UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Unidad de instrucciones para la ejecución paralela de los saltos

González Colás, Antonio María

UNIVERSIDAD POLITECNICA DE CATALUÑA

FACULTAD DE INFORMATICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES

UNIDAD DE INSTRUCCIONES PARA LA EJECUCION PARALELA DE LOS SALTOS

TESIS DOCTORAL

AUTOR: *Antonio González Colás*

DIRECTOR: *José María LLabería Griñó*

BARCELONA, Abril 1989

UNIVERSIDAD POLITECNICA DE CATALUÑA

FACULTAD DE INFORMATICA

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES

UNIDAD DE INSTRUCCIONES PARA LA EJECUCION PARALELA DE LOS SALTOS

TESIS DOCTORAL

Presentada en la
UNIVERSIDAD POLITECNICA DE CATALUÑA
para la obtención del título de
DOCTOR EN INFORMATICA

AUTOR: *Antonio González Colás*

DIRECTOR: *José María LLabería Griñó*

BARCELONA, Abril 1989

A Inma

AGRADECIMIENTOS

Mi más sincero agradecimiento a todas aquellas personas que de alguna forma han colaborado en la realización de este trabajo.

En especial a José María, mi director, por su constante ayuda y sus inestimables sugerencias.

A Mateo por su continuo impulso y su colaboración en gran número de discusiones.

A Rafa por su ayuda en la realización VLSI de la unidad de instrucciones presentada en este trabajo.

Este trabajo ha sido subvencionado por la Comisión Asesora de Investigación Científica y Técnica (CAICYT PA85-0314)

Indice

Prólogo

La técnica denominada segmentación desempeña un papel importante en el diseño de procesadores de alta velocidad. Su uso permite la ejecución concurrente de varias instrucciones en una misma unidad funcional.

El rendimiento que aporta la segmentación puede verse reducido por las penalizaciones debidas a las dependencias existentes entre las instrucciones de un programa. Un caso particular de estas dependencias son las ocasionadas por las instrucciones de transferencia de control. Debido a la elevada frecuencia de este tipo de instrucciones, el impacto que pueden tener sobre el rendimiento de una unidad segmentada es considerable. En muchos de los procesadores actuales, las instrucciones de salto son una de las principales barreras que impiden conseguir la ejecución de una instrucción por ciclo.

En este trabajo se propone un mecanismo (ESCO) que, además de intentar eliminar las penalizaciones debidas a los saltos, permita ejecutar las instrucciones de salto en paralelo con el resto de intrucciones. De esta forma puede conseguirse que el coste de un salto sea nulo y por lo tanto llegar a ejecutar incluso más de una instrucción por ciclo.

La evaluación de mecanismos para la ejecución de saltos requiere la obtención de medidas sobre programas de prueba de tamaño considerable para poder obtener resultados significativos. El tiempo requerido para la obtención de medidas mediante simulación es elevado. Además esta simulación se efectúa sobre un número limitado de programas de prueba. En esta tesis se desarrollan una serie de modelos analíticos que permiten por una parte reducir significativamente el coste de evaluación del mecanismo ESCO y además obtener

medidas para un espectro de parámetros mayor que el correspondiente a los programas de prueba disponibles.

En primer lugar se describe desde el punto de vista funcional el mecanismo ESCO, para seguidamente desarrollar un modelo analítico que caracteriza su comportamiento.

Este modelo analítico es utilizado para obtener medidas de rendimiento para diferentes parámetros del mecanismo, lo que nos permitirá determinar la configuración más idonea.

Seguidamente se propone una implementación del mecanismo ESCO mediante una unidad de instrucciones que hace uso de una memoria cache convencional. La evaluación del sistema nos permitirá obtener el valor óptimo de diversos parámetros de memoria cache. Las medidas de rendimiento muestran una ganancia considerable (entre un 18 y un 38%) sobre el rendimiento de otros mecanismos frecuentemente utilizados en arquitecturas segmentadas.

Finalmente se presenta una implementación del mecanismo ESCO utilizando como memoria cache una memoria de instrucciones destino de salto. El comportamiento del sistema es caracterizado mediante un modelo analítico que utilizaremos para analizar de forma sencilla la influencia en el rendimiento del procesador de diversos parámetros del sistema. La implementación de ESCO puede llevarse a cabo mediante el uso de técnicas de prebúsqueda espacial o temporal. El coste de implementación de la prebúsqueda espacial es bastante menor que el de la temporal. Los modelos analíticos desarrollados nos permiten estimar que el rendimiento de ambas alternativas es bastante similar, por lo que la prebúsqueda espacial es el mejor compromiso entre rendimiento y coste.

CAPITULO 1

Introducción

En este capítulo se describe la técnica denominada segmentación aplicada al diseño de procesadores, analizando los diversos parámetros que tienen una influencia directa en el rendimiento obtenido. De todos estos parámetros, se hace un especial hincapié en el tratamiento de las instrucciones de salto, ya que éste va a ser el tema central del presente trabajo.

1.1. UNIDADES FUNCIONALES SEGMENTADAS

Uno de los objetivos fundamentales en el diseño de computadores es maximizar el rendimiento del sistema. Generalmente este rendimiento se mide en el número de operaciones procesadas por unidad de tiempo. En el desarrollo de sistemas de alta velocidad han contribuido una serie de técnicas conocidas con el nombre general de *concurrencia*. Bajo este término se designan a aquellas técnicas que permiten que en un instante determinado, un sistema pueda estar procesando simultáneamente más de una operación básica. Dentro de las técnicas de concurrencia existen dos diferentes filosofías de diseño, *paralelismo* y *segmentación*, también conocidas por *paralelismo espacial* y *paralelismo temporal*.

El paralelismo espacial consiste en la replicación de una determinada unidad funcional. La eficiencia de esta técnica reside en asignar a cada una de las unidades funcionales una operación, de forma que todas ellas se lleven a cabo de forma concurrente. El rendimiento obtenido es función del número de unidades funcionales que trabajan en paralelo.

La segmentación se basa en dividir una unidad funcional en una serie de subunidades que operan concurrentemente sobre diferentes operaciones. Cada una de estas subunidades se denomina *etapa*. Cuando una operación entra en una unidad funcional segmentada, "viaja" a través de ella utilizando diferentes etapas de las que consta. Dado que en cada instante de tiempo una operación no ocupa la totalidad de las etapas, ello permite que varias operaciones estén "viajando" a la vez por la unidad funcional. De esta forma el rendimiento de la unidad funcional no es función del tiempo de ejecución de una operación, sino del intérvalo de tiempo entre la entrada de dos operaciones consecutivas.

Las etapas por las que pasa una operación y el orden en que lo hace pueden representarse mediante una matriz de dos dimensiones, una de ellas correspondiente a las diferentes etapas de la segmentación y la otra al tiempo. A esta matriz se le denomina *tabla de reserva* [Kogg81].

Según su forma de operar, las unidades funcionales segmentadas pueden clasificarse en *configuradas estática o dinámicamente*. Las configuradas estáticamente son aquellas en que todas las operaciones utilizan la misma tabla de reserva. En caso contrario diremos que la unidad funcional está configurada dinámicamente.

Un tipo especial de segmentación estática es aquel en que cada operación pasa por todas y cada una de las etapas sin pasar dos veces por la misma. A este tipo de segmentación lo denominaremos *lineal*. En una unidad funcional con segmentación lineal el control del flujo de operaciones es muy sencillo ya que permite la iniciación de una nueva operación cada pulso de reloj.

Una área de aplicación de las técnicas de segmentación es el diseño de procesadores. La mayoría de los procesadores recientes catalogados como tipo RISC [CGLl88], [GiMi89], [Henn84], [Hopk85], [Patt85], utilizan una segmentación lineal.

1.2. DEPENDENCIAS Y CONFLICTOS

La aplicación de técnicas de concurrencia en la resolución de un problema debe hacerse de tal forma que se respeten las dependencias existentes entre operaciones. Las dependencias existentes entre un par de instrucciones *i*, *j* (asumiendo que *i* precede a *j*) pertenecen a una de las siguientes categorías [Kell75]:

a) Lectura después de escritura (LDE). La instrucción *j* lee un objeto (registro, posición de memoria, códigos de condición) que es modificado por la instrucción *i*. Si la lectura se realiza antes de la modificación, el valor obtenido será incorrecto.

b) Escritura después de lectura (EDL). La instruccion *i* lee un objeto que es modificado por *j*. En este caso, para que el dato leido sea correcto, la lectura debe realizarse antes de la escritura.

c) Escritura después de escritura (EDE). Ambas instrucciones modifican el mismo objeto. Para que el resultado sea correcto la modificación hecha por *j* debe realizarse con posterioridad a la realizada por *i*.

Un *conflicto* es cualquier circunstancia que impide que la unidad funcional procese instrucciones a la frecuencia máxima que es capaz. Un conflicto puede estar ocasionado por alguna de las siguientes dos causas. La primera es la existencia de una dependencia entre dos instrucciones. Nótese que una dependencia sólo podrá causar un conflicto cuando las correspondientes dos instrucciones estén lo suficientemente cercanas para que la unidad funcional pueda solapar su ejecución. La otra posible causa es que dos instrucciones requieran utilizar la misma etapa a la vez. A este tipo particular de conflictos los denominaremos *colisiones*.

1.3. RESOLUCION DE CONFLICTOS

La existencia de un conflicto implica una cierta penalización en la eficiencia de una unidad funcional. En este apartado presentamos una visión general de las técnicas utilizadas en el diseño de procesadores para disminuir o eliminar este efecto negativo.

Las técnicas que se utilizan para disminuir o eliminar el efecto negativo de los conflictos podemos dividirlas en *estáticas y dinámicas. Técnicas estáticas o software* son aquellas que se aplican en tiempo de compilación. La idea básica de estas técnicas consiste en detectar los posibles conflictos que se puedan producir y mediante una reordenación del código intentar eliminarlos o reducir el retardo que ocasionan. *Las técnicas dinámicas o hardware* se basan en añadir circuiteria adicional al procesador que detecte y resuelva los posibles conflictos que puedan producirse en tiempo de ejecución.

Un ejemplo de aplicación de técnicas software podemos encontrarlo en el procesador MIPS (Microprocessor without Interlocked Pipe Stages) [PGHJ84]. En esta máquina casi todos los conflictos se resuelven en tiempo de compilación. De esta forman el compilador es el responsable de generar secuencias de instrucciones que efectúen el cálculo deseado teniendo en cuenta la estructura de la segmentación del MIPS. Por lo tanto, el uso de estas técnicas implica hacer visible al compilador (o programador de lenguaje máquina) determinados aspectos de la implementación.

Como ejemplo citaremos la instrucción LOAD que accede a memoria para leer un dato y dejarlo en un registro. Debido a la estructura de esta máquina, la instrucción que sigue a un LOAD no puede utilizar como fuente el registro destino del LOAD (dependencia del tipo LDE). El compilador debe encargarse por lo tanto de buscar una instrucción que no utilice dicho registro para colocarla a continuación del LOAD, y en caso de que no encuentre ninguna que cumpla este requisito deberá poner una instrucción de no operación (NOP).

Un ejemplo de técnica dinámica es el tratamiento de las instrucciones LOAD en el IBM RT PC [IBMC86]. En este procesador puede ocurrir que hasta 2 de las instrucciones que siguen a un LOAD no puedan disponer del dato en cuestión. Para resolver este posible conflicto, existe un circuito que detecta cuando una

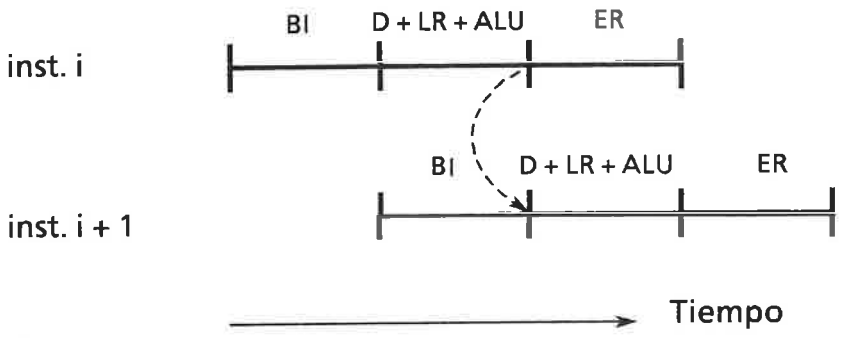
instrucción cuya ejecución va a iniciarse necesita acceder a un registro que es destino de un LOAD actualmente en progreso. Cuando ocurre este evento, el procesador deja de enviar nuevas instrucciones a la unidad funcional encargada de su ejecución, hasta que el acceso a memoria haya finalizado.

Cada uno de estos dos grupos de técnicas tiene sus ventajas e inconvenientes. Las técnicas software generalmente complican el diseño del compilador ya que hacen más compleja la visión que el programador a nivel de lenguaje máquina tiene de la arquitectura y por lo tanto hacen más costoso el proceso de generación de código. Esto no ocurre con las técnicas hardware, sin embargo, éstas complican el control del procesador pudiendo llegar a causar un incremento del tiempo de ciclo. Además, la implementación de estás técnicas requiere una determinada área física. Por lo tanto, su incorporación puede suponer una penalización para otras funciones que también debe realizar el procesador, bien porque se dispone de menos área para implementar éstas, o bien porque la implementación del conjunto excede alguna restricción de empaquetamiento (un chip, una placa, etc) que no se excedería sin la implementación de dicha técnica. En cualquier caso la elección entre unas y otras va a depender, además de los motivos que acabamos de citar, del rendimiento obtenido con cada mecanismo en cuestión. De hecho, la mayoría de procesadores utilizan tanto técnicas software como hardware, y en algunas ocasiones implementan mecanismos que se basan en la colaboración entre ambos tipos de técnicas.

En las máquinas tipo RISC generalmente la segmentación es lineal, existe una única etapa donde se leen los datos necesitados por la instrucción y otra única etapa (generalmente la última) donde se modifican todos los objetos necesarios. Debido a estas características, los únicos conflictos posibles son del tipo LDE.

Los conflictos del tipo LDE en instrucciones que operan sobre registros suelen resolverse mediante la implementación hardware de técnicas denominadas *cortocircuitos* que son un caso particular de una técnica más general denominada *forwarding* [Kogg81]. Un ejemplo podemos encontrarlo en el procesador RISC II [Kate85], cuya segmentación consta de 3 etapas (ver figura 1.1). Cuando la instrucción $i+1$ tenga como registro fuente el registro destino de la instrucción i (dependencia LDE), se producirá un conflicto ya que cuando $i+1$ va a leer dicho registro la instrucción anterior todavía no lo ha modificado. Para resolver este conflicto se ha "cortocircuitado" la salida de la ALU con su entrada, de forma que

un resultado obtenido en el ciclo x pueda utilizarse como operando fuente el ciclo $x+1$.



BI : Búsqueda de la instrucción
D + LR + ALU: Decodificación, lectura registros y operación de ALU
ER : Escritura del resultado

Figura 1.1: Cortocircuito entre la salida y la entrada de la ALU en el procesador RISC II.

En cuanto a las colisiones, una de las más frecuentes ocurre en los accesos a memoria. Para la ejecución de una instrucción se requieren realizar una serie de accesos a algún tipo de memoria. Estos accesos son necesarios para leer la instrucción, acceder a sus operandos y almacenar el resultado. Para evitar posibles colisiones, sería deseable que el procesador tuviera varios caminos de acceso a memoria, y en especial que tuviera caminos diferentes para datos e instrucciones.

Para ello, algunos procesadores como el Am29000 [Case87] disponen de una memoria de instrucciones y otra de datos a las que está conectado mediante 2 buses independientes (aunque tiene un único bus de direcciones, los posibles

conflictos sobre éste se reducen utilizando un protocolo modo ráfaga en los accesos al exterior).

En otras máquinas como el SPUR [Katz85] se propone la utilización de una memoria cache de instrucciones implementada en el mismo chip que el procesador y un solo camino de acceso al exterior, generalmente utilizado para acceder a los datos. Los conflictos en este bus ocurren únicamente cuando se produce un fallo en la memoria cache de instrucciones.

Una opción que se ha demostrado efectiva es destinar una parte del área del chip para almacén de datos. Ello nos permitirá un acceso más rápido a los datos, a la vez que reducirá el número de conflictos debidos a accesos al exterior. En muchas máquinas este almacén toma forma de registros, bién de uso general o de propósito específico. En otras, como es el caso del procesador CRISP [DiMB87] estos registros están organizados de una manera especial constituyendo lo que se ha dado en llamar una *cache de la pila*. Las ventajas e inconvenientes de cada alternativa pueden encontrarse en [DiMc82].

1.4. DESACOPLO DE LAS ETAPAS

El tipo de conexión entre etapas es otro parámetro que influye en el rendimiento de una unidad funcional segmentada. En este apartado se analizan diferentes alternativas para esta conexión. Finalmente se hace una reflexión de las consecuencias que la aplicación de estas alternativas tiene cuando la etapa en cuestión es la búsqueda de la instrucción.

La conexión más simple entre dos etapas p y q (asumiendo que p precede a q) consiste en un único registro donde la etapa p deja el resultado y por lo tanto, donde la etapa q va a buscar alguno de sus operandos fuente. En este caso ambas etapas están *rígidamente conectadas*. Este tipo de conexión, suponiendo que las etapas tienen la misma duración, permite que ambas etapas estén trabajando al 100% de sus posibilidades. Si estas operaciones son de duración variable ocurrirá que en determinados momentos la etapa q estará bloqueada a la espera de la finalización de la etapa p (cuando q sea más rápida que p), mientras que en otros momentos (cuando p sea más rápida que q) estará p bloqueada a la espera de que q consuma el dato del registro para poder enviarle el siguiente.

Una conexión mucho más flexible consiste en reemplazar el registro que conecta dos etapas por una cola FIFO, o cualquier otro tipo de memoria con capacidad de almacenar varios elementos. Esta cola *desacopla* la dos etapas de forma que cada una de ellas puede trabajar a un ritmo diferente aumentando así la utilización de cada etapa y por lo tanto el rendimiento de la unidad funcional.

