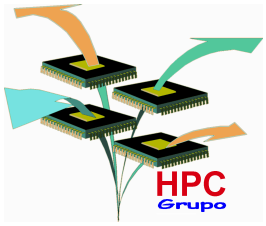


Estudio detallado del impacto de los códigos de condición del repertorio x86 sobre la ejecución superescalar

Informe técnico TR-UAH-AUT-GAP-2006-23-es

Virginia Escuder, Raúl Durán, Rafael Rico

Departamento de Automática, Universidad de Alcalá, España

Marzo 2006

English version:

In-depth analysis of x86 instruction set condition codes influence on superscalar execution

Technical Report TR-UAH-AUT-GAP-2006-23-en

Virginia Escuder, Raúl Durán, Rafael Rico

Department of Computer Engineering, Universidad de Alcalá, Spain

Resumen:

El diseño del repertorio de instrucciones representa un aspecto crucial en la arquitectura de computadores. Los requisitos que deben cumplir han ido evolucionando a lo largo de la historia. En el ámbito del procesamiento superescalar, el más extendido hoy en día, un aspecto decisivo es evitar el acoplamiento del código por dependencias de datos. Sin embargo, los repertorios de instrucciones pueden incorporar características que repercuten negativamente sobre la disponibilidad de paralelismo y que es conveniente analizar.

La arquitectura del omnipresente repertorio x86 reúne características negativas para el procesamiento superescalar que pueden incidir en el rendimiento final: uso dedicado de registros, uso de operandos implícitos, complicados mecanismos de cómputo de direcciones, uso de códigos de condición, etc. Es, por tanto, un candidato idóneo para ser sometido a evaluación. Concretamente, vamos a estudiar el impacto del uso de códigos de condición.

En este trabajo se proponen dos métodos para abordar el problema. Por una parte, un estudio desde un punto de vista estadístico, a partir de las distribuciones de uso tanto de operaciones como de operandos. Por otra parte, se va a realizar un análisis matemático, basado en la teoría de grafos, cuantificando la contribución al acoplamiento de los códigos de condición según los diferentes tipos de dependencias de datos.

Finalmente, se hará una valoración de cómo influye el uso de códigos de condición a nivel de microoperaciones y se propondrán algunas posibles soluciones para mejorar el rendimiento.

Palabras clave: Arquitectura del repertorio de instrucciones, paralelismo a nivel de instrucción, trazado de instrucciones, uso de operaciones, uso de operandos, teoría de grafos.

1. Introducción

El diseño del repertorio de instrucciones ha sido un tema fundamental desde los primeros tiempos de la computación. En el popular texto de Hennessy y Patterson podemos encontrar un resumen histórico, más exhaustivo y referenciado del que aquí podemos hacer, que da cuenta de este hecho [12]. En los últimos tiempos han aparecido desarrollos comerciales como EPIC [21] o proyectos experimentales como el "*EDGE Instrution Set*" [7] que manifiestan el interés actual por la cuestión.

Los criterios de diseño han evolucionado a lo largo del tiempo: dar soporte a la protección de memoria, simplificar la compilación de lenguajes de alto nivel, minimizar el espacio de representación, simplificar la arquitectura, implantar la ejecución condicionada, etc. Se han realizado estudios teóricos en los que se proponen otras muchas especificaciones [6, 16].

Sin embargo, la evaluación de la arquitectura de los repertorios de instrucciones no ha sido el propósito de un volumen de trabajos acorde con la importancia del tema. En nuestra opinión, resulta paradigmático el artículo de Lunde "*Empirical Evaluation of Some Features of Instruction Set Processor Architectures*" [15] ya que constituye una valoración del impacto de una característica de la arquitectura del repertorio de instrucciones (número de registros) sobre el rendimiento en tiempo de ejecución. En este sentido, aunque el trabajo es antiguo, nos planteamos rescatar su filosofía proponiendo el análisis de las arquitecturas de los repertorios de instrucciones puesto que consideramos que su influencia en el rendimiento final es decisiva.

El estudio en profundidad de la arquitectura de los repertorios de instrucciones es un enfoque que ha caído en desuso por un doble motivo: por un lado, la presunción de que obtenemos una solución más completa a los desafíos computacionales analizando de manera conjunta el repertorio y el *hardware* que lo interpreta y, por otro, el uso y abuso de los simuladores como método más extendido de evaluación del rendimiento cuya propia esencia no permite diferenciar el impacto del repertorio del impacto de los recursos físicos [22].

En este trabajo reivindicamos la necesidad de someter a estudio las arquitecturas de los repertorios de instrucciones. El estudio teórico de la arquitectura de los repertorios de instrucciones, independientemente de las otras capas del proceso de computación, puede aportar luz al diseño de la capa física y al diseño de compiladores además de caracterizar convenientemente el propio repertorio.

Hoy en día, uno de los objetivos más interesantes es el desacoplo del código, evitando dependencias de datos, con el fin de conseguir un procesamiento superescalar lo más concurrente posible. Debemos abordar el problema del rendimiento superescalar en toda su extensión: desde su origen en el algoritmo, pasando por el lenguaje de alto nivel que lo implementa y el proceso de compilación y sin olvidar la arquitectura del repertorio de instrucciones; evitando quedarse en el órgano gestor (*hardware*). En este sentido, el enfoque debe desligarse de la capa física: se impone prestar una mayor atención a la capa del lenguaje máquina en sí misma. Concretamente, el código puede llegar sobreordenado al momento de la ejecución por culpa de ciertas características de la arquitectura del repertorio de instrucciones que o bien no tienen solución en la capa física o bien hacen aumentar su complejidad y consumo de potencia. Los repertorios de instrucciones incorporan características que repercuten negativamente sobre la disponibilidad de paralelismo: uso dedicado de registros, uso de operandos implícitos, complicados mecanismos de cómputo de direcciones, uso de códigos de condición, etc.

Es necesario, por tanto, analizar las arquitecturas de los repertorios de instrucciones atendiendo a las características esenciales que deben cumplir para una óptima ejecución superescalar, es decir, cómo y por qué se acoplan las instrucciones a través de las dependencias de datos. Sin embargo, la gran mayoría de los trabajos que evalúan repertorios se limitan a estudiar la distribución de uso de instrucciones: por ejemplo, [8] sobre *VAX* o [1] sobre x86. Muy pocos trabajos evalúan la distribución de uso de los datos [13].

2. El repertorio x86

Los criterios de diseño del repertorio de instrucciones x86 son básicamente dos: reducir la distancia semántica entre los lenguajes de alto nivel y el procesador y minimizar el espacio de representación de la imagen binaria de los ejecutables. Aunque estos criterios están obsoletos, el repertorio así diseñado se sigue utilizando en aras de la compatibilidad binaria. Sin embargo, en el ámbito del procesamiento superescalar su comportamiento es bastante deficiente. El uso dedicado de registros, el uso de operandos implícitos, el cómputo de direcciones utilizando hasta 3 registros, la evaluación de saltos basada en códigos de condición, etc. son características que ocultan las posibles oportunidades de concurrencia que subyacen en la tarea computacional. A partir de la distribución de uso de los datos en programas compilados para x86, se han identificado estas fuentes de potenciales acoplamientos del código [20].

El efecto de estas características indeseables se manifiesta en una sobreordenación del código de la imagen binaria. Es, por tanto, una penalización impuesta en la capa del código máquina antes de alcanzar el tiempo de ejecución (capa física). En consecuencia, las instrucciones aparecen más acopladas en la capa física de lo que cabría esperar observando el código fuente de alto nivel (capa de programa).

Por otra parte, los trabajos acerca del rendimiento obtenido en procesadores x86 y no-x86, para diferentes propuestas arquitectónicas, manifiestan claramente una enorme diferencia. Los trabajos

aplicados a procesadores no-x86 dan un IPC medio en el rango 2,5 – 15 con picos alrededor de 30 [24, 25, 26] mientras que los estudios dedicados a procesadores x86 (mucho más escasos) dan un IPC en el rango 0,5 – 3,5 en las situaciones más favorables [17, 23]. Todo esto nos lleva a pensar que la propia arquitectura del repertorio x86 supone un factor limitante para la disponibilidad de paralelismo de grano fino.

Debido a las limitaciones que exhibe en el procesamiento superescalar y a la gran extensión de uso de que disfruta, la arquitectura del repertorio x86 es un buen candidato para ser analizado. Entre los aspectos más interesantes se encuentra el estudio del impacto sobre la disponibilidad de paralelismo a nivel de instrucción del uso de los códigos de condición y de los diferentes mecanismos de cómputo de direcciones efectivas de memoria.

Recientemente, se ha establecido en cerca del 13% el promedio de mejora en el grado de paralelismo conseguido al evitar la contribución al acoplamiento de los códigos de condición sobre un banco de pruebas [20]. Este dato es una primera aproximación al caso ya que determina un límite superior ideal en la aceleración, sólo alcanzable si no utilizamos este tipo de datos. No obstante, se hace necesario realizar un estudio más preciso que indique la funcionalidad asignada en la práctica a los códigos de condición, en qué tipo de dependencias de datos se concreta el acoplo de las instrucciones a través de los mismos, qué características de los programas potencian su impacto, cuál es la relevancia computacional real del acoplo y, a partir de todo esto, qué soluciones podemos ofrecer para mejorar el rendimiento superescalar.

3. Los códigos de condición en las arquitecturas de los repertorios de instrucciones

Los códigos de condición son una de las posibles soluciones a la implementación de las bifurcaciones condicionales. Así, la evaluación de un estado se lleva a cabo comprobando el valor de uno o varios de los bits de condición. Desde el punto de vista práctico, los bits que describen las diferentes condiciones se agrupan en un registro de estado. Normalmente, como consecuencia de la ejecución de operaciones de proceso, se actualiza el estado escribiendo una serie de bits que describen el resultado: valor nulo, signo, paridad, acarreo, desbordamiento, etc. De ahí la denominación de arquitecturas basadas en registro de estado.

Desde el punto de vista estrictamente de la ejecución superescalar, los códigos de condición incrementan el orden de las instrucciones ya que pasan información de una instrucción, típicamente de proceso, a un salto condicional. El repertorio x86 utiliza este mecanismo.

En el plano teórico podemos encontrar otras dos maneras de organizar los saltos condicionales: la evaluación de un registro en la propia instrucción de salto con respecto a algún criterio y la comparación y bifurcación en una única instrucción. La primera alternativa es simple y óptima desde el punto de vista superescalar mientras que la segunda resulta de la fusión de dos instrucciones en una, complicando el diseño del cauce. Comercialmente se ajustan al la primera *Alpha* y *MIPS* mientras que encontramos la segunda en *PA-RISC* y *VAX*.

