Tesis de Licenciatura
Reuso de Computación

Mario Daniel Bergotto
{mbergott@dc.uba.ar}
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Argentina
Directora: Dra. Patricia Borensztejn
{patricia@dc.uba.ar}

24 de julio de 2000

El significado de la finalización de este trabajo excede enormemente lo estríctamente académico, ya que marca un antes y un después en mi vida. Es por eso que además de agradecer infinitamente a todas las personas que colaboraron directamente con este trabajo, lo dedico especialmente a:

Mi familia, por el inmenso apoyo y amor que me supieron dar en todo este tiempo, y que no dudo continuará por siempre.
Malvina, por darme tanta felicidad y hacerme ver mucho mas lejos de lo que se puede utilizando la razón.
Mis amigos, que me acompañaron en todo momento.

Este trabajo no hubiera sido posible sin la colaboración e infinita paciencia de Patricia Borensztejn, a quien siempre agradeceré todo el tiempo y esfuerzo que dedicó para llevar a buen término este proyecto.

Además, quisiera expresar mi gratitud a Daniel Etiemble, quien ayudó a dar el puntapié inicial de este trabajo.

Nuevamente, gracias a todos.

Abstract

En su larga evolución, los microprocesadores han incorporado una serie de técnicas para solucionar las trabas que se encuentran en los programas que impiden aprovechar todos los recursos hardware disponibles. Los accesos a memoria principal, las dependencias de control, las dependencias falsas, son limitantes que casi todos los procesadores actuales atacan de alguna manera. Por otro lado, hasta hace pocos años, el cuello de botella que imponen las dependencias de datos verdaderas había sido dejado sin tratar. En años recientes se le ha prestado una gran atención al grupo de técnicas denominadas superespeculativas. En general, el objetivo que estas técnicas persiguen es sobreponerse a la limitación de las dependencias de datos verdaderas, colapsándolas, haciendo posible que instrucciones que deberían ejecutarse de forma secuencial lo hagan en forma paralela.

En este trabajo se presenta el concepto de reuso de computación, como una generalización del reuso de instrucciones. Como característica distintiva, se puede mencionar que el reuso de computación es útil aun en el caso de un cambio de contexto, lo cual lo hace especialmente útil en ambientes multitarea, que son la mayoría de los casos en la actualidad. El fenómeno de reuso de computación se cuantificó para las instrucciones de aritmética entera, y se encontró que la mayoría de ellas puede ser reusada. Además, en un intento por mejorar la eficiencia del esquema se desarrolló un mecanismo de filtrado basado en los valores de los operandos.

También se discute una implementación hardware para aprovechar el fenómeno de reuso de computación basado en una cache asociativa de 4 vias. Este punto no es para nada trivial, ya que ninguna de las entradas de la función de direccionamiento de la cache tiene una distribución uniforme que podría distribuir uniformememte los datos en la cache. Se desarrollaron varias funciones de direccionamiento y se compararon.

Por último, se evalúan de forma cuantitativa los beneficios del esquema de reuso de computación propuesto mediante simulaciones de un procesador superescalar que incorpora el esquema presentado en el trabajo. Se observan ganancias de performance en todos los programas simulados, y el reuso de computación presenta un ventajas sobre un esquema de reuso de instrucciones simulado. Esta ventaja puede ampliarse de manera significativa si se tiene en cuenta un ambiente multitarea.

Índice General

operación se pueden obtener esquemas que permitan mayores índices de reuso y que son útiles aun en el escenario de un cambio de contexto.

En este trabajo se ha cuantificado el fenómeno de reuso de computación mediante la realización de distintas simulaciones variando el tamaño de la cache utilizada. Estas simulaciones han arrojado como resultado que un alto porcentaje de las instrucciones dinámicas son reusables según la definición presentada.

Además, se observó que a medida que el tamaño de la cache decrece, la eficiencia, en el sentido de usar la cache para almacenar las instancias dinámicas que tienen una mayor probabilidad de ser repetidas, se transforma en un factor cada vez mas importante.

Por ello, se desarrolló un esquema de filtrado para evitar que instrucciones que tienen pocas probabilidades de ser reusadas sean insertadas en la cache, con el consiguiente desperdicio de espacio.

La idea de un mecanismo de filtrado no es nueva, ya que [29] y [27] sugieren una política de insercion 'inteligente' para el buffer de reuso, ya que solamente un 20% de las instrucciones insertadas constituyen la mayor parte del reuso. El enfoque aquí propuesto para esa política es evitar la inserción de instrucciones en el buffer de reuso, aplicando ciertas reglas sobre los valores de los operandos.

Para desarrollar el mecanismo de filtrado se realizaron pruebas para determinar cuáles son los valores mas utilizados en los programas, con el objetivo de que el filtro a desarrollar permita el paso de las instrucciones que los usen.

Pensando en un diseño útil para una implementación real, se diseñó un esquema de reuso de computación implementado con una cache de asociatividad limitada. Este desarrollo trajo implicó la experimentación de varias funciones de hashing basadas en el código de operación y los valores de los operandos. Se consiguió una función que, segun la medida aquí establecida, es aun mejor que la usada en el esquema de reuso de instrucciones contra el que se compara. De todas maneras, un mayor estudio sobre este punto debería contribuir a mejorar el índice de aciertos, para acercarse aun mas al conseguido por el esquema totalmente asociativo.

Por último, se realizaron medidas de performance sobre un procesador superescalar de cuatro vias. El resultado mas importante es que el esquema de reuso de computación utilizado mejora efectivamente la performance en todos los benchmarks simulados. Aun mas, las ganancias obtenidas son mayores que las del esquema de reuso de instrucciones con el que se compara.

Capítulo 2

Conceptos

2.1 Conceptos básicos

La referencia obligada cuando se comienza un trabajo en arquitectura de procesadores es el modelo de computación que John Von Neumann desarrolló en el año 1945 [1], sentando las bases de las actuales computadoras.

El computador posee tres elementos básicos: una unidad de procesamiento donde se realizan las distintas operaciones, una unidad de almacenamiento donde residen tanto los datos como el programa a ejecutar, y una conexión que comunica a ambos. El concepto que distingue a este modelo de lo existente hasta ese momento es la separación entre el control y el almacenamiento, ya que anteriormente el programa y la unidad de control componí una entidad indivisible. El programa, o sea, la secuencia de instrucciones que la máquina debe ejecutar, reside en la unidad de almacenamiento, y para determinar cual es la siguiente instrucción a ejecutar, en la unidad de control existe el registro PC (Program Counter), que indica la dirección de memoria de la próxima instrucción a ejecutar. Al separar el control del almacenamiento, este modelo brinda una gran flexibilidad, que no se poseía hasta el momento de su desarrollo. La ejecución de una instrucción en este modelo comprende varios pasos:

- *Fetch:* Acceder a la celda de memoria indicada por el registro PC, pasando el contenido del registro por la conexión. El dato referenciado pasa por la conexión hasta la unidad de procesamiento. Se actualiza el contenido del registro PC para que apunte a la próxima instrucción a ejecutar.

- *Lectura de operandos*: Se transfieren los operandos requeridos por la operación desde la unidad de almacenamiento hasta la unidad de procesamiento. La transferencia de cada operando comprende transferir la dirección de memoria del mismo hacia la unidad de almacenamiento por la conexión, el acceso al dato propiamente dicho y la transferencia hacia la unidad de procesamiento utilizando la conexión.

- *Ejecución:* Se realiza la operación indicada por la instrucción.

- *Escritura:* Se transmite por la conexión el resultado de la operación, y por último se lo guarda en la unidad de almacenamiento.

Analizando los computadores de los últimos 50 años, podemos observar que el concepto de programa almacenado permanece válido hasta nuestros días, y a pesar de que grandes mejoras se han producido en muchos aspectos, este concepto se ha mantenido inalterable con el paso de los años. Estas mejoras, que se describen en las próximas secciones, apuntan a mejorar el rendimiento del procesador, por lo que antes de estudiarlas en detalle, pasaremos a definir métricas para el rendimiento.

2.2 Medidas de rendimiento

¿Cómo decidir si un procesador es *mejor* que otro? Lo primero que hay que hacer es definir que es lo que uno espera del mismo.

Típicamente, lo que un usuario final desea tener es un buen *tiempo de respuesta*, o sea, el tiempo que toma la realización de la tarea. Por otro lado otro tipo de usuario, por ejemplo el administrador de un servidor de bases de datos, se interesa más en obtener una mejor *productividad*, definida como la cantidad de trabajo realizado en un tiempo determinado [3]. Para esta persona, no es tan importante el hecho de que un único trabajo finalice rápidamente, sino que lo importante es que la totalidad del tiempo insumido en completar el conjunto de trabajos sea la mínima posible.

Cuando se comparan dos procesadores, por ejemplo X e Y, es lógico querer tener una medida del rendimiento relativo de uno con respecto a otro para alguna tarea o un conjunto de ellas. Para hacer esta comparación, se mide el tiempo que toma la/s tareas tanto en X como en Y y luego se calcula:

$$R_{xy} = \frac{T_x}{T_y}$$

Esta ecuación indica cuanto mas rápido es el procesador X que el Y para la tarea medida. Ahora bien, ¿cuáles son los componentes del tiempo de ejecución de un programa? El proceso de ejecución de un determinado conjunto N de instrucciones insume una determinada cantidad de ciclos de ejecución. Por otro lado, también influye en el tiempo que tomará ejecutar ese conjunto de instrucciones el tiempo que toma cada uno de esos ciclos. Una manera muy utilizada de resumir estas tres variables (cantidad de instrucciones, cantidad de ciclos y tiempo de ciclo) es la siguiente ecuación [3]:

$$Tiempo\ de\ CPU = \frac{(N*CPI)}{Frecuencia}$$

Siendo N la cantidad de instrucciones en lenguaje máquina del programa, CPI el cociente entre la cantidad de ciclos utilizados y la cantidad de instrucciones ejecutadas y *Frecuencia* la inversa del tiempo de ciclo del reloj del procesador. Esta ecuación permite resumir de

una manera sencilla los factores que afectan el tiempo de ejecución de un programa. Por otro lado, permite aproximar el tiempo que tomará ejecutar un programa, si se dispone de la cantidad de instrucciones que se van a ejecutar, el tiempo de ciclo del procesador y una buena aproximación del CPI. Es importante notar que en la mayor parte de los casos de la vida real el CPI es altamente dependiente del código que se ejecuta, ya que pueden existir instrucciones que tomen mas ciclos que otras.

En la fórmula anterior se introduce un nuevo concepto: el de ciclo de reloj. Un microprocesador se compone de dos tipos de elementos lógicos: elementos de estado y elementos de lógica combinatoria. Los elementos de lógica combinatoria son los que realizan las operaciones, es decir, al poner un dato a la entrada el mismo se transforma, obteniendo un resultado en la salida luego de un determinado tiempo de retardo. En otras palabras, su salida depende únicamente de la entrada. La razón de ésto es que los elementos de lógica combinatoria no poseen memoria interna.

Por otro lado, los elementos de estado almacenan información, o sea, poseen una memoria interna. Los registros y memorias son ejemplos de elementos de estado. Un típico elemento de estado posee dos entradas y una salida. Las entradas corresponden al valor del dato a escribir y a la señal de reloj, que determina cuándo se almacena el dato de entrada. La salida proporciona el valor que se escribió en el ciclo de reloj anterior.

Todos los microprocesadores actuales tienen un funcionamiento *sincrónico* , es decir, gobernado por un reloj central. La función de este reloj en un microprocesador es la de proveer la señal de actualización de los elementos de estado dentro del mismo. El tiempo que transcurre entre dos ticks de reloj se llama tiempo de ciclo del reloj.

La ecuación anterior resume los tres parámetros pricipales que debe manejar un arquitecto a la hora de mejorar el rendimiento de la implementación de una arquitectura. Por un lado, el tiempo medio de ejecución de una instrucción (CPI) es altamente dependiente de las decisiones de diseño a nivel lógico que se hagan. El diseño del pipeline, la cantidad de etapas, memorias cache, predictores de salto, etc., afectan este factor. El tiempo de ciclo del procesador depende mucho de la tecnología que esté disponible al momento de la implementación real del hardware. Mejor tecnología implica compuertas lógicas mas rápidas, lo que implica un menor tiempo de propagación de las señales dentro del procesador. Al disminuir este tiempo, la frecuencia del reloj puede incrementarse, lo cual lleva a un procesador mas rápido. La cantidad de instrucciones, por otro lado, es un ámbito totalmente distinto, ya que depende exclusivamente del compilador y el programador. La influencia del arquitecto en este sentido se limita al momento en el cual se diseña el conjunto de instrucciones, ya que dependiendo de qué instrucciones decida incluir en el conjunto, la cantidad de instrucciones necesarias para realizar una determinada operación cambiará de manera acorde.

Frecuentemente, en especial cuando se habla de una arquitectura en la que se inicia la ejecución de más de una instrucción por ciclo de reloj, en lugar de hablar de CPI se suele hablar de su inversa, el IPC. Este valor, que se define como $1/CPI$, indica la cantidad de instrucciones que se ejecutan por ciclo. Actualmente, dada la proliferación de procesadores superescalares, tal es el nombre de los procesadores con dicha característica, es mucho mas

común hablar en términos de IPC que de CPI.

Utilizando la ecuación anterior se puede realizar la comparación entre dos máquinas para un determinado programa o conjunto de programas. Por otro lado, no siempre uno tiene una tarea específica en la cual comparar dos máquinas, sino que lo que se desea obtener es una idea mas general sobre la velocidad de la misma. Además, no siempre existe la posibilidad de disponer de todas las máquinas que uno quiere comparar para tomar las mediciones de tiempo deseadas.

Lo deseable para estos casos sería tener un número o índice para cada máquina que indique su potencia de cómputo. Teniendo este número, el problema anterior se solucionaría trivialmente comparando el índice de todas las máquinas a analizar. Lamentablemente, la obtención de un único número que resuma el rendimiento de una máquina no es tarea sencilla [4].

Una manera posible para obtener este índice consiste en establecer un conjunto de programas de prueba o *benchmarks* para ser ejecutados en las máquinas a analizar y luego combinar de alguna manera [5] los tiempos de ejecución obtenidos. Estos programas de prueba deben ser seleccionados de forma tal que el rendimiento que se muestra sobre estos programas se refleje mas tarde en el rendimento del sistema aplicandole la carga real. Una vez que estos conjuntos de programas se estandarizan son los mismos fabricantes los que publican los resultados obtenidos como una manera de dar a conocer el potencial de sus productos. Los beneficios de este sistema para el usuario final son grandes. Al estar estandarizados los programas que se ejecutan, el ambiente de ejecución y la forma de combinar los resultados, la tarea de comparar se vuelve mucho mas fácil.

Uno de los conjuntos de benchmarks mas difundidos es el propuesto por la System Performance Evaluation Cooperative (SPEC) [6]. De esta iniciativa, de la cual participan representantes de muchas compañías, es que surge el conjunto de benchmarks SPECCPU, que consiste en 8 programas de aritmética entera y 10 de coma flotante. Para obtener una medida del rendimiento global del sistema sobre los benchmarks los resultados obtenidos se combinan utilizando la media geométrica de las razones entre los tiempos de ejecución obtenidos y los de una máquina base. Las tablas 2.1 y 2.2 muestran los distintos programas que componen la versión 1995 de este conjunto.

La versión mencionada de este conjunto de benchmarks es la utilizada en este trabajo para medir los resultados de los esquemas propuestos. Sin embargo, al trabajar exclusivamente con simulaciones, los tiempos de ejecución no son significativos, con lo cual no pueden ser utilizados para computar el índice propuesto. La solución que se aplica en estos casos consiste en obtener el IPC del procesador simulado para cada uno de los benchmarks. Al tener el IPC y la cantidad de instrucciones ejecutadas, el tiempo total de CPU queda librado a la implementación hardware que se realice del procesador simulado. Es por esto que en las simulaciones de performance que se realizan en este trabajo hablaremos en términos de IPC para comparar rendimientos.

Tabla 2.1: Benchmarks de Aritmética entera

Benchmark	Descripción
go	An internationally ranked go-playing program.
m88ksim	A chip simulator for the Motorola 88100 microprocessor.
gcc	Based on the GNU C compiler version 2.5.3.
compress	A in-memory version of the common UNIX utility.
li	Xlisp interpreter.
ijpeg	Image compression/decompression on in-memory images.
perl	An interpreter for the Perl language.
vortex	An object oriented database.

Tabla 2.2: Benchmarks de Aritmética de coma flotante

Benchmark	Descripción
tomcatv	Vectorized mesh generation.
swim	Shallow water equations.
su2cor	Monte-Carlo method.
hydro2d	Navier Stokes equations.
mgrid	3d potential field.
applu	Partial differential equations.
turb3d	Turbulence modeling.
apsi	Weather prediction
fpppp	From Gaussian series of quantum chemistry benchmarks
wave5	Maxwell's equations

2.3 Técnicas para mejorar el rendimiento

Muchas son las mejoras que han surgido desde que se construyeron los primeros procesadores para mejorar su rendimiento. Por un lado, los adelantos tecnológicos a nivel de integración de circuitos y diseño lógico permitieron una mejor implementación hardware del diseño del arquitecto, lo que implica un ciclo de reloj mas corto.

Por otro lado, en cuanto a mejoras a nivel lógico, se pueden destacar dos: la segmentación y el paralelismo. Ninguna de las dos ideas supone un cambio conceptual en el modelo de computación utilizado, por lo que ambas ideas pueden considerarse como agregados sobre el modelo Von Neumann para intentar mejorar el rendimiento de los procesadores basados en él.

La segmentación es una técnica en la cual varias instrucciones se superponen durante su ejecución. El concepto básico es dividir el trabajo a realizar en etapas y superponer la ejecución de varios trabajos, mientras que estén en etapas distintas. El caso típico para ilustrar

la técnica de segmentación es el de una fábrica de autos. El procedimiento de fabricación de un auto consiste en cientos de etapas que tienen como producto final un auto perfectamente funcional. En cada una de las etapas se realiza un trabajo fijo y determinado sobre el auto en fabricación, y al concluir la ejecución de la misma, el auto en fabricación pasa a la etapa siguiente. Al liberarse los recursos asignados a la etapa, otro auto en fabricación puede entrar en esa fase de producción. Este modo de trabajo permite que se estén produciendo a la vez tantos autos como etapas, y que el intervalo de salida entre dos autos al final de la línea de producción sea el tiempo de la etapa mas larga. A pesar de que el tiempo de fabricación de un único auto no disminuye, y hasta aumente, la *productividad* aumenta de forma drástica.

En nuestro caso, el producto a fabricar es el resultado de la instrucción, y las etapas en las que se divide el proceso de la ejecución son, básicamente, las mismas que en el modelo Von Neumann.

La técnica de paralelismo también implica superposición en la ejecución de las instrucciones, pero en este caso tambien se multiplican los recursos para ejecutarla. En el caso de la fábrica de autos el ejemplo sería la replicación de todos los recursos necesarios para la fabricación de un auto, que incluso podría ser la línea de montaje completa, en cuyo caso estaríamos en un caso en una situación en la que se combinan la técnica de segmentación con la de paralelismo.

A continuación se analizará con mas detalle la arquitectura de un procesador segmentado y los problemas que traen la implementación de las técnicas de segmentación y paralelismo. Por simplicidad, se trabajará con un procesador de tipo RISC, cuyas únicas instrucciones de acceso a memoria son load y store. Esta característica simplifica mucho el diseño del procesador y permite una aplicación mas pura de los conceptos explicados en este trabajo. De trabajar con un procesador con instrucciones mas complejas, deberían resolverse cuestiones mas relacionadas con la adaptación de las técnicas mencionadas a las instrucciones complejas específicas del procesador que con las técnicas en sí mismas.

2.3.1 Procesador segmentado

El objetivo que se persigue al diseñar un procesador segmentado es el de completar la ejecución de una instrucción por ciclo del procesador, o sea, obtener un IPC igual a 1. Comencemos la descripción de un procesador segmentado por las distintas etapas que componen la ejecución de una instrucción y los recursos asociados con cada etapa. La división en etapas que se utiliza a continuación es simplemente un ejemplo, ya que dependiendo de distintas decisiones de diseño la cantidad de etapas y las operaciones que se realizan en cada una de ellas puede cambiar de manera significativa.

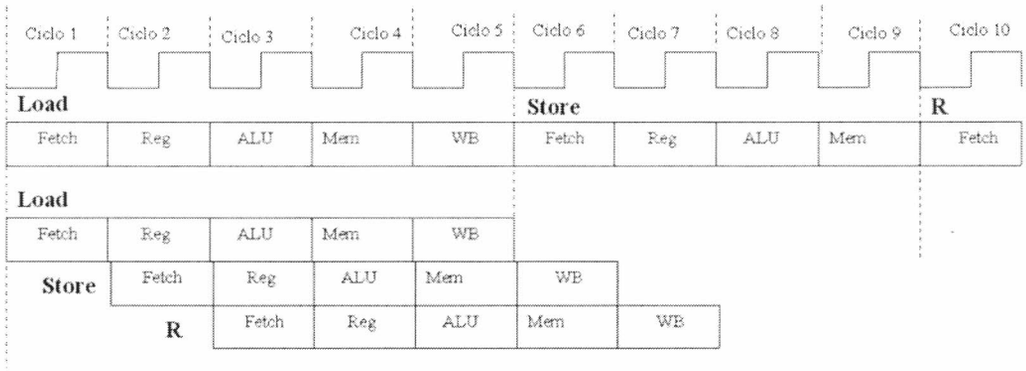
Fetch: lectura del registro PC, acceso a memoria para buscar la instrucción, almacenamiento de la instrucción en el registro de instrucción actual, incremento del PC.

Decodificación: lectura del registro de instrucción y determinación de las señales de control necesarias para la ejecución de la instrucción.

Lectura: lectura de los operandos del banco de registros.

