TEMA 5. COMPUTADORES VECTORIALES

5.1 Introducción.

5.2 Arquitectura vectorial básica.

5.3 Velocidad de iniciación y tiempo de arranque vectorial.

5.4 Longitud del vector y separación entre elementos.

5.5 Modelo para el cálculo del rendimiento vectorial.

5.6 Compiladores para máquinas vectoriales.

5.7 Eficacia de las técnicas de vectorización.

5.8 Comparación de las máquinas vectoriales con las máquinas escalares.

5.1 INTRODUCCIÓN

Existen límites en la mejora del rendimiento que la segmentación puede conseguir. Estos límites están impuestos por dos factores principales:

- Duración del ciclo de reloj ⇒ La duración del ciclo de reloj se puede hacer más pequeña aumentando el número de etapas o segmentos, pero hará aumentar el número de dependencias, y con ello el valor CPI.
- Velocidad en la búsqueda y decodificación de las instrucciones ⇒ Está limitada. Imposibilita la búsqueda y emisión de más de unas pocas instrucciones por ciclo de reloj ⇒ *Cuello de botella* (Flynn, 1966).

Aunque los procesadores segmentados de alta velocidad son particularmente útiles para grandes aplicaciones científicas y de ingeniería, para los programas científicos grandes, cuya ejecución es larga, con conjuntos de datos activos muy grandes y con baja localidad, dejan de ser eficientes en su jerarquía de memoria (y hay que tener en cuenta que la existencia de la caché en estos procesadores segmentados resulta necesaria para no permitir que las instrucciones con referencia a memoria tengan una latencia muy alta). El problema de la poca eficiencia de las jerarquías de memoria por la falta de localidad, podría superarse si se dejara de utilizar la caché y fuese posible determinar los patrones de acceso a memoria para segmentarlos eficientemente.

Los *computadores vectoriales* proporcionan operaciones de alto nivel que trabajan sobre vectores (arrays lineales de números) ⇒ La instrucción vectorial es equivalente a un bucle completo.

PROPIEDADES IMPORTANTES DE LAS OPERACIONES VECTORIALES (que resuelven la mayoría de los problemas mencionados anteriormente):

- El cálculo de cada resultado es independiente de los cálculos de los resultados anteriores, permitiendo un gran nivel de segmentación sin generar ningún riesgo por dependencia de datos ⇒ Los riesgos por dependencias de datos los ha resuelto el compilador o el programador que ha decidido que sea una operación vectorial.
- Una simple instrucción vectorial especifica una gran cantidad de trabajo (equivalente a ejecutar un bucle completo) ⇒ El requerimiento de anchura de

banda de las instrucciones es reducido y el cuello de botella de Flynn se reduce considerablemente.

- Las instrucciones vectoriales que acceden a memoria tienen un patrón de acceso conocido ⇒ Si los elementos del vector son todos adyacentes, extraer el vector de un conjunto de bancos de memoria entrelazados funciona muy bien. Además se obtiene un alto rendimiento en la jerarquía de memoria.
- Como se sustituye un bucle completo por una instrucción vectorial cuyo comportamiento está predeterminado, los riesgos de control que normalmente podían surgir del salto del bucle son inexistentes.

Por todas las razones anteriores, las operaciones vectoriales pueden hacerse más rápidas que una secuencia de operaciones escalares sobre el mismo número de elementos de datos.

Los *procesadores vectoriales* segmentan las operaciones (no sólo las aritméticas, sino también los accesos a memoria y el cálculo de las direcciones efectivas) sobre los elementos de un vector; permitiendo además la mayoría de ellos que se hagan múltiples operaciones vectoriales a la vez, creando paralelismo entre las operaciones sobre diferentes elementos (en este caso, y sólo en este caso, se consideran computadores MISD).

5.2 ARQUITECTURA VECTORIAL BÁSICA

Un procesador vectorial consta típicamente de:

- **Una unidad escalar segmentada**.
- **Una unidad vectorial** ⇒ Todas sus unidades funcionales tienen una latencia de varios ciclos de reloj; lo que permite un ciclo de reloj de menor duración, siendo compatible con operaciones vectoriales de larga ejecución que pueden ser segmentadas a nivel alto sin generar riesgos.

La mayoría de las máquinas vectoriales permiten que los vectores sean tratados como números en punto flotante (FP), como enteros, o como datos lógicos.

Hay dos tipos principales de arquitecturas vectoriales:

- ***Máquina vectorial con registros*** ⇒ Responden al modelo de ejecución *registro – registro* (excepto en las operaciones de carga y almacenamiento). Son el equivalente vectorial de una *arquitectura escalar de carga /*

almacenamiento ⇒ Son las que han tenido mayor éxito, por necesitar menor ancho de banda que las de memoria – memoria. De este tipo son: las máquinas de Cray Research (CRAY-1, CRAY-2, X-MP e Y-MP), los supercomputadores japoneses (NEC SX/2, Fujitsu VP200 y el Hitachi S820) y los minisupercomputadores (Convex C-1 y C-2).

- ***Máquina vectorial memoria – memoria*** ⇒ Responden al modelo de ejecución *memoria – memoria.* En estas máquinas todas las operaciones vectoriales son de memoria a memoria ⇒ Aunque las primeras máquinas vectoriales fueron de este tipo [las máquinas STAR-100 de CDC y ASC de TI (1972)], no han tenido éxito por tener demasiado gasto de tiempo en el arranque.

MÁQUINA VECTORIAL CON REGISTROS

En la **Figura 5.1** se muestra el esquema de una máquina vectorial con registros, está basada aproximadamente en el CRAY-1, la llamaremos DLXV (con parte entera DLX).

Los componentes principales de la máquina DLXV son:

- **Registros vectoriales** ⇒ Bancos de longitud fija que contienen un sólo vector cada uno de ellos ⇒ DLXV tiene 8 registros vectoriales de 64 dobles palabras cada uno de ellos.
- **Unidades funcionales vectoriales** ⇒ Cada unidad vectorial está completamente segmentada y puede comenzar una operación nueva en cada ciclo de reloj. Se necesita una unidad de control para detectar riesgos, sobre conflictos en las unidades funcionales (riesgos estructurales) y sobre conflictos en los accesos a memoria (riesgos por dependencias de datos) ⇒ DLXV tiene cinco unidades funcionales.
- **Unidad de Carga/Almacenamiento de vectores** ⇒ Es una unidad que carga/almacena un vector desde/en memoria ⇒ En DLXV las cargas y almacenamientos 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, después de una latencia inicial.
- **Conjunto de registros escalares** ⇒ Pueden proporcionar datos como entradas a las unidades funcionales vectoriales, así como calcular direcciones para pasar a la unidad de Carga/Almacenamiento de vectores ⇒ Son los 32

registros de propósito general (GPR) y los 32 registros de punto flotante (FPR) de DLX.

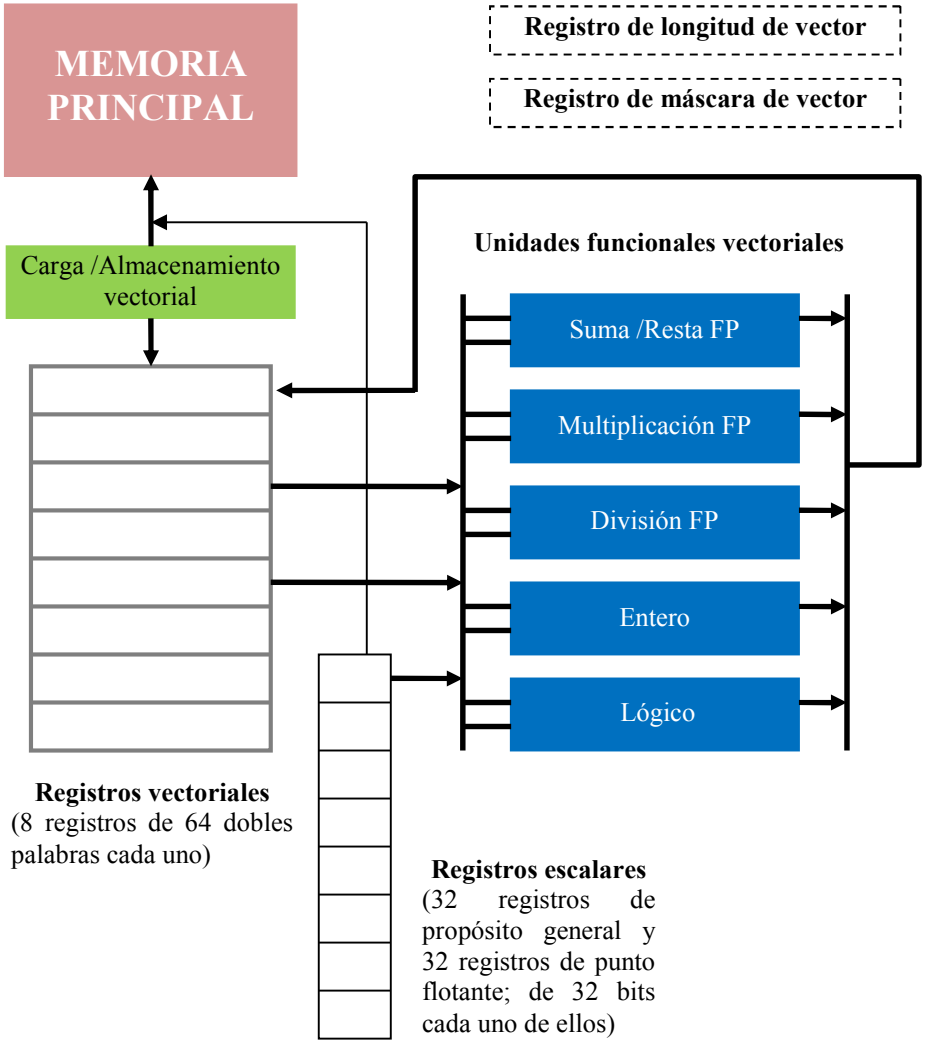

Figura 5.1. Estructura básica de una arquitectura con registros vectoriales, DLXV.