Todos estos mecanismos, sus ventajas y sus inconvenientes son bien conocidos y han sido expuestos con claridad en el texto de Hennessy y Patterson [12].

En cuanto a los códigos de condición, se puede analizar su impacto en profundidad observando el comportamiento del bloque básico en lo que a la disponibilidad de paralelismo se refiere. Veamos un ejemplo: en la Fig. 1 se presenta un bloque básico prototipo y el grafo de dependencias de datos correspondiente a la contribución de los códigos de condición. Observamos como la escritura de estado en la instrucción 2 genera una dependencia verdadera (lectura después de escritura) con la instrucción 3. Dicha dependencia tiene significado computacional e impone una cierta serialización del programa ya que hemos de emplear dos pasos de computación para procesar el bloque básico. El mecanismo en sí mismo limita la disponibilidad de paralelismo.

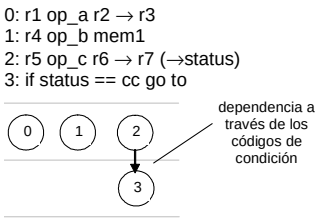


Fig. 1. Impacto de los códigos de condición sobre el paralelismo en un bloque básico ejemplo.

Ahora bien, veamos otro tipo de bloque básico igualmente posible. El caso presentado en la Fig. 2 es similar al anterior pero en éste el bloque básico cuenta con 2 instrucciones de proceso y ambas escriben el estado. El grafo de dependencias de datos generado por la contribución de los códigos de condición manifiesta una menor disponibilidad de paralelismo y debe ser procesado en 3 pasos de computación. Sin embargo, resulta muy interesante comprobar como la nueva dependencia de datos es de salida (escritura después de escritura). Este tipo de dependencia es debida a la limitación de recursos físicos o, lo que es igual, al uso de una misma ubicación física para almacenar información. La típica solución, implementada en *hardware*, que deshace la dependencia consiste en cambiar la ubicación en la que se escribe (técnica de renombramiento). En cualquier caso, hay que destacar que la dependencia desactivada o no carece de cualquier significado computacional, es decir, no supone ningún procesamiento requerido por la tarea que se está llevando a cabo. Es, simplemente, una consecuencia directa de la arquitectura del repertorio de instrucciones.

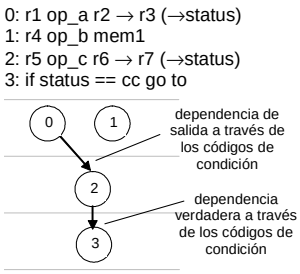


Fig. 2. Impacto de los códigos de condición sobre el paralelismo en un bloque básico ejemplo con 2 escrituras de estado.

El conocimiento de este efecto negativo sobre la ejecución superescalar ha sido tenido en cuenta en el diseño del repertorio de instrucciones del procesador *PowerPC* que también pertenece, como el x86, a las arquitecturas basadas en registro de estado. Así, el formato del *PowerPC* incluye un bit que determina si se almacena o no el estado resultante de la ejecución de una operación de proceso. El compilador decide si es necesario o no crear la dependencia de datos. Este mecanismo incorporado en el *PowerPC* asegura que solamente se producen acoplamientos a través de los códigos de condición cuando tienen significado computacional. La Fig. 3 ilustra lo descrito a partir del bloque básico ejemplo de la Fig. 2. Ahora la secuencia de código se puede procesar en menos pasos de computación tal y como muestra su grafo de dependencias de datos.

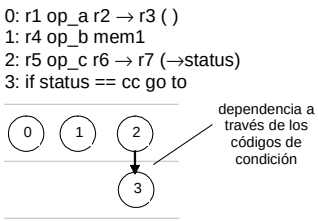


Fig. 3. Impacto de los códigos de condición sobre el paralelismo en un bloque básico ejemplo con 2 operaciones de proceso pero un única escritura de estado.

La arquitectura del repertorio de instrucciones x86 no ha resuelto este problema debido a la necesidad de mantener la compatibilidad binaria. Esta característica genera dependencias de datos sin significado computacional que tienen un efecto negativo sobre la ejecución superescalar, complican el proceso de compilación y demandan una solución en tiempo de ejecución que implica un coste adicional en términos de potencia, área y diseño.

4. Los códigos de condición en la arquitectura del repertorio x86

Desde el punto de vista hardware, los códigos de condición se agrupan en el registro de estado. Este registro reúne bajo una única denominación toda una serie de bits con significado propio. A su vez, estos bits se pueden clasificar en dos grupos:

- banderas de control; y
- banderas de estado.

Las banderas de control se utilizan para determinar ciertos aspectos de la funcionalidad del procesador: gestión de interrupciones, autoincremento o autodecremento de posiciones de memoria en determinadas operaciones, modo paso a paso, etc. Las banderas de estado dan cuenta de cómo es el resultado de una operación de proceso y son las utilizadas normalmente para la toma de decisiones en los saltos condicionales. Así, el registro de estado mantiene la dualidad de ser considerado como una única ubicación de datos pero, a la vez, guarda un significado independiente para cada bit que se analiza o evalúa independientemente.

Las banderas de estado contienen los códigos de condición utilizados para pasar información a los saltos condicionales tal y como corresponde a las arquitecturas basadas en registro de estado. Sin embargo, en el caso del repertorio x86, los códigos de condición son usados también como operandos de entrada en algunas operaciones de proceso, como por ejemplo, rotaciones a través del bit indicador de acarreo, operaciones de ajuste BCD u operaciones extendidas a tamaños superiores a la palabra.

Desde el punto de vista del rendimiento del procesamiento superescalar, la utilización de los códigos de condición como dato de entrada a una operación, tiene una importancia potencialmente relevante. Cuando pasamos información de una instrucción a otra a través de un código de condición, las dependencias de datos que se generan son verdaderas ya que la lectura del código de condición se asocia necesariamente con una escritura previa (lectura después de escritura). Dicha dependencia de datos verdadera serializa la ejecución de instrucciones ya que establece una cadena de operaciones dependientes. Se hace, por tanto, necesario identificar estos casos y analizar su impacto en el acoplamiento final del código.

Puesto que en la arquitectura del repertorio de instrucciones x86 encontramos diversas clases de instrucciones que acceden a los códigos de condición en lectura y escritura, trataremos de clasificar las instrucciones que acceden al registro de estado en diversos grupos. Nos concentraremos en las banderas de estado (O: desbordamiento, S: signo, Z: cero, A: acarreo auxiliar, P: paridad, C: acarreo) que son las que recogen información con significado computacional obviando las de control.

Grupo I:

En primer lugar, podemos hablar de las operaciones de movimiento del registro de estado. La Tabla 1 muestra estas instrucciones que se encargan de hacer transferencias entre el registro de estado y el acumulador o la pila. Se ha incluido la llamada y el retorno de interrupciones *software* ya que al salvar el contexto transfieren con la pila el registro de estado.

Grupo I												
	lectura						escritura					
Transferencias	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
LAHF		X	X	X	X	X						
POPF							X	X	X	X	X	X
PUSHF	X	X	X	X	X	X						
SAHF								X	X	X	X	X
INT	X	X	X	X	X	X						
IRET							X	X	X	X	X	X

Tabla 1. Instrucciones de transferencia que acceden a los códigos de condición.

Grupo II:

Un segundo grupo lo constituyen las instrucciones de proceso que utilizan los códigos de condición como operando de entrada adicional para realizar una transformación de los datos. La Tabla 2 enumera estas instrucciones agrupadas en tres clases: ajustes (ASCII y decimal), sumas-restas y rotaciones a través de la bandera de acarreo. Obsérvese que además de leer algunos códigos de condición (acarreo C y acarreo auxiliar A), también los escriben dando cuenta de cómo es el resultado de la operación de proceso.

Grupo II												
	lectura						escritura					
Ajustes	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
AAA				X						X		X
AAD									X	X		X
AAM								X	X		X	
AAS				X						X		X
DAA				X		X	X	X	X	X	X	X
DAS				X		X		X	X	X	X	X
Suma/resta	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
ADC						X	X	X	X	X	X	X
SBB						X	X	X	X	X	X	X
Rotaciones	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
RCL						X	X					X
RCR						X	X					X

Tabla 2. Instrucciones de proceso que leen y escriben códigos de condición.

El empleo de instrucciones de ajuste tiene que ver con la codificación BCD. Por otra parte, el uso de operaciones de suma y resta a través de la bandera de acarreo está relacionado con la ejecución de operaciones extendidas a un ancho de palabra doble. El empleo, hoy en día, de ambos tipos de instrucciones debería ser muy escaso, puesto que la codificación BCD no se usa directamente y los tamaños de palabra de los computadores de propósito general son suficientemente anchos para la mayor parte de las aplicaciones de uso corriente.

Grupo III:

En el tercer grupo encontramos las instrucciones de proceso que solamente acceden a los códigos de condición en escritura para dejar constancia de cómo ha quedado el resultado después de la operación. Están constituidas por dos grandes categorías: las aritméticas y las lógicas. La Tabla 3 especifica cuales son y qué códigos de condición se escriben en cada caso. Pueden ser responsables, a través del registro de estado, de dependencias verdaderas (con saltos condicionales u otras de proceso que lean el estado) y de dependencias de salida.

Grupo III												
	lectura						escritura					
Aritméticas	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
ADD							X	X	X	X	X	X
CMP							X	X	X	X	X	X
DEC							X	X	X	X	X	
DIV							X	X	X	X	X	X
IDIV							X	X	X	X	X	X
IMUL							X	X	X	X	X	X
INC							X	X	X	X	X	
MUL							X	X	X	X	X	X
NEG							X	X	X	X	X	X
SUB							X	X	X	X	X	X
Lógicas	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
AND							X	X	X		X	X
OR							X	X	X		X	X
ROL							X					X
ROR							X					X
SHL/SAL							X	X	X	X	X	X
SAR							X	X	X	X	X	X
SHR							X	X	X	X	X	X
TEST							X	X	X	X	X	X
XOR							X	X	X		X	X

Tabla 3. Instrucciones de proceso que solo acceden a los códigos de condición en escritura.

