UNIVERSITAT POLITÈCNICA DE CATALUNYA

Departament d'Arquitectura de Computadors

RECURSOS ANCHOS: UNA TÉCNICA DE BAJO COSTE PARA EXPLOTAR PARALELISMO AGRESIVO EN CÓDIGOS NUMÉRICOS

Autor: David López Alvarez
Directores: Mateo Valero Cortés
Josep Llosa i Espuny

Capítulo 4.
Métodos de compactación

4.1 Introducción

En el capítulo anterior veíamos cómo la técnica de *widening* requería compactar operaciones para poderlas ejecutar en unidades funcionales anchas. En este capítulo veremos las técnicas que hemos desarrollado para realizar dicha compactación.

La compactación se puede realizar en tiempo de compilación (compactación estática) o en tiempo de ejecución (compactación dinámica). En nuestro conocimiento, no existe ningún trabajo previo que compacte operaciones dinámicamente para el uso de recursos anchos.

Respecto a la compactación estática, la única máquina que utiliza buses anchos tal y como los definimos nosotros es el IBM POWER2 [WhDh94] que aplica un grado de *widening* 2 y sólo en los buses. Su compilador, el *AIX Fortran 77 compiler*, es capaz de compactar operaciones. Desgraciadamente, el código de este compilador no es público, ni sus algoritmos de compactación, de manera que sólo hemos podido comparar resultados analizando el código

máquina generado por dicho compilador ante determinados códigos fuente y viendo si era capaz o no de compactar.

Respecto al uso de FPUs y banco de registros anchos, la única máquina anunciada es el AltiVec [MR98], un coprocesador multimedia para PowerPC, que implementa un banco de registros y una FPU de anchura dos. La información disponible a la hora de escribir estas líneas es más bien escasa. Las primeras informaciones indican que el código que sacará partido del AltiVec serán las bibliotecas gráficas, que se programan "a mano", de manera que el compilador no realizará ningún tipo de estudio de compactación.

Por otro lado existen las técnicas de vectorización, donde cada operación vectorial se podría ver como una operación ancha, con un ancho igual a la longitud de los vectores. Un procesador vectorial puede sacar provecho de *strides* diferentes de 1, al contrario de la técnica de *widening*, que sólo saca provecho de *stride* 1. Sin embargo, un procesador vectorial divide los bucles en una parte vectorizable, que se ejecuta en las unidades vectoriales, y una parte no vectorizable, que se ejecuta en las unidades escalares. La técnica de *widening* no tiene unidades anchas y unidades no anchas, sino que todos los recursos son de ancho n (siendo n el grado de *widening* de la arquitectura) y pueden ejecutar instrucciones de ancho 1 o de ancho n. Se propone compactar todas las operaciones posibles, obteniendo un bucle con operaciones compactadas y operaciones no compactadas y planificar después estas operaciones conjuntamente para una arquitectura VLIW, pudiendo aplicar técnicas de segmentación software donde se mezclen operaciones anchas y operaciones no anchas.

4.2 Compactación estática

Los nodos compactables suelen ser aquellos que representan una instanciación de la misma operación en iteraciones consecutivas, especialmente en operaciones aritméticas, aunque en operaciones de memoria lo que realmente marca la compactabilidad es el *stride*, que puede darse entre operaciones de la misma iteración, iteraciones sucesivas o iteraciones no

sucesivas. Por tanto, un paso previo que hay que realizar con los grafos es aplicar un grado de *unrolling* múltiplo del grado de compactación que se quiere realizar.

La idea básica es la siguiente: dos (o más) operaciones de memoria pueden compactarse siempre y cuando sean el mismo tipo de operación (load o store), no exista ninguna dependencia entre ellas y estén almacenadas consecutivamente en memoria (es decir, tengan *stride* 1). Dos (o más) operaciones aritméticas pueden compactarse siempre y cuando sean el mismo tipo de operación y no exista ninguna dependencia, directa o indirecta, entre ellas. Al no tener la limitación del *stride*, escogemos compactar diferentes instanciaciones de la misma operación en iteraciones sucesivas.