Una de las etapas de un procesador segmentado donde más útil se ha demostrado la aplicación de esta técnica es en la búsqueda de instrucciones. La búsqueda de instrucciones es generalmente la primera etapa de todo procesador segmentado, y la encargada de suministrar trabajo al resto de etapas. Su duración es variable, ya que el tiempo de acceso a una instrucción depende del nivel de la jerarquía de memoria en que dicha instrucción se encuentre y de posibles colisiones que se puedan producir con otros accesos a memoria. Por otra parte, la velocidad de proceso del resto de las etapas del procesador también puede ser variable, ya que puede haber instrucciones que empleen determinadas etapas durante un tiempo mayor que otras instrucciones (por ejemplo, en el IBM RT PC, la instrucción de suma necesita la ALU durante un solo ciclo, mientras que la instrucción que realiza un paso de la división la ocupa durante tres), o simplemente debido a la posible existencia de determinados conflictos que en ciertos momentos pueden retardar el proceso de una instrucción y no hacerlo en otras ocasiones.

Está estrategia nos divide el procesador en dos grandes bloques que generalmente se conocen por el nombre de *unidad de instrucciones* y *unidad de ejecución*. La unidad de instrucciones es la encargada de realizar la búsqueda de éstas y suministrarlas a la unidad de ejecución para que las procese. Esta técnica ha sido utilizada, entre otros, en el procesador CRISP [DiMB87]. En este caso el almacén utilizado para desacoplar la búsqueda de instrucciones del resto de la segmentación es una memoria cache de instrucciones decodificadas. En algunos procesadores como el CRISP o el WISQ [PGHJ87] la unidad de instruciones se encarga además de realizar la tarea de secuenciamiento. De esta forma la unidad de ejecución debe encargase únicamente de procesar las instrucciones de manipulación de datos.

Cuando la búsqueda de instrucciones está desacoplada del resto de etapas del procesador, las instrucciones ya no son buscadas en el instante en que debe empezar su ejecución, sino que, en muchas ocasiones la unidad de instrucciones

irá a buscarlas con una determinada anticipación. Cuando ocurre esto diremos que el procesador utiliza técnicas de *prebúsqueda*. Uno de los principales objetivos de esta técnica es reducir el efecto de la latencia variable de memoria, permitiendo solapar un acceso lento (por ejemplo, cuando se produce un fallo de memoria cache) con la ejecución de varias instrucciones que han sido prebuscadas anteriormente y que por lo tanto se encuentran disponibles en el almacén que comunica ambas unidades. Un ejemplo de procesador que utiliza técnicas de prebúsqueda es el IBM 360/91 [Ibbe82].

1.5. CONFLICTOS DEBIDOS A LAS INSTRUCCIONES DE TRANSFERENCIA DE CONTROL

En este apartado nos centraremos sobre un tipo particular de conflictos, aquellos ocasionados por las instrucciones de transferencia de control. Para ello, previamente se realiza un análisis de las operaciones que requiere la ejecución de este tipo de instrucciones.

Con el nombre *Instrucciones de transferencia de control* designamos a las instrucciones de salto, llamadas y retornos de subrutina, aunque generalmente nos referiremos a este subconjunto de instrucciones como *saltos*, haciendo mención explícita cuando bajo este término no incluyamos a las instrucciones de llamada y retorno de subrutina. Estas instrucciones se utilizan para producir un cambio del flujo de instrucciones que alimenta a la unidad funcional segmentada. Para su ejecución el procesador debe realizar las siguientes operaciones:

- Cálculo de la dirección destino del salto.
- Evaluación de la condición que determina si el salto debe ser efectivo o no.

1.5.1. Cálculo de la dirección destino

La dirección destino de un salto está codificada en la propia instrucción según diferentes modos de direccionamiento. Es importante destacar que para la gran mayoría de saltos el compilador conoce el valor de esta dirección. Un motivo por el que las instrucciones de salto no llevan especificada explícitamente esta dirección es por razones de compactación de código y por evitar tener

instrucciones de longitud variable. Por ello, en muchos procesadores nos encontraremos que el método de direccionamiento más frecuentemente utilizado por los saltos es el *relativo al PC* (las siglas PC designan al contador de programa), que consiste en codificar en el salto un desplazamiento relativo a su posición.

En algunos casos mucho menos frecuentes, la dirección destino del salto es función del contenido de algún registro de propósito general. Generalmente estos saltos se utilizan para traducir sentencias de lenguaje de alto nivel similares al "case" de PASCAL, o bién para traducir llamadas a procedimiento en las que el nombre del procedimiento destino es variable, y por lo tanto cada vez que se ejecuta esta sentencia el procedimiento llamado puede ser diferente. A este tipo de saltos los denominaremos *saltos calculados*.

1.5.2. Evaluación de la condición

Las instrucciones de salto llevan codificada una condición que determina en que casos el salto debe ser efectivo, y por lo tanto, si las siguientes instrucciones a ejecutar deben ir a buscarse a partir de la dirección destino del salto o son las que siguen en secuencia a la propia instrucción. Un subconjunto particular son los saltos incondicionales, que pueden verse como saltos condicionales en los que la condición siempre evalúa a cierto. Según la forma de especificar la condición a evaluar, distinguiremos entre *arquitecturas con códigos de condición* y *arquitecturas sin códigos de condición*.

En las arquitecturas con códigos de condición, la condición es evaluada por una instrucción anterior al salto. Este resultado queda almacenado en algún lugar determinado de la máquina constituyendo lo que se ha dado en llamar *códigos de condición*. Estos bits son consultados durante la ejecución de la instrucción de salto para decidir si el salto debe ser efectivo.

En las arquitecturas sin códigos de condición, la propia instrucción de salto lleva especificada una determinada operación a realizar (generalmente comparar dos objetos) cuyo resultado determinará si el salto debe ser efectivo. A este tipo de instrucción se le denomina generalmente *comparación-y-salto*.

Un caso poco frecuente ocurre en el procesador Intel 80960 [InCo88], el cual dispone de códigos de condición y saltos que los utilizan y además, dispone de instrucciones de *comparación -y-salto*.

El uso o no uso de los códigos de condición ha sido siempre motivo de controversia. En [DeLe87] y [HJBG82] pueden encontrarse diferentes argumentos a favor y en contra de cada una de estas alternativas.

1.5.3. Conflictos

Una vez analizadas las operaciones que requiere la ejecución de un salto veamos los conflictos que pueden producirse durante su ejecución, siendo todos ellos del tipo LDE:

a) El salto debe evaluar una condición que puede depender del resultado de alguna instrucción precedente todavía no finalizada.

b) La instrucción de salto debe calcular la dirección de la siguiente instrucción a ejecutar. Esta dirección puede ser función de algún cálculo precedente.

c) Para iniciarse la ejecución de la instrucción que debe ejecutarse tras el salto, es preciso conocer su dirección. Esta dirección es calculada por la instrucción de salto y por lo tanto no estará disponible hasta que el salto haya pasado por una determinada etapa.

Si estos conflictos no son resueltos de forma que se aminore o elimine su efecto negativo, el coste en tiempo de ejecución de una instrucción de salto puede verse sustancialmente incrementado debido a los ciclos perdidos a la espera de resolver estos conflictos. Debido a la elevada frecuencia de utilización de estas instrucciones, la cual varía de una máquina a otra pero que en cualquier caso suele estar entre un 15 y un 30% del total de instruciones ejecutadas, la degradación que pueden producir en el rendimiento de un procesador es considerable.

1.6. REDUCCION DEL COSTE DE LOS SALTOS

Dado el impacto que las instrucciones de salto tienen sobre el rendimiento del procesador, un aspecto muy a tener en cuenta en su diseño será la implementación de algún mecanismo que permita que la ejecución de estas instrucciones no cause demasiadas penalizaciones en el rendimiento de la máquina.

A lo largo de la historia se han propuesto diversos mecanismos basados en técnicas estáticas, dinámicas o una combinación de ambas para resolver los conflictos provocados por los saltos [Lilj88], [McHe86]. En este apartado pretendemos dar una visión general de las principales características de estas técnicas. La mayoría de estos mecanismos están encaminados a reducir las penalizaciones ocasionadas por los conflictos catalogados como (a) y (c) en el apartado anterior, ya que los de tipo (b) sólo ocurren en los *saltos calculados*, que tal como se ha explicado anteriormente, son muy poco frecuentes.

Definimos como *coste de una instrucción de salto* el tiempo necesario para ejecutar la propia instrucción más el tiempo debido a los posibles conflictos ocasionados por ésta.

1.6.1. Salto retardado

Los conflictos que ocurren durante la ejecución de una instrucción de salto se traducen en un número determinado de ciclos tras la iniciciación de su ejecución en los que todavía no está determinada cual será la siguiente instrucción a ejecutar. El mecanismo denominado *salto retardado* consiste en redefinir la semántica de las instrucciones de salto.

Hasta ahora habíamos supuesto en todo momento que el efecto de un salto tenía lugar justo en el punto del código de programa donde aparecía. Un salto retardado de longitud n es una instrucción de salto cuyo efecto tiene lugar tras la ejecución de n instrucciones. Al retardar n instrucciones el efecto del salto, la dependencia entre el salto y la instrucción situada a continuación de él deja de existir. La instrucción que ahora depende del salto es aquella situada n posiciones más adelante pero esta distancia impide que dicha dependencia ocasione un

conflicto. Un ejemplo de utilización de este mecanismo podemos encontrarlo en el procesador RISC II [Kate85]. En la figura 1.2 se representa la ejecución de un salto retardado en esta máquina. En este caso el salto retardado es de longitud 1.

La longitud de un salto retardado vendrá determinada por la posición relativa de la etapa que evalúa la condición y la etapa que calcula la dirección destino. Cuanto más alejadas del inicio de la ejecución están estas etapas, mayor será la longitud del salto retardado.

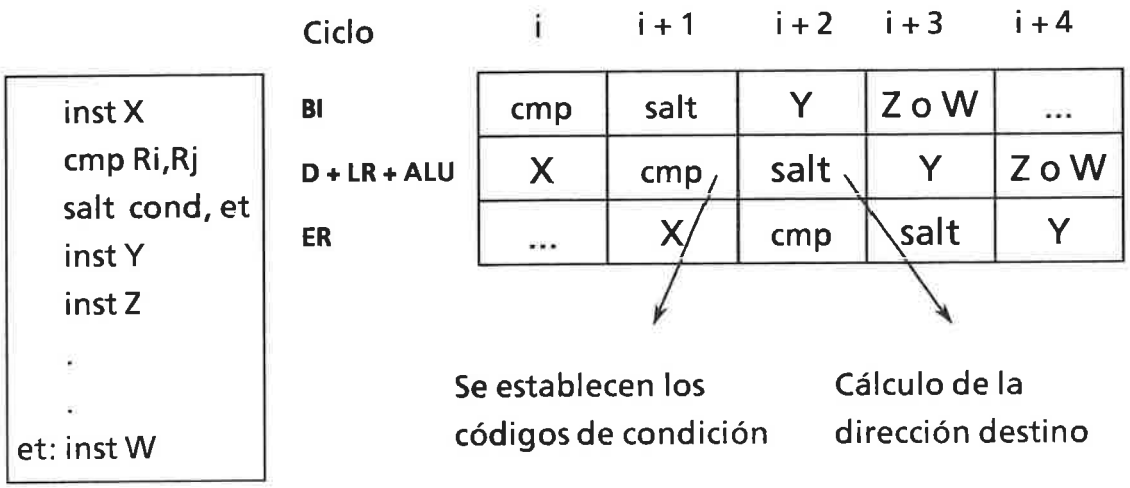


Figura 1.2: Ejecución de un salto retardado en el procesador RISC II.

El compilador (o el programador de lenguaje máquina) es el responsable de obtener provecho de este mecanismo ya que es el encargado de buscar las instrucciones que deben ponerse en las n posiciones tras cada salto. Una solución trivial es poner en esas posiciones instrucciones de NOP. Sin embargo podremos obtener más rendimiento del procesador si en esas posiciones colocamos instrucciones que realicen algún trabajo útil. Las tres opciones que el compilador tiene para rellenar estas posiciones se ilustran en la figura 1.3 (en este ejemplo se supone un salto retardado de longitud 1). Un análisis más detallado puede encontrarse en [GrHe82].

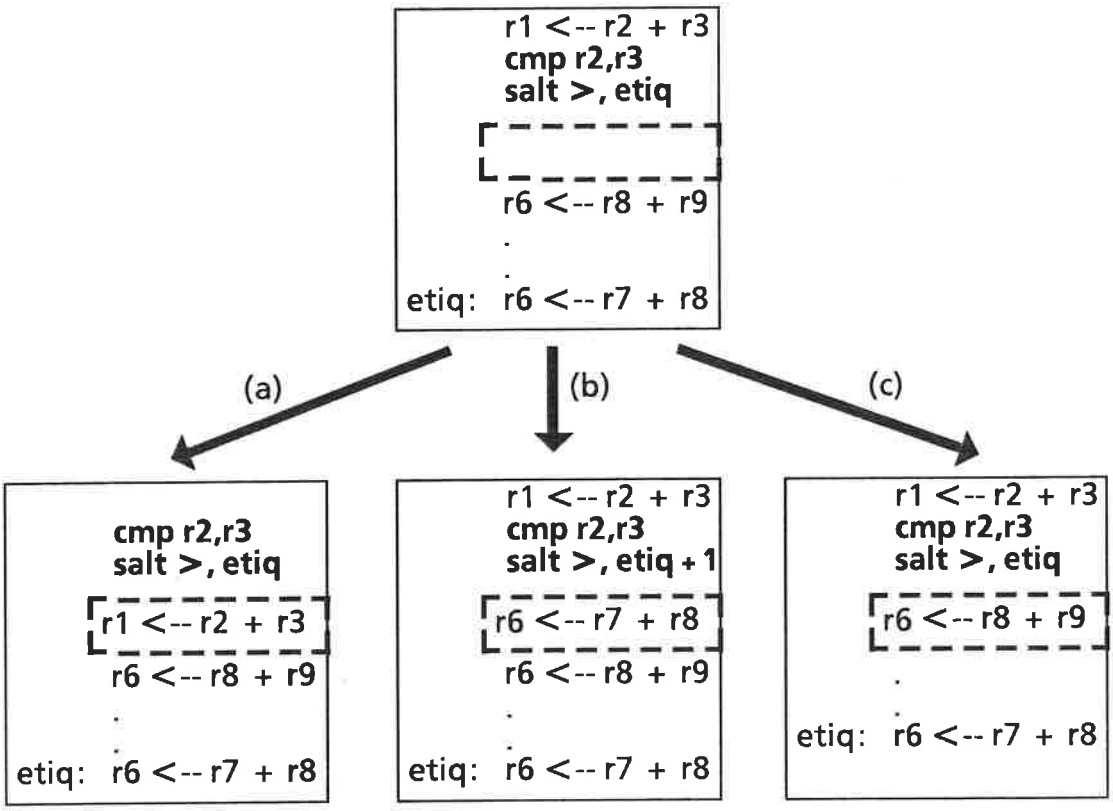


Figura 1.3: Optimización de los saltos retardados.

La opción (a) únicamente puede utilizarse cuando el salto no depende de las instrucciones ubicadas en las posiciones de retardo. Las opciones (b) y (c) sólo pueden utilizarse cuando la ejecución de las instrucciones situadas tras el salto no modifique el algoritmo tanto si el salto es efectivo como si no. Con la opción (b) sólo se reduce el coste del salto en los casos en que el salto es efectivo. En caso contrario, como estas instrucciones no realizan ningún trabajo útil, pueden contabilizarse como NOPs. En la opción (c), las instrucciones en las posiciones de retardo únicamente son útiles cuando el salto es no efectivo. Otra característica que puede apreciarse en la figura 1.3 es que la opción (b), en determinadas ocasiones aumenta el tamaño del código. La supresión de la instrucción en la

posición "etiq" va a depender de si dicha posición es alcanzable por otro camino diferente al salto analizado.

La proporción de posiciones de retardo que pueden ser optimizadas depende de las características del lenguaje máquina, del tipo de aplicaciones y del número de posiciones de retardo asociadas a cada salto. En [CoJo88] se presentan medidas de la frecuencia de utilización de las posiciones de retardo para el procesador RISC II con diferentes números de posiciones de retardo por salto. Al aumentar la longitud de los saltos retardados la eficiencia del mecanismo decrece rápidamente debido al número de posiciones de retardo que quedan sin optimizar.

Otros procesadores, además del RISC II, que utilizan este mecanismo son el Am29000 [John87], el MIPS [PGHJ84] y el Motorola 88000 [MoIn88].

Una aplicación algo peculiar de esta técnica se propone en el procesador PIPE [Good85]. Esta máquina utiliza saltos retardados, pero a diferencia de las anteriores, el número de posiciones de retardo asociadas a cada salto no es fijo, sino que es un número entre 0 y 7 especificado en la propia instrucción de salto. A esta instrucción los autores la denominan *prepare-to-branch*. Esta técnica tiene dos ventajas frente al salto retardado convencional.

Por una parte, en el salto retardado convencional, cuando el compilador no es capaz de encontrar suficientes instrucciones útiles para rellenar todas las posiciones de retardo, es necesario el uso de NOPs. Con este nuevo esquema basta ajustar el número de posiciones de retardo especificado en la instrucción de salto, con lo que se evita el aumento del tamaño del código ocasionado por las NOPs. Lógicamente, este esquema necesita de ayuda hardware para resolver aquellos conflictos que no haya sido capaz de eliminar por completo el compilador. Esta ayuda hardware consiste en detectar estos casos y retardar el envío de instrucciones a la unidad de ejecución hasta que se conozca el resultado del salto.

Por otra parte, cuando las dependendias entre el salto y el resto de instrucciones lo permitan, la separación entre un salto y el punto donde ocurre la transferencia de control, puede llegar a ser mayor incluso que la necesaria para evitar el conflicto. Ello puede ser útil para contrarrestar el efecto de la latencia de memoria externa cuando la instrucción a ejecutrar tras el salto no se encuentra en la memoria cache de instruciones interna. Tal como se comenta en el apartado 1.4,

la latencia de la memoria externa puede ser contrarrestada mediante el uso de técnicas de prebúsqueda.

