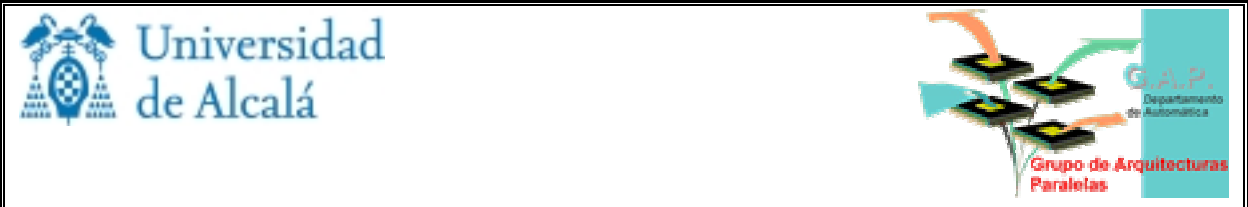


Aplicación de la teoría de grafos al análisis del paralelismo a nivel de instrucción

Nota técnica TN-UAH-AUT-GAP-2005-01-es

Raúl Durán, Rafael Rico

Departamento de Automática, Universidad de Alcalá, España

Enero 2005

English version:

On Applying Graph Theory to ILP Analysis

Technical Note TN-UAH-AUT-GAP-2005-01-en

Raúl Durán, Rafael Rico

Department of Computer Engineering, Universidad de Alcalá, Spain

January 2005

Resumen:

La evaluación de las arquitecturas de computadores requiere nuevas herramientas que complementen las habituales simulaciones. En este sentido, la teoría de grafos tradicional puede ayudar a crear un nuevo marco de análisis del paralelismo de grano fino. Las diferencias encontradas entre el rendimiento superescalar en procesadores x86 y no-x86 y las características peculiares de la arquitectura del repertorio x86 recomiendan realizar un estudio exhaustivo del paralelismo disponible en la capa del lenguaje máquina.

Partiendo de nociones básicas de la teoría de grafos se introducen conceptos nuevos tales como la *valencia reducida* para llegar a la *matriz de dependencias de datos D* que caracteriza de manera matemática una secuencia de código. Dicha matriz cumple una serie de propiedades y restricciones y describe cuál es la disposición del código para ser procesado concurrentemente. Entre otros detalles, se establece una relación entre la longitud del camino crítico y el grado de paralelismo y se ofrecen técnicas para calcularlo a partir de la propia matriz *D*.

Finalmente se muestra cómo las diferentes fuentes de dependencias de datos se pueden componer facilitando un modo de analizar su influencia final en el grado de paralelismo. Se ilustra la técnica con un ejemplo del que se obtienen algunas conclusiones.

Palabras clave: Evaluación de arquitecturas de computadores, paralelismo a nivel de instrucción, arquitectura del repertorio de instrucciones, teoría de grafos, cuantificación basada en DDG.

1. Introducción: nuevos retos en evaluación de arquitectura de computadores.

La evaluación cuantitativa es un punto crucial en la investigación de arquitectura de computadores. Tal y como explica Kevin Skadron *et al.* la simulación se ha convertido en la primera herramienta de evaluación tanto en la industria como en la investigación [35]. Desafortunadamente, la construcción de buenos simuladores y la selección de cargas de trabajo adecuadas se han convertido en tareas hercúleas. Estas dificultades han conducido a una situación en la que únicamente se trabaja en temas donde las herramientas tienen una calidad contrastada abandonando temas de investigación realmente interesantes como, por ejemplo, el multiprocesamiento.

Casi contemporáneo con el trabajo referido en el párrafo anterior y presentado también en *IEEE Computer*, Dror G. Feitelson señala que normalmente se confía en las medidas del rendimiento para comparar la calidad de varios sistemas sin reparar en que las diferencias pueden provenir de la propia metodología de evaluación [14]. Concretamente, concluye que la métrica y la carga de trabajo usadas pueden afectar a los resultados ya que son susceptibles de interactuar entre si.

Otros aspectos a tener en cuenta en la evaluación cuantitativa deberían ser los siguientes [35]:

- el lenguaje de programación secuencial en el que están escritos los simuladores no contribuye a la correcta descripción de los sistemas paralelos que se pretende evaluar
- los programas de prueba más utilizados (SPEC) no están convenientemente caracterizados
- el análisis estadístico empleado comúnmente (promedio, desviación estándar, etc.) no es riguroso ya que el comportamiento a ráfagas de muchos procesos relacionados con la computación no generan distribuciones gaussianas
- los resultados presentados en las publicaciones no suelen ser verificados independientemente.

Respecto a la verificación independiente, Skadron apunta como causa el hecho de no estar convenientemente recompensada. A nosotros nos gustaría añadir que además de ello, muchas veces es imposible reproducir los experimentos ya que no se proporciona suficiente información sobre el simulador, su carga de trabajo, las suposiciones hechas o la parametrización. En este sentido, sería conveniente alcanzar, al menos, el mismo grado de "reproducibilidad" que observamos en otros campos de investigación.

En consecuencia, parece que ha llegado el momento de reajustar el rumbo para corregir la tendencia y para hacer frente a los nuevos retos surgidos en la evaluación de las arquitecturas de computadores. En el trabajo de Skadron se hace referencia a algunas de las recomendaciones que enumeraron un grupo de investigadores reunidos en Diciembre de 2001 con el objetivo de tratar estos temas. De todas ellas, a nosotros nos ha parecido especialmente interesante la de proponer métodos analíticos.

Estamos de acuerdo con Skadron en que se detecta cierta hostilidad hacia los métodos analíticos en congresos y revistas. Esto no debería ser así ya que el modelo analítico contribuye a comprender aspectos que no son detectables mediante simulación. Es más, el modelo analítico puede servir para validar el simulador y para predecir el comportamiento de ciertas propuestas arquitectónicas.

Tal y como ocurre en otros campos de investigación, la formalización matemática facilita la descripción de los fenómenos, permite predecir comportamientos, favorece la "reproducibilidad" y simplifica la transferencia de conocimientos.

a. Aplicación de la teoría de grafos al análisis del paralelismo de grano fino.

Hemos considerado conveniente dar los primeros pasos para aplicar la teoría de grafos al análisis del paralelismo de grano fino. La teoría de grafos nos proporciona una formalización matemática eficiente que promete ser muy útil en el modelado analítico que nos hemos marcado como reto impulsados por los trabajos antes mencionados. Por otra parte, los grafos ya han sido aplicados con éxito al estudio de otros aspectos de la computación como, por ejemplo, el paralelismo de grano medio o grueso extraído por los compiladores [3, 45, 46]. En este sentido, Padua y Wolfe llegan a afirmar que el código paralelo será tan bueno como lo sea el grafo de dependencias de datos correspondiente [27]. Otros campos en los que trabajamos con grafos son: las estructuras de datos [2, 3, 8, 20], el diseño de sistemas operativos [33], la descripción de software [10, 11], la lógica de los automatismos [28], el diseño electrónico [9], etc.

Sin embargo, hay más razones para abordar este tema. En primer lugar, el estudio del paralelismo de grano fino ha sido y es un tema de gran relevancia, donde la simulación es la técnica de evaluación usada con más frecuencia. Quizá por ello necesite, más que otros temas, de un modelado analítico. En segundo lugar, normalmente, cuando se identifican los factores que contribuyen al paralelismo de grano fino, se olvida el impacto de la arquitectura del repertorio de instrucciones haciendo referencia tan sólo a aspectos físicos. Quizá el motivo sea nuevamente el simulador: es más fácil modelar aspectos físicos que modelar el comportamiento de diferentes características del repertorio de instrucciones. Nosotros hemos constatado una significativa diferencia entre el grado de paralelismo de grano fino establecido en la literatura para procesadores x86 y no-x86. Esto nos ha llevado a pensar que quizá se ha minimizado el impacto de la arquitectura del repertorio de instrucciones sobre el paralelismo de grano fino disponible.

b. Diferencia ILP entre procesadores x86 y no-x86.

Efectivamente, como se ha mencionado anteriormente, la cuantificación del paralelismo a nivel de instrucción es uno de los temas más populares en arquitectura de computadores. Algunas revisiones describen el espacio de soluciones hardware como, por ejemplo, la de Jouppi y Wall [19] o la de Smith y Sohi [37]. En la literatura se pueden encontrar numerosos estudios identificando factores limitantes, cuantificando sus efectos, aportando posibles soluciones y evaluando los resultados. Entre los factores evaluados tenemos: la pérdida de paralelismo debida a los saltos condicionales [31], la predicción de saltos con una unidad de búsqueda ideal y para una ventana de instrucciones de 32 entradas [36], el tamaño de la ventana de instrucciones bajo renombramiento, predicción de saltos y diversos tamaños del banco de registros [42, 43], las interrupciones precisas [7], las limitaciones del flujo de control [22], la desambiguación de memoria [40], el rendimiento de los predictores de saltos híbridos [12], el uso de multihebra para aumentar la ocupación de recursos físicos [41], la presión sobre el banco de registros bajo los programas del SPEC92 y para un buffer de reordenación de 256 entradas [13], el número de puertos de memoria en combinación con el tamaño del buffer de reordenación y del banco de registros [15], la predicción de datos [23, 44], el impacto de los programas del SPEC95 [29], el efecto del reordenamiento del código sobre la predicción de saltos [30], la liberación temprana de registros [24], por citar solamente algunos de los más importantes.

Todos estos trabajos tienen en común que presentan procesadores no-x86, que la evaluación se ha realizado mediante simuladores y que los factores limitantes identificados se refieren siempre a la capa física. Los resultados IPC promedio referidos se encuentran en el rango 2,5-15 pero con picos alrededor de 30 IPC (por ejemplo, en el caso de desambiguación de memoria [40]) y con un límite superior bajo condiciones ideales de 50 IPC (caso presentado en [15]).

Los trabajos donde se ha utilizado procesadores de repertorio x86 son menos frecuentes. En este caso, los niveles de paralelismo referidos no son tan buenos como los reseñados en el párrafo anterior. El grupo de Y. N. Patt ha propuesto técnicas tales como la planificación segmentada [38] o la anticipación de búsqueda de instrucciones [25] sobre procesadores x86 alcanzando unos valores promedio de IPC en el rango entre 0,5 y 3,5 en las mejores situaciones, con la mayor parte de los valores algo por encima de 1. Huang y Xie miden el paralelismo a nivel de microoperaciones (MLP) [18]. El MLP promedio es 1,32 sin renombramiento y 2,20 con renombramiento. Bhandarkar y Ding caracterizan el rendimiento del Pentium Pro mediante los contadores hardware que incluye el propio procesador [5]. El CPI para los SPECint95 está en el rango 0,75 a 1,6.