Ejecución: ejecución de la/s operaciones. En el caso de una instrucción de aritmética

Figura 2.1: Procesador Tradicional y Procesador Segmentado [2]



entera, se utilizará la ALU entera, para una de coma flotante se utilizará una ALU de coma flotante. Para las instrucciones de acceso a memoria, en esta etapa se calcula la dirección efectiva del acceso. Para las instrucciones de salto, en esta etapa se realiza el cálculo de la condición.

Memoria: en esta etapa se realizan los accesos a memoria, tanto de lectura como de escritura. Las instrucciones de salto actualizan el registro PC. Para cualquier otro tipo de instrucción, no hay ninguna actividad asociada con esta etapa.

Escritura (Write-Back): escritura del resultado en el banco de registros.

En cada ciclo del procesador cada instrucción avanza a la próxima etapa, hasta finalizar la etapa de escritura. En la figura 2.1 se puede observar cómo sería la ejecución de un mismo código en un procesador tradicional y en uno segmentado. Se puede observar claramente como la superposición de tareas permite un importante ahorro de tiempo.

El balance del trabajo a realizar en cada etapa es crítico en el diseño del procesador, ya que el tiempo que toma la etapa mas larga determina el tiempo de ciclo del procesador. Cada arquitectura divide la ejecución de la instrucción de manera diferente , mas aun, es común que distintas implementaciones de la misma arquitectura posean una división en etapas distinta. Un buen ejemplo de esta situación es la arquitectura IA32, de la cual su primera implementación fue el microprocesador Intel 80386, y luego evolucionó hasta lo que hoy en día es el Pentium III [7]. Ambos microprocesadores son capaces de ejecutar código del primero de ellos, a pesar de que ambos procesadores son drásticamente distintos.

2.3.2 Problemas asociados con la segmentación

El ejemplo de la fábrica de autos hace pensar que la segmentación de la ejecución de instrucciones es algo bastante sencillo. Sin embargo, el solapamiento de la ejecución de varias instrucciones trae aparejados algunos problemas que deben solucionarse.

En los ejemplos de código que se exponen en el resto del trabajo se utiliza el mismo len-

guaje ensamblador estilo MIPS que en [3]. En lineas generales, el formato de una instrucción en este ensamblador es:

op dest, fuente1[, fuente2]

Las operaciones que se utilizan en el trabajo son:

Instrucción	Descripción
add $d,$r,$t	$d=$r + $t
mul $d,$r,$t	$d=$r * $t
sub $d,$r,$t	$d=$r - $t
and $d,$r,$t	$d=$r AND $t
or $d,$r,$t	$d=$r OR $t
load (o lw) $d,DIR($r)	$d=MEM[DIR+$r]
store (o sw) DIR($r),$t	MEM[DIR+$r]=$t
beq $r,$t, DEST	Si $r==$t, entonces PC=DEST
bne $r,$t, DEST	Si $r!=$t, entonces PC=DEST

Tratamiento de dependencias de control.

Al escribir una instrucción despues de otra el programador determina una dependencia de control entre ambas, ya que la primera debe ejecutarse antes que la segunda. Por otro lado, existen dependencias que se determinan en tiempo de ejecución, mediante las instrucciones de salto condicional. Al ejecutar una instrucción de este tipo el procesador debe esperar a conocer el resultado de la condición para saber cuál es la próxima instrucción a ejecutar. En un procesador segmentado esta situación representa un problema grave [8], [9], [10], porque al momento de determinarse la condición del salto, las instrucciones que seguían en secuencia al salto ya han ingresado al pipeline, a pesar de no conocerse todavía si deben ejecutarse o no.

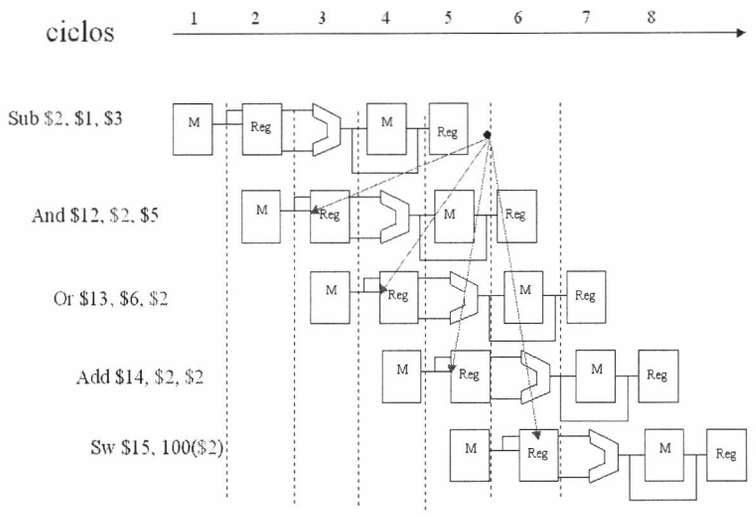
Dependencias de datos.

Consideremos el código en lenguaje ensamblador de la figura 2.2 y analicemos las dependencias de datos que se presentan.

Cada instrucción i del programa posee un conjunto de entradas $D(i)$ (dominio) y de salida $R(i)$ (rango). Se dice que existe una dependencia de datos entre las instrucciones i y j con $j > i$ en las siguientes situaciones:

$$R(i) \bigcap D(j) \neq \emptyset$$
$$\bigvee$$
$$R(i) \bigcap R(j) \neq \emptyset$$
$$\bigvee$$
$$D(i) \bigcap R(j) \neq \emptyset$$

Figura 2.2: Codigo con dependencias de datos [2]



Utilizando estas definiciones, podemos clasificar las dependencias en dos grupos: las dependencias verdaderas y las dependencias falsas. El primer grupo de dependencias está formado por el primer conjunto de dependencias (Rango-Dominio). Este tipo de dependencia se llama verdadera pues es la que se da en el caso que una instrucción necesita el valor obtenido por una anterior. El otro grupo de dependencias, las falsas, se compone de los otros dos tipos de dependencias (Rango-Rango y Dominio-Rango). Estas dependencias se producen cuando se utiliza un mismo registro o posición de memoria para almacenar el resultado de dos instrucciones distintas. La clasificación de 'falsa' proviene del hecho que de utilizar registros o posiciones de memoria diferentes la dependencia dejaría de existir y se obtendría un código semánticamente equivalente.

En la figura 2.2 se han destacado las dependencias verdaderas. Analizando la forma en que el procesador segmentado ejecutaría esta secuencia de instrucciones, se nota que el resultado que se obtendrá dista mucho de ser el que el programador pensó, ya que las instrucciones AND, OR y ADD no tienen disponible en la etapa de decodificación y lectura el resultado de la instrucción SUB.

Distinto es el caso de la instrucción SW (Store Word), pues al llegar a la etapa de decodificación y lectura la instrucción SUB ya ha escrito su resultado en el banco de registros, por lo que no existe problema alguno con respecto a la instrucción SUB.

En este breve ejemplo hemos visto que aunque todas las instrucciones posteriores a la primera tenían dependencias con ella, no todas estas dependencias eran riegosas para la correcta ejecución del programa. Diremos entonces que las dependencias entre la instrucción SUB y la AND, OR y ADD constituyen *riesgos*, mientras que la dependencia entre la instrucción SUB y la SW no lo es. Que una dependencia se transforme en un riesgo o no depende en gran parte del diseño del pipeline. Un buen ejemplo de ésto son las dependencias

dominio/rango en el diseño utilizado en esta sección. Dado que todas las instrucciones pasan por la misma cantidad de etapas, este tipo de dependencia nunca se puede convertir en un riesgo.

Dado que los riesgos pueden llegar a afectar la correctitud del resultado obtenido, el procesador debe tener un mecanismo que los detecte y realice alguna acción preventiva o correctiva.

2.3.3 Procesador superescalar

Cuando a la segmentación del camino de datos se le agrega la replicación del mismo se dice que estamos en el caso de un procesador *superescalar*.

En este tipo de arquitectura el objetivo que se persigue es ejecutar mas de una instrucción por ciclo, tantas como sea posible, aprovechando el hecho de que no todas las instrucciones tienen dependencias de datos muy cercanas. Es en este tipo de arquitecturas que, tal como se mencionaba arriba, el concepto de CPI deja su lugar al de IPC, más apto para explicar el hecho de que más de una instrucción puede terminar por ciclo.

Los problemas que se presentan en estas arquitecturas son básicamente los mismos que en un procesador segmentado, sólo que al haber un número mayor de instrucciones ejecutándose en paralelo, la magnitud de los problemas se incrementa de manera acorde.

2.4 Solución de los problemas asociados con la segmentación

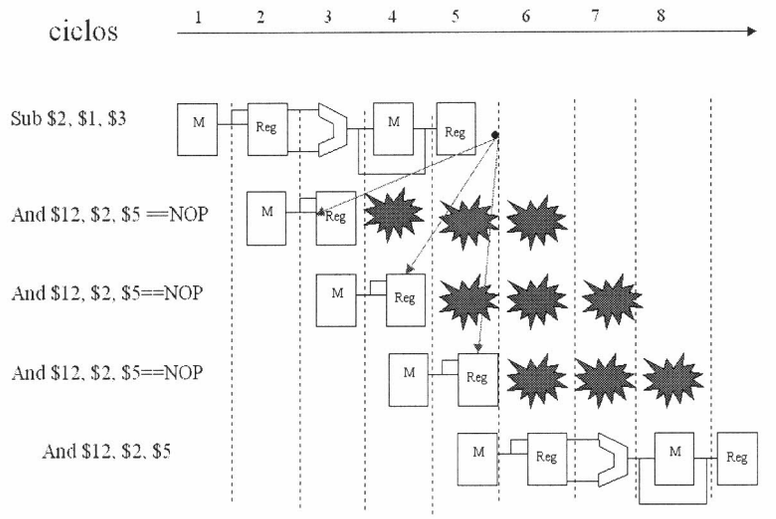
Como se mencionaba anteriormente, los problemas básicos que se encuentran al diseñar un procesador segmentado son los riesgos producidos por dependencias de datos y de control. A continuación se analizarán distintas alternativas para solucionar ambos problemas.

2.4.1 Riesgos producidos por dependencias de datos

En la figura 2.2 veíamos un trozo de código y algunas de las dependencias verdaderas existentes en él. Como mencionabamos antes, las flechas en sentido contrario al avance del tiempo indican los riesgos que se encuentran en la ejecución de ese fragmento de código.

La solución mas sencilla para estos problemas consiste en detener el avance de las instrucciones con problemas hasta que desaparezca el riesgo. Esta detención se consigue mediante la detención del mecanismo de fetch, para evitar que nuevas instrucciones ingresen al pipeline, y a la inserción de instrucciones NOP (No OPeración) en la etapa de decodificación. Esto último se realiza generando en dicha etapa las señales de control de la instrucción NOP en lugar de las que corresponden a la instrucción original, haciendo 'creer' al resto de las etapas del pipeline que la instrucción original era una NOP. La figura 2.3 muestra un ejemplo de esta solución.

Figura 2.3: Detención del pipeline [2]

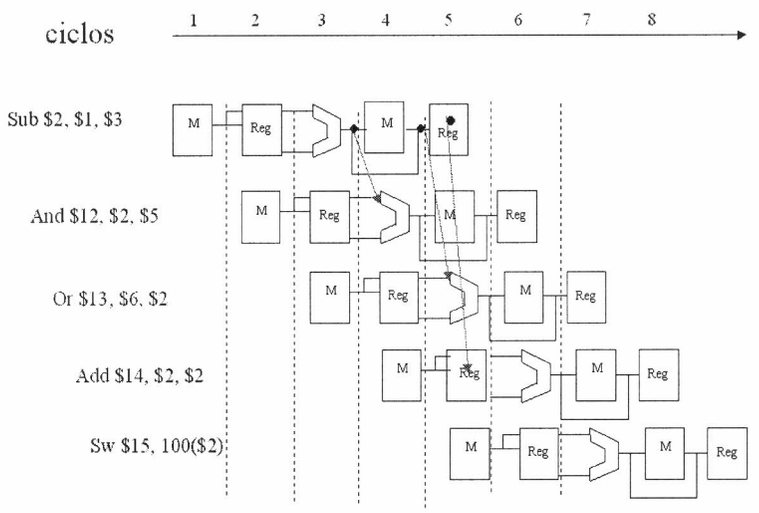


En ese ejemplo, se ve como la instrucción 2 se detiene en la etapa de decodificación hasta el ciclo 5 esperando que el operando que necesita esté disponible. Mediante esta detención se asegura que la instrucción 2 operará con los datos esperados por el programador.

El inconveniente que se presenta con esta solución es que nos aparta del objetivo que se persigue al segmentar el procesador, que es ejecutar una instrucción por ciclo, de modo tal que se necesita alguna mejora. En el ejemplo de la figura 2.3, notemos que el dato que la instrucción 2 precisa en la etapa de ejecución ya está calculado al finalizar el ciclo 3, que es cuando se realiza la operación aritmética requerida. Si existiera una manera de que ese valor pasara directamente desde el final de la etapa de ejecución al principio de la misma etapa, no sería necesaria detención alguna. Esta idea de "cortar camino" es lo que se llama un *cortocircuito*. Este nombre viene dado porque observando los nuevos caminos de datos que se agregan, éstos parecieran cortocircuitar el pipeline. Utilizando cortocircuitos de manera adecuada es posible eliminar gran parte de las detenciones por riesgos de datos. Veamos como quedaría nuestro ejemplo utilizando algunos cortocircuitos. La figura 2.4 muestra la ejecución del mismo fragmento de código, pero utilizando cortocircuitos que se destacan en el gráfico.

Lamentablemente, no todas las detenciones pueden ser evitadas utilizando cortocircuitos. El caso típico de esta situación se puede observar en la figura 2.5. En ese ejemplo, al no estar disponible el dato que se necesita al comienzo de la etapa de ejecución, no es posible utilizar un cortocircuito que evite la detención. Éste problema, el de no disponer del dato producido por un load inmediatamente anterior, es común en muchas arquitecturas. Tanto es así que existe un mecanismo específico para solucionarlo, llamado 'load retardado', que consiste en que o bien el programador o el compilador saben que cuando se escribe un load, la instrucción inmediata posterior no tendrá disponible el dato que carga el load. Esta idea se utilizó un

Figura 2.4: Utilización de cortocircuitos [2]



varias arquitecturas, por ejemplo MIPS [11], dado que no es necesario agregar ningún tipo de hardware para implementarla. Por otro lado, esta técnica tiene desventajas, ya que no siempre será posible encontrar una instrucción que no dependa del load para poner en la posición inmediata siguiente al load, y en ese caso es necesario insertar con una no-operación en esa posición, lo que es equivalente a realizar una detención del pipeline. Otra desventaja, mayor que la anterior, es que al evolucionar la implementación de la arquitectura, si se desea compatibilidad binaria será necesario mantener esta característica, que es necesaria simplemente por cuestiones de implementación. Al incorporar cuestiones de implementación dentro de la definición de la arquitectura, el espacio de diseño disponible en el futuro se reduce considerablemente.

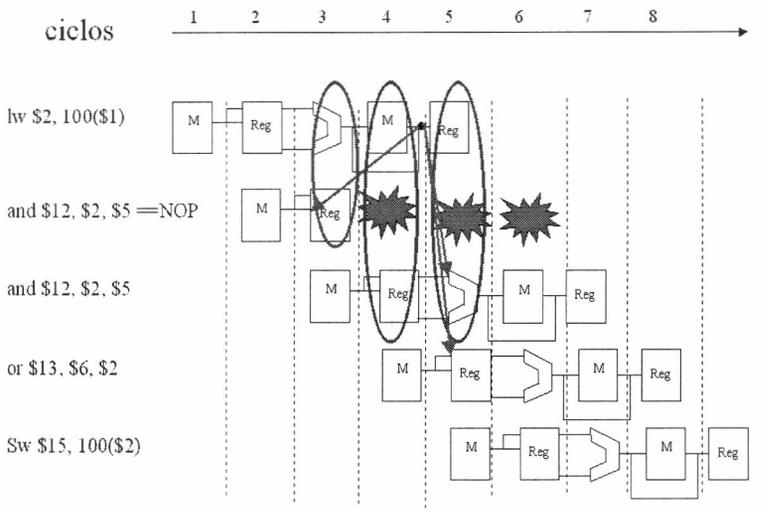
Así como para el pipeline que se describe en esta sección la latencia para tener disponible el dato que se carga en una instrucción es de un ciclo, otros diseños de pipeline podrían definir latencias mayores.

2.4.2 Riesgos producidos por dependencias de control

El otro problema que se presenta al diseñar un procesador segmentado es el de las dependencias de control. Como se mencionaba en una sección anterior, el problema radica en que recién en el momento en que se ejecuta el salto es que se conoce cual es la próxima instrucción a ejecutar, pero en ese punto las instrucciones que seguían en secuencia al salto ya han ingresado al pipeline. La figura 2.6 es un ejemplo de esta situación.

Al igual que en el caso del load retardado, se puede plantear un salto retardado, que consiste simplemente en que el programador o el compilador conoce que una cantidad de instrucciones después del salto serán ejecutadas sin importar si el salto es tomado o no. El

Figura 2.5: Riesgos producidos por instrucción load [2]



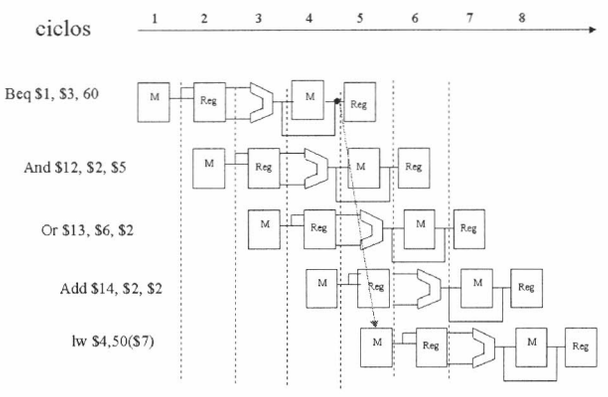
problema se traslada ahora al compilador o al programador, que debe encontrar instrucciones que no dependan del resultado del salto para insertar inmediatamente después.

Esta solución, aunque permite en teoría cumplir con el objetivo de ejecutar una instrucción por ciclo, está lejos de ser ideal, ya que por un lado, no siempre existen esas instrucciones libres que no dependen del resultado del salto, y por otro lado, la cantidad de instrucciones que se necesita tener disponibles es dependiente del diseño del pipeline, lo cual limita futuras evoluciones de la arquitectura.

Una solución que simplifica el trabajo del compilador es detener el ingreso de nuevas instrucciones al pipeline hasta conocer el resultado del salto. De esta manera, el problema anterior desaparece, a costa de la pérdida de rendimiento que supone la detención del procesador. Una mejora sobre este esquema consiste en suponer que el salto no será tomado y permitir que las instrucciones subsiguientes ingresen al pipeline, y en caso de que el salto sea tomado, anular las instrucciones que no debían ejecutarse. La anulación de estas instrucciones es bastante mas sencillo de lo que parece ser a simple vista, ya que con inhibir la escritura del resultado de la ejecución de la instrucción en el banco de registros o la memoria se consigue el efecto deseado.

Esta técnica, que llamaremos 'suponer no tomado' (not taken), se implementó en varios procesadores, como por ejemplo el Intel 486 [12], dado que con un costo realmente bajo en hardware, permite al compilador olvidarse del problema de las dependencias de control, con una pérdida de performance mas razonable que la alternativa de detención. La alternativa de 'suponer no tomado' tiene una particularidad que es importante notar: hace que el CPI de las instrucciones de salto condicional dependa de si el salto es tomado o no tomado. Para nuestro modelo de pipilene, en caso de un salto no tomado, el CPI es 1, pero si es no tomado es 4. Este hecho cambia totalmente la forma de predecir el tiempo de ejecución de

Figura 2.6: Dependencias de control en un pipeline [2]



Beq $1,$3,60
And $12,$2,$5
Or $13,$6,$2
Add $14,$2,$2
lw $4,50($7)
.
.

un programa, ya que el mismo dependerá de la cantidad de saltos tomados y no tomados, lo cual sólo se sabe en tiempo de ejecución.

Así como esta política favorece los saltos no tomados, se puede pensar en un mecanismo para favorecer a los saltos tomados. Esta política, sin embargo, no es tan fácil de implementar como la anterior, ya que en primer lugar, la dirección destino del salto no siempre es conocida al ciclo siguiente de hacer el fetch del salto, y por otro lado, la recuperación cuando se encuentra un salto no tomado es sensiblemente mas dificultosa. Esto último es porque las instrucciones de salto calculan la dirección a ejecutar en el caso que el mismo sea tomado, pero no la dirección en el sentido no-tomado, con lo que debe ser el control del procesador el que recuerde la dirección que sigue a la instrucción de salto previendo que el mismo no sea tomado.

Los dos mecanismos anteriormente mencionados son conceptualmente importantes porque son los primeros que favorecen algunas instancias de instrucciones de salto sobre otras. Dicho de otra manera, estas técnicas mejoran el rendimiento *especulando* sobre el resultado de las instrucciones de salto condicional. Técnicas muchísimo mas complejas que estas se han estudiado e implementado y en las próximas secciones se explicarán las ideas más importantes en este campo.

2.5 Técnicas para mejorar el rendimiento

Las técnicas explicadas en la sección anterior solucionan los problemas asociados con la aplicación de la segmentación y el paralelismo que afectan la correctitud del resultado obtenido. Como se ha visto, estas soluciones traen aparejada una pérdida de performance que nos aleja de los objetivos planteados al incorporar dichas técnicas. A continuación se explicarán algunas de las técnicas más importantes para mejorar el rendimiento de un procesador

1: $R5 \leftarrow 3$
2: $R10 \leftarrow ...$
3: $R10 \leftarrow R10...$
4: $R3 \leftarrow R5...$
5: $R3 \leftarrow mem[R3 + R10]$
6: $R9 \leftarrow R9...$
7: $R7 \leftarrow R9...$
8: $R4 \leftarrow R3 - R7$
9: bne $R4, R0, 1$
10: $R9 \leftarrow R9...$
11: $R5 \leftarrow R5$

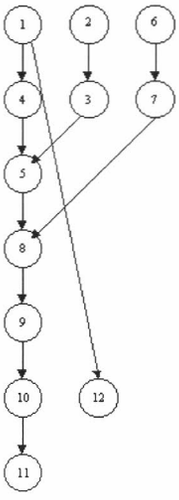


Figura 2.7: Algoritmo y plan de ejecución asociado [2]

segmentado o superescalar.