Es por tanto evidente que el punto principal de la compactación de operaciones de memoria radica en un buen estudio de los *strides* entre las operaciones de memoria. Por esta razón los grafos estudiados son extendidos (*extended dependence graph*, EDG) con información adicional sobre los *strides*, como se describió en el capítulo dos. En este punto describiremos el concepto de *unrolling*, a continuación veremos una técnica que proponemos para incrementar la información disponible sobre *strides*, que denominamos algoritmo de deducción de *strides*. Por último presentaremos nuestro algoritmo de compactación.

4.2.1 Unrolling

No es cierto que tengas treinta años de experiencia...
Simplemente has tenido la experiencia de un año 30 veces.

J.L. Carr. The Harpole Report. 1972

La técnica de unrolling replica el cuerpo de un bucle con paso **s** un número de veces **u** (llamado grado o factor de *unrolling*), e itera usando un paso **u*s** en lugar del paso original. Esta técnica permite una mejor planificación pues diversas iteraciones pueden ser planificadas conjuntamente incrementando el número de instrucciones disponibles para el compilador y dándole, por tanto, más oportunidades para realizar una planificación óptima. *Unrolling* también reduce el *overhead* causado por las instrucciones de control y la actualización de

índices. Se puede encontrar un estudio de los beneficios de unrolling para diferentes arquitecturas en [DoHi79].

La técnica de segmentación software usa *unrolling* para hacer coincidir el número de recursos disponibles en la arquitectura con los recursos requeridos por el bucle o para planificar bucles con MII fraccionario; por ejemplo, sea una arquitectura con dos sumadores y un bucle con cinco sumas, y supongamos que el sumador es el recurso más escaso, en este caso el MII es de 5/2 = 2.5 ciclos por iteración. La planificación se debe hacer con una tabla de dimensiones enteras, por tanto se ejecutará una nueva iteración cada 3 ciclos. Si aplicamos *unrolling* de grado 2, tendremos 10 sumas para las dos iteraciones y 2 sumadores, es decir 10/2=5 ciclos para 2 iteraciones y 2.5 ciclos por iteración. Sin embargo, y como ya hemos dicho, nosotros utilizamos la técnica de *unrolling* para poder disponer de tantas iteraciones como nuestro grado de *widening*, ya que las operaciones compactables suelen ser diferentes instanciaciones de la misma operación en iteraciones consecutivas.

El algoritmo de *unrolling* es muy conocido, pero hemos tenido que hacer una pequeña adaptación para los arcos de *stride* (el último paso del algoritmo). El algoritmo queda como puede verse en la figura 4.1.

- Sea **G** el grafo fuente, **u** el grado de *unrolling* y **G2** el grafo desenrollado resultante
- Para todo nodo original (**Nx**) del grafo **G**, se generan **u** nodos ($\mathbf{Nx_0}$ ... $\mathbf{Nx_{u-1}}$) en el grafo **G2**
- Para toda arista del grafo **G** que no sea de tipo *stride* entre los nodos **Nx** y **Ny** de distancia **d**, se generan **j**=0.. **u**-1 aristas entre los nodos $\mathbf{Nx_j}$ y $\mathbf{Ny_{((j+d) \bmod u)}}$, de distancia (**j**+**d** div **u**) en el grafo **G2**
- Para toda arista de tipo *stride* del grafo **G** entre los nodos **Nx** y **Ny** de distancia **d**, se generan **j**=0 .. **u**-2 aristas entre los nodos $\mathbf{Nx_j}$ y $\mathbf{Ny_{j+1}}$ de distancia **d** en el grafo **G2**

Figura 4.1: Algoritmo de unrolling

4.2.2 Algoritmo de deducción de strides

'You must not count on that' said Strider sharply.
'They will return. And more are coming. There are others. I know their number'

J.R.R. Tolkien. The Lord of the Rings. Book I, Chapter X.

Si el grafo de dependencias se ha calculado antes de realizar el *unrolling* (como suele pasar en la mayoría de los compiladores), se pueden deducir nuevos arcos de *stride*. La figura 4.2 muestra un grafo de dependencias extendido (EDG) donde puede deducirse un nuevo arco de *stride*. Analizando el código original (figura 4.2a), no se detecta ningún arco de *stride* 1, pero una vez desenrollado el código (figura 4.2c) un arco de *stride* 1 se ve claramente definido entre los nodos B^1 y A^0.

