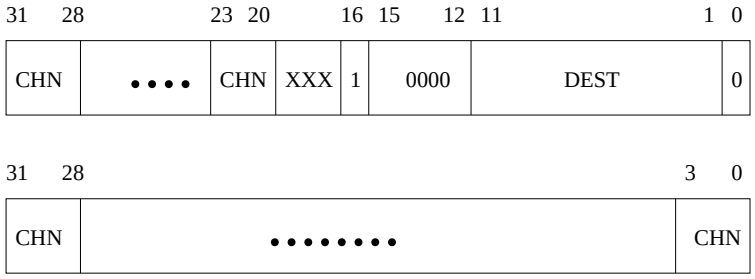


Figura 3.16: Un ejemplo del formato en una trama de creación de un circuito. (CHN = Número de canal; DEST = Dirección destino; XXX = No definido.)

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 3.13. 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:

$$\begin{aligned} t_{circuit} &= t_{setup} + t_{data} \\ t_{setup} &= D[t_r + 2(t_s + t_w)] \\ t_{data} &= \frac{1}{B}\left\lceil \frac{L}{W} \right\rceil \end{aligned} \tag{3.1}$$

3.2.3 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 3.17 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 3.18 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:

$$t_{packet} = D\left\{ t_r + (t_s + t_w)\left\lceil \frac{L+W}{W} \right\rceil \right\} \tag{3.2}$$

Esta expresión sigue el modelo de encaminador (*router*) mostrado en la figura 3.13,

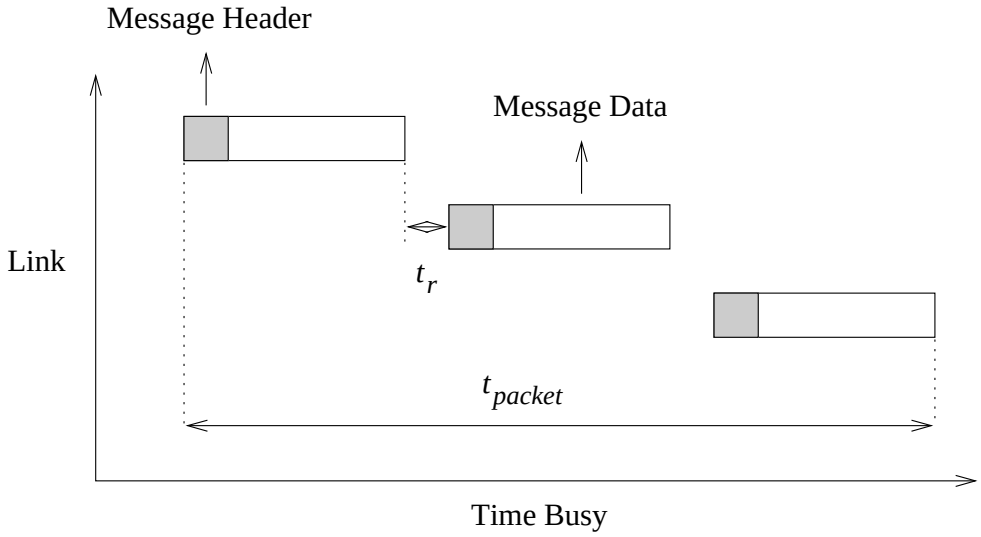


Figura 3.17: Cronograma de un mensaje por conmutación de paquetes.

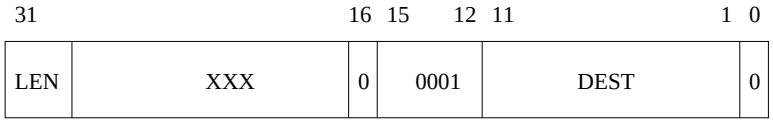


Figura 3.18: Un ejemplo de formato de la cabecera del paquete. (DEST = Dirección destino; LEN = Longitud del paquete en unidades de 192 bytes; XXX = No definido.)

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.

3.2.4 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 3.13. 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.

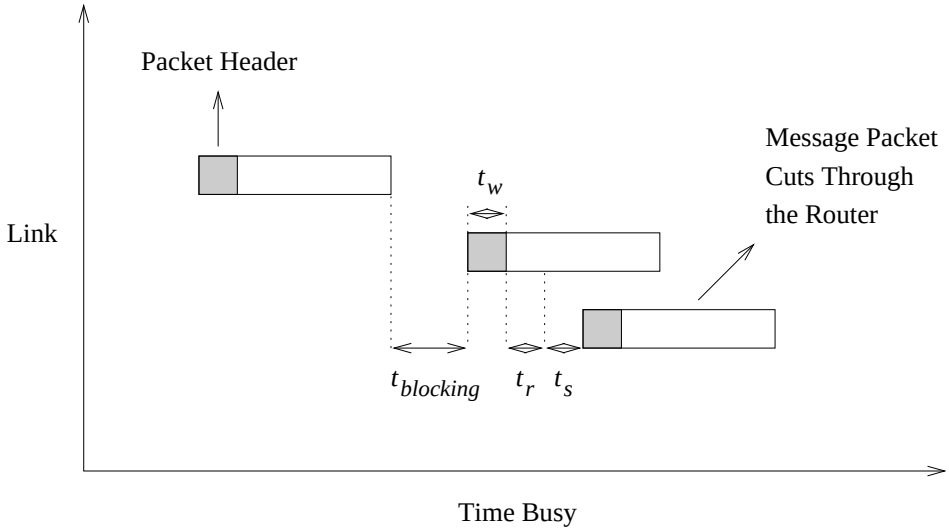


Figura 3.19: Cronograma para un mensaje conmutado por VCT. ($t_{blocking}$= Tiempo de espera en un enlace de salida.)

La figura 3.19 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:

$$t_{vct} = D(t_r + t_s + t_w) + \max(t_s, t_w) \left\lceil \frac{L}{W} \right\rceil \tag{3.3}$$

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.

3.2.5 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 3.20 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.

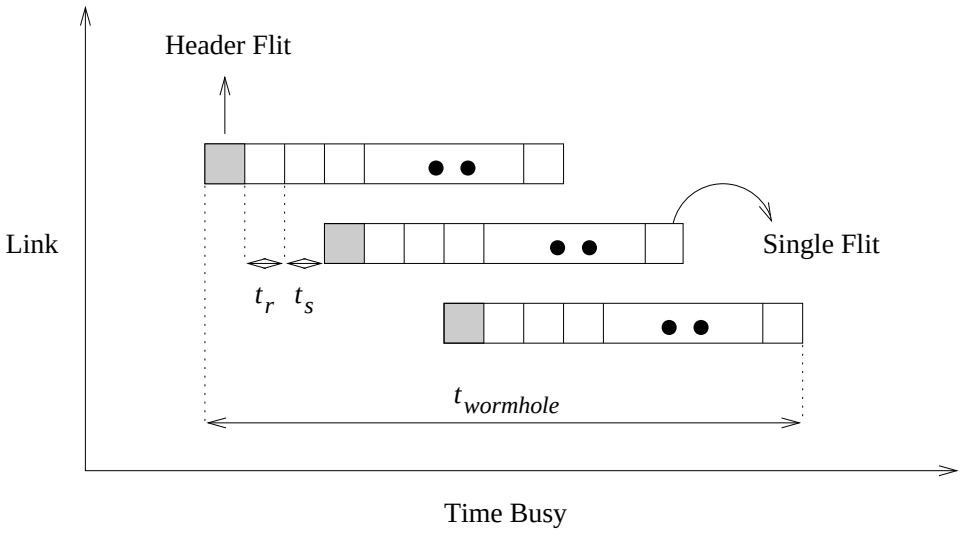


Figura 3.20: Cronograma de un mensaje conmutado mediante wormhole.

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 3.21 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 3.22 muestra el formato de los paquetes en el Cray T3D.

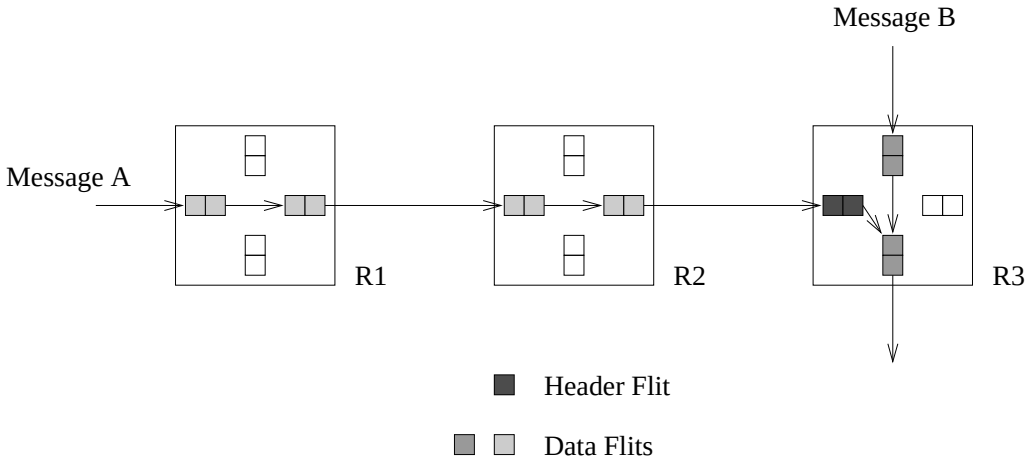


Figura 3.21: Un ejemplo de mensaje bloqueado con la técnica wormhole.

