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.

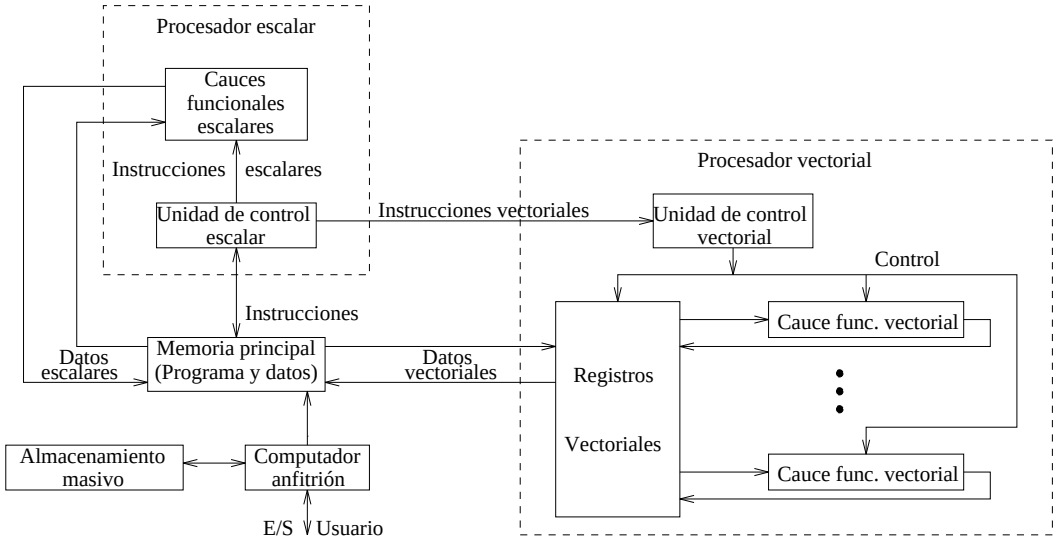


Figura 1.1: La arquitectura de un supercomputador vectorial.

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.

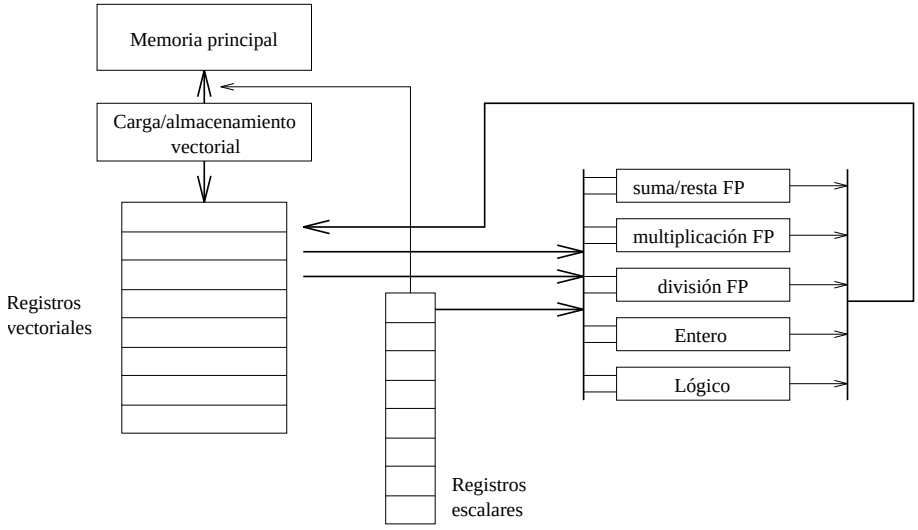


Figura 1.2: Estructura básica de una arquitectura vectorial con registros, DLXV.

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.