La idea principal del algoritmo de deducción de *strides* [Lop95] es que si un nodo **w** tiene dos arcos de *stride* como entrada (salida) desde (hacia) los nodos **u** y **v**, ambos arcos con distancia σ diferente, entonces existe un arco de *stride* entre estos dos nodos. Si los dos arcos tuvieran la misma distancia, serían la misma posición de memoria, pero como los loads/stores redundantes han sido eliminados del grafo de dependencias extendido (la herramienta que los ha generado, Ictíneo elimina loads/stores redundantes), esta situación no puede darse.

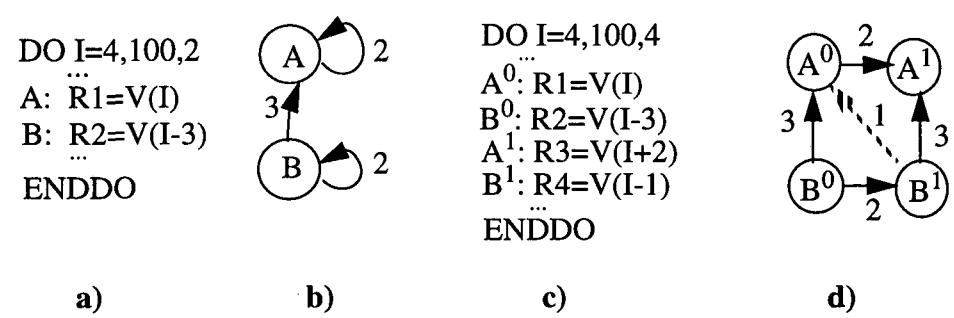

Figura 4.2: a) Código original y b) su EDG asociado (sólo arcos de stride). c) El mismo código tras aplicar un grado de unrolling u=2 y d) su EDG asociado con el nuevo arco deducido (flecha rayada).

El algoritmo de deducción de *strides* queda como puede verse en la figura 4.3. El algoritmo es iterativo, y se debe aplicar a todos los nodos de un grafo que cumplen las condiciones señaladas, hasta que en una iteración no se haya añadido ningún arco nuevo al grafo.

Para comparar nuestro algoritmo con uno ya existente, se probaron una serie de tests sintéticos donde nuestro algoritmo era capaz de detectar nuevos *strides*, con el código ensamblador generado por el compilador AIX Fortran77 versión 3.2 de un IBM SP2, basado en el procesador POWER2, que como ya hemos indicado, implementa buses de ancho dos. En ninguno de los casos probados el compilador de IBM fue capaz de compactar las operaciones con *stride* 1 deducible. En los bucles de nuestro juego de pruebas (los bucles del Perfect Club Benchmarks), el algoritmo propuesto fue capaz de deducir 492 nuevos arcos de *stride*, de los cuales 150 eran de distancia 1.

- Sean A, B y C tres operaciones de memoria con el mismo comportamiento (3 loads o 3 stores a la misma variable no escalar)
- caso 1: (un nodo A con dos arcos de salida a dos nodos diferentes B y C). Hay un arco de *stride* de A hacia B (A,B) con $\sigma_{(A,B)}$=m, y otro arco (A,C) con $\sigma_{(A,C)}$=n
  - caso (m>n): añadir un nuevo arco de *stride* (C,B), σ(C,B)=m-n si no existía
  - caso (m=n): no posible (el grafo está simplificado)
  - caso (m<n): añadir un nuevo arco de *stride* (B,C), $\sigma_{(B,C)}$=n-m si no existía
- caso 2: (un nodo C con dos arcos de entrada desde dos nodos diferentes A y B). Hay un arco de stride de (A,C) con $\sigma_{(A,C)}$=m, y otro arco (B,C) con $\sigma_{(B,C)}$=n
  - caso (m>n): añadir un nuevo arco de *stride* (A,B), σ(A,B)=m-n si no existía
  - caso (m=n): no posible (el grafo está simplificado)
  - caso (m<n): añadir un nuevo arco de *stride* (B,A), $\sigma_{(B,A)}$=n-m si no existía

Figura 4.3: Algoritmo de deducción de strides

4.2.3 Algoritmo de compactación

La unión de dos palabras que uno suponía
que no podían juntarse: eso es poesía.