Las operaciones vectoriales en DLXV tienen el mismo nombre que en DLX añadiéndole la letra "V". Estas son operaciones vectoriales de punto flotante y doble precisión (**ADDV** es una suma de dos vectores en doble precisión). Las operaciones vectoriales toman como entradas o un par de registros vectoriales (**ADDV**) o un registro vectorial y un registro escalar (**ADDSV**). Las operaciones vectoriales siempre tienen como destino un registro vectorial. Las instrucciones **LV** y **SV** denotan respectivamente carga y almacenamiento de vector de datos (componentes) en doble precisión; un operando es el registro vectorial que se va a cargar o almacenar, y el otro operando un registro de propósito general (GPR) que contiene la dirección de comienzo del vector en memoria.

Máquina	Anunciada año	Registros vectoriales	Elementos de 64 bits por registro vectorial	Unidades funcionales vectoriales	Unidades vectoriales de carga/alm.
CRAY-1	1976	8	64	6: suma, multiplicación, recíproco, suma, entera, lógica, desplazamiento	1
CRAY X-MP CRAY Y-MP	1983 1988	8	64	8: suma FP, multiplicación FP, recíproco FP, suma entera, 2 lógicas, desplazamiento, población cuanta/paridad	2 cargas 1 almac.
CRAY-2	1985	8	64	5: suma FP, multiplicación FP, recíproco raíz cuadrada FP, (suma desplazamiento, cuenta población) entera, lógica	1
Fujitsu VP 100/200	1982	8-256	32-1024	3: suma/lógica entera o FP	2
Hitachi S810/820	1983	32	256	4: 2 suma entera/lógica, una multiplicación-suma y una multiplicación/división-suma	4
Convex C-1	1985	8	128	4: multiplicación, suma, división, entera/lógica	1
NEC SX/2	1984	8+8192	256 variable	16: 4 suma entera/lógica, 4 multiplicación/división FP, 4 suma FP, 4 desplazamientos	8
DLXV	1990	8	64	5: multiplicación, división, suma, suma entera, lógica	1

Figura 5.2. Características de algunas arquitecturas vectoriales con registros.

Además de los registros vectoriales, necesitamos dos registros adicionales de propósito general: ***registro de longitud de vector*** y ***registro de máscara de vector*** (posteriormente se explicará el propósito de estos registros).

Problema vectorial típico. **Y = a . X + Y** (**X** e **Y** son dos vectores, que inicialmente residen en memoria, y **a** es un escalar) $\Rightarrow$ Bucle SAXPY o DAXPY (según si es simple o doble precisión), que forma el bucle interno del benchmark de Linpack (Linpack es una colección de rutinas de álgebra lineal, de donde se ha utilizado la parte de eliminación Gaussiana como benchmark).

Vamos a suponer que el número de elementos o longitud de un registro vectorial (64) coincide con la longitud de la operación vectorial, y que las direcciones de comienzo de **X** y de **Y** están en Rx y Ry, respectivamente.

CÓDIGO BUCLE DAXPY PARA **DLX**

```
        LD      F0, a           ; carga de a
        ADDI    R4, Rx, #512    ; Última dirección a cargar (512 = 64 x 8)
LOOP:
        LD      F2, 0(Rx)       ; Carga X(i)
        MULTD   F2, F0, F2      ; a . X(i)
        LD      F4, 0(Ry)       ; Carga Y(i)
        ADDD    F4, F2, F4      ; a . X(i) + Y(i)
        SD      0(Ry), F4       ; almacena en Y(i)
        ADDI    Rx, Rx, #8      ; Incrementa el índice a X
        ADDI    Ry, Ry, #8      ; Incrementa el índice a Y
        SUB     R20, R4, Rx     ; Calcula el límite del vector X
        BNZ     R20, LOOP       ; Se comprueba si se ha terminado con el bucle
```

CÓDIGO BUCLE DAXPY PARA **DLXV**

```
LD      F0, a       ; Carga escalar de a
LV      V1, Rx      ; Carga vector X
MULTSV  V2, F0, V1  ; a . X (multiplicación escalar por vector)
LV      V3, Ry      ; Carga  vector Y
ADDV    V4, V2, V3  ; a . X(i) + Y(i) (suma vectorial)
SV      Ry, V4      ; almacena el resultado
```

RESULTADOS DE LA COMPARACIÓN DE LOS DOS BUCLES:

- La máquina vectorial (DLXV) ejecuta 6 instrucciones frente a ≈ 600 de DLX.
- Frecuencia de interbloqueos de las etapas ⇒ En DLX 64 veces mayor que en DLXV ⇒ En el DLX, cada ADDD debe esperar por un MULTD, y cada SD debe esperar por un ADDD. En DLXV, cada instrucción vectorial opera sobre todos los elementos del vector independientemente; sólo se requiere una detención por operación vectorial.

Aunque las detenciones de la segmentación se pueden eliminar en DLX utilizando software o desenrrollamiento de bucles, no se puede reducir la gran diferencia del ancho de banda de las instrucciones.

5.3 VELOCIDAD DE INICIACIÓN Y TIEMPO DE ARRANQUE VECTORIAL

El tiempo de ejecución de cada operación vectorial del bucle tiene dos componentes:

- ***Tiempo de arranque (Start-up)*** ⇒ Depende de la latencia de las etapas o segmentos de la operación vectorial. Está determinado principalmente por el número de etapas necesarias de la unidad funcional utilizada (una latencia de *n*

ciclos de reloj significa que la operación tarda *m* ciclos de reloj y que se necesita pasar por *m* etapas).

- ***Tiempo de iniciación (Velocidad de iniciación)*** ⇒ Es el tiempo por cada resultado una vez que una instrucción vectorial está en ejecución (normalmente una por ciclo de reloj, aunque algunos supercomputadores pueden producir dos o más resultados por ciclo de reloj) ⇒ La *velocidad de terminación* debe igualar como mínimo a la velocidad de iniciación, en otro caso no hay sitio para poner resultados.

$$\textbf{\textit{Tiempo operación vectorial}} = Tiempo\ de\ arranque + n * Tiempo\ de\ iniciación$$

n ≡ longitud del vector

$$\textbf{\textit{Ciclos de reloj por resultado}} \text{ (por cada componente del vector)} = \frac{Tiempo\ total\ operación\ vectorial}{Longitud\ del\ vector} = \frac{Tiempo\ de\ arranque + n * Tiempo\ de\ iniciación}{n}$$

Refiriéndonos a la fórmula que nos da el *tiempo de la operación vectorial* (en ciclos de reloj por ejemplo), vemos que el efecto de incrementar el *tiempo de arranque* en un vector de ejecución lento (*tiempo de operación vectorial* alto por tener muchas componentes) es pequeño; mientras que si se aplica el mismo incremento al *tiempo de arranque* en un vector de ejecución rápido (*tiempo de operación vectorial* pequeño por tener pocas componentes), el efecto es grande.

En la **Figura 5.3** se muestra gráficamente como, incrementando hasta 50 ciclos el *tiempo de arranque*, el *tiempo total de operación vectorial* se incrementa hasta un 78% [(50 / 64) x 100 = 78.125) partiendo de una ejecución de operaciones a *1 ciclo de reloj por resultado*, hasta un 39 % [(50 / 128) x 100 = 39.062] partiendo de una ejecución de operaciones a *2 ciclos de reloj por resultado*, y hasta menos de un 20 % [(50 / 256) x 100 = 19.531] partiendo de una ejecución de operaciones a *4 ciclos de reloj por resultado*.