Parece claro que el paralelismo disponible en procesadores x86 es menor que el obtenido en procesadores no-x86 a la luz de los artículos que podemos estudiar en la literatura sobre el tema. Esto nos hace pensar que la propia arquitectura del repertorio de instrucciones (ISA) puede suponer un importante factor limitante del paralelismo de grano fino disponible.

c. El repertorio x86 y el modelo superescalar.

Con el fin de salvaguardar la compatibilidad binaria con los procesadores previos, lo cual ha proporcionado innegables beneficios, la arquitectura del repertorio de instrucciones x86 ha heredado características de diseño adecuadas con requerimientos del pasado pero claramente perjudiciales en el ámbito del procesamiento superescalar. Concretamente, podríamos citar entre las características indeseables desde el punto de vista de la explotación del paralelismo las siguientes: el uso dedicado de registros, los operandos implícitos (ligados a la operación y no especificados por el programador), el uso de registro de estado y el gran número de registros involucrados en la aritmética de direcciones.

El efecto de estas características indeseables se manifiesta en la sobre ordenación del código a través de dependencias impuestas en la capa del lenguaje máquina pero que no son estrictamente necesarias para preservar el significado computacional de la tarea compilada. En consecuencia, las instrucciones llegan a la capa física más acopladas de lo que originalmente correspondería observando el programa de alto nivel [32].

La arquitectura del repertorio de instrucciones tiene un impacto significativo en la disponibilidad de paralelismo de grano fino antes de llegar a la capa física que puede disminuir el grado de paralelismo explotable en tiempo de ejecución. La Fig. 1 ilustra de manera esquemática los factores que afectan al paralelismo disponible en cada capa del proceso de computación.

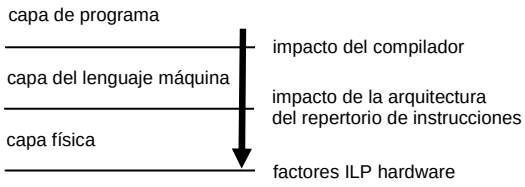


Fig. 1. Factores que afectan al paralelismo disponible en las diferentes capas del proceso de computación.

En la actualidad, los estudios acerca de la arquitectura del repertorio de instrucciones son escasos y se orientan más bien a máquinas específicas como, por ejemplo, los procesadores DSP [34], a ofrecer herramientas de simulación [26] u otros aspectos colaterales.

En cuanto a los trabajos centrados en el repertorio de instrucciones x86, los análisis en la capa del lenguaje máquina se limitan a realizar recuentos y calcular frecuencias de uso. Adams y Zimmerman realizaron un estudio sobre la frecuencia de uso de instrucciones x86 en aplicaciones DOS [1]. Sin embargo, este trabajo no incluye el uso de operandos. Más cercano en el tiempo, Huang y Peng han realizado recuentos del uso de instrucciones con diferentes operandos [17]. No obstante, no se detienen a analizar las dependencias entre operandos. Finalmente, Huang y Xie presentan un estudio evaluando el paralelismo a nivel de microoperaciones (núcleo RISC de los procesadores CISC de Intel) donde se tiene en cuenta la operación y los modos de direccionamiento [18].

d. Métricas.

La métrica empleada de forma más habitual en los trabajos de cuantificación de paralelismo a nivel de instrucción es el IPC (instrucciones por ciclo). Se trata de hallar la razón del recuento de instrucciones frente al tiempo de ejecución (simulado) en ciclos. Es por ello que se requieren simuladores complejos, si se desea que sean precisos, en los que las suposiciones y simplificaciones afectan significativamente al resultado final. Las medidas dependen fuertemente de las características de la implantación física y, por tanto, ésta métrica es adecuada al estudio de las diferentes propuestas arquitectónicas de la capa física.

Existe una métrica ligeramente distinta a la anterior en la que el grado de paralelismo es la razón del número total de instrucciones ejecutadas frente a las rejillas de tiempo planificadas. En cualquier caso, también está orientada al estudio de la organización de los recursos físicos.

Cuando se trata de medir otra clase de eventos distinta de la ejecución de instrucciones como, por ejemplo, fallos de caché, fallos de predicción de saltos, etc., utilizamos un mecanismo similar: la tasa respecto al total.

Hay que hacer notar que la métrica basada en IPC lleva asociado habitualmente un tratamiento estadístico que, como hemos recordado más arriba, tiene su propio impacto en los resultados. Generalmente se dan valores promedio o se usa la desviación estándar sin tener en cuenta una característica intrínsecamente ligada a la computación: que el paralelismo se presenta a ráfagas, como señala Kumar [21]. Así pues, éstos resultados estadísticos no resultan útiles, como nos recuerda el trabajo de Skadron [35], ya que raras veces los eventos medidos se ajustan a una distribución de Gauss.

Una métrica mucho menos utilizada es la que consiste en medir la longitud del camino crítico de una secuencia de código. El camino crítico es la mayor cadena de dependencias de datos que podamos encontrar entre instrucciones de dicha secuencia. Así, el número mínimo de pasos de computación necesario para procesar la secuencia, si se dispone de recursos físicos suficientes, será igual a esa longitud. El grado de paralelismo máximo será entonces igual al número total de instrucciones procesadas entre dicha longitud, dando idea de la cantidad de instrucciones que se pueden procesar de forma concurrente en cada paso de computación.

El algoritmo utilizado para medir las cadenas de dependencias anota qué instrucción produce cada una de las escrituras y lecturas en cada ubicación de datos y seguidamente cuenta el número de “eslabones” escritura-lectura correlativos.

La cuantificación del paralelismo a partir de la longitud del camino crítico ha sido utilizada previamente por Kumar. En este caso, el estudio se realizó sobre código fuente escrito en FORTRAN tomando sentencias en lugar de instrucciones y variables en lugar de ubicaciones físicas de datos [21]. Es, por tanto, un trabajo que podemos situar en la capa de programa. Resulta muy interesante ya que obtiene un grado de paralelismo muy superior al que luego encontramos en la literatura para los trabajos relativos a la capa física. Sin duda el propio proceso de computación degrada el paralelismo disponible.

El camino crítico también se ha usado alguna vez para evaluar características de la capa física. Concretamente, los trabajos de Austin y Sohi [4], Postiff, Greene, Tyson y Mudge [29] y Stefanovic y Martonosi [39] obtienen sus resultados en función de la longitud del camino crítico. Estos estudios parten de trazas de código ejecutado realmente sobre las que se realiza la medición aplicando ciertas reglas que modelan las características propias del hardware que desean evaluar. Los resultados generados así asumen, por tanto, las especificaciones de la capa física.

Aunque el método de medida sobre el camino crítico permite realizar una cuantificación del grado de paralelismo en la capa de lenguaje máquina, independientemente de las restricciones de la capa física, no se encuentra en la literatura ningún trabajo así. Sería interesante contar con datos acerca de la posible degradación del paralelismo en esta capa.

Un método de medida alternativo es el que se basa en los grafos de dependencias de datos (DDG). Se trata de construir los DDG a partir de secuencias de código real. La caracterización, en este caso, es independiente de la implementación física. Se sitúa en un escalón anterior en el proceso de computación: en la capa del lenguaje máquina. Sin embargo, incorpora el impacto del proceso de compilación. Estas medidas dan idea de cómo ha de ser el hardware para aprovechar el orden parcial especificado por el grafo sin imponer más restricciones al proceso de ejecución fuera de orden.

La cuantificación basada en DDG es una potente herramienta de análisis cuando se utiliza la representación matricial ya que, en este caso, permite realizar un procesamiento matemático. Así, además de la longitud del camino crítico y, en consecuencia, el grado de paralelismo de una ventana de instrucciones, también podemos determinar el tiempo de vida de los operandos, la reutilización o compartición de los datos, las fuentes de dependencias más relevantes, la

distribución del paralelismo y otros parámetros importantes. Asimismo, la formalización matemática nos va a permitir componer diferentes fuentes de dependencias de datos con el fin de encontrar el origen del acoplamiento del código.

Finalmente, los DDG también pueden incorporar las especificaciones de la capa física empleando descripciones formales del hardware.

2. Representación de secuencias de instrucciones mediante grafos. Una revisión.

Las dependencias de datos en una secuencia de instrucciones pueden ser representadas mediante un grafo *G(V, E)*, donde *V* es el conjunto de vértices y *E* es el conjunto de arcos. Cada vértice en *V* representa una instrucción y cada arco en *E* una dependencia de datos. Dos vértices relacionados por un arco se dice que son adyacentes. La topología de un grafo puede representarse mediante una matriz llamada *matriz de adyacencia A*:

$$a_{ij} = \begin{cases} 1, & \text{si los vértices } i \text{ y } j \text{ son adyacentes;} \\ 0, & \text{en el resto} \end{cases} \quad (1)$$

La matriz *A* es simétrica de dimensión *n* x *n* donde *n* es el número de instrucciones en el grafo, con la diagonal nula y $a_{ij} \in \{0, 1\}$ [6, 16].

Definimos la *matriz de incidencia B* como:

$$b_{ij} = \begin{cases} 1, & \text{si } e_j \text{ incide sobre el vértice } v_i; \\ 0, & \text{en el resto} \end{cases} \quad (2)$$

Si el grafo tiene *n* vértices y *m* arcos, la dimensión de *B* es *n* x *m*.

Este grafo admite dos posibles orientaciones: aquella en la que "la instrucción *i* produce datos para *j*" (identificada por σ) o aquella en la que "la instrucción *j* consume datos (depende) de *i*" (identificada por $\bar{\sigma}$). En cada una los arcos apuntan en sentidos opuestos con un significado complementario: la primera orientación muestra el flujo de datos mientras que la segunda manifiesta las dependencias de datos. La Fig. 2 propone una sencilla secuencia de código x86 (subconjunto de 16 bits) e ilustra ambas orientaciones del grafo.

```
0: XOR BX, BX
1: MOV SI, 07AB
2: MOV BL, ES:[SI]
3: INC SI
4: CMP BL, DL
5: JNE/JNZ IP+F7
```

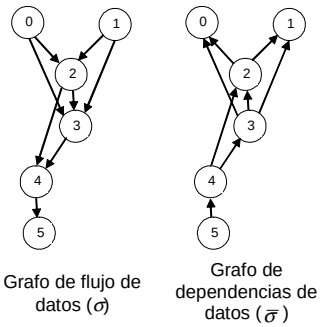


Fig. 2. Secuencia de código ejemplo y las dos posibles orientaciones del grafo asociado.