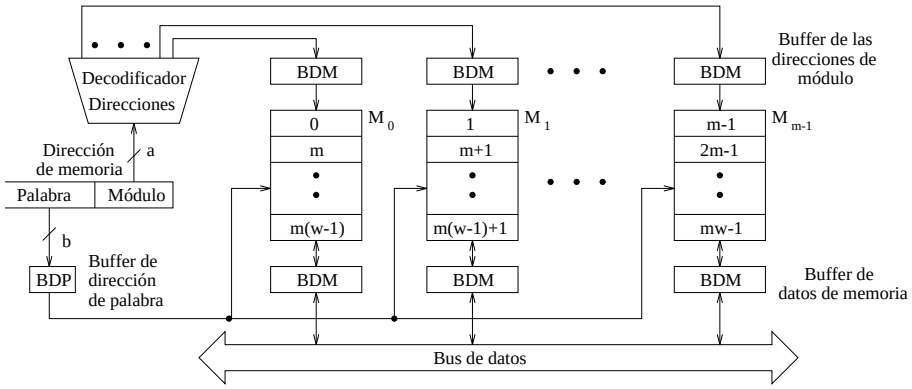
Consideremos una memoria principal formada por $m = 2^a$ módulos, cada uno con $w = 2^b$ palabras o celdas de memoria. La capacidad total de la memoria es $m\,w = 2^{a+b}$ palabras. A estas palabras se les asignan direcciones de forma lineal. Las diferentes formas en las que se asignan linealmente las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

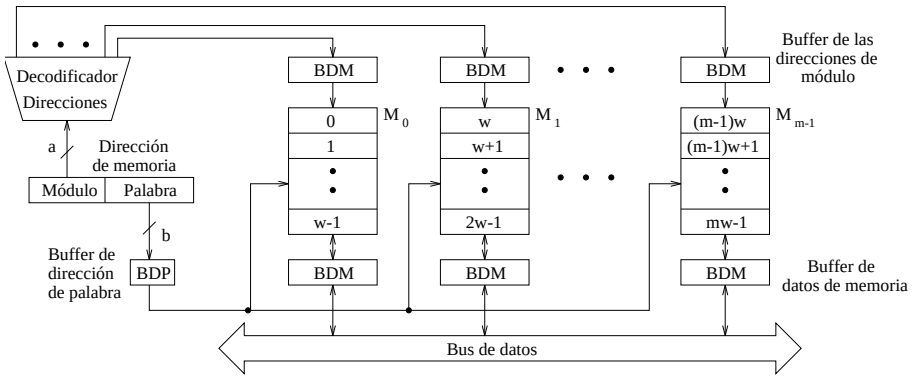
La figura 1.7 muestra dos formatos de direcciones para realizar una memoria entrelazada. El *entrelazado de orden bajo* (figura 1.7(a)) reparte las localizaciones contiguas de memoria entre los m módulos de forma horizontal. Esto implica que los a bits de orden bajo de la dirección se utilizan para identificar 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).

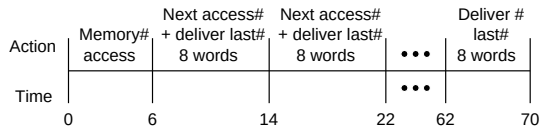


Figura 1.8: Tiempo de acceso para las primeras 64 palabras de doble precisión en una lectura.

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.

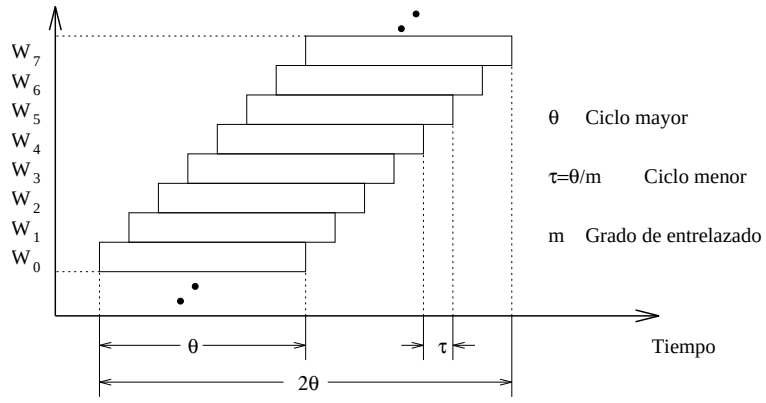


Figura 1.9: Acceso segmentado a 8 palabras contiguas en una memoria de acceso C.

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.