Grupo IV:

El cuarto grupo lo constituyen las instrucciones de salto condicional. Son instrucciones que únicamente acceden a los códigos de condición en lectura. Su fin es evaluar si se cumple o no la condición de salto realizando una comparación sobre uno o más de esos códigos de condición (con diferentes operadores de relación: AND, OR según el caso).

La Tabla 4 detalla los tipos se salto condicional y las banderas evaluadas en cada caso. Se han sombreado las condiciones que son complementarias de otras previas ya que, en realidad, sólo hay 8 tipos de accesos a los códigos de condición: en unos casos buscando un valor '0' y en otros un valor '1'. Obsérvese cómo la bandera de acarreo auxiliar (A) no se utiliza nunca como entrada en este grupo (su uso se restringe a las operaciones de ajuste del Grupo II). Generan dependencias verdaderas con aquellas instrucciones que escriben estado.

Grupo IV												
	lectura						escritura					
Bifurcaciones	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
JB/JNAE						X						
JBE/JNA			X			X						
JE/JZ			X									
JL/JNGE	X	X										
JLE/JNG	X	X	X									
JNB/JAE						X						
JNBE/JA			X			X						
JNE/JNZ			X									
JNL/JGE	X	X										
JNLE/JG	X	X	X									
JNO	X											
JNP/JPO					X							
JNS		X										
JO	X											
JP/JPE					X							
JS		X										

Tabla 4. Instrucciones de salto condicional.

Grupo V:

En el quinto y último grupo encontramos otras instrucciones que acceden a los códigos de condición:

los bucles por condición, los prefijos de repetición en operaciones con cadenas (bucles en realidad), instrucciones de exploración y comparación en cadenas, la interrupción condicional de desbordamiento y las instrucciones de manejo de la bandera de acarreo. La Tabla 5 da cuenta de la relación de instrucciones del grupo.

Grupo V												
	lectura						escritura					
Bucles por condición	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
LOOPNZ/LOOPNE			X									
LOOPZ/LOOPE			X									
prefijos de repetición	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
REPZ/REPE			X									
REPNZ/REPNE			X									
cadenas	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
CMPS							X	X	X	X	X	X
SCAS							X	X	X	X	X	X
interrupciones	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
INTO	X											
bandera de acarreo	**O**	**S**	**Z**	**A**	**P**	**C**	**O**	**S**	**Z**	**A**	**P**	**C**
CLC												X
CMC						X						X
STC												X

Tabla 5. Otras instrucciones que acceden a los códigos de condición.

En conclusión, en todos los casos el acceso a los códigos de condición se realiza de manera implícita, es decir, depende exclusivamente de la operación realizada sin permitir ninguna intervención del programador, e inevitable, es decir, no puede dejar de hacerse cuando no tiene significado computacional.

Las instrucciones JCXZ, LOOP y REP son espaciales ya que, aunque representan operaciones de control de flujo, la bifurcación condicional no se realiza en función de los códigos de condición si no en función del valor del registro CX (si ha alcanzado el cero o no), siendo, en realidad, una bifurcación condicional del tipo de las que realizan la evaluación de un registro en la propia instrucción de salto.

5. Dispositivo experimental: metodología y banco de pruebas

La evaluación del impacto de los códigos de condición sobre el procesamiento superescalar se puede abordar desde dos puntos de vista complementarios: por un lado, un estudio estadístico basado en las distribuciones de uso tanto de instrucciones como de datos y, por otro, un estudio analítico del acoplamiento del código basado en la teoría de grafos.

a. Estudio estadístico

Abordaremos el estudio estadístico obteniendo recuentos con el fin de encontrar la distribución detallada de instrucciones que acceden a los códigos de condición.

Puesto que el primer uso de los códigos de condición es la evaluación de situaciones de salto condicional, parece que la secuencia de programa acotada por el bloque básico plantea un escenario muy conveniente para intentar obtener alguna correlación entre lo que en ella sucede y el rendimiento del programa en su conjunto.

En resumen, a partir el análisis estadístico intentaremos, por tanto, predecir el comportamiento de los programas, en lo que al acoplamiento por dependencias de datos se refiere, utilizando los mapas de acceso al registro de estado y el comportamiento restringido al bloque básico.

b. Estudio analítico

A continuación, con el fin de obtener una cuantificación precisa, aplicaremos el modelo matemático basado en la teoría de grafos propuesto en [11]. Se trata de una formalización adaptada al paralelismo a nivel de instrucción y orientada al estudio de las dependencias de datos. Esta técnica nos permite trabajar con rigor matemático una vez que describimos las secuencias de código con una formalización matricial. Se han determinado las restricciones y propiedades que ha de cumplir la matriz de dependencias de datos *D* así como las operaciones que sobre ella se pueden realizar y la información que de ella se puede obtener. La exposición detallada en profundidad y extensión se puede encontrar en [10].

Una de las herramientas más útiles que nos proporciona la formalización matricial de éste modelo es la posibilidad de componer las diversas fuentes de dependencias. Así, aislando las contribuciones debidas a los diferentes tipos de datos podemos estudiar su impacto, la clase de dependencias de datos que genera y estimar su peso relativo al conjunto.

c. Banco de pruebas

Como banco de pruebas utilizaremos el mismo conjunto de programas propuesto en el trabajo mencionado anteriormente [20], que incluye una cuantificación del impacto de los códigos de condición sobre el grado de paralelismo disponible. De esta manera, se pretende comprobar los resultados presentados en él así como aportar conclusiones más detalladas.

Es un banco de pruebas de enteros compilado para la plataforma DOS en modo real que incluye varias utilidades del sistema operativo (*comp*, *find* y *debug*) así como aplicaciones de uso común: el compresor-descompresor de ficheros *rar* (versión 1.52) y el compilador de lenguaje C *tcc* (versión 1.0). El programa *go* de los SPECint95 también ha sido incluido bajo 2 compilaciones: una optimizada en tamaño y otra optimizada en velocidad. En [18] se puede encontrar una memoria detallada sobre el banco de pruebas.

La obtención de las trazas de ejecución se ha realizado utilizando el modo de ejecución paso a paso. Las cargas de trabajo de los diferentes programas del banco de pruebas han sido seleccionadas para los recuentos de instrucciones procesadas no sean demasiado grandes. A pesar de ello se han trazado casi 190 millones de instrucciones.

Es importante hacer notar que, al trabajar con trazas de instrucciones realmente procesadas, los saltos condicionales son seguidos de la instrucción efectivamente tomada durante la ejecución del programa. Esto nos permitirá analizar secuencias de

código tan grandes como deseemos en la seguridad de tener la predicción perfecta del salto.

6. Distribución de instrucciones que acceden a los códigos de condición

a. Estudio analítico

Se ha realizado un análisis detallado de la mezcla de instrucciones que encontramos en los programas del banco de pruebas. El conjunto completo de los datos de dicho estudio pueden ser examinados en [19]. Se puede comprobar que son consistentes con los aparecidos en trabajos similares en la literatura. Concretamente con los ofrecidos por [1, 13].

Aquí, nosotros estamos interesados en conocer solamente la distribución de instrucciones que acceden a los códigos de condición. Queremos determinar a

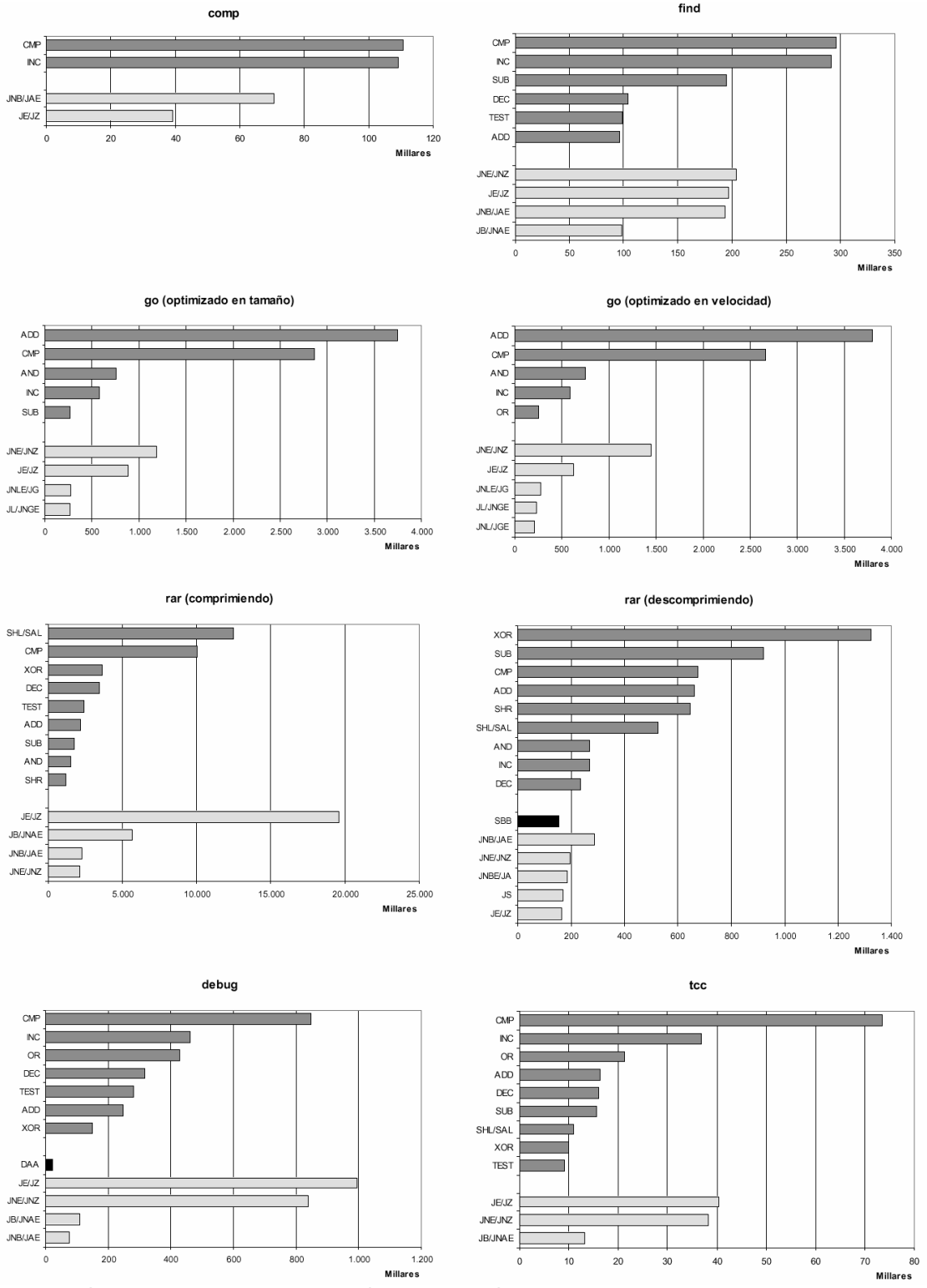


Fig. 4. Distribución de las instrucciones que manejan códigos de condición. El primer grupo corresponde a las instrucciones que escriben códigos de condición (en gris oscuro). El segundo grupo está constituido por las instrucciones que leen códigos de condición. En negro las que los usan como operando y en gris claro las de salto condicional, que los usan para evaluar una condición de salto.

qué grupo de los detallados en la Sección 4 pertenecen, el tipo de acceso realizado (lectura y/o escritura) y su función. El objetivo es determinar qué tipo de acoplamientos se van a producir en las instrucciones por dependencias entre códigos de condición.