Podemos definir la matriz de incidencia B^{σ} con respecto a la orientación σ, como la matriz *n* x *m*:

$$b_{ij}^{\sigma} = \begin{cases} +1, & \text{si } v_i \text{ es el extremo entrante de } e_j; \\ -1, & \text{si } v_i \text{ es el extremo saliente de } e_j; \\ 0, & \text{en el resto} \end{cases} \quad (3)$$

Definimos *valencia* de un vértice como el número total de arcos que inciden en dicho vértice. La *matriz de valencias* Δ es una matriz *n* x *n* diagonal donde la componente *(i, i)* es la valencia del vértice *i*. La relación entre la matriz de adyacencia y la de incidencia para una orientación σ viene dada por:

$$Q = B^{\sigma} \cdot (B^{\sigma})^{t} = \Delta - A \quad (4)$$

El producto $B^{\sigma} \cdot (B^{\sigma})^{t}$ se conoce como *matriz Laplaciana Q* y es independiente de la orientación.

Por otra parte, la representación de grafos basada en la matriz de adyacencia *A* cuenta con el valor añadido por las propiedades del polinomio característico dado por $\det(\lambda I - A)$ que es un aspecto central en teoría de grafos [6, 16].

En la Fig. 3 se dan la matriz de adyacencia *A*, las matrices de incidencia para ambas orientaciones B^{σ} y $B^{\bar{\sigma}}$ y la matriz de valencias Δ correspondientes al ejemplo propuesto.

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$B^{\sigma} = \begin{pmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$B^{\bar{\sigma}} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

$$\Delta = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 3. Matrices *A*, B^{σ}, $B^{\bar{\sigma}}$ y Δ correspondientes al ejemplo propuesto.

Obsérvese como las matrices de adyacencia y de valencias son invariantes con la orientación mientras que la de incidencia si depende de ella. Las dos primeras vienen dadas por la relación de incidencia mientras que la tercera depende de la relación de incidencia y del sentido de los arcos.

a. Valencia reducida.

Definimos *valencia reducida* de un vértice como el número total de arcos que entran a dicho vértice. La valencia reducida depende, por tanto, de la orientación seleccionada.

La *matriz de valencias reducida* a la orientación σ, V^{σ}, es una matriz n x n diagonal donde la componente (i, i) es la valencia reducida a la orientación σ del vértice i.

Igualmente cabe pensar en una definición especial para la matriz de incidencia considerando una única orientación. Así podemos definir la *matriz de incidencia reducida* I^{σ} con respecto a la orientación σ, como:

$$i_{ij}^{\sigma} = \begin{cases} +1, & \text{si } v_i \text{ es el extremo entrante de } e_j; \\ 0, & \text{en el resto} \end{cases} \tag{5}$$

Si el grafo tiene n vértices y m arcos, la dimensión de I^{σ} es n x m.

Proposición 1: el producto $I^{\sigma} \cdot (I^{\sigma})^t$ genera la matriz de valencias reducida V^{σ} para la orientación seleccionada.

Demostración: Si calculamos la componente del producto:

$$\left[I^{\sigma} \cdot \left(I^{\sigma} \right)^t \right]_{ij} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{kj}^{\sigma^t} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\sigma} \tag{6}$$

Ahora bien, $i_{ik}^{\sigma} \cdot i_{jk}^{\sigma} \neq 0$ si y solo si $i = j$, pues cada arco incide sobre un único vértice. Además, como $i_{ik}^{\sigma} \in \{0, 1\}$, se cumple $(i_{ij}^{\sigma})^2 = i_{ij}^{\sigma}$:

$$\left[I^{\sigma} \cdot \left(I^{\sigma} \right)^t \right]_{ij} = \begin{cases} \sum_{k=0}^{m-1} i_{ik}^{\sigma} = \begin{cases} \text{nº de extremos} \\ \text{entrantes si } i = j; \end{cases} \\ 0, \text{ si } i \neq j \end{cases} \tag{7}$$

Resultado que coincide con la definición de la matriz de valencias reducida a una única orientación.

$$V^{\sigma} = I^{\sigma} \cdot (I^{\sigma})^t \tag{8}$$

■

La Fig. 4 presenta las matrices de incidencia reducida y las matrices de valencia reducida para ambas orientaciones correspondientes a la secuencia de código ejemplo.

$$I^{\sigma} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$I^{\bar{\sigma}} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$V^{\sigma} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$V^{\bar{\sigma}} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Fig. 4. Matrices de incidencia reducida y matrices de valencia reducida para ambas orientaciones: I^{σ}, $I^{\bar{\sigma}}$, V^{σ} y $V^{\bar{\sigma}}$.

La inspección de la Fig. 3 y la Fig. 4 nos permite comprobar como la matriz de incidencia se puede construir a partir de las de incidencia reducida y la de valencia a partir de las de valencia reducida. Tendremos:

$$B^{\sigma} = I^{\sigma} - I^{\bar{\sigma}} \tag{9}$$

$$\Delta = V^{\sigma} + V^{\bar{\sigma}} \tag{10}$$

Las ecuaciones anteriores formalizan una relación bastante intuitiva que no entraremos a demostrar aquí.

Por otra parte, podemos observar que los valores de la matriz de valencia reducida V^{σ} corresponden a la suma por filas de las componentes de I^{σ} lo cual se corresponde con lo expresado en la ecuación (7). La interpretación es inmediata: si la valencia reducida es el total de extremos entrantes a cada vértice bastará con practicar un recuento haciendo un recorrido por filas de la matriz I^{σ} para calcularlo. Lo mismo podemos decir para la orientación contraria $\bar{\sigma}$.

b. La matriz de dependencias de datos *D*.

Definimos la *matriz de dependencias de datos D* como:

$$d_{ij} = \begin{cases} 1, & \text{si la instrucción } i \text{ depende de } j; \\ 0, & \text{en el resto} \end{cases} \tag{11}$$

Así, la instrucción i está asociada con un vector de dependencias de datos $\vec{d}_i$ cuya j-ésima componente es 1 si existe dependencia directa con la instrucción j a través de algún dato y 0 en caso contrario. Así, la matriz D es el conjunto de todos los vectores de dependencias de datos $\vec{d}_i$ de una secuencia de código. Queremos recalcar que la matriz D representa el camino de dependencia de datos directo o de longitud unidad, es decir, la instrucción i consume un dato procesado directamente por la instrucción j sin intermediarios.

La Fig. 5 muestra la matriz de dependencias de datos para la secuencia de código ejemplo.

$$D = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

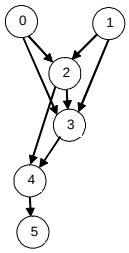


Fig. 5. Matriz de dependencias de datos *D* y grafo de flujo de datos.

La matriz *D* valora el arco solamente en el extremo positivo mientras que la matriz *A* valora ambos extremos. En este sentido la orientación del grafo que manifiesta en primera instancia es la que corresponde al flujo de datos.

Aunque la matriz *D* se usa comúnmente en teoría de compiladores (por ejemplo, en transformaciones de bucles [45, 46]) no ha sido utilizada en el análisis del lenguaje máquina ni en la caracterización del paralelismo a nivel de instrucción. Los autores opinan que el motivo puede ser que los análisis de las arquitecturas de los repertorios de instrucciones se consideran superados o que se asume que su influencia en la ejecución superescalar es menor confiando en exceso en la capa física.

c. Propiedades topológicas y restricciones en ILP para la matriz de dependencias de datos.

Uno de los objetivos del álgebra de la teoría de grafos es determinar de manera precisa cómo se reflejan las propiedades de los grafos en las propiedades algebraicas de esas matrices. Nosotros pretendemos extraer, además, consecuencias en el ámbito del procesamiento paralelo de instrucciones.

• **El etiquetado de los vértices no debe afectar a las propiedades de la matriz de dependencias de datos**. La matriz *D* puede asociarse a un grafo que consta de un conjunto *V* de vértices, un conjunto *E* de arcos y una relación de incidencia que es un subconjunto de *V* x *E*. El conjunto $\{v_0, v_1, v_2, \ldots, v_{n-1}\}$ corresponde a un etiquetado arbitrario de los vértices. Esto significa que las matrices asociadas al grafo pueden sufrir permutaciones de filas y columnas en función del etiquetado. Estamos interesados en aquellas propiedades que permanecen invariantes bajo dichas permutaciones.

En el ámbito del paralelismo a nivel de instrucción, debemos considerar que el punto de partida es una serie de instrucciones que conserva la secuencialidad natural del modelo de programación de Von Neumann fijado por el estricto orden de precedencia en el que están escritas en el programa. El etiquetado original de los vértices del grafo (instrucciones) es el que induce el propio contador de programa. A este etiquetado le llamaremos *etiquetado programático*.

El *etiquetado programático* nos va a facilitar descubrir propiedades que, sin embargo, se cumplen bajo cualquier permutación.

• **Entre los vértices del grafo de dependencias de datos existe una relación de precedencia**. Cualquier tarea computable conlleva alguna relación de precedencia u ordenación parcial entre las tareas (instrucciones) a realizar, propia de un proceso que se desarrolla en una sucesión ordenada y finita de pasos. Es consustancial al algoritmo un cierto ordenamiento que, a veces, se acentúa al pasar de una capa a otra en el proceso de computación tal y como se ilustra en la Fig. 1 (del algoritmo al programa, del programa a la imagen compilada en código máquina, del código máquina a la capa física).

• **Una instrucción no depende de sí misma**. Un dato no puede tener la misma instrucción como fuente y como destino. Como consecuencia de esto, la matriz *D* tiene la diagonal nula. Es decir, $d_{ii} = 0 \quad 0 \le i \le n-1$.

Esto es cierto incluso en los bucles. Un bucle es una manera compacta de escribir una secuencia de código que, en su versión desarrollada, repite una serie de operaciones pero sobre datos diferentes. Cada nueva iteración supone una nueva instancia del cuerpo del bucle pero sobre datos nuevos. Las operaciones del cuerpo del bucle requieren una instrucción de salto condicional entre una iteración y la siguiente. En ese caso, la instrucción de salto condicional puede insertarse en el grafo de flujo de datos como una operación especial que manipula el contador de programa haciendo que las instrucciones del cuerpo del bucle queden separadas en cada nueva instanciación.

• **Las dependencias de datos no son simétricas**. Una instrucción no puede depender de otra que depende de ella al mismo tiempo ya que esta situación no establece una relación de precedencia sino un ciclo de dependencias de datos. Como consecuencia, la matriz *D* no es simétrica. Matemáticamente $d_{ij} \neq d_{ji} = 1 \quad \forall i, j \le n-1$.

Por tanto, si *D* no es simétrica, el polinomio característico asociado a cualquier matriz de dependencias de datos $p(G; \lambda) = \det(\lambda I - D)$, sea cual sea la relación de incidencia, es siempre el mismo: $p(G; \lambda) = \lambda^n$. El valor descriptivo de este polinomio es muy pequeño.