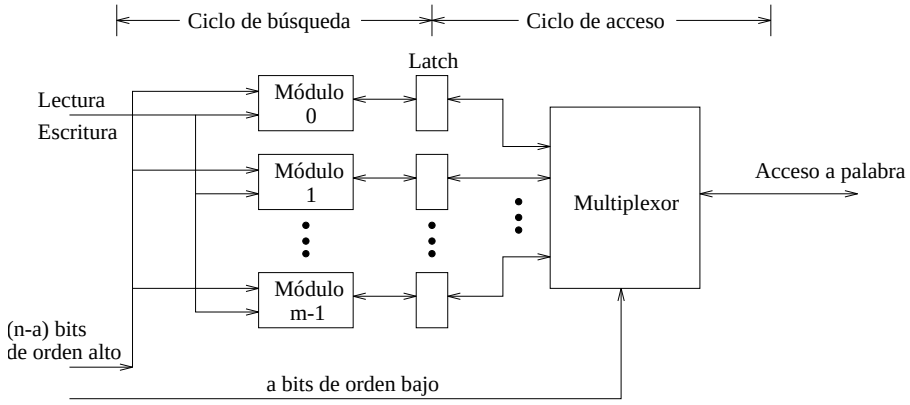


Figura 1.10: Organización de acceso S para una memoria entrelazada de m vías.

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.

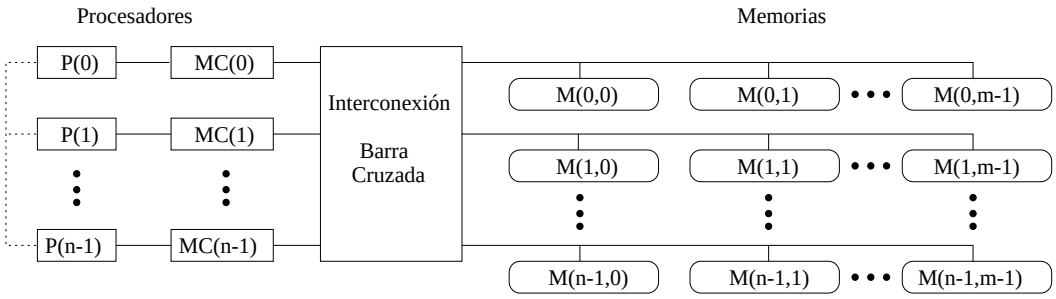


Figura 1.11: Organización de acceso C/S.

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.

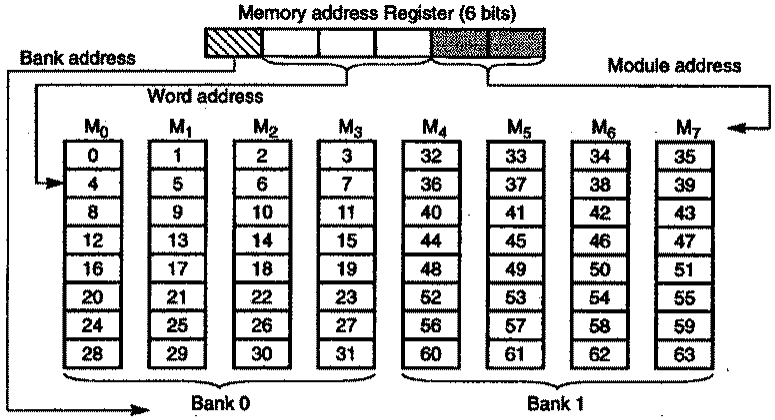
Tolerancia a fallos

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.

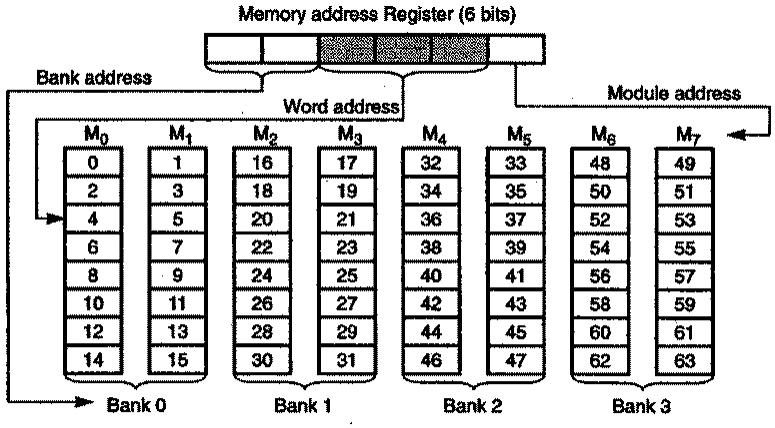
En muchas ocasiones interesa tener ambas características a un tiempo, es decir, por un lado interesa tener memoria entrelazada de orden bajo para acelerar el acceso a la memoria, pero por otro interesa también una memoria entrelazada de orden alto para tener la memoria dividida en bloques y poder seguir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