1.6.2. Salto retardado con anulación

El hecho de que las instrucciones en las posiciones de retardo de un salto retardado se ejecuten siempre, independientemente de la dirección tomada por el salto, hace que en muchas ocasiones el compilador no encuentre ninguna instrucción candidata a ocupar estas posicones, y por lo tanto se ve obligado a poner NOPs en dichas posiciones. La posibilidad de anular la ejecución de las instrucciones en las posiciones de retardo en función de la dirección tomada por el salto, incrementa la proporción de estas posiciones que pueden ser optimizadas. La implementación de este mecanismo requiere que el procesador tenga la posibilidad de anular la ejecución de instrucciones ya iniciadas, ya que el resultado del salto se conoce cuando la ejecución de las instrucciones en las posiciones de retardo está ya en proceso.

Con este esquema, la instrucción de salto debe llevar codificados los casos en que debe producirse la anulación de las instrucciones que le siguen. Básicamente existen tres posibilidades:

a) No anular nunca. En este caso el comportamiento del salto es exactamente igual al del salto retardado.

b) Anular si el salto es no efectivo.

c) Anular si el salto es efectivo.

Para codificar estas tres posibilidades son precisos dos bits de la instrucción. Cuando varias de estas opciones sean posibles, el compilador eligirá aquella que proporcione una eficiencia mayor. Para la mayoría de los saltos, las opciones (b) y (c) son casi siempre posibles. Cuando esto ocurre, y suponiendo que la opción (a) no sea posible, la opción que producirá un rendimiento mayor depende del comportamiento del salto en cuestión.

Si el salto es más frecuentemente efectivo que no, en las posiciones de retardo pondremos la n primeras instrucciones del destino del salto, sumaremos n a la dirección destino y escogeremos la opción (b). De esta forma, las instrucciones en

las posiciones de retardo realizan trabajo útil siempre que el salto es efectivo, que es la mayoría de las veces. De forma similar, si el salto es más frecuentemente no efectivo, la opción que producirá un rendimiento mayor es la (c).

Por lo tanto, para poder elegir entre (b) y (c), es preciso que el compilador sea capaz de preveer cual será el comportamiento de cada salto durante la ejecución del programa que está traduciendo. Esta información puede provenir de la sentencia de alto nivel con la que se corresponde el salto en cuestión. Por ejemplo, un salto al final de un bucle que controla la finalización de este será efectivo la mayoría de las veces. Mucho más difícil es preveer el comportamiento de una estructura tipo IF-THEN-ELSE. Una posibilidad es obtener esta información mediante una *preejecución* del programa. Cuando la probabilidad de que un salto sea efectivo es igual a la probabilidad de que sea no efectivo, la opción (c) tiene la ventaja de que no incrementa el tamaño del código del programa, mientras que la opción (b), necesita en bastantes ocasiones duplicar las instrucciones en las posiciones de retardo por las mismas razones citadas en el apartado anterior para el caso del salto retardado. El mecanismo de salto retardado con anulación contemplando estas tres posibles opciones lo encontramos en el procesador SPUR [Hill86].

Dado que en muchos casos al compilador le es muy difícil o imposible detectar que saltos van a ser principalmente no efectivos, el incremento de rendimiento que nos aporta el disponer de la opción (c) es mínimo. Por lo tanto, algunos procesadores como el MIPS-X [ChHo87] hacen una simplificación del mecanismo de forma que sólo contemplan las opciones (a) y (b). Para ello, basta con destinar un bit de la instrucción para especificar el tipo de anulación deseada.

Un caso intermedio entre el SPUR y el MIPS-X es el procesador HP Precision [MBMH86]. En esta máquina se destina un único bit de la instrucción para codificar el tipo de anulación deseada. Si el bit vale 1 significa que la instrucción siguiente al salto no se debe anular nunca. En caso contrario la anulación va a depender, además del resultado de la condición, de la dirección destino. Para saltos hacia adelante (generalmente se corresponden con estructuras IF-THEN-ELSE) la instrucción se anula si el salto es efectivo (como la opción (c) anterior), mientras que para saltos hacia atrás (generalmente un salto hacia atrás cierra un bucle) la instrucción se anula si el salto es no efectivo (similar a la opción (b) anterior). De esta forma para los bucles se saca provecho del hecho de que estos saltos son efectivos con mayor frecuencia, mientras que en las estructuras IF-

THEN-ELSE, ya que a priori se desconoce la dirección que va a tomar el salto con más frecuencia, se reduce el tamaño del código del programa.

Una característica adicional del procesador HP Precision es que cualquier instrucción aritmética puede realizar un salto condicional de distancia 1. Para ello la instrucción lleva especificada una condición que es evaluada sobre el resultado. Si se cumple dicha condición, el procesador anula la ejecución de la instrucción siguiente. Existen además determinadas instrucciones aritméticas que en función de esta condición, en vez de anular la siguiente instrucción, tienen la posibilidad de generar un TRAP. Ello permite una implementación eficiente de operaciones tales como la comprobación de la salida de rango durante el acceso a una estructura de datos tipo vector.

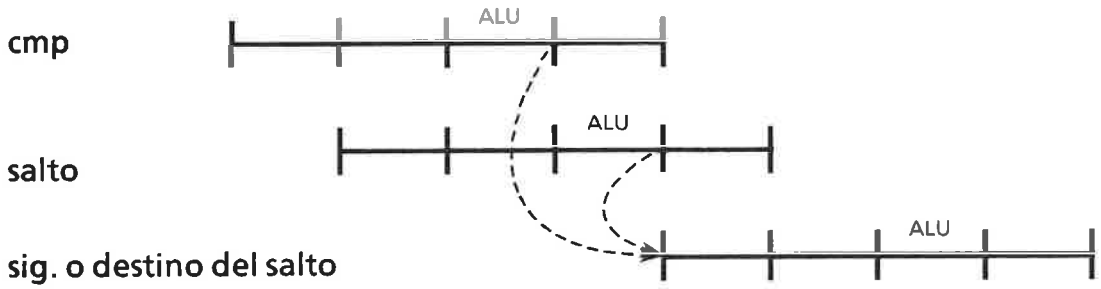
1.6.3. Anticipación del salto

Una forma de eliminar los conflictos debidos a las instrucciones de salto es retardar el punto donde un salto debe tener efecto. En este concepto se basa el salto retardado. Otra forma de eliminar, o reducir la penalización de estos conflictos es adelantar las operaciones que una instrucción de salto debe realizar. A esta técnica la denominaremos *anticipación del salto*. Como ya sabemos, las operaciones que un salto debe realizar son el cálculo de la dirección destino y la evaluación de la condición. El retardo producido por un salto vendrá determinado por aquella de estas dos operaciones que finalice en último lugar, y para reducirlo será preciso adelantar el resultado de esta última operación o el de ambas en caso de que ambas finalicen a la vez. Una vez más, para conseguir esta anticipación tenemos dos opciones, hacerlo mediante técnicas software o hardware.

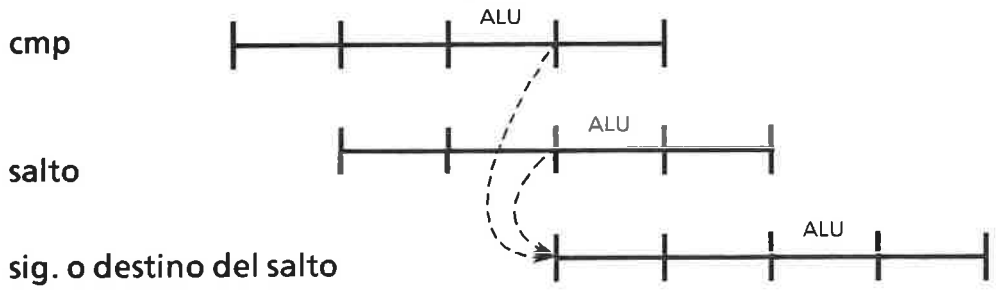
En general, las dos operaciones que un salto debe realizar precisan de la ALU para ser llevadas a cabo. Normalmente, en arquitecturas sin códigos de condición estas dos operaciones se realizan en paralelo, por lo que es preciso disponer de un sumador adicional, utilizado únicamente por las instrucciones de salto, durante la etapa de la ALU. Este es el caso del procesador MIPS-X [Chow86].

En arquitecturas con códigos de condición, la condición se evalúa por una instrucción previa al salto, durante la etapa de la ALU. El cálculo de la dirección destino lo realiza la instrucción de salto también en la ALU. Por lo tanto, en este caso, el retardo del salto vendrá determinado por la finalización de esta última

operación (ver figura 1.4.a). En estas circunstancias puede ser útil disponer de un sumador adicional que permita adelantar este cálculo, de forma que se realice en paralelo con la operación de la ALU de la instrucción anterior (en muchas ocasiones esta operación de la ALU será la evaluación de la condición) (ver figura 1.4.b).



a) Evaluación de la condición y cálculo de la dirección destino en la ALU



b) Evaluación de la condición en la ALU y cálculo de la dirección destino en un sumador adicional

Figura 1.4: Anticipación del cálculo de la dirección destino.

Una anticipación mayor no es siempre factible, ya que la evaluación de la condición suele depender de las instrucciones que la preceden, y por lo tanto al adelantarla entraría en conflicto con éstas. Sí que es posible en un gran número de casos anticipar todavía más el cálculo de la dirección destino, ya que la mayoría de los saltos son relativos al PC, y por lo tanto no existe ninguna dependencia con instrucciones anteriores. Ahora bién, de esta forma no disminuimos el retardo del salto ya que éste estará determinado por la evaluación de la condición. Sin embargo, ello nos puede permitir realizar una prebúsqueda de las instrucciones destino del salto y disminuir el efecto de la latencia de memoria.

Otra manera de adelantar el cálculo de la dirección destino es mediante el uso de una memoria donde se almacena la dirección destino de los últimos saltos ejecutados. Esta memoria puede ser accedida, mediante la dirección de la instrucción de salto, en paralelo con la búsqueda del salto. De esta forma, en caso de acierto, el procesador puede disponer de la dirección destino tan pronto como finaliza la búsqueda de un salto. A esta técnica se le denomina *memoria de direcciones destino de salto* ("branch target buffer"). Un procesador que implementa esta técnica es el MU-5 [MoIb79].

Una alternativa posible es utilizar esta memoria para almacenar las primeras instrucciones del destino del salto. Con ello no se adelanta realmente el cálculo de la dirección, pero se dispone de más tiempo para realizarlo, ya que puede llevarse a cabo en paralelo con la ejecución de estas instrucciones. Una ventaja adicional de esta estrategia es que reduce el efecto negativo de la latencia de memoria, al solapar el tiempo de acceso a ésta con la ejecución de las instrucciones obtenidas. En el procesador Am29000 [AdMD87] se implementa esta técnica únicamente con el objetivo de reducir el efecto de la latencia de memoria, ya que para la anticipación de la dirección destino se utiliza una técnica software que describiremos en este mismo apartado. Esta memoria recibe el nombre de *memoria de instrucciones destino de salto* ("branch target cache[1]" en el Am29000).

También es posible utilizar una combinación de ambas técnicas, es decir, una memoria que almacene la dirección destino del salto más las primeras instrucciones a partir de esta dirección. Si esto se utiliza en una máquina con códigos de condición, cuando se produce un acierto, a la instrucción de salto ya no le queda ninguna operación a realizar, y por lo tanto no tiene ningún sentido el

enviarla a la unidad de ejecución. De esta forma puede llegar a conseguirse en determinados casos, que los saltos se ejecuten con coste nulo. En [CoJo88] se describe una posible implementación de esta técnica.

Algunas técnicas software consisten en introducir determinadas características en la definición del lenguaje máquina, que permitan adelantar alguna de las operaciones del salto.

Un ejemplo lo encontramos en el procesador Am29000 [John87]. En este procesador las instrucciones de salto deben especificar la dirección destino utilizando el método de direccionamiento *registro-indirecto*. Este método de direccionamiento implica que la dirección destino se encuentra en un registro, en este caso de propósito general, y por lo tanto para su cálculo no es preciso realizar operación alguna, basta con leer dicho registro. De esta forma se adelanta un ciclo el cálculo de dicha dirección. Este registro debe ser inicializado en una instrucción precedente, ahora bién, para los saltos que están dentro de un bucle, que son la mayor parte de saltos ejecutados, esta operación puede ubicarse fuera del bucle y por lo tanto sólo habrá que realizarla una vez.

En el procesador PIPE [PlFa86] se propone el uso de esta misma técnica con la diferencia de que en este caso los registros que almacenan la dirección de salto son utilizados únicamente para este propósito.

Desde el punto de vista conceptual, el mecanismo utilizado en el Am29000 y el PIPE puede verse como un memoria de direcciones destino de salto ("branch target buffer") gestionada con instrucciones de lenguaje máquina, y por lo tanto controlada por el programador.

Para eliminar o reducir el efecto negativo de los conflictos debidos a la evaluación de la condición en arquitecturas con códigos de condición, el compilador podría intentar separar las instrucciones de comparar y saltar, para que cuando se inicie la ejecución de un salto, la comparación ya haya finalizado o esté lo más adelantada posible. La técnica para realizar este movimiento de código es muy similar a la presentada en el apartado dedicado al salto retardado. La principal diferencia reside en que en este caso, las posiciones de retardo no forman parte de la definición de la instrucción a nivel de lenguje máquina. Ello implica que el compilador no es el responsable de resolver este posible conflicto, aunque puede reducir o eliminar su efecto negativo. Por lo tanto, la arquitectura

debe disponer de un mecanismo hardware para detectar y resolver los conflictos que no hayan sido eliminados por el compilador. Esta técnica se utiliza en el procesador CRISP [DiMc87] donde se le ha dado el nombre de "*branch spreading*".

1.6.4. Predicción del salto

Denominamos anticipación del salto a aquellas técnicas que pretenden adelantar el cálculo del resultado de un salto. Otra forma de adelantar el resultado de un salto es predecir su resultado. A esta técnica la denominaremos *predicción del salto*. Lógicamente, para utilizar estás técnicas es preciso que el procesador pueda anular la ejecución ya iniciada de determinadas instrucciones en los casos en que la predicción haya sido errónea.

Dado que para la mayoría de los saltos es factible adelantar el cálculo de la dirección destino (frecuentemente no depende de instrucciones anteriores), las técnicas de predicción se utilizan para reducir o eliminar el retardo introducido por el cálculo de la condición de salto.

La predicción del salto puede hacerse de forma estática o dinámica. La predicción estática la realiza el compilador, y básicamente, su implementación consiste en destinar algunos bits de la instrucción para su codificación. Las técnicas utilizadas para rellenar estos bits son básicamente las mismas que en el caso del salto retardado con anulación, la diferencia es que aquí, el salto tiene efecto justo en el punto del programa donde aparece. La predicción dinámica se realiza cada vez que el salto debe ejecutarse. Su objetivo es predecir el resultado del salto, es decir, si será efectivo o no, a partir del comportamiento pasado de éste.

En [LeSm84] se proponen y evalúan diferentes métodos de predicción dinámica. La idea clave en la que se basan estos esquemas es en disponer de una tabla en la que se almacena la historia del salto. Ya que el coste de disponer de una entrada en la tabla por cada salto del programa sería disparatado, un solución generalmente adoptada es compartir cada entrada de la tabla entre diferentes saltos, por ejemplo entre aquellos saltos cuyos bits de menor peso de la dirección sean iguales. Mediante los bits almacenados en una entrada de la tabla se puede implementar un máquina de estados finitos. En la figura 1.5 se muestra una posible implementación utilizando 2 bits por cada entrada [LeSm84].

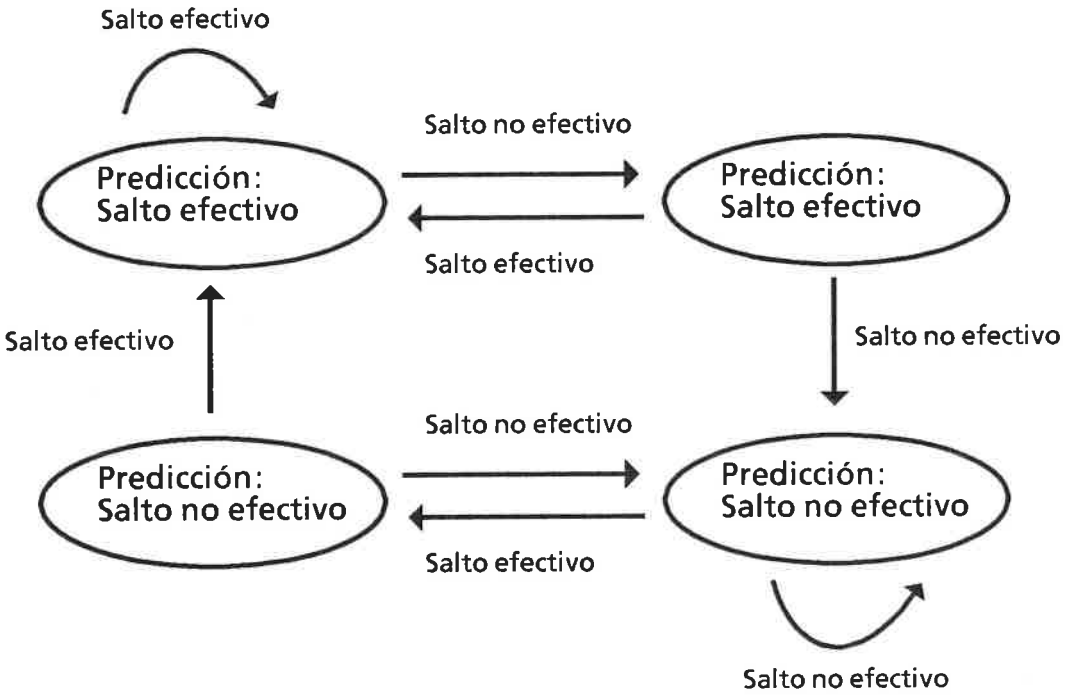


Figura 1.5: Predicción dinámica mediante 2 bits para codificar el pasado.

En [DiMc87] se presentan medidas acerca del rendimiento obtenido en el procesador CRISP mediante predicción estática y predicción dinámica. Los resultados muestran que el rendimiento de la predicción estática es casi tan bueno como el de la dinámica para 1, 2 y 3 bits de historia. Debido a la mayor complejidad del esquema dinámico, la elección adoptada para el CRISP fué implementar la estrategia estática. En la arquitectura del procesador Intel 80960 [Ryan88] se propone también el uso de técnicas de predicción estática.