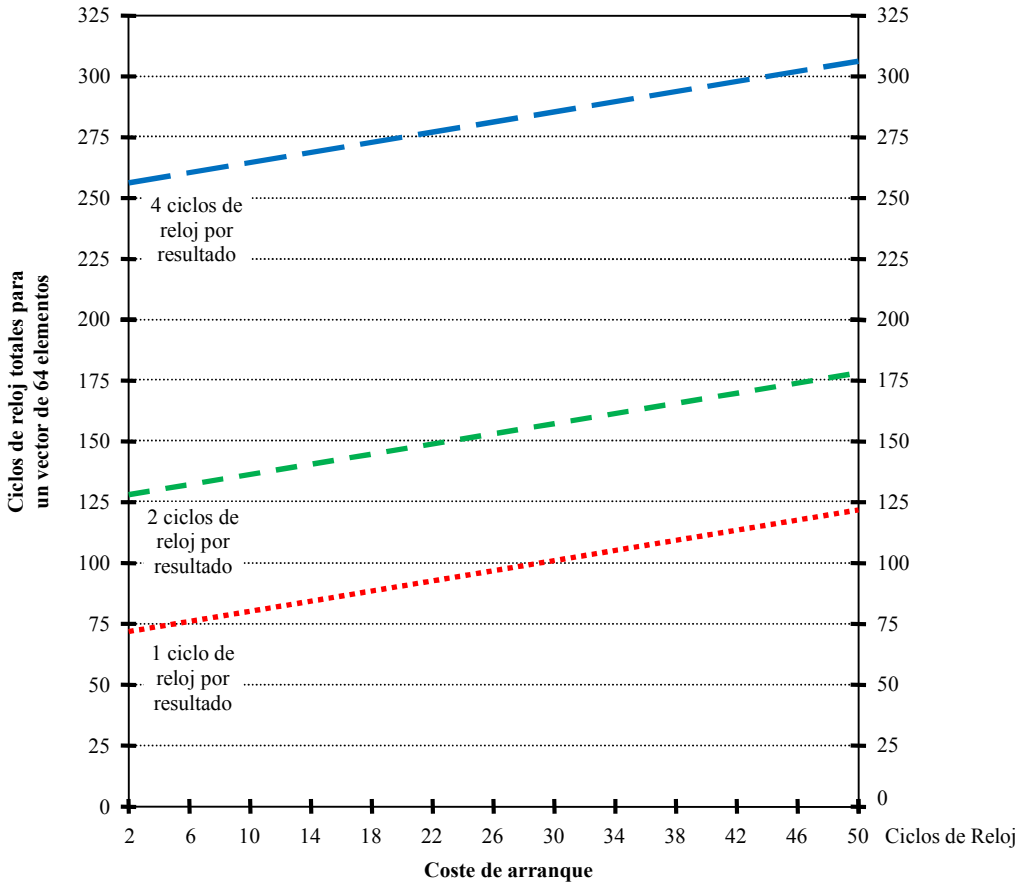

Figura 5.3. Evolución del *tiempo total de ejecución* incrementando desde 2 a 50 ciclos de reloj el *tiempo de arranque*.

¿Qué determina la *velocidad de arranque* y la *velocidad de iniciación*?

- **Operaciones que no involucran accesos a memoria (operaciones registro-registro)** ⇒ El *tiempo de arranque* en (ciclos de reloj) es igual al número de etapas de la unidad funcional, ya que este es el tiempo para obtener el primer resultado (Veremos posteriormente que hay otros costes involucrados que incrementan el tiempo de arranque) ⇒ El *Tiempo de iniciación* está determinada por la frecuencia con que la unidad funcional vectorial correspondiente pueda aceptar nuevos operandos. Si está totalmente segmentada, puede empezar una operación sobre nuevos operandos cada ciclo de reloj (*Tiempo de iniciación* de 1 ciclo de reloj). Si queremos mantener el *tiempo de iniciación* a 1 ciclo de reloj, entonces:

$$\textbf{\textit{Profundidad de la segmentación}} = \frac{\textit{Tiempo total de la unidad funcional}}{\textit{Duración del ciclo de reloj}}$$

Para DLXV, se eligen los mismos valores de la máquina CRAY-1. Todas la unidades funcionales están totalmente segmentadas. Los tiempos requeridos son:

- 6 ciclos de reloj para sumas en punto flotante.
- 7 ciclos de reloj para multiplicaciones en punto flotante.
- Si un cálculo vectorial depende de un resultado incompleto y necesitase ser detenido, añade una penalización extra de arranque de 4 ciclos de reloj (esta penalización es el tiempo para escribir y después leer los operandos y existe sólo cuando hay una dependencia).
- Las operaciones vectoriales independientes que utilizan diferentes unidades funcionales, pueden realizarse sin ninguna penalización o retardo.
- Las operaciones vectoriales independientes, también pueden estar completamente solapadas, y la emisión de cada instrucción sólo necesita un ciclo de reloj.

Como DLXV está totalmente segmentado, el *tiempo de iniciación* para una instrucción vectorial es siempre 1. Sin embargo una secuencia de operaciones vectoriales no se podrá ejecutar a esa velocidad, debido a los costes de arranque ⇒ Se usa el término ***tiempo sostenido (velocidad sostenida)*** ⇒ Tiempo por elemento para un conjunto de operaciones vectoriales relacionadas (p.e., en el bucle SAXPY, cada resultado de Y = a . X + Y) ⇒ Todas las operaciones vectoriales relacionadas con un elemento deben producir un resultado.

Ejemplo de cálculo del *tiempo sostenido*. Calcular el *tiempo sostenido* y el *nº efectivo de operaciones de punto flotante por ciclo de reloj*, para la secuencia de operaciones, con vectores de longitud 64, siguientes:

MULTV V1, V2, V3

ADDV V4, V5, V6

Los tiempos asociados a las mismas son:

Operación	Comienzo	Tiempo arranque	Tiempo iniciación	Finalización
MULTV	0	7	1	0 + 7 + 64 · 1 = 71
ADDV	1	6	1	1 + 6 + 64 · 1 = 71

Como las dos operaciones son independientes, la máquina vectorial puede emitirlas en unidades diferentes (en DLXV, en las unidades de Multiplicación FP y Suma/Resta FP), y obteniéndose por tanto un *tiempo sostenido* de 1 elemento en 71 ciclos de reloj (0.014 elementos/ciclo de reloj).

Operación	***Penalización de arranque***
Suma vectorial	6
Multiplicación vectorial	7
División vectorial	20
Carga vectorial	12

Cuando una instrucción vectorial depende de otra que no se ha completado en el instante que aparece la segunda instrucción vectorial, la penalización de arranque aumenta en 4 ciclos de reloj.

Figura 5.4. *Penalizaciones de arranque* de las operaciones vectoriales

La secuencia ejecuta 128 FLOP (Operaciones en punto Flotante) en 71 ciclos de reloj [64 MULTV y 64 ADDV] ⇒ 128 / 71 = 1.8 FLOP/ciclo de reloj. Una máquina vectorial puede mantener una productividad de más de una operación por ciclo de reloj, al emitir operaciones vectoriales independientes en diferentes unidades funcionales vectoriales.

- **Unidad vectorial de Carga/Almacenamiento** ⇒ El *tiempo de arranque* para una carga es el tiempo necesario para leer la primera palabra de memoria y escribirla en un registro [para DLXV vamos a suponer que es 12 ciclos de reloj para la lectura, y de 0 ciclos de reloj para la escritura (ya que la escritura no produce resultados; aunque, cuando una instrucción debe esperar que una escritura en memoria finalice, tendremos que esperar llegando a ver hasta 12 ciclos de latencia)]. En la **Figura 5.4** se resumen las *penalizaciones de arranque* para las operaciones vectoriales de DLXV.

  Si el resto del vector se puede suministrar sin detenciones, entonces el *tiempo de iniciación* del vector es igual a la que se extraen o almacenan las siguientes palabras ⇒ Para mantener una velocidad de iniciación de una palabra leída o escrita de/en memoria por ciclo de reloj, el sistema de memoria debe ser capaz de producir o aceptar esa cantidad de datos ⇒ Hay que distribuir la información en varios *bancos de memoria* (el concepto es similar al de *entrelazado de memoria*, pero no es exactamente igual), permitiendo cada uno de ellos realizar una lectura o escritura de una palabra independientemente de los otros

bancos. De esta forma, las palabras se transfieren desde memoria a la máxima frecuencia (1 palabra por ciclo de reloj en DLXV) ⇒ Para ello hay que hacerlo de alguna de las dos formas siguientes:

- Sincronizar todos los bancos ⇒ Para acceder a ellos en paralelo y en el mismo instante. Una vez leídos, se guardan juntos todos los resultados y, mientras se transfieren uno a uno al procesador, los bancos empiezan a servir otro grupo de peticiones ⇒ Necesita muchos elementos para almacenar los resultados de las lecturas pero el circuito de control es simple.
- Acceso a los bancos asíncrono o desfasado ⇒ Después del primer acceso en el que todos los bancos han sido accedidos en paralelo, las palabras leídas son enviadas al procesador una a una desde cada banco y, una vez un banco ha transmitido o almacenado su dato, el mismo comienza el próximo servicio inmediatamente ⇒ Necesita sistema de control complejo pero no necesita elementos para almacenar los resultados.