Federico García Lorca

Definimos operaciones compactables como un número **n** de operaciones aritméticas o de memoria que pueden ejecutarse con una unidad funcional de ancho **n**.

Un número **n** de operaciones aritméticas ($\mathbf{a_0}$ .. $\mathbf{a_{n-1}}$) pertenecientes a un grafo **G** pueden compactarse si

- todas las operaciones $\mathbf{a_0}$ .. $\mathbf{a_{n-1}}$ son el mismo tipo de operación (suma, producto).
- $\forall$ $\mathbf{a_x}$, $\mathbf{a_y} \in \{\mathbf{a_0}$ .. $\mathbf{a_{n-1}}\}$ no existe ningún camino Φ $(\mathbf{a_x}, \mathbf{a_y}) \in \mathbf{G}$, es decir todas las operaciones son independientes entre sí, directa e indirectamente.

Un número n de operaciones de memoria ($\mathbf{m_0}$.. $\mathbf{m_{n-1}}$) pueden compactarse si se dan las dos condiciones anteriores y

- $\forall$ $\mathbf{m_x}$, $\mathbf{m_{x+1}} \in \{\mathbf{m_0}$ .. $\mathbf{m_{n-1}}\}$ $\exists$ un arco de *stride* $(\mathbf{m_x}, \mathbf{m_{x+1}})$ con $\sigma_{(mx,mx+1)}=1 \in \mathbf{G}$, es decir, los datos están ubicados en posiciones consecutivas de memoria.

Estas condiciones marcan la compactabilidad de las operaciones. El algoritmo de compactación es el descrito en la figura 4.4. Nuevamente comparamos los resultados de nuestro algoritmo de compactación con el código ensablador generado por el compilador de IBM AIX Fortran77 versión 3.2, encontrando que nuestro algoritmo conseguía compactar todos los casos detectados por el compilador de IBM, e incluso conseguíamos compactar alguna operación que el compilador no detectaba, debido a los arcos de *stride* deducidos. Puntualicemos que las pruebas sólo pudieron realizarse para compactación de operaciones de memoria, dado que el IBM POWER2 implementa buses anchos, pero no FPUs o banco de registros anchos.

- aplicar el algoritmo de deducción de *strides* a un grafo G'=EDG(G,S,σ)
- eliminar todos los arcos de stride con $\sigma_{(u,v)} \neq 1$
- marcar una cadena de operaciones compactables asumiendo recursos de anchura infinita
- dividir las cadenas de operaciones compactables en bloques de **n** elementos, uno detrás del otro

Figura 4.4: Algoritmo de compactación para recursos de anchura ***n***

4.3 Compactación dinámica

Toma los datos como vengan y haz lo que puedas con ellos.

Arthur C. Clarke. El martillo de Dios.

La compactación estática (en tiempo de compilación) requiere instrucciones especiales (instrucciones de datos anchos), con lo que el lenguaje máquina varía cuando se aplica la técnica de *widening*. No siempre se puede cambiar el lenguaje máquina de una familia de procesadores, pues puede tener todos los códigos de operación reservados. Pero, aunque se pueda cambiar, tendría el problema de la pérdida de compatibilidad binaria: un procesador debe ser capaz de ejecutar código máquina generado para una versión anterior de la misma máquina. Además, el cambio de lenguaje máquina hace perder la *backward compatibility*, es decir, un computador no puede ejecutar el código máquina generado para una versión posterior de la máquina.

Este problema puede solucionarse realizando la compactación en tiempo de ejecución (compactación dinámica), en la que el código máquina es el mismo y la compactación la hace un hardware especialmente diseñado a tal efecto. La ventaja es que el código es el mismo; si la máquina dispone de recursos anchos y del hardware especial, entonces sacará ventajas del uso de recursos anchos.