Un caso intermedio es el IBM 360/91 [AnST87]. Este procesador utiliza predicción dinámica, pero esta predicción no se basa en el pasado del salto, sino en la dirección destino. Para los saltos hacia atrás a una distancia menor o igual a 8 instrucciones la predicción es de salto efectivo (entra en lo que denomina "modo

bucle"). En caso contrario la predicción es de salto no efectivo (entra en lo que denomina "modo condicional").

1.6.5. Salto pendiente de resolución ("Branch bypass") y prebúsqueda múltiple

Para eliminar la necesidad de predecir que camino es el que va a tomar un salto, el procesador podría ejecutar en paralelo los dos caminos, y descartar uno de ellos cuando conociera el resultado de la condición. Ello supone que determinado hardware del procesador debe de estar duplicado. Pero aún así no basta, ya que en estas ramas que se ejecutan de forma condicional puede a su vez aparecer algún salto lo que nos desdoblaría dicha rama en dos, y así sucesivamente.

Riseman y Foster [RiFo72] simularon el rendimiento de una máquina hipotética con capacidad infinita de almacenamiento e infinito número de unidades funcionales. El rendimiento de esta máquina está limitado únicamente por las dependencias existentes entre las instrucciones. Entre ellas están las dependencias debidas a los saltos, cuyo efecto puede ser eliminado ejecutando en paralelo ambas ramas de éste, lo que no presenta ningún problema en una máquina con infinitos recursos hardware. Los resultados obtenidos mostraron que el incremento en velocidad es proporcional a $\sqrt{j}$, siendo j el número máximo de saltos que pueden estar pendientes de resolución. Por ejemplo, un procesador que pudiera tener 16 saltos pendientes de resolución iría 2 veces más rápido que si sólo pudiera tener 4. Esta relación se cumplía hasta 32 saltos. Dado que el número de caminos cuya ejecución debe realizarse en paralelo es 2^j, el coste asociado a ésta técnica no justifica el redimiento que aporta.

Riseman y Foster querían analizar la posibilidad de incrementar sustancialmente el rendimiento de un procesador ampliando su hardware. Si únicamente pretendemos reducir el coste debido al retardo introducido por un salto, sin pretender ejecutar en paralelo todas aquellas intrucciones que podamos, como era el caso del experimento realizado por Riseman y Foster, las necesidades hardware van a ser mucho menores. Una aplicación de este mecanismo es la denominada técnica de *prebúsqueda múltiple*. La idea básica es que tras una instrucción de salto se realiza una prébusqueda de las primeras instrucciones de cada rama, de forma que cuando se conozca el resultado del salto, la etapa de búsqueda de la instrucción estará ya realizada, independientemente de cual sea el

resultado del salto. Para ello será preciso que el procesador tenga varios caminos de acceso a memoria de instrucciones, como es el caso del IBM 360 / 91, donde un camino es a memoria externa y el otro a un buffer de instrucciones existente en la unidad de instrucciones.

Otro ejemplo de prebúsqueda múltiple lo encontramos en la propuesta hecha por Katevenis en [Kate85] denominada *comparación-y-salto rápido ("*Fast compare-and-branch*")*. Dada la estructura de la segmentación del procesador RISC II, el retardo introducido por un salto es de una instrucción. Si en el ciclo siguiente a la búsqueda de una instrucción de comparar-y-saltar, es decir, mientras se realiza la comparación, la unidad de instrucciones pudiera ir a buscar tanto la instrucción siguiente al salto como la primera del destino del salto, el retardo del salto sería nulo. Para ello Katevenis propone utilizar una memoria cache de instrucciones con dos puertos de lectura.

Un problema adicional es que para acceder a la primera instrucción del destino del salto, es preciso haber calculado su dirección. Ello se resuelve mediante una codificación especial del modo de direccionamiento relativo al PC. La idea clave es que la instrucción de salto contiene los bits menos significativos de la dirección destino en vez del desplazamiento. Si estos bits son suficientes para especificar una dirección de la memoria cache, el acceso puede iniciarse en el ciclo siguiente al de la búsqueda del salto. Los bits más significativos pueden ser calculados en paralelo con el acceso a cache, ya que su resultado no se necesita hasta una vez finalizado el acceso, para comparar los tags y ver si hay acierto o fallo. Este podría ser un ejemplo de técnica de anticipación del salto (figura 1.6).

1.6.6. Ejecución paralela de los saltos

Las técnicas presentadas en la secciones precedentes intentan reducir el efecto negativo causado por las depencias de los saltos. Un aumento aún mayor de rendimiento se conseguiría si la ejecución de las instrucciones de salto se solapara completamente con el resto de instrucciones. De esta forma el coste, en cuanto a tiempo de ejecución, de una instrucción de salto podríamos considerar que es nulo.

Varias propuestas han sido hechas en este sentido [DiMc87], [CoLl87b], [PGHJ87]. La idea básica de todas ellas consiste en desacoplar la unidad de intrucciones de la unidad de ejecución, haciendo responsable a la primera del

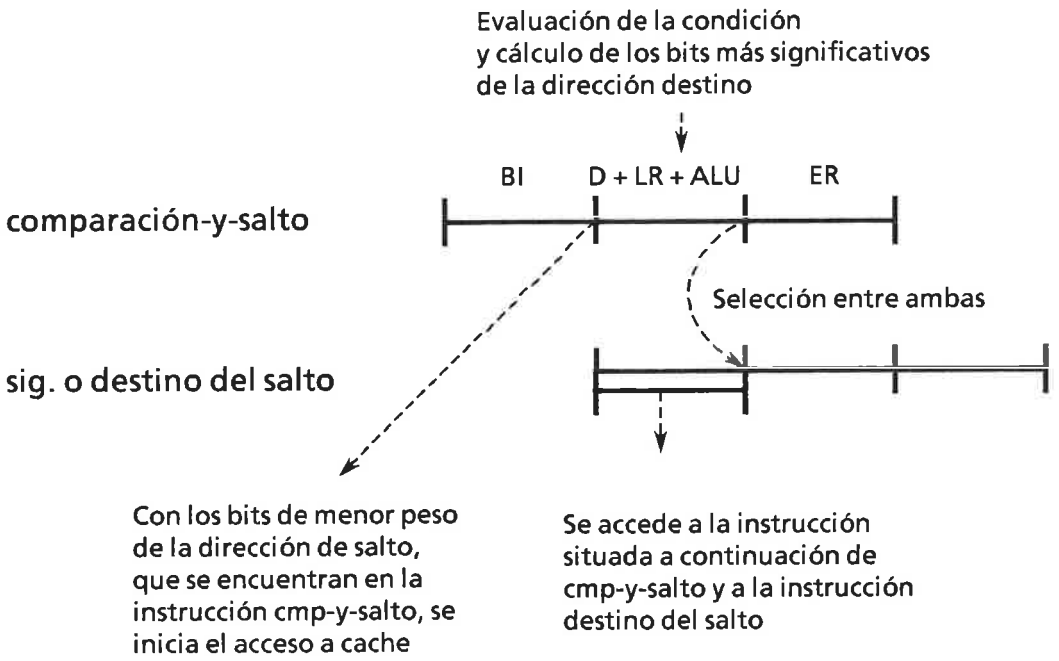


Figura 1.6: Ejecución de una instrucción comparación-y-salto mediante el esquema fast compare-and-branch.

tratamiento de las instrucciones de salto, y por lo tanto del secuenciamiento del programa. De esta forma la unidad de ejecución debe encargarse únicamente de las instrucciones de manipulación de datos, que son suministradas por la unidad de instrucciones. El rendimiento obtenido dependerá de la habilidad de esta última unidad para proveer de instrucciones a la unidad de ejecución.

El mecanismo implementado en el CRISP [DiMc87] se basa en la utilización de una memoria cache de instrucciones decodificadas que desacopla la unidad de instrucciones de la unidad de ejecución. La unidad de instrucciones, denominada unidad de prebúsqueda y decodificación por los autores, es la responsable de la búsqueda de instrucciones y de su decodificación. El resultado de la decodificación queda almacenado en la memoria cache de instrucciones decodificadas. Uno de los campos de la instrucción decodificada es la dirección de la siguiente instrucción a

ejecutar. De esta forma, al leer una instrucción de memoria cache, inmediatamente se dispone de la dirección para realizar el siguiente acceso. El hecho de que toda instrucción disponga de un campo con la dirección de la instrucción que le sigue significa que toda instrucción tiene la posibilidad de realizar un salto, y por lo tanto, no hay ninguna necesidad de disponer de instrucciones específicas para realizar bifurcaciones.

Durante la decodificación, la unidad de instrucciones detecta cuando una instrucción de manipulación de datos va seguida de una instrucción de salto y cuando esto ocurre, las junta en una única instrucción decodificada (a esta técnica los autores la denominan "branch folding").

Dado que un salto condicional tiene dos posibles instrucciones sucesoras, las instrucciones decodificadas tienen dos campos para almacenar ambas direcciones. Cuando un salto condicional es leido de la cache de instrucciones, se selecciona una de estas dos direcciones, y se procede a ejecutar las instrucciones del camino elegido, guardando la dirección alternativa por si la elección hecha ha sido errónea.

La elección entre los dos posibles caminos la realiza el compilador (predicción estática). Cuando el procesador conoce el resultado de la condición de salto analiza si el camino elegido es el correcto, y en caso contrario anula la ejecución de las instrucciones iniciadas tras el salto y procede a ejecutar la rama alternativa. La penalización de una predicción errónea depende de la distancia entre la instrucción de comparación y la instrucción de salto. Para disminuir esta penalización se utiliza la técnica de anticipación del salto anteriormente comentada denominada "branch spreading".

En resumen, para que el coste en tiempo de ejecución de un salto en el procesador CRISP sea nulo deben cumplirse las siguientes tres condiciones:

a) La instrucción de salto debe haber sido buscada con la antelación suficiente para poder ser fusionada con la instrucción que le precede.

b) La predicción realizada debe ser correcta o bién la separación entre la evaluación de la condición y el salto debe ser tal que haga innecesaria la predicción del resultado.

c) La siguiente instrucción a ejecutar debe encontrarse en la memoria cache de instrucciones decodificadas.

En [CoLl87b] se propone otro mecanismo que permite ejecutar los saltos con coste nulo. La idea básica se basa en la aplicación de las técnicas de prebúsqueda múltiple y en la anticipación del cáculo de la dirección de salto.

La anticipación del cálculo de la dirección de salto se realiza en tiempo de compilación mediante una reordenación del código, moviendo la instrucción de salto a la primera posición del bloque básico al que pertenece (un bloque básico es un conjunto de instrucciones consecutivas que siempre se ejecutan en secuencia). Al separar la instrucción de salto del punto donde el salto debe tener efecto, es preciso disponer de algún mecanismo para indicar donde debe realizarse la transferencia de control. Para ello se destina un bit en todas las instrucciones que indica si tras su ejecución debe tener efecto el salto anteriormente encontrado. Una alternativa posible sería destinar algunos bits de la instrucción de salto para indicar cuantas instrucciones más adelante debe tener efecto.

Al adelantar el cálculo de la dirección destino, el retardo del salto vendrá determinado por la evaluación de la condición. Para reducir este retardo se propone utilizar una memoria cache de instrucciones con dos puertos de acceso, de forma que mediante prebúsqueda por las dos ramas de un salto puede reducirse en un ciclo el retardo introducido por la evaluación de la condición (ver figura 1.7).

Al mover la instrucción de salto a la primera posición del bloque básico, dependiendo del número de instrucciones del bloque en cuestión, la anticipación conseguida puede ser mayor que la necesaria para eliminar el conflicto causado por el cálculo de la dirección destino. Esta mayor anticipación no disminuye el retardo del salto, ya que éste viene también determinado por la evaluación de la condición, pero permite reducir el efecto de la latencia de memoria externa en caso de producirse un fallo en la memoria cache de intrucciones. Dado que las operaciones que debe realizar una instrucción de salto son llevadas a cabo por la unidad de instrucciones, no es necesario que estas instrucciones sean enviadas a la unidad de ejecución.

En la arquitectura WISQ [PGHJ87] encontramos otro ejemplo de arquitectura desacoplada. En este caso el desacoplo se consigue mediante dos colas, una situada entre la unidad de instrucciones y la unidad de ejecución y otra,

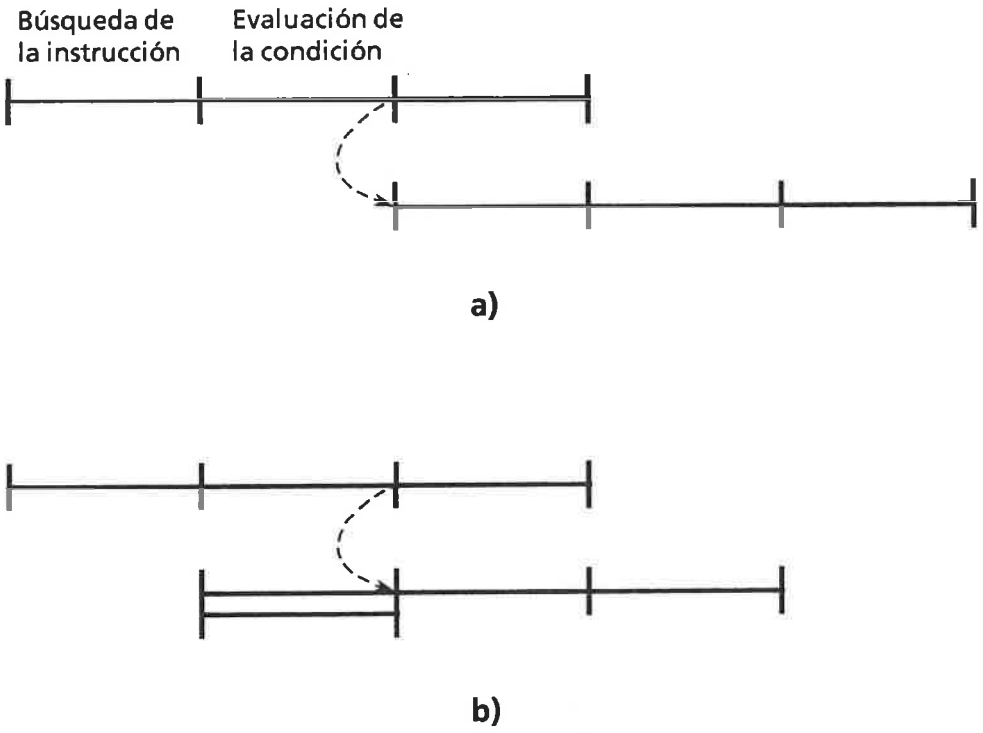


Figura 1.7: Retardo introducido por la evaluación de la condición.

a) Sin técnicas de prebúsqueda.

b) Mediante prebúsqueda por las dos ramas del salto

denominada "cola de reordenación", que desacopla la última etapa de la segmentación, consistente en la escritura del resultado en el registro destino. Para reducir el coste de los saltos en esta arquitectura se propone la utilización de las siguientes técnicas:

- Uso del salto retardado con retardo variable (misma técnica que en el PIPE denominan "prepare-to-branch").
- Anticipación de los saltos mediante el uso del modo de direccionamiento registro indirecto para especificar la dirección destino del salto.

- Ejecución de los salto en la unidad de instrucciones, en paralelo con la actividad realizada por la unidad de ejecución.

1.7. ORGANIZACION DEL PRESENTE TRABAJO

En este trabajo se propone un mecanismo para reducir las penalizaciones introducidas por los saltos en los procesadores segmentados. El objetivo del esquema propuesto es intentar reducir a cero el coste de las instrucciones de salto. Para que este coste sea nulo será preciso eliminar por completo los conflictos ocasionados por los saltos y además ejecutar los saltos en paralelo con el resto de instrucciones.

Para ejecutar el salto en paralelo será preciso la utilización de técnicas de anticipación que permitan calcular el resultado del salto antes de que éste tenga que ser efectivo. Esta anticipación se conseguirá mediante técnicas de prebúsqueda, cuya implementación requerirá el desacoplo entre la unidad de instrucciones y la unidad de ejecución. Para eliminar los conflictos debidos a los saltos utilizaremos técnicas de prebúsqueda multiple y en algunos casos, dependiendo del número de etapas de la segmentación, haremos uso del esquema denominado salto retardado o de técnicas de predicción.

Desgraciadamente, habrá casos en que la ejecución del salto no podrá llevarse a cabo con coste nulo debido a que no será posible su ejecución en paralelo o a que no se podrán eliminar por completo todos los conflictos que ocasiona. Estos casos serán más o menos frecuentes dependiendo principalmente del tipo de segmentación del procesador, de las carácterísticas de la aplicación ejecutada y del tipo de sistema de memoria. En este trabajo se analizan la influencia de estos parámetros y se muestra que para valores típicos el mecanismo propuesto permite ejecutar una gran parte de las instrucciones de salto con coste nulo.

Este trabajo está estructurado de la forma siguiente. En el capítulo 2 se describe desde un punto de vista funcional el mecanismo propuesto y se analizan las principales características arquitectónicas que requiere su implementación. Seguidamente se desarrolla un modelo analítico que caracteriza el comportamiento del esquema propuesto.

Este modelo es utilizado en el capítulo 3, donde se analizan varias alternativas de diseño de la unidad de instrucciones, evaluando el rendimineto obtenido con cada una de ellas y el coste de su implementación.

En el capítulo 4 se analiza la influencia de los parámetros del sistema de memoria en el rendimiento del procesador en general y en el del mecanismo ESCO en particular.

En el capítulo 5 se presenta el diseño elegido como resultado del análisis realizado en los anteriores capítulos. Este diseño utiliza un memoria cache de instrucciones convencional. El mecanismo es evaluado para diferentes parámetros del sistema de memoria y los resultados se comparan con otros mecanismos frecuentemente utilizados para reducir el coste de los saltos.

En el capítulo 6 se analiza el comportamiento del sistema en el caso de sustituir la memoria cache de instruciones convencional por una memoria de instrucciones destino de salto. Esta organización de la memoria de instruciones, junto con la utilización de un protocolo modo ráfaga para la comunicación con el nivel superior de la jerarquía de memoria, se ha demostrada efectiva en el procesador Am29000 [AdMD87]. En este capítulo proponemos un nuevo diseño de la unidad de instruciones que se adapte a este sistema de memoria. Para obtener los parámetros de diseño de la unidad de instrucciones, su comportamiento es evaluado mediante un modelo analítico. A fin de demostrar la efectividad de esta técnica, el rendimineto del sistema obtenido mediante simulación es comparado con el conseguido por el mecanismo de salto retardado. Finalmente se analizan dos tipos alternativos de prebúsqueda (espacial y temporal) evaluando el coste y rendimiento del mecanismo ESCO en cada caso.