2.5.1 Planificación dinámica

En un procesador segmentado tradicional, las instrucciones entran al pipeline en el orden especificado por el programador. Una vez realizado el fetch, la instrucción no pasa a etapas posteriores del pipeline hasta que todas las dependencias de datos hayan sido satisfechas. Este enfoque, que hace recaer en el compilador el problema de la planificación de las instrucciones para minimizar los ciclos de detención, está pensado principalmente para mantener la correctitud semántica del código ejecutado. Otra posibilidad, mucho mas interesante desde varios puntos de vista, es que el plan de ejecución de las instrucciones lo realice directamente el procesador de forma dinámica en tiempo de ejecución. Este último enfoque presenta varias ventajas. Por un lado, simplifica el compilador, ya que el mismo puede desentenderse de casos en los cuales las dependencias de datos no se conocen en tiempo de compilación. Por ejemplo, se puede mencionar el caso de instrucciones tipo load dentro de ciclos, cuya dirección efectiva se calcula en cada iteración del ciclo. Por otro lado, permite que un código que se compila para un diseño de pipeline determinado funcione eficientemente en otro distinto.

Analicemos el fragmento de código de la figura 2.7:

El grafo de la figura 2.7 resume las dependencias de datos existentes en el programa de la misma figura. Por ejemplo, podemos observar un arco dirigido desde el nodo 2 hasta el 3, dado que la instrucción 3 requiere el valor producido por la instrucción 2 en R10.

Un procesador segmentado escalar ejecutaría las instrucciones según el siguiente plan:

1,2,3,4,5,6,7,8,9,...

Si suponemos una latencia de un ciclo para poder utilizar el resultado de una instrucción,

y guiandonos por el grafo de dependencias, este plan de ejecución causaría detenciones entre las instrucciones 2 y 3, 4 y 5, 6 y 7, 7 y 8, y 8 y 9.

En cambio, un procesador con planificación dinámica podría realizar el siguiente plan:

1,2,6,3,7,4,5,8,9,...

En este caso sólo se generarían detenciones entre las instrucciones 4 y 5, y 8 y 9. Aunque a primera vista el plan de ejecución que se obtiene con la planificación dinámica es superior al tradicional desde el punto de vista de cantidad de detenciones del pipeline, no todas son ventajas.

La ejecución *en orden* tiene una ventaja muy importante, y es la simplicidad de la implementación. Dado el diseño del pipeline, una gran cantidad de dependencias de datos, por ejemplo las rango-rango, se resuelven (no se transforman en riesgos) por la construcción misma del pipeline, lo cual reduce de manera significativa la complejidad de la operación de verificar si una instrucción puede avanzar en el pipeline o no. Al desordenarse la ejecución de las instrucciones, esta ventaja desaparece por completo, y el cálculo de dependencias se vuelve extremadamente complejo, especialmente cuando en lugar de tener un procesador segmentado se tiene un procesador superescalar que intenta ejecutar más de una instrucción por ciclo.

Por otro lado, el tratamiento de excepciones se vuelve mucho mas complejo, porque el retiro de una instrucción del pipeline no implica que todas las anteriores del programa hayan sido retiradas. Esta situación puede dejar al procesador en un estado inconsistente, ya que al producirse una excepción, el estado de la máquina puede no ser consistente con el del programa según la última instrucción ejecutada sin excepciones.

Es por esta situación que al implementarse un esquema de planificación dinámica también debe implementarse un método de tratamiento de excepciones que permita mantener en todo momento la consistencia entre el estado del procesador y el del programa según la última instrucción que se haya retirado del pipeline. Con este objetivo es que en general las etapas del pipeline de un procesador superescalar suelen ser las siguientes:

- Fetch: obtención de las instrucciones. Actualización del PC.
- Decodificación: cálculo de las señales de control para la instrucción.
- Inicio: Renombre de registros, predicción de saltos y verificación de espacio en la etapa de Despacho para enviar la instrucción a dicha etapa.
- Despacho: inicio de la ejecución de aquellas instruciones libre de dependencias de datos verdaderas.
- Ejecución: ejecución de la operación propiamente dicha.
- Write-Back: escritura del resultado en una estructura de datos auxiliar.

- Commit: retiro en orden de la instrucción. Escritura del resultado definitivo en los registros de la arquitectura.

La estructura auxiliar que se menciona en la etapa de Write-Back permite almacenar los resultados de las instrucciones que se ejecutan fuera de orden, para que luega la etapa de Commit retire de la misma los resultados de las instrucciones segun el orden especificado por el programador.

2.5.2 Renombre de registros

Un problema que se suele dar a menudo en un procesador con planificación dinámica es el de la figura 2.8.

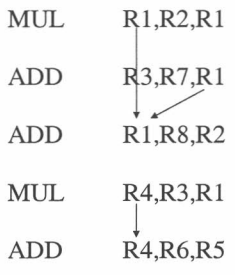


Figura 2.8: Ejemplo de dependencias falsas

En este caso, como el programador está reutilizando los registros R1 y R4, ya sea porque no conoce el funcionamiento interno del procesador o porque no tiene ningún otro registro para utilizar, el procesador no puede explotar el paralelismo existente en el algoritmo. En ese ejemplo, el procesador no puede iniciar la ejecución en paralelo de las instrucciones 2 y 3, ya que la instrucción 3 podría escribir el registro R1 antes de que la 2 lea su contenido. Una situación similar ocurre entre las instrucciones 4 y 5, ya que si el programador hubiera elegido un registro distinto de R4 como destino en la última, ambas se podrían haber ejecutado en paralelo.

Este problema, el de las dependencias falsas, se vuelve mas grave cuando la cantidad de registros de la arquitectura es pequeña. Una solución posible para ésto es definir una cantidad de registros mayor en la arquitectura, o sea, incrementar el número de registros visibles al programador. Al haber una mayor cantidad de registros para utilizar, se minimiza la cantidad de reusos, y con ellos las detenciones por riesgos causados por antidependencias. Esta solución, aunque sencilla, tiene la desventaja de modificar la arquitectura, lo cual no es para nada deseable.

Otra alternativa, que ha sido implementada en muchos procesadores, es implementar la técnica de renombre de registros [19]. Esta técnica, que a continuación explicaremos con un mayor nivel de detalle, consiste en definir una mayor cantidad de registros físicos que los que son visibles al programador (lógicos), y hacer que sea el procesador el que maneje el mapeo entre registros físicos y lógicos.

La técnica de renombre de registros se utiliza para determinar las dependencias de datos entre las instrucciones y proveer un manejo de excepciones preciso. Cuando un registro es *renombrado*, los registros lógicos referenciados por una instrucción se mapean en registros físicos utilizando una tabla de mapeo. Un registro lógico se mapea en un nuevo registro físico cada vez que es el registro destino de una instrucción. Por lo tanto, cuando una instrucción almacena un nuevo valor en un registro lógico, ese registro lógico es renombrado para utilizar un nuevo registro físico. Sin embargo, el valor anterior permanece en el viejo registro físico, lo cual permite recuperarlo en caso de que la instrucción sea abortada por una excepción o un salto incorrectamente predicho.

Durante la ejecución de las instrucciones se genera una cantidad de resultados temporarios. Estos valores temporarios son almacenados en el banco de registros junto con valores permanentes. Los valores temporarios se transforman en nuevos valores permanentes cuando la instrucción correspondiente se *gradúa*. Diremos que una instrucción se gradúa cuando todas las instrucciones anteriores en el orden especificado por el programa se han completado exitosamente.

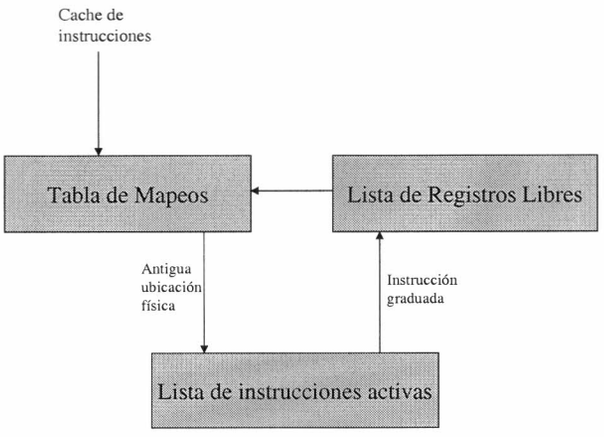
El chequeo de dependencias se realiza mientras cada instrucción es renombrada. En ese momento los nombres de sus registros lógicos se comparan para determinar las dependencias entre todas las instrucciones que se decodifican en el mismo ciclo.

Las estructuras de datos utilizadas para implementar el renombre de registros son una tabla de mapeo, una lista de instrucciones activas y una lista de registros libres.

Supongamos un procesador con P registros físicos. En todo momento el valor de cada registro físico se encuentra en alguna de estas listas. Cuando se hace el fetch de una instrucción se la coloca en la tabla de mapeos. La lista de instrucciones activas mantiene un listado de todas las instrucciones presentes en el pipeline en cada momento. Esta lista se mantiene siempre en orden. Las instrucciones en las colas pueden ser ejecutadas fuera de orden, pero antes de que el resultado pueda ser almacenado finalmente en el banco de registros, debe ser almacenado en orden segun determina la lista de instrucciones activas. Una vez que el valor se almacena de forma definitiva, se transforma en obsoleto y por lo tanto la instrucción que lo generó deja de ser activa. En este momento se dice que la instrucción se ha *graduado*. El registro físico puede entonces ser retornado a la lista de libres. La figura 2.9 ilustra los distintas etapas por las que pasa una instrucción. Cada instrucción puede ser identificada unívocamente por su ubicación dentro de la lista de activos. Un valor de unos pocos bits llamado el *tag* de la instrucción acompaña a cada instrucción durante su ejecución y permite que sea fácilmente ubicada dentro de la lista de instrucciones activas para ser marcada como 'finalizada' cuando la instrucción se gradúe.

Cuando un valor se saca de la lista de libres se pasa a la tabla de mapeo y ésta se actualiza. El valor del registro en particular ahora contiene el valor actual de un operando. El viejo

Figura 2.9: Diagrama de bloques del esquema de renombre de registros



valor de la tabla de mapeo se ubica entonces en la lista de activos. El valor permanece en la lista de activos hasta que la instrucción se gradúa, indicando que ha finalizado en el orden especificado por el programa. Una instrucción sólo se puede graduar después de que ella misma y todas las instrucciones anteriores hayan finalizado exitosamente. Una vez que una instrucción se ha graduado, todos los valores anteriores se pierden.

2.5.3 Predicción de saltos

Es conocido que aproximadamente entre un 15% y un 20% de las instrucciones ejecutadas son instrucciones de salto, con lo cual si tomamos una secuencia de 5 instrucciones, es altamente probable que una de ellas sea un salto. En un procesador segmentado, esto significa que la mayor parte del tiempo habrá una instrucción de salto en el pipeline. Para cuantificar la pérdida de performance que esto implica, supongamos un CPI promedio de 2 para un salto y una proporción del 15% de saltos. Entonces :

$$CPI_{Promedio} = 85\% * 1 + 15\% * 2 = 1.15$$

Lo que significa una pérdida de 15% del rendimiento simplemente debido a los saltos. A pesar de que esta situación no parece tan mala, si en lugar de un procesador segmentado clásico hablamos de un procesador superescalar que inicia la ejecución de 4 instrucciones por ciclo, los cálculos son mucho mas pesimistas, ya que es muy probable que en cada grupo de cuatro instrucciones que ingresen al procesador se encuentre una instrucción de salto.

Figura 2.10: Ejemplos de saltos predecibles

```
1: i ← 0
2: while i ≤ 1000 do
3:    arrayAVG[i]=(array1[i]+array2[i])/2
4:    i ← i + 1
5: end while
```

```
1: if Vacio(ConjuntoA) then
2:    ImprimirResultadoFinal()
3: else
4:    SeguirProcesando()
5: end if
```

En este caso, las pérdidas por un salto mal predicho son muchísimo mayores (cantidad de instrucciones iniciadas por ciclo multiplicadas por la latencia de ejecución del salto).

Las técnicas expuestas anteriormente para resolver la problemática de las dependencias de control no son lo suficientemente efectivas como para resolver de manera eficiente un problema de tal magnitud. Es por eso que se han desarrollado infinidad de mecanismos [13], [14], [15], [16] para reducir la influencia negativa de las dependencias de control lo más posible.

La última idea que se había expuesto en secciones anteriores consistía en suponer, especular, con que la dirección del salto sería o bien tomado o bien no tomado. Sin embargo, al beneficiar solamente un tipo de saltos, la utilidad de estas técnicas es mas bien reducida. Las alternativas que se explicarán a continuación se basan en la idea de especular dinámicamente sobre la dirección del salto.

Para poder especular hay que tener información que nos permita tomar una decisión. En el caso de los saltos, esta información consiste en el resultado de la ejecución de las instrucciones de salto anteriores. Por ejemplo, supongamos que para cada instrucción de salto que se ejecuta, se almacena su dirección y la dirección que toma; cuando se realiza el fetch de una instrucción se compara la dirección que se accede contra la información almacenada, y en caso de encontrarla se decide cual será la próxima dirección de fetch según el resultado almacenado de la última ejecución del salto. Este mecanismo [14], fácilmente implementable en hardware, es uno de los primeros predictores de saltos que fueron implementados en procesadores comerciales, como el AMD K5 [18] y el MIPS R8000 [17]. El funcionamiento del mismo está basado en el hecho de que una gran parte de los saltos tiene un comportamiento bastante determinado hacia un sentido, con lo cual al almacenar el resultado de la última ejecución se tiene cierta seguridad de que en próximas ejecuciones se repetirá el resultado. Es fácil imaginar el porqué de esta última afirmación si se piensa en los ciclos, o construcciones IF simples como los que figuran en la figura 2.10. En el ejemplo del ciclo, es bastante sensato apostar que la mayoría de las veces el salto que determina si se entra o no al ciclo, hará que se entre en el mismo. En el ejemplo del IF, también será bastante seguro apostar a que el conjunto sobre el que se está operando no estará vacío la mayor parte de las veces. Estos son dos casos típicos, aunque si uno observa con detenimiento muchos fragmentos de código es posible encontrar muchos casos más de saltos con un comportamiento muy marcado en un sentido determinado.

Generalizando el esquema anterior, en lugar de utilizar un solo bit que indica el resultado de la última ejecución del salto, se puede plantear el uso de más bits, que interpretados como números en base 2 representan los distintos estados de un autómata finito determinístico.

Cada estado poseerá entonces dos transiciones, segun el resultado de la ejecución, y además cada estado tendrá asociada una predicción. En la figura 2.11 se puede apreciar un ejemplo utilizando dos bits. En este ejemplo los estados '00' y '01' tienen asociada la predicción 'no tomado', mientras que los estados '10' y '11' estan asociados a la predicción 'tomado'. Una técnica particular que se implementó en muchos procesadores, como el Pentium [7] y el UltraSparc [20] es la de la figura 2.12, llamada bimodal [14].

Figura 2.11: Predictor de cuatro estados

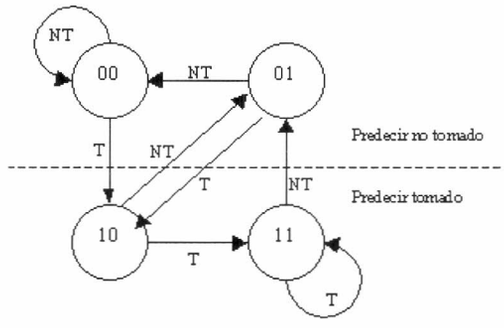
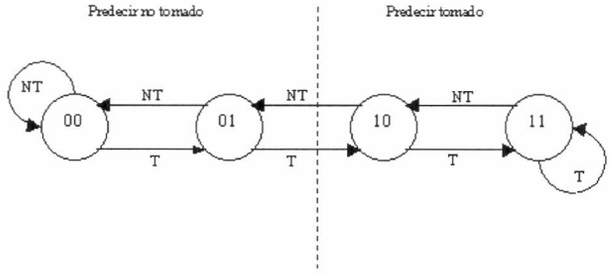


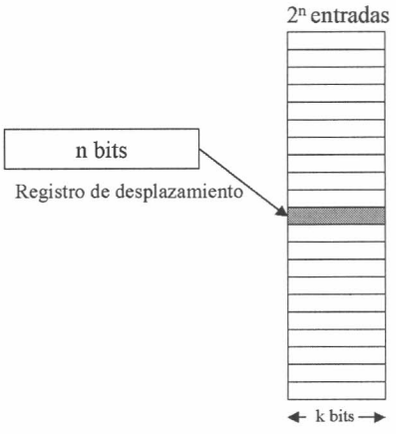
Figura 2.12: Predictor bimodal



Al utilizar mas bits para predecir el salto es lógico pensar que la precisión de la predicción debe aumentar, aunque esto no siempre es así, ya que aunque muchos saltos presentan un comportamiento bastante marcado, muchos otros tienen un comportamiento mas errático, aunque también predecible. Analicemos el siguiente ejemplo:

```
1: for i = 1 to k do
2:     if i mod k = 0 then
3:         st1;
4:     else
5:         st2;
```

Figura 2.13: Esquema de un predictor global



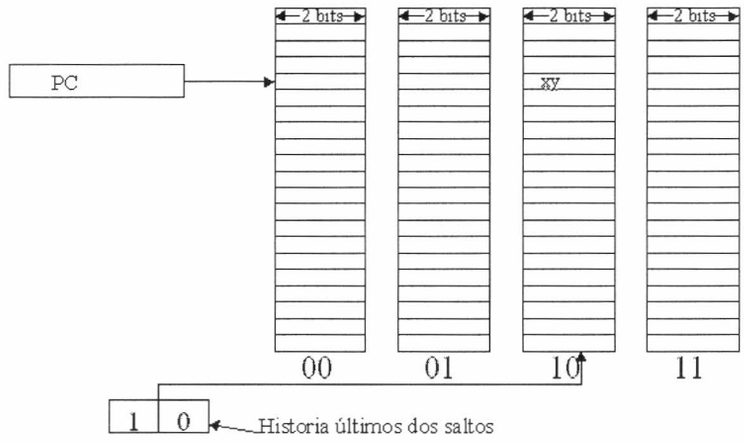
6: **end if**
7: **end for**

En este ejemplo, el salto de la instrucción IF tiene un comportamiento totalmente predecible, pero a la vez sin ninguna tendencia hacia una rama o la otra. En estos casos, utilizar únicamente información sobre la tendencia del salto en un sentido u otro no es suficiente. Lo que se necesita es almacenar una historia sobre los últimos acontecimientos de forma tal de aprender de ellos para realizar una mejor predicción. Por ejemplo, se podría memorizar que si en las últimas n ejecuciones del salto se obtuvo determinado patrón, entonces en la próxima ejecución se predecirá determinado camino.

Existen dos técnicas basadas en esta idea: la técnica global y la técnica local. La primera de ellas consiste en almacenar la historia de los últimos n saltos en un registro de desplazamiento de n bits. El contenido de este registro se utiliza para indexar una tabla, que contiene contadores de k bits, que dan la predicción. De una manera intuitiva, lo que se hace es asociar a cada patrón de ejecución una predicción. En la figura 2.13 se puede apreciar un esquema de un predictor global.

Por otro lado, la técnica local almacena información combinada. Por un lado, al igual que en la técnica global, se almacena la historia de los últimos n saltos en un registro de desplazamiento. Por otro lado, existen 2^n tablas con contadores del tipo del predictor bimodal indexadas por la dirección de la operación de salto. En este caso, el registro de desplazamiento se utiliza para determinar cual de las 2^n tablas posibles para una dirección determinada se utilizará para realizar la predicción. Intuitivamente, lo que se hace es guardar la tendencia de cada salto, pero asociada también con los eventos anteriores. Esta combinación de información global y local es muy poderosa, y ya existen procesadores que la incorporan, como el Intel Pentium Pro [21].

Figura 2.14: Esquema de predictor local



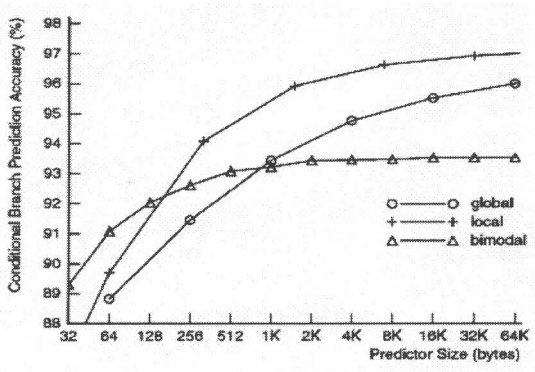
La figura 2.15 resume los resultados que se obtienen para los predictores comentados, segun el tamaño de la estructura de datos utilizada. Como se puede ver, en general a medida que aumenta la cantidad de información almacenada, mejor es la predicción. Sin embargo, esto no es una regla general, ya que, por ejemplo, el predictor bimodal *satura* cerca del 94% de aciertos. A partir de este punto, aunque se duplique la cantidad de almacenamiento utilizada no se consiguen mejoras. La situación para los otros predictores no es muy distinta, sólo que el punto de saturación se encuentra bastante mas alejado.

2.6 El próximo paso: Predicción de valores y Reuso de instrucciones

Como hemos visto hasta ahora, los procesadores superescalares actuales emplean un abanico importante de técnicas especulativas para mejorar su rendimiento. El ejemplo mas importante son los predictores de saltos, a los cuales se ha dedicado mucho estudio desde hace ya varios años. Sin embargo, este no es el único ejemplo de aplicación de técnicas especulativas. Desde hace ya muchos años, las memorias caches, tanto de datos como de instrucciones, se han incorporado en todo tipo de procesadores. Este es otro ejemplo típico de especulación, ya que las memorias caches no almacenan solamente el dato puntual accedido, sino que almacenan además datos vecinos, suponiendo, especulando, con que los datos mas próximos serán accedidos proximamente.