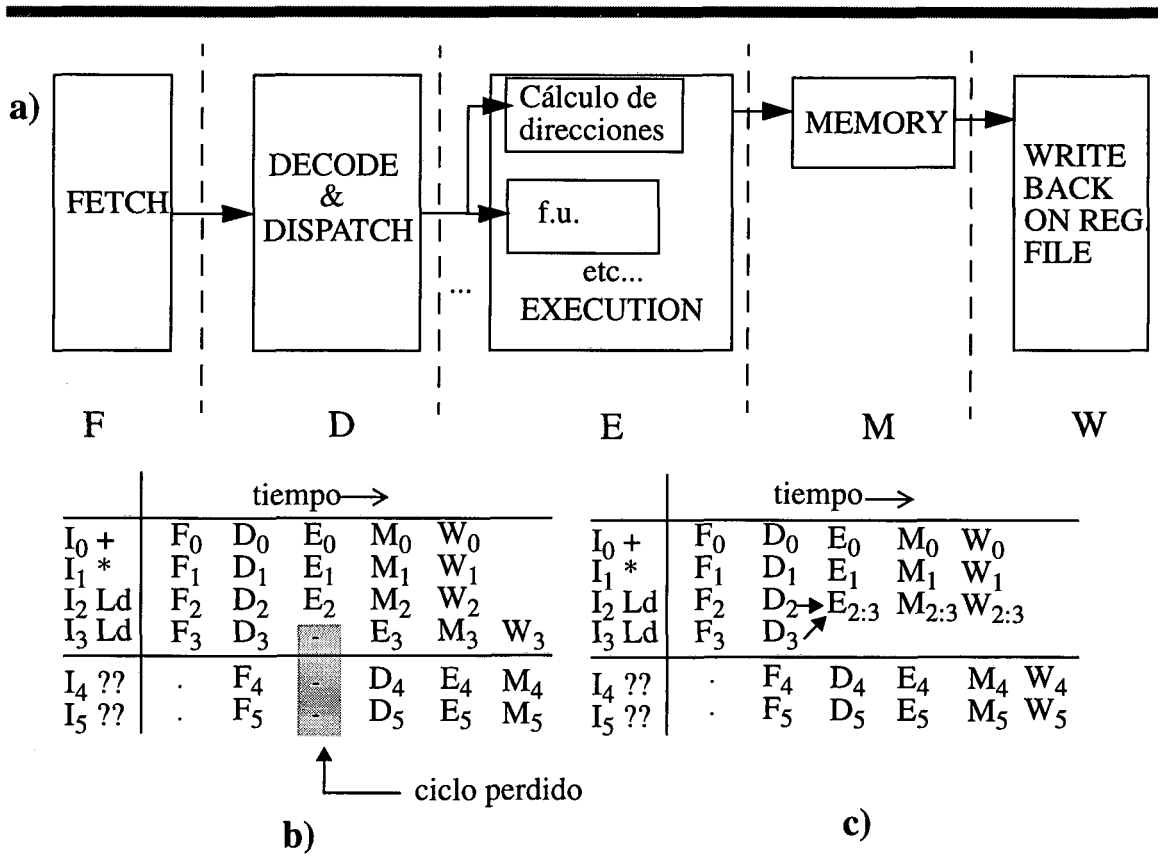

Figura 4.5: a) Segmentación básica de un procesador superescalar. Ejecución de instrucciones b) Sin hardware de compactación y c) Con hardware de compactación

Dos de las arquitecturas que más explotan ILP son las arquitecturas VLIW y las superescalares. En una arquitectura VLIW todo el estudio de la explotación del ILP se realiza en tiempo de compilación. Así pues este punto de estudio de compactación dinámica está centrado en arquitecturas superescalares y para explotar buses anchos [LVLA97].