NOTAS DE PIE DE PAGINA

(1) "Branch target cache" es una marca registrada de AMD

CAPITULO 2

ESCO: Ejecución de los saltos con coste cero

En este capítulo se presenta una descripción a nivel funcional del mecanismo propuesto para la ejecución de los saltos con coste nulo (ESCO) [GLlC88a], [GLlC88b]. En primer lugar nos centraremos en arquitecturas con códigos de condición y analizaremos el funcionamiento del mecanismo para un procesador segmentado con un número cualquiera de etapas. A continuación modelizaremos su comportamiento para un tipo determinado de segmentación. Finalmente comentaremos su aplicación a arquitecturas sin códigos de condición.

2.1. FUNDAMENTOS DEL MECANISMO

A lo largo de este apartado supondremos un procesador con códigos de condición segmentado en N etapas. El procesador tendrá por lo tanto instrucciones distintas para evaluar la condición y para calcular la dirección destino. A la instrucción que evalúa la condición y deja el resultado en los códigos de condición la denominaremos *comparación* (aunque en algunos casos esta instrucción no es propiamente una comparación sino que puede ser cualquier instrucción que modifique los códigos de condición para que más tarde sean utilizados por una instrucción de salto). En los ejemplos gráficos, la etapa en que se evalúa la condición la representaremos como se muestra en la figura 2.1.a. A la instrucción que calcula la dirección destino la denominaremos *salto*. Gráficamente la etapa

en la que se realiza dicho cálculo la representaremos como se indica en la figura 2.1.b.

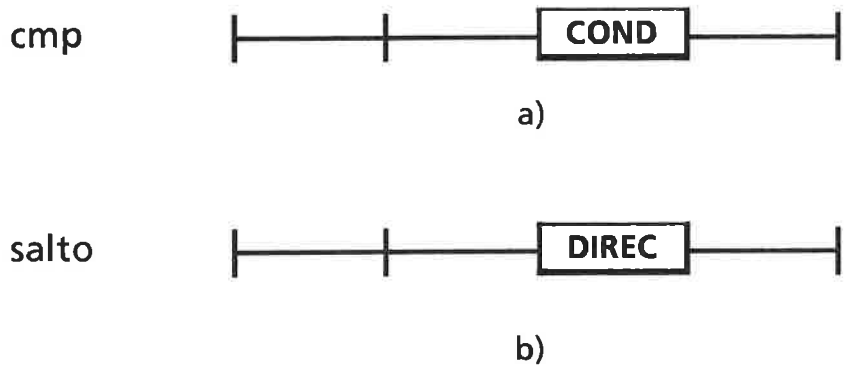


Figura 2.1: Representación gráfica de las instrucciones de comparación (a) y salto (b).

Supongamos un procesador segmentado en N etapas que dispone de una única ALU que opera en la etapa M ($1 \leq M \leq N$). La evaluación de la condición de salto y el cálculo de la dirección destino deberán realizarse por lo tanto en la etapa M de su correspondiente instrucción. El coste de una instrucción de salto será M ciclos si no se dispone de ningún mecanismo para reducir el efecto de los conflictos ocasionados (ver figura 2.2 donde $N=5$ y $M=4$).

En la figura 2.2 se pone de manifiesto que la instrucción que debe ejecutarse después del salto no puede iniciarse hasta que la instrucción de salto ha calculado la dirección destino. Si adelantamos este cálculo, el inicio de la ejecución de la instrucción siguiente al salto estará determinado por la finalización de la evaluación de la condición, y por lo tanto el coste del salto será M-1 ciclos (ver figura 2.3).

En el esquema de la figura 2.3, el cálculo de la dirección destino no necesariamente debe realizarse en la etapa M-1, podría llevarse a cabo en cualquier etapa anterior, aunque no por ello se disminuiría el coste del salto. En cualquier caso este cálculo no puede realizarse en la ALU, ya que en ese ciclo está

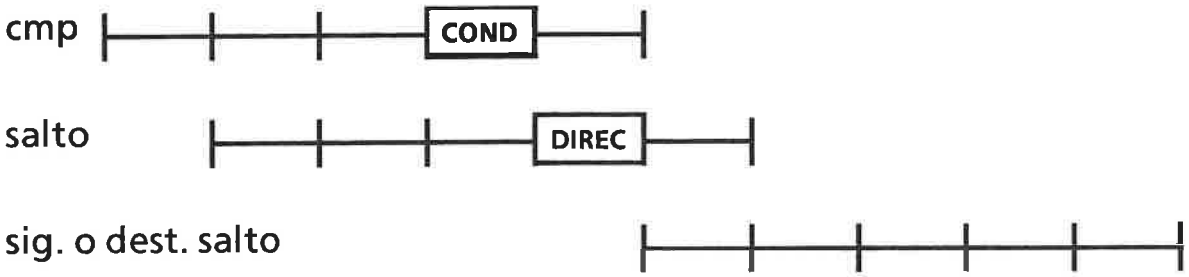


Figura 2.2: Coste de una instrucción de salto en el caso de no disponer de ningún mecanismo para reducir los conflictos ocasionados.

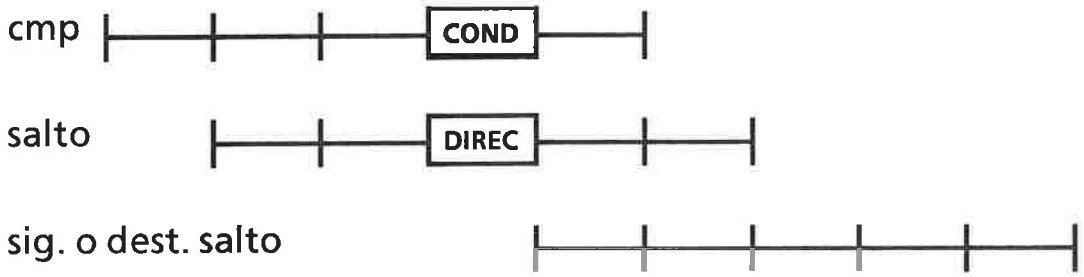


Figura 2.3: Coste de una instrucción de salto en caso de adelantar el cálculo de la dirección destino.

siendo utilizada por una instrucción precedente. Para realizarlo proponemos equipar al procesador con un sumador adicional únicamente utilizado para tal fin o utilizar la codificación propuesta por Katevenis para la codificación de la

dirección destino del salto [Kate85]. Podríamos utilizar cualquier otra técnica descrita en el capítulo 1 para la anticipación del cálculo de la dirección destino, pero creemos que el mejor compromiso entre rendimiento y coste es utilizar las aquí propuestas.

Para reducir todavía más el coste del salto podemos utilizar técnicas de prebúsqueda múltiple (ver figura 2.4). De esta forma, mientras se evalúa la condición se va a buscar tanto la instrucción destino como la siguiente al salto, y al final de esta etapa se escoge entre una de ellas. Estos accesos no tienen porque realizarse en este ciclo en concreto. Podrían llevarse a cabo en cualquier ciclo anterior a éste y en cualquier caso conseguiríamos ejecutar el salto con un coste de M-2 ciclos.

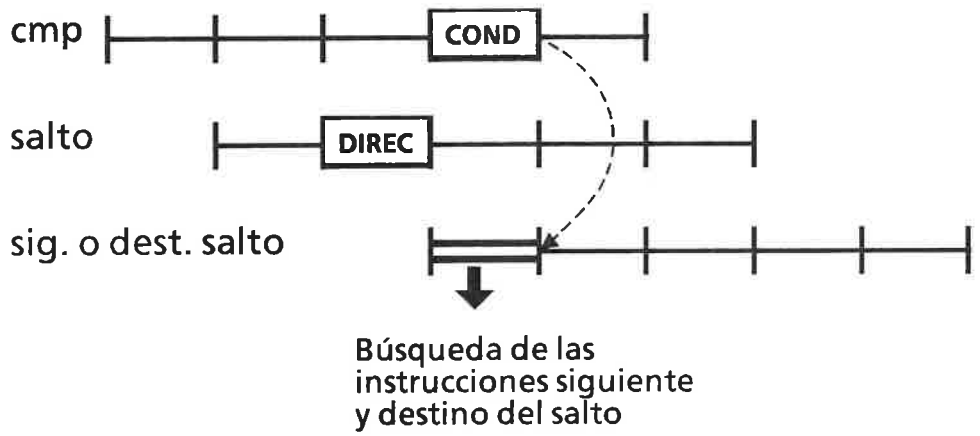


Figura 2.4: Coste de una instrucción de salto en caso de adelantar el cálculo de la dirección destino y realizar prebúsqueda por ambas ramas del salto.

El precio que hay que pagar para aplicar esta técnica es un aumento del ancho de banda de la memoria de instrucciones. Si inicialmente se necesitaba un ancho de banda de una instrucción por ciclo y si suponemos que de cada K instrucciones

ejecutadas una es un salto, será preciso un ancho de banda en media de 1+ 1 / K instrucciones por ciclo.

Podemos obtener un coste todavía menor si utilizamos además técnicas de predicción (estáticas o dinámicas) o saltos retardados. Ambas técnicas permiten que el procesador pueda estar realizando trabajo útil en los ciclos que transcurren desde la iniciación del salto hasta que puede iniciarse la ejecución de la instrucción determinada por el resultado del salto (ver figura 2.5).

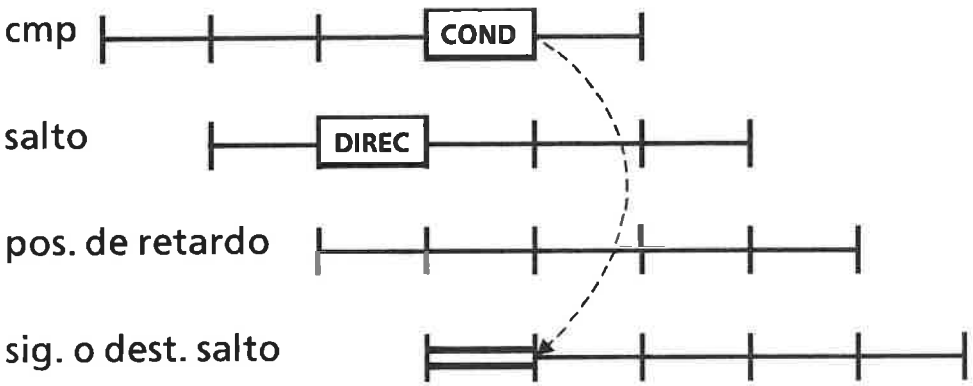


Figura 2.5: Coste de una instrucción de salto en caso de adelantar el cálculo de la dirección destino, realizar prebúsqueda por ambas ramas del salto y utilizar técnicas de predicción o saltos retardados para eliminar el conflicto introducido por el salto.

Con el esquema de la figura 2.5, el coste del salto va a depender de número de instrucciones situadas en las posiciones de retardo que realizan un trabajo útil. Si el número de posiciones de retardo es pequeño, puede conseguirse que una gran parte de los saltos se ejecuten con coste de 1 ciclo.

Finalmente, para conseguir que el coste de un salto sea cero, debemos de ejecutar la instrucción de salto en paralelo con el resto de instrucciones. Para ello es preciso disponer de la instrucción de salto con una determinada anticipación.

Esta anticipación puede conseguirse mediante el uso de técnicas de prebúsqueda, para lo que será preciso desacoplar la búsqueda de instrucciones del resto de la segmentación. Con las técnicas de prebúsqueda, la búsqueda de una instrucción puede realizarse varios ciclos antes de que empiece su ejecución. Para representar gráficamente este hecho utilizaremos la notación presentada en la figura 2.6. Cuando la primera etapa de la segmentación aparece en trazo grueso y discontinuo significa que la búsqueda de instrucciones está desacoplada y por lo tanto dicha operación puede haber ocurrido en el ciclo en cuestión o en cualquier otro ciclo anterior.

En la figura 2.7 se muestra la ejecución de una instrucción de salto utilizando el conjunto de técnicas que componen el mecanismo propuesto en este trabajo (ESCO). El cálculo de la dirección destino de salto se realiza en la UI en paralelo con la ejecución del resto de instrucciones. Para que no se produzca ninguna penalización, la dirección destino del salto debe conocerse como muy tarde el ciclo anterior al que se evalúa la condición.

Figura 2.6: Representación gráfica del desacoplo entre búsqueda de instrucciones y resto de la segmentación.

Con este esquema el procesador queda dividido en dos grandes bloques, la unidad de instrucciones (UI) y la unidad de ejecución (UE). La primera es la encargada de realizar el secuenciamiento del programa y de suministrar instrucciones a la unidad de ejecución. La UE se utiliza exclusivamente para la ejecución de instrucciones de manipulación de datos.

Uno de los objetivos de la UI es prebuscar instrucciones de forma que la dirección destino de los saltos pueda calcularse con algunos ciclos de anticipación. Tras ello, la UI prebusca instrucciones de las dos ramas por las que puede seguir

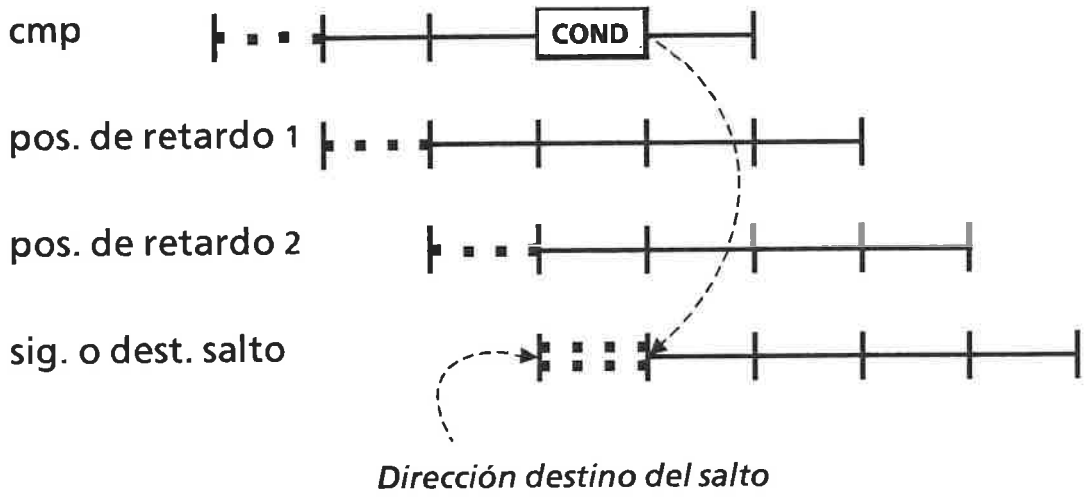


Figura 2.7: Ejecución de una instrucción de salto mediante el mecanismo ESCO.

el salto. Cuando finaliza la evaluación de la condición, la UI conoce cual de las dos ramas prebuscadas debe ser descartada y cual debe de alimentar a la UE a partir del próximo ciclo. Si M es la etapa de la segmentación que evalúa la condición, tras cada intrucción de salto existirán M-2 posiciones de retardo que deberán ser utilizadas mediante técnicas de salto retardado o bién mediante predicción del salto.

2.2. SEGMENTACION DEL PROCESADOR

Para continuar la presentación del mecanismo ESCO vamos a centrarnos en una determinada segmentación del procesador. En la figura 2.8 se muestran las etapas de esta segmentación, que coinciden con las del procesador SPUR [Katz85]. A diferencia de éste, nosotros suponemos una arquitectura con códigos de condición.

Con esta segmentación, el coste de un salto sería de dos ciclos si no se dispusiera de ningún mecanismo para reducir los conflictos. El número de posiciones de retardo al ejecutar los saltos mediante el mecanismo ESCO será cero

BI : Búsqueda de la instrucción
D + LR + ALU: Decodificación, lectura registros y operación de ALU
MEM: Acceso a memoria (utilizado unicamente por inst. LOAD y STORE)
ER : Escritura del resultado

Figura 2.8: Segmentación del procesador

(M-2=0). Para una segmentación diferente en la que fuera necesario el uso de posiciones de retardo, el rendimiento del mecanismo ESCO podría obtenerse fácilmente combinando los resultados obtenidos para cero posiciones de retardo con la probabilidad de optimizar las posiciones de retardo. En este trabajo se presentan los valores de estas probabilidades para una posición de retardo en varios programas de prueba.

2.3. DESCRIPCION FUNCIONAL

El mecanismo ESCO precisa de una memoria de instrucciones con un ancho de banda mayor que una instrucción por ciclo. En concreto, si de cada K instrucciones ejecutadas una es un salto, el ancho de banda mínimo será $1+2/K$. Lógicamente, el valor de K es variable durante la ejecución de un programa, siendo uno su valor mínimo. K igual a uno significa que tras un salto debemos ejecutar otro salto. La ejecución consecutiva de dos saltos ocurre con poca frecuencia, y además, gran parte de estas situaciones pueden ser eliminadas por el compilador en la fase de optimización. Como veremos más adelante, este mecanismo no es capaz de ejecutar dos saltos consecutivos de forma paralela y por lo tanto, cuando esto ocurra será necesario introducir una NOP entre ambos saltos. De esta forma, el ancho de banda máximo que debe soportar la memoria

será en el caso de que K valga dos. Este ancho de banda es igual a dos instrucciones por ciclo.

Para implementar este ancho de banda tenemos básicamente dos posibilidades. Utilizar una memoria de instrucciones con dos puertos de acceso independientes, cada uno de ellos con un ancho de banda de una instrucción por ciclo (figura 2.9.a) o bién disponer de un único puerto de acceso con un ancho de banda de dos instrucciones por ciclo (ver figura 2.9.b). En este último caso, si organizamos la memoria en bloques de dos instrucciones y restringimos el acceso a dos instrucciones que formen un bloque, el diseño de la memoria se simplifica considerablemente.