• **Existe un etiquetado de los vértices del grafo bajo el cual la matriz *D* es triangular inferior.** Una instrucción depende sólo de las precedentes en el programa, nunca de las siguientes (principio de causalidad). Esto significa que las instrucciones solamente procesan datos entregados por las instrucciones escritas más arriba en el programa nunca de las que están por venir en la secuencia. Según esto, el *etiquetado programático* de los vértices del grafo de dependencias genera una matriz *D* triangular inferior ya que $d_{ij} = 0$ siempre que $j > i$. O expresado de otra manera, un etiquetado que se ajuste al orden impuesto por el contador de programa nos asegura que todas las componentes *(i, j)* de *D* son 0 cuando $j > i$ ya que una instrucción no puede tener dependencias de datos con las siguientes. A la matriz de dependencias de datos cuyo etiquetado de vértices la convierte en triangular

inferior la denominaremos *forma canónica de D* o *matriz D canónica* (D_c).

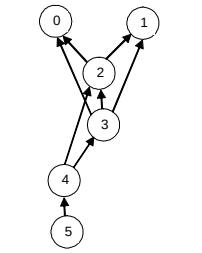
Por todo esto y a modo de resumen enumeramos a continuación las propiedades que cumple la matriz de dependencias de datos *D*:

- es cuadrada de dimensión *n* x *n* siendo *n* igual al número de instrucciones incluidas en el grafo de la secuencia de código
- sus valores son binarios, es decir, $d_{ij} \in \{0, 1\}$
- tiene la diagonal nula
- no es simétrica y, en consecuencia, su polinomio característico, sea cual sea la relación de incidencia del grafo, es igual a λ^n
- existe al menos un etiquetado de los vértices (instrucciones) –*etiquetado programático*– bajo el cual es triangular inferior y da lugar a la forma canónica D_c
- los diferentes etiquetados de los vértices representan permutaciones que dan lugar a isomorfismos
- encierra una ordenación parcial de las instrucciones

d. La matriz de dependencias de datos y la matriz de adyacencia.

Existe una relación inmediata entre la orientación que genera el grafo de dependencias de datos y la que genera el grafo de flujo de datos. Ambos comparten el mismo conjunto de arcos (relación de incidencia) variando únicamente la orientación. Podemos asegurar que si "*i* depende de *j*"entonces "*j* produce datos para *i*". Se cumple que si $(d_{ij})^{\sigma} = 1$ bajo una orientación $(d_{ji})^{\bar{\sigma}} = 1$ bajo la contraria, lo cual corresponde a la definición de matriz traspuesta. Obsérvese en la Fig. 6 como la matriz D^t valora los extremos positivos del grafo correspondiente a la orientación de las dependencias de datos.

$$D^t = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



Grafo de dependencias de datos

Fig. 6. Matriz D^t y la orientación del grafo valorada.

Como la matriz de adyacencia *A* valora ambos extremos de cada arco entonces *A* es la suma de la matriz de dependencias *D* más su traspuesta D^t, es decir:

$$A = D + D^t \tag{12}$$

A partir de *D* podemos llegar a construir la matriz de adyacencia *A*. Esto nos permite trabajar con la representación matricial completa del grafo y disponer de toda la potencia algebraica asociada a su *polinomio característico*.

e. La matriz de dependencias de datos y la valencia reducida.

Que la matriz de dependencias de datos valore únicamente un extremo de cada arco sugiere algún tipo de relación con la valencia reducida. Asimismo, la inspección de la matriz *D* en la Fig. 5 y de la matriz I^{σ} en la Fig. 4 nos permite apreciar cierta similitud en la disposición de los valores no nulos. En realidad ha de ser así pues en ambas subyace el mismo grafo: en I^{σ} relacionando arcos y vértices mientras que en *D* se relacionan vértices con vértices. La misma formalización matemática que hemos desarrollado apunta en la misma dirección ya que las ecuaciones (9), (10) y (12) sugieren una relación entre la matriz *D* y la incidencia reducida I^{σ} a través de la ecuación (4).

Proposición 2: el producto $I^{\sigma} \cdot I^{\bar{\sigma}^t}$ genera la matriz *D*.

Demostración 1: Realizando la sustitución de las ecuaciones (9), (10) y (12) en la ecuación (4) obtenemos:

$$(I^{\sigma} - I^{\bar{\sigma}}) \cdot (I^{\sigma} - I^{\bar{\sigma}})^t = V^{\sigma} + V^{\bar{\sigma}} - D - D^t \tag{13}$$

Operando ahora con el primer término de la ecuación anterior:

$$\begin{aligned} &(I^{\sigma} - I^{\bar{\sigma}}) \cdot (I^{\sigma^t} - I^{\bar{\sigma}^t}) = \\ &= I^{\sigma} \cdot I^{\sigma^t} + I^{\bar{\sigma}} \cdot I^{\bar{\sigma}^t} - I^{\sigma} \cdot I^{\bar{\sigma}^t} - I^{\bar{\sigma}} \cdot I^{\sigma^t} \end{aligned} \tag{14}$$

Sabemos que la ecuación (8) relaciona la valencia reducida con la incidencia reducida de forma que podemos simplificar el resultado anterior:

$$I^{\sigma} \cdot I^{\bar{\sigma}^t} + I^{\bar{\sigma}} \cdot I^{\sigma^t} = D + D^t \tag{15}$$

Si *D* valora los extremos positivos de la orientación del grafo de flujo de datos y D^t los de la orientación del grafo de dependencias de datos, tendremos:

$$I^{\sigma} \cdot I^{\bar{\sigma}^t} = D \tag{16}$$

■

Demostración 2: Si calculamos la componente del producto:

$$\left[I^{\sigma} \cdot \left(I^{\bar{\sigma}} \right)^t \right]_{ij} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{kj}^{\bar{\sigma}^t} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\bar{\sigma}} \tag{17}$$

El sumatorio realiza un barrido de todos los arcos en el índice *k*. El producto es distinto de cero únicamente para el *k*-ésimo arco si entra en el vértice *i* ($i_{ik}^{\sigma} = 1$) saliendo del vértice *j* o, lo que es lo mismo, entrando al vértice *j* bajo la orientación contraria ($i_{jk}^{\bar{\sigma}} = 1$). Ahora bien, esto coincide con la

definición de la matriz de dependencias de datos D que hemos hecho en la ecuación (11).

$$\sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{jk}^{\bar{\sigma}} = d_{ij} \tag{18}$$

De donde:

$$D = I^{\sigma} \cdot I^{\bar{\sigma}^t} \tag{16}$$

■

Ya se ha visto cómo la valencia reducida para una orientación se puede obtener haciendo el recorrido por filas contando los extremos entrantes (componente igual a 1) tal y como sugiere la ecuación (7). A la vista de la relación expresada en la ecuación (16) y de la similitud en la disposición de los valores no nulos de D e I^{σ} cabe pensar en utilizar la matriz D para encontrar la valencia reducida.

Proposición 3: el recuento de arcos recorriendo la matriz D por filas permite generar la matriz de valencia reducida para la orientación del grafo de flujo de datos.

Demostración: Supongamos que la relación es cierta y sustituyamos la componente de D por su valor según la ecuación (18):

$$v_{ii} = \sum_{p=0}^{n-1} d_{ip} = \sum_{p=0}^{n-1} \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot i_{pk}^{\bar{\sigma}} \tag{19}$$

Podemos intercambiar los sumatorios sin que se altere el resultado final y sacar el término que no depende de los índices:

$$v_{ii} = \sum_{k=0}^{m-1} \sum_{p=0}^{n-1} i_{ik}^{\sigma} \cdot i_{pk}^{\bar{\sigma}} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot \left(\sum_{p=0}^{n-1} i_{pk}^{\bar{\sigma}} \right) \tag{20}$$

Ahora bien, dado el k-ésimo arco solamente hay un vértice sobre el que tenga un extremo entrante y se cumple:

$$v_{ii} = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \cdot 1 = \sum_{k=0}^{m-1} i_{ik}^{\sigma} \tag{21}$$

Lo cual se corresponde con la ecuación (7) y demuestra la equivalencia entre efectuar el recorrido por filas en la matriz I^{σ} y hacerlo en la matriz D para calcular los recuentos de arcos entrantes, permitiendo dar una nueva definición para la matriz de valencia reducida:

$$v_{ij}^{\sigma} = \begin{cases} \sum_{k=0}^{n-1} d_{ik} = \text{nº de arcos si } i = j; \\ 0, \text{ si } i \neq j \end{cases} \tag{22}$$

■

Como vemos, la matriz de dependencias de datos D exhibe una gran capacidad descriptiva ya que permite:

- recuperar la matriz de adyacencia A y su polinomio característico
- calcular la valencia reducida para la orientación del grafo de flujo de datos
- referir el grafo de dependencias de datos mediante su traspuesta
- obtener información acerca de la secuencia de código sin utilizar ningún otro instrumento matemático

f. Acoplamiento del código.

El concepto de valencia reducida es muy útil en el ámbito del procesamiento paralelo de instrucciones. Recordamos que si la orientación σ seleccionada es aquella en la que los arcos entrantes señalan el consumo de los datos (grafo de flujo de datos), la *valencia reducida* cuantifica lo acoplada (ligada) que se encuentra una instrucción con las demás, es decir, hace referencia a lo trabada que está una secuencia de código. Una *valencia reducida* grande, en este caso, indica que una instrucción consume datos procedentes de varias instrucciones y que, por tanto, debe esperar a que todos ellos estén disponibles para poder ejecutarse. Consecuentemente, si el acoplamiento es grande la ordenación parcial del código es potencialmente mayor al establecerse más relaciones de precedencia. Si la ordenación parcial se acentúa el paralelismo disponible disminuye.

La diagonal de la matriz de valencias reducida para la orientación correspondiente al grafo de flujo de datos (V^{σ}) nos informa del acoplamiento que tiene cada instrucción.

La traza de la matriz de valencias reducida para la orientación correspondiente al grafo de flujo de datos es el número total de arcos, es decir, cuantifica la cantidad de relaciones de dependencias entre instrucciones. Este valor nos da información de lo acoplada (ligada, trabada) que está una secuencia de código. A este valor lo denominamos *acoplamiento* C:

$$\mathrm{C} = \mathrm{tr}\, V^{\sigma} \tag{23}$$

El sumatorio por filas en la matriz D nos da la valencia reducida de cada instrucción para la orientación del grafo de flujo de datos (22). Ahora bien, ya hemos señalado que existe un etiquetado de los vértices según el cual la matriz D es triangular (forma canónica D_c). En ese caso y sabiendo, además, que la diagonal es nula tenemos:

$$v_{ij}^{\sigma} = \begin{cases} \sum_{k=0}^{i-1} d_{c_{ik}} = \text{nº de arcos si } i = j \neq 0; \\ 0, \text{ en el resto} \end{cases} \tag{24}$$

En consecuencia, la traza de la matriz de valencias reducida para la orientación correspondiente al grafo de flujo de datos (acoplamiento C) será:

$$\mathrm{C} = \mathrm{tr}V^{\sigma} = \sum_{i=0}^{n-1} v_{ii}^{\sigma} = \sum_{i=1}^{n-1}\sum_{k=0}^{i-1} d_{c_{ik}} \qquad (25)$$

El número máximo de dependencias de datos (arcos) en el grafo viene dado por todas las posibles parejas ordenadas de vértices, es decir, es la cantidad máxima de posibles combinaciones de n elementos tomados de 2 en 2 (coeficiente binomial $\binom{n}{2}$). Luego el acoplamiento C está acotado por:

$$0 \le \mathrm{C} \le \binom{n}{2} = \frac{n^2 - n}{2} \qquad (26)$$

Con el fin de obtener una medida del acoplamiento independiente de la cantidad de instrucciones en la secuencia, definimos acoplamiento normalizado, C_N, como la razón de C frente al número de instrucciones n de la secuencia. Cuando C_N es nulo no hay ninguna dependencia mientras que en el peor de los casos cada instrucción depende de todas las precedentes.

$$0 \le \mathrm{C}_N \le \frac{n-1}{2} \qquad (27)$$

g. Reutilización de datos.