La Fig. 4 muestra los resultados. Se observa que las instrucciones que acceden a los códigos de condición efectivamente tan sólo pertenecen al Grupo III, al Grupo IV y esporádicamente y únicamente en dos programas del banco de pruebas, al Grupo II.

En todas las trazas aparecen dos grupos importantes: el grupo III y el grupo IV. El grupo III corresponde a las instrucciones que escriben en los códigos de condición para dejar constancia de cómo queda el resultado de una operación de proceso. En ningún caso los leen. En grupo IV está constituido por las instrucciones de salto condicional. Su función es leer los códigos de condición con el fin de tomar una decisión evaluando una situación. En consecuencia, podemos afirmar que el uso real que le dan los programas a los códigos de condición del repertorio x86 es el inicialmente previsto y descrito en la Sección 3 de este trabajo, es decir, pasar información a una instrucción de salto condicional. En este sentido, cobra especial relevancia la secuencia de programa acotada por el bloque básico.

Las 2 instrucciones del grupo II en la traza de *rar descomprimiendo* (SBB) y *debug* (DAA) tienen como entrada de la operación de proceso códigos de condición (bandera de acarreo C y bandera de acarreo auxiliar A). Puesto que realizan una lectura del registro de estado, generan dependencias verdaderas con las instrucciones que previamente accedan a los códigos de condición en escritura.

A la vista de los resultados expuestos en la Fig. 4, podemos decir que el uso de los códigos de condición prácticamente se restringe a recoger información de estado de cara a la toma de decisiones en los saltos condicionales.

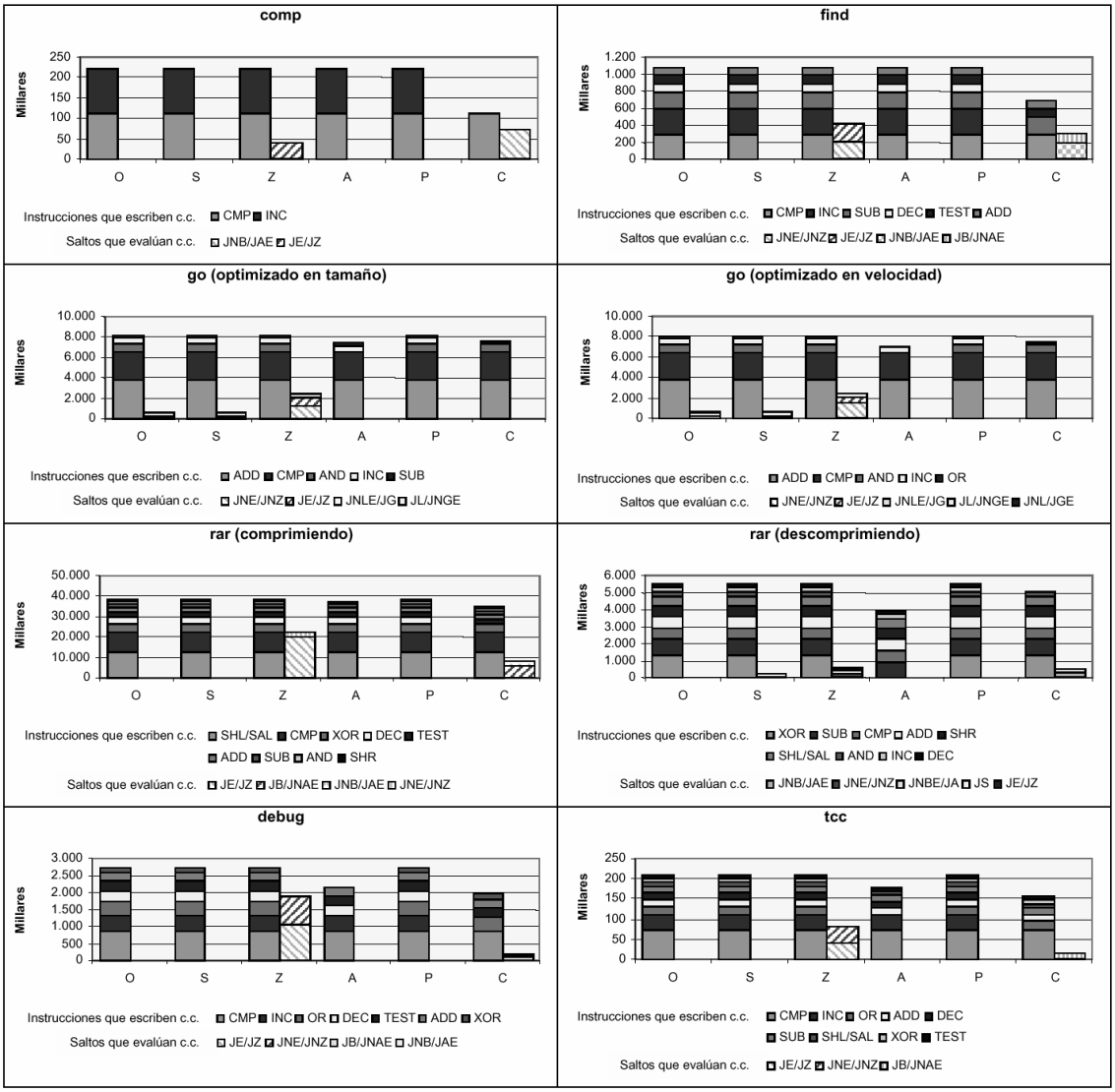


Fig. 5. Contribución por instrucciones a los accesos en lectura y escritura a los códigos de condición.

Si atendemos solamente a las instrucciones relativas a los saltos condicionales, la distribución de accesos a los códigos de condición es la presentada en la Fig. 5. En las gráficas se han señalado las instrucciones que contribuyen a los accesos en escritura y lectura. Como se puede observar las escrituras de códigos de condición superan en cantidad a las lecturas y afectan a más códigos de condición de los que realmente se necesitan para tomar decisiones en las bifurcaciones condicionales. En general las escrituras afectan a la gran mayoría de las banderas de estado con escasas excepciones para el acarreo auxiliar (A) y el acarreo (C). En cuanto a las lecturas encontramos que las condiciones evaluadas pueden afectar a una única bandera o a 2 o incluso 3 banderas simultáneamente. Sin embargo, la distribución más frecuente corresponde a la lectura de la bandera de cero (Z) o la bandera de acarreo (C).

Tan sólo en los casos de los programas *rar comprimiendo* y *debug* la cantidad de lecturas de algún código de condición (en ambos casos la bandera de cero Z) supera el 50% de las escrituras realizadas sobre ese mismo código de condición.

b. El acoplamiento en el bloque básico

Un bloque básico se define como una secuencia de código sin bifurcaciones [2, 3, 4, 12]. Esta estructura es bien conocida y utilizada con frecuencia en teoría de compiladores ya que las sentencias de un bloque básico constituyen una unidad sobre la cual se aplican las optimizaciones locales. En el estudio del impacto de los códigos de condición sobre la disponibilidad de paralelismo también representa un escenario adecuado al fin de interpretar el comportamiento de los diferentes tipos de acoplamientos de datos. En este sentido, de la mezcla de instrucciones del repertorio que componen la secuencia del bloque básico a nosotros solamente nos interesan las que acceden a los códigos de condición. El objetivo es conocer de qué manera los accesos a códigos de condición imponen un orden de precedencia en la ejecución.

La Tabla 6 muestra los diferentes promedios de tamaño del bloque básico (número de instrucciones que lo componen) para cada traza de los programas del banco de pruebas así como el promedio de instrucciones de proceso que se pueden encontrar en cada bloque básico.

Programa	Instrucciones por BB	Instrucciones de proceso por BB
Comp	6,00	1,94
Find	7,68	1,56
Go tamaño	10,31	3,14
Go velocidad	10,16	2,89
Rar comprimiendo	3,19	1,25
Rar descomprimiendo	12,56	5,52
Debug	3,92	1,35
Tcc	8,98	2,28

Tabla 6. Promedios del tamaño de instrucciones de proceso por bloque básico para cada programa del banco de pruebas.

Estadísticamente no se puede concluir cual es la mezcla de instrucciones que compone el bloque básico tipo en cada traza, con la excepción de *comp* en el que la mezcla es tan reducida que podemos afirmar casi con un 100% de probabilidad que el bloque básico cuenta entre sus 6 instrucciones de tamaño promedio con CMP e INC como operaciones que acceden al registro de estado. La estimación es consistente con la funcionalidad del programa.

Cada bloque básico contiene necesariamente, teniendo en cuenta solamente las dependencias debidas a códigos de condición, un acoplamiento por dependencia de datos verdadera entre la última instrucción que escribe el registro de estado y la bifurcación condicional que lo evalúa. Desde un punto de vista estrictamente estadístico, a mayor número de bloques básicos mayor número de acoplamientos verdaderos debidos a códigos de condición. O lo que es igual, aumentan las dependencias verdaderas provocadas por códigos de condición cuanto menor sea el tamaño del bloque básico. De los datos mostrados en la Tabla 6 se deduce que los programas *rar comprimiendo* y *debug*, con los bloques básicos más pequeños del banco de pruebas, son los candidatos a experimentar una mejora mayor en ausencia de acoplamientos por códigos de condición. Esta hipótesis es avalada por los resultados presentados en [20].