Suponiendo que cada banco tiene un ancho de una palabra en doble precisión (64 bits), si se mantiene una *velocidad de iniciación* de 1 por ciclo de reloj, debe cumplirse que:

$$\textit{Número de bancos de memoria} \geq \textit{Tiempo de acceso al banco de memoria}$$
(en ciclos de reloj)

Para simplificar direccionamientos, el número de bancos de memoria, normalmente, es una potencia de dos.

Examinemos el primer acceso por cada banco. Después de un tiempo igual al tiempo de acceso a memoria, todos los bancos de memoria habrán extraído una palabra en doble precisión, y las palabras pueden empezar a volver a los registros del vector. (Esto requiere por supuesto que los accesos estén alineados en límites de dobles palabras). Las palabras se envían en serie desde los bancos, comenzando con la extraída del banco con la dirección más baja. Si los bancos están sincronizados, los accesos siguientes comienzan inmediatamente; si los bancos funcionan de manera asíncrona, entonces el acceso siguiente comienza después que un elemento se transmite desde el

banco. Debido a que el tiempo de acceso a memoria en ciclos de reloj es menor que el número de bancos de memoria, y que las palabras se transfieren desde los bancos en orden de petición a la velocidad de una transferencia por ciclo de reloj, un banco completará el acceso siguiente antes que empiece de nuevo su turno para la transmisión de datos. Para simplificar direccionamientos, el número de bancos de memoria, normalmente, es una potencia de dos. Los diseñadores en general, quieren tener un número de bancos superior al mínimo necesario para minimizar los retardos, al acceder a memoria.

Ejemplo de lecturas en *bancos de memoria*. Suponiendo que se desea leer un vector de 64 elementos, comenzando en el byte de dirección 136, y que un acceso a memoria necesita 6 ciclos de reloj. ¿Cuántos bancos de memoria debe haber? ¿Con qué direcciones se accede a los bancos de memoria? ¿Cuándo llegan a la CPU los diversos elementos?.

Seis ciclos de reloj por acceso requieren al menos 6 bancos, pero como queremos que el número de bancos sea una potencia de dos, elegimos **8 bancos**. En la **Figura 5.5** se muestran las direcciones a nivel de byte que cada banco accede en cada periodo de tiempo. Recordar que un banco comienza un nuevo acceso tan pronto como ha completado el anterior acceso. La **Figura 5.6** muestra el diagrama de tiempo para los primeros conjuntos de accesos en un sistema de 8 bancos con una latencia de acceso de 6 ciclos de reloj.

Hay que hacer dos observaciones importantes a las Figuras 5.5 y 5.6:

1. La dirección exacta extraída por un banco está determinada por los bits de pesos 2^5, 2^4 y 2^3 ($....A_8A_7A_6\mathbf{A_5A_4A_3}A_2A_1A_0$), con la combinación correspondiente al número de banco (banco 0 con $A_5A_4A_3 = 000$, banco 1 con $A_5A_4A_3 = 001$, banco 2 con $A_5A_4A_3 = 010$, ..., banco 7 con $A_5A_4A_3 = 111$); sin embargo, el acceso inicial a un banco está siempre en 8 dobles palabras de la dirección inicial.
2. Una vez que se supera la latencia inicial (6 ciclos de reloj en este caso), el patrón es acceder un banco cada n ciclos de reloj, donde n es el número total de bancos (8 en este caso).

El número de bancos del sistema de memoria y el número de segmentos usados en las unidades funcionales son esencialmente conceptos equivalentes, ya que determinan las velocidades de iniciación para las operaciones que utilizan estas unidades.

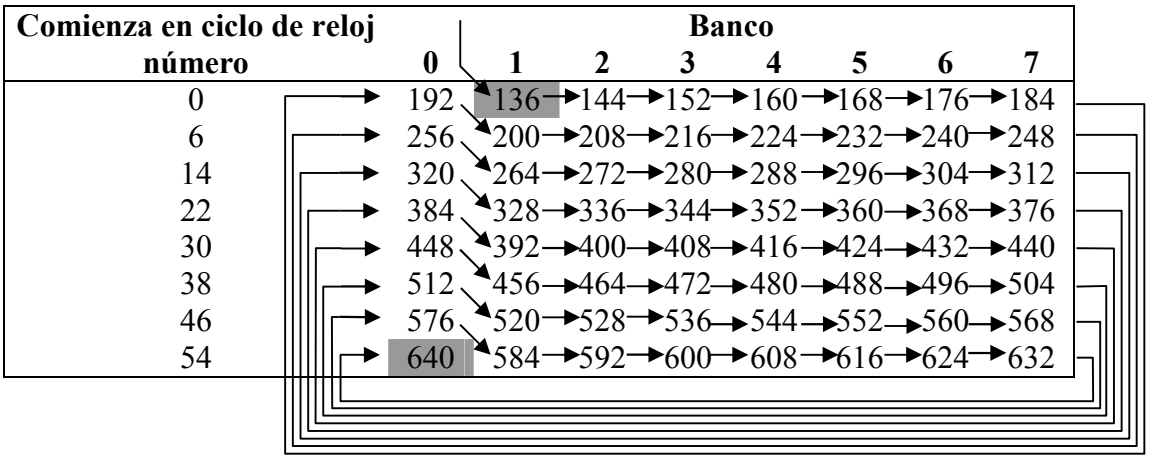

Figura 5.5. Direcciones de memoria (en bytes) por número de bancos e instante de tiempo en que comienza el acceso. El instante exacto en que un banco transmite sus datos está dado por la dirección a la que éste accede menos la dirección de comienzo (primera dirección que contiene ese banco de ese vector), dividida por 8, y más la latencia de memoria (6 ciclos de reloj). Es importante observar que el banco 0 accede a 192 en lugar de 128 y a 256 en lugar de 192, ..., puesto que si comenzase en la dirección 128, necesitaríamos un ciclo extra para transmitir el dato, y transmitiríamos un valor innecesariamente. (Un valor en este ejemplo, si tuviéramos por ejemplo, 64 bancos, podrían presentarse hasta 63 ciclos de reloj y transferencias innecesarias). El hecho de que el banco 0 no acceda a una palabra en el mismo bloque de 8 distingue a este tipo de sistemas de memoria (***banco de memoria***) de la ***memoria entrelazada***. Normalmente, los sistemas de ***memoria entrelazada*** combinan la dirección del banco y la dirección de base de comienzo por concatenación en lugar de adición. También, las ***memorias entrelazadas*** se implementan casi siempre con accesos sincronizados; los ***bancos de memoria*** requieren cerrojos de dirección para cada banco, que normalmente no son necesarios en un sistema con sólo entrelazado.

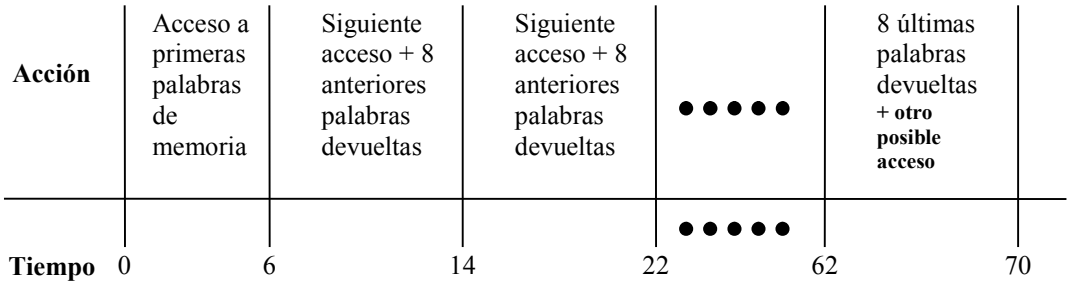

Figura 5.6. Temporización de accesos para las 64 primeras palabras de la carga de doble precisión.

5.4 LONGITUD DEL VECTOR Y SEPARACIÓN ENTRE ELEMENTOS

En los programas reales nos podemos encontrar con que la longitud del vector no es exactamente la misma que la correspondiente a los registros vectoriales (64 en DLXV) y/o a veces los elementos consecutivos de un vector no están adyacentes en memoria después de que la matriz se ha almacenado en la misma.

CONTROL DE LA LONGITUD DEL VECTOR

Una máquina con registros vectoriales o bancos de registros tiene una longitud natural de los vectores determinada por el número de elementos de cada registro vectorial (64 para DLXV). Esta longitud es improbable que coincida con la longitud real de los vectores en un programa. Además, en un programa real, la longitud de una operación particular es, con frecuencia, desconocida en tiempo de compilación.