La figura 4.5a muestra una segmentación básica de un procesador superescalar. Tomemos para ilustrar la idea un procesador de planificación estática capaz de lanzar (*issue*) 4 instrucciones por ciclo (similar al Alpha 21164), pero con una única unidad de memoria. Si dos operaciones de memoria son lanzadas en el mismo ciclo (como I_2 e I_3 en el ejemplo de la figura 4.5b), el load I_3 no puede ejecutarse (sólo disponemos de una unidad de traducción,

acceso a memoria etc...), de manera que se para la siguiente instrucción 1 ciclo, que se pierde. Si añadimos un hardware sencillo que permita compactar estas dos operaciones I_2 e I_3 en una única instrucción $I_{2;3}$ (figura 4.5c) que ejecuta las dos instrucciones simultáneamente, el siguiente bloque de instrucciones no se para, de manera que no se pierde ningún ciclo. Con esta propuesta, dos operaciones compactables siguen ocupando dos espacios en las instrucciones que se lanzan, pero requieren sólo una unidad de memoria (un puerto de TLB, una cache de un puerto, etc...).

Para mostrar el hardware de compactación, fijémonos primero en el modo de direccionamiento habitual en los procesadores RISC, que es el *base+desplazamiento*. De hecho, en algunos procesadores es el único modo disponible. Dos operaciones de load (o store) consecutivas tienen el mismo registro base (normalmente codificado con 5 bits) y dos desplazamientos (normalmente 16 bits) que difieren una cantidad *tamaño de operando*, que suele ser 4 u 8 bytes.

La figura 4.6 muestra el hardware necesario para detectar si dos accesos tienen el mismo registro base (r1=r2) y desplazamientos consecutivos (d1+tamaño=d2), en cuyo caso pueden ser compactados. Aún así se requiere que el compilador detecte estas operaciones y las planifique juntas, de manera que el hardware sea capaz de detectarlas y compactarlas.

Para evaluar esta propuesta, se compararon dos máquinas, una con instrucciones explícitas de memoria anchas (*Wide Machine* WM), cuyo juego de instrucciones había sido extendido, y otra máquina sin instrucciones explícitas para operaciones anchas, pero con el hardware especial (*Hardware Machine*, HM). Ambas máquinas han sido evaluadas con dos configuraciones diferentes, I4 e I8 con las siguientes características:

- La configuración I4 puede lanzar 4 instrucciones por ciclo (tiene un *issue* de 4), tiene 2 sumadores, 2 multiplicadores, un divisor/raíz cuadrada y un único bus. Los sumadores y los multiplicadores tienen una latencia de 2 ciclos y están perfectamente segmentados. El divisor/raíz cuadrada tiene un latencia de 16 ciclos para la división y