Otro tipo de acoplamiento a través de los códigos de condición típico del bloque básico es el correspondiente a las dependencias de datos de salida entre sucesivas instrucciones de proceso. La pérdida de paralelismo por este concepto es computable solamente a la arquitectura del repertorio de instrucciones y no tiene, en modo alguno, significado computacional. Un buen ejemplo, es el caso de la traza del programa *comp* del que hemos hablado anteriormente. La tarea de comparación que realiza se implementa entre la instrucción CMP y el salto condicional. Sin embargo, el bloque básico cuenta con otra instrucción de proceso (INC), que sirve para actualizar un puntero, pero que lateralmente genera una dependencia de datos de salida adicional con la operación de comparación (CMP).

Estadísticamente, la longitud promedio de las cadenas de acoplamientos por dependencias de salida crece con el número de instrucciones de proceso por bloque básico. Los datos de la Tabla 6 señalan como candidatos a tener un gran acoplamiento por dependencias de salida a los programas *rar descomprimiendo* y *go*. Los bloques básicos más grandes también son los que potencialmente pueden contener más instrucciones de proceso.

Finalmente, las dependencias verdaderas entre una instrucción de escritura en el registro de estado y una instrucción de proceso que consuma dicha información de estado, son prácticamente inexistentes a la vista de lo mostrado en la Fig. 4.

En resumen, a la luz de las distribuciones de uso y del comportamiento descrito en el bloque básico, es muy razonable afirmar que:

- Un tamaño grande del bloque básico disminuye el riesgo de dependencias verdaderas debidas a códigos de condición.
- Un tamaño grande del bloque básico puede aumentar la longitud de las cadenas de dependencias de salida por códigos de condición.

7. El método de evaluación cuantitativa derivado de la teoría de grafos

El estudio estadístico nos proporciona una aproximación cualitativa al conocimiento del impacto de los códigos de condición sobre la ejecución superescalar. En esta Sección evaluaremos cuantitativamente dicho impacto mediante la aplicación de un método derivado de la teoría de grafos.

La evaluación cuantitativa de las arquitecturas de los repertorios de instrucciones basada en la aplicación de la teoría de grafos tiene grandes ventajas:

- proporciona una descripción sencilla de los fenómenos
- permite predecir el comportamiento
- simplifica la transferencia de conocimientos
- desacopla el estudio de la características del repertorio de instrucciones del hardware que lo interpreta

Particularmente interesante resulta el último punto ya que permite estudiar las diversas capas del proceso de computación aisladamente cosa que no sucede en los métodos basados en simuladores. Hoy en día la simulación se ha convertido en la primera herramienta de evaluación pero en los simuladores se produce un acoplo, inherente al método, que une el comportamiento del repertorio con el comportamiento de la capa física. Es deseable encontrar métodos alternativos con los que podamos realizar una cuantificación del rendimiento aislada [22].

Para conseguir la aplicación de la teoría de grafos al paralelismo de grano fino con el mismo éxito con el que se ha hecho en otros campos de la computación se requiere la definición de una serie de propiedades, restricciones y métodos de cuantificación. A partir de la definición de la matriz de dependencias D y estableciendo una serie de restricciones y propiedades adecuadas al nivel de instrucción somos capaces de introducir una serie de parámetros calculados matemáticamente que caracterizan el código desde el punto de vista de la disponibilidad de paralelismo. El desarrollo matemático se puede consultar con mayor amplitud y profundidad en [10, 11]. Resumidamente, tenemos:

● La matriz de dependencias de datos D formaliza matemáticamente el grafo de dependencias de datos correspondiente a una determinada secuencia de código, de acuerdo a la siguiente definición:

$$d_{ij} = \begin{cases} 1, & \text{if } i \text{ instruction depends on } j; \\ 0, & \text{otherwise.} \end{cases} \qquad (1)$$

Normalmente, la matriz de dependencias de datos D se evalúa para una secuencia de código de n instrucciones conocida como ventana de instrucciones.

● El acoplamiento C mide la cantidad de dependencias de datos presentes en una ventana de instrucciones. Se obtiene de la matriz D y está acotado según se indica a continuación:

$$C = \sum_{i=0}^{n-1} \sum_{k=0}^{n-1} d_{ik} \qquad 0 \le C \le \binom{n}{2} \qquad (2)$$

● La longitud del camino crítico L cuantifica a partir de la matriz D lo larga que es la cadena de dependencias de datos más larga que podemos encontrar en la ventana de instrucciones:

$$L = l \text{ computation steps if and only if } D^l = \mathbf{0} \qquad (3)$$

● El grado de paralelismo G_p es una magnitud derivada de L que representa la cantidad de paralelismo disponible que podemos encontrar en la secuencia de código de la ventana de n instrucciones. Su expresión y acotamiento se dan seguidamente:

$$G_p = n / L \qquad G_p \in [1, n] \qquad (4)$$

● Una de las herramientas más potentes que nos ofrece el método se deriva de la propiedad según la cual la matriz D, correspondiente a la contribución completa de todas las fuentes de dependencias de datos de una secuencia de código, es la composición, de acuerdo a la siguiente expresión, de todas y cada una de las matrices D_{si} obtenidas al contabilizar las contribuciones parciales de todas las fuentes de dependencias aisladas, *si*, que afectan a la secuencia de código.

$$D = D_{s1} \text{ OR } D_{s2} \text{ OR } \ldots \text{ OR } D_{sn} \qquad (5)$$

● Por otro lado, sabemos que la longitud del camino crítico L correspondiente a la contribución completa de las fuentes de dependencias de datos que afectan a la ventana de código en cuestión, está acotada en función de las longitudes de los caminos críticos L_{si} debidas a las fuentes de dependencias de datos parciales de acuerdo a la expresión siguiente:

$$\max_i \{L_{si}\} \le L \le \min\left\{\sum_i L_{si}, n\right\} \qquad (6)$$

Según esta ecuación, el grado de paralelismo resultante de la composición de todas las posibles fuentes de dependencias de datos nunca será mejor que el que represente una de sus componentes.

a. Combinaciones de fuentes de dependencias de datos

El mapa completo de las posibles fuentes de dependencias de datos se ha de elaborar adecuándolo al objetivo del estudio. Luego, evaluando las composiciones resultantes de tener en cuenta o de excluir las diversas contribuciones de fuentes de dependencias de datos conseguimos un conocimiento preciso de su relevancia.

En nuestro caso, el objetivo es evaluar el impacto de los códigos de condición. Así pues, contamos con 2 tipos

de datos: los códigos de condición y el resto. Construiremos matrices de dependencias de datos que tengan en cuenta ambas contribuciones o sólo una de ellas.

Por otra parte, las dependencias de datos se pueden manifestar como:

- Dependencias verdaderas: lectura después de escritura
- Antidependencias: escritura después de lectura
- Dependencias de salida: escritura después de escritura.

Hay que tener en cuenta que las antidependencias y las dependencias de salida son dependencias no verdaderas ya que, en realidad, son producidas por la utilización del mismo recurso físico; basta con cambiar el recurso físico para que se deshagan.

La Tabla 7 muestra el listado completo de composiciones de dependencias de datos en función de la contribución o no de cada una de las categorías del mapa de dependencias de datos. Puesto que tenemos 5 posibles contribuciones a tener o no en cuenta, existen 2^5 (32) posibles combinaciones.

		Tipo de dependencia			**Tipo de dato**	
Id		Verdadera	Anti	Salida	Códigos de condición	Otros
	a	✓	✓	✓	✓	✓
1	**b**	✓	✓	✓	✓	∅
	c	✓	✓	✓	∅	✓
		✓	✓	✓	∅	∅
	a	✓	✓	∅	✓	✓
4	**b**	✓	✓	∅	✓	∅
	c	✓	✓	∅	∅	✓
		✓	✓	∅	∅	∅
	a	✓	∅	✓	✓	✓
3	**b**	✓	∅	✓	✓	∅
	c	✓	∅	✓	∅	✓
		✓	∅	✓	∅	∅
	a	✓	∅	∅	✓	✓
5	**b**	✓	∅	∅	✓	∅
	c	✓	∅	∅	∅	✓
		✓	∅	∅	∅	∅
	a	∅	✓	✓	✓	✓
2	**b**	∅	✓	✓	✓	∅
	c	∅	✓	✓	∅	✓
		∅	✓	✓	∅	∅
	a	∅	✓	∅	✓	✓
6	**b**	∅	✓	∅	✓	∅
	c	∅	✓	∅	∅	✓
		∅	✓	∅	∅	∅
	a	∅	∅	✓	✓	✓
7	**b**	∅	∅	✓	✓	∅
	c	∅	∅	✓	∅	✓
		∅	∅	✓	∅	∅
		∅	∅	∅	✓	✓
		∅	∅	∅	✓	∅
		∅	∅	∅	∅	✓
		∅	∅	∅	∅	∅

Tabla 7. Listado de posibles composiciones de dependencias de datos en función de la contribución o no de cada una de las categorías. El identificador es el mismo utilizado para presentar los resultados.

No todas las combinaciones posibles tienen significado real ya que cuando no se considera la contribución de ningún tipo de datos o cuando no se considera la contribución de ningún tipo de dependencia, no hay realmente ninguna composición. En la Tabla 7 se han sombreado las combinaciones posibles. Así, de las 32 combinaciones solamente existen 21. Estas 21 se pueden agrupar en 7 clases a partir de las diferentes combinaciones de contribución por tipo de dependencia. Podemos considerar todos los tipos de dependencias o bien excluir una cada vez o bien tener en cuenta solamente una de ellas. Estos 7 grupos, quedan entonces como sigue:

1) contribución de todos los tipos de dependencias
2) contribución exceptuando las verdaderas (o contribución únicamente de las no verdaderas)
3) contribución exceptuando las antidependencias
4) contribución exceptuando las de salida
5) contribución únicamente de las verdaderas
6) contribución únicamente de las antidependencias
7) contribución únicamente de las de salida

De todos los grupos enumerados, el grupo 3 y 4 no aporta información relevante ya que combina la contribución de un tipo de dependencias verdadero con una de las clases de dependencias no verdaderas. Estos grupos los vamos a evitar en nuestro análisis.

A cada uno de los 5 grupos útiles (1, 2, 5, 6, 7) se le puede descomponer, a su vez, en tres más dependiendo de los tipos de datos cuya contribución se tome en consideración. Así, podemos componer la contribución al acoplamiento de todos los tipos de datos o bien solamente la debida a los códigos de condición o solamente la debida al resto de datos. Es decir, tendremos las siguientes posibilidades:

a. contribución de todos los tipos de datos
b. contribución solamente de los códigos de condición
c. contribución solamente de los que no son códigos de condición

Para cada grupo básico disponemos, por tanto, de información sobre la contribución conjunta de todos los datos, de la contribución aislada de los códigos de condición y de la contribución debida al resto de datos excluyendo los códigos de condición.