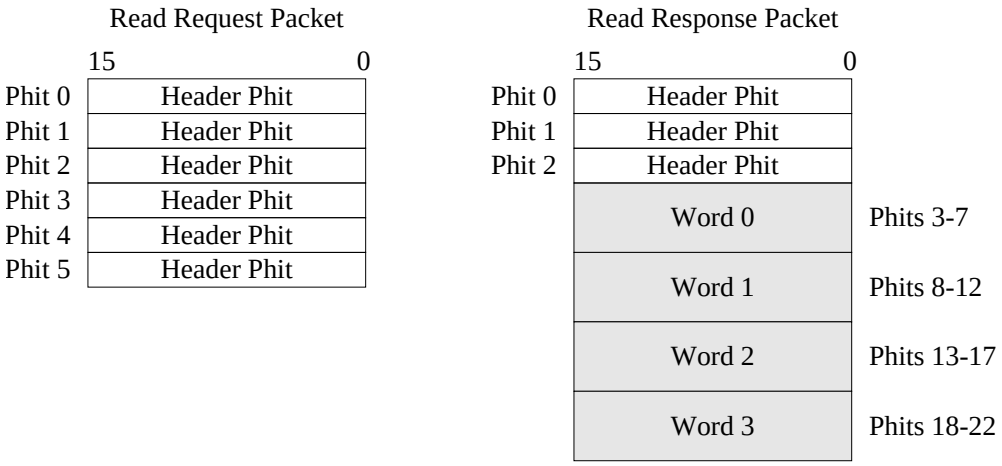


Figura 3.22: Formato de los paquetes conmutados mediante wormhole en el Cray T3D.

La latencia base para un mensaje conmutado mediante wormhole puede calcularse como sigue:

$$t_{wormhole} = D(t_r + t_s + t_w) + \max(t_s, t_w) \left\lceil \frac{L}{W} \right\rceil \tag{3.4}$$

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.

3.2.6 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.

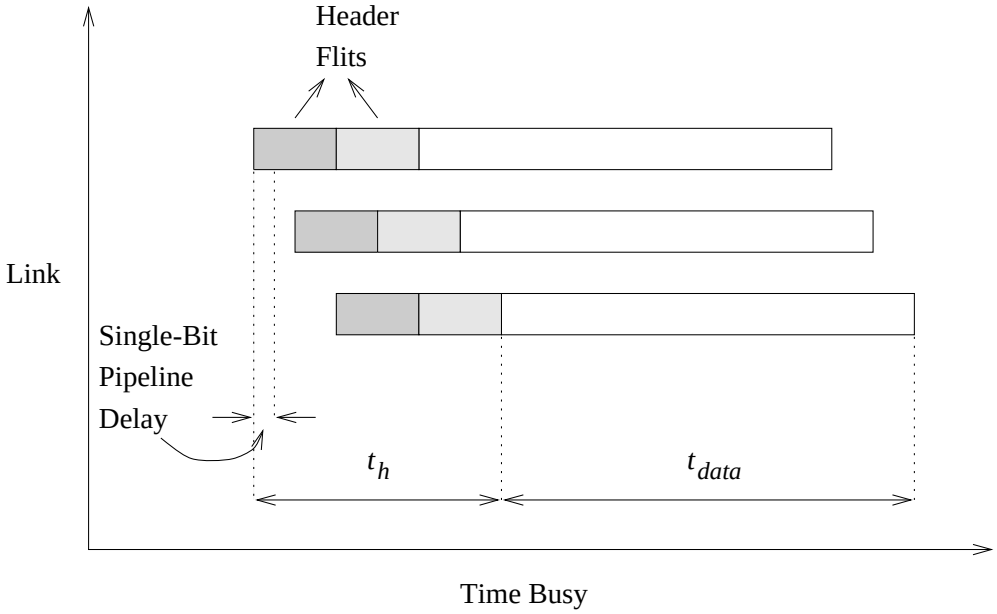


Figura 3.23: Cronograma de la transmisión de un mensaje usando la conmutación del cartero loco.

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 3.23 muestra un cronograma de la transmisión de un mensaje que se transmite sobre tres enlaces usando esta técnica de conmutación.

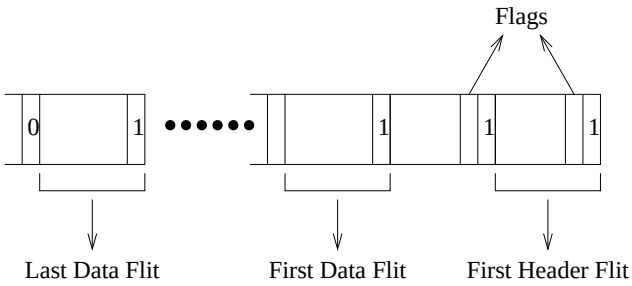


Figura 3.24: Un ejemplo del formato de un mensaje en la técnica de conmutación del cartero loco.

Es necesario considerar algunas restricciones en la organización de la cabecera. la figura 3.24 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

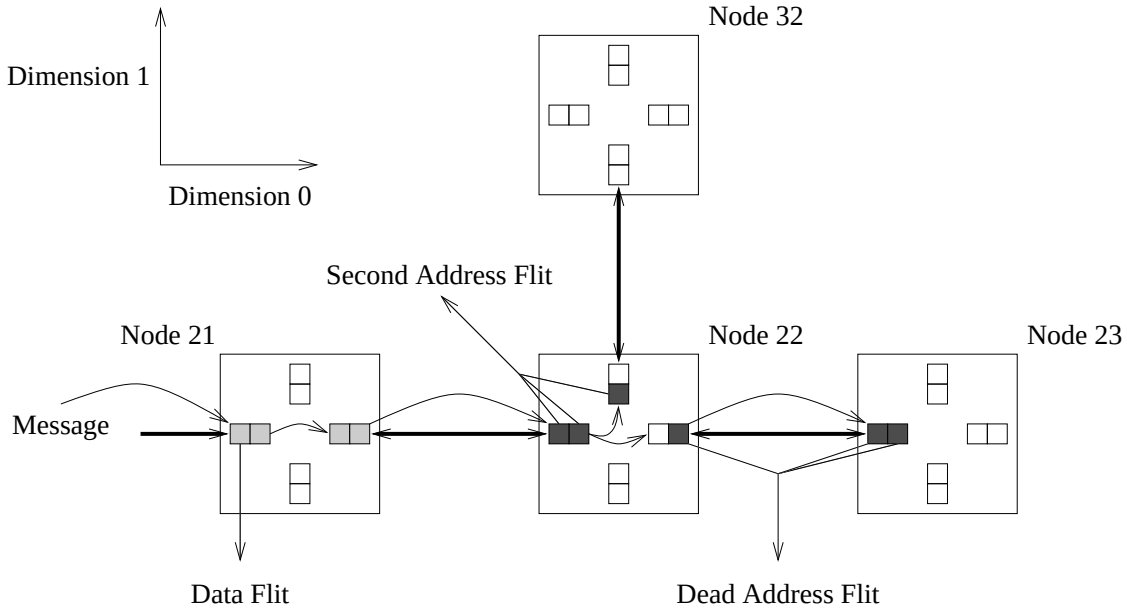


Figura 3.25: Ejemplo de encaminamiento con la conmutación del cartero loco y la generación de flits de dirección muertos.

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 3.25 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:

$$\begin{aligned} t_{madpostman} &= t_h + t_{data} \\ t_h &= (t_s + t_w)D + \max(t_s, t_w)W \\ t_{data} &= \max(t_s, t_w)L \end{aligned} \tag{3.5}$$

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

$$t_{madpostman} = D(t_s + t_w) + \max(t_s, t_w) \left\lceil \frac{W}{C} \right\rceil + \max(t_s, t_w) \left\lceil \frac{L}{C} \right\rceil \tag{3.6}$$

Por razones de comparación, en este caso la expresión de la conmutación segmentada habría sido

$$t_{wormhole} = D \left\{ t_r + (t_s + t_w) \left\lceil \frac{W}{C} \right\rceil \right\} + \max(t_s, t_w) \left\lceil \frac{L}{C} \right\rceil \tag{3.7}$$

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.

3.2.7 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 3.26. 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.

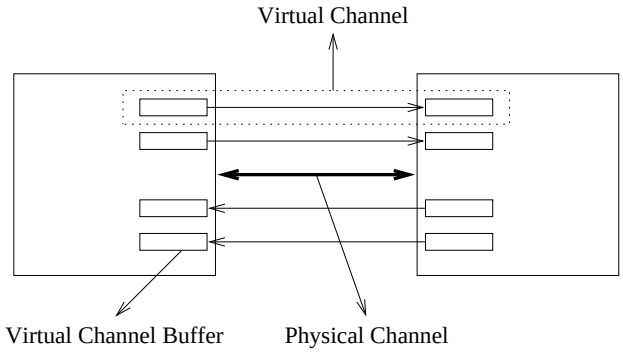


Figura 3.26: Canales virtuales.