Solución si el tamaño del vector es menor que el número de elementos de cada registro vectorial de la máquina ⇒ Crear un ***registro de longitud vectorial (VLR)*** ⇒ Controla la longitud de cualquier operación vectorial, incluyendo la carga o almacenamiento de un vector. El valor de VLR no puede ser mayor que la longitud de los registros vectoriales.

Solución si el tamaño del vector (n) no se conoce en tiempo de compilación y además puede que sea mayor que la *longitud máxima del vector (MVL)* ⇒ Se utiliza la técnica de ***seccionamiento (strip mining)*** ⇒ Es la generación de código tal que cada operación vectorial se realiza para un tamaño de código menor o igual que la MVL ⇒ La longitud de la primera sección (ship) es "*n módulo MVL*", y la de todos los segmentos siguientes es "*MVL*" (Se muestra en la **Figura 5.7**).

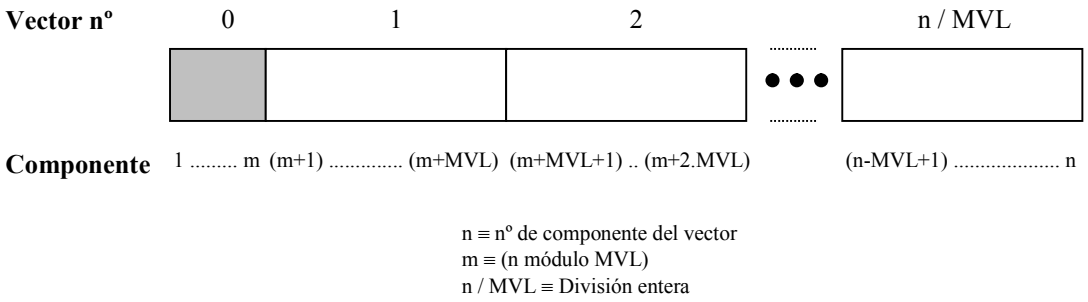

Figura 5.7. Vector de longitud arbitraria (n) procesado con seccionamiento (strip mining).

Con múltiples operaciones vectoriales ejecutándose en paralelo, el hardware debe copiar el valor de VLR cuando se emita una operación vectorial, en el caso que VLR sea cambiado por una operación vectorial posterior.

Cuando teníamos vectores con el mismo número de componentes que la longitud de los registros vectoriales, los *costes de arranque* pueden calcularse independientemente para cada operación vectorial. Con el ***seccionamiento (strip mining)***, un porcentaje significativo de los *costes de arranque* estará en los *costes de seccionamiento* y, por tanto, calcular los *costes de arranque* será algo más complejo.

Los costes añadidos son muy significativos. Por ejemplo, para un simple bucle, en el que el compilador genere varios bucles anidados, cada iteración del bucle de la operación original del vector requeriría por ejemplo dos ciclos de reloj si no hubiera penalizaciones de arranque de ningún tipo; en cambio, con penalizaciones de arranque (que son de dos tipos: costes de arranque del vector y costes de seccionamiento), podría requerir 16 ciclos más, sin ni siquiera tener en cuenta los costes de seccionamiento. La **Figura 5.8** muestra el impacto sólo del coste de arranque vectorial cuando el vector crece de longitud 1 a longitud 64. Este coste de arranque puede disminuir la velocidad (*throughput*) en un factor de 9, dependiendo de la longitud del vector.

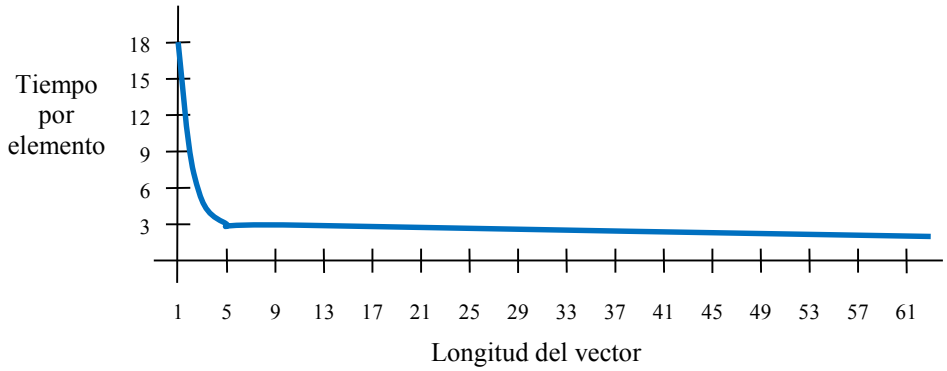

Figura 5.8. Impacto del coste de arranque del vector en un bucle consistente en una asignación vectorial. Para vectores cortos, el impacto del coste de arranque de 16 ciclos es enorme, decreciendo el rendimiento hasta nueve veces. No se ha incluido el coste de seccionamiento.

SEPARACIÓN ENTRE ELEMENTOS DE UN VECTOR

Cuando una matriz se ubica en memoria, se linealiza y debe organizarse por filas o columnas (*row major* o *column major*). El *almacenamiento por filas* consiste en asignar posiciones consecutivas a elementos consecutivos de cada fila, haciendo adyacentes a los elementos A(i,j) y A(i,j+1) de la matriz A. El *almacenamiento por columnas* hace adyacentes a los elementos A(i,j) y A(i+1,j). Si por ejemplo hay que realizar una multiplicación de dos matrices A y B, ubicadas en memoria con *almacenamiento por filas*, se realizarán accesos adyacentes a los elementos de la matriz A, mientras a los elementos de la matriz B se accederán con una distancia entre los mismos del tamaño de la fila multiplicado por el número de bytes por elemento $\Rightarrow$ A la distancia entre los elementos consecutivos que van a formar un vector se le denomina ***separación (stride)*** (La separación entre los elementos de la matriz A será de 1 [1 doble palabra (8 Bytes) si consideramos que cada componente tiene 8 bytes)] y la de los elementos

de la matriz B la del tamaño de la fila [multiplicado por 8 bytes, considerando el mismo tamaño anterior por cada componente]).

Una vez que un vector se carga en un registro vectorial, actúa como si tuviera los elementos lógicamente adyacentes. Esto posibilita que una máquina con registros vectoriales maneje separaciones entre elementos mayores que uno (*separaciones no unitarias*), haciendo las operaciones de almacenamiento y carga de los vectores más generales.

Por todo lo especificado, *es deseable que las operaciones de carga y almacenamiento de vectores especifiquen una separación entre elementos además de una dirección de comienzo* ⇒ Esta cuestión se resuelve de dos formas:

1. Máquina con instrucciones de *Cargar Vector con Separación* (**LVWS** en DLXV) y *Almacenar Vector con Separación* (**SVWS** en DLXV) ⇒ Tanto la separación del vector como su dirección de comienzo pueden ponerse en un registro de propósito general, durante la duración de la operación vectorial.
2. Máquinas sin instrucciones de carga y almacenamiento con separación específicas ⇒ El valor de la separación siempre está almacenado en un registro para tal fin.

El valor de la separación se debe calcular dinámicamente, ya que el tamaño de la matriz puede no conocerse en tiempo de compilación, o la longitud del vector puede cambiar para diferentes ejecuciones de la misma sentencia.

En la *Unidad de memoria*, al soportar separaciones entre elementos mayores que la unidad, se pueden presentar complicaciones ⇒ Veíamos que una operación memoria – registro podía proceder a velocidad completa si el número de bancos de memoria era como mínimo tan grande como el tiempo de acceso a memoria en ciclos de reloj. Sin embargo, si se introducen separaciones no unitarias, es posible pedir accesos al mismo banco a una frecuencia mayor que el tiempo de acceso a memoria ⇒ ***Conflicto del banco de memoria*** ⇒ Cuando se le pide al mismo banco que realice un acceso antes que se haya completado otro ⇒ Cada carga necesita un mayor tiempo de acceso a memoria ⇒ Un conflicto de banco de memoria, y por consiguiente una detención, se presentará si:

$$\frac{\text{Mínimo común múltiplo (separación, nº de bancos)}}{\text{Separación}} < \text{Latencia de acceso de memoria}$$

Ejemplo de *conflicto en bancos de memoria*. Supongamos que tenemos 16 bancos de memoria con un tiempo de acceso de 12 ciclos de reloj. ¿Cuánto se tardará en completar la lectura de un vector de 64 elementos con una separación de 1?; ¿y con una separación de 32?.

Con una separación de 1, como el número de bancos es mayor que la latencia de cada módulo, la lectura empleará 12 + 64 = 76 ciclos de reloj (1.19 ciclos de reloj por elemento).

La peor separación posible es un valor que sea múltiplo del número de bancos de memoria, como en este caso con una separación de 32 (con 16 bancos de memoria). Cada acceso a memoria colisionará con el anterior. Por lo tanto, el tiempo de acceso será de 12 ciclos de reloj por elemento, y un tiempo total para la carga del vector completo de 12 x 64 = 768 ciclos de reloj.