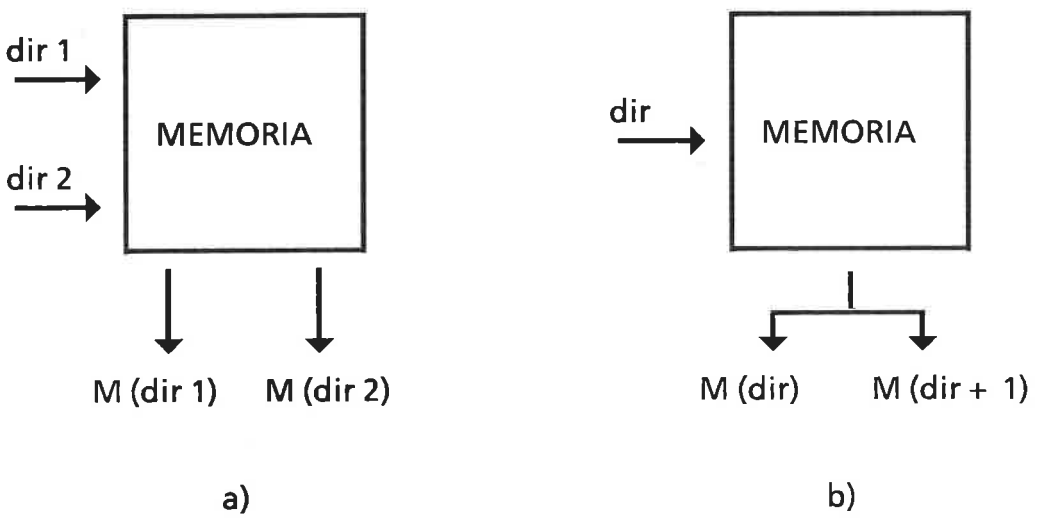


Figura 2.9: Organizaciones de memoria para disponer de un ancho de banda de dos instruciones por ciclo.

a) Dos puertos independientes con un ancho de banda de 1 instr./ciclo.

b) Un único puerto con un ancho de banda de 2 instr./ciclo.

En la figura 2.10 se representa la organización del sistema propuesto. La UI consta de varias colas donde se almacena información relativa a las instrucciones prebuscadas. En cada ciclo de procesador, la UI envia una petición a memoria para acceder a dos instrucciones y envía una de las que tiene ya prebuscadas a la UE. Mientras no se dispone del próximo salto a ejecutar estas dos intrucciones son consecutivas y pertenecientes al flujo de instrucciones que deberá alimentar a la UE en un futuro inmediato, antes de producirse el salto. Es decir, estamos haciendo prebúsqueda para localizar el próximo salto lo antes posible.

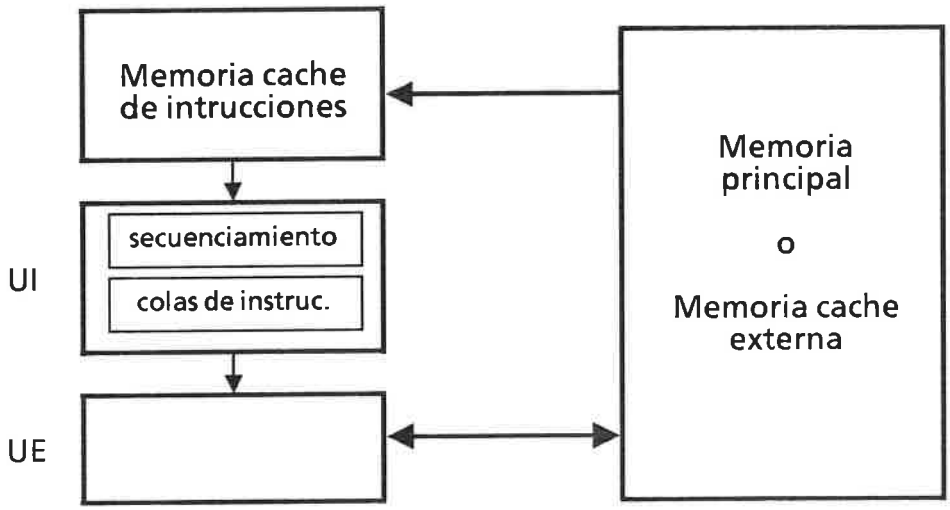


Figura 2.10: Organización del sistema

Cuando la UI encuentra una instrucción de salto, prebusca instrucciones de ambas ramas del salto. Para llevar a cabo esta prebúsqueda hay básicamente dos posibilidades. Realizarla en paralelo por las dos ramas a razón de una instrucción de cada rama por ciclo o bién prebuscar cada ciclo dos instrucciones consecutivas alternando de rama en cada ciclo. Si disponemos de una memoria de dos puertos ambas alternativas son posibles mientras que en el caso de disponer de una memoria de un puerto únicamente podemos utilizar la segunda.

Por lo tanto tenemos dos variantes del mecanismo propuesto. Denominaremos ESCO-1 al que utiliza una memoria de instrucciones de un puerto y ESCO-2 al que dispone de una memoria de dos puertos y en consecuencia, la prebúsqueda por ambas ramas se realiza de forma paralela.

En ambos esquemas se utilizan técnicas de prebúsqueda para anticiparnos en conocer la dirección destino del salto. Por lo tanto la anticipación conseguida será función del número de instrucciones entre dos salto consecutivos y del tiempo necesario para efectuar el cálculo de la dirección efectiva. Cuando el número de instrucciones es muy pequeño, la anticipación conseguida no es suficiente para poder ejecutar el salto en paralelo. Esta anticipación puede incrementarse en un ciclo si se utiliza el mecanismo propuesto por Katevenis para la codificación de la dirección destino de los saltos [Kate85]. Por lo tanto, en ambos esquemas haremos uso de esta codificación.

En la figura 2.11 se muestra la ejecución de un determinado código de programa para cada uno de los dos esquemas propuestos. En ambos casos se ha supuesto que al final del ciclo *i-1* la UI ha prebuscado únicamente la instrucción "*a*". El comportamiento de ambos esquemas es exactamente igual hasta que una de las instrucciones prebuscadas es un salto (ciclo $i+1$). A partir de aquí ESCO-2 busca una instrucción de cada rama del salto en cada ciclo, mientras que ESCO-1 busca dos instrucciones consecutivas de la misma rama, alternando de rama en cada ciclo. En el ejemplo de la figura 2.11 hemos supuesto que el primer acceso es a la rama destino del salto, sin embargo podría haberse empezado la prebúsqueda por la otra rama. En el siguiente capítulo analizaremos el efecto de cada una de estas alternativas. En el ciclo $i+3$ se evalúa la condición de salto. En este ejemplo hemos supuesto que el salto es efectivo, y por lo tanto, al final de este ciclo, la UI descarta las instrucciones $a+5$ y $a+6$ que previamente habían sido prebuscadas y envía la instrucción b a la unidad de ejecución.

2.4. MODELO ANALITICO

En este apartado se desarrolla un modelo que caracteriza el comportamiento del mecanismo ESCO para las dos alternativas presentadas en el apartado anterior[GLlC88b]. En primer lugar desarrollaremos un modelo que caracteriza el comportamiento del mecanismo para la ejecución de un salto, centrándonos en

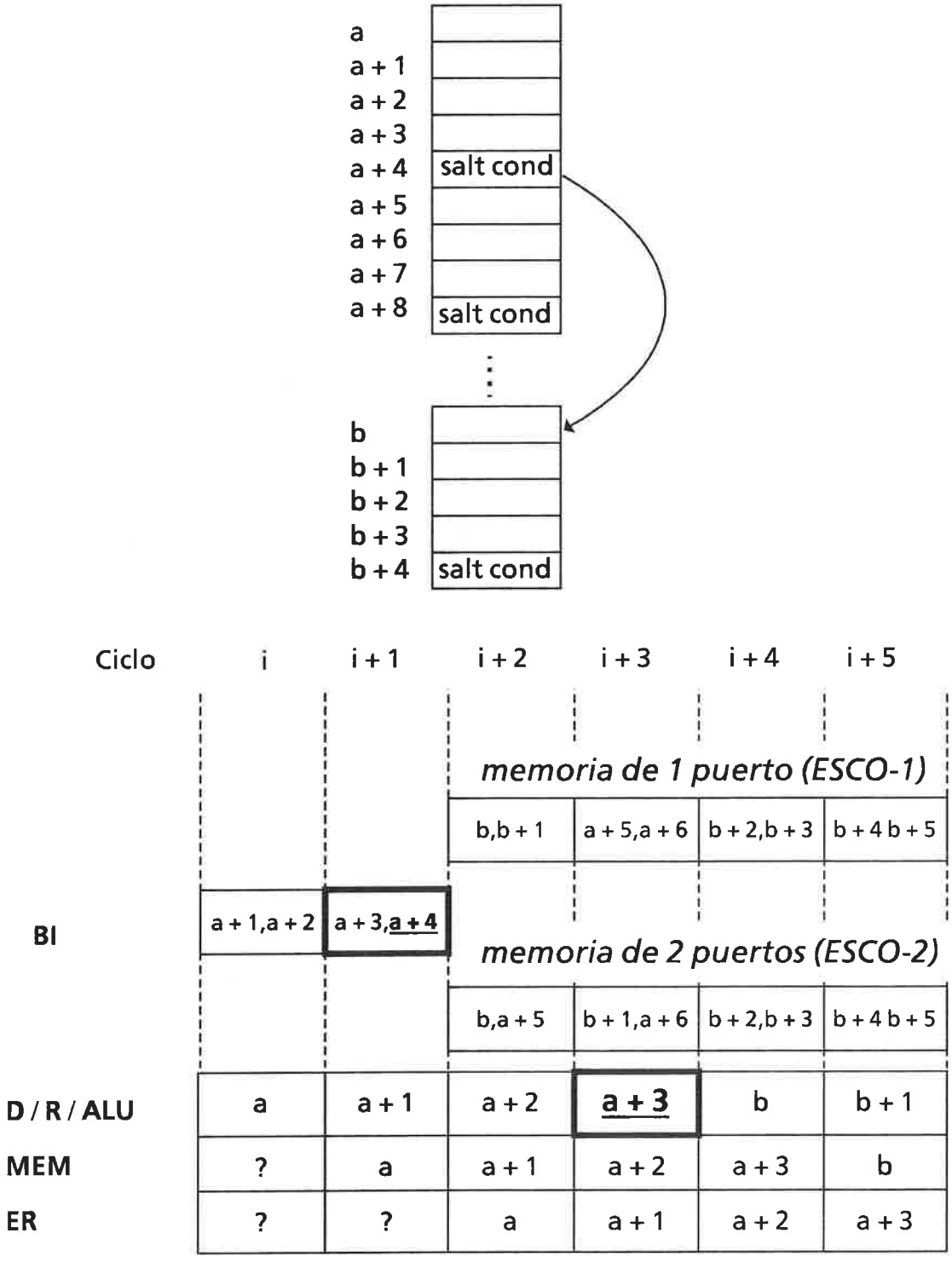


Figura 2.11: Comportamineto de los mecanismos ESCO-1 y ESCO-2

los saltos condicionales relativos al PC. Seguidamente se estudia la aplicación del mecanismo ESCO, y por lo tanto la extensión del modelo, para el resto de instrucciones de salto. A continuación, mediante un análisis de este modelo se deducen diversas características del comportamiento del mecanismo ESCO. Finalmente se modeliza el comportamiento del mecanismo durante la ejecución de un programa mediante un proceso de Markov. Los parámetros de este proceso de Markov se obtienen a partir del modelo desarrollado anteriormente.

Para explicar el modelo necesitamos introducir la siguiente definición. Un *bloque básico dinámico* (BBD) es el conjunto de instrucciones ejecutadas entre 2 saltos, sean efectivos o no, excluyendo el primero e incluyendo el segundo. De esta forma, la ejecución de cualquier programa puede representarse mediante un grafo dirigido de BBD.

La metodología utilizada para caracterizar el comportamiento del mecanismo ESCO se basa en la siguiente propiedad. El comportamiento de la UI durante la ejecución de un BBD puede determinarse analíticamente a partir de dos parámetros. Estos parámetros son la longitud del BBD (número de instrucciones) y el número de instrucciones prebuscadas justo antes de iniciarse la ejecución del BBD. A partir de estos 2 parámetros puede determinarse el coste del salto situado al final del BBD en cuestión. Del mismo modo, también puede calcularse el número de instrucciones prebuscadas del siguiente BBD a ejecutar, y de esta forma puede volverse a aplicar el modelo al siguiente BBD.

2.4.1. Caracterización del mecanismo durante la ejecución de un BBD

Sea L la longitud del BBD a analizar. Sea PI el número de instrucciones prebuscadas justo antes de iniciarse la ejecución de este BBD y sea PF el número de instrucciones del siguiente BBD a ejecutar prebuscadas justo al finalizar la ejecución del BBD actual.

El valor de PF para un determinado BBD será el valor de PI del próximo BBD a ejecutar. Si al finalizar la ejecución de un BBD el valor de PF es cero, significa que la UI no dispone de ninguna instrucción del siguiente BBD a ejecutar y por lo tanto, se producirá un ciclo de retardo. Durante este ciclo la UI irá a buscar alguna instrucción de la nueva rama a ejecutar para que al final de este ciclo, la primera instrucción de dicha rama pueda enviarse a la UE. En los modelos que

vamos a exponer a continuación este ciclo de retardo lo contabilizaremos como perteneciente al BBD donde se encuentra la instrucción de salto que lo ha ocasionado. De esta forma podemos asegurar que al finalizar la ejecución de un BBD, incluyendo en esta ejecución los ciclos de retardo (que pueden verse como NOPs) el valor de *PF* es mayor que cero y por lo tanto, el valor de *PI* para todo BBD es mayor que cero.

A continuación vamos a presentar detalladamente la obtención del modelo correspondiente a los dos esquemas propuestos en el apartado anterior. Todas las expresiones que aparecen en este apartado deben ser evaluadas en aritmética entera, a menos que se indique lo contrario.

2.4.1.1. ESCO-2

En primer lugar analizaremos el caso general donde supondremos que durante la ejecución de un BBD no se produce ningún ciclo de retardo. Dividiremos el análisis en dos subapartados: a) $PI<L$ y b) $PI \geq L$. Seguidamente estudiaremos los casos en que la instrucción de salto no puede ejecutarse con coste cero y veremos cual es el comportamiento del mecanismo en estas circunstancias.

Dado un BBD con L instrucciones, si no se produce ningún ciclo de retardo, y dado que los saltos se ejecutan en paralelo, se tardarán $L-1$ ciclos en ejecutar las instrucciones que lo componen.

2.4.1.1.1. PI < L

Cuando *PI* es menor que L, significa que al iniciarse la ejecución del BBD todavía no se ha encontrado el salto que está al final del bloque. De los $L-1$ ciclos que dura la ejecución de dicho BBD, habrá unos cuantos al principio durante los que se accederá a instrucciones del propio BBD a razón de dos por ciclo. El número de instrucciones que faltan por prebuscar de este BBD es $L-PI$ y el número de ciclos que se tardará en realizar estos accesos es $(L-PI+1) / 2$. Durante estos ciclos se accederán a $L-PI$ instrucciones si $L-PI$ es par, o $L-PI+1$ si $L-PI$ es impar. En este último caso, la última instrucción buscada es la primera instrucción del siguiente BBD. El resto de los $L-1$ ciclos, en cada uno de ellos, la UI irá a buscar una instrucción del BBD destino del salto y una del siguiente BBD. El número de estos

ciclos será:

$$L-1-\frac{L-PI+1}{2}=\frac{2L-2-L+PI-1+1}{2}=\frac{L+PI}{2}-1$$

$$\text{en aritmética entera, si } a\geq b>0 \text{ entonces } a-\frac{b}{2}=\frac{2a-b+1}{2}$$

Por lo tanto, si el salto es efectivo, como en cada uno de estos ciclos se accede a una instrucción del BBD destino del salto, $PF = (L+PI)/2 - 1$ instrucciones.

Si el salto no es efectivo, tendremos que la UI habrá prebuscado del siguiente BBD $(L+PI)/2-1$ instrucciones a las que hay que añadir una más si $L-PI$ es impar por las razones citadas anteriormente. Si $L-PI$ es impar, $L+PI$ también lo será, por lo tanto, cuando el salto no sea efectivo $PF=(L+PI+1)/2 - 1$ instrucciones.

En la figura 2.12 se ilustra de forma gráfica el comportamiento de ESCO-2 cuando $PI<L$.

2.4.1.1.2. PI ≥ L

Analicemos ahora el comportamiento del mecanismo cuando $PI \geq L$. Cuando esto ocurre, al inicarse la ejecución de un BBD la UI dispone ya de todas sus instrucciones incluida el salto. En este caso los $L-1$ ciclos que dura la ejecución del BBD se dedicarán a buscar instrucciones de las 2 ramas candidatas, una de cada rama en cada ciclo, y por lo tanto el número de instrucciones prebuscadas de cada rama será $L-1$. De esta forma, si el salto es efectivo, $PF=L-1$ instrucciones. Si el salto no es efectivo, a estas $L-1$ instrucciones habrá que añadir las que ya habían sido prebuscadas antes de iniciarse la ejecución del BBD y que pertenecen al siguiente BBD. Este número de instrucciones es igual a $PI-L$. Por lo tanto, si el salto no es efectivo, $PF=(L-1) + (PI-L) = PI-1$ (ver figura 2.13).