Jugando con las diferentes composiciones, podemos obtener información valiosa del impacto efectivo de los códigos de condición en programas reales. La Ecuación 6 nos asegura que la contribución a la longitud del camino crítico de una determinada fuente de dependencias de datos establece una mínima cota inferior para el acoplamiento de la composición completa. Luego, tomaremos como cota inferior efectiva de la composición completa el máximo valor de las longitudes de los caminos críticos de todas las posibles contribuciones parciales.

Así, si tomamos en consideración la contribución aislada de los códigos de condición conoceremos el acoplamiento de las instrucciones por esta fuente y determinaremos una cota de impacto que el resto de fuentes de dependencias pueden igualar o asumir en la composición pero nunca disminuir. Cuando determinamos la contribución parcial del resto de datos en ausencia de los códigos de condición, obtenemos una cuantificación del impacto del resto de datos y también una información sobre la relevancia de la contribución excluida en función de la posible mejora obtenida.

identificador de la composición		identificador de la composición	
1	a b c	TODAS	Todos los datos Sólo cc No cc
5	a b c	VERDADERAS	Todos los datos Sólo cc No cc
2	a b c	NO VERDADERAS	Todos los datos Sólo cc No cc
6	a b c	ANTIDEPENDENCIAS	Todos los datos Sólo cc No cc
7	a b c	DE SALIDA	Todos los datos Sólo cc No cc

Tabla 8. Listado de identificadores de las composiciones y contribuciones efectivas para cada uno.

8. Cuantificación del impacto del acceso a códigos de condición

La aplicación del método de evaluación cuantitativa descrito en la Sección 7 se ha automatizado desarrollando los programas adecuados al efecto [27]. Estos programas permiten analizar ventanas de código de tamaño variable obtenidas a partir de trazas de ejecución de programas. Generan las matrices de dependencias de datos correspondientes a una plantilla de contribuciones establecida al comienzo del análisis y determinan los parámetros presentados anteriormente para cada una de ellas y para la composición.

Las ventanas de instrucciones son estáticas de tamaño 512 instrucciones. El uso de ventanas deslizantes, propio de la capa física de procesadores reales y utilizado en simuladores, resulta muy pesado para las aplicaciones de análisis automatizado y, dado un tamaño suficiente de ventana, no aporta ninguna precisión. Se han realizado los análisis para ventanas de tamaño 2048 instrucciones obteniendo prácticamente los mismos resultados en un tiempo de cómputo considerablemente mayor. Por otra parte, en la literatura se encuentran trabajos que confirman que, para tamaños grandes de ventana, la información obtenida a partir de ventanas deslizantes y ventanas estáticas son iguales [26]. Cualitativamente, también hay razones para considerar que los efectos en los bordes derivados de tomar en consideración una ventana estática se hacen irrelevantes cuando la ventana es grande ya que el número de ubicaciones posibles para los datos es muy pequeño en comparación (asumiendo que consideramos a la memoria como un recurso único).

a. Longitud del camino crítico

La Tabla 9 muestra los datos obtenidos para la longitud del camino crítico para cada una de las trazas del banco de pruebas y diferentes composiciones de dependencias de datos.

identificador de la composición		comp	find	go (t)	go (v)	rar (c)	rar (d)	debug	tcc
1	a	398,70	321,90	306,98	311,72	419,67	405,71	390,39	336,11
	b	257,53	150,36	197,00	199,22	341,78	240,05	312,34	170,23
	c	337,30	304,35	273,23	274,56	238,99	324,11	247,66	289,81
5	a	248,32	229,91	94,30	85,14	132,12	177,16	132,41	151,87
	b	2,01	1,98	2,00	2,00	2,00	2,28	2,81	2,28
	c	247,51	229,83	92,88	83,65	131,40	175,96	124,35	147,76
2	a	233,41	271,40	215,56	220,97	247,30	320,78	258,55	237,68
	b	172,48	116,59	146,13	147,63	214,59	204,98	192,59	119,09
	c	177,85	261,52	161,10	161,86	212,68	230,88	238,80	211,57
6	a	148,86	212,85	110,81	112,82	137,22	189,31	137,25	175,41
	b	2,01	1,98	2,00	2,00	2,49	10,02	4,11	3,52
	c	148,65	212,04	108,48	109,91	136,04	183,69	132,72	174,17
7	a	178,34	230,24	175,53	176,87	230,57	235,39	248,43	182,82
	b	172,29	116,43	146,03	147,53	214,26	204,87	192,28	118,98
	c	147,62	228,36	136,96	139,04	197,98	135,48	234,92	164,16

Tabla 9. Longitud del camino critico, medida en pasos de computación, para una ventana estática de 512 instrucciones y diferentes composiciones de fuentes de dependencias de datos en cada una de las trazas del banco de pruebas.

Las longitudes de los caminos críticos son mayores, en general, cuantas más contribuciones de fuentes de dependencias de datos se toman en cada composición. Sin embargo, en el caso del grupo 5 (contribución parcial de las dependencias verdaderas) y del grupo 6 (contribución parcial de las antidependencias) la contribución aislada de los códigos de condición (subgrupo b) resulta irrelevante. El caso señalado da lugar una longitud próxima a los 2 pasos de computación para casi todas las trazas. Una longitud de 2 pasos de computación significa que existen solamente 2 conjuntos de instrucciones: un conjunto de instrucciones no dependientes generadoras de datos que son directamente consumidos por el segundo grupo de instrucciones dependientes. En el grupo 5b estamos ante las parejas de instrucciones formadas por la *escritura de estado-evaluación de estado* interna al bloque básico y en el grupo 6b estamos ante las parejas de instrucciones formadas por la *evaluación de estado-escritura de estado* entre bloques básicos consecutivos.

b. Grado de paralelismo G_p

De la Tabla 9 podemos derivar el grado de paralelismo G_p, según la Ecuación 4, para cada una de

las diferentes composiciones de dependencias de datos. La Fig. 6 muestra el grado de paralelismo que presenta cada traza cuando se tiene en cuenta la contribución de todas las fuentes de dependencias.

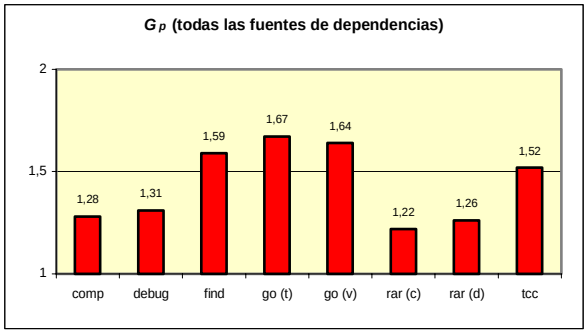
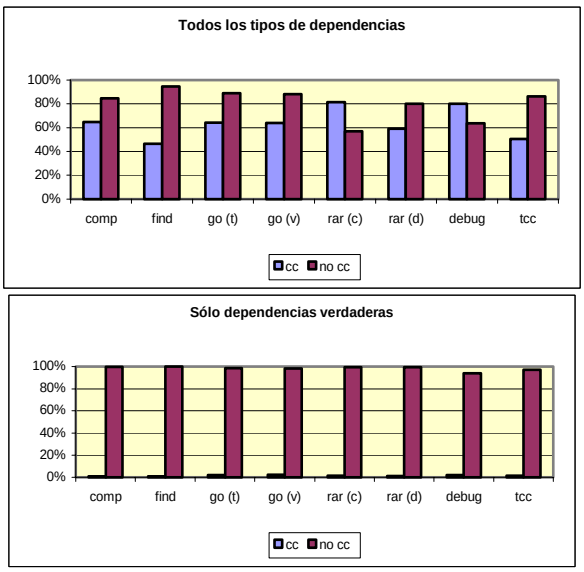


Fig 6. Grado de paralelismo G_p para cada traza del banco de pruebas cuando se tiene en cuenta la contribución de todas las fuentes de dependencias de datos.

Los resultados del grado de paralelismo G_p mostrados en la Fig. 6 son coherentes con los presentados en otros trabajos realizados sobre máquinas de repertorio de instrucciones x86 [5, 13, 14, 17, 20, 23]. Esta circunstancia puede considerarse como un argumento más a favor de la validación del método.

c. Análisis del impacto por tipos de dependencias

La Fig. 7 presenta las gráficas ilustrativas del impacto de los códigos de condición sobre la longitud del camino crítico cuando se tiene en cuenta su contribución o se excluye para las diferentes composiciones de tipos de dependencias de datos: todos los tipos, verdaderas, no verdaderas, antidependencias y salida. En cada gráfica el 100% del porcentaje corresponde a la longitud del camino crítico cuando se tiene en cuenta la contribución de todas las clases de datos (códigos de condición más el resto) solamente a la composición establecida de los tipos de dependencias de datos (verdaderas, antidependencias, etc.).



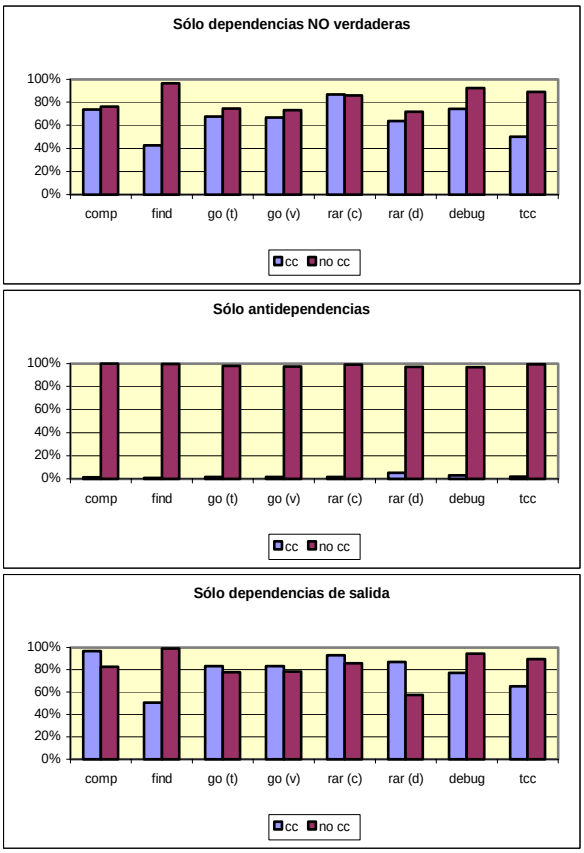


Fig. 7. Gráficas del impacto de los cc/no cc sobre los diferentes tipos de dependencias de datos.