Si la orientación es la contraria $\bar{\sigma}$, es decir, los arcos son entrantes para señalar a quien se entregan los datos (grafo de dependencias de datos), la valencia reducida cuantifica la reutilización de los datos. Si los datos generados por una instrucción son utilizados por muchas otras éstos se han de conservar afectando al consumo de recursos destinados al almacenamiento temporal de variables.

La diagonal de la matriz de valencias reducida para la orientación correspondiente al grafo de dependencias de datos ($V^{\bar{\sigma}}$) nos informa de la reutilización de los datos producidos por cada instrucción. Cuando este valor es 0 significa que el dato no es consumido. Si el valor es 1 significa que cada dato generado por una instrucción únicamente se consume en otra. Sin reutilización, el tiempo de vida de los datos en los elementos de almacenamiento es mínimo. Existe una situación especial que se produce cuando todos los valores no nulos de la diagonal de $V^{\bar{\sigma}}$ son iguales a 1. Esta situación se ilustra en la Fig. 7.(a). Matemáticamente se manifiesta en que el producto $D \cdot D^t$ genera una matriz diagonal que coincide con V^{σ}. La explicación es inmediata: para los elementos de fuera de la diagonal todos los productos $d_{ik} \cdot d_{jk}$ se anulan ya que si la instrucción i depende de la k ninguna otra dependerá de k.

Cuando el valor de la reutilización es mayor que 1 significa que varias instrucciones consumen un dato. Bajo esta circunstancia, resulta interesante conocer el tiempo de vida (almacenamiento) de los datos. En la Fig. 7.(b) y (c) se ilustran dos posibles situaciones. En ambos casos la instrucción 0 genera datos para la 1 y la 2. En el caso (b) los datos son consumidos en el siguiente paso de computación mientras que el caso (c) no es posible ya que existe un camino de dependencias de longitud 2 que relaciona también la instrucción 2 con la 0. Deducimos que el tiempo de vida de un dato producido por i y consumido por j debe ser al menos igual al camino de dependencias más largo entre ambos. Por esa razón hablaremos de tiempo de vida mínimo $T_{mín}$.

$$T_{mín} = n \quad \text{tal que } [D^n]_{ij} \neq 0 \text{ y } [D^{n+1}]_{ij} = 0 \qquad (28)$$

El caso de la Fig. 7.(d) nos hace ver que este tiempo depende finalmente de los criterios de planificación (capa física) ya que la instrucción 1 puede planificarse un paso de computación antes que la instrucción 3 o a la vez que la 0 con lo que el tiempo de almacenamiento sería de 2 pasos de computación.

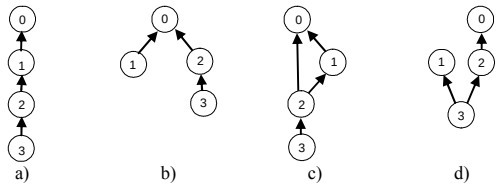


Fig. 7. Diferentes tipos de reutilización de datos.

En el caso de la reutilización no tiene sentido calcular su valor para el grafo completo de manera similar a como hemos hecho para el acoplamiento C ya que los recursos de almacenamiento atienden a las necesidades individuales de los datos. Por otra parte, calcular la traza de $V^{\bar{\sigma}}$ no es más que un recuento total de arcos en el grafo –por columnas de D en lugar de filas como en la ecuación (24)– que coincide con al acoplamiento C.

h. Caminos de dependencias de datos de longitud mayor que 1.

Por definición, un camino de longitud l (arcos) en un grafo $G(V, E)$, desde el vértice v_i hasta el v_j, es una secuencia finita de $l + 1$ vértices distintos que comienza en v_i y termina en v_j tal que dos vértices consecutivos son adyacentes [6, 16].

La matriz de dependencias de datos D muestra los caminos de dependencias de longitud 1 o dependencias directas entre instrucciones. Sin embargo, pueden existir cadenas de dependencias de datos entre instrucciones de longitud mayor. Por ejemplo, si $d_{ij} = 1$ y $d_{jk} = 1$ entonces la instrucción i depende de k a través de la j con un camino de longitud 2. En general, podemos afirmar que existe un camino de dependencias de longitud 2 desde la instrucción i hasta la j que pasa a través de, al menos, una de las instrucciones presentes en el grafo si se cumple:

$$(d_{i0} \cdot d_{0j}) + (d_{i1} \cdot d_{1j}) + \cdots + (d_{in-1} \cdot d_{n-1j}) = \\ = \sum_{k=0}^{n-1} (d_{ik} \cdot d_{kj}) \neq 0 \qquad (29)$$

que corresponde a la componente *(i, j)* de la matriz producto $D \cdot D$. Según esto, la matriz D^2 representa los caminos de dependencias de datos de longitud 2.

• **D^l representa los caminos de dependencias de datos de longitud *l* (arcos)**. Generalizando lo explicado para el camino de longitud 2, cada potencia de la matriz *D* representa los caminos de dependencias de longitud igual al grado de la potencia. Dicho de otra forma, el número de caminos de dependencias de datos de longitud *l* en *G(V, E)*, según una orientación σ, desde v_i hasta v_j es el valor de la componente *(i, j)* de la matriz D^l. En [6] podemos encontrar una demostración por inducción a este último enunciado basada en la matriz de adyacencia *A* que se puede extender al caso de la matriz de dependencias de datos *D*. Nótese que la longitud se mide en arcos.

• **La *n*-ésima potencia de *D* es nula**. La longitud máxima de un camino de dependencias de datos es $n - 1$ (arcos) siendo *n* el número de instrucciones de la secuencia de código. De esta manera, D^n será nula necesariamente.

• **No hay ciclos de dependencias**. No hay caminos de dependencias cerrados ya que este caso no establece una relación de precedencia. En consecuencia, el grafo es acíclico (DAG o *Directed Acycled Graph*). En realidad, en un ciclo, una instrucción depende de sí misma a través de otras y eso significa que una tarea no tiene solución en un número finito de pasos. Algebraicamente esto se manifiesta en que una instrucción no puede depender de sí misma bajo ningún camino de longitud *l*. Por tanto, la diagonal de cualquier potencia de la matriz de dependencias (D^l) es nula: $d^l_{ii} = 0 \quad 1 \leq l \leq n - 1$.

Visto desde el ángulo opuesto, si la *n*-ésima potencia de *D* no es nula significa que hay ciclos de dependencias y, por tanto, la matriz *D* no representa una tarea computable (solucionable en un número finito de pasos).

Resumiendo, en cuanto a los caminos de dependencias de datos de longitud mayor que la unidad, podemos destacar:

- cada potencia de la matriz *D* representa los caminos de dependencias de longitud igual al grado de la potencia (medida en número de arcos)
- D^n debe ser nula necesariamente ya que en caso contrario no representa una tarea computable
- no hay ciclos de dependencias y, por tanto, la diagonal de cualquier potencia de *D* es nula

i. Longitud del camino crítico y grado de paralelismo.

Uno de los datos más importantes que podemos extraer de las matrices de dependencias de datos es el grado de paralelismo a nivel de instrucción del que disponen las secuencias de código por ellos representadas. Dicha información es independiente de las limitaciones que posteriormente pueda imponer la capa física. Es un dato que atañe sólo a la capa del lenguaje máquina y que tiene que ver con el algoritmo empleado, la plasmación del mismo en un programa, la influencia del compilador y la propia arquitectura del repertorio de instrucciones.

El paralelismo disponible está inversamente relacionado con la longitud de las cadenas de dependencias entre instrucciones. Cuanto más largas son estas cadenas más fuerte es la ordenación parcial de la secuencia de código imponiendo una ejecución muy secuencial de las instrucciones con capacidad limitada de procesamiento concurrente. Por el contrario, las cadenas de dependencias cortas implican un ordenamiento débil entre instrucciones susceptible de ser aprovechado para la ejecución concurrente.

Definimos *longitud del camino crítico* L*(D)* (abreviadamente L) de una secuencia de código representada por su matriz de dependencias de datos *D*, como la longitud del camino de dependencias más largo. Ahora bien existen dos posibles métricas: la cuantificación en arcos –expuesta en la sección precedente conforme a la teoría de grafos tradicional– o la cuantificación en pasos de computación –más propia del ámbito de la computación a nivel de instrucción–.

Existe una relación inmediata entre las dos métricas de la longitud del camino crítico L y el número de vértices involucrado. Si la longitud del camino crítico L involucra $l + 1$ vértices, entonces dicho camino tiene *l* arcos y el número mínimo de pasos de computación requeridos para procesar la secuencia de código asociada es $l + 1$. La ejecución de instrucciones no dependientes en cada paso de computación libera una dependencia de la cadena más larga. Después de *l* pasos de computación, estaremos en disposición de procesar la última o últimas instrucciones (liberadas de dependencias) de la secuencia. Necesitamos, por tanto, $l + 1$ pasos de computación para terminar de ejecutar las $l + 1$ instrucciones de la cadena del camino crítico.

Sabemos que D^l representa los caminos de dependencias de datos de longitud *l* (arcos). Según esto la primera potencia de *D* que sea nula señala la longitud del camino crítico. Esto es:

$$\mathrm{L} = l \quad \text{pasos de computación sii} \quad D^l = 0 \qquad (30)$$

Bajo esta métrica, L está acotado de la siguiente forma:

$$1 \leq \mathrm{L} \leq n \qquad (31)$$

La longitud L mínima es 1. Esto significa que no existen dependencias de datos entre las instrucciones y, por tanto, en caso de que los recursos disponibles lo permitan, se pueden procesar todas de forma concurrente en un único paso de computación. El

valor máximo de L es *n*. En ese caso, sólo es posible lanzar una instrucción en cada paso de computación y la secuencialidad es total requiriendo *n* pasos de computación para finalizar el procesamiento del código.