A lo largo de este trabajo se han explicado varios problemas y se han mostrado soluciones bastante eficientes para todos ellos. Por otro lado, nunca se ha intentado atacar el problema de las dependencias verdaderas [9], ya que las mismas constituyen la esencia mis-

Figura 2.15: Indice de predicción [15]

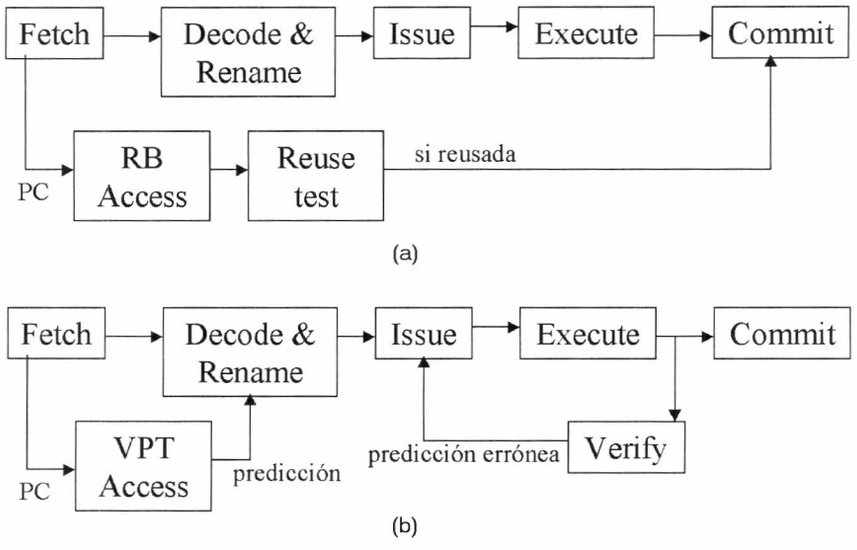


ma del programa y de respetarlas depende la correctitud de los resultados. Como veremos en las próximas secciones, las técnicas de predicción de valores [22], [23], [24], [25] y reuso de instrucciones [27] tratan directamente con las dependencias verdaderas, colapsándolas y haciendo que instrucciones que tendrían que ejecutarse secuencialmente lo puedan hacer en paralelo. A diferencia de las técnicas anteriormente explicadas, no existen procesadores comerciales en la actualidad que hayan incorporado una de estas técnicas, ya que las mismas se encuentran todavía en plena etapa de investigación. La predicción de valores es una técnica especulativa que consiste en predecir los valores de los operandos que no están disponibles en el momento en que se los necesita. En una etapa posterior, cuando los operandos están disponibles, esta predicción se verifica contra los datos reales. En caso de haber tenido éxito con la predicción, la instrucción puede retirarse, y en caso de fallo debe reejecutarse con los operandos correctos. La ventaja de esta técnica consiste en que al no tener que esperar a sus operandos, la instrucción puede ejecutarse en paralelo con otras instrucciones que normalmente deberían finalizar antes.

La técnica de reuso de instrucciones es una técnica no especulativa que evita la ejecución de instrucciones que van a generar el mismo resultado que en una ejecución anterior. Al evitar cálculos innecesarios el camino crítico se acorta, reduciendo el tiempo de ejecución. Como hemos dicho, esta técnica es "no especulativa", pero esto debe entenderse en el sentido de que es una técnica segura, ya que en el caso de intentar reusar una instrucción y no lograrlo, no hay ninguna penalización asociada, al contrario de la técnica de predicción de valores.

La figura 2.16 muestra dos pipelines, ejemplificando una posible implementación de un pipeline con predicción de valores y reuso de instrucciones en cada caso. La diferencia mas importante entre ambos esquemas reside en el camino hacia atrás que se encuentra en el pipeline que incorpora la predicción de valores, indicando la posibilidad de reejecución de una instrucción en el caso que la predicción haya sido erronea. En la figura 2.17 se puede observar como es la ejecución de tres instrucciones en un procesador superescalar tradicional, en uno que incorpora predicción de valores (VP) y en uno con reuso de instrucciones (IR).

Figura 2.16: Pipeline incorporando (a) Reuso de instrucciones (b) Predicción de Valores [26]



Las instrucciones I, J y K de este ejemplo forman una cadena de dependencias de datos, lo cual hace que en el procesador superscalar base sea posible hacer el fetch y la decodificación de las tres en paralelo, pero que a partir de la etapa de ejecución sea necesario que avancen en forma secuencial. En el pipeline con predicción de valores la cadena de dependencias se rompe prediciendo los valores de los resultados de I y J, lo cual hace que las tres instrucciones puedan avanzar en paralelo. En el último caso, los resultados anteriores de las instrucciones son reusados en paralelo con la decodificación de las mismas.

2.7 Estado del arte en predicción de valores y reuso de instrucciones

Como se explicaba anteriormente el tiempo de ejecución de un programa (T) se compone de tres factores:

- N: el número de instrucciones.
- CPI: Cantidad de ciclos de reloj por instrucción.
- t_{ciclo}: duración del ciclo de reloj.

Todas las técnicas implementadas hasta el momento para mejorar el rendimiento (o sea, minimizar T), atacan la ecuación intentando disminuir CPI o t_{ciclo}.

Figura 2.17: Comparación entre camino normal, con predicción de valores y con reuso de instrucciones [

	Superescalar base						Con VP				Con IR		
Pipeline	1	2	3	4	5	6	1	2	3	4	1	2	3
Fetch	I,J,K						I,J,K				I,J,K		
Dec&Ren		I,J,K						I,J,K				I,J,K	
Execute			I	J	K				I,J,K				
Commit				I	J	K				I,J,K			I,J,K

Complementando las técnicas actuales que minimizan el CPI o el tiempo de ciclo, el reuso de instrucciones apunta a minimizar T reduciendo la cantidad de instrucciones ejecutadas efectivamente por el procesador. El principio es el mismo que una cache de datos, que reduce la cantidad efectiva de accesos a memoria principal almacenando en un pequeño repositorio los datos mas requeridos.

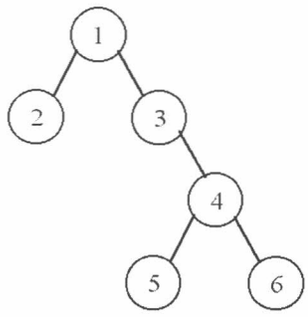
La idea de reusar operaciones tiene ya varios años. Sin embargo, sólo recientemente se le ha dado mayor atención a esta idea, existiendo numerosos trabajos publicados sobre el tema [28], [29], [27].

El concepto básico sobre el cual se fundamentan todos los trabajos es la localidad de valores [30]. Así como una cache de datos se basa en el principio de localidad temporal y espacial, suponiendo que si un dato se accede en un determinado instante es altamente probable que el mismo dato o uno muy cercano sea accedido proximamente, algo similar ocurre con el resultado de las operaciones, en el sentido de que si una instancia de una instrucción genera determinado valor, es probable que la próxima vez que se ejecute se obtenga el mismo valor.

Ampliando el principio de localidad es que surge la idea de predictibilidad de valores [31], [32] . De la misma manera en que existen instrucciones que obtienen una y otra vez el mismo resultado, existen muchas otras que no lo hacen, pero que obtienen resultados 'predecibles'. Un ejemplo típico es la variable de control en un ciclo: la instrucción que la actualiza siempre obtiene un valor distinto, pero la secuencia de valores que genera es extremadamente sencilla de calcular, por ejemplo de cuatro en cuatro, o de uno en uno.

Basándonos en estos dos conceptos es que surgen básicamente dos técnicas para mejorar el rendimiento de un procesador: la mencionada de reuso de instrucciones y la predicción de valores.

Un trabajo básico en el tema de cuantificación de la redundancia presente en los programas es [29]. Este trabajo consiste básicamente en un estudio tanto cuantitativo como cualitativo de la redundancia en los programas y sus causas.




```
while NodoAccedido.Indice != IndiceBuscado do
  if NodoAccedido.Indice¡IndiceBuscado then
    if NodoAccedido.HijoDerecho != NULO then
      NodoAccedido ←NodoAccedido.HijoDerecho
    else
      Return(No Encontrado)
    end if
  else
    if NodoAccedido.HijoIzquiero != NULO then
      NodoAccedido ←NodoAccedido.HijoIzquierdo
    else
      Return(No Encontrado)
    end if
  end if
end while
Return(NodoAccedido)
```

Figura 2.18: Arbol binario y algoritmo de búsqueda asociado

El fenómeno estudiado en ese trabajo se utilizó en un principio para desarrollar técnicas para mejorar la performance de las instrucciones de acceso a memoria, dado que son un componente importante del camino crítico en la ejecución del programa y que en gran parte operan sobre datos repetidos. Esto último es así pues muchas veces se utiliza la memoria para almacenar constantes que son accedidas repetidamente, o cuando se recorren estructuras complejas en memoria, ya que la mayoría de las veces no cambia la estructura utilizada para acceder a los datos, sino solamente los datos. Como ejemplo de esta situación, se puede citar un árbol binario donde los datos se almacenan solamente en las hojas. El proceso de acceder a una determinada hoja contiene varios accesos a memoria, pero cada instrucción, excepto la última en el caso en que se vaya actualizando el dato almacenado en la hoja en cada acceso, devuelve exactamente el mismo resultado.

El grafo de la figura 2.18 junto con el algoritmo de búsqueda asociado es un ejemplo concreto de la situación planteada en el párrafo anterior. Si se accede repetidamente al nodo 5, los accesos a memoria para obtener la dirección de los nodos 1,3,4 y 5, siempre darán el mismo resultado. Incluso, si luego de acceder al nodo 5 intentamos acceder al nodo 6, nuevamente tendremos operaciones "repetidas", ya que recorreremos nuevamente el camino 1,3,4 para acceder a dicho nodo.

Los autores de [29] definen en ese trabajo que la repetición de una instrucción ocurre cuando dos instancias dinámicas de la misma instrucción estática generan el mismo resultado. Esto sucede si (y no sólo si) ambas instancias dinámicas operan sobre los mismos inputs. Esto también puede suceder si los inputs no son los mismos, ya que en el caso de instrucciones que devuelven un valor booleano, es típico que muchas combinaciones de operandos devuelvan el mismo resultado. Por otro lado, también puede suceder que a pesar de tener los mismos operandos, la operación obtenga distintos resultados. Este comportamiento es típico de las instrucciones de load, ya que al ir cambiando el contenido de la memoria, distintos 'loads' de la misma dirección de memoria obtienen distintos resultados.

A continuación se analizarán los resultados de una de las simulaciones más importantes que componen el trabajo mencionado. Los resultados a analizar corresponden a simulaciones de programas del conjunto de benchmarks SPEC95 realizadas con el conjunto de herramientas SimpleScalar [33]. La simulación consiste en ejecutar cada uno de los benchmarks seleccionados, almacenando hasta 2000 instancias únicas por instrucción estática. Para determinar si una instancia dinámica se encuentra repetida, se verifica que sus operandos sean los mismos que los de una instancia anterior. Notemos que esta condición es mas fuerte que la que surge de la definición que se provee en ese trabajo [29].

Tabla 2.3: Estadísticas de reuso de instrucciones [29]

Benchmark	**Inputs**	**Instrucciones dinámicas**		**Instrucciones estáticas**		
		Total (millones)	**Repetidas %**	**Total**	**Ejecutadas % del total**	**Repetidas % sobre ejec.**
go	null.in (ref)	1000	85.2	84552	62.9	93.4
m88ksim	ctl.in (ref)	1000	98.8	37824	4.5	97.7
ijpeg	vigo.ppm (traim)	942.2	79.3	58894	25.4	98.1
perl	scrabble.in (train)	555.6	84.2	73850	22.3	65.6
vortex	vortex.in (train)	1000	93.2	125018	28.3	93.5
li	22.lsp files (ref)	1000	77.8	23026	23.6	92.0
gcc	reload.i (ref)	666.3	75.5	299988	39.5	87.7
compress	bigtest.in (ref)	1000	56.9	13798	13.1	66.3

Como se ve en la columna 'Repetidas', la cantidad de instrucciones repetidas es realmente muy importante. Estas estadísticas son las que justifican las técnicas de reuso de instrucciones y predicción de valores, del mismo modo que un estudio sobre localidad espacial y temporal de datos avala el uso las memorias cache en un procesador.

La pregunta que surge cuando se ven estos resultados es, naturalmente, cuál es la causa de tanta redundancia. La ejecución de un programa consiste en la aplicación de un conjunto de operaciones sobre un conjunto de entradas, pero existe una diferencia importante entre el conjunto dinámico de operaciones y el estático, o sea el programa que escribe el programador. Esta diferencia existe porque cuando uno escribe un programa, no escribe explícitamente toda la secuencia de operaciones a realizar. Por ejemplo, cuando se desea obtener la suma de los elementos de un arreglo de 20 posiciones, uno no escribe directamente las 20 sumas necesarias para obtener el resultado deseado, sino que escribe un ciclo que recorre el arreglo sumando sus componentes. Al ejecutarse el programa, el cuerpo del ciclo va acumulando la suma de los componentes del vector, y las instrucciones que controlan el flujo del programa determinan si hay que seguir acumulando o se ha llegado al final.

En ese ejemplo se nota claramente que hay una separación en la función que cumplen las instrucciones en un programa. Por un lado, las instrucciones del cuerpo del ciclo se dedican a realizar la suma solicitada. Por otro lado, las instrucciones que controlan el flujo están construyendo la secuencia dinámica de operaciones. Conceptualmente, están construyendo en forma dinámica un programa compuesto por veinte sumas para obtener la sumatoria de

los elementos del vector. En el resto del trabajo, nos referiremos a estas instrucciones como aquellas dedicadas a construir el camino dinámico de ejecución. Además, diremos que las otras instrucciones están dedicadas al cómputo específico del programa.

Esta diferencia entre el conjunto dinámico y el estático de las operaciones de un programa se produce porque es deseable tener una representación estática compacta de la computación a realizar y porque pretendemos tener programas que puedan operar sobre distintos conjuntos de entradas. Si reemplazaramos el ciclo para sumar los elementos del vector por veinte sumas, no podríamos utilizar vectores de distinta longitud distinta a 20.

Otra razón para explicar la existencia de tanta redundancia es que, tal como se mencionaba mas arriba, los datos sobre los cuales se opera están organizados en estructuras de datos, por lo que hay instrucciones que se ejecutan para direccionar y acceder a los datos para realizar el proceso de computación propiamente dicho.

Una vez comprobada la existencia de esta redundancia en los programas, el próximo paso es analizar la forma de aprovecharla para mejorar el tiempo de ejecución de los programas.

Un punto en común para todos estos esquemas es la forma de aprovechar una instrucción repetida. Todos los esquemas que revisaremos a continuación están basados en evitar que las instrucciones repetidas pasen por la etapa de ejecución en el pipeline. De esta manera, una instrucción 'repetida' pasará por las siguientes etapas: fetch, despacho, inicio, write-back y commit, pasando por alto la etapa de ejecución, tal como se especifica en la figura 2.16.

Dado que este trabajo se relaciona de manera directa con el tema de reuso de instrucciones, de aquí en adelante se dejará de lado el tema de predicción de valores para profundizar sobre reuso de instrucciones.

Existen varios trabajos sobre reuso de instrucciones publicados recientemente, pero uno de los trabajos en los que esta idea se aplica de una manera mas generalizada es [27]. En dicho trabajo se presentan tres esquemas de reuso de instrucciones implementables en hardware, aunque el énfasis no está puesto en los detalles de la posible implementación sino en las funcionalidades requeridas por cada uno de los esquemas para funcionar.

2.7.1 Esquema Sv

El esquema Sv es una implementación directa del concepto de reuso de instrucciones. Los valores de los operandos se almacenan junto con el resultado de la ejecución. El funcionamiento del esquema es el siguiente:

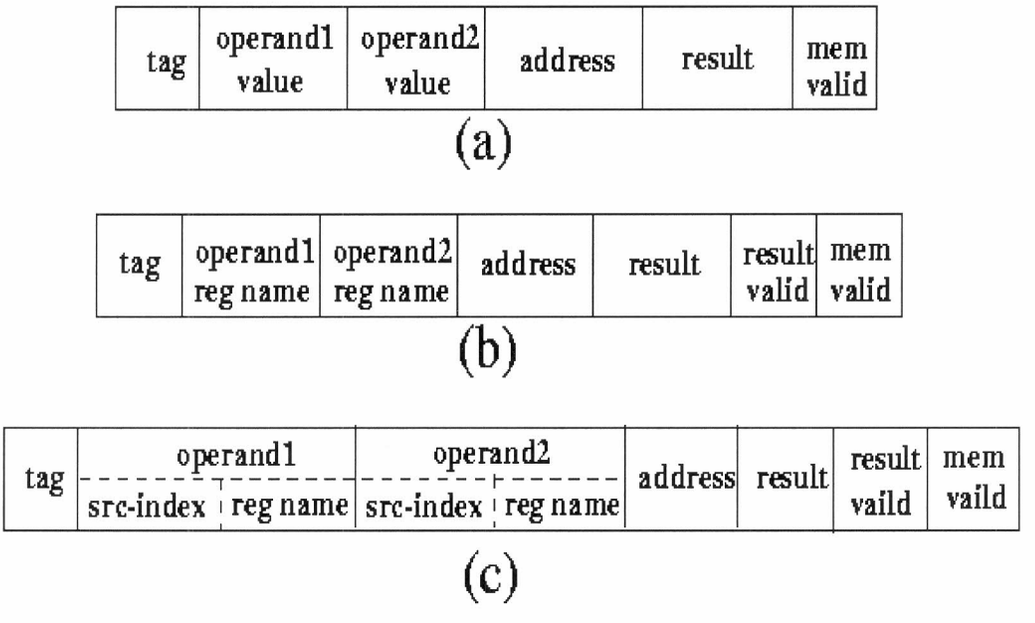
Cuando se decodifica una instrucción, se comparan los valores de los operandos actuales contra los almacenados en el buffer de reuso. Si son los mismos, entonces el resultado es reusado. Las instrucciones de acceso a memoria, dado que en realidad son dos operaciones en una (cálculo de dirección y el acceso a memoria en si mismo) precisan una operatoria distinta. Por un lado, el cálculo de la dirección efectiva de memoria puede ser reusado si los operandos no cambiaron. Por otro lado, el acceso a memoria de la instrucción sólo podrá ser reusado en caso de que ningún store anterior haya modificado la posición de memoria referenciada. En el caso de los store, el acceso a memoria, la escritura, no puede ser reusado.

En la figura 2.19 se muestra la estructura del buffer de reuso en cuestión.

El campo *tag* contiene parte del PC para direccionar la tabla. En los campos *result*, *operand value1* y *operand value2* se almacenan el resultado y los valores de los operandos de la instrucción. Estos campos son los que se utilizan para determinar la reusabilidad de la instrucción. Los campos *memvalid* y *address* son utilizados si el cálculo de dirección de memoria puede ser reusado o no. El bit *memvalid* indica si el dato cargado de memoria (que está en el campo *result*) es válido. El campo *address* almacena la dirección efectiva del acceso.

El test de reuso en este esquema consiste en comparar los valores de los operandos de la instrucción contra los almacenados en la entrada correspondiente del buffer. Si hay coincidencia, entonces el campo *result* (si la instrucción no es un load) o el campo *address* (para instrucciones load) son válidos. Si la instrucción es un load, entonces además de ese chequeo debe se debe revisar el bit *memvalid* para determinar si el resultado del load (almacenado en el campo *result*) puede reusarse.

Figura 2.19: Estructura de los buffers de reuso [27]



Para instrucciones que no son load, no es necesario realizar ningún tipo de invalidación en la tabla, ya que los operandos determinan de forma unívoca el resultado. Por otro lado, las entradas que corresponden a instrucciones de tipo load se invalidan cuando se realiza un store a la misma dirección. Mas precisamente, cuando se realiza un store se deben buscar

las entradas en la tabla cuyo campo *address* sea igual al de la dirección del store, y se pone a cero el bit *memvalid*.

2.7.2 Esquema Sn

El objetivo de este esquema es simplificar el test de reuso. En lugar de almacenar en el buffer de reuso los valores de los operandos, se almacenan los nombres de los registros de los operandos de la instrucción. La figura 2.19(b) muestra el esquema de una entrada en el buffer de reuso de este esquema. Se puede apreciar que la diferencia mas importante entre ambos esquemas reside en que en el lugar donde están los valores de los operandos en la entrada del Sv se encuentra ahora el nombre del registro. La segunda diferencia es la existencia del big *resultvalid*, que indica la validez del resultado, que es puesto a uno cuando un registro es insertado por primera vez en el buffer.

El test de reuso consiste simplemente en revisar el estado de los bits *resultvalid* y *memvalid*. El cálculo de dirección efectiva en las instrucciones load/store y el resultado de cualquier otro tipo de instrucción puede ser reusado si el bit *resultvalid* está en uno. El resultado de una instrucción load puede ser reusado si tanto el bit *resultvalid* como el bit *memvalid* están en uno.

A diferencia del esquema Sv, es necesario un algoritmo de invalidación para actualizar el bit *resultvalid*. Además de que un store invalida a un load a la misma dirección, cuando un registro se escribe, se busca en el buffer de reuso las entradas cuyos campos de operandos coinciden con el nombre del registro escrito. El bit *resultvalid* se resetea en las entradas coincidentes, indicando que el resultado ya no puede ser reusado.