En la primera gráfica se han dibujado las composiciones en las que se tienen en cuenta todos los tipos de dependencias de datos. Observamos como en los casos de las trazas de *rar comprimiendo* y *debug*, la longitud del camino crítico debida a la contribución aislada de los códigos de condición es mayor que la contribución aislada del resto de datos. La contribución al acoplamiento de los códigos de condición, en esos casos, representa un mínimo en la cota inferior de la composición completa que esconde cualquier posible mejora realizada sobre el resto de datos. Estos resultados son consistentes con los presentados en [20]. En el trabajo citado, la ausencia de dependencias debidas a los códigos de condición supone una mejora muy importante para esos mismos programas.

La segunda gráfica demuestra que la contribución de los códigos de condición a través de las dependencias verdaderas es prácticamente despreciable en comparación con la contribución del resto de datos a dicho tipo de dependencias de datos. Ahora bien, en la Tabla 9 vemos como la traza del programa *debug* presenta la mayor longitud del camino crítico para la contribución aislada de los códigos de condición a las verdaderas. Esta circunstancia está de acuerdo con la hipótesis introducida en la Sección 6 de que un tamaño reducido del bloque básico incrementa el riesgo de dependencias verdaderas a través de los códigos de condición. Por otra parte, queda demostrado que la muy baja frecuencia de uso de las instrucciones del

Grupo II (las de proceso que leen códigos de condición como un dato más de entrada) hace que no sea relevante su análisis desde el punto de vista de la ejecución superescalar.

La tercera gráfica ilustra el impacto de los códigos de condición a través de las dependencias no verdaderas. En este caso su contribución es comparable a la del resto de datos. En el caso de la traza de *rar comprimiendo*, la contribución al acoplamiento del código es incluso ligeramente mayor.

Las gráficas cuarta y quinta establecen cómo aparecen las dependencias no verdaderas por códigos de condición. Son producidas, básicamente a través de dependencias de salida. Comparativamente, la mayor contribución a favor de los códigos de condición por este concepto se da en el caso de la traza del programa *rar descomprimiendo*. Aparentemente, este hecho corrobora la hipótesis introducida en la Sección 6 según la cual un tamaño grande del bloque básico puede aumentar la longitud de las cadenas de dependencias de salida por códigos de condición.

d. Análisis del impacto por tipos de datos

La Fig. 8 muestra la contribución aislada de los códigos de condición a las diferentes composiciones de tipos de dependencias de datos. La Fig. 9 hace lo mismo con el resto de datos excluyendo los códigos de condición. En ambos casos el 100% del porcentaje corresponde a la longitud del camino crítico debido a la contribución de todos los datos y todos los tipos de dependencias de datos, es decir, la longitud del camino crítico de la composición completa.

La Fig. 8 ilustra el hecho de que básicamente la contribución de los códigos de condición se manifiesta en forma de dependencias de salida. Sin embargo, de manera general, la combinación de las dependencias verdaderas y antidependencias con las de salida provoca un alargamiento de las cadenas de dependencias. Es algo así como si las pocas dependencias que no son de salida favorecieran el enlazamiento de cadenas más pequeñas constituyendo una nueva más larga.

Otro hecho interesante a observar es que acoplamientos por dependencias de salida de magnitudes similares, cuando se combinan con el resto de tipos de dependencias de datos (verdaderas y antidependencias), provocan acoplamientos finales de muy diferentes magnitudes. En nuestro banco de pruebas, tienen valores similares la traza de *comp*, ambas trazas de *go*, ambas trazas de *rar* y la traza de *debug*, sin embargo, el valor resultante de la composición completa es mucho mayor en los casos de las trazas de *rar comprimiendo* y *debug*. Parece existir una correlación, por tanto, con el pequeño tamaño del bloque básico de esos dos programas del banco de pruebas.

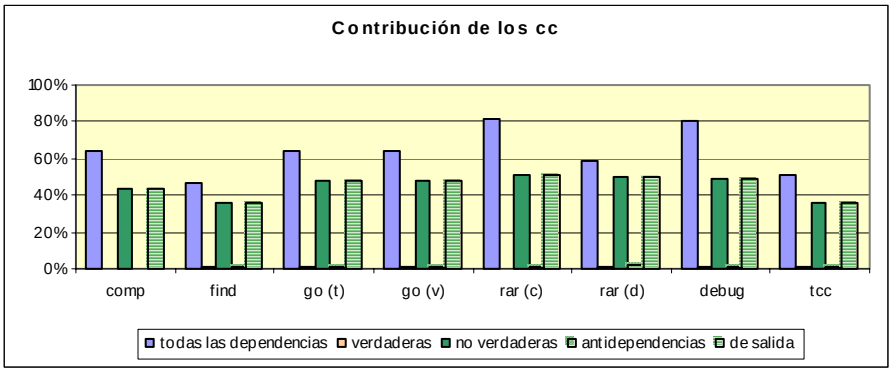


Fig. 8. Contribución aislada de los códigos de condición a las diferentes composiciones de tipos de dependencias de datos para cada traza del banco de pruebas.

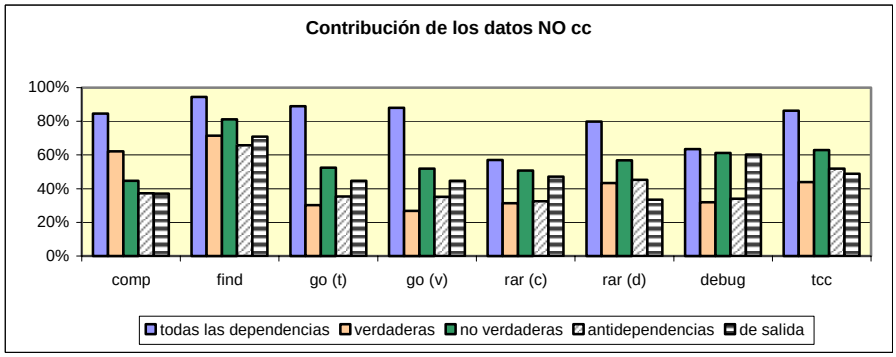


Fig. 9. Contribución aislada de los datos que no son códigos de condición a las diferentes composiciones de tipos de dependencias de datos para cada traza del banco de pruebas.

La Fig. 9 muestra una contribución más equilibrada entre los diferentes tipos de dependencias de datos cuando se trata de datos que no son códigos de condición. Las dependencias verdaderas tienen una relevancia semejante a las de las dependencias no verdaderas aunque, salvo en el caso del programa *comp*, éstas últimas son algo mayores. La composición de ambos tipos alarga las longitudes del camino crítico en todos los casos aunque de una manera bastante heterogénea.

9. Impacto en el nivel de microoperaciones

Los procesadores de la familia x86 han incrementado el rendimiento utilizando una microarquitectura de 2 niveles. El nivel superior trabaja como una interfase con el repertorio de instrucciones CISC, traduciendo éstas a microperaciones de tipo RISC que son procesadas en el nivel inferior. La decodificación se lleva a cabo en tres unidades diferentes: simple, general y secuenciador. Las instrucciones decodificadas por el secuenciador son ejecutadas serializadamente mientras que las provenientes de los otros dos decodificadores son ejecutadas en modo superescalar.

Huang y Peng han determinado la distribución del número de microoperaciones en las que se descompone una instrucción CISC en promedio para un banco de pruebas [13]. La Tabla 10 presenta dichos datos. La mayor parte de las instrucciones CISC (67%) se traducen en una única microoperación RISC y el resto hasta casi el 90% en 2. El valor medio ponderado es de 1,41 microoperaciones por instrucción CISC. Otros trabajos presentan datos muy similares: 1,26 en [14] y 1,35 en [5].

Microoperaciones por instrucción	
1	67%
2	22%
3	7%
4	3%
más de 5	<0’5%

Tabla 10. Distribución del número de microoperaciones en las que se descompone una instrucción CISC.

A la vista de estos resultados, podemos afirmar que la transformación de CISC a RISC tan sólo hace aumentar el número de nodos del grafo en algo menos de 1,5 veces, lo que significa que la estructura del grafo de dependencias de datos no se ve alterada notablemente.

Por otra parte, los acoplamientos de datos entre instrucciones se deben conservar a pesar de la transformación. La pregunta es, ¿cómo afecta la transformación a la longitud de las cadenas de dependencias? Para encontrar la respuesta debemos analizar a qué se debe el que una instrucción CISC se pueda descomponer en varias RISC. Básicamente es consecuencia de los modos de direccionamiento. Cuando un operando de una instrucción de proceso CISC reside en memoria, automáticamente se descompone la operación CISC en dos operaciones RISC del tipo carga/almacenamiento: el operando de memoria se carga/almacena en/desde registro y la operación de proceso RISC se realiza entre operandos situados en registros. En definitiva, lo único que sucede es que las cadenas de dependencias de datos se alargan en la misma proporción que aumenta el número de nodos del grafo. La Fig. 10 muestra las potenciales transformaciones del grafo CISC. La transformación “atómica”, es decir, sin división del nodo desde el punto de vista del acoplamiento, presentada en el caso (b) es la más probable.

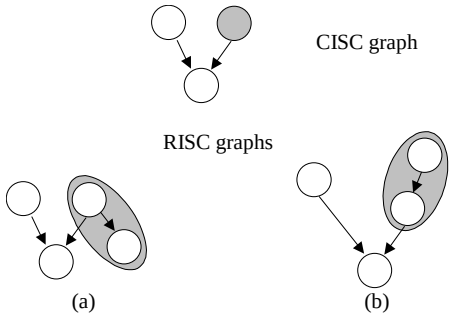


Fig. 10. Possible CISC to RISC graph transformations.

El ligero alargamiento de la longitud de los caminos de dependencias que se produce en la transformación RISC combinado con el aumento en la misma proporción del número de instrucciones a procesar en el grado RISC, tiene como consecuencia que el grado de paralelismo RISC permanezca básicamente inalterado respecto al paralelismo presente en la secuencia de código CISC.

El paralelismo aprovechable en tiempo de ejecución, como consecuencia de un modo de ejecución superescalar, está limitado por el grado de paralelismo disponible en la capa del lenguaje máquina, por la capacidad del *hardware* que lo interpreta para extraer paralelismo y por la disponibilidad de recursos en la capa física. Asumiendo que la capa física dispone de los recursos suficientes para aprovechar el paralelismo disponible en la capa de lenguaje máquina y que dicho grado de paralelismo no sufre cambios sustanciales al convertir código CISC a RISC, sólo nos queda analizar cuál es la capacidad de la capa física para extraer paralelismo.