Los *conflictos en los bancos de memoria* no se presentarán si la separación entre elementos y el número de bancos son relativamente primos entre sí y hay suficientes bancos de memoria para evitar conflictos en el caso de separación unidad.

Aumentar el número de bancos de memoria a un número mayor del mínimo para prevenir detenciones con una separación de longitud unidad, disminuirá la frecuencia de detenciones para las demás separaciones. (Por ejemplo, con 64 bancos: una separación de 32, parará cada dos accesos; una separación de 16, parará cada cuatro accesos; una separación de 8, parará cada ocho accesos; ...) $\Rightarrow$ En los años 90, la mayoría de los supercomputadores vectoriales tenían como mínimo 64 bancos, y algunos tenían 512.

5.5 MODELO PARA EL CÁLCULO DEL RENDIMIENTO VECTORIAL

Hay tres componentes clave del tiempo de ejecución de un bucle seccionado cuyo cuerpo es una secuencia de instrucciones vectoriales:

1. $T_{elemento}$ $\Rightarrow$ El tiempo de cada operación vectorial en el bucle para procesar un elemento (una componente), ignorando los costes de arranque. Cuando la secuencia vectorial tiene un único resultado, como ocurre con frecuencia, $T_{elemento}$ es el tiempo en producir un elemento de ese resultado. Si la secuencia vectorial produce múltiples resultados, $T_{elemento}$ es el tiempo en producir un elemento de cada resultado. Este tiempo depende solamente de la ejecución de las instrucciones del vector; depende principalmente del hardware.
2. *Coste adicional (overhead) de las instrucciones vectoriales para cada bucle seccionado*. Dependen de la máquina (hardware) y del compilador $\Rightarrow$ Está formado por:
    a. T_{bucle} $\Rightarrow$ Coste de ejecución del código escalar para seccionamiento de cada bloque.
    b. $T_{arranque}$ $\Rightarrow$ Coste de arranque del vector para cada bloque.
3. T_{base} $\Rightarrow$ *Costes adicionales (overhead) del cálculo de las direcciones de comienzo y la escritura del vector de control*. Se presenta una vez para la operación completa del vector. Este tiempo se corresponde únicamente con instrucciones escalares.

Estos componentes determinan el *tiempo total de ejecución para una secuencia vectorial operando sobre un vector de longitud n (T_n)*.

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

Por simplicidad, utilizaremos valores constantes para T_{base} y T_{bucle}, en DLXV. En base a una serie de medidas de ejecución vectorial en el CRAY-1, los valores escogidos son T_{base} = 10 y T_{bucle} = 15. Aunque inicialmente se puede pensar que esos valores son muy pequeños, especialmente el de T_{bucle}, las instrucciones escalares que implican (inicializar las separaciones y las direcciones de comienzo del vector, incrementar los contadores y ejecutar un salto de bucle) pueden ejecutarse de manera solapada con las instrucciones vectoriales, minimizando el tiempo empleado en estas funciones de coste adicional. Por supuesto, aunque los valores T_{base} y T_{bucle} dependen de la estructura del bucle, la dependencia es pequeña (por eso se simplifica y consideran valores constantes para esos parámetros) comparada con la correspondiente al código vectorial y los valores $T_{elemento}$ y $T_{arranque}$.

Ejemplo de cálculo del *tiempo de ejecución de una operación vectorial*. ¿Cuál es el tiempo de ejecución para la operación vectorial $\mathbf{A} = \mathbf{B} \cdot s$, donde *s* es un escalar y la longitud de los vectores $\mathbf{A}$ y $\mathbf{B}$ es de 200?.

El código seccionado de DLXV, suponiendo que las direcciones de $\mathbf{A}$ y $\mathbf{B}$ están inicialmente en R_a y R_b y *s* está en F_s, puede ser el siguiente:

CÓDIGO SECCIONADO PARA **DLXV**

```
        ADDI    R2, R0, #1600   ; Nº de bytes del vector

        ADD     R2, R2, Ra      ; Fin del vector A
        ADDI    R1, R0, #8      ; Longitud de seccionamiento
        MOVI2S  VLR, R1         ; Longitud del vector
        ADDI    R1, R0, #64     ; Longitud en bytes
        ADDI    R3, R0, #64     ; Longitud del vector de otras piezas
LOOP:   LV      V1, Rb          ; Carga B
        MULTVS  V2, V1, Fs      ; Multiplicación del vector por el escalar
        SV      Ra, V2          ; Almacena A
        ADD     Ra, Ra, R1      ; Siguiente sección de A
        ADD     Rb, Rb, R1      ; Siguiente sección de B
        ADDI    R1, R0, #512    ; Longitud total del vector (en bytes)

        MOVI2S  VLR, R3         ; Se inicializa la longitud a 64
        SUB     R4, R2, Ra      ; ¿Se ha llegado al fin de A ?
        BNZ     R4, LOOP        ; Si no, volver atrás
```

Según el código anterior, $T_{elemento}$ = 3 (carga, multiplicación y almacenamiento de cada elemento del vector). Además, nuestras suposiciones para DLXV son: T_{bucle} = 15 y T_{base} = 10. Haciendo uso de la fórmula básica, se tiene:

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \cdot \left(T_{bucle} + T_{arranque} \right) + n \cdot T_{elemento} \Rightarrow T_{200} = 10 + (4) \cdot \left(15 + T_{arranque} \right) + 200 \cdot 3$$

$$T_{200} = 670 + 4 \cdot T_{arranque}$$

El valor de $T_{arranque}$ es la suma de:

- El arranque de la carga del vector de 12 ciclos de reloj.
- La detención de 4 ciclos de reloj, debido a la dependencia entre la carga y la multiplicación.
- Un arranque de 7 ciclos de reloj más, para la multiplicación.
- Una parada de 4 ciclos de reloj debido a la dependencia entre la multiplicación y el almacenamiento.

$$T_{arranque} = 12 + 4 + 7 + 4 = 27$$

Por lo tanto: $T_{200} = 670 + 4 \cdot 27 = 778$; y, el tiempo de ejecución por elemento, incluyendo los costes de arranque, es $T_{elemento\ con\ costes\ arranque} = \frac{778}{200} = 3.9$ ($T_{elemento} = 3$ es el caso ideal).

La **Figura 5.9** muestra el coste adicional y las velocidades efectivas por elemento para el ejemplo anterior ($\mathbf{A} = \mathbf{B} \cdot s$) con varias longitudes vectoriales. Si se toma como referencia el modelo más simple de arranque, ilustrado en la **Figura 5.8**, vemos que los costes (en ciclos de reloj) para todos los tamaños de vector se incrementan.

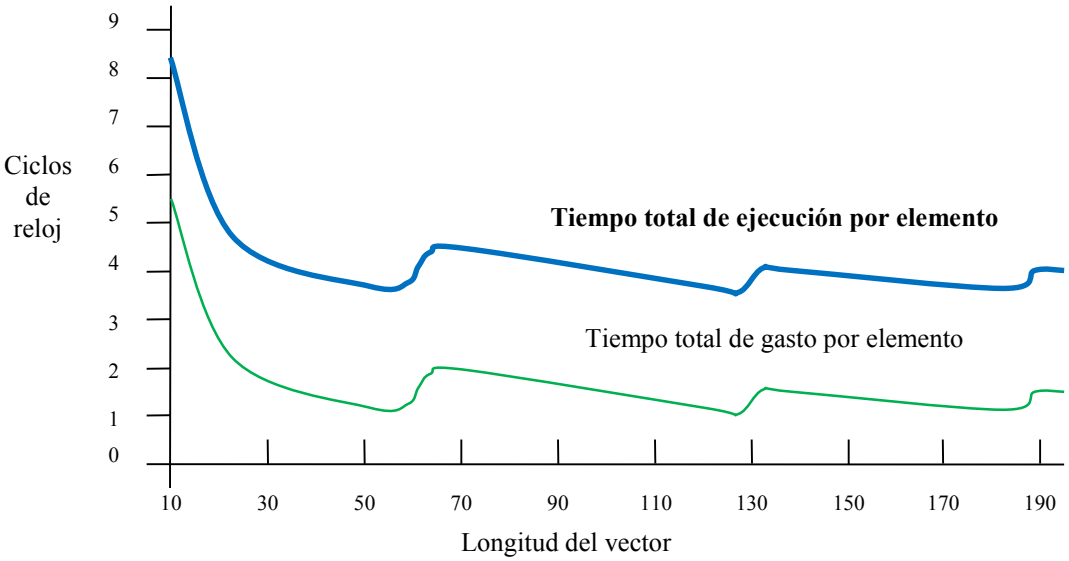

Figura 5.9. Tiempo total de ejecución por elemento y tiempo total de gasto por elemento, frente a la longitud vectorial para el ejemplo mostrado. Para vectores cortos, el tiempo total de arranque es más de la mitad del tiempo total; mientras que para vectores grandes, se reduce aproximadamente a una tercera parte del tiempo total. Las bifurcaciones repentinas se presentan cuando la longitud del vector cruza un múltiplo de 64, forzando a otra iteración del código seccionado y a la ejecución de un conjunto de instrucciones vectoriales. Estas operaciones incrementan T_n en $T_{bucle} + T_{arranque}$.