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 3.27 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.

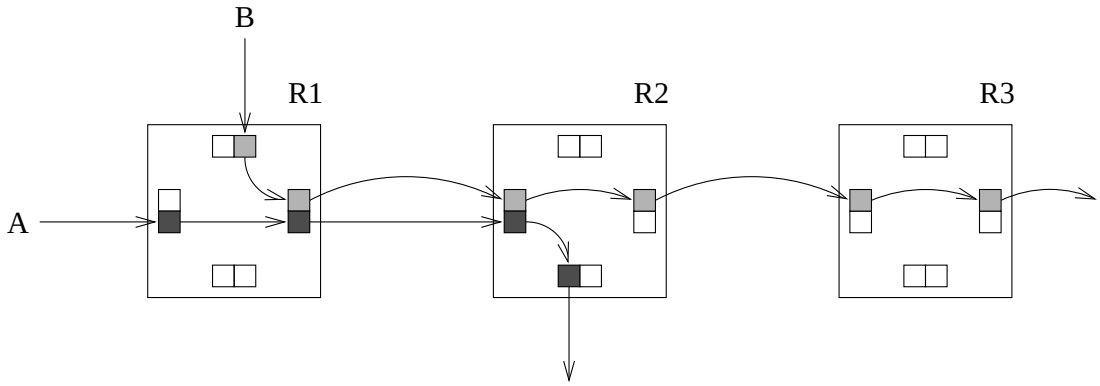


Figura 3.27: Un ejemplo de la reducción del retraso de la cabecera usando dos canales virtuales por canal físico.

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.

3.2.8 Mecanismos híbridos de conmutación

Conmutación encauzada de circuitos

Conmutación de exploración

3.2.9 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.

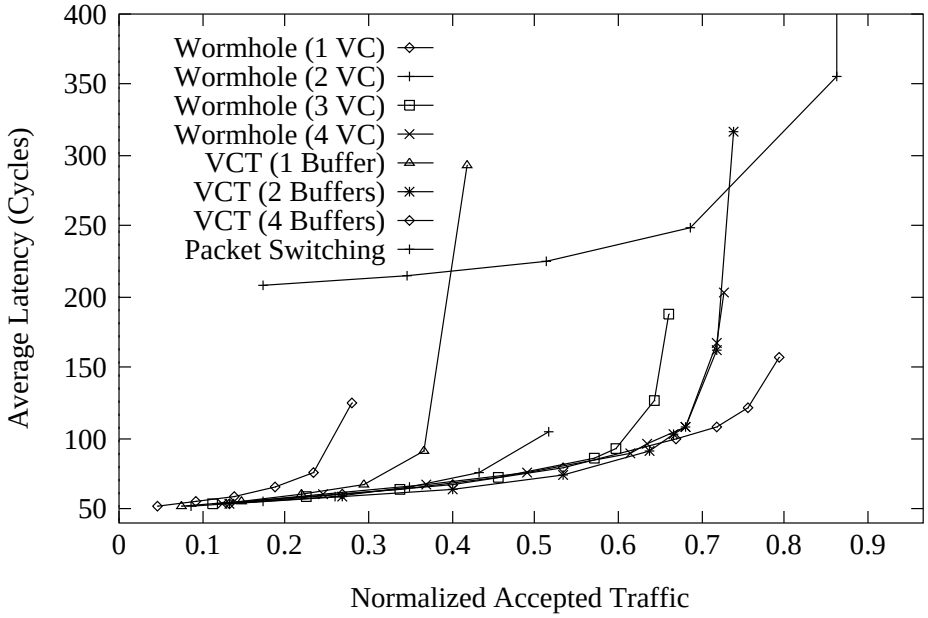


Figura 3.28: Latencia media del paquete vs. tráfico aceptado normalizado en una malla 16×16 para diferentes técnicas de conmutación y capacidades de buffer. (VC = Virtual channel; VCT = Virtual cut-through.)

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.

3.3 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.

3.3.1 Clasificación de los algoritmos de encaminamiento

La figura 3.29 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).

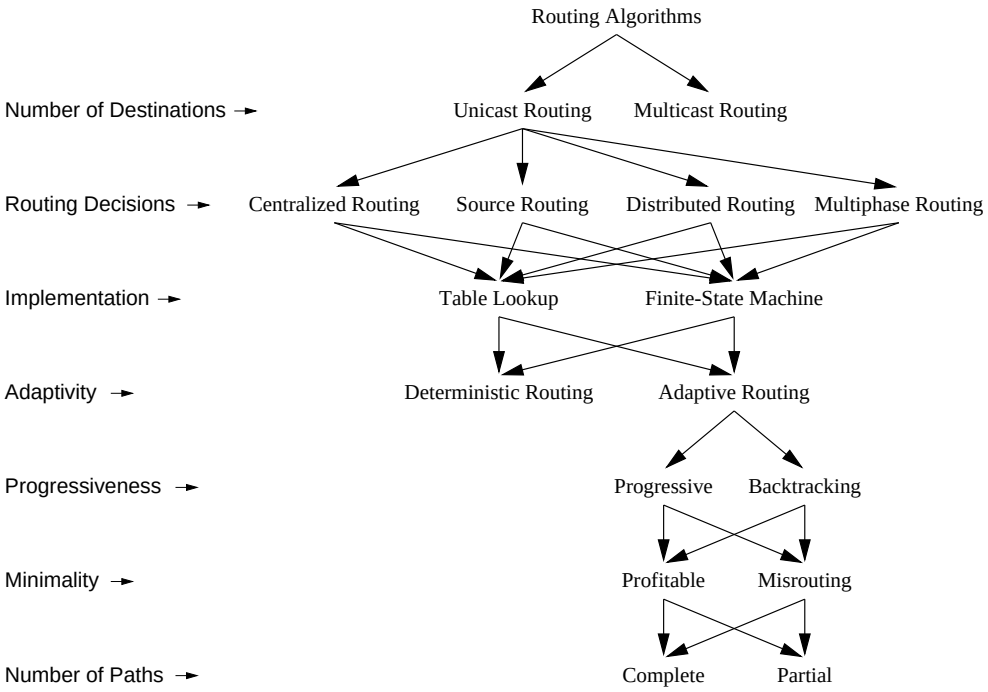


Figura 3.29: Una taxonomía de los algoritmos de encaminamiento

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*.

3.3.2 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 3.30 muestra un ejemplo de esta situación.

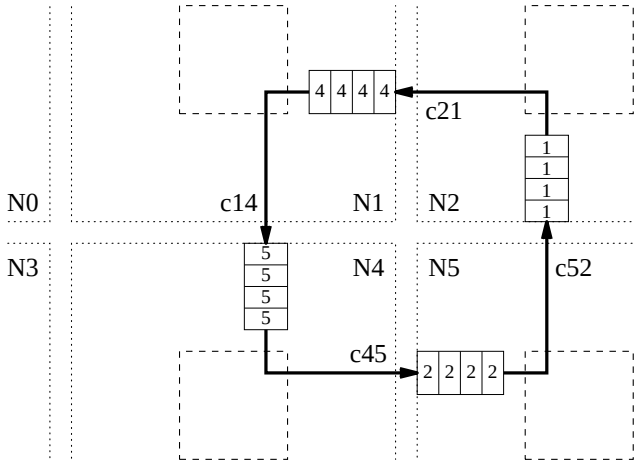


Figura 3.30: Configuración de bloqueo en una malla 2-D.

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 3.31 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

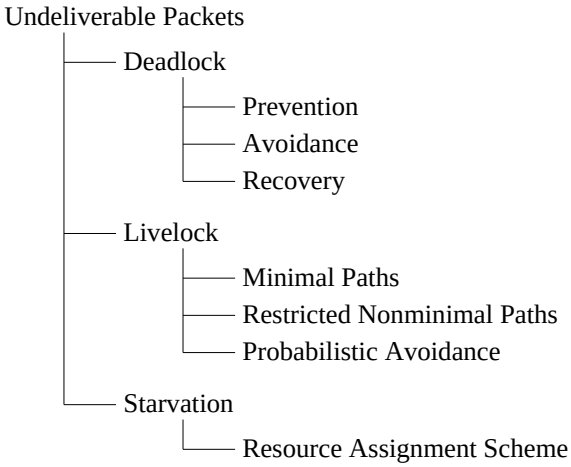


Figura 3.31: Una clasificación de las situaciones que pueden impedir el envío de paquetes.

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.

3.3.3 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 3.30 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.

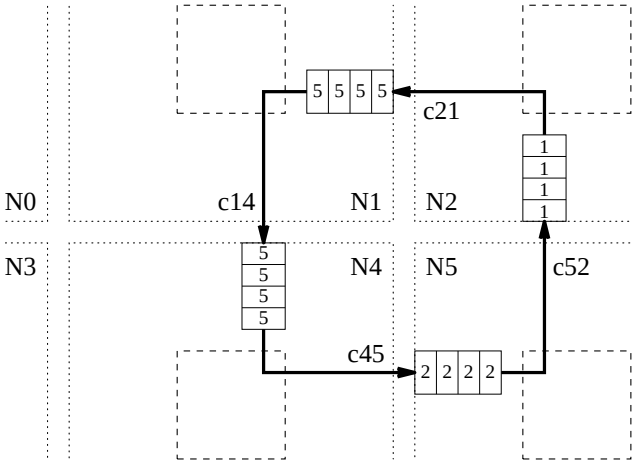


Figura 3.32: Configuración ilegal para R.