2.4.1.1.3. Saltos con coste mayor que cero.

Nos queda únicamente por determinar y analizar los casos en que la ejecución de un salto produce algún ciclo de retardo. En primer lugar tenemos aquellos BBD

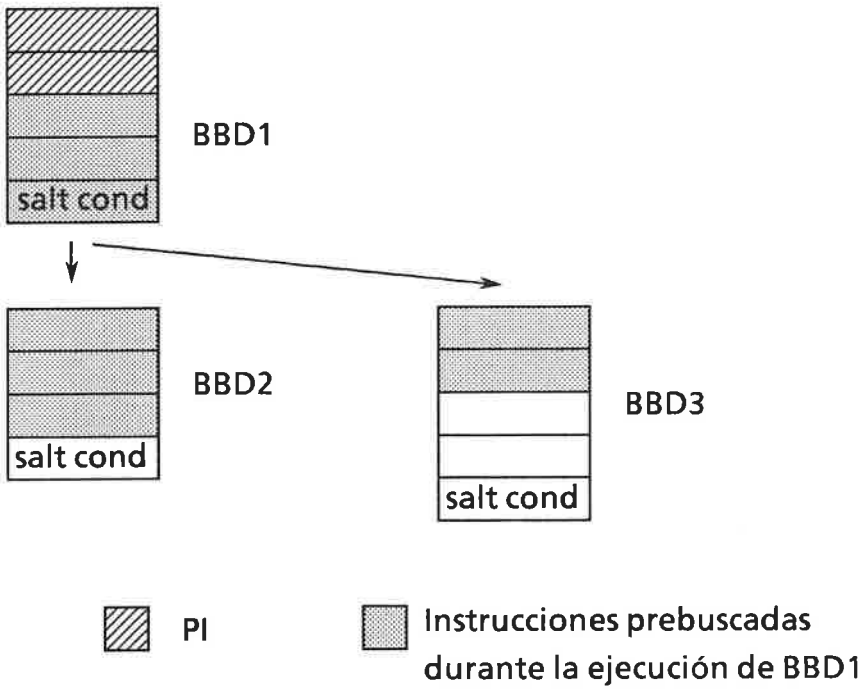


Figura 2.12: Comportamineto del mecanismo ESCO-2 cuando PI<L

cuya única instrucción es un salto, es decir, cuando L vale 1. Dado que aparte del salto no existe ninguna otra instrucción en el BBD, la ejecución del salto no puede llevarse a cabo en paralelo con la ejecución de alguna instrucción de manipulación de datos, y por lo tanto, durante el ciclo que dura su ejecución la UE no podrá realizar ningún trabajo útil. Durante este ciclo de retardo la UI busca una instrucción de cada una de las dos ramas candidatas. En consecuencia $PF=1$ si el salto es efectivo y $PF = PI$ si el salto no es efectivo (ver figura 2.14).

El resto de saltos que ocasionan ciclos de retardo son aquellos en que dicha instrucción es prebuscada en el mismo ciclo en el que se empieza a ejecutar la instrucción que lo precede. En estas circunstancias, la UI no puede tomar la decisión de salto en este mismo ciclo, ya que el test que hay que realizar sobre los códigos de condición está codificado en la instrucción de salto y por lo tanto se

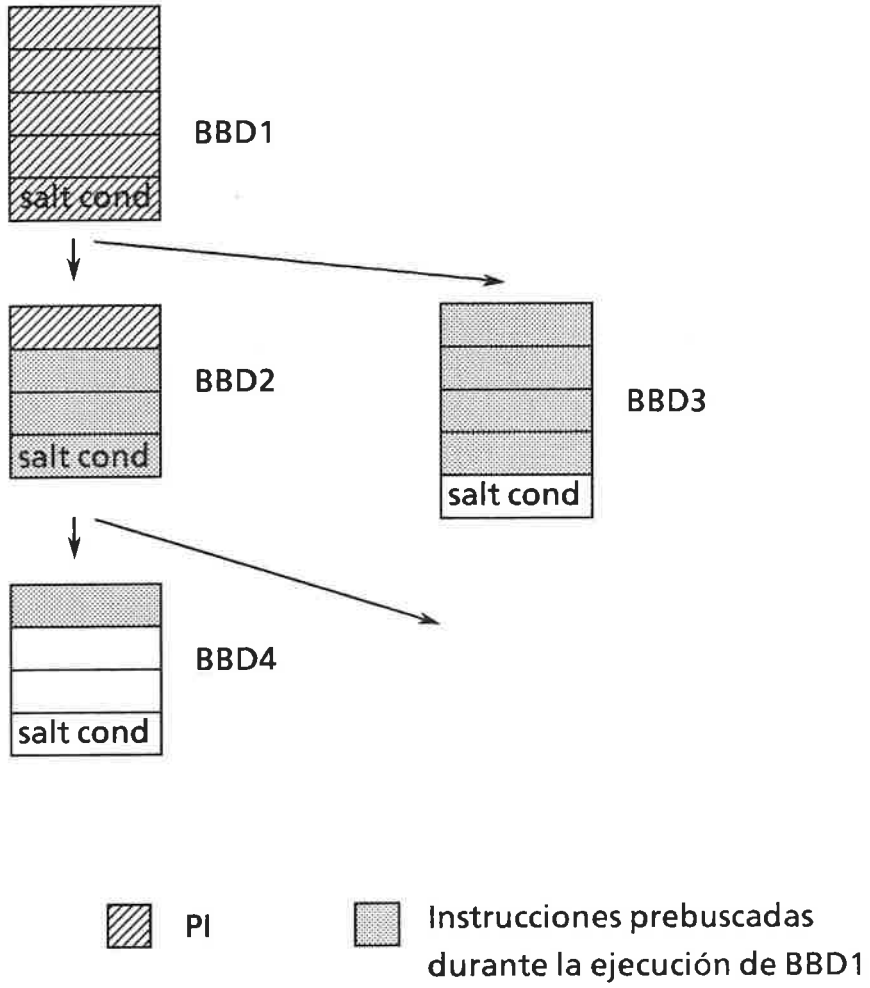


Figura 2.13: Comportamineto del mecanismo ESCO-2 cuando PI ≥ L

desconoce hasta el final de este ciclo. En el siguiente capítulo se analiza la posibilidad de que la UI tomara la decisión de salto en este mismo ciclo.

Si el salto se va a buscar en el mismo ciclo en que se ejecuta la instrucción que lo precede, la UI no habrá podido prebuscar ninguna instrucción de la rama destino del salto. Por lo tanto, todos estos casos serán aquellos en que aplicando las fórmulas halladas para el caso general obtenemos que *PF* vale cero cuando el salto es efectivo. Para hallarlos bastará resolver la siguiente ecuación.

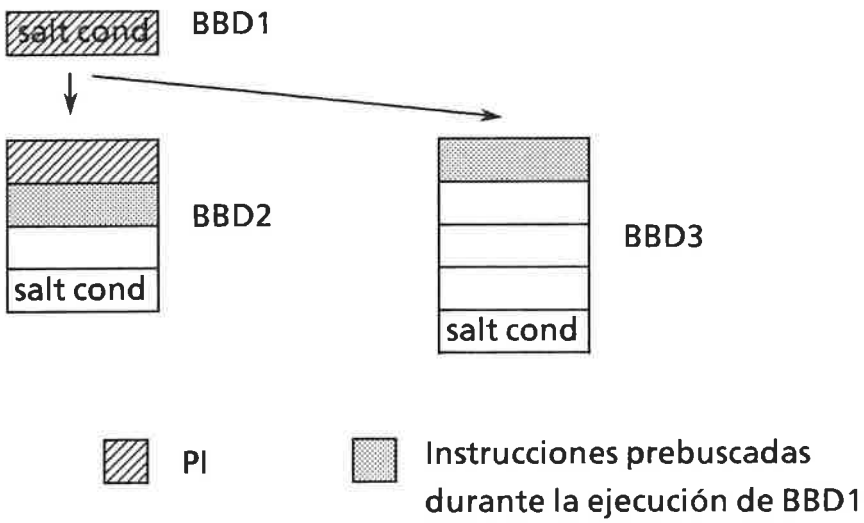


Figura 2.14: Comportamineto del mecanismo ESCO-2 cuando L = 1

$$PF = \left\{ \begin{array}{ll} L-1 & si\, PI \geq L \\ \dfrac{PI+L}{2} - 1 & si\, PI < L \end{array} \right\} = 0$$

Del conjunto de soluciones a esta ecuación, aquellas en que $L=1$ ya han sido contabilizadas anteriormente. De esta forma, los únicos saltos causantes de un ciclo de retardo que todavía no hemos tenido en cuenta son aquellos que pertenecen a un BBD con $L=2$ y $PI=1$. Durante la ejecución de este ciclo de retardo la UI irá a buscar una instrucción de cada una de las dos ramas candidatas, y por lo tanto, $PF=1$ si el salto es efectivo y $PF=2$ si el salto no es efectivo (ver figura 2.15).

Faltaría por determinar si existe algún caso todavía no analizado para el que el valor de PF es cero cuando el salto no es efectivo. Nos faltan por analizar aquellos BBD en que la instrucción de salto es buscada antes del ciclo en que se inicia la ejecución de la instrucción que lo precede. Bajo estas circunstancias, la UI tiene como mínimo un ciclo para prebuscar instrucciones de las dos ramas del

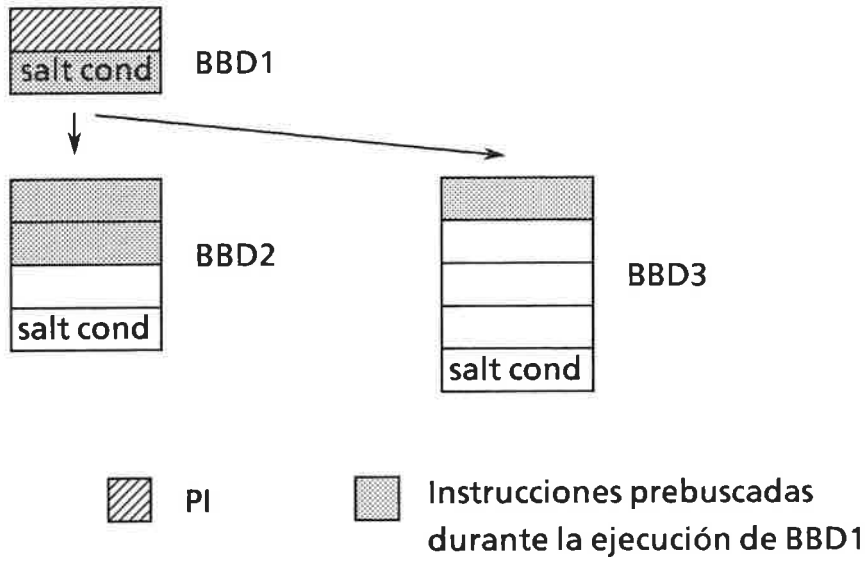


Figura 2.15: Comportamineto del mecanismo ESCO-2 cuando L = 2 y PI = 1

salto. Dado que esta prebúsquda se realiza en paralelo por ambas ramas podemos asegurar que en ninguno de estos casos *PF* valdrá cero.

En la figura 2.16 se resume el modelo que caracteriza el comportamiento de ESCO-2 durante la ejecución de un BBD.

2.4.1.2 ESCO-1

Con el mecanismo ESCO-1 la UI dispone de un ancho de banda de dos instrucciones por ciclo. Ahora bién, estas dos instrucciones no pueden ser cualesquiera, sino que deben de estar formando un bloque. Por lo tanto sus direcciones deben ser consecutivas, y la dirección de la primera de ellas debe ser múltiplo del tamaño del bloque (en este caso 2).

De esta forma, cuando la UI realice el primer acceso a la rama destino de un salto, dependiendo de si esta dirección es par o impar, podrá leer en dicho acceso la primera instrucción de esta rama y la siguiente o bién la primera instrucción y la

	SALTO EFECTIVO				SALTO NO EFECTIVO			
	L = 1	L = 2 y PI = 1	(L = 2 y PI > 1) ó L > 2		L = 1	L = 2 y PI = 1	(L = 2 y PI > 1) ó L > 2	
			PI < L	PI ≧ L			PI < L	PI ≧ L
PF	1	1	(L + PI)/2-1	L-1	PI	2	(L+PI+1)/2-1	PI-1
Coste del salto	1	1	0	0	1	1	0	0

Figura 2.16: Modelo de la ejecución de un BBD con el mecanismo ESCO-2 (Las expresiones deben evaluarse utilizando aritmética entera).

anterior. En este último caso, de las dos instrucciones sólo será útil la primera, y por lo tanto, a nivel de rendimiento será como si la UI pudiera acceder únicamente a una instrucción. En consecuencia, para modelizar el comportamiento del mecanismo ESCO-1 necesitamos introducir una variable más que denominaremos *IMPAR*. Su valor será uno cuando la dirección destino de un salto sea impar y cero en caso contrario.

Vamos a suponer que la prebúsqueda de instrucciones se inicia por la rama destino del salto. En el capítulo siguiente se justifica esta elección.

Al igual que con ESCO-2 vamos a empezar analizando los BBD en los que no se produce ningún ciclo de retardo durante la ejecución de su salto. Por lo tanto, para ejecutar un BBD con L instrucciones harán falta $L-1$ ciclos.

2.4.1.2.1. PI < L

Si $PI<L$, dado que ESCO-1 y ESCO-2 se comportan exactamente igual hasta encontrar una instrucción de salto, podemos remitirnos a los resultados de la sección anterior donde hallamos que tras prebuscar las instrucciones del actual BBD, la UI dispone de $(L+PI)/2-1$ ciclos para prebuscar instrucciones por ambas ramas. Durante estos ciclos podrá acceder a

$$\frac{\frac{L+PI}{2}}{2} x2 - IMPAR = \frac{L+PI}{4} x2 - IMPAR$$

instrucciones de la rama destino del salto y a

$$\frac{\frac{L+PI}{2}-1}{2} x2 = \frac{\frac{L+PI-2}{2}}{2} x2 = \frac{L+PI-2}{4} x2$$

instrucciones de la rama siguiente en secuencia al salto. Ahora bién, a este número de instrucciones hay que añadirle una más si *L-PI*, que es el número de instrucciones que faltan por buscar del propio BBD, es impar. Por lo tanto el valor de PF cuando el salto sea no efectivo será

$$\frac{L+PI-2}{4} x2 + (L-PI) \, mod \, 2$$

2.4.1.2.2. PI ≥ L

Cuando *PI* ≥ *L*, los *L-1* ciclos que dura la ejecución del BBD se dedican a prebuscar instrucciones de ambas ramas del salto. Por lo tanto el número de instruciones prebuscadas de la rama destino del salto será *L / 2 x 2 -IMPAR*. Si el salto es no efectivo el valor de *PF* será

$$\frac{L-1}{2} x2 + PI - L = PI - 2 + L \, mod \, 2$$

2.4.1.2.3. Saltos con coste mayor que cero.

Al igual que en ESCO-2, cuando un BBD tiene como una única instrucción un salto, dicho salto no puede ejecutarse en paralelo. Durante su ejecución en la UI la UE está ejecutando una NOP, mientras que se prebuscan las dos primeras instrucciones de la rama destino del salto. Cuando *PI* valga 1, dado que el ciclo de ejecución de la NOP se dedica a prebuscar instruciones del destino del salto, si el salto no es efectivo, la UI no tendrá prebuscada ninguna instrucción de la nueva

rama. Por lo tanto será preciso otro ciclo de retardo para buscar las 2 primeras instrucciones que siguen en secuencia a la instrucción de salto.

Cuando $L=2$ y $PI=1$, durante el primer ciclo de ejecución del BBD, mientras se ejecuta su primera instrucción la UI accede a memoria para buscar la instrucción de salto y la siguiete en secuencia. Por los mismos motivos expuestos anteriormente para ESCO-2, vamos a suponer que la UI no puede tomar una decisión de salto en el mismo ciclo en que la instrucción de salto es accedida. Por lo tanto, independientemente de si el salto es efectivo o no, se producirá un ciclo de retardo, durante el que se accederá al primer bloque del destino del salto. El valor de *PF* será *2-IMPAR* si el salto es efectivo y 1 si es no efectivo.

El resto de BBD cuyo salto produce un ciclo de retardo podemos encontrarlos analizando para qué casos las ecuaciones que describen el comportamiento en el caso general dan como resultdado $PF=0$. Estos son los siguientes.

- $PI=1, L=3$ y salto no efectivo
- $PI=2, L=2$ y salto no efectivo

En ambos casos, durante el ciclo de retardo se van a buscar las dos instrucciones situadas justo a continuación del salto. Por lo tanto, el valor de PF será 2.

En la figura 2.17 se resume el modelo que caracteriza el comportamiento de ESCO-2 durante la ejecución de un BBD.

2.4.2. Soporte a otros tipos de salto

La tabla 2.1 muestra el porcentaje de las diferentes clases de instrucciones de salto ejecutadas en varios programas de prueba. Las características de estos programas se comentan en el siguiente capítulo.

Los modelos presentados en el apartado anterior son válidos para saltos condicionales relativos al PC, que en los programas de prueba analizados representan entre un 69% y un 97% del total de saltos. Los saltos incondicionales relativos al PC (entre 12% y 1% del total de saltos) pueden tratarse como un caso particular de los anteriores, es decir, como saltos condicionales que siempre son

	SALTO EFECTIVO			
	L = 1	L = 2 y PI = 1	(L = 2 y PI > 1) ó L > 2	
			PI < L	PI ≧ L
PF	2-IMPAR	2-IMPAR	(L + PI)/4x2-IMPAR	L/2x2-IMPAR
Coste del salto	1	1	0	0

	SALTO NO EFECTIVO						
	L = 1 y PI = 1	L = 1 y PI > 1	L = 2 y PI = 1	L = 2 y PI = 2	L = 3 y PI = 1	(L = 2 y PI > 2) ó (L = 3 y PI > 1) ó L > 3	
						PI < L	PI ≧ L
PF	2	PI-1	1	2	2	(L + PI-2)/4x2 + (L-PI) mod 2	PI-2 + L mod 2
Coste del salto	2	1	1	1	1	0	0

Figura 2.17: Modelo de la ejecución de un BBD con el mecanismo ESCO-1 (Las expresiones deben evaluarse utilizando aritmética entera)

efectivos. Una alternativa que analizaremos en el siguiente capítulo podría ser que la UI detectara estos casos y se limitara a prebuscar únicamente instrucciones de la rama destino del salto.

El siguiente tipo de saltos que vamos a analizar son los saltos no relativos al PC. La dirección destino de estos saltos se calcula en función del contenido de un registro cualquiera de propósito general y por lo tanto, el contenido de este registro puede ser modificado por cualquiera de las instrucciones del actual BBD. En consecuencia, la UI no puede conocer la dirección destino hasta que la instrucción anterior al salto haya finalizado su etapa de ALU. En estos casos la UI no puede prebuscar ninguna instrucción de la rama destino del salto y el rendimiento del sistema es bastante menor que para los saltos relativos al PC. Sin embargo, tal y como se aprecia en la tabla 2.1, este tipo de saltos son muy poco

%	LEX	NROFF	PCC	YACC	Media
CR	96.9	76.3	69.3	93.9	84.1
CNR	0	0	0	0	0
IR	1.5	2.4	12.2	3.2	4.83
INR	0	0.1	1.3	0	0.35
LLR	0.8	11.4	7.6	2.4	5.55
LLNR	0	0.1	1.6	0	0.42
RET	0.8	9.7	8.0	0.5	4.75

CR: Saltos condicionales relativos al PC
CNR: Saltos condicionales no relativos al PC
IR: Saltos incondicionales relativos al PC
INR: Saltos incondicionales no relativos al PC
LLR: LLamadas a subrutina relativas al PC
LLNR: LLamadas a subrutina no relativas al PC
RET: Retornos de subrutina

NOTA: El número de llamadas a subrutina es superior al de retornos debido a que las llamadas a la librería del sistema se han simulado como instrucciones de salto a la siguiente instrucción, y por lo tanto, en estos casos no llega a ejecutarse nunca el correspondiente retorno.