5.6 COMPILADORES PARA MÁQUINAS VECTORIALES

Para poder utilizar con efectividad una máquina vectorial, el compilador debe poder reconocer que un bucle (o parte de un bucle) es vectorizable, y generar el código vectorial apropiado. Esto implica determinar las dependencias que existen entre los operandos en el bucle.

Si se considera un bucle como el siguiente:

BUCLE

	DO	$10 \quad i = 1,100$	; Iteraciones del bucle
S1		$\mathbf{A}(i+1) = \mathbf{A}(i) + \mathbf{B}(i)$	; Cuerpo del bucle
S2		$\mathbf{B}(i+1) = \mathbf{B}(i) + \mathbf{A}(i+1)$	; Cuerpo del bucle
10	**CONTINUE**		; Termina el bucle

Los posibles tipos de dependencias diferentes son:

1. La sentencia S1 utiliza un valor calculado por la misma sentencia S1 en la iteración anterior. Esto es cierto para S1 ya que la iteración $i+1$ utiliza el valor $\mathbf{A}(i)$ que se calculó en la iteración i como $\mathbf{A}(i+1)$. Para S2 ocurre lo mismo con $\mathbf{B}(i)$ y $\mathbf{B}(i+1)$.
2. La sentencia S1 utiliza un valor calculado por la sentencia S2 en una iteración anterior. Esto es cierto ya que S1 utiliza el valor de $\mathbf{B}(i+1)$ en la iteración $i+1$, que S2 calcula en la iteración i.
3. La sentencia S2 utiliza un valor calculado por la sentencia S1 en la misma iteración. Esto es cierto para el valor de $\mathbf{A}(i+1)$.

Como las operaciones vectoriales están segmentadas y la latencia puede ser bastante grande, una iteración anterior puede no completarse antes de que comience una iteración posterior; entonces, los valores que debe escribir la iteración anterior pueden no estar escritos antes de que comience la iteración posterior. Por lo tanto, si existe la situación 1 o la situación 2, la vectorización del bucle introducirá un riesgo RAW (riesgo que una máquina vectorial no comprueba). Esto significa que si existe alguna de las tres dependencias en las situaciones 1 y 2, el bucle no es vectorizable, y el compilador no generará instrucciones vectoriales para este código. En la situación 3, el hardware normal de detección de riesgos podría manejar la situación; es decir, un bucle que contenga únicamente dependencias como las de la situación 3, puede vectorizarse. Las dependencias de las situaciones 1 y 2, que involucran el uso de valores calculados en iteraciones anteriores del bucle, se denominan ***dependencias entre iteraciones del bucle*** *(**loop-carried dependences**)*.

La primera tarea del compilador es determinar si en el cuerpo del bucle hay dependendia entre iteraciones del bucle. El compilador realiza esta operación haciendo uso de un algoritmo de análisis de dependencias. Como las sentencias del cuerpo del bucle involucran arrays, el análisis de dependencias es complejo (si no hubiese arrays, no habría nada que vectorizar). El caso más simple se presenta cuando aparece un nombre de array sólo en una parte de la sentencia de asignación. Por ejemplo, el bucle siguiente no es más que una variación del anterior:

BUCLE

	DO	10 $i = 1,100$	; Iteraciones del bucle
S1		$\mathbf{A}(i) = \mathbf{B}(i) + \mathbf{C}(i)$	; Cuerpo del bucle
S2		$\mathbf{D}(i) = \mathbf{A}(i) \cdot \mathbf{E}(i)$	; Cuerpo del bucle
10	**CONTINUE**		; Termina el bucle

Si los arrays **A**, **B**, **C**, **D** y **E** son diferentes, entonces no pueden existir dependencias entre iteraciones del bucle. Hay una dependencia entre las dos sentencias para el vector **A**. Si el compilador cae en la cuenta de que había dos accesos a **A**, puede tratar de no releer **A** en la segunda sentencia haciendo en su lugar la multiplicación vectorial, utilizando el registro de resultados de la suma vectorial. En este caso, el procesador vería el riesgo potencial RAW y pararía la emisión de la multiplicación de los vectores. Si el compilador almacenó **A** y lo releyó, entonces la lecturas y almacenamientos se presentan en orden, lográndose una ejecución correcta.

Con frecuencia, en un bucle aparece el mismo nombre como fuente y destino (por ejemplo en el bucle SAXPY), apareciendo en ambos lados de la asignación:

BUCLE

	DO	10 $i = 1,100$	; Iteraciones del bucle
		$\mathbf{Y}(i) = a \cdot \mathbf{X}(i) \cdot \mathbf{Y}(i)$	; Cuerpo del bucle
10	**CONTINUE**		; Termina el bucle

En este caso no hay dependencias arrastradas por el bucle, porque la asignación a **Y** no depende del valor de **Y** calculado en la iteración anterior. Sin embargo, el bucle siguiente, que se denomina ***recurrencia***, contiene una dependencia entre iteraciones del bucle:

BUCLE

```
      DO      10  i = 2,100          ; Iteraciones del bucle
                  Y(i) = Y(i-1) + Y(i)   ; Cuerpo del bucle
10    CONTINUE                     ; Termina el bucle
```

La dependencia puede verse desenrollando el bucle: en la iteración j se utiliza el valor de $\mathbf{Y}(j-1)$, pero ese elemento se calcula en la iteración $(j-1)$, creando una dependencia entre iteraciones.

En general, un compilador detecta las dependencias de la siguiente forma. Si suponemos que hemos escrito un elemento del array con un valor índice $a \cdot i + b$ y es accedido con el valor índice $c \cdot i + d$, donde i es la variable índice del bucle que varía de m a n. Existe una dependencia si se mantienen dos condiciones:

1. Hay dos índices de iteración, j y k, ambos dentro de los límites del bucle.
2. El bucle almacena un elemento del array indexado por $a \cdot j + b$ y más tarde busca ese mismo elemento del array cuando éste es indexado por $c \cdot k + d$. Esto es, $a \cdot j + b = c \cdot k + d$.

En general, a veces no se puede determinar si existe dependencia en tiempo de compilación. Por ejemplo, los valores a, b, c y d pueden no ser conocidos, siendo imposible decir si existe alguna dependencia. En otros casos, la comprobación de dependencias puede ser muy cara, pero decidible en tiempo de compilación. Por ejemplo, los accesos pueden depender de los índices de iteración de múltiples bucles anidados. Muchos programas no contienen estas estructuras complejas, pero en su lugar contienen simples índices donde a, b, c y d son constantes. Para estos casos, es posible imaginar tests razonables de dependencias.

Un tests sencillo y suficiente utilizado para detectar dependencias es el ***máximo común divisor*** (**GCD**). Está basado en la observación de que si existe una dependencia entre iteraciones del bucle, entonces GCD (c, a), debe dividir a ($d - b$). Esta prueba nos garantiza la no existencia de dependencia; sin embargo, hay casos donde esta prueba tiene éxito y no existe dependencia. Por ejemplo, esto puede suceder porque el test GCD no tiene en cuenta los límites o extremos del bucle. Un test más complejo es el de U. Barnerjee (1979), que tiene en cuenta los límites o extremos del bucle; aunque, todavía este test no es del todo exacto. Siempre puede

hacerse un test exacto resolviendo ecuaciones con valores enteros pero, para estructuras complejas de bucle, puede ser bastante caro.

Ejemplo de utilización de la *prueba GCD*. Determinar si existen dependencias en el siguiente bucle:

BUCLE PARA APLICAR LA PRUEBA GCD

```
      DO      10  i = 1,100                ; Iteraciones del bucle
              X(2·i+3) = X(2·i)·5.0        ; Bucle
10    CONTINUE
```

En este caso los valores son $a = 2, b = 3, c = 2$ *y* $d = 0$; que nos da GCD(a, c) = 2, y $d - b = -3$. Por lo tanto, como 2 no divide a -3, no hay dependencia posible.

Una ***dependencia verdadera de datos*** surge de un riesgo RAW y prevendrá la vectorización del bucle como una simple secuencia vectorial. Hay casos donde el bucle se puede vectorizar como dos secuencias vectoriales separadas. Hay también dependencias correspondientes a riesgos WAR (escritura después de lectura), denominadas ***antidependencias***, y a riesgos WAW (escritura después de escritura), denominadas ***dependencias de salida***. Estas *antidependencias* y *dependencias de salida* no son verdaderas dependencias de datos; son conflictos de nombres que se pueden eliminar renombrando los registros en el compilador con un método similar al que usa el algoritmo de Tomasulo cuando renombra los registros en tiempo de ejecución. Los compiladores que vectorizan utilizan con frecuencia renombramiento en tiempo de compilación, para eliminar *antidependencias* y *dependencias de salida*.