Cuantas más instrucciones se incluyan en el grafo de dependencias de datos la longitud del camino crítico L será potencialmente mayor. Con el fin de obtener una medida del paralelismo a nivel de instrucción independiente de la cantidad de vértices del grafo, definimos la *longitud del camino crítico normalizada*, L_N, como la razón de la longitud del camino crítico (L) expresada en pasos de computación frente al valor máximo de L ($L_{máx}$).

$$L_N = \frac{L}{L_{máx}} = \frac{L}{n} \tag{32}$$

Cuando L_N se aproxima a 1 no hay paralelismo, y cuanto más cerca de 0 esté su valor, mayor será el paralelismo encerrado en el código.

$$L_N \in (0,1] \tag{33}$$

También definimos grado de paralelismo, G_p, como el recíproco de L_N expresado en pasos de computación ($G_p = L_N^{-1}$). G_p va desde 1 (ausencia de paralelismo) hasta *n* (máximo grado de paralelismo).

$$G_p \in [1, n] \tag{34}$$

El significado de G_p es inmediato. Indica el número de instrucciones que se pueden procesar de forma concurrente en cada paso de computación. Si en un paso de computación puedo lanzar las *n* instrucciones de la secuencia significa que no existen dependencias de datos entre ellas y se pueden procesar de forma concurrente. Por el contrario, si en cada paso de computación únicamente puede procesar 1 instrucción el secuenciamiento es total.

A la secuencia de código de la Fig. 2 le corresponde un grafo con una longitud del camino crítico igual a 4 arcos o 5 pasos de computación y con un grado de paralelismo G_p de 5/6, es decir, en cada paso de computación se procesan 5/6 de instrucción. El ejemplo resulta bastante secuencial ($G_p = 0{,}83 \approx 1$), en realidad, tan solo en el primer paso de computación somos capaces de lanzar 2 instrucciones de manera concurrente.

Veamos un punto interesante. El grado de paralelismo es función del número de instrucciones presentes en la secuencia de código, es decir, $G_p = f(n)$. Es razonable pensar que según se incrementa *n* también crece G_p sin límite. De este modo, a mayor número de instrucciones en la secuencia mayor será el potencial paralelismo. Siguiendo esta lógica las ventanas de instrucciones de algunos procesadores superescalares se han hecho mayores en la esperanza de encontrar más instrucciones independientes listas para ser ejecutadas simultáneamente. En contra de esta idea, las simulaciones de David Wall encontraron un comportamiento asintótico [42, 43]. Es decir, llega un momento en que, a pesar de examinar más instrucciones, no se encuentran independientes. Visto desde otro punto, la longitud de las cadenas de dependencias crece cuanto mayor sea la secuencia de código ya que los recursos lógicos de la arquitectura software del repertorio de instrucciones son limitados.

j. Calculo algorítmico de la longitud del camino crítico.

El cálculo de las sucesivas potencias de la matriz *D* nos permite determinar la longitud del camino crítico mediante un procedimiento algebraico según expresa la ecuación (30). Sin embargo, el cálculo basado en el producto matricial es muy pesado (complejidad $O(n^4)$ con operaciones de producto) y hace este método impracticable.

El algoritmo para encontrar la ordenación parcial que encierra un grafo sirve también para hallar la longitud del camino crítico. Este procedimiento tiene un menor coste de computación (complejidad $O(n^2)$ con operaciones de suma) que lo hace atractivo para el análisis automático de secuencias de código. Consiste en establecer la precedencia entre instrucciones en función de sus dependencias. En cada paso de computación hacemos un listado de las instrucciones no dependientes y liberamos de dependencias a aquellas a las que entregan datos. Todas las instrucciones que pertenecen a la lista de un paso de computación comparten el mismo nivel de precedencia y, por tanto, pueden procesarse de forma concurrente. El número de pasos de computación requerido para ordenar todas las instrucciones es la longitud del camino crítico. Por otra parte, la ordenación parcial extraída establece una lista de planificación de las instrucciones que puede utilizarse para asignar recursos físicos. La figura siguiente ilustra el pseudocódigo de este algoritmo.

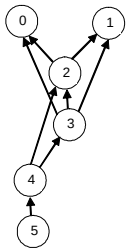
```
/* Procedimiento de ordenación parcial */
1: GIVEN: paso de computación paso = 0, lista de instrucciones para
el paso p Lista(p), matriz de dependencias D;

2: while (quedan instrucciones por ordenar)
3: {
4:      while (quedan instrucciones no dependientes)
5:      {
6:              Lista(p) = AñadirIndependienteListaPaso(p);
7:      }
8:      EliminarDependencias (Lista(p), D);
9:      paso ++;
10: }
```

Fig. 8. Algoritmo de ordenación parcial.

Este algoritmo nos permite tanto conocer la longitud del camino crítico (expresado en pasos de computación) como elaborar el listado del orden parcial.

La Fig. 9 muestra el orden parcial asociado al grafo ejemplo. Véase como dicho orden parcial permite, si hay recursos físicos suficientes, procesar la secuencia de código origen del grafo de dependencias

de datos en 5 pasos de computación dando lugar a un grado de paralelismo $G_p = 5/6 = 0{,}83$ instrucciones por paso de computación.



Paso de computación 0: 0, 1
Paso de computación 1: 2
Paso de computación 2: 3
Paso de computación 3: 4
Paso de computación 4: 5

Fig. 9. Ordenación parcial correspondiente al grafo ejemplo.

3. Composición de fuentes de dependencias

Si una secuencia de código puede representarse por la matriz de dependencias de datos D y ésta es una formalización de la ordenación parcial encerrada en el código, podemos analizar con precisión el impacto de las diferentes fuentes de dependencias ya que estas se componen de una manera sencilla para generar el mapa de dependencias final. Si D_{s1} es la matriz debida a una fuente de dependencias y D_{s2} es la matriz debida a otra fuente de dependencias, la matriz D resultante será:

$$D = D_{s1} \text{ OR } D_{s2} \tag{35}$$

De otra manera, $d_{ij} = d_{s1ij} \text{ OR } d_{s2ij}$ significa que la instrucción i depende de la j si depende por la causa 1 o si depende la causa 2 o por ambas causas.

La Fig. 10 ilustra lo dicho acerca de la descomposición de fuentes de dependencias de datos con un ejemplo sencillo. Sea la secuencia de código presentada en el ejemplo de la Fig. 2. Se propone descomponer las dependencias de datos en aquellas debidas a operandos explícitos y aquellas debidas a operandos implícitos (los que vienen impuestos por el propio código de operación). Según esto tendremos:

$$D_{op\ expl.} \quad \text{OR} \quad D_{op\ impl.} = D$$

$$\begin{pmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 1&1&0&0&0&0 \\ 0&1&1&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix} \text{OR} \begin{pmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \end{pmatrix} = \begin{pmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 1&1&0&0&0&0 \\ 1&1&1&0&0&0 \\ 0&0&1&1&0&0 \\ 0&0&0&0&1&0 \end{pmatrix}$$

Fig. 10. Composición de fuentes de dependencias.

Que genera, como se puede ver, la matriz de dependencias del ejemplo de la Fig. 2. Se puede observar como la composición de ambas fuentes de dependencias de datos ha provocado una mayor secuencialidad en el grafo resultante que la que produce cada fuente de dependencias por separado.

Desafortunadamente, la matriz de dependencias de datos resultante o final no admite el proceso inverso o descomposición, que nos permitiría realizar un análisis más metódico y preciso del impacto de cada fuente de dependencias de datos, ya que el solapamiento de dependencias no permite identificar, conociendo la matriz final, su procedencia.

a. Algunas composiciones posibles.

La construcción de los grafos de dependencias de datos admite, como hemos visto previamente, la composición de diferentes categorías. Una primera aproximación sería la que diferencia entre dependencias verdaderas, dependencias de salida y antidependencias.

$$D = D_{verdaderas} \text{ OR } D_{salida} \text{ OR } D_{antidependencias} \tag{36}$$

Esto nos permite distinguir el tipo de dependencia de datos y predecir si tiene relevancia computacional (dependencias verdaderas) o sólo es debida a la reutilización del almacenamiento físico (dependencias de salida y antidependencias).

No obstante, es posible un análisis más fino diferenciando las posibles fuentes de dependencias de datos. En este caso tendremos:

$$D = D_{s1} \text{ OR } D_{s2} \text{ OR } D_{s3} \text{ OR } \cdots \tag{37}$$

Estas fuentes diversas de dependencias de datos pueden ser:

- número limitado de registros *software* de propósito general
- reutilización de registros
- uso de operandos implícitos
- escrituras de códigos de condición
- procesamiento de direcciones de memoria
- tráfico con la pila, etc.

Todas ellas pueden ser consideradas, además, en las tres variantes presentadas en primer lugar: verdaderas, antidependencias y de salida.

b. Longitud del camino crítico de la composición.

Todas las propiedades y tratamientos propuestos para la matriz D se pueden realizar para cada una de las matrices que representan diferentes fuentes de dependencias de datos (D_{sn}).

Es interesante considerar qué sucede con la longitud del camino crítico de la matriz resultante en función de sus componentes. El grafo resultante no tiene una longitud de camino crítico necesariamente igual a la suma de los caminos de sus componentes ya que las cadenas de dependencias de datos se pueden solapar. No es posible, por tanto, determinar de manera directa la longitud de la composición en función de las componentes. Es necesario conformarse con acotar los resultados.

En cuanto al límite superior, sabemos que la longitud final está acotada por el número de instrucciones presentes en el grafo. También podemos afirmar que la longitud del camino crítico de la composición nunca será mayor que la suma de los caminos de las componentes asumiendo el caso más

desfavorable en el que todas ellas contribuyeran a crear cadenas de dependencias de datos. Por tanto, el límite superior vendrá dado por el mínimo de ambos:

$$\min\left\{\sum_i \mathrm{L}(D_{si}), n\right\} \geq \mathrm{L}(D) \tag{38}$$

La ecuación (38) utiliza la métrica basada en pasos de computación.

En cuanto al límite inferior nunca será menor que el correspondiente a la componente del mayor camino crítico, asumiendo que todos los demás se solapan con él, es decir, que no se provocan dependencias cruzadas entre las componentes. Formalmente:

$$\mathrm{L}(D) \geq \max_i\left\{\mathrm{L}(D_{si})\right\} \tag{39}$$

En definitiva, la longitud del camino crítico de la matriz D compuesta L(D) es un número acotado en el rango:

$$\min\left\{\sum_i \mathrm{L}(D_{si}), n\right\} \geq \mathrm{L}(D) \geq \max_i\left\{\mathrm{L}(D_{si})\right\} \tag{40}$$

Es decir, L(D) siempre será igual o mayor que el mayor camino de dependencias de entre las componentes y siempre será menor o igual a la suma de las longitudes con el límite n del número de instrucciones presentes en la secuencia de código.

c. Ejemplo ilustrativo

Se propone un ejemplo basado en una secuencia de código x86. En la Sección 1.c se enumeran algunas de las características de la arquitectura de este repertorio de instrucciones en relación a su comportamiento en ejecución superescalar. Las instrucciones del ensamblador x86 utilizan operandos explícitos –escritos por el programador– e implícitos –asociados necesariamente al código de operación–. Esta característica, junto a otras tales como el uso dedicado de determinados registros, el peculiar modo de acceder a posiciones de memoria o el uso del registro de estado hace muy interesante el análisis de las diferentes fuentes de dependencias de datos [32].