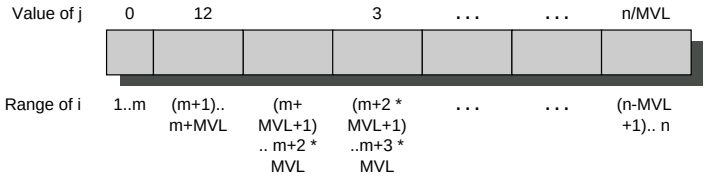


Figura 1.13: Un vector de longitud arbitraria procesado mediante seccionamiento. Todos los bloques menos el primero son de longitud MVL. En esta figura, la variable m se usa en lugar de la expresión ($n \mod MVL$).

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.

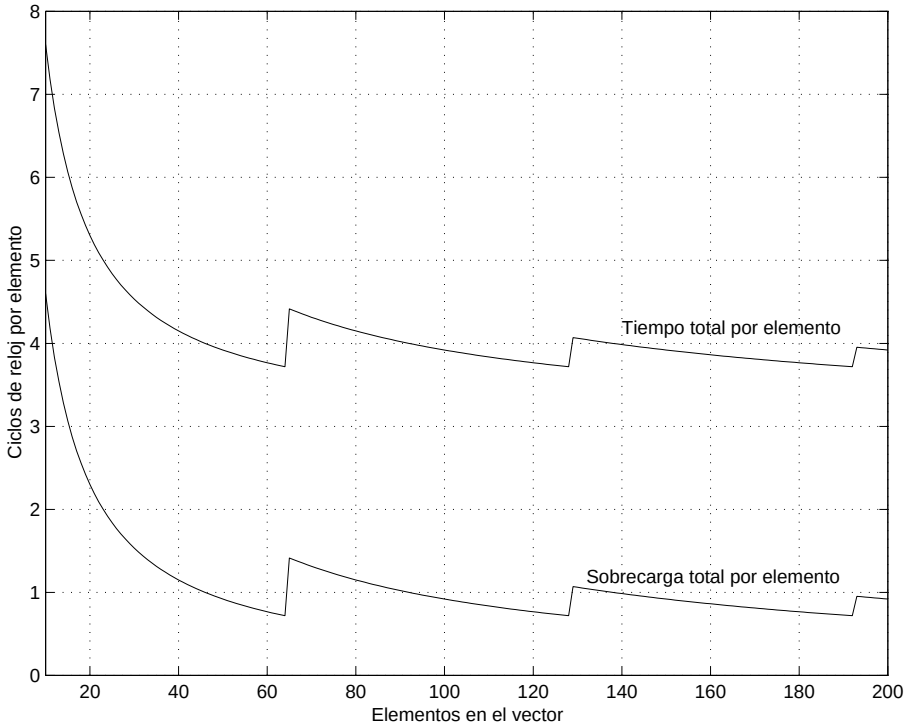


Figura 1.14: Tiempo de ejecución por elemento en función de la longitud del vector.

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

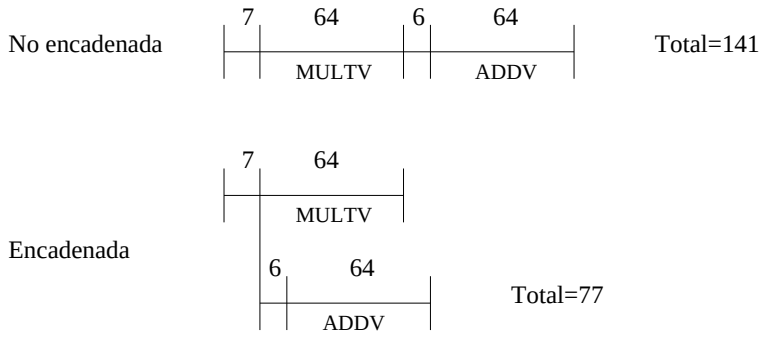


Figura 1.15: Temporización para la ejecución no encadenada y encadenada.

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.