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 3.32 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 3.33(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.

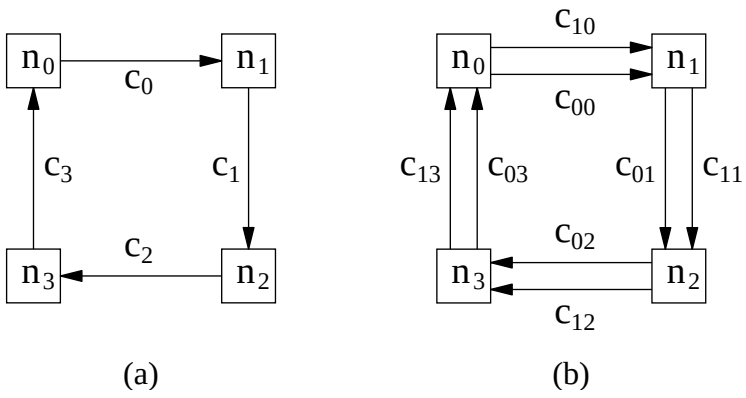


Figura 3.33: Redes del ejemplo anterior.

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 3.33(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 3.34 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}.

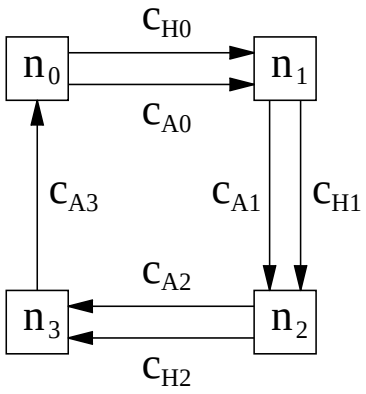


Figura 3.34: Red del ejemplo anterior.

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.

3.3.4 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 3.35 y 3.36, 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 3.37. 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 3.3.3.

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 3.35: 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 3.36: 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.

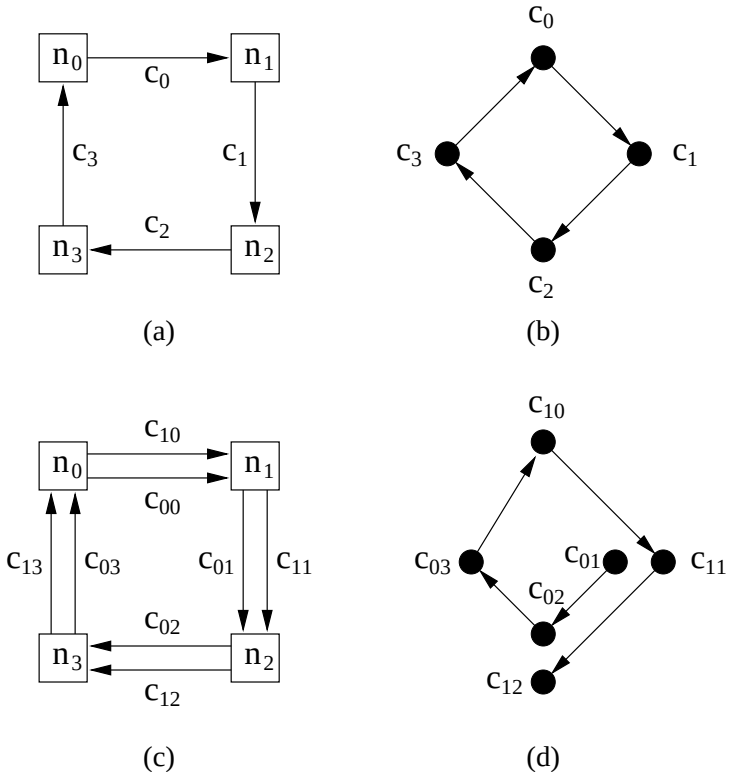


Figura 3.37: Grafo de dependencias entre canales para anillos unidireccionales.

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 3.37(a) muestra la red. La figura 3.37(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 3.37(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 3.37(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 3.38 muestra el algoritmo de encaminamiento por dimensiones para los 2-cubos k-arios (toros bidimensionales).

3.3.5 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 3.38: 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 3.3.3.

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 3.39 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 3.39(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 3.39(b) y 3.39(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.

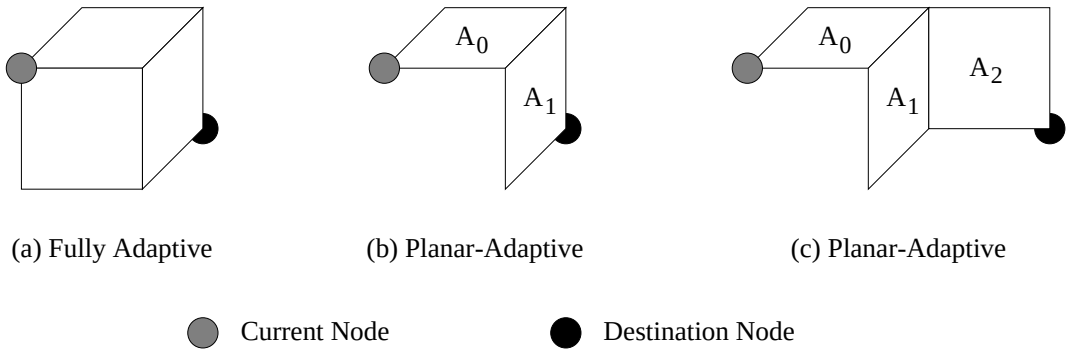


Figura 3.39: Caminos permitidos en el encaminamiento totalmente adaptativo y adaptativo por planos.

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.

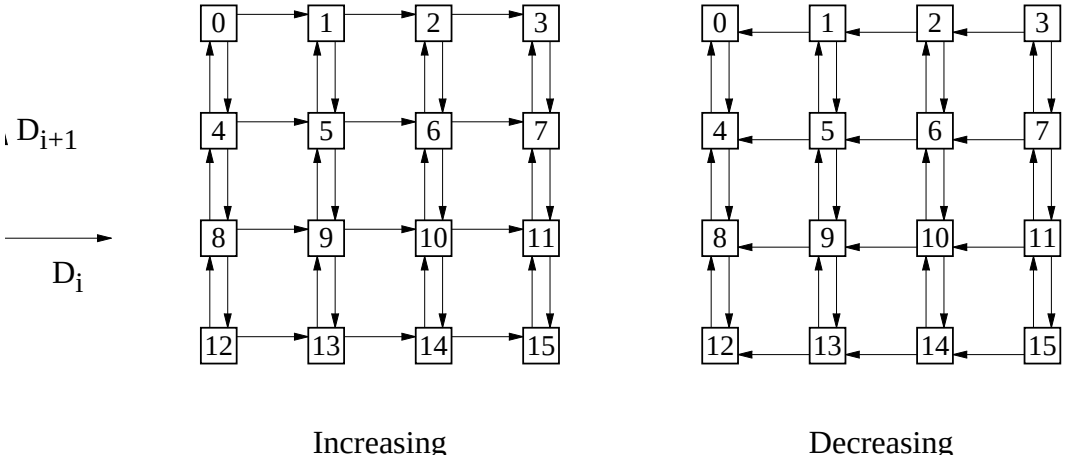


Figura 3.40: Redes crecientes y decrecientes en el plano A_i para el encaminamiento adaptativo por planos.

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 3.40). 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

3.3.6 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.

3.3.7 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

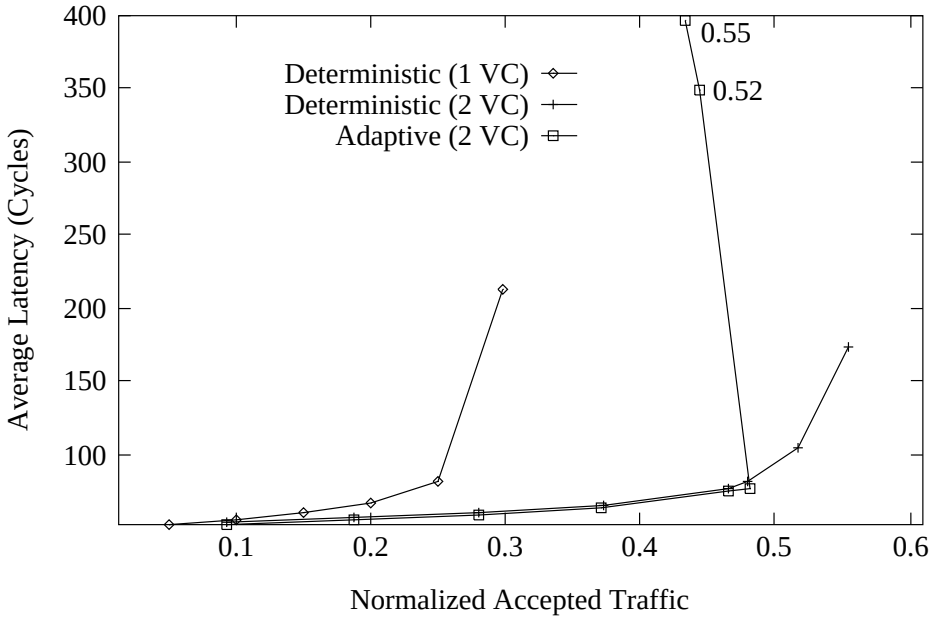


Figura 3.41: Latencia media del mensaje vs. tráfico normalizado aceptado para mallas 16×16 para una distribución uniforme del destino de los mensajes.