En la Tabla 1 proponemos una secuencia de código que bien puede representar un bloque básico típico. Se ha utilizado el subconjunto de 16 bits por simplicidad. Los operandos utilizados por cada operación se clasifican en dos grandes conjuntos: operandos leídos y operandos escritos. Dentro de cada uno de ellos se agrupan por su funcionalidad: datos ubicados en registros o en memoria, registros utilizados en el cálculo de direcciones efectivas de memoria, registros involucrados en accesos a la pila y registro de estado. A su vez se han separado los operandos explícitos (incluidos en el formato de la instrucción) de aquellos implícitos (asociados necesariamente al código de operación). Cada una de estas categorías representa una posible fuente de dependencias de datos cuyo impacto podemos estudiar separadamente.

Los accesos a los operandos se clasifican de acuerdo a las categorías explicadas cuando se está seguro de su pertenencia. En algunos casos no resulta fácil conocer la funcionalidad real que tendrá un acceso a un operando. Es por ello que la escritura explícita en operandos relacionados con la aritmética de direcciones o con la pila aparecen vacías en la tabla.

Respecto a las ubicaciones de memoria se asume la situación más pesimista, es decir, se considera la memoria como un recurso único a la hora de generar dependencias de datos: los accesos no se diferencian por su puntero. No así la pila ya que son accesos ordenados por el puntero de pila.

A partir de la Tabla 1 se construyen las matrices de dependencias de datos D para cada una de las fuentes seleccionadas y para los tres tipos de dependencias de datos: dependencias verdaderas, antidependencias y dependencias de salida. La Fig. 11 muestra todos los resultados.

Tabla 1. Secuencia de código y los operandos utilizados en cada operación.

secuencia de código		operandos leídos								operandos escritos							
		explícitos				implícitos				explícitos				implícitos			
		reg	dir	pila	estado	reg	dir	pila	estado	reg	dir	pila	estado	reg	dir	pila	estado
0:	MOV DX, 6B42	-	-	-	-	-	-	-	-	DX	-	-	-	-	-	-	-
1:	MOV CS:[BX], DX	DX	CS, BX	-	-	-	-	-	-	MEM	-	-	-	-	-	-	-
2:	SUB BX, AX	AX, BX	-	-	-	-	-	-	-	BX	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
3:	MOV AH, 30	-	-	-	-	-	-	-	-	AH	-	-	-	-	-	-	-
4:	INT 21	-	-	-	-	AX, CS, IP	-	SS, SP	Flags	-	-	-	-	AX, BX, CX, CS, IP	-	SP	IF, TF
5:	CLI	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	IF
6:	MOV BP, [BX][SI]	MEM	BX, SI	-	-	-	DS	-	-	BP	-	-	-	-	-	-	-
7:	MOV DS, DX	DX	-	-	-	-	-	-	-	DS	-	-	-	-	-	-	-
8:	OR SS:[DI], AX	AX, MEM	SS, DI	-	-	-	-	-	-	MEM	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
9:	CWD	-	-	-	-	AX	-	-	-	-	-	-	-	AX, DX	-	-	-
10:	XOR CX, BX	BX, CX	-	-	-	-	-	-	-	CX	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
11:	DEC DI	DI	-	-	-	-	-	-	-	DI	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
12:	INC SI	SI	-	-	-	-	-	-	-	SI	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
13:	MOV BL, ES:[SI]	MEM	ES, SI	-	-	-	-	-	-	BL	-	-	-	-	-	-	-
14:	TEST [BX][DI], AL	AL, MEM	BX, DI	-	-	-	DS	-	-	MEM	-	-	-	-	-	-	OF, SF, ZF, AF, PF, CF
15:	JNE/JNZ IP+F7	-	-	-	ZF	-	-	-	-	-	-	-	-	IP	-	-	-

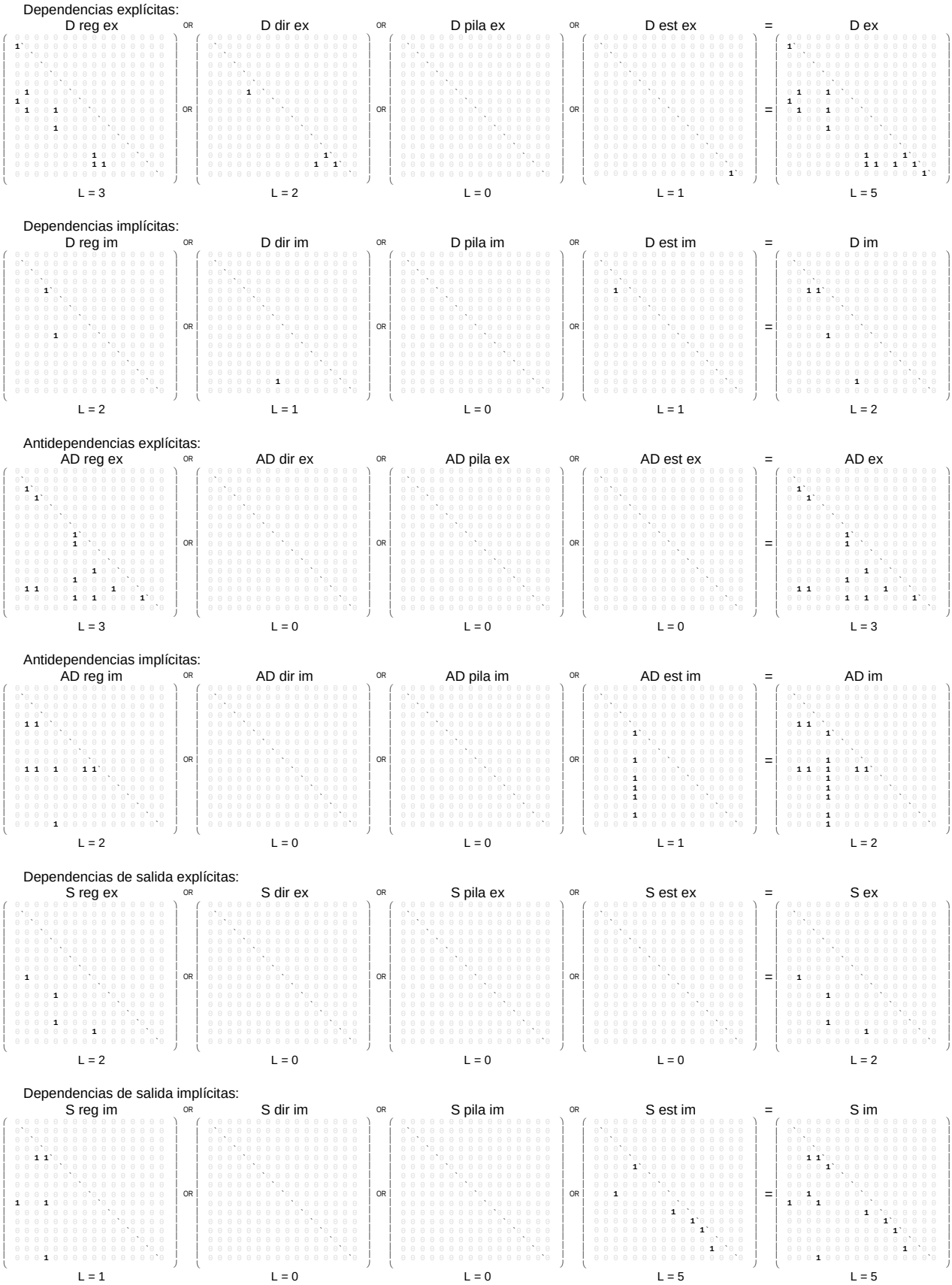


Fig. 11. Composición de diferentes fuentes de dependencias de datos para la secuencia de código ejemplo.

Se han combinado las dependencias por grupos componiendo las 4 fuentes básicas para cada tipo de dependencia y separadamente en operandos explícitos e implícitos. Para cada matriz, ya sea de una clase de dependencias o una compuesta, se ha calculado la longitud del camino crítico L. Se comprueba como se ajustan las longitudes de las matrices compuestas a las acotaciones mostradas en la ecuación (40).

Se ha evaluado también la composición secuencial de cada fuente de dependencias y cada tipo básico dando lugar a la gráfica de la Fig. 12. El orden de la composición no altera el resultado final pero la secuencia utilizada no es en absoluto arbitraria. Comenzamos por las dependencias verdaderas, que son las que no tienen solución hardware, continuando por las antidependencias y las dependencias de salida. Dentro de cada tipo básico se comienza por las derivadas de los operandos explícitos y luego los implícitos. Finalmente, las funciones de los operandos también tienen un orden: comenzamos por los accesos a operandos con un mayor significado computacional ya que se refieren a la manipulación directa de los datos, seguimos por la aritmética de direcciones que resulta indispensable para acceder a variables pero que no está directamente involucrada en el cálculo computacional principal, seguidamente se evalúa el impacto de los operandos relacionados con el acceso a la pila y terminamos con los accesos al registro de estado cuya única función es salvar los códigos de condición utilizados en saltos.

La secuencia de la composición ilustra cómo, a las dependencias generadas por el procesamiento con verdadero significado computacional, se le van sumando sobreordenaciones por dependencias de datos generadas por una carga computacional colateral (aritmética de direcciones de memoria, accesos a la pila y salvaguarda del estado).

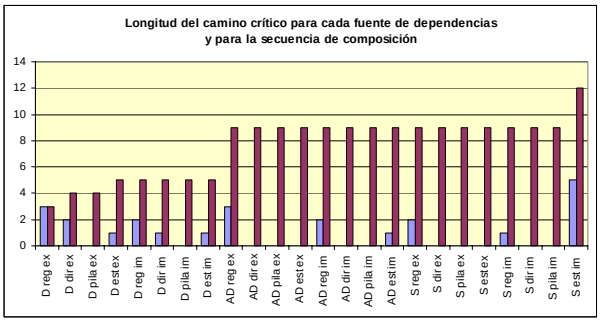


Fig. 12. Longitud del camino crítico para cada fuente de dependencias de datos y la composición de cada una de ellas.

Del ejemplo se pueden sacar algunas consecuencias inmediatas que en trabajos posteriores habrá que comprobar si pueden generalizarse.