Ejemplo para determinar los tipos de dependencias existentes en un bucle. El siguiente bucle tiene una *antidependencia* (WAR) y una *dependencia de salida* (WAW). Determinar todas las *dependencias verdaderas*, *dependencias de salida* y *antidependencias* y eliminar posteriormente las *dependencias de salida* y *antidependencias* renombrándolas.

BUCLE

```
      DO      10  i = 1,100                ; Iteraciones del bucle
S1            Y(i) = X(i) / s              ; Cuerpo del bucle
S2            X(i) = X(i) + s              ; Cuerpo del bucle
S3            Z(i) = Y(i) + s              ; Cuerpo del bucle
S4            Y(i) = s - Y(i)              ; Cuerpo del bucle
```

```
10      CONTINUE                                  ; Termina el bucle
```

Hay *dependencias verdaderas* entre la sentencia S1 y la sentencia S3, y entre la sentencia S1 y la sentencia S4 a causa de $\mathbf{Y}(i)$. Estas no son arrastradas por el bucle, así que no evitarán la vectorización. Sin embargo, las dependencias forzarán a que las sentencias S3 y S4 esperen a que se complete la sentencia S1, aún cuando las sentencias S3 y S4 utilicen unidades funcionales diferentes a las de la sentencia S1. Existen técnicas que nos permiten eliminar esta serialización (no se tratará aquí por no extender este tema en exceso).

En el ejemplo que se considera, existe una *antidependencia* entre la sentencia S1 y la sentencia S2, y una *dependencia de salida* entre la sentencia S1 y la sentencia S4. La siguiente versión del bucle elimina esta *falsas* (o *pseudo*) *dependencias*:

BUCLE

```
        DO      10  i = 1,100        ; Iteraciones del bucle
                                     ; Y renombrado a T para eliminar
                                     ; dependencia de salida
S1              T(i) = X(i) / s      ; Cuerpo del bucle
                                     ; X renombrado a X1 para eliminar
                                     ; antidependencia
S2              X1(i) = X(i) + s     ; Cuerpo del bucle
                                     ; Se modifican las sentencias según el
                                     ; renombramiento realizado
S3              Z(i) = T(i) + s      ; Cuerpo del bucle
S4              Y(i) = s - T(i)      ; Cuerpo del bucle
10      CONTINUE                     ; Termina el bucle
```

En el código que sigue al bucle, el compilador puede sencillamente sustituir el nombre **X** de la variable por **X1**. Para renombrar no se requiere una operación real de copia, puede hacerse por sustitución de nombres o por asignación de registros.

El compilador, además de decidir qué bucles son vectorizables, debe generar código de seccionamiento y asignar los registros vectoriales. Muchas transformaciones de vectorización se hacen a nivel de fuente, aunque algunas optimizaciones involucren coordinar transformaciones fuente de alto nivel con transformaciones dependientes de la máquina (de nivel más bajo). La optimización eficiente de los registros vectoriales es otra optimización, pero es quizás la optimización más difícil, la que muchos compiladores que vectorizan ni siquiera intentan.

5.7 EFICACIA DE LAS TÉCNICAS DE VECTORIZACIÓN

Dos factores afectan al éxito con que se puede ejecutar un programa en modo vectorial:

1. La estructura del mismo programa ⇒ ¿Tienen los bucles dependencias verdaderas de datos, o se pueden reestructurar para que no tengan dichas dependencias?. Este factor está influenciado por los algoritmos escogidos y, en alguna extensión, por la forma en que están codificados.

2. La capacidad del compilador ⇒ Aunque el compilador no pueda vectorizar un bucle donde no exista paralelismo entre las iteraciones del bucle, hay una tremenda variación en las posibilidades de los compiladores para determinar si se puede vectorizar un bucle.

En la **Figura 5.10** se muestran, como indicación del nivel de vectorización que se puede conseguir en programas científicos, los niveles de vectorización observados en los benchmarks de "Perfect Club". Estos benchmarks son grandes aplicaciones científicas reales.

Nombre del benchmark	Operaciones FP	Operaciones FP ejecutadas en modo vectorial
ADM	23%	68%
DYFESM	26%	95%
FLO52	41%	100%
MDG	28%	27%
MG3D	31%	86%
OCEAN	28%	58%
QCD	14%	1%
SPICE	16%	7%
TRACK	9%	23%
TRFD	22%	10%

Figura 5.10. Nivel de vectorización entre los benchmarks de Perfect Club cuando se ejecutan en el CRAY X-MP. La primera columna contiene el porcentaje de operaciones de punto flotante, mientras que la segunda columna contiene el porcentaje de operaciones de punto flotante ejecutadas en instrucciones vectoriales.

La amplia variación en el nivel de vectorización se ha observado en varios estudios de rendimiento de aplicaciones en máquinas vectoriales. Mientras los mejores compiladores pueden mejorar el nivel de vectorización de algunos de estos programas, la mayoría necesitará reescribirse para conseguir incrementos significativos de vectorización.

También hay una tremenda variación en la forma en que los compiladores vectorizan los programas. Como resumen del estado de los compiladores que vectorizan, la **Figura 5.11** muestra la extensión de la vectorización para diferentes máquinas utilizando un grupo de pruebas de 100 “kernels” de Fortran escritos a mano. Los “kernels” fueron diseñados para probar la capacidad de vectorización y, como característica común, todos se podían vectorizar manualmente.

Existen varias técnicas para mejorar el rendimiento vectorial; entre éstas se encuentran el *encadenamiento*, técnicas para *sentencias ejecutadas condicionalmente y matrices dispersas*, y las técnicas para un bucle de tipo *reducción vectorial*. La primera, encadenamiento, se empezó a aplicar en el CRAY-1, pero ahora está soportada en muchas máquinas vectoriales. Las otras dos técnicas se han desarrollado en diversas máquinas y, en general, van más allá de las capacidades proporcionadas en las arquitecturas CRAY-1 o CRAY X-MP. (En estos apuntes no se desarrollarán estas técnicas; para ampliación consultar el libro *Arquitectura de Computadores. Un enfoque cuantitativo*, de John A. Hennessy y David A. Patterson (págs. 405-411).

Máquina	Compilador	Completamente vectorizados	Parcialmente vectorizados	No vectorizados
Ardent Titan-1	FORTRAN V1.0	62	6	32
CDC CYBER-205	VAST-2 V2.21	62	5	33
Series Convex C	FC5.0	69	5	26
CRAY X-MP	CFT77 V3.0	69	3	28
CRAY X-MP	CFT V1.15	50	1	49
CRAY-2	CFT2 V3.1ª	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77/SX V.040	66	5	29
Stellar GS 1000	F77 preliberación	48	11	41

Figura 5.11. Resultado de usar compiladores que vectorizan a 100 núcleos (kernels) FORTRAN de test. Para cada máquina se indica el número de bucles completamente vectorizados, parcialmente vectorizados y no vectorizados. Estos bucles fueron seleccionados por Callahan, Dongarra y Levine (1988). Dos compiladores diferentes para el CRAY X-MP muestran la gran dependencia en tecnología de compiladores.

5.8 COMPARACIÓN DE LAS MÁQUINAS VECTORIALES CON LAS MÁQUINAS ESCALARES SEGMENTADAS

A finales de los años ochenta, el rápido incremento del rendimiento en las máquinas escalares con buenas segmentaciones, condujo a una disminución espectacular del

espacio vacío entre los supercomputadores vectoriales (de millones de dólares) y los microprocesadores rápidos VLSI segmentados (de menos de 100.000 dólares). La razón básica fue el rápido decrecimiento del CPI de las máquinas escalares.

Las últimas tendencias en el diseño de máquinas vectoriales se centran en el alto rendimiento vectorial y el multiprocesamiento. Mientras tanto, las máquinas escalares de alta velocidad se concentran en mantener la relación entre el rendimiento máximo y el rendimiento sostenido cercana a la unidad. Por lo tanto, si las velocidades máximas (de pico) avanzan comparablemente, las velocidades sostenidas de las máquinas escalares avanzarán más rápidamente. Estas máquinas escalares pueden rivalizar o superar el rendimiento de las máquinas vectoriales con frecuencias de reloj comparables, especialmente para niveles de vectorización inferiores al 70%. Se pueden esperar máquinas escalares segmentadas de alta velocidad construidas con frecuencias de reloj que rivalizan con las de los supercomputadores vectoriales. Sin embargo, las máquinas vectoriales conservan ventajas en el rendimiento para problemas con vectores muy largos que puedan utilizar múltiples accesos a memoria y conseguir rendimientos próximos al máximo.