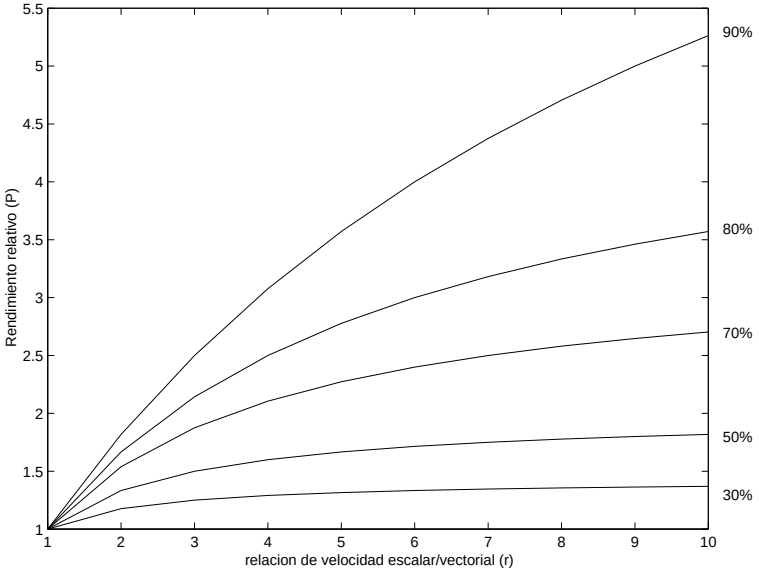


Figura 1.16: Rendimiento relativo escalar/vectorial.

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.

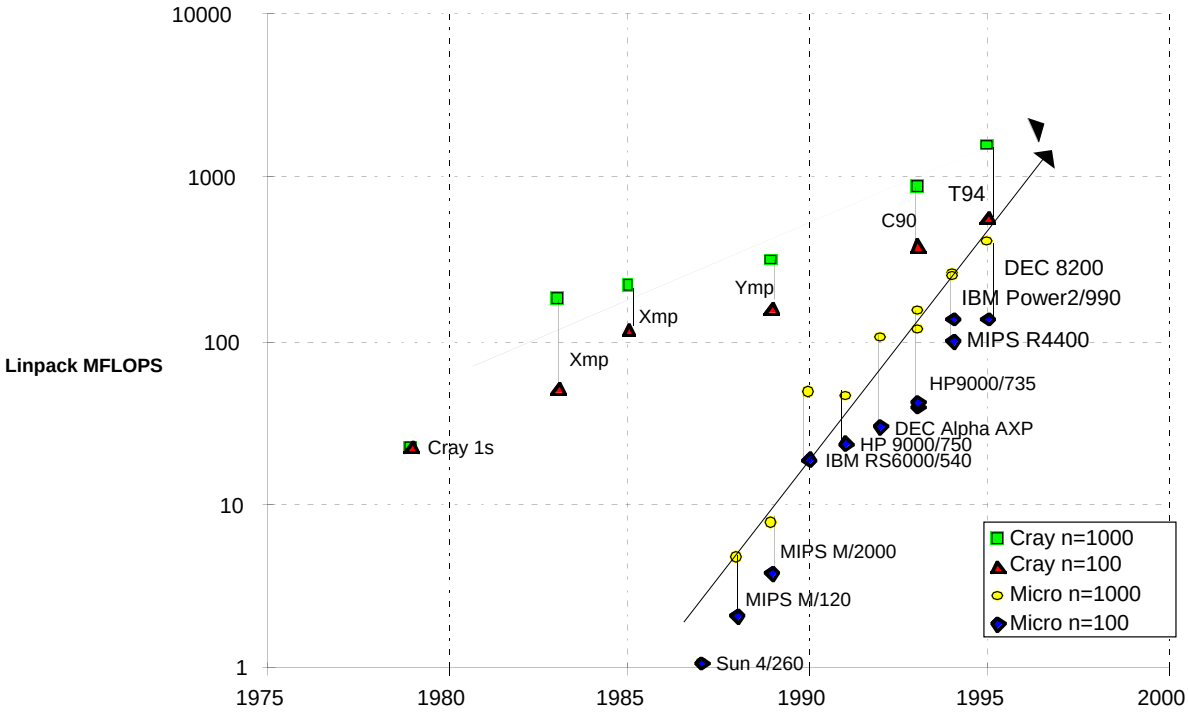


Figura 1.18: Comparación del rendimiento de los procesadores vectoriales y los microprocesadores escalares para la resolución de un sistema de ecuaciones lineales denso (tamaño de la matriz=$n \times n$).