Comenzamos por las dependencias verdaderas. A las dependencias entre registros de datos explícitos se le suman las dependencias generadas por la aritmética de direcciones. En consecuencia, es está una causa de degradación del paralelismo a nivel de instrucción. La siguiente pérdida de paralelismo viene de la mano de los códigos de condición. Las dependencias verdaderas a través de operandos implícitos ya no afectan más, en este caso, a la longitud del camino crítico. Encontramos, por tanto, dos fuentes con impacto relevante en la pérdida de paralelismo:

- la aritmética de direcciones; y
- los códigos de condición.

Las antidependencias entre registros explícitos proporcionan una nueva degradación en el paralelismo que, en nuestro ejemplo, ya no se ve alterada hasta la última contribución debida a las dependencias de salida en implícitos, concretamente la debida a los códigos de condición. Es decir, entre las dependencias de datos debidas a las limitaciones de recursos físicos las más relevantes parecen ser:

- los registros de uso explícito en antidependencias; y
- los códigos de condición en dependencias de salida.

4. Conclusiones y trabajo futuro.

Se ha propuesto un modelo de análisis aplicable al proceso de computación en la capa del lenguaje máquina que permite evaluar cuantitativamente el impacto sobre la ejecución superescalar tanto de la arquitectura del repertorio de instrucciones como del propio procedimiento de compilación.

Se han identificado las propiedades topológicas y restricciones que ha de cumplir la matriz *D* en el ámbito ILP y se ha establecido la manera de cuantificar el grado de ordenación del código, la reutilización de los datos y su tiempo de vida a partir de *D*. También se ha definido una métrica para evaluar el grado de paralelismo disponible.

Se muestra cómo las diferentes fuentes de dependencias de datos se pueden componer permitiendo un análisis preciso del impacto de cada una sobre el grado de paralelismo final.

El modelo analítico propuesto ha sido aplicado a la evaluación de algunos aspectos de la arquitectura del repertorio x86 y se ha obtenido información valiosa [32].

En posteriores trabajos se estudiará la contribución de cada una de las fuentes de dependencias analizando su comportamiento sobre un conjunto mayor de programas. También parece posible extender el uso de los grafos para modelar las limitaciones de la capa física y los procesos de asignación-planificación.

A modo de resumen, enumeramos a continuación las aportaciones que se han realizado a lo largo de la nota técnica:

- a partir de las definiciones tradicionales de la teoría de grafos y apoyados en el concepto novedoso de la valencia reducida introducimos la matriz de dependencias de datos *D*
- se han identificado una serie de propiedades topológicas y restricciones que ha de cumplir la matriz de dependencias de datos *D* en el ámbito del procesamiento paralelo de instrucciones
- se ha determinado la relación entre la matriz *D* y la matriz de adyacencia *A* tradicionalmente usada en teoría de grafos

- se ha identificado la trasposición de la matriz *D* como la manera de relacionar las dos orientaciones del grafo
- se ha demostrado la relación entre la matriz *D* y la matriz de valencia reducida para una orientación dada
- se ha introducido el concepto de acoplamiento del código como método de cuantificación del grado de ordenación del código
- se ha establecido la manera de cuantificar el grado de reutilización de datos
- se ha identificado la relación entre la matriz *D* y los caminos de dependencias de datos de longitud mayor que 1
- se ha identificado la relación entre la longitud del camino crítico y las potencias de la matriz *D*
- se han definido variables que relacionan la longitud del camino crítico y el grado de paralelismo disponible en el código
- se ha propuesto un método algorítmico para calcular la longitud del camino crítico
- las diferentes fuentes de dependencias de datos se pueden componer y se han propuesto algunas fuentes de dependencias
- se ha realizado una acotación de la longitud del camino crítico de la composición
- se han sugerido algunas posibles líneas de estudio a partir de un ejemplo ilustrativo

5. Referencias.

[1] T. L. Adams and R. E. Zimmerman, "An analysis of 8086 instruction set usage in MS DOS programs," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989, pp. 152 - 160.

[2] A. V. Aho, J. E. Hopcroft, J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Co., 1983.

[3] A. V. Aho and J. Ullman. *Foundations of Computer Science*. Computer Science Press, 1992.

[4] T. M. Austin and G. S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," in *Proceedings of the 19th International Symposium on Computer Architecture*, 1992, pp. 342-351.

[5] D. Bhandarkar and J. Ding, "Performance characterization of the Pentium Pro processor," in *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, 1997, pp. 288 –297.

[6] N. L. Biggs, *Algebraic Graph Theory* (2nd. edn.), ISBN: 0-521-45897-8, Cambridge University Press, 1993.

[7] M. Butler, Tse-Yu Yeh, Y. Patt, M. Alsup, H. Scales and M.Shebanow. "Single Instruction Stream is Greater than Two," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 163-174.

[8] T. H. Cormen, C. E. Leiserson and R. L. Rivert. *Introduction to Algorithms*. Mit Press, McGraw Hill, 1996.

[9] R. David, "Modular design of asynchronous circuits defined by graphs," *IEEE Transactions on Computers*, vol. C-26, 8, pages 727-737, August 1977.

[10] A. L. Davis and R. M. Keller, "Data flow program graphs," *IEEE Computer*, vol. 15, 2, February, 1982.

[11] J. B. Dennis, "Concurrency in software systems," in *Advanced Course in Software Engineering*, Springer-Verlag, pages 111-127, 1973.

[12] S. McFarling, "Combining Branch Predictors", *W.R.L. Technical Note TN-36*. Digital Equipment Corporation. Palo Alto, CA. June 1993.
Available at: http://www.research.compaq.com/wrl/techreports/index.html.

[13] K. I. Farkas, N. P. Jouppi and P. Chow. "Register File Design Considerations in Dynamically Scheduled Processors," in *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA)*, 1996, pp. 40–51.

[14] D. G. Feitelson. "Metric and Workload Effects on Computer Systems Evaluation," *IEEE Computer*, vol. 36, 9, September, 2003.

[15] J. González and A. González. "Identifying Contributing Factors to ILP," in *Proceedings of the 22nd Euromicro Conference (Euromicro'96)*, 1996, pp. 45-50. Short Contribution.

[16] C. D. Godsil and G. F. Royle, *Algebraic Graph Theory*, ISBN: 0-387-95220-9, Springer-Verlag, 2001.

[17] I. J. Huang and T. C. Peng, "Analysis of x86 Instruction Set Usage for DOS/Windows Applications and Its Implication on Superscalar Design," *IEICE Transactions on Information and Systems*, Vol.E85-D, No. 6, pp. 929-939, June 2002. (SCI).

[18] I. J. Huang and P. H. Xie, "Application of Instruction Analysis/Scheduling Techniques to Resource Allocation of Superscalar Processors," *IEEE Transactions on VLSI Systems*, vol. 10, no. 1, pp. 44-54, February 2002.

[19] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272-282, April 1989.

[20] L. Joyanes and I. Zahonero. *Estructura de datos. Algoritmos, abstracción y objetos*. Mc Graw Hill, 1998.

[21] M. Kumar, "Measuring parallelism in computation intensive scientific/engineering applications," *IEEE Transactions on Computers*, 37(9), pp. 1088-1098, 1988.

[22] M. Lam and R. Wilson. "Limits of Control Flow on Parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 46-56.

[23] M. H. Lipasti and J. P. Shen. "Exceeding the Dataflow Limit Via Value Prediction," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226-237, 1996.

[24] T. Monreal, V. Viñals, A. González and M. Valero. "Hardware Schemes for Early Register Release," in *Proceedings of the International Conference on Parallel Processing (ICPP'02)*, 2002, pp. 5-13.

[25] O. Mutlu, J. Stark, Ch. Wilkerson and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, 2003, pp. 129–140.

[26] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr and A. Hoffmann. "A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation," in *Proceedings of the 39th Design Automation Conference (DAC 2002)*, pp. 22-27, 2002.

[27] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, 29(12), pages 1184-1201, December 1986.

[28] C. A. Petri, "Comunication with automata," *Supplement 1 to Technical Report RADC-TR-65-377*, vol. 1, 1966.

[29] M. A. Postiff, D. A. Greene, G. S. Tyson and T. N. Mudge, "The Limits of Instruction Level Parallelism in SPEC95 Applications," in *Proceedings of the 3rd Workshop on Interaction Between Compilers and Computer Architecture*, 1998.

[30] A. Ramirez, J. L. Larriba-Pey and M. Valero, "The effect of code reordering on branch prediction," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 189-198, October 2000.

[31] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. C-21, pages 1405-1411, December 1972.

[32] R. Rico, J. I. Pérez, J. A. Frutos. "The impact of x86 instruction set architecture on superscalar processing," *Journal of Systems Architecture*, vol. 51-1, pages 63-77, January 2005.

[33] M. Silva. *Las redes de Petri: en la automática y la informática*. Editorial AC, 1985.

[34] P. Simonen, I. Saastamoinen, M. Kuulusa and J. Nurmi. "Advanced Instruction Set Architectures for Reducing Program Memory Usage in a DSP Processor," in *Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA '02)*, pp. 477-479, 2002.

[35] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Hill and V. S. Pai. "Challenges in Computer Architecture Evaluation," *IEEE Computer*, vol. 36, 8, August, 2003.

[36] M. Smith, M. Johnson and M. Horowitz. "Limits on Multiple Instruction Issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* 1989, pp. 272-282.

[37] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," in *Proceedings of the IEEE*, 83(12), pp. 1609-1624, December, 1995.

[38] J. Stark, M. D. Brown and Y. N. Patt. "On Pipelining Dynamic Instruction Scheduling Logic," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture,* 2000, pp. 57-66.

[39] D. Stefanovic and M. Martonosi, "Limits and Graph Structure of Available Instruction-Level Parallelism," in *Proceedings of the European Conference on Parallel Computing (Euro-Par 2000)*, 2000.

[40] K. B. Theobald, G. R. Gao and L. J. Hendren, "On the Limits of Program Parallelism and its Smoothability," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 10-19, 1992.

[41] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 392-403.

[42] D. W. Wall, "Limits of instruction-level parallelism," *W.R.L. Technical Note TN-15*. Digital Equipment Corporation. Palo Alto, CA. December 1990.
Available at: http://www.research.compaq.com/wrl/techreports/index.html.

[43] D. W. Wall, "Limits of instruction-level paralelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176-188, April 1991.
Also as:
W.R.L. Research Report 93/06. Digital Equipment Corporation. Palo Alto, CA. 1993. Available at: http://www.research.compaq.com/wrl/techreports/index.html.

[44] K. Wang and M. Franklin. "Highly accurate data value prediction using hybrid predictors," in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 281-290, 1997.

[45] M. Wolfe. *High Performance Compiler for Parallel Computing*. Addison-Wesley, CA, 1996.

[46] H. Zima and B. Chapman. "Supercompilers for Parallel and Vector Supercomputers," *ACM Press Frontiers Series*, 1990.