2.7.3 Esquema Sn+d

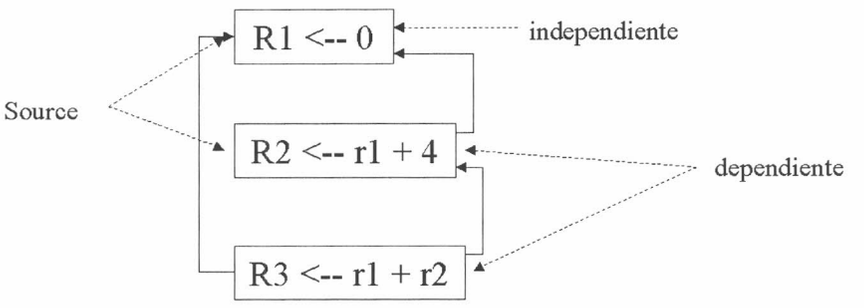
El esquema Sn+d extiende al Sn intentando establecer cadenas de instrucciones dependientes, y manteniendo el estado de la reusabilidad de las cadenas de instrucciones.

Para explicar el funcionamiento del esquema es necesario primero establecer algunas definiciones. Las instrucciones que producen valores usados por otras instrucciones en una cadena de instrucciones se llaman instrucciones *fuente*. Las instrucciones cuyas instrucciones fuente no están en la cadena, lo cual indica que su información sobre dependencias de datos no está disponible, se llaman *independientes*. Por último, aquellas instrucciones cuya instrucción fuente está en la cadena se llaman *dependientes*. La figura 2.20 ejemplifica estas definiciones.

Cada entrada en el buffer de reuso es bastante parecida a la del esquema Sn, excepto por el agregado del campo *src-index*. Los links de dependencias se crean almacenando el índice del buffer de reuso de la instrucción fuente. Un valor inválido se inserta en este campo si la instrucción fuente no existe en el buffer de reuso.

Además del buffer de reuso existe en este esquema una tabla auxiliar denominada RST (*Register Source Table*). La RST posee una entrada para cada registro de la arquitectura y mantiene cuál es la entrada del buffer de reuso que contiene o contendrá el último valor para ese registro. Cuando una entrada para una instrucción es reservada en el buffer, la entrada

Figura 2.20: Instrucciones fuente, dependientes e independientes [27]



en la RST correspondiente al registro destino de la instrucción se actualiza para apuntar a la entrada reservada. Si la instrucción que es la última productora de un registro no se encuentra en el buffer, entonces la entrada en la RST correspondiente a ese registro se marca como inválida.

El test de reuso para las instrucciones independientes es el mismo que en el esquema Sn. En el caso de las instrucciones dependientes, éstas pueden ser reusadas si sus instrucciones fuente (apuntadas por el campo *src-index*), son efectivamente las últimas productoras de sus operandos. Esta condición se determina con ayuda de la RST.

De la misma manera que en los esquemas Sv y Sn, los stores invalidan a los loads de la misma dirección reseteando el bit *memvalid*. Al igual que en el esquema Sn, las instrucciones independientes son invalidadas cuando sus operandos fuente son sobreescritos reseteando el bit *resultvalid*. Las instrucciones dependientes no necesitan ser invalidadas ya que su reusabilidad se establece utilizando su información de dependencias. En cambio, estas instrucciones se invalidan cuando sus instrucciones fuente son desalojadas del buffer de reuso. Para realizar esta operación se deben buscar en el buffer aquellas instrucciones cuyo campo *src-index* concuerde con el índice de la instrucción que se desaloja. El bit *resultvalid* de aquellas instrucciones que concuerden es reseteado indicando la no validez de la entrada.

2.8 Este trabajo...

El aporte de este trabajo consiste en un nuevo esquema de reuso basado en una definición alternativa de repetición. El estudio de esta nueva definición y los esquemas de reuso propuestos están apoyados en simulaciones realizadas con la herramienta de simulación SimpleScalar [33].

Estudiando detenidamente la definición presentada en [29] surge que requerir la existencia de una ejecución anterior de la misma instrucción estática para poder reusarla es demasiado fuerte. Si se relaja esta condición y solamente se requiere una ejecución anterior de la misma *operación* se pueden obtener esquemas que permitan mayores índices de reuso y que son útiles aun en el escenario de un cambio de contexto. Esto último es una gran ventaja sobre los esquemas tradicionales de reuso de instrucciones, ya que en trabajos anteriores, como el buffer de reuso se direcciona utilizando el contenido del registro PC, el buffer de reuso debe ser vaciado en cada cambio de contexto para evitar una ejecución errónea. Los esquemas que se presentan en este trabajo no utilizan el registro PC para indexar ninguna estructura de datos, por lo que las mismas no necesitan ser vaciadas en un cambio de contexto.

La definición de reuso que se utiliza en este trabajo lleva esquemas en los que no se reusan instrucciones, en el sentido de una operación asociada a una posición de memoria, sino que lo que se está reusando es la computación propiamente dicha. Es por esto que los esquemas que se presentan en este trabajo se llaman de reuso de computación y no de instrucciones. El esquema de reuso propuesto es comparable al Sv descripto anteriormente. En este caso, cada entrada en el buffer de reuso contendrá el código de operación, los valores de ambos operandos y el resultado obtenido. Entonces, el test de reuso consiste en verificar que tanto el código de operación y los valores de los operandos de la instrucción a reusar coincidan con los almacenados. El caso de las instrucciones de acceso a memoria es especial, y será analizado en el capítulo 4. Por último, si dejamos por un momento de lado las instrucciones de acceso a memoria, es claro que no es necesario realizar ningún tipo de invalidación de entradas en el buffer de reuso.

Como primer paso para el desarrollo de un esquema hardware de reuso de computación se debe realizar una cuantificación del fenómeno de repetición según la definición usada en este trabajo. Este estudio permitirá conocer la magnitud del fenómeno y analizar la mejor manera de explotarlo para obtener una mejora de performance.

En el capítulo 4 se analizarán las simulaciones realizadas para cuantificar el fenómeno de reuso de computación. Luego se analizarán los resultados obtenidos con el objetivo de desarrollar un esquema de reuso que permita aprovechar la redundancia observada en la primer parte.

Capítulo 3

La herramienta de simulación SimpleScalar

En la actualidad, dado el amplio espacio de diseño en arquitectura de procesadores y el costo de una implementación real, la situación habitual es trabajar utilizando simulaciones. Las simulaciones son una alternativa mucho mas eficiente en términos de costo y versatilidad que experimentar con hardware real, dada la flexibilidad intrínseca del software. Existen muchos tipos de simuladores, desde aquellos que simulan cada una de las compuertas lógicas del circuito, hasta los que simplemente simulan el comportamiento definido por la arquitectura.

Un simulador de arquitectura es una herramienta que reproduce el comportamiento de un determinado dispositivo, permitiendo obtener las mismas salidas que el dispositivo simulado y además métricas sobre su comportamiento. En nuestro caso, el dispositivo que intentamos simular es un procesador. Los simuladores de arquitectura pueden dividirse en dos grandes ramas: los funcionales y los de rendimiento. Los simuladores funcionales implementan la arquitectura del procesador que se estudia, o sea, la visión del programador. En este tipo de simuladores no se provee ningún detalle sobre el proceso interno que involucra la ejecución de una instrucción. Por otro lado, los simuladores de rendimiento implementan la microarquitectura del dispositivo estudiado, con lo cual se logra una simulación detallada del comportamiento microarquitectónico del procesador que se estudia.

A partir de este punto se puede seguir abriendo cada una de las ramas de esta clasificación. Por el lado de los simuladores funcionales, podemos dividirlos segun trabajen con trazas o por ejecución de código. La traza de un programa se compone de todas las instrucciones dinámicamente ejecutadas, junto con todos los accesos a memoria realizados. Los simuladores que procesan trazas son los mas simples, ya que no precisan implementar la funcionalidad de cada instrucción de la arquitectura, sino que obtienen el resultado de la ejecución de cada instrucción de la traza misma. Al no ejecutar cada instrucción estos simuladores suelen ser extremadamente rápidos. Por otro lado, existe una desventaja importante, ya que la traza debe ser generada anteriormente a la simulación, lo cual puede ser un proceso complejo. Un simulador que trabaja por ejecución de código ejecuta el programa original, lo que es equivalente a generar la traza en tiempo de ejecución. Esta característica hace mas difícil

su implementación, pero por otro lado otorga muchas ventajas, sobre todo en cuanto a la flexibilidad que brinda con respecto a los programas a ejecutar. Los simuladores por ejecución también pueden ser divididos según su forma de ejecutar el código. Una manera es hacerlo como un intérprete, o sea, leer cada instrucción e implementar su funcionalidad dentro del ambiente del simulador mismo. Por otro lado, en el caso que el procesador simulado sea compatible a nivel de arquitectura con el anfitrión, existe la posibilidad de ejecutar realmente cada instrucción, copiando luego el resultado obtenido en el estado del procesador simulado.

Dado que en el trabajo a realizar es necesario realizar tanto estudios de performance como análisis de los valores almacenados en estructuras de datos internas del procesador, es indispensable trabajar con un simulador que trabaje por ejecución de código. En la actualidad existen numerosas herramientas de ese tipo, por lo que la elección de SimpleScalar se debió en un grado importante a que la modularidad de los simuladores permite realizar modificaciones en cualquier aspecto de la arquitectura simulada de forma simple. Por otro lado, la arquitectura que define el simulador es ampliamente utilizada en los trabajos de investigación sobre reuso, lo cual permite comparar con mayor precisión los resultados obtenidos con los de otros trabajos. Finalmente, el código fuente de dicha herramienta es público, al igual que los benchmarks SPEC95 compilados para la arquitectura, lo cual implica un ahorro de tiempo muy considerable.

La herramienta SimpleScalar es un conjunto de módulos que permiten implementar simuladores de arquitectura tanto funcionales como de rendimiento, basados en ejecución interpretada de código. Además de estos módulos básicos se incluyen varios simuladores funcionales y un simulador de rendimiento, que permiten al desarrollador contar con una base para experimentar.

3.1 Arquitectura SimpleScalar

El primer paso para trabajar con una herramienta de simulación es conocer la arquitectura que ésta simula. La arquitectura SimpleScalar es un derivado de la arquitectura MIPS-IV. El conjunto de herramientas define versiones de la arquitectura tanto big-endian como little-endian de la arquitectura para incrementar la portabilidad del mismo, ya que la versión a utilizar debe ser coincidente con el tipo de endianness de la máquina anfitriona.

La arquitectura del conjunto de instrucciones SimpleScalar es un superconjunto del MIPS con algunas diferencias, como por ejemplo la no existencia de delay slots y el formato de instrucción de 64 bits. Los registros de la arquitectura, cuyo nombre y semántica se pueden observar en la tabla 3.1, son los mismos que en la arquitectura MIPS-IV.

El formato de instrucción, que es una extensión del MIPS a 64 bits, está diseñado para facilitar la exploración sobre conjuntos de instrucciones. Los campos para especificar registros son de 8 bits para permitir el agregado de nuevos registros en la definición de la arquitectura. El código de operación de 16 bits está en una posición fija en los distintos tipos de instrucciones para acelerar la decodificación de las instrucciones. Además, existe

Tabla 3.1: Registros en la arquitectura SimpleScalar [33]

Nombre hardware	Nombre software	Descripción
$0	$zero	registro fijo a cero
$1	$at	reservado por el ensablador
$2 - $3	$v0 - $v1	registros de retorno resultado de función
$4 - $7	$a0 - $a3	registros de valor de argumento de función
$8 - $15	$t0 - $t7	registros temporarios, salvados por llamador
$16 - $23	$s0 - $s7	registros salvados, guardados por llamado
$24 - $25	$t8 - $t9	registros temporarios, salvados por llamador
$26 - $27	$k0 - $k1	reservados por el SO
$28	$gp	Puntero global
$29	$sp	Puntero de pila
$30	$s8	registros salvados, guardado por llamado
$31	$ra	registro de dirección de retorno
$hi	$hi	registro de resultado alto
$lo	$lo	registro de resultado bajo
$f0	$f31	registros de coma flotante
$fcc	$fcc	Código de condición de coma flotante

un campo para anotaciones, que puede ser modificado luego de compilado el código, que permite modificar el juego de instrucciones sin necesidad de alterar el ensamblador.

SimpleScalar utiliza un espacio virtual de direcciones de 31 bits, organizada según indica la tabla 3.1.

3.2 Generación de código para el simulador

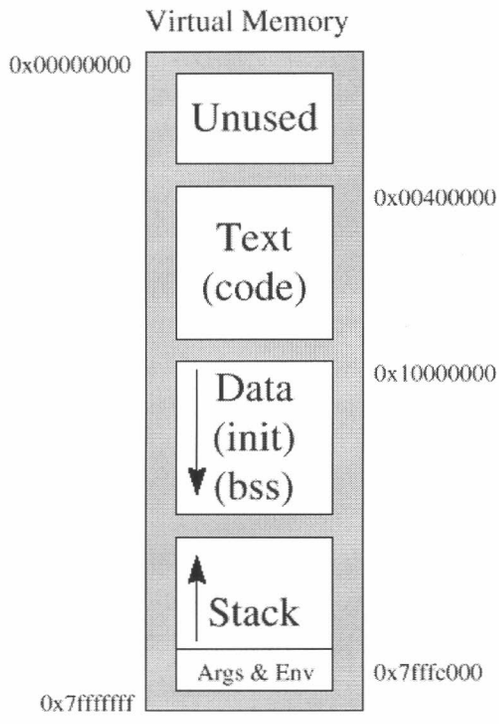
El primer paso para realizar una simulación es compilar el programa deseado para la arquitectura SimpleScalar. Es claro que para realizar dicha tarea es necesario disponer de compiladores, ensamblador y bibliotecas de funciones standard para dicha arquitectura.

El conjunto de herramientas SimpleScalar incluye un port de las herramientas GNU GCC, GAS, GLD y de la biblioteca GLIBC para la arquitectura SimpleScalar, de modo tal que cualquier programa puede ser compilado para ser ejecutado posteriormente por el simulador.

Este diseño permite maximizar la utilidad del conjunto, ya que el problema de la generación de código para un simulador muchas veces limita los programas que es posible ejecutar en él a un conjunto bastante reducido.

La secuencia de generación de código, según se especifica en la figura 3.2, comienza con un código Fortran o C. El código Fortran debe pasar por la herramienta *f2c* para ser traducido a C. Una vez disponible el fuente en C, se lo compila utilizando la versión de *GCC* para la arquitectura SimpleScalar. Posteriormente se ensambla con *GAS* y se linkea con las bibliotecas standard de C, tambien portadas para la arquitectura. Como resultado

Figura 3.1: Organización de memoria en SimpleScalar [33]



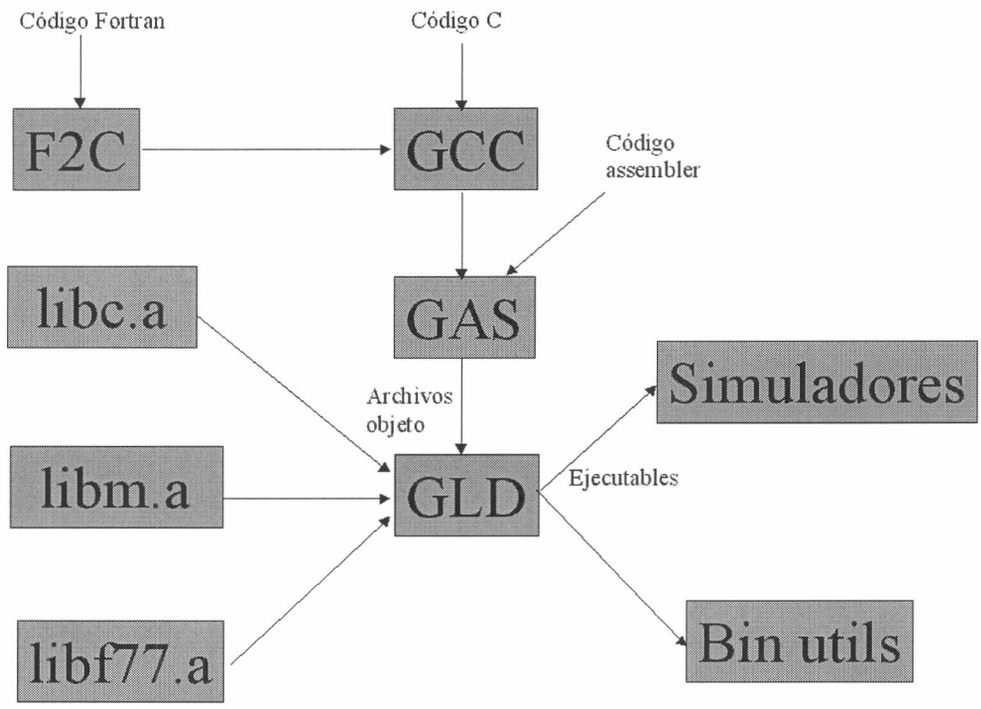
de este proceso se obtiene un ejecutable para el simulador. Como nota adicional en cuanto a programas disponibles para el simulador, es importante notar que están disponibles junto con el simulador los benchmarks del SPEC95 compilados. Dado que este es uno de los conjuntos de benchmarks mas utilizados en investigación sobre arquitectura, esto representa una gran ventaja a la hora de reducir el tiempo de preparación de las simulaciones. Además, en la versión 3.0 se incorporó en el paquete la posibilidad de ejecutar código binario de la arquitectura Alpha [34].

3.3 Estructura interna del simulador

A pesar de que se incluyen algunos simuladores base, SimpleScalar está diseñado de forma tal que sea el usuario final mismo el que programe los simuladores. Con este objetivo es que el diseño está compuesto por varios módulos que se unen para formar los distintos simuladores.

Un programa para la arquitectura SimpleScalar se compone de instrucciones pertenecientes al SimpleScalar ISA y de llamadas al sistema operativo. Esta clasificación determina que el núcleo funcional del simulador está compuesto por dos módulos: el de la definición de la máquina simulada y el proxy de llamadas al sistema operativo. El módulo de definición

Figura 3.2: Cadena de generación de código en SimpleScalar [33]

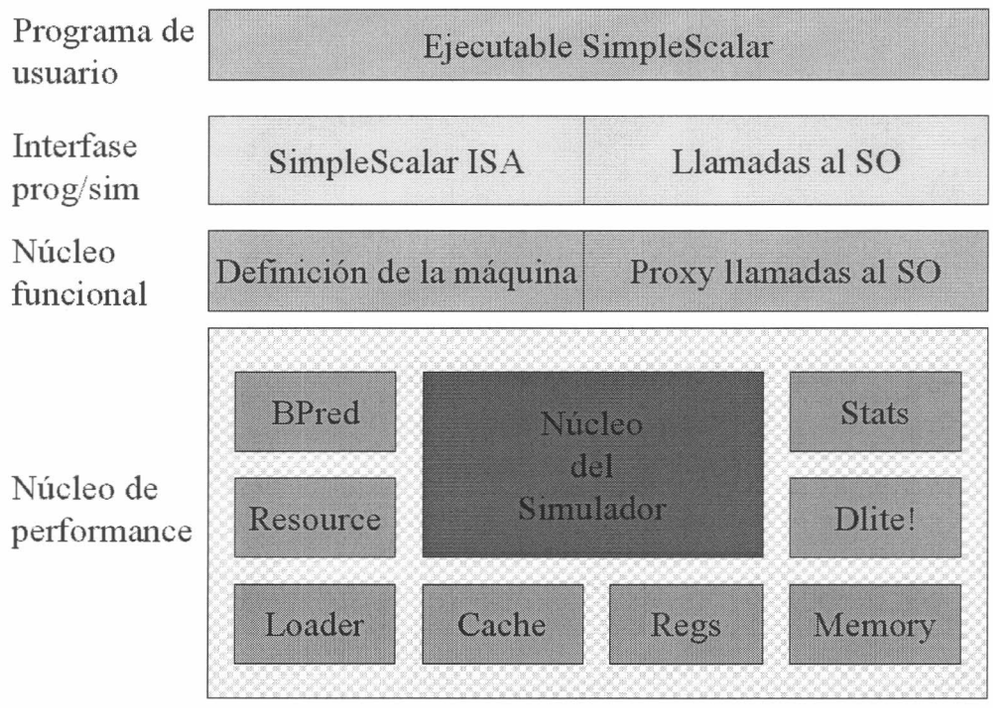


de la máquina especifica el conjunto de instrucciones y la implementación de cada una de ellas en el simulador. El proxy interpreta las llamadas al sistema operativo simulado, las ejecuta en el sistema operativo real, y copia los resultado en la máquina simulada. Esta es una característica muy importante, ya que permite ejecutar casi cualquier tipo de código en el simulador.

A partir de estos dos componentes básicos es que se construyen los simuladores. Para hacer mas sencilla esta tarea es que existe otro grupo de módulos, los pertenecientes al núcleo de rendimiento. En este grupo podemos encontrar el módulo cargador, el de registros, el de memoria, el de cache, el predicción de saltos, estadísticas, etc. La mayoría de los módulos de este grupo son opcionales para construir un simulador. Por ejemplo, lo mas común es que un simulador utilice el módulo cargador y de registros, pero el programador podría decidir utilizar un módulo de memoria distinto del provisto en la distribución.

Los simuladores provistos en la distribución son una excelente base para experimentar. En la mayoría de los casos, sólo es necesario realizar modificaciones pequeñas sobre alguno de estos simuladores base para realizar las simulaciones deseadas.

Figura 3.3: Estructura de SimpleScalar [33]



3.4 Ejecución de un proceso en SimpleScalar

En esta sección se describirán los procesos involucrados en la ejecución de un proceso en SimpleScalar. Esta descripción se realizará enumerando las distintas llamadas que se realizan en el procedimiento *main* del simulador.

- Registro de las opciones del simulador (sim_reg_options)
- Chequeo de las opciones del simulador (sim_check_options)
- Registro de las estadísticas específicas del simulador (sim_reg_stats)
- Inicialización del estado del simulador (sim_init)
- Comienzo de la simulación (sim_main)
- Vuelco de las estadísticas recolectadas durante la simulación (sim_aux_stats)
- Des-inicialización del estado del simulador (sim_uninit)

El punto mas interesante de este proceso está en la función sim_main, que es la que realiza la simulación propiamente dicha. El trabajo del programador consiste en modificar esta función para adaptarla a sus necesidades. Los módulos sim-safe, sim-fast, sim-bpred, sim-outorder, entre otros, son algunos de los módulos donde se implementa esta función, cada uno incorporando características particulares a la simulación.