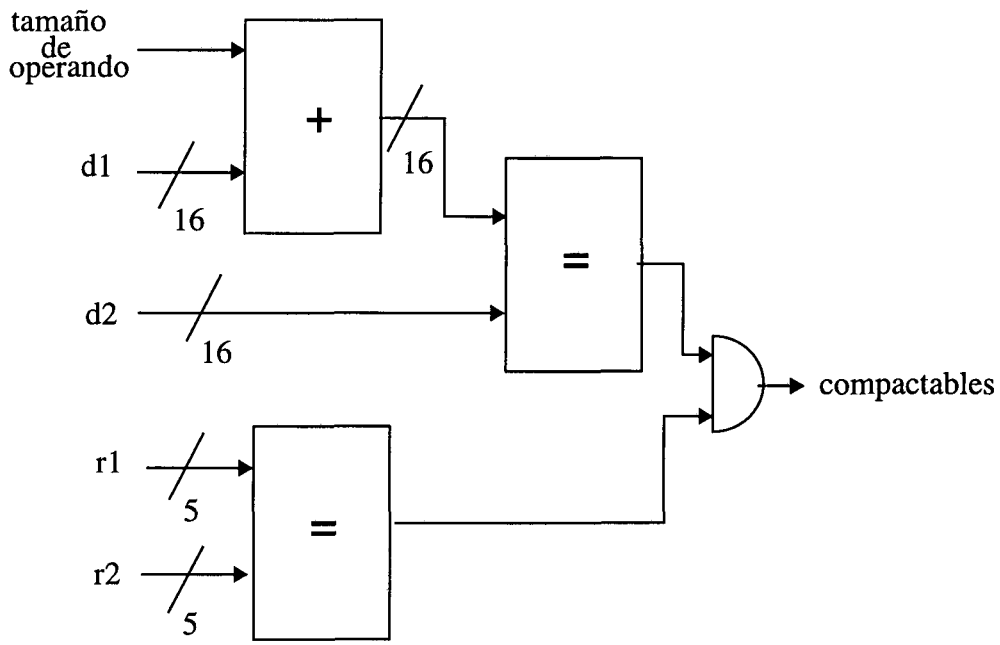

Figura 4.6: Hardware para detectar accesos compactables

30 para la raíz cuadrada, y ninguna de las operaciones está segmentada. El bus puede ejecutar un load en 2 ciclos (segmentado) y un store en un ciclo.

- La configuración I8 tiene un *issue* de 8 instrucciones (puede lanzar 8 instrucciones por ciclo) y tiene 4 sumadores, 4 multiplicadores, 2 divisores/raíz cuadrada y 2 buses, con idénticas características de latencia y segmentación de la configuración I4.

Los resultados de la evaluación de estas configuraciones puede verse en las gráficas de la figura 4.7. En la figura 4.7a se presenta el incremento de rendimiento para la configuración I4, en la que la configuración base tiene un bus de ancho 1, y se compara el incremento de rendimiento de tener un bus de ancho 2, con instrucciones explícitas (WM) y sin instrucciones

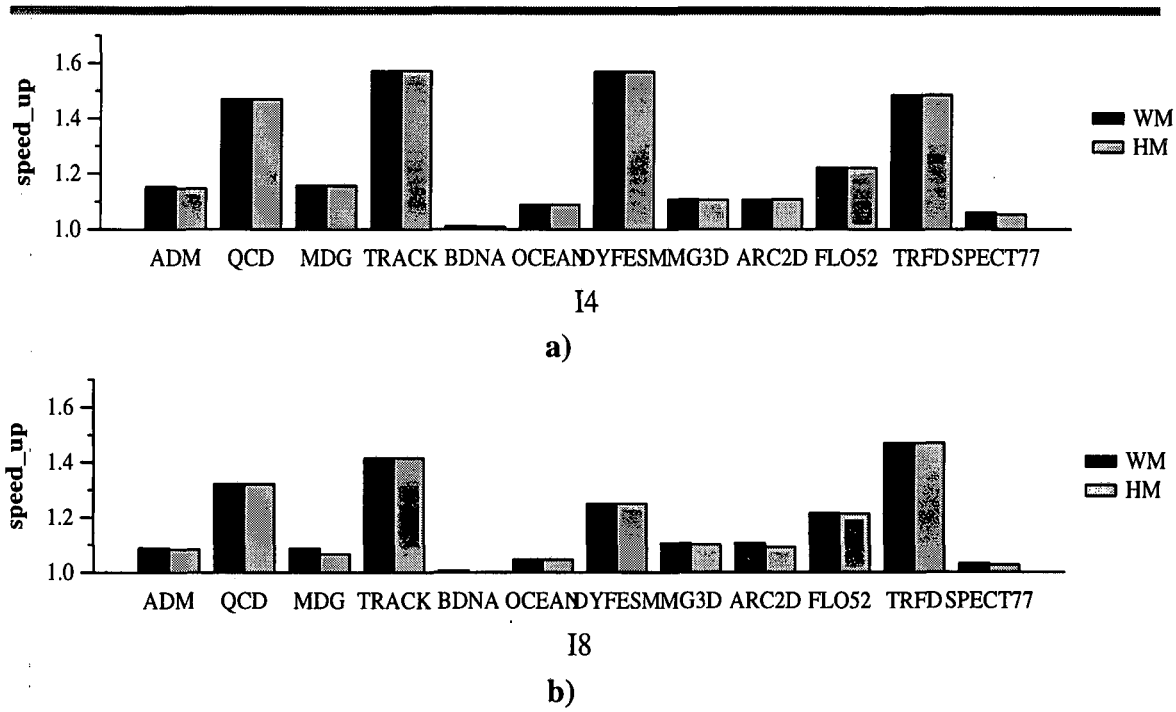

Figura 4.7: a) Incremento de rendimiento para la configuración I4 pasando de 1 bus de ancho 1 (configuración base) a 1 bus de ancho 2 con instrucciones explícitas (WM) y con el hardware propuesto (HM). b) La misma comparación para la configuración I8 al pasar de 2 buses de ancho 1 a 2 buses de ancho 2