La figura 3.41 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.

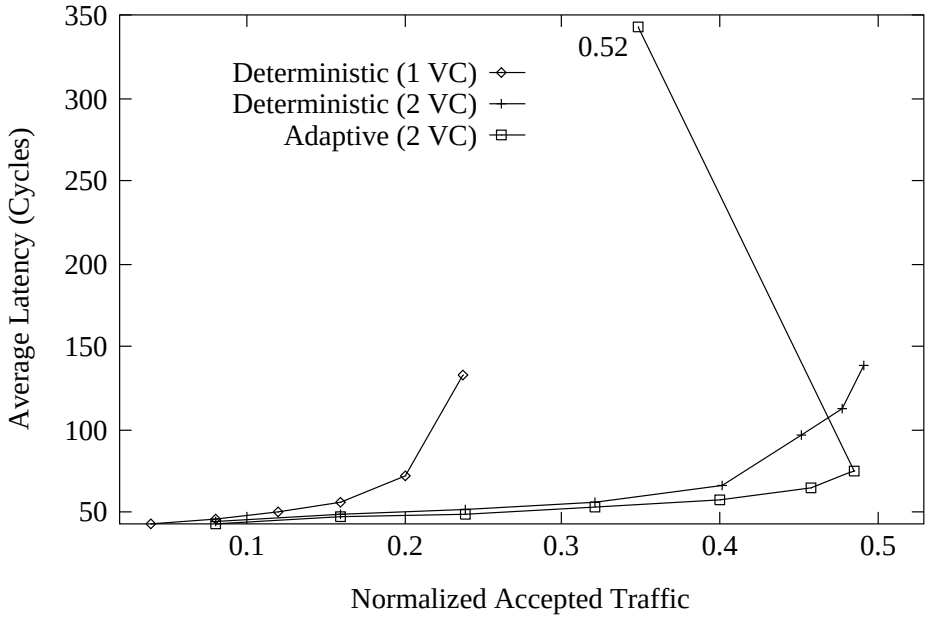


Figura 3.42: Latencia media del mensaje vs. tráfico normalizado aceptado para mallas $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes.

La figura 3.42 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 3.43 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

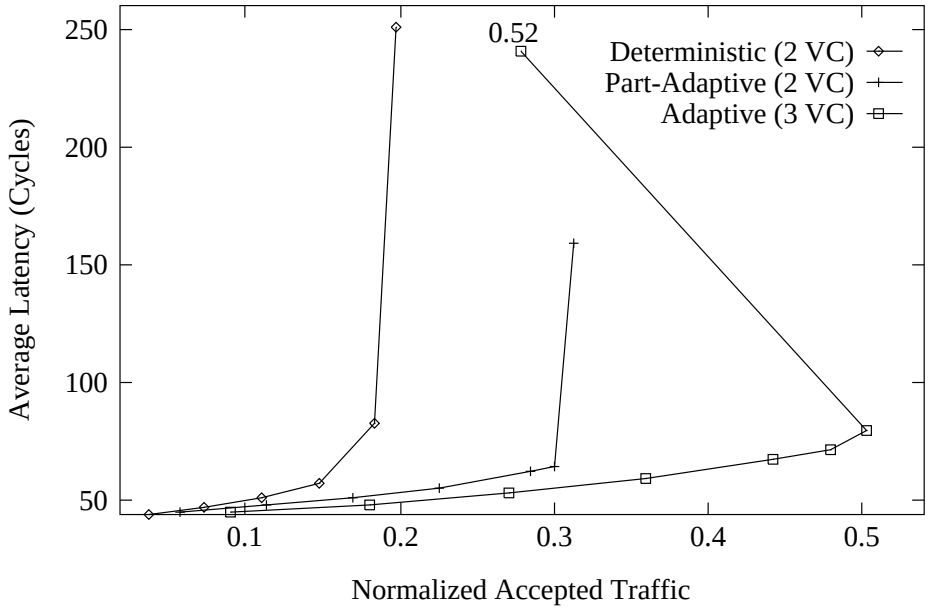


Figura 3.43: Latencia media del mensaje vs. tráfico normalizado aceptado para toros 16×16 para una distribución uniforme del destino de los mensajes.

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 3.44 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 3.43, 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 3.45 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.

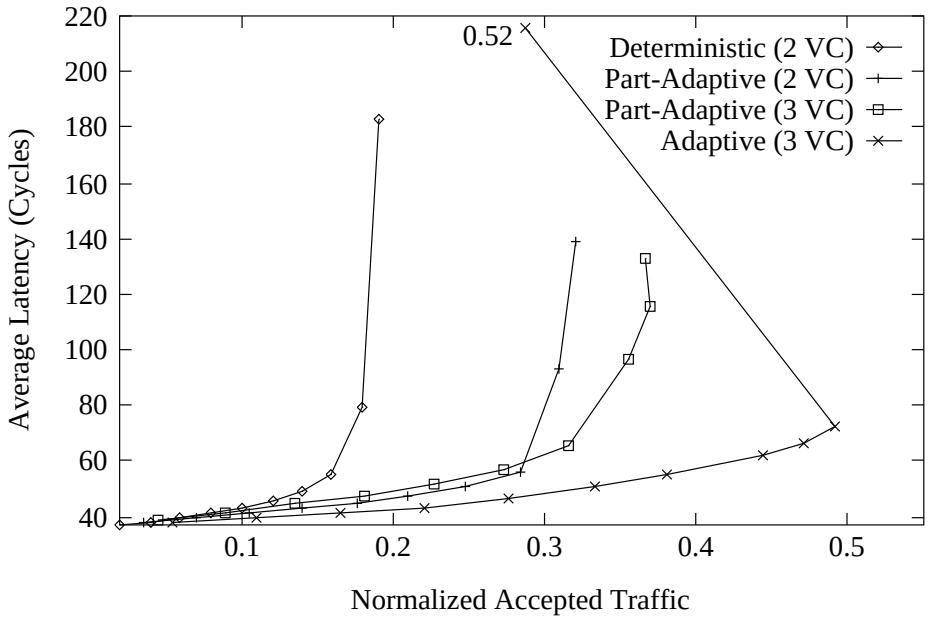


Figura 3.44: Latencia media del mensaje vs. tráfico normalizado aceptado para toros $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes.

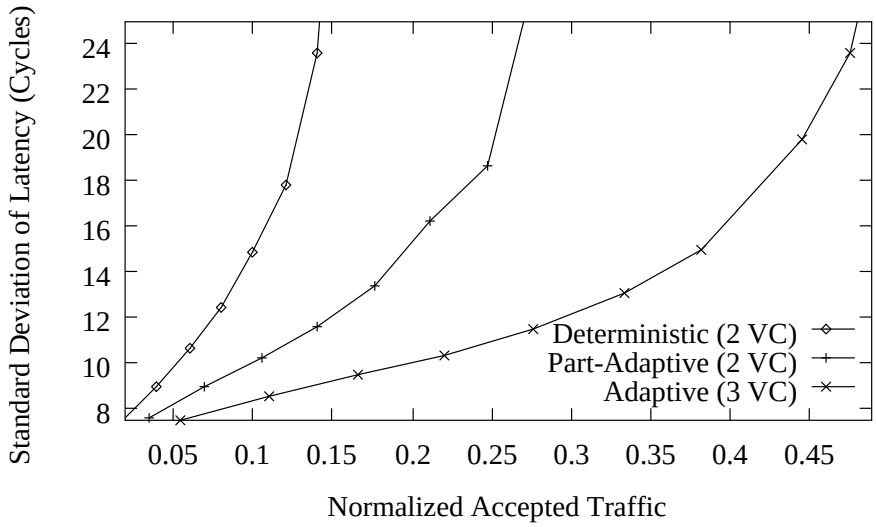


Figura 3.45: Desviación estándar de la latencia vs. tráfico normalizado aceptado en un toro $8 \times 8 \times 8$ para una distribución uniforme del destino de los mensajes

Capítulo 4

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.

4.1 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.