3.5 Sim-Safe, un simulador funcional básico

Sim-safe es el simulador más sencillo provisto en la distribución SimpleScalar. Es un simulador funcional que provee mínimo detalle y casi no provee estadísticas sobre el proceso ejecutado. Su funcionamiento es muy sencillo, ya que consiste en un ciclo infinito en el cual se hace el fetch de la instrucción corriente y se la ejecuta en cada iteración del mismo. Este ciclo se corta únicamente al ejecutar el proceso una llamada al sistema operativo simulado, para terminarlo.

3.6 Sim-Outorder, simulador de rendimiento

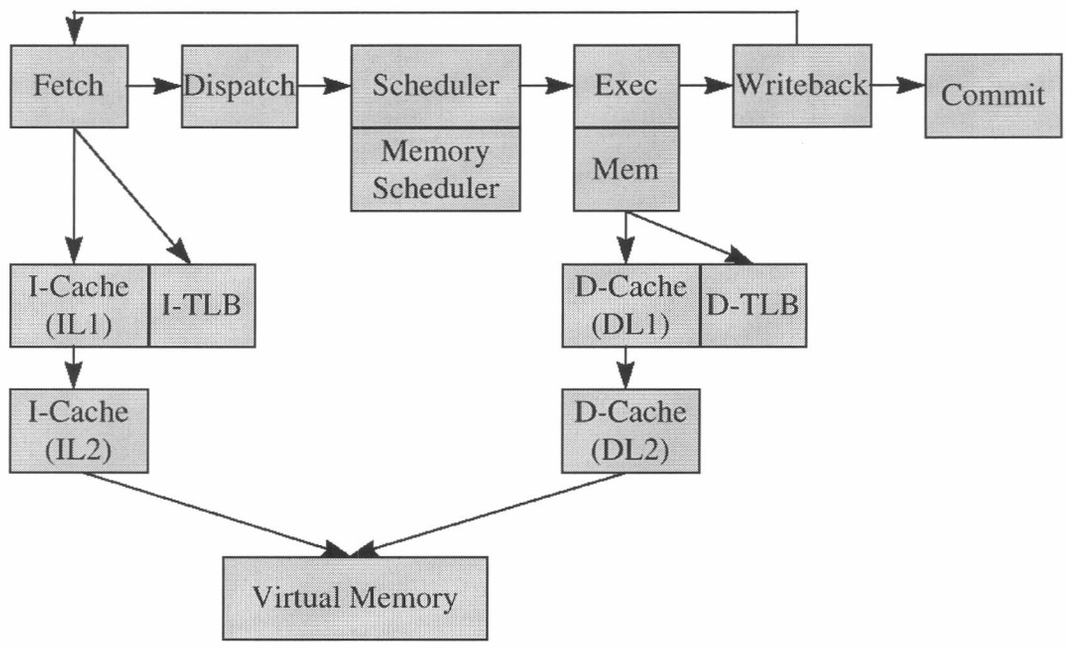
Sim-Outorder es el simulador más detallado que se incluye en la distribución SimpleScalar. Este programa simula el estado microarquitectónico de un procesador superescalar ciclo a ciclo, permitiendo obtener estadísticas muy detalladas sobre todos los aspectos del procesador. La arquitectura del pipeline del procesador simulado es la que se muestra en la figura 3.4. Al igual que Sim-Safe, el funcionamiento básico del simulador está controlado por un ciclo infinito, pero en este caso en cada iteración del ciclo se ejecuta un procedimiento por cada etapa del pipeline. La ejecución se realiza en el sentido inverso del avance de las instrucciones, ya que de esta manera se resuelven los problemas de sincronización de los elementos de estado entre las etapas con una sola pasada por cada etapa del pipeline por ciclo.

- **ruu_commit:** Esta función maneja las instrucciones de la etapa de writeback que están listas para ser retiradas, realizando el retiro en orden de las mismas.

- **ruu_writeback:** La etapa writeback del procesador se realiza en esta función. En cada ciclo busca si hay eventos de writeback planificados para ese ciclo, para cada uno de ellos recorre la lista de dependencias de la instrucción correspondiente y marca las instrucciones dependientes indicando que se ha satisfecho la dependencia. En caso de que la instrucción ya no tenga ninguna dependencia, la rutina la marca como lista para su inicio. La etapa de writeback también detecta predicciones de salto incorrectas; cuando se determina que se ha realizado una predicción incorrecta se vuelve atrás el estado del procesador al último checkpoint, descartando las instrucciones ejecutadas equivocadamente.

- **ruu_issue y ruu_refresh:** la etapa issue del procesador se implementa en estas dos funciones. Estas dos funciones implementan el despertar de las instrucciones y su

Figura 3.4: Pipeline de Sim-Outorder [33]



asignación a unidades funcionales, manteniendo el estado de las dependencias, tanto de registros como de memoria. En cada ciclo, las rutinas de planificación determinan cuáles son las instrucciones tales que todas sus dependencias de datos en lo referente a registros están satisfechas. El inicio de instrucciones de tipo load se detiene en caso de que algún store anterior todavía no tenga resuelta su dirección efectiva de acceso. Si la dirección de un store anterior coincide con la del load, el resultado del store es enviado directamente a la instrucción load. En caso contrario, el load es enviado al sistema de memoria. La etapa de ejecución también se maneja en *ruu_issue*. En cada ciclo se toman tantas instrucciones listas de la cola del planificador como sea posible, sin superar el máximo especificado en la configuración del simulador. La disponibilidad de unidades funcionales también se verifica, y si existen ports disponibles las instrucciones son iniciadas. Por último, la rutina planifica los eventos de writeback utilizando los tiempos de latencia de las unidades funcionales.

- **ruu_dispatch:** en esta función es donde se implementa la decodificación de instrucciones y el renombre de registros. Una vez por ciclo, esta función toma tantas instrucciones como puede de la IFQ (sin pasarse del máximo establecido por la configuración del simulador), y las coloca en la cola del planificador.

- **ruu_fetch:** esta función implementa la etapa de fetch del procesador. La unidad de fetch modela el ancho de banda de memoria para realizar el fetch de instrucciones, y toma como entrada el contenido del registro PC, el estado del predictor y la señal de predicción incorrecta de las unidades ejecutoras de saltos. Como resultado, se alimenta la IFQ (*Instruction Fetch Queue*) con las instrucciones obtenidas de la cache de instrucciones. En cada fetch no se pueden obtener instrucciones de más de una línea de cache. Además, esta etapa se bloquea ante un fallo de la cache hasta que el fallo se resuelve. Una vez colocadas las instrucciones en la IFQ, se consulta al predictor de próxima línea para saber qué instrucciones traer en el próximo ciclo.

3.7 Otros simuladores

Además de los dos simuladores anteriormente detallados, SimpleScalar provee los siguientes simuladores:

- *sim-fast*: versión acelerada de sim-safe. Elimina los controles de errores para lograr una simulación funcional mucho mas rápida que sim-fast.
- *sim-bpred*: Agrega un predictor de saltos al simulador funcional básico. A las estadísticas propias del simulador funcional se agregan las del predictor (hits, accesos).
- *sim-cache*: simula el comportamiento de caches de nivel 1 y 2, permitiendo configurar la cantidad de conjuntos, nivel de asociatividad, tamaño de bloque y política de reemplazo.

3.8 Modificando un simulador

En esta sección se describe el modo de modificar el simulador sim-outorder para implementar reuso de computación, de forma tal de ejemplificar cómo trabajar con SimpleScalar en otros proyectos.

Para este proyecto en particular, las modificaciones a realizar consisten en:

- Agregar opciones para habilitar o deshabilitar el reuso de computación, los mecanismos de filtrado y establecer la cantidad máxima de instrucciones a ejecutar.
- Modificar la etapa de decodificación para acceder al buffer de reuso.
- Agregar a las estadísticas a imprimir al finalizar la simulación aquellas relacionadas con el esquema de reuso de computación.

A continuación analizaremos cada una de estas modificaciones en detalle.

3.8.1 Opciones

Cuando el usuario ejecuta el simulador desde la línea de comando, existen numerosas opciones que se pueden especificar para configurar el comportamiento de la máquina simulada. Dadas las necesidades específicas de la simulación que se intenta desarrollar es necesario agregar algunas opciones mas.

SimpleScalar ya provee un módulo para el tratamiento de opciones del simulador. En nuestro caso, deseamos agregar a las opciones que sim-outorder provee las siguientes :

- *-cr_enabled* Si el usuario desea habilitar el reuso de computación deberá especificar este parámetro en *true*.
- *-cr_filtering* Este parámetro se utilizará para habilitar o deshabilitar el mecanismo de filtrado del esquema de reuso.
- *-max:inst* Con esta opción el usuario podrá especificar la cantidad máxima de instrucciones a ejecutar.

Para registrar estas opciones hay que modificar la función *sim_reg_options* de sim-outorder, agregando una llamada por cada opción a la función de registro del módulo *option* que corresponda según el tipo de dato de la opción a registrar. Estas funciones permiten asociar una variable con cada opción. Para nuestro caso, será necesario llamar a la función *opt_reg_flag* para las opciones cr_enabled y cr_filtering, y a la función *opt_reg_uint* para la opción max:inst. Con el simple hecho de agregar estas tres llamadas, el usuario ya tiene a su disposición esas tres opciones para el simulador. Mas tarde, el programador deberá agregar el código para que estas opciones se comporten como se diseñó.

3.8.2 Incorporación del reuso de computación

Este item involucra dos modificaciones. Por un lado, se debe incorporar dentro del pipeline el test de reuso de computación, y además se debe modificar el pipeline para alterar el camino que siguen las instrucciones reusadas. Para incorporar el test de reuso es necesario modificar la etapa *dispatch* para poder obtener el código de operación y los valores de los operandos que las instrucciones que pasan por esa etapa utilizan. Una vez obtenidos estos valores, se llama a una función propia que verifica si hay un acierto o no en el buffer de reuso. En caso de haber acierto, se marca la instrucción como reusada. Para que las instrucciones reusadas eviten la etapa de ejecución, se altera la función dispatch de la siguiente manera. En su versión original, esta función encola las instrucciones listas para que luego la ruu_issue la envíe a ejecutar cuando haya unidades funcionales disponibles. El cambio a realizar consiste en que, si la instrucción a encolar ha sido marcada como reusada, entonces directamente se la envía a la etapa de write-back.

3.8.3 Incorporación de estadísticas de reuso de computación

Para analizar el comportamiento del buffer de reuso es necesario conocer la cantidad de accesos realizados, la cantidad de inserciones y la cantidad de hits obtenidos. Teniendo en cuenta que esas tres estadísticas se mantienen en tres variables, lo único que resta por hacer es indicarle al simulador que esas tres variables mantienen estadísticas que deben imprimirse al finalizar la simulación. Para realizar el registro de estas variables, se debe incluir una llamada a la función sim_reg_counter por cada una de las variables dentro del procedimiento del simulador donde se registran las estadísticas específicas del simulador (sim_reg_stats).

Capítulo 4

Cuantificando el reuso de computación

4.1 Introducción

En este capítulo se analiza de forma cuantitativa el fenómeno de reusabilidad de computación. Para realizar dicho análisis es necesario obtener estadísticas de reuso sobre programas reales, las cuales se obtienen mediante la simulación de la ejecución de los mismos en una arquitectura dada. Los resultados que se obtienen de estas simulaciones permite pasar a una segunda etapa, la de diseñar un esquema implementable en hardware que aproveche la redundancia existente en beneficio de una reducción del tiempo de ejecución.

Según se establece en [27], el fenónemo de repetición se produce cuando dos instancias dinámicas de la misma instrucción estática producen el mismo resultado. Dicho fenómeno se puede aprovechar para acelerar la ejecución de un programa, ya que no es necesario realizar la ejecución de aquellas instrucciones para las cuales se conoce cuál es el resultado que generarán. En dichos casos, no sería necesario que la instrucción pase por la etapa de ejecución del pipeline, sino que la misma podría pasar directamente desde la etapa de decodificación a la etapa de commit. La principal desventaja que presenta esta definición de repeteción es que, al estar atada a las instrucciones estáticas, no es posible realizar reuso entre diferentes threads de ejecución. En el resto de este trabajo se considerará que una instancia dinámica de una instrucción estática está repetida si tiene el mismo código de operación y los mismos inputs que una instancia dinámica anterior, ya sea de la misma instrucción estática o no. Notemos que esta definición , a diferencia de la usada en otros trabajos, no requiere que ambas instancias pertenezcan al mismo thread de ejecución. Esta es una diferencia muy importante, ya que permite que los esquemas basados en esta definición sean útiles incluso en el escenario de ambientes multitarea, cosa que no sucede en esquemas tradicionales. Normalmente, ante un cambio de thread de ejecución las estructuras de datos deben vaciarse debido a que los datos que contienen son específicas del thread que se está ejecutando, con lo cual utilizar esta información al cambiar de thread puede llevar a resultados incorrectos. El vaciamiento de la estructura implica una pérdida de performance debida al tiempo que

toma completar nuevamente la estructura con información sobre el nuevo thread.

4.2 Instrucciones sobre las que se aplicará el esquema de reuso.

Tabla 4.1: Estadísticas de reuso de valores

Benchm.	# Inst (millones)	Integer		Float Integer		Single		Double	
		Insert.	Hit %	Insert.	Hit %	Insert.	Hit %	Insert.	Hit %
applu	1000	492.91M	76.40	359.88M	77.98	0.0	0	243.12M	38.61
cc1	235.58	156.4M	80.89	0.1M	99.99	0.0M	0	0.0M	0
compress	1000	632.54M	63.09	152.71M	48.51	0	0	14.58M	0.29
fpppp	1000	140.13M	91.35	734.98M	71.28	0.09M	99.87	325.19M	45.63
go	1000	818.13	77.78	0.0M	0	0.0M	0	0.0M	0
hydro2d	1000	438.53M	45.13	330.13	95.56	0.0M	0	148.77M	94.02
li	537.87	333.20M	85.04	0.004M	99.97	0.0M	0	0.0M	0
m88ksim	244.75	178.86M	94.09	0.0M	85.89	0.0M	62.96	0.0M	88.33
mgrid	1000	424.58M	35.70	547.73M	69.80	0.0M	0	264.12M	10.12
su2cor	1000	627.07M	81.52	26.87M	99.99	0.0M	0	8.88M	99.99
swim	1000	372.26M	59.03	263.79M	75.50	215.02M	33.32	32.78M	66.75
tomcat	1000	551.14M	79.95	77.5M	96.21	0.0M	0	58.52M	78.93
turb3d	1000	595.76M	55.60	190.82M	94.04	0.0M	93.96	137.78M	92.26
wave5	1000	431.47M	72.96	236.36M	81.26	0.0M	0	139.29M	33.24
MEDIA			71.32	-	78.28	-	20.71	-	46.29

El esquema de reuso propuesto en este trabajo se aplicará únicamente a las instrucciones de aritmética entera, por lo que se dejan de lado las instrucciones de transferencia de control, aritmética de coma flotante y acceso a memoria. Varias son las razones para acotar las instrucciones sobre las que se aplicará el esquema de reuso, principalmente la búsqueda de una implementación hardware sencilla y una buena relación costo/beneficio. Bajo estas dos premisas es que se limita el esquema a desarrollar, teniendo en cuenta que la redundancia que yace en el proceso del cálculo del camino dinámico de ejecución y las operaciones booleanas debería ser mayor que en otro tipo de procesos, y que la mayor parte de esta computación se realiza con operaciones de aritmética entera.

4.2.1 Análisis de las instrucciones de salto

Las instrucciones de salto condicional podrían considerarse como un tipo de instrucción de aritmética entera, teniendo en cuenta que el cálculo de la condición generalmente consiste en una operación bastante sencilla. Bajo esta consideración, la aplicación de un esquema de reuso a estas instrucciones no agregaría demasiada complejidad. Sin embargo, no es éste el punto por el cual estas instrucciones no se tratan en este trabajo. El punto principal

consiste en que lo más importante a estudiar para estas instrucciones es la interrelación entre la predicción de saltos y un esquema de reuso. A simple vista, el esquema de reuso disminuiría la penalización para los saltos mal predichos, pero en realidad un estudio de la interrelación entre predictores de salto y esquemas como el que se presenta aquí, merecería un trabajo aparte.

4.2.2 Análisis de las instrucciones de load/store

Con respecto a las instrucciones de acceso a memoria, el aspecto principal por el cual han sido omitidas es que las mismas no pueden ser tratadas exactamente de la misma manera que las otras instrucciones. Esto es así porque es típico de estas instrucciones obtener resultados diferentes con los mismos inputs, ya que un store sobre una dirección de memoria cambia el resultado de los loads posteriores sobre la misma dirección. Bajo estas condiciones, para incorporar estas instrucciones en un esquema de reuso sería necesario agregar un esquema de invalidación similar al que se utiliza en los esquemas presentados en [27], con lo se agregaría bastante complejidad al hardware.

4.2.3 Análisis de las instrucciones de aritmética en coma flotante

A pesar de que las instrucciones de coma flotante por lo general toman mas de un ciclo en su etapa de ejecución, con lo que potencialmente podrían generar un gran ahorro de tiempo si fueran reusadas, se tomó la decisión de no incluirlas en el esquema de reuso.

Para comprender esta decisión es necesario remitirse a las primeras simulaciones que se realizaron para este trabajo. Estas simulaciones fueron realizadas con el objetivo de estudiar el grado de redundancia en los valores escritos en los bancos de registros enteros y de coma flotante. Como conclusión de estas simulaciones se pudo determinar que las instrucciones de coma flotante en precisión simple son muy pocas como para justificar el uso del esquema de reuso, y las de doble precisión generan resultados muy poco redundantes.

Con respecto a las operaciones con números enteros realizadas en la unidad de coma flotante, a pesar de que se observa bastante redundancia en la tabla de resultados, en una implementación real sería necesario duplicar el hardware del esquema de reuso solamente para las instrucciones de coma flotante, lo cual significa un costo demasiado alto.

4.3 Simulaciones

Para poder cuantificar el fenómeno de reuso de computación es necesario analizar todas y cada una de las instrucciones que ejecuta el procesador y determinar si la misma es reusable o no utilizando un determinado esquema de reuso. En la siguiente sección se describen el ambiente de simulación y el esquema de reuso utilizados con tal propósito. Una vez obtenidos los resultados de estas simulaciones, estos deben analizarse para diseñar un esquema de reuso que, aprovechando la redundancia existente, mejore la performance.

4.3.1 Ambiente de simulación

El núcleo central de todas las simulaciones realizadas en este trabajo es el conjunto de herramientas *Simplescalar* [33]. Este set de programas provee varios simuladores que van desde un simulador funcional muy simple hasta un simulador de performance que permite analizar el comportamiento de un procesador superescalar ciclo por ciclo, el cual puede ser configurado casi en todos sus detalles arquitectónicos. Las simulaciones se realizaron utilizando 14 programas seleccionados del conjunto de benchmarks SPEC95. Los valores de los inputs fueron tomados del conjunto 'Reference' de los benchmarks, que figuran en la tabla 4.2.

Tabla 4.2: Inputs utilizados

Benchmark	**Input**
go	null.in
m88ksim	ctl.raw
gcc	expr.i
compress	bitest.in
li	deriv.lsp
vortex	vortex.raw
tomcatv	tomcatv.in
swim	swim.in
su2cor	su2cor.in
hydro2d	hydro2d.in
mgrid	mgrid.in
applu	applu.in
turb3d	turb3d.in
fpppp	natoms.in
wave5	wave5.in

En todos los casos se ejecutaron los primeros 200 millones de instrucciones del programa. Es importante notar que aunque la cantidad de instrucciones simuladas es bastante importante, no lo es tanto para hacer una simulación de performance detallada. Sin embargo, dado que el objetivo de esta simulación es tener una primer medida de la redundancia existente en los programas segun la definición dada en el principio, y que el tiempo de simulación es excesivamente largo para los tamaños de cache mas grandes, la cantidad de instrucciones simulada se estableció en ese valor. Otro aspecto que vale la pena considerar es el de las instrucciones de inicialización de cada benchmark. Es conocido que una cantidad de instrucciones que depende de los parámetros de entrada no pertenece al cómputo propio del benchmark, sino que se dedica a inicializar estructuras de datos en memoria, verificar la correctitud de los parámetros, etc. Por lo general se intenta saltear la ejecución de estas instrucciones para que los resultados de la fase de inicialización no afecten los resultados del núcleo del benchmark, pero en este caso se decidió ejecutar el programa completo para estudiar el programa completo.

4.3.2 Esquema simulado

El esquema utilizado para cuantificar la reusabilidad consiste en una cache totalmente asociativa con política de reemplazo FIFO. Varios tamaños de la cache fueron usados de forma tal de poder analizar el comportamiento del esquema en función del tamaño de la cache. El funcionamiento del esquema es el siguiente. Cada vez que se inicia la ejecución de una instrucción de aritmética entera se accede a la cache utilizando el código de operación y el valor de los operandos. En caso de existir un acierto la instrucción se considera reusada. Si no hubiera acierto la instruccion es insertada en el primer lugar de la cache. Otra forma de actualización del buffer que fue estudiada consistió en insertar la instrucción en el buffer aun en el caso de encontrarla. La idea intuitiva para hacerlo es darle un mayor tiempo de vida dentro del buffer a una instrucción que ha demostrado que genera repeticiones. La desventaja que presenta esta política es que se desperdicia espacio en el buffer con instancias repetidas. Las simulaciones realizadas demostraron que esta desventaja es mucho peor que los beneficios que se pensaban.

4.3.3 Resultados

La tabla 4.4 y la figura 4.1 muestra los resultados obtenidos en las simulaciones utilizando 1K, 4K y 64K entradas en la cache. En la tabla 4.3 se describe el contenido de cada columna.