Cuando hablamos de la capacidad del *hardware* de extracción de paralelismo, nos referimos a la capacidad de encontrar operaciones independientes, dispuestas a ser procesadas. En este caso, se trata de microoperaciones. Las dependencias de datos verdaderas a través de los códigos de condición, no pueden deshacerse más que por el procesamiento de las instrucciones precedentes según el orden parcial impuesto por el grafo de dependencias de datos. Las dependencias de datos no verdaderas pueden deshacerse gracias al mecanismo del renombramiento. La transformación CISC a RISC es un buen momento para aplicar el renombramiento. En el caso que nos ocupa, las dependencias no verdaderas debidas a los códigos de condición, se manifiestan prácticamente de manera íntegra en forma de dependencias de salida: escrituras después de escrituras. Sin embargo, hay que

tener en cuenta, por un lado, que el registro a renombrar (registro de estado) es especial y, por otro, que las dependencias de salida a través de los códigos de condición son, como ya se ha indicado, mera consecuencia de la arquitectura del repertorio de instrucciones pero sin ningún significado computacional.

10. Soluciones y su coste

Para terminar, se plantean algunas propuestas con el fin de minimizar el impacto de los códigos de condición del repertorio de instrucciones x86, analizando su coste, ventajas e inconvenientes. Podemos pensar en dos tipos de soluciones. Por un lado, implementaciones en la capa física y, por otro, extensiones el repertorio de instrucciones.

Las soluciones que vamos a proponer no suponen un cambio sustancial en la arquitectura del repertorio de instrucciones x86. Es decir, seguirá utilizando códigos de condición como medio de evaluar las operaciones de salto condicional y leerá el estado en aquellas instrucciones de proceso que lo requieran. En definitiva, las soluciones que proponemos se aplicaran a los casos en los que se producen dependencias de datos no verdaderas.

En cuanto a las implementaciones en la capa física, el renombramiento es una técnica sencilla y de comprobada solvencia. Se trataría de disponer de un banco de registros de estado tan grande como el tamaño de la propia ventana de instrucciones de la que se extraen instrucciones independientes para ser lanzadas. Cada nueva escritura en el registro de estado se realiza físicamente sobre un nuevo elemento de almacenamiento.

Aunque sencillo, el renombramiento es muy ineficiente ya que de todos los bits escritos en el registro de estado sólo son aprovechados (leídos) unos pocos, como demuestra la distribución de uso de los accesos a códigos de condición que hemos comentado con anterioridad. Además, usar el *hardware* para extraer paralelismo incrementa significativamente el coste en área de silicio, en complejidad de diseño y en consumo de potencia. Y para colmo, como se ha indicado en la Sección anterior, todo este coste es vano ya que las dependencias de salida a través de códigos de condición no acarrean ningún significado computacional. Estaríamos ante un caso más en el que incrementar los recursos del procesador [para soportar tolerancia a grandes latencias en ejecución fuera de orden] no produce beneficios ya que muchos de estos recursos permanecen sin utilizar [9] o se utilizan sin necesidad.

En cuanto a las soluciones basadas en extensiones al repertorio de instrucciones podemos aprovecharnos de la conversión CISC a RISC de la que hemos hablado con anterioridad. Como tenemos el condicionante de la compatibilidad binaria, el formato CISC no se puede variar, sin embargo, el comportamiento del núcleo RISC si puede cambiar. De lo que se trata es de habilitar un modo de escritura condicional en el registro de estado. Podemos utilizar la instrucción de 'no operación' (NOP) como código de escape. Una secuencia de 3 instrucciones NOP sirve para indicar al núcleo RISC que debe pasar a modo de escritura condicional sobre el registro de estado. En dicho modo, las microoperaciones nunca escriben en el registro de estado, aunque la instrucción CISC de la que provengan lo haga, a no ser que se indique explícitamente. Para dar la orden de escritura en el registro de estado se utiliza nuevamente el código de escape NOP incluido previamente a la instrucción CISC que debe actualizar el registro de estado. Será el compilador el encargado de incluir los códigos de escape de manera adecuada al significado computacional del programa.

Un programa escrito para un procesador con esta funcionalidad se podrá aprovechar de la mejora de rendimiento superescalar al evitar las dependencias de salida a través del registro de estado. El mismo programa podrá ser ejecutado en otro procesador compatible binario pero sin ésta mejora. En ese caso, sencillamente incluirá un recuento de instrucciones algo mayor que redundará en un ligero incremento de tiempo de procesamiento.

De manera recíproca, un programa que contenga la secuencia de escape pero que no haya sido compilado para un procesador con modo de escritura condicional, podría dar lugar a una ejecución errónea, por no escribir nunca el registro de estado, cuando se intenta correr en un procesador que sí dispone de ese modo de ejecución. Para prever esta eventualidad bastará con comprobar la existencia de un código de escape antes de la ocurrencia de la primera lectura del registro de estado.

11. Conclusiones

En la arquitectura del repertorio de instrucciones x86, los códigos de condición, además de utilizarse para evaluar las situaciones de salto condicional también son usados como operandos de entrada en algunas operaciones de proceso, aunque los estudios de distribución nos indican que dicho uso es residual.

Las distribuciones de acceso a los códigos de condición indican que se escriben más banderas de estado de las que luego se leen y, cada una de ellas, se escribe muchas más veces de las que es leída. Esta circunstancia va a resultar relevante desde el punto de vista del tipo de acoplamiento de las instrucciones por dependencias de datos.

En cuanto al bloque básico, podemos decir que un tamaño grande del mismo disminuye la probabilidad del acoplo por dependencias verdaderas aunque puede aumentar la longitud de las cadenas de dependencias de salida, en ambos casos a través de códigos de condición.

La cuantificación del impacto del acceso a los códigos de condición mediante el método de evaluación derivado de la teoría de grafos arroja los siguientes resultados. En primer lugar, los códigos de condición reducen la disponibilidad de paralelismo

fundamentalmente a través de dependencias de salida. Este tipo de dependencias de datos son evitables mediante técnicas de renombramiento. Ahora bien, estas escrituras sucesivas no tienen ningún significado computacional, son resultado, tan sólo, de la arquitectura del repertorio de instrucciones x86 y, por tanto, hacen que una solución *hardware* basada en renombramiento suponga un derroche absoluto de recursos.

En segundo lugar, el resto de dependencias de datos alargan la longitud de las cadenas de dependencias de datos. Se aprecia una correlación con el tamaño del bloque básico de forma que si éste es pequeño, el efecto de alargamiento es mayor debido a la contribución de las dependencias verdaderas.

La transformación del flujo de instrucciones de tipo CISC a flujo de instrucciones RISC no produce una modificación sustancial del impacto de los códigos de condición sobre la disponibilidad de paralelismo ya sea a nivel de instrucción o de microoperación.

Finalmente, se propone un mecanismo, implantado en la capa del lenguaje máquina, que no afecta a la compatibilidad binaria y que habilita un modo de ejecución interno consistente en la escritura condicional del registro de estado. Este modo de ejecución evita las dependencias de datos superfluas, sin sentido computacional, mejorando la disponibilidad de paralelismo y el rendimiento superescalar. Una ventaja notable del mecanismo propuesto, sobre otras soluciones *hardware*, es que elude aumentar la complejidad de la capa física y por tanto el consumo en términos de área y potencia.

12. 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*, pages: 152 – 160, April 1989.

[2] A. Aho, R. Sethi and J. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

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

[4] A. Aho and J. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[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*, pp. 288 –297, 1997.

[6] P. Bose. *Instruction Set Design for Support of High-Level Languages*. Ph. D. Dissertation, University of Illinois at Urbana-Champaign, 1983.

[7] D. Burger, S. W. Keckler *et al*. "Scaling to the End of Silicon with EDGE Architectures," *IEEE Computer*, vol. 37, 7, pages: 44 – 55, July 2004.

[8] D. Clark and H. Levy. "Measurement and analysis of instruction set use in the VAX-11/780," in *Proceedings of the 9th Symposium on Computer Architecture*, pages: 9 – 17, April 1982.

[9] A. Cristal, J. F. Martínez, J. Ll., and Mateo Valero. A Case for Resource-conscious Out-of-order Processors. Computer Architecture Letters, IEEE Computer Society, Vol. 2, no. 2, November 2003.
También como: Technical Report No. CSL-TR-2003-1034, May 2003.

[10] R. Durán and R. Rico, "On Applying Graph Theory to ILP Analysis," *Technical Note TN-UAH-AUT-GAP-2005-01*, March 2005. Disponible en: http://atc2.aut.uah.es/~gap/

[11] R. Durán and R. Rico, "Quantification of ISA Impact on Superscalar Processing," in *Proceeding of EUROCON2005*, pages: 701 – 704, November 2005.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach. 2nd edition.* Morgan Kaufmann Publishers, 1996.

[13] 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, pages: 929 – 939, June 2002.

[14] 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.

[15] A. Lunde. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Communications of the ACM*, vol. 20(3), pages: 143 – 153, March 1977.

[16] W. D. Maurer. "A theory of computer instructions," *Journal of the ACM*, 13(2), pages: 226 – 235, April 1966.

[17] 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)*, pp. 129–140, 2003.

[18] R. Rico, "Proposal of test-bench for the x86 instruction set (16 bits subset)," *Technical Report TR-UAH-AUT-GAP-2005-21*, November 2005. Disponible en: http://atc2.aut.uah.es/~gap/

[19] R. Rico, "Analysis of x86 Data Usage (16 bits subset)," *Technical Report TR-UAH-AUT-GAP-2005-22*, November 2005. Disponible en: http://atc2.aut.uah.es/~gap/

[20] 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.

[21] M. S. Schlansker and B. R. Rau. "EPIC: Explicitly Parallel Instruction Computing," *IEEE Computer*, pages 37-45, February 2000.

[22] 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.

[23] 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*, pp. 57-66, 2000.

[24] 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.

[25] 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.

[26] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176-188, April 1991.

[27] Herramienta *software* (código fuente, ficheros de configuración y documentación): analizador de dependencias de datos ADD; Repositorio CVS (usuario anónimo): CVSROOT:pserver:anoncvs@atc2.aut.uah.es:2401/home/cvsmgr/repositorio