4.1.1 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 4.1 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;
```

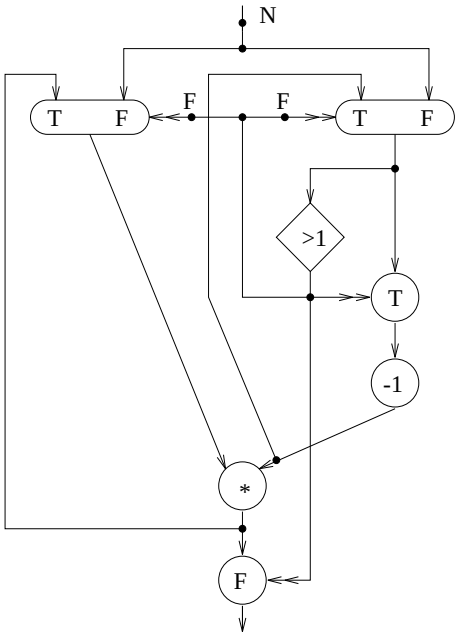


Figura 4.1: Grafo de flujo de datos para calcular $N!$

4.1.2 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 4.2. 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.

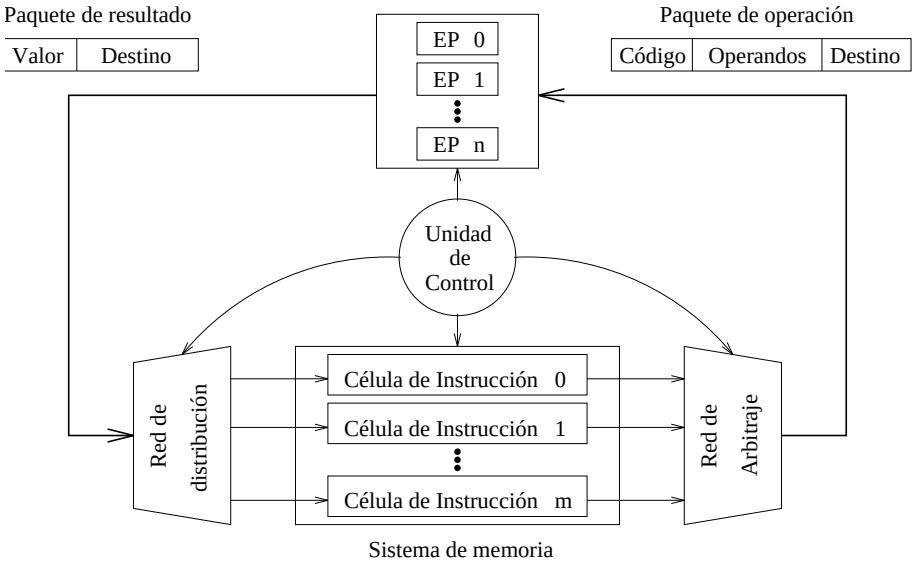


Figura 4.2: La máquina MIT de flujo de datos.

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 4.3. 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.

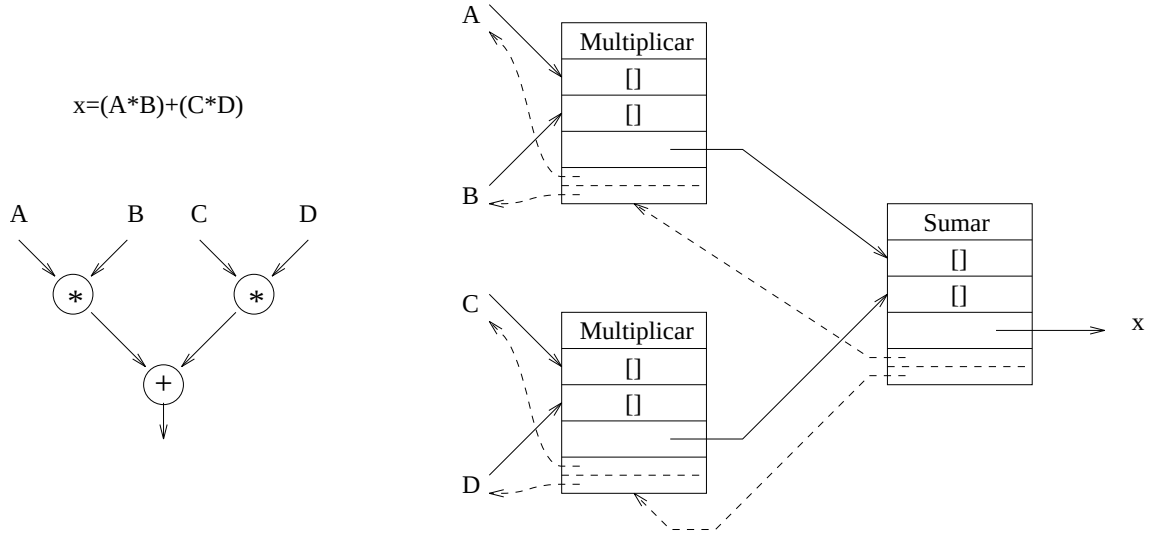


Figura 4.3: Ejemplo de máquina de flujo de datos estática (las flechas discontinuas son las señales de reconocimiento).

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.

4.2 Otras arquitecturas

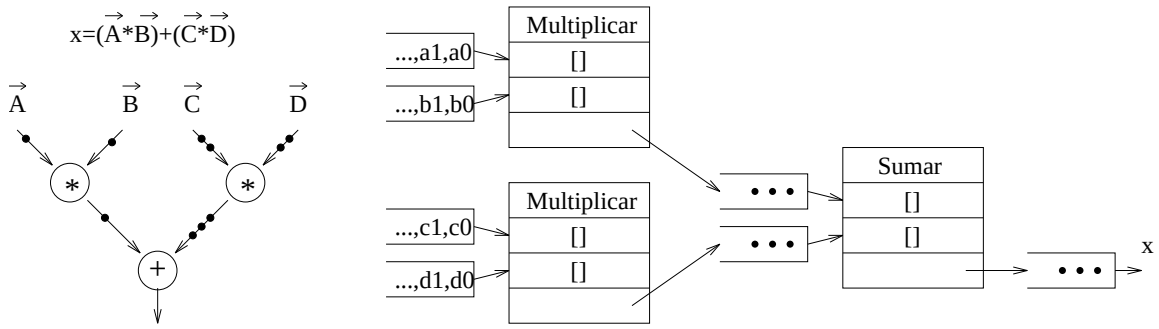


Figura 4.4: Ejemplo de máquina de flujo de datos dinámica. Un arco en grafo de flujo de datos puede llevar más de una etiqueta a un tiempo.

Apéndice A

Introducción a las arquitecturas avanzadas

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].

A.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).

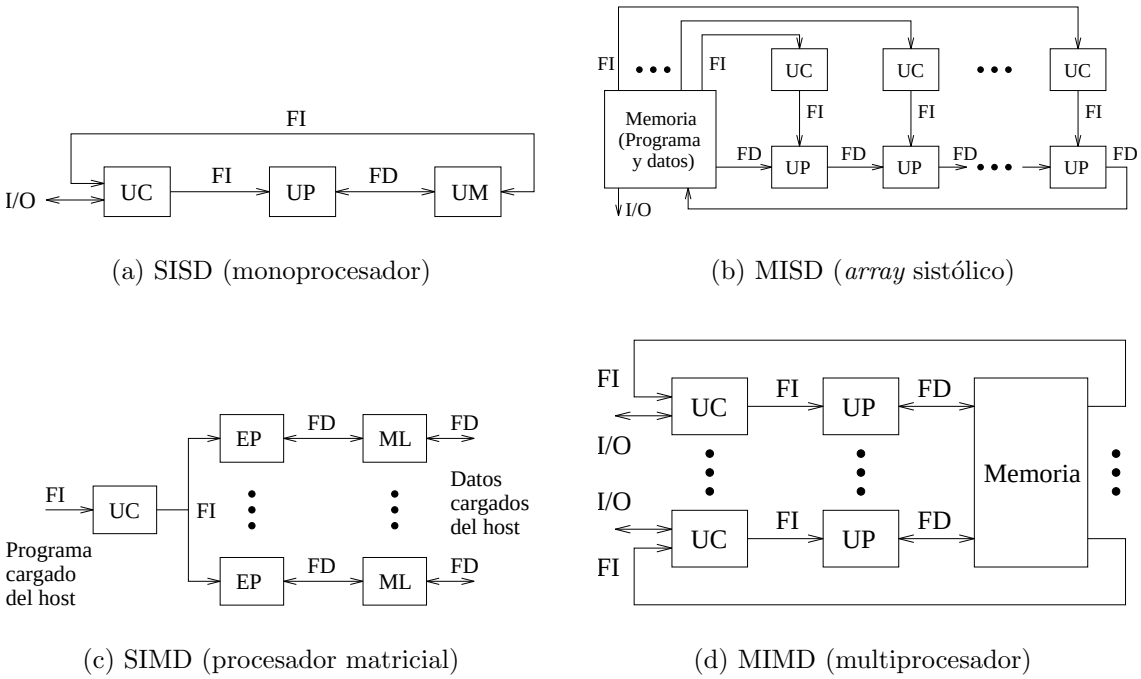
MISD (*Multiple Instruction stream, Single Data stream*) Flujo múltiple de instrucciones y único flujo de datos. Esto significa que varias instrucciones actúan sobre el mismo y único trozo de datos. Este tipo de máquinas se pueden interpretar de dos maneras. Una es considerar la clase de maquinas que 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 em-

presa, 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 A.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 A.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.)

A.2 Otras clasificaciones

La clasificación de Flynn ha demostrado funcionar bastante bien para la tipificación de sistemas ya que 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 A.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.

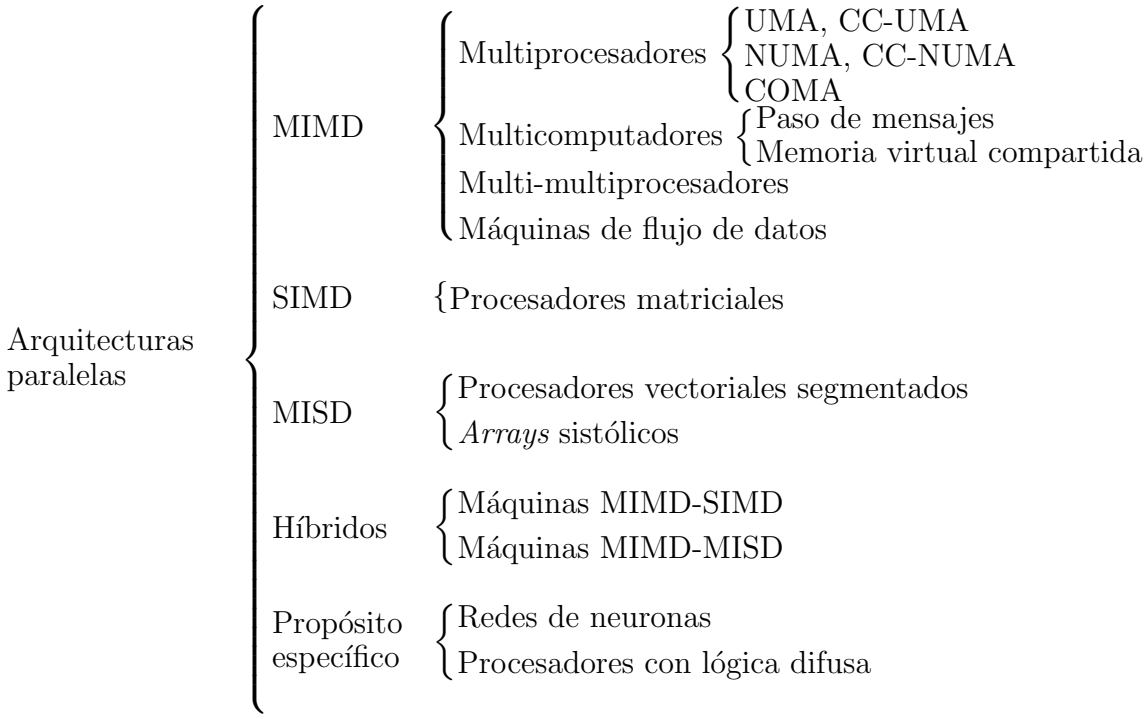


Figura A.2: Clasificación de las arquitecturas paralelas.

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 A.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 A.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*).

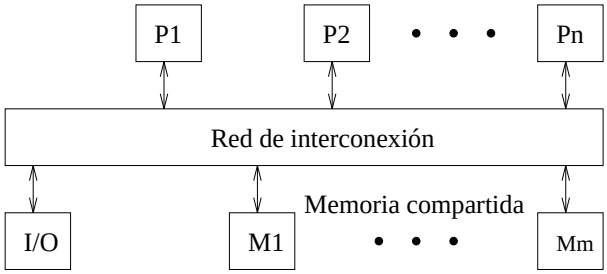


Figura A.3: El modelo UMA de multiprocesador.

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 A.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.

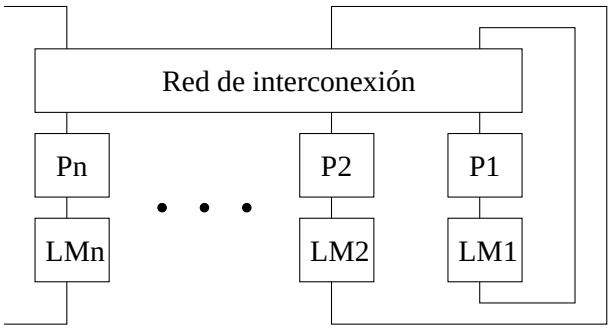


Figura A.4: El modelo NUMA de multiprocesador.

COMA (*Cache Only Memory Access*) Un multiprocesador que sólo use caché como memoria es considerado de tipo COMA. La figura A.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.

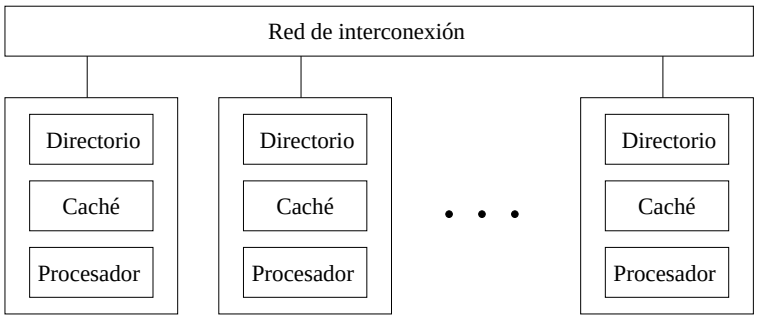


Figura A.5: El modelo COMA de multiprocesador.