Tabla 4.3: Significado de las columnas de la tabla 4.4

Columna	Contenido
Insertado %	Cantidad de accesos a la cache sobre el total de instrucciones ejecutadas
Hit %	Cantidad de hits en la cache sobre el total de instrucciones ejecutadas
Eficiencia %	Total de hits dividido por el total de accesos a la cache

En la tabla 4.4 se puede observar que el 35% de las instrucciones ejecutadas son reusables para el caso de la cache de 64K entradas. Ademas es muy importante notar que al disminuir el tamaño de la cache a 4K entradas el promedio de instrucciones insertadas baja sólo al 30%, lo cual es muy positivo ya que permite pensar que las estructuras de datos necesarias para una implementación hardware no necesariamente tienen que ser grandes. Por otro lado la eficiencia de la cache no se comporta de una manera tan bondadosa como la cantidad de hits, ya que en el caso de la cache de 64K entradas la misma se ubica en torno al 93% pero en la de 1K entradas disminuye hasta un 76%. Este resultado no favorece una implementación real, ya que a pesar de que el resultado obtenido para la cache de 64K entradas es muy bueno, para una implementación hardware es deseable trabajar con estructuras mas pequeñas que permitan un tiempo de ciclo mas pequeño.

Con estas consideraciones en mente, de ahora en adelante se utilizará una cache de 4K entradas

Tabla 4.4: Estadísticas de reuso de computación

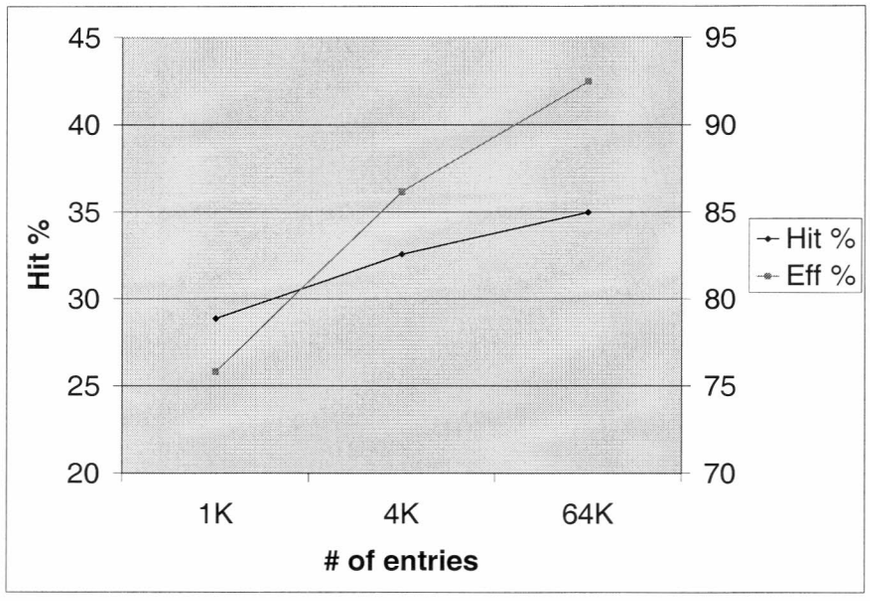
Benchmark	Insertado	64K entradas		4K entradas		1K entradas	
		Hit	Eficiencia	Hit	Eficiencia	Hit	Eficiencia
	%	%	%	%	%	%	%
applu	30.01%	24.76%	82.51%	23.53%	78.41%	18.88%	62.91%
cc1	39.71%	37.70%	95.17%	35.48%	89.36%	32.67%	82.27%
compress	57.42%	49.87%	86.85%	47.78%	83.21%	43.93%	76.50%
fpppp	3.04%	2.97%	97.75%	2.82%	92.70%	2.57%	84.53%
go	54.54%	54.45%	99.82%	45.41%	83.24%	45.35%	83.14%
hydro2d	38.21%	36.92%	96.64%	35.28%	92.33%	33.41%	87.43%
li	30.79%	30.78%	99.94%	27.73%	90.03%	24.29%	78.88%
m88ksim	47.79%	44.18%	92.46%	43.07%	90.12%	41.17%	86.14%
mgrid	36.78%	31.66%	86.07%	27.55%	74.89%	17.95%	48.80%
su2cor	41.69%	40.36%	96.81%	37.14%	89.10%	35.31%	84.69%
swim	23.80%	23.41%	98.35%	21.64%	90.95%	17.42%	73.19%
tomcat	37.18%	37.03%	99.58%	35.59%	95.73%	31.39%	84.42%
turb3d	58.85%	53.45%	90.92%	51.83%	88.07%	42.18%	71.67%
wave5	30.94%	22.26%	71.92%	21.15%	68.36%	17.65%	57.04%
MEDIA	37.91%	34.98%	92.48%	32.57%	86.17%	28.86%	75.82%

4.4 Conclusiones

En este capítulo se ha cuantificado el fenómeno de reuso de computación mediante la realización de distintas simulaciones variando el tamaño de la cache utilizada. Estas simulaciones han arrojado como resultado que un alto porcentaje de las instrucciones dinámicas son reusables según la definición utilizada en este trabajo.

A medida que el tamaño de la cache decrece, la eficiencia, en el sentido de usar la cache para almacenar las instancias dinámicas que tienen una mayor probabilidad de ser repetidas, se transforma en un factor cada vez mas importante. Al disponer de una menor cantidad de entradas en la cache, es necesario realizar un uso mucho mas eficiente del espacio disponible. El enfoque usado en este trabajo para incrementar la eficiencia consiste en capturar la redundancia expuesta por la computación utilizada para determinar el camino dinámico de ejecución, esperando que esta computación sea altamente redundante. Esta suposición se realiza sobre la base de que el resultado de las instrucciones de salto es altamente predecible, con lo cual es razonable suponer que las instrucciones que producen sus operandos también deben serlo. Para capturar las instrucciones involucradas en esta porción del flujo de ejecución, se necesita algun tipo de mecanismo para filtrar de la cache las instancias que no tienen una alta probabilidad de ser repetidas, de forma tal de aumentar la eficiencia. En el siguiente capítulo se elabora un mecanismo de filtro para lograr ese objetivo. La idea de un mecanismo de filtrado no es nueva, ya que [29] y [27] sugieren una política de insercion 'inteligente' para el buffer de reuso, ya que solamente un 20% de las instrucciones insertadas constituyen la mayor parte del reuso. El enfoque aquí propuesto para esa política es evitar la inserción de instrucciones en el buffer de reuso, aplicando ciertas reglas sobre los valores

Figura 4.1: Porcentaje de aciertos y eficiencia



de los operandos.

Capítulo 5

Filtros de instrucciones

5.1 Consideraciones iniciales

La idea detrás de filtrar instrucciones de la cache consiste en que es esperado que algunas instrucciones tengan mayor probabilidad de repetirse que otras. Por ejemplo, podríamos esperar que si

$$R1 = 1 \text{ y } R2 = 1$$

la operación

$$R3 = R1 + R2$$

tendrá mayores posibilidades de ser repetida que si

$$R1 = 143294 \text{ y } R2 = 1998123$$

Llamemos *filtro* a un mecanismo que permite que ciertas entradas pasen a través de el y no otras mediante la aplicación de un conjunto de reglas. El objetivo del filtro que es de interés para este trabajo es el de llenar la cache sólo con aquellas instancias relacionadas con el proceso de construcción del camino dinámico de ejecución y no con aquellas relacionadas con el proceso de computación específico del programa. Se espera que la computación involucrada en el primer grupo posea una mayor reusabilidad que otros ya que el objetivo final de las instrucciones pertenecientes a dicho grupo es proveer los operandos de las instrucciones de salto. Dado que, como se ha mostrado en secciones anteriores, el resultado de las instrucciones de salto es altamente predecible, lo mismo debería suceder con las operaciones que generan los operandos.

Analicemos el ejemplo de la figura 5.1. En ese pequeño ejemplo se pueden observar los dos tipos de operaciones mencionados. Por un lado, en la condición del 'IF' se pueden observar operaciones booleanas sencillas. En el programa assembler resultante este 'IF' concluye con un salto condicional que determinará cual de las ramas seguir. En general, se puede esperar

Figura 5.1: Ejemplo de código

```
if ((a AND b) OR (!c)) then
   Result[i] ←c[i]*d[i]+exp(p[i])
else
   Result[i] ←exp(p[i])
end if
```

que este salto sea altamente predecible, por lo que es razonable pensar que lo mismo debe ser para las operaciones realizadas para calcular los operandos de la instrucción de salto. Por otro lado, en las ramas 'THEN' y 'ELSE' hay llamadas a funciones, acceso a operandos en memoria y operaciones complejas, de las cuales no se espera que generen tanta redundancia.

En este ejemplo, el objetivo a lograr mediante el filtro es evitar la inserción en el buffer de reuso de las instrucciones en las ramas 'THEN' y 'ELSE'.

5.2 Definiciones básicas

Definimos que el filtro $[a, b]$ se aplica a un determinado esquema de reuso de computación cuando se requiere que al menos uno de los operandos de las instrucciones a ser insertadas en el buffer de reuso esté en rango [a,b]. Por otro lado, se define el *ancho* de un filtro como $|b - a|$. Uno de los objetivos mas importantes de este trabajo es el de encontrar un filtro que deje pasar la mayor cantidad posible de instrucciones, pero de manera tal de mantener la eficiencia tan alta como sea posible, para no desperdiciar espacio en el buffer de reuso con instrucciones que no van a ser reusadas.

5.3 Filtros iniciales

El primer filtro simulado fue el [0,1], ya que las operaciones que realizan la computación que se intenta capturar trabajan sobre valores booleanos la mayor parte del tiempo. Por otro lado, dado que el procesador simulado utiliza el valor 0 para 'Falso' y 1 para 'Verdadero', se obtiene naturalmente este filtro. En la tabla 5.1 se muestra el efecto de la aplicación del filtro [0,1] al esquema original con la cache de 4K entradas. Como resultados sobresalientes, podemos mencionar los siguientes puntos: se puede notar que la eficiencia se incrementa de forma significativa, lo cual indica que las instrucciones que se insertan en la cache tienen una muy alta probabilidad de ser reusadas.

Como efecto negativo de la aplicación de este filtro se puede ver que la cantidad de instrucciones insertadas en el buffer se ha reducido drásticamente también. Si consideramos el cociente entre la cantidad de instrucciones insertadas en el esquema sin filtrar y el que incorpora el fitro [0,1], se obtiene que sólo un 46% de las instrucciones que se habían insertado en el esquema original han logrado pasar el filtro propuesto. Este compromiso entre

Tabla 5.1: Estadísticas del filtro [0,1]

Benchmark	**Insertado** %	**Hit** %	**Eficiencia** %
applu	7.49%	7.30%	97.41%
cc1	21.64%	20.85%	96.37%
compress	32.94%	31.40%	95.32%
fpppp	1.05%	1.04%	98.99%
go	31.77%	31.74%	99.90%
hydro2d	21.73%	20.59%	94.75%
li	15.36%	14.42%	93.88%
m88ksim	27.91%	27.20%	97.46%
mgrid	12.68%	12.62%	99.49%
su2cor	23.94%	22.53%	94.09%
swim	8.32%	8.32%	99.98%
tomcat	21.02%	20.37%	96.91%
turb3d	24.62%	23.69%	96.21%
wave5	10.47%	9.42%	89.97%
MEDIA	18.64%	17.96%	96.48%

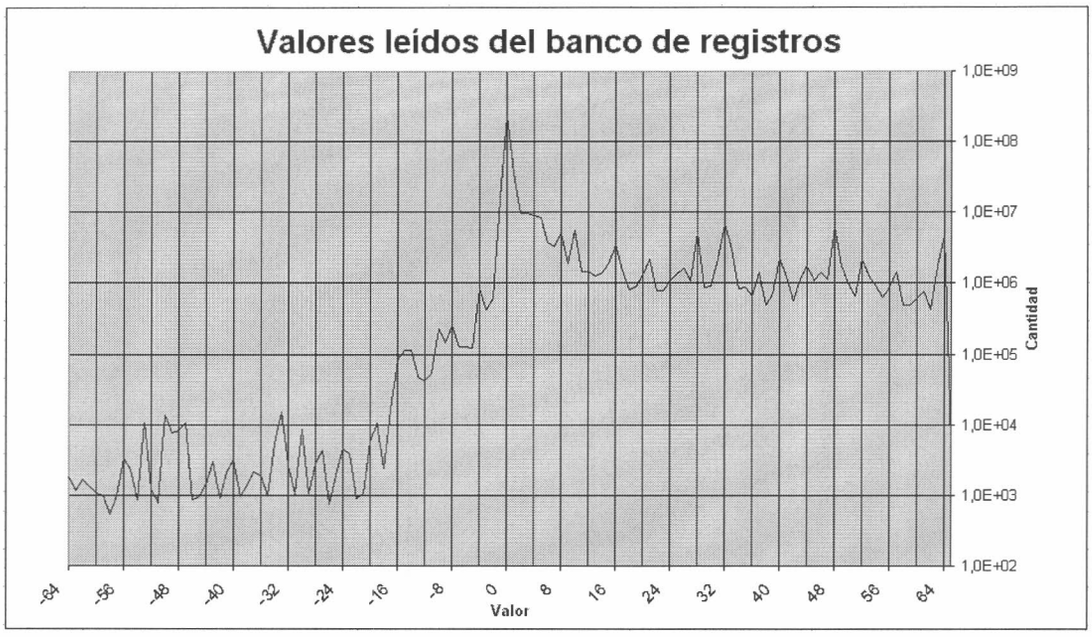
incrementar la eficiencia y disminuir la cantidad de instrucciones insertadas es el principal problema que se encontró en el desarrollo de filtros.

Es deseable tener filtros que permitan que una mayor cantidad de instrucciones sean insertadas en la cache, pero que a la vez mantengan la eficiencia del filtro [0,1]. Para desarrollar este filtro es necesario una mejor comprensión de cuales son los valores sobre los que las instrucciones operan la mayor parte de las veces. Una vez adquirido un mayor entendimiento sobre cuales valores son mas frecuentemente usados, se pueden desarrollar filtros que permitan mantener altos índices de inserción, de aciertos y de eficiencia.

5.4 Análisis de valores operados

Para hacer el análisis mencionado, se realizaron simulaciones para contabilizar que valores son leidos o escritos en el banco de registros. Esta simulación consiste en una modificación de un simulador funcional provisto por SimpleScalar en la que se alteró la rutina de acceso al banco de registros para que mantenga estadísticas sobre los valores escritos o leídos. Dado que es extremadamente difícil almacenar todos los valores referenciados, en estas simulaciones se limitó el rango de los valores observados a [-64,+64]. Como resultado de estas simulaciones se obtuvo que los valores positivos cercanos a cero son los mas frecuentemente usados. En las figuras 5.2 y 5.3 se pueden observar los resultados obtenidos. Es importante notar que la cantidad de ocurrencias de cada valor se presenta en escala logarítmica. Lo mas destacado en ambos gráficos es el pico en el valor 0, y la gran diferencia en cantidad de ocurrencias que existe entre los valores positivos y negativos. Asimismo, tambien es interesante la tendencia a decrecer que se observa para los valores positivos a medida que se van alejando del cero.

Figura 5.2: Valores leídos del banco de registros



5.5 Filtros desarrollados

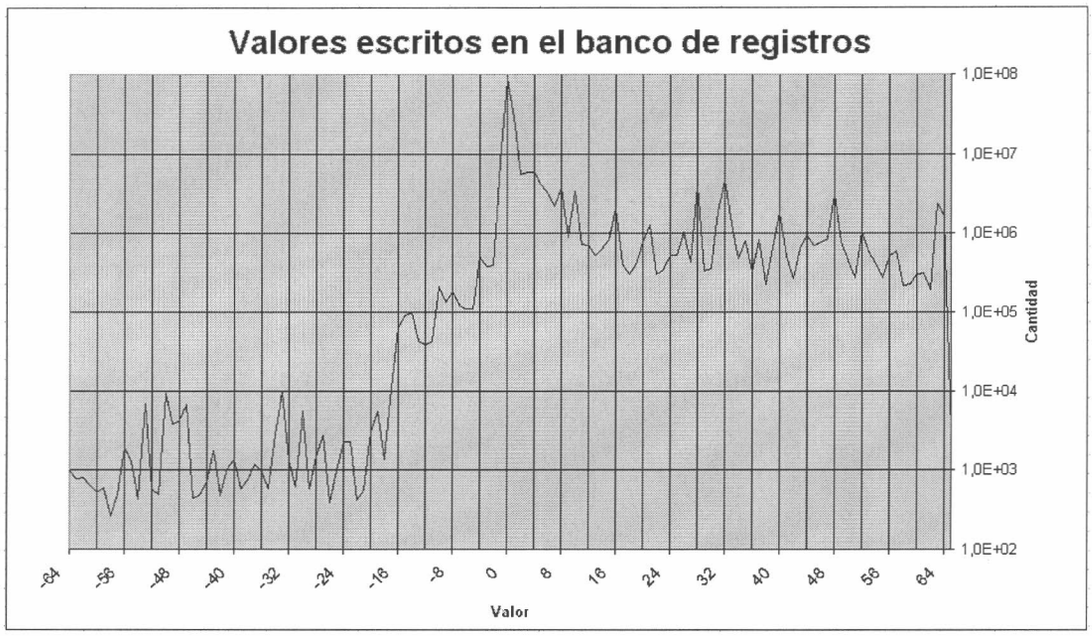
Con los resultados de las simulaciones anteriores se desarrolló el filtro [0,3]. La simplicidad para implementar en hardware este filtro, al igual que el [0,1], viene del hecho de que es muy sencillo de verificar si un valor va a pasarlo o no. Por ejemplo, para el filtro [0,3] lo único que debe verificarse es que todos los bits menos los dos menos significativos sean cero.

La tabla 5.2 muestra el efecto de aplicar el filtro al esquema original. Como resultado, en promedio 7% mas instrucciones que en caso del filtro [0,1] fueron insertadas, mientras que la eficiencia se mantuvo sin cambios significativos.

Dado el resultado obtenido al ampliar el filtro, se simuló el filtro [0,15], obteniendo los resultados que se muestran en la tabla 5.3.

En este caso se insertaron 23% instrucciones mas que en el caso del filtro [0,1], llegando a un 60% del total de las instrucciones insertadas en el esquema original. Por otro lado, la eficiencia del esquema se ubica en torno al 94%, lo cual es un valor bastante cercano al que se obtiene con los filtros anteriormente mencionados. De aquí en más, cuando se referiera a un esquema con filtrado, se lo hará al esquema con el filtro [0,15].

Figura 5.3: Valores escritos en el banco de registros



5.6 Conclusiones

En este capítulo se desarrolló un esquema de filtrado para evitar que instrucciones que tienen pocas probabilidades de ser reusadas sean insertadas en la cache, con el consiguiente desperdicio de espacio.

Para desarrollar el mecanismo de filtrado se realizaron pruebas para determinar cuáles son los valores mas utilizados en los programas, con el objetivo de que el filtro a desarrollar permita el paso de las instrucciones que los usen.

Los esquemas de reuso planteados hasta el momento en este trabajo no son útiles para una implementación real, ya que el requerimiento de que la cache sea totalmente asociativa implicaría un tiempo de ciclo demasiado largo. Para solucionar este problema, en las próximas secciones se discutirá un esquema de reuso de computación implementado con una cache de asociatividad limitada.

Tabla 5.2: Estadísticas del Filtro [0,3]

Benchmark	**Insertado** %	**Hit** %	**Eficiencia** %
applu	9.24%	8.58%	92.90%
cc1	23.20%	22.31%	96.16%
compress	34.40%	32.86%	95.52%
fpppp	1.17%	1.15%	99.03%
go	31.77%	31.75%	99.90%
hydro2d	22.17%	21.03%	94.84%
li	17.20%	16.24%	94.41%
m88ksim	29.22%	28.40%	97.18%
mgrid	13.50%	13.20%	97.77%
su2cor	24.53%	23.11%	94.22%
swim	8.78%	8.78%	99.98%
tomcat	21.50%	20.84%	96.96%
turb3d	27.66%	26.27%	94.97%
wave5	10.93%	9.87%	90.34%
MEDIA	19.66%	18.88%	96.01%

Tabla 5.3: Estadísticas del Filtro [0,15]

Benchmark	**Insertado** %	**Hit** %	**Eficiencia** %
applu	14.73%	13.51%	91.67%
cc1	27.28%	25.74%	94.37%
compress	38.18%	36.59%	95.83%
fpppp	1.52%	1.51%	99.01%
go	36.37%	36.32%	99.86%
hydro2d	25.21%	23.72%	94.07%
li	19.99%	18.04%	90.28%
m88ksim	32.48%	29.84%	91.87%
mgrid	16.31%	15.51%	95.13%
su2cor	29.16%	27.13%	93.00%
swim	8.80%	8.79%	99.93%
tomcat	24.22%	23.52%	97.10%
turb3d	34.77%	32.21%	92.63%
wave5	13.25%	10.25%	77.35%
MEDIA	23.01%	21.62%	93.72%

Capítulo 6

Implementación

6.1 Consideraciones iniciales

En este capítulo se discute una posible implementación hardware de reuso de computación.

A pesar de que los esquemas que se han planteado hasta este punto son relativamente sencillos de implementar sobre hardware real, el hecho de tener una cache totalmente asociativa de 4K entradas implica un tiempo de ciclo excesivamente largo, con lo cual se desvanecería toda posible ventaja del esquema de reuso. Para resolver este problema es necesario desarrollar un esquema de reuso de computación basado en una cache con asociatividad limitada, dado que a mayor nivel de asociatividad mas largo deberá ser el tiempo de ciclo. Hechas estas consideraciones, se trabajará en el desarrollo de un esquema de reuso con una cache de asociatividad 4. Este nivel de asociatividad se puede encontrar en muchas estructuras de datos implementadas en hardware en procesadores actuales, con lo cual se puede asegurar que no representa una limitación para el tiempo de ciclo. Como ejemplos podemos citar el cache de nivel 1 del procesador Intel i486 [12].