explícitas, pero con el hardware de compactación (HM). En la figura 4.7b se presentan la misma comparación para la configuración I8, donde la configuración base tiene dos buses de ancho 1, y se presenta el rendimiento para dos buses de ancho 2. No comentaremos ahora el rendimiento de usar buses anchos (esto se hará en capítulos posteriores), sino que nos centraremos en las diferencias entre HM y WM. Podemos observar que, a pesar de que la máquina con hardware HM requiere un hueco (*issue slot*) más para lanzar operaciones compactables, el rendimiento de ambas máquinas es prácticamente el mismo. Sobre el total de ciclos de mejora en el rendimiento, la configuración con el hardware propuesto consigue el 99.89% del rendimiento de la máquina con instrucciones explícitas en la configuración I4, y el 97.74% en la configuración I8. Así concluimos que usando el hardware propuesto, máquinas de planificación estática pueden conseguir el mismo rendimiento de los buses anchos sin cambiar el lenguaje máquina y sin la perdida de compatibilidad que ello supone.

Un trabajo similar en ciertos aspectos fue presentado por Wilson, Olukotun y Roseblum [WOR96]. En este trabajo se proponían diversas técnicas para explotar buses anchos sin cambiar el juego de instrucciones. Las técnicas se basaban en encolar los loads y traer al procesador la línea de cache completa del dato pedido; los loads pendientes que requerían datos que estaban en dicha línea de cache eran servidos en aquel instante. Estas técnicas eran aplicadas en la fase de memoria y sólo se podían aplicar a procesadores de planificación dinámica. Además, estas técnicas continuaban necesitando la generación y traducción de diversas direcciones por ciclo. La técnica presentada permite compactar en la fase de decodificación/expedición (*decode/dispatch*) de manera que puede aplicarse a procesadores con planificación dinámica y con planificación estática, además de necesitar la generación y traducción de menos direcciones de memoria.

4.4 Sumario y contribuciones

La técnica *widening* requiere la compactación de operaciones simples (cada una opera sobre un dato) en operaciones anchas (cada una opera sobre varios datos) para sacar partido de los recursos anchos. Dicha compactación puede realizarse en tiempo de compilación (compactación estática) y de ejecución (compactación dinámica). En este capítulo hemos presentado técnicas para realizar ambos tipos de compactación.

La compactación estática suele realizarse juntando operaciones que son instanciaciones de la misma operación en iteraciones consecutivas, con lo que es necesario aplicar la técnica de *unrolling*. Dado que la compactación estática requiere disponer de información sobre *strides*, se aplicó unrolling sobre el *Extended Dependence Graph*, una representación de grafos (explicada en el capítulo 2) que incluye dicha información. Esto nos forzó a hacer una adaptación del algoritmo de *unrolling* para poderlo aplicar sobre arcos de *stride*.

Como la compactación estática necesita la máxima información posible sobre *strides*, hemos desarrollado un algoritmo para deducir nuevos arcos de *stride* en el grafo. También se ha presentado en este capítulo un algoritmo para realizar la compactación.

Respecto a la compactación dinámica, se ha presentado un hardware que permite compactar operaciones en tiempo de ejecución, en un entorno de máquinas superescalares con planificación estática, sin necesidad de cambiar el juego de instrucciones (y por tanto sin pérdida de compatibilidad) que ofrece un rendimiento prácticamente similar a una máquina con instrucciones anchas explícitas.

Las principales aportaciones de este capítulo son:

- Algoritmo de deducción de *strides*, que permite aumentar la información sobre *strides* del grafo, dando más oportunidades al compilador de compactar operaciones.
- Algoritmo de compactación de operaciones, tanto de memoria como aritméticas.
- Propuesta de un hardware para compactación dinámica en máquinas superescalares con planificación estática.