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 A.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 A.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.

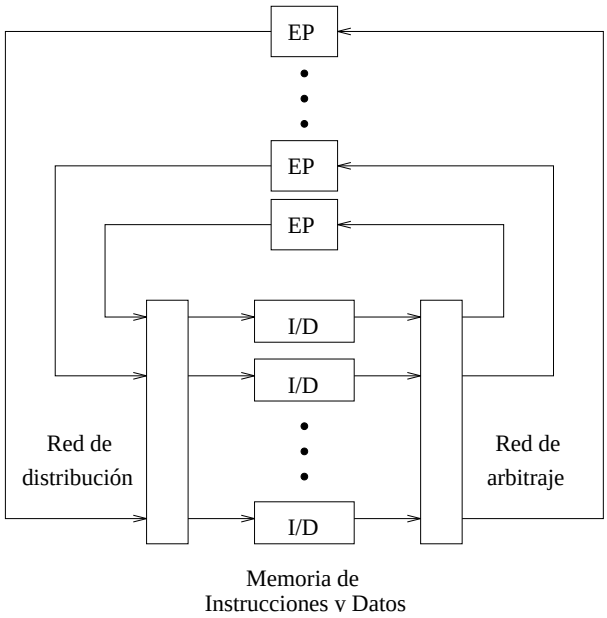


Figura A.6: Diagrama de bloques de una máquina de flujo de datos.

Procesadores matriciales

En el capítulo 2 se verá con más detalle esta arquitectura que como sabemos es la representativa del tipo SIMD, es decir, una sola instrucción y múltiples datos sobre las que opera dicha instrucción.

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

El capítulo 1 está dedicado a este tipo de procesadores. 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.

A.3 Introducción al paralelismo

A.3.1 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

A.3.2 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 A.7 muestra el concepto que subyace tras el paralelismo de control

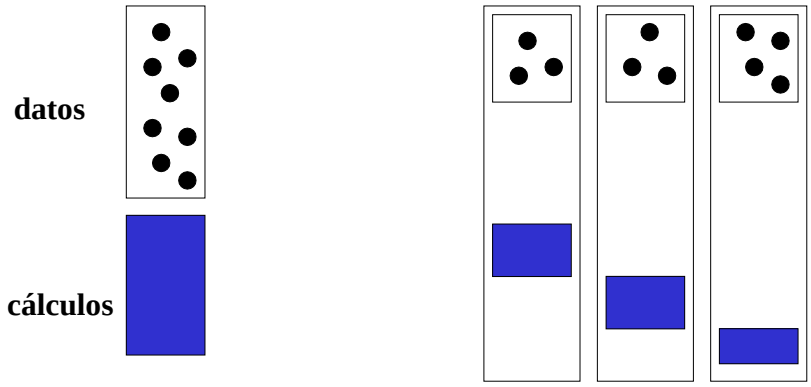


Figura A.7: Paralelismo de control.

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 A.8 muestra un ejemplo de paralelismo de control aplicado a la ejecución simultánea de instrucciones.

A.3.3 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 A.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

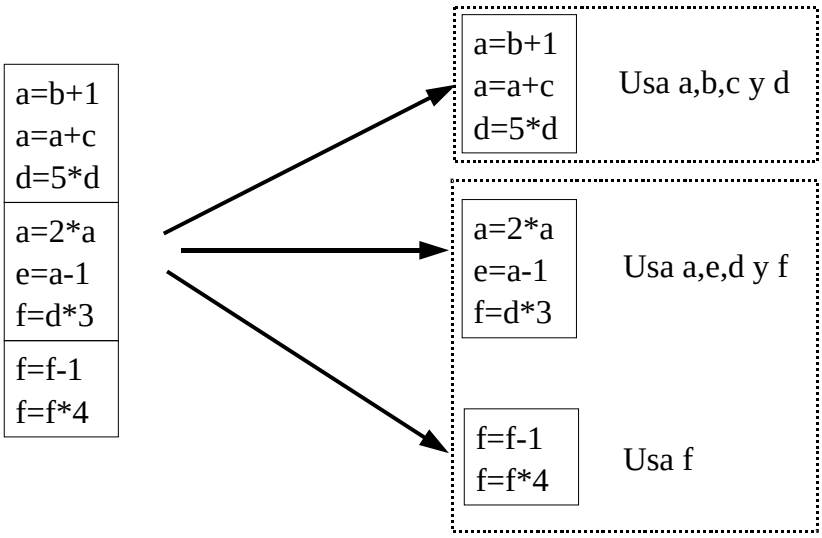


Figura A.8: Ejemplo de paralelismo de control.

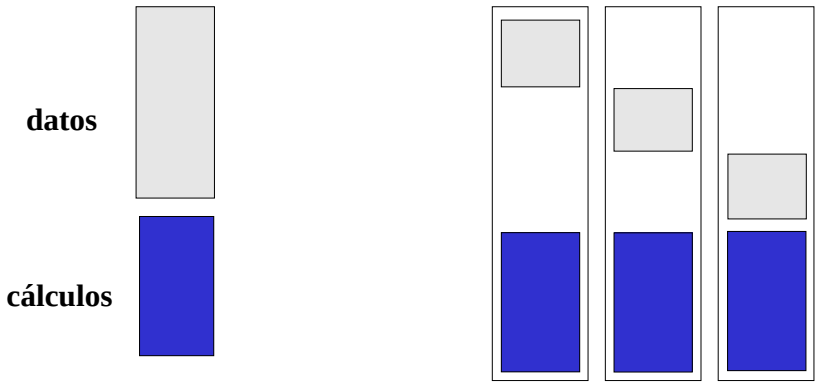


Figura A.9: Paralelismo de datos.

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 A.10 se muestra un ejemplo de paralelismo de datos.

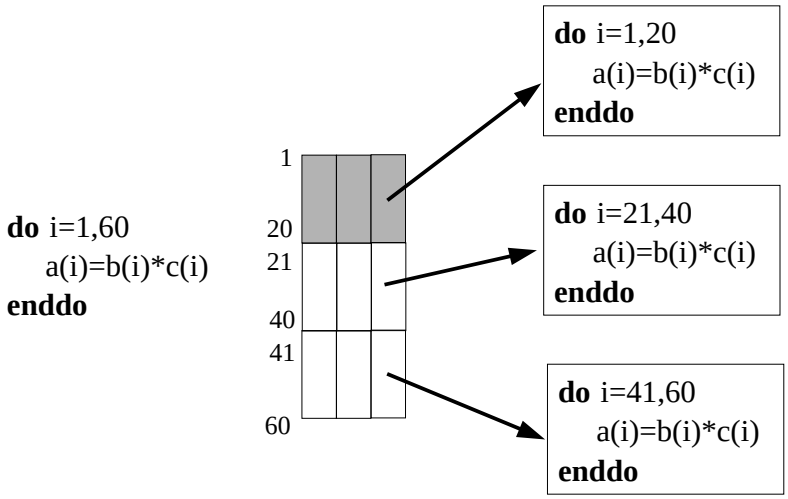


Figura A.10: Ejemplo de la aplicación del paralelismo de datos a un bucle.

A.3.4 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 A.11 muestra de forma gráfica el concepto de paralelismo de flujo.

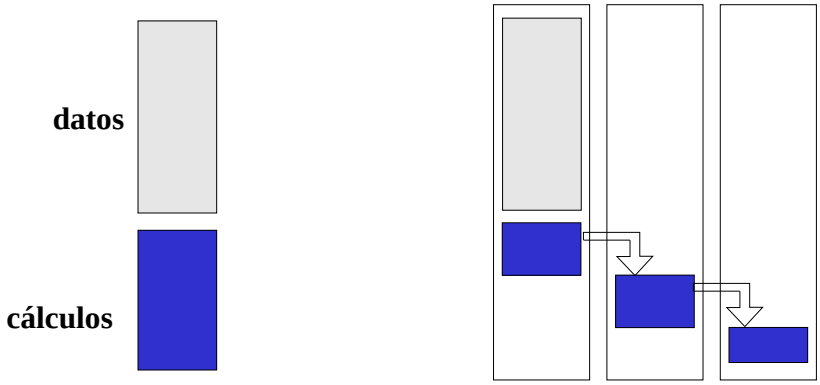


Figura A.11: Paralelismo de flujo.

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.

Apéndice B

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

Árbol, 48
    grueso, 48

Acceso C, 13
Acceso C/S, 15
Acceso S, 14
Actores en grafos de flujo, 94
ADA, 94, 106
Agujero de gusano, 50
Amdahl
    ley de, 23, 27
Anillo, 46
Anillos acordes, 46
Arrays sistólicos, 100, 107

Barrel shifter, *véase* Desplazador de barril
Biseeción, 50
Bloqueo activo, 73
Bloqueo mortal, 73
Bloqueo por inanición, 73
Bloqueos, 72
    Evitación, 75
Bloqueos en Toros, 82

Caché, 11, 104
Cache vectorial, 15
Campanadas, 8, 19
Canales virtuales, 66
CC-NUMA, 104
CC-UMA, 103
Ciclo Cubo Conectado, 47
Ciclo mayor de memoria, 13
Ciclo menor de memoria, 13
COMA, 102, 104
Conflicto
    del banco de memoria, 22
Conmutación, 52
    Mecanismos híbridos, 68
    Paso a través, 59
Conmutación cartero loco, 63
Conmutación de circuitos, 56
Conmutación de exploración, 68
Conmutación de Lombriz, 61
Conmutación de paquetes, 58
Conmutación encauzada de circuitos, 68