Al pasar de un esquema totalmente asociativo a uno parcialmente asociativo surge la necesidad de contar con una función de direccionamiento que determine en cuál de todos los conjuntos debe realizarse la búsqueda asociativa. Para las caches de memoria principal esto no es un problema pues siempre se utiliza alguna porción del contenido del registro PC para hacerlo. En este caso esto no es posible, ya que usar ese registro implicaría perder una de las ventajas mas importantes del reuso de computación frente al reuso de instrucciones, que es la no obligatoriedad de vaciar el buffer en cada cambio de contexto.

6.2 Función de indexado del buffer

El principal problema a solucionar cuando se limita la asociatividad de la cache es el desarrollo de la función de direccionamiento que determine el conjunto en el cual se debe realizar la búsqueda. En este caso el problema es mas complejo que en los casos tradicionales, ya que esta función debe construirse unicamente a partir del código de operación de la instrucción

y de los valores de sus operandos. Como se menciona arriba, no es posible utilizar el registro PC por la definición de reusabilidad usada en este trabajo. La construcción de esta función de direccionamiento de la cache puede pensarse como el desarrollo de una función de hashing, que debe asignar a cada posible valor de entrada una ubicación en la cache.

La función que se intenta desarrollar debe cumplir con dos objetivos. El primero de ellos es básicamente de correctitud. O sea, si al insertar el dato 'a' en la cache la función asigna la entrada k, al consultar sobre la existencia del dato 'a', la función debe devolver la entrada k como lugar a buscar.

El segundo objetivo es bastante mas ambicioso. Supongamos que se construye una función de indexado tal que a todo input le es asignada la entrada 0. Obviamente esta función es 'correcta', en el sentido definido anteriormente. Ahora bien, por supuesto que esta función no sirve absolutamente para nada, ya que se está *desperdiciando* lugar en la cache. Ninguna de las entradas (excepto una) es usada para nada, con lo que los recursos dedicados a la cache han sido malgastados por completo. Lo que es deseable, en otras palabras, es que la función de hashing a obtener distribuya los datos de la manera mas uniforme posible. Para obtener tal función de hashing es necesario conocer la distribución tanto de los códigos de operación como de los valores de los operandos.

6.3 Estudio de la distribución de los códigos de operación y de los valores de los operandos

Complementando el estudio realizado en el capítulo 4 en el que se determinó que los valores mas frecuentemente utilizados son los positivos cercanos a cero, se realizaron mas simulaciones para establecer la distribución de los códigos de operación. Es muy importante notar que este estudio es totalmente dependiente del procesador simulado, ya que el juego de instrucciones de cada arquitectura determina cuales serán las instrucciones mas utilizadas. Además, cada juego de instrucciones asigna a cada operación un código distinto. Un trabajo accesorio al presente podría consistir en el diseño de un juego de instrucciones optimizado para los esquemas de reuso aqui presentados.

Como resultado de las simulaciones se obtuvo que muy pocos códigos de operación son los mas usados.

Este resultado no favorece en lo absoluto el segundo objetivo de la función de hashing buscada, ya que teniendo en cuenta que también los valores de los operandos utilizados se ubican en torno al cero, ninguno de los inputs de la función posee una distribución uniforme. El trabajo a realizar consistirá entonces en buscar una función que combine estos dos parámetros de manera tal de obtener una distribución lo mas uniforme posible. Por otro lado, dado que se compararán varias funciones de direccionamiento, será necesario tener una medida de la bondad de dicha función, con lo cual se deberá establecer una función que permita comparar el grado de uniformidad de la distribución obtenida.

6.4 Desarrollo de la función de hashing

Tal como se puede suponer dado que la función debe ser implementada en hardware, las operaciones que compondrán la función serán funciones lógicas básicas, de forma tal que el cálculo sea lo mas rápido posible. Los componentes que se usarán son compuertas AND, OR y XOR. A continuación se explica de una manera intuitiva cual es el efecto deseado al aplicar cada una de las compuertas.

La compuerta AND se utilizará cuando se desee que haya mayor probabilidad de obtener un cero que un uno en la posición donde se la aplique. Dada la distribución de los inputs de la función, no se precisará demasiado.

La compuerta OR tiene como efecto dar una mayor probabilidad de obtener un uno que un cero en la posición donde se la aplique. La compuerta XOR es un punto intermedio entre las dos anteriores. Estas dos últimas compuertas serán el núcleo de la función.

Dada la gran cantidad de posibilidades de combinar las compuertas arriba mencionadas es necesario tener una medida de la uniformidad de la distribución obtenida. Por ello es que se estableció la siguiente función para medirla:

$$d = \frac{\sum_{i=1}^{n} |a_i - (\frac{\sum_{j=1}^{n} a_j}{n})|}{\sum_{j=1}^{n} a_j}$$

Siendo n la cantidad de conjuntos en el buffer y a_i la cantidad de accesos al $i - esimo$ conjunto.

Para llegar a esta función se tuvo en cuenta en un primer momento el cálculo de la distancia entre la distribución obtenida por la función a testear y una distribución uniforme. Para obtener una distribución uniforme equivalente en cuanto a total de accesos al buffer de reuso se recurrió al siguiente cálculo:

$$\frac{\sum_{j=1}^{n} a_j}{n}$$

De esta manera se calcula la cantidad de accesos que habría en cada posición del buffer si la distribución fuera absolutamente uniforme. Ahora, sumarizando el valor absoluto de la diferencia entre la cantidad de accesos realizados por la función a testear en cada posición del buffer y el obtenido por la función anterior se obtiene:

$$\sum_{i=1}^{n} |a_i - (\frac{\sum_{j=1}^{n} a_j}{n})|$$

Esta fórmula calcula cuanto se aparta la función propuesta de una distribución uniforme en forma absoluta. El problema ahora reside, justamente, en que esta es una medida absoluta, dependiente de la cantidad total de accesos. Lo deseable es tener una medida que sea independiente de la cantidad total de accesos realizados en la simulación. Es por eso que en la fórmula final se divide la misma por la cantidad total de accesos.

Para verificar que esta función provee de una buena medida sobre la uniformidad de la función, notemos que en el caso que los a_i sean todos iguales, el dividendo de la función es

Tabla 6.1: Funciones de direccionamiento desarrolladas

Nombre	Definición
cr3	((OpA XOR OpB) AND 0x3F8)<< 3 + (OpCode XOR OpA XOR OpB) AND 0x7
cr4	(((OpA OR OpB) AND 0x1c)<<5) + ((OpB AND 0x3)<<5) + ((OpA AND 0x3)<<3) + (OpCode OR OpA OR OpB) AND 0x7
cr6	(((OpCode XOR+OpA XOR OpB) AND 0x7)<<7) + ((OpA OR OpB) AND 0x7f)
cr8	((OpA XOR OpB) AND 0x3F8)<<3 + (OpA XOR OpB) AND 0x7

0. Luego, a medida que los a_i se apartan del promedio de accesos, la función crece. A modo de ejemplo, en el caso de tener n accesos y m entradas en la cache y que todos los accesos se realicen en una sola entrada, la fórmula anterior se transforma en:

$$\frac{n-\frac{n}{m}+\frac{(m-1)n}{m}}{n}$$

Si simplificamos esta expresión, se puede observar que el máximo de la misma se acerca a 2. Además de comparar las distintas funciones de hashing entre sí, se compararon contra los valores obtenidos para la función de direccionamiento del esquema de reuso de instrucciones Sv [27]. Ese esquema está basado en un buffer de reuso indexado por el registro PC. Este buffer contiene los valores de ambos operandos y el resultado de la operación. El test de reuso consiste en chequear el tag para verificar que se trate de la misma instrucción estática y luego en comparar los valores almacenados de los operandos contra los actuales. A simple vista, este esquema es muy similar al de este trabajo, la diferencia radica en la necesidad de verificar el tag en el de reuso de instrucciones o el código de operación en el de reuso de computación. Una modificación que se realizó sobre el esquema original de [27] es que en ese trabajo el esquema se aplica a todo tipo de instrucciones con excepcion de las de coma flotante. En la versión que se simula en este trabajo el esquema se aplica únicamente a las instrucciones de aritmética entera, de forma tal de hacer una comparación equitativa con el esquema de reuso de computación.

6.5 Resultados

La tabla 6.2 muestra el promedio de la función d sobre todos los benchmarks, para todas las funciones desarrolladas, tanto en el esquema con filtrado como sin filtrado.

La columna 'Hit' muestra el índice de hits obtenido; la columna 'd' muestra el valor de la función d según fue definida anteriormente. Por último, la columna 'Full Asoc' muestra la relación de hits entre el esquema con asociatividad limitada y el totalmente asociativo.

A partir de los resultados obtenidos se decidió usar la función cr3 para indexar la cache. Es importante notar que esta función, de acuerdo con la medida definida anteriormente, es aun mejor que la usada en el esquema de reuso de instrucciones en cuanto a uniformidad. De todas maneras, un mayor estudio sobre este punto debería contribuir a mejorar el índice de aciertos, para acercarse aun mas al conseguido por el esquema totalmente asociativo.

Tabla 6.2: Comparación de funciones de direccionamiento

	Sin filtrado			Con filtrado		
Función	**Hit %**	**d**	**Tot. Asoc. %**	**Hit %**	**d**	**Tot. Asoc. %**
cr3	27.40%	1.312485	84.12%	19.29%	1.556623	89.22%
cr4	24.29%	1.640304	74.57%	18.77%	1.718545	86.84%
cr6	25.75%	1.828952	79.06%	19.48%	1.607808	90.10%
cr8	25.89%	1.371022	79.49%	18.73%	1.568553	86.63%
IR-Sv	21.22%	1.454280	N/A	N/A	N/A	N/A

Capítulo 7

Mediciones de performance

Para medir el impacto del esquema de reuso de computación sobre la performance global en la performance del procesador, se realizaron simulaciones modificando un simulador de performance provisto por el conjunto de herramientas usado en el resto del trabajo. Este simulador permite una simulación detallada ciclo a ciclo de un procesador superescalar, configurando si el modo de operación es en orden o fuera de orden, la cantidad de unidades funcionales, la latencia de las mismas, la configuración de las caches, predictor de saltos, etc. La configuración utilizada para las simulaciones se detalla en la tabla 7.1.

Se realizaron simulanciones del procesador sin ningun esquema de reuso, incorporando los esquemas de reuso de computación con y sin filtrado e incorporando el esquema de reuso de instrucciones Sv, de forma tal de realizar una comparación. En este caso, a diferencia de las simulaciones anteriores, los benchmarks se ejecutaron durante 1000 millones de instrucciones. Los inputs utilizados fueron los mismos que en las simulaciones anteriores.

De la manera que en las simulaciones anteriores, la política de reemplazo del buffer es FIFO y el filtro utilizado es el [0-15]. La tabla 7.2 expone los resultados obtenidos en la simulación.

El resultado mas importante observable a simple vista es que el esquema de reuso de computación utilizado mejora efectivamente la performance en todos los benchmarks simulados. Aun mas, las ganancias obtenidas son mayores que las del esquema de reuso de instrucciones con el que se compara.

Además e los resultados obtenidos con el esquema de reuso de computación, con y sin filtrado, se muestran los resultados de las simulaciones utilizando el esquema Sv restringido mencionado anteriormente. Se puede apreciar que el esquema de reuso de computación obtiene mayores ganancias que el de instrucciones. Es importante notar que a pesar de que la diferencia no es muy importante en el caso de benchmarks monotarea, la misma debería ampliarse al considerar benchmarks que simulen un ambiente multitarea dadas las características distintivas del reuso de computación.

Tabla 7.1: Configuración del procesador simulado

Instruction Fetch	4 instructions per cycle
Instruction Cache	16K bytes, direct mapped, 32 byte line, 6 cycles miss latency
Branch Predictor	Bimodal, 2048 entries
Speculative execution mechanism	Out of order issue of 4 instructions per cycle, 32 entry reorder buffer , 32 entry load/store queue.
Architected registers	32 integer, hi, lo, 32 floating point, fcc
Functional units	8-integer ALUs, 2 load/store units, 4-FP ALUs, 1 integer multiplier/divider, 1 FP multiplier/divider
Functional unit latency (total/issue)	integer ALU-1/1, load/store 1/1, integer multiplier/divider 3/1, integer divider 20/19, FP adder 2/1, FP multiplier 4/1, FP divider 12/12, FP sqrt 24/24
Data Cache	16K bytes, direct mapped, 32 byte line, 6 cycles miss latency

Tabla 7.2: Estadísticas de performance

Benchmark	**Original IPC**	**Con filtrado IPC**	**Sin filtrado IPC**	**Esquema Sv IPC**
applu	1.8065	1.8487 (+2.33%)	1.8521 (+2.52%)	1.8384 (+1.76%)
cc1	1.0740	1.0970 (+2.14%)	1.1038 (+2.77%)	1.0953 (+1.98%)
compress	1.3419	1.3755 (+2.50%)	1.3866 (+3.33%)	1.3508 (+0.66%)
fpppp	0.8511	0.8513 (+0.02%)	0.8514 (+0.03%)	0.8514 (+0.03%)
go	2.6063	2.6657 (+2.27%)	2.7065 (+3.84%)	2.7065 (+3.84%)
hydro2d	1.3935	1.4002 (+0.48%)	1.4060 (+0.89%)	1.3978 (+0.30%)
li	1.8350	1.8595 (+1.33%)	1.9163 (+4.43%)	1.8631 (+1.53%)
m88ksim	1.9858	2.0406 (+2.75%)	2.0481 (+3.13%)	2.0175 (+1.59%)
mgrid	2.1861	2.1971 (+0.50%)	2.2072 (+0.96%)	2.1913 (+0.23%)
su2cor	1.2375	1.2616 (+1.94%)	1.2676 (+2.43%)	1.2562 (+1.51%)
swim	1.7047	1.7076 (+0.17%)	1.7109 (+0.36%)	1.7104 (+0.33%)
tomcat	1.1509	1.1653 (+1.25%)	1.1730 (+1.92%)	1.1666 (+1.36%)
turb3d	2.1255	2.1882 (+2.94%)	2.2154 (+4.22%)	2.1612 (+1.67%)
wave5	1.7345	1.7533 (+1.08%)	1.7537 (+1.10%)	1.7536 (+1.10%)
MEDIA	1.6452	1.6722 (+1.64%)	1.6856 (+2.45%)	1.6685 (+1.42%)

Capítulo 8

Conclusiones

En este trabajo se presentó una alternativa al reuso de instrucciones llamada reuso de computación. Se cuantificó el fenómeno de reuso de computación en instrucciones de aritmética entera y se encontró en las simulaciones realizadas que utilizando una cache totalmente asociativa de 64K entradas, mas del 92% de las instrucciones ejecutadas pueden ser reusadas. Para caches con una menor cantidad de entradas los resultados son tambien muy positivos, ya que para el caso de una cache con 4K entradas se alcanzan valores de reuso del orden del 86% en promedio, e incluso para el caso de una cache de tan solo 1K entradas, más del 75% de las instrucciones son reusadas.

El siguiente paso consistió en la evaluación del impacto del uso de un filtro de de instrucciones. Este es un mecanismo diseñado para permitir que la gran mayoría de las instrucciones puedan acceder a la cache, pero a la vez haciendo un uso mas eficiente del espacio disponible rechazando la inserción de instrucciones que tienen una baja probabilidad de ser reusadas. Al utilizar el filtro propuesto, se consiguen índices de eficiencia del 94%, manteniendo mas del 66% de los hits que en el esquema sin filtrado.

Todas estas simulaciones se realizaron utilizando una cache totalmente asociativa. Como la implementación real de ese esquema es demasiado costosa en términos de espacio y tiempo de ciclo, se analizó cómo implementar un esquema de reuso de computación con una cache asociativa de 4 vias. En la búsqueda de esa implementación se encontró que el punto crítico a resolver es el de la función de direccionamiento a usar, ya que las entradas que se utilizan para dicha función tienen una distribución estadística muy alejada de la uniforme.

El último paso consistió en realizar simulaciones que mostraron la mejora de performace que se puede obtener al utilizar reuso de computación sobre instrucciones de aritmética entera, a comparación de un esquema de reuso de instrucciones. Como resultado mas destacado, se observó que el esquema de reuso de computación propuesto obtiene una performance levemente superior al esquema de reuso de instrucciones. Este punto merece especial atención.

Existen dos temas que fueron dejados sin tratar en este trabajo. Por un lado, las instrucciones de salto, acceso a memoria y punto flotante no fueron incluidas para el esquema de reuso propuesto. El estudio del fenómeno de reuso de computación sobre estos grupos de instrucciones merece un trabajo mas extenso. Por otro lado, se mencionó que una de las

ventajas mas importantes del reuso de computación sobre el de instrucciones es que el primero es útil en un ambiente multitarea. Esta afirmación, aunque conceptualmente verdadera, necesita ser estudiada de una manera mucho mas profunda, y un estudio cuantitativo de esta característica sería muy importante para establecer una diferencia aun mas clara entre el reuso de computación y el reuso de instrucciones.

Bibliografía

[1] Burks, A. W., H. H. Goldstine y J. von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument", Informe al Ejrcito de los Estados Unidos, 1946.

[2] Borensztejn, P. y Bergotto, M. Transparencias del curso "Arquitectura de Procesadores" (http://www.dc.uba.ar/ap/clases). Facultad de Ciencias Exactas y Naturales, UBA, 1999.

[3] Henessy, J.L. y D.A. Patterson, "Computer Organization & Design. The Hardware/Software Interface", Morgan Kaufmann Publishers, Inc. San Francisco, California, 1999.

[4] Smith, J.E. "Characterizing computer performance with a single number", Comm. ACM 31:10 1202-06, 1988.

[5] Flemming, P.J. y J.J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results", Comm. ACM 29:3 218:21, 1986.

[6] SPEC "SPEC Benchmark suite release 1.0", Santa Clara, CA., 1989.

[7] Alpert D. y Avnon D. Architecture of the Pentium microprocessor. IEEE Micro, Junio 1993, 11-21.

[8] E.M. Riseman y C.C. Foster. The inhibition of potential parallelism by conditional jumps. IEEE Transactions on Computers, páginas 1405-1411, Diciembre 1972.

[9] D. Wall. Limits of Instruction-level parallelism. En Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, páginas 176-188, 1991.

[10] M. Lam y R. Wilson. Limits of control flow on parallelism. En Proceedings of the 19th International Symposium on Computer Architecture, páginas 46-57, 1992.

[11] Mirapuri S., Woodacre M. Vasseghi N. The MIPS R4000 processor. IEEE Micro, Abril 1992, 10-22.

[12] Crawford J. H. The i486 CPU: executing instructions in one clock cycle. IEEE Micro, Febrero 1990, 27-36

[13] T. Y. Yeh y Y. N. Patt. Two-level adaptive training branch prediction. En Proceedings of the 24th Annual Symposium on Microarchitecture , páginas 51-61. Noviembre 1991.

[14] J. Smith. A study of branch prediction strategies. Proceedings of the 8th Annual Symposium on Computer Architecture, May 1981.

[15] S. MacFarling. Combining Branch Predictors. Technical report TN-36, WRL, June 1993

[16] S. McFarling and J. Hennessy. Reducing the Cost of Branches, 13th International Symposium on Computer Architecture. June 1996.

[17] Hsu P. Y-T. Designing the FPT microprocessor. IEEE Micro, Abril 1994, 23-33.

[18] Slater M. AMD's K5 designed to outrun Pentium. Microprocessor Report, 8(14),1-11.

[19] R.M. Tomasulo An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development, 11(1):25-33, Enero 1967.

[20] Greenley D. et al. UltraSparc: the next generation superscalar 64-bit SPARC. Proc. COMPCON 1995, pp. 442-51.

[21] Colwell R. P., Steck R.L. A 0.6 micron BiCMOS processor with dynamic execution. Proc. ISSCC95, pp. 176-177.

[22] M.H. Lipasti y J.P. Shen. "Approaching 10 IPC via superspeculation"Technical Report CMU-MIG-1, Carnegie Mellon University, 1997.

[23] F. Gabbay y A. Mendelson. Speculative Execution based on Value Prediction. Techinal Report EE Department TR 1080, Technion - Israel Institute of Technology, Noviembre 1996.

[24] F. Gabbay y A. Mendelson. Using Value Prediction to Increase the Power of Speculative Execution Hardware. ACM Transactions on Computer Systems (TOCS), Agosto 1998.

[25] M.H. Lipasti y J. P. Shen. Exceeding the Dataflow Limit Via Value Prediction. En Proceedings of 29th International Symposium on Microarchitecture, páginas 226-237, Diciembre 1996.

[26] Avinash Sodani, and Gurindar S. Sohi, Understanding the differences between value prediction and instruction reuse, University of Wisconsin–Madison, 1998.

[27] Avinash Sodani, and Gurindar S. Sohi, Dynamic Instruction Reuse, University of Wisconsin–Madison, June 1997.

[28] S. E. Richardson "Caching function results: Faster arithmetic by avoiding unnecesary computation". Technical report, Sun Microsystems Laboratories, 1992.

[29] Avinash Sodani, and Gurindar S. Sohi, An Empirical Analysis of instruction repetition, University of Wisconsin–Madison, 1998.

[30] M.H. Lipasti, C.B. Wilkerson y J.P. Shen. Value Locality and Load Value Prediction. En Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems, páginas 138-147, Septiembre 1996.

[31] Y. Sazeides y J.E. Smith. The Predictability of Data Values. En Proceedings of 30th Annual international Symposium on Microarchitecture (MICRO-30), páginas 248-258, Diciembre 1997.

[32] Y. Sazeides y J.E. Smith. Modeling Program Predictability. En Proceedings of the 25th Annual International Symposium on Computer Architecture, páginas 73-84. Junio-Julio 1998.

[33] D. Burger, T. M. Austin, and S. Bennet. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-96-1308 , University of Wisconsin–Madison, July 1996

[34] P. Bannon y J. Keller. "Internal architecture of Alpha 21164 microprocessor". COMPCON 95, 1995.