Contador de programa, 100
Control de flujo, 52
Control de máscara vectorial, 24
Controlador de enlace, 55
Convoy, 8
Copia de actividad, 95

DAXPY, 6
Desplazador de barril, 47
Dispersar-agrupar, 25
Duato, algoritmo, 87

Encadenamiento
    de operaciones vectoriales, 23
Encaminadores, 52
    Modelo, 54
Encaminamiento, 70
    Clasificación, 71
    Determinista, 81
Encaminamiento adaptativo por planos, 85
Encaminamiento completamente adaptativo, 87
Encaminamiento parcialmente adaptativo, 84
Entrelazado de orden alto, 12
Entrelazado de orden bajo, 11
Estrella, 48

Flit, 52
Flujo de datos
    arquitectura, 94–97, 105
    estática, 96
Flynn
    clasificación, 100
    cuello de botella de, 2
Fortran, 18, 21
Función de encaminamiento, 75
Función de selección, 75

Grado de entrelazado, 13
Grafo de dependencias, 80
Grafo de flujo de datos, 94
Granularidad, 40

Hipercubos, 43, 44, 81

Illiac, 43

Lógica difusa, 108
Ley
de Amdahl, 23, 27
Livelock, 73
Longitud vectorial máxima, 18

Mad-Postman switching, 63
Mallas, 43
Matrices dispersas, 25–26
Matrices sistólicas, 50
Matriz lineal, 45
Memoria compartida, 35, 102, 103
Memoria de coincidencias, 97
Memoria distribuida, 35, 104, 105
Memoria entrelazada, 11–17
Memoria-memoria
máquina vectorial, 2
MIMD, 101
MISD, 100
MIT
máquina de flujo de datos, 96
Modelo de giro, 87
MPP, 50
Multicomputadores, 104
con memoria virtual compartida, 105
Multiprocesadores, 102
simétricos/asimétricos, 103
MVL, 18

Neumann, 40
NUMA, 102, 103

Orden de dimensión, 81

Paquetes, 52
Formato, 58
Paso de mensajes, 39, 105
Phit, 52
Procesadores
adheridos, 103
matriciales, 107
Procesadores matriciales, 101
Procesadores vectoriales, 1–32, 100, 107
tipos, 2
Prolog, 94, 106

Red
completamente conectada, 46
Red de arbitraje, 106
Redes, 41
Clasificación, 41
Estrictamente ortogonales, 41
Redes crecientes, 87
Redes decrecientes, 87
Redes deterministas y adaptativas, encaminamiento, 87
Redes virtuales, encaminamiento, 87
Registro
de encaminamiento, 36
de longitud vectorial, 18
Registro de máscara vectorial, 24
Registros
máquina vectorial con, 2
Registros vectoriales, 3
Routers, 52
Routing, 70

Salto negativo, 87
SAXPY, 6
Scatter-gather, *véase* Dispersar-agrupar
Seccionamiento, 18
Separación de elementos, 21
SIMD, 100, 107
SISD, 100
Sistemas
débilmente acoplados, 105
fuertemente acoplados, 103
Sistemas distribuidos, 105
Spice, 24
Stride, *véase* Separación de elementos
Strip mining, *véase* Seccionamiento

Tasa de inicialización, 10–11
Tiempo de arranque vectorial, 9–10
Token ring, 46
Toros, 43, 44
Transputer, 50

UMA, 102, 103
Unidades de carga/almacenamiento, 10–11

Vector
longitud, 18–20
Velocidad de inicialización, 8
Virtual Cut Through, 59
VLR, 18
VLSI, 40, 108
VLSI, arquitecturas, 93
Von Neumann, 99

Wormhole routing, *véase* Agujero de gusano
Wormhole switching, 61