Tabla 2.1: Porcentaje de los diferentes tipos de saltos.

frecuentes y por lo tanto tienen un efecto casi despreciable sobre el rendimiento global del sistema.

Supondremos que el modo no relativo al PC sólo puede utilizarse para saltos incondicionales y llamadas a subrutina. En la tabla 2.1 puede comprobarse que esta hipótesis se cumple para los programas de prueba analizados. En cualquier otro caso, el error introducido sería mínimo ya que principalmente es en este tipo de saltos donde más utilidad tiene un direccionamiento no relativo al PC.

El coste de un salto no relativo al PC es: a) un ciclo en el que la UE lee el registro correspondiente y le suma el desplazamiento especificado, más b) los ciclos necesarios para que que la UI tenga disponible alguna instrucción de la nueva rama a ejecutar. Este último sumando es igual a uno ya que tras haberse calculado la dirección destino en la UE, la UI necesita este ciclo para acceder a las primeras instrucciones del destino del salto. Por lo tanto, en ESCO-1 el valor de PF para este tipo de BBD será 2-IMPAR mientras que en ESCO-2 tenemos que PF=2.

Las instrucciones de llamada a subrutina equivalen a saltos incondicionales con la peculiaridad de que en algún momento debe guardarse la dirección de retorno. Para tratar este tipo de instrucciones existen básicamente dos alternativas.

La primera es que la UI disponga de una pila donde guarde la dirección de retorno cada vez que se envía a ejecutar la instrucción que precede a una llamada. Esto va a complicar el diseño de la UI pero va a proporcionar un mayor rendimiento ya que permitirá que las instrucciones de llamada y retorno de subrutina sean transparentes a la UE. En este caso, desde el punto de vista del modelo, las instrucciones de retorno de subrutina pueden modelizarse como saltos incondicionales de los que la UI conoce la dirección destino, ya que ésta se encuentra en la cabeza de la pila. Por lo tanto, los modelos desarrollados en el apartado anterior pueden utilizarse para caracterizar las instrucciones de llamada y retorno de subrutina

La otra alternativa, que analizaremos en detalle en el siguiente capítulo, es que sea la UE la que se encargue de gestionar la dirección de retorno. En este caso, la UI trataría estas instruciones como saltos incondicionales con la particulariad de que dichas instrucciones serían enviadas a la UE para que ésta

almacenase o obtuviera la dirección de retorno. Con esta última alternativa, el rendimiento será menor pero el diseño de la UI va a ser más simple.

2.4.3. Análisis del modelo análitico

Una aportación de los modelos desarrollados en este capítulo es que explican de una forma concisa el comportamiento del mecanismo propuesto. Mediante un análisis de ellos podemos intuir cual será el comportamiento del mecanismo ESCO.

A partir de los modelos expuestos en las figuras 2.16 y 2.17 podemos preveer que el coste medio de una instrucción de salto va a ser bastante reducido. Por ejemplo, para ESCO-2, siempre que un BBD tenga más de 2 instrucciones el coste del salto asociado va a ser cero. Para el resto de saltos su coste va a depender, además de la longitud de su BBD, de la cantidad de instrucciones prebuscadas al iniciarse la ejecución de dicho BBD. En cualquier caso, dado que la proporción de BBD con una o dos intrucciones generalmente no es demasiado elevado (entre 5 y 40% en los programas de prueba analizados), y teniendo en cuenta que sólo aquellos en que *PI* valga 1 no podrán ejecutar su salto con coste cero , es de esperar que el coste medio de un salto sea bastante reducido.

Analizando estos modelos podemos también realizar una comparación cualitativa entre ESCO-1 y ESCO-2. El comportamiento de ambos mecanismos es bastante similar para los saltos efectivos. Como puede observarse en las figuras 2.16 y 2.17, el coste de los saltos efectivos es el mismo para ambos esquemas. La única diferencia está en la cantidad de instrucciones prebuscadas del siguiente BBD a ejecutar (PF). El valor medio de PF para aquellos salto cuyo coste es cero es el mismo en los dos mecanismos e igual a la siguiente expresión. (NOTA: Las expresiones siguientes, así como el resto de expresiones que aparecen en este apartado deben evaluarse con aritmética real)

$$Si PI \geq L \rightarrow L-1$$

$$Si PI < L \rightarrow \frac{L+PI}{2} - 1.25$$

Por otra parte, para aquellos saltos cuyo coste es diferente a cero, el valor medio de *PF* es algo superior para ESCO-1, ya que en este caso vale 1.5 mientras que en ESCO-2 es igual a 1.

En las figuras 2.16 y 2.17 puede observarse que los saltos no efectivos se ejecutarán en ESCO-2 con coste cero en más ocasiones que en ESCO-1. Por lo tanto el coste de los saltos no efectivos es algo superior en ESCO-1. El valor medio de *PF* para los saltos con coste cero en ESCO-1 es igual a

$$Si PI \geq L \rightarrow PI - 1.5$$

$$Si PI < L \rightarrow \frac{L+PI}{2} - 1.25$$

mientras que para ESCO-2 obtenemos

$$Si PI \geq L \rightarrow PI - 1$$

$$Si PI < L \rightarrow \frac{L+PI}{2} - 0.75$$

Para el resto de los saltos no efectivos ($L=1$, $L=2$ *y* $PI \leq 2$, $L=3$ *y* $PI=1$) el valor de *PF* es en algunos casos una unidad mayor en ESCO-1 mientras que en otros es una unidad menor. Si no tenemos en cuenta estos casos, tendremos que en media el valor de *PF* es media instrucción mayor en ESCO-2.

En consecuencia, el rendimiento obtenido con ambos mecanismos será bastante similar para los saltos efectivos (ligeramente mejor con ESCO-1). Para los saltos no efectivos ESCO-2 se comportará algo mejor que ESCO-1. Dado que la proporción de saltos efectivos generalmente es bastante superior a la de saltos no efectivos (ver tabla 2.2) cabe esperar que el rendimiento obtenido con ambos esquemas sea bastante similar.

	LEX	NROFF	PCC	YACC	Media
Saltos efectivos	89%	69%	71%	75%	76%
Saltos no efect.	11%	31%	29%	25%	24%

Tabla 2.2: Porcentaje de saltos efectivos y no efectivos en los programas de prueba

2.4.4 Modelización del comportamiento del sistema mediante cadenas de Markov

El rendimiento del mecanismo ESCO podemos medirlo como el coste medio de una instrucción de salto. Para estimar este coste a partir de los modelos desarrollados anteriormente necesitamos conocer: a) la función de densidad de probabilidad de la longitud de un BBD, b) la probabilidad de que un salto sea efectivo, c) la probabilidad de que un salto sea relativo al PC y d) la función de densidad de probabilidad del número de instrucciones prebuscadas al iniciarse la ejecución de un BBD.

Los tres primeros parámetros pueden obtenerse de la ejecución de un conjunto de programas de prueba representativos de la carga real del sistema. Dicha ejecución puede llevarse a cabo en la propia máquina en la que se desea implemetar el mecanismo (si se dispone de ella), o bién puede ser emulada utilizando las técnicas propuestas en [CoLl87a]. En cualquier caso, estos tres parámetros son independientes del mecanismo utilizado para tratar las instrucciones de salto, y por lo tanto, para su obtención no es preciso ni implemetar ni simular dicho mecanismo.

Por otra parte, el número de instrucciones prebuscadas al iniciarse la ejecución de un BBD (*PI*) vendrá determinado por el comportamiento del mecanismo ESCO. El valor de esta variable será 1 al iniciarse la ejecución de un

programa e irá variando cada BBD en función de su valor actual, de la longitud del BBD y del tipo de salto ejecutado al final del BBD. Por lo tanto, con estas hipótesis, podemos modelizar el comportamiento del mecanismo mediante una cadena de Markov donde el estado vendrá determinado por el valor de la variable *PI*, y las transiciones entre estados podrán ocurrir cada BBD ejecutado.

2.4.4.1. Definiciones previas

- Sea $f_L(L)$ la función de densidad de probabilidad de la longitud de un BBD.
- Sea *PE* la probabilidad de que un salto sea efectivo.
- Definimos como estado del sistema el número de instrucciones prebuscadas justo antes de iniciarse la ejecución de un BBD (*PI*).
- Sea $P=(p_{ij})$ la matriz de probabilidades de transición de estado, donde p_{ij} representa la probabilidad de pasar del estado i al estado j en un solo paso, es decir, trás la ejecución de un BBD.

2.4.4.2. Cálculo de las probabilidades de transición

A partir de los modelos de las figuras 2.16 y 2.17 podemos determinar los valores de los elementos de la matriz P para ESCO-1 y ESCO-2. Las expresiones que definen el valor de los elementos de la matriz P son las siguientes.

ESCO-1

1) *Transiciones:* $2j<i-2$

$$p_{i,2j} = [f_L(2j)+f_L(2j+1)]\,0.5\,PE$$
$$p_{i,2j-1} = [f_L(2j)+f_L(2j+1)]\,0.5\,PE$$

Excepciones :

$$p_{i,2} = [f_L(1)+f_L(2)+f_L(3)]\,0.5\,PE \qquad i>4$$
$$p_{i,1} = [f_L(1)+f_L(2)+f_L(3)]\,0.5\,PE \qquad i>4$$

2) *Transiciones: $2j=i-2$*

$$p_{i,i-2} = [f_L(i-2)+f_L(i-1)]\,0.5\,PE + (1-PE)\sum_{k=1}^{i/2} f_L(2k)$$

$$p_{i,i-3} = [f_L(i-2)+f_L(i-1)]\,0.5\,PE$$

Excepciones:

$$p_{4,2} = [f_L(1)+f_L(2)+f_L(3)]\,0.5\,PE + [f_L(2)+f_L(4)]\,(1-PE)$$

$$p_{4,1} = [f_L(1)+f_L(2)+f_L(3)]\,0.5\,PE$$

3) *Transiciones: $2j=i-1$*

$$p_{i,i-1} = [f_L(i-1)+f_L(i)+f_L(i+1)]\,0.5\,PE + (1-PE)\sum_{k=0}^{(i+1)/2} f_L(2k+1)$$

$$p_{i,i-2} = [f_L(i-1)+f_L(i)+f_L(i+1)]\,0.5\,PE + (1-PE)\sum_{k=1}^{(i-1)/2} f_L(2k)$$

Excepciones:

$$p_{3,2} = [f_L(1)+f_L(2)+f_L(3)+f_L(4)]\,0.5\,PE + [f_L(1)+f_L(3)+f_L(5)]\,(1-PE)$$

$$p_{3,1} = [f_L(1)+f_L(2)+f_L(3)+f_L(4)]\,0.5\,PE + f_L(2)\,(1-PE)$$

4) *Transiciones: $2j=i$*

$$p_{i,i} = [f_L(i)+f_L(i+1)+f_L(i+2)+f_L(i+3)]\,0.5\,PE + [f_L(i+2)+f_L(i+4)]\,(1-PE)$$

$$p_{i,i-1} = [f_L(i)+f_L(i+1)+f_L(i+2)+f_L(i+3)]\,0.5\,PE + (1-PE)\sum_{k=0}^{i/2} f_L(2k+1)$$

Excepciones:

$$p_{2,2} = [f_L(1)+f_L(2)+f_L(3)+f_L(4)+f_L(5])\,0.5\,PE + [f_L(2)+f_L(4)+f_L(6)]\,(1-PE)$$

$$p_{2,1} = [f_L(1)+f_L(2)+f_L(3)+f_L(4)+f_L(5)]\,0.5\,PE + (f_L(1)+f_L(3))\,(1-PE)$$

5) *Transiciones: $2j=i+1$*

$$p_{i,2j} = [f_L(i+2)+f_L(i+3)+f_L(i+4)+f_L(i+5)]\,0.5\,PE + [f_L(i+4)+f_L(i+6)]\,(1-PE)$$

$$p_{i,2j-1} = [f_L(i+2)+f_L(i+3)+f_L(i+4)+f_L(i+5)]\,0.5\,PE + [f_L(i+1)+f_L(i+3)]\,(1-PE)$$

Excepciones:

$$p_{1,2} = [f_L(1)+f_L(2)+f_L(3)+(f_L(4)+f_L(5)+f_L(6)]\;0.5PE + [f_L(1)+f_L(3)+f_L(5)+f_L(7)](1-PE)$$

$$p_{1,1} = [f_L(1)+f_L(2)+f_L(3)+(f_L(4)+f_L(5)+f_L(6)]\,0.5\,PE + [f_L(2)+f_L(4)]\,(1-PE)$$

6) *Transiciones: $2j>i+1$*

$$p_{i,2j} = [f_L(4j-i)+f_L(4j-i+1)+f_L(4j-i+2)+f_L(4j-i+3)]0.5PE + [f_L(4j-i+2)+f_L(4j-i+4)](1-PE)$$

$$p_{i,2j-1} = [f_L(4j-i)+f_L(4j-i+1)+f_L(4j-i+2)+f_L(4j-i+3)]0.5PE + [f_L(4j-i-1)+f_L(4j-i+1)](1-PE)$$

ESCO-2

1) *Transiciones:* $j<i-1$ $\quad p_{i,j} = f_L(j+1)\,PE$

   Excepción $\quad p_{i,1} = [f_L(1) + f_L(2)]\,PE \qquad si\ i \geq 3$

2) *Transiciones:* $j=i-1$ $\quad p_{i,i-1} = [f_L(i) + f_L(i+1)]\,PE + (1\text{-}PE) \sum_{k=2}^{i} f_L(k)$

   Excepción $\quad p_{2,1} = [f_L(1) + f_L(3)]\,PE + f_L(2)$

3) *Transiciones:* $j=i$ $\quad p_{i,i} = f_L(i+2) + f_L(i+3)\,PE + [f_L(i+1) + f_L(1)]\,(1\text{-}PE)$

   Excepción $\quad p_{1,1} = f_L(3) + [f_L(2) + f_L(4)]\,PE + f_L(1)$

4) *Transiciones:* $j>i$ $\quad p_{i,j} = f_L(2(j+1)\text{-}i) + f_L(2(j+1)\text{-}i+1)PE + f_L(2(j+1)\text{-}i\text{-}1)\,(1\text{-}PE)$

   Excepción $\quad p_{1,2} = f_L(5) + f_L(6)\,PE + [f_L(2) + f_L(4)]\,(1\text{-}PE)$

Estas expresiones no contemplan el efecto producido por los saltos no relativos al PC. Debido al bajo porcentaje de este tipo de saltos (tabla 2.2) el error cometido sería mínimo. Sin embargo, con una pequeña modificación de la matriz P podemos conseguir tener en cuenta este tipo de saltos. Para ello definimos *PREL* como la probabilidad de que un salto sea relativo al PC. Los retornos de subrutina se contabilizan como relativos al PC. Realmente no lo son pero cuando las direcciones de retorno son gestionadas por la UI, a efectos funcionales equivale a que lo fueran ya que su dirección destino es conocida tan pronto como se inicia la ejecución de su BBD.

Para contemplar los saltos no relativos al PC en la matriz de transición, bastará realizar los siguientes modificaciones a la matriz anteriormente calculada.

ESCO-1

$$p_{i1} = p_{i1}\,PREL + (1\text{-}PREL)\,0.5$$
$$p_{i2} = p_{i2}\,PREL + (1\text{-}PREL)\,0.5$$
$$p_{ij} = p_{ij}\,PREL \qquad j>2$$

ESCO-2

$$p_{i2} = p_{i2}\,PREL + (1\text{-}PREL)\,0.5$$
$$p_{ij} = p_{ij}\,PREL \qquad j \neq 2$$

2.4.4.3. Análisis del comportamiento asintótico

Para calcular las probabilidades de estado debemos estudiar el comportamiento asintótico del proceso de Markov caracterizado por la matriz de transición P previamente calculada. Para ello es preciso calcular la matriz

$$\Phi = \lim_{n\to\infty} P^n$$

Para todos los programas de prueba analizados se ha podido comprobar que la matriz Φ existe y además que esta matriz tiene todas sus filas iguales. Ello indica que se trata de procesos de Markov monodésmicos, en los que, la probabilidad de encontrarse en el estado j no depende del estado inicial. La probabilidad de estar en el estado j viene dada por el elemento Φ_{ij} de la matriz Φ (el valor de i es indiferente ya que todas sus filas son iguales). En la tabla 2.3 se muestran estas probabilidades para cada uno de los programas de prueba analizados.

2.4.5. Aplicaciones del modelo analítico

Como acabamos de ver, una primera aplicación de los modelos analíticos desarrollados en este capítulo es que nos permiten modelizar el comportamiento del sistema mediante un proceso de Markov. Esta modelización nos permitirá obtener de forma muy simple medidas sobre el rendimiento del procesador.

Por otra parte, si queremos obtener medidas más precisas podemos recurrir a realizar una simulación del comportamineto del mecanismo ESCO durante la ejecución de los programas de prueba. Esta simulación será más costosa en cuanto a tiempo de cálculo (sobre todo para programas grandes). No obstante, los modelos analíticos de las figuras 2.16 y 2.17 nos van a proporcionar una considerable simplificación ya que no será preciso modelar todos los componentes de la UI y realizar una simulación a nivel de transferencia de registro. La simulación

Instrucciones prebuscadas	LEX	NROFF	PCC	YACC
1	0.396	0.326	0.375	0.219
2	0.410	0.370	0.405	0.272
3	0.081	0.123	0.092	0.215
4	0.072	0.096	0.072	0.179
5	0.017	0.037	0.027	0.053
6	0.014	0.024	0.019	0.043
>6	0.010	0.024	0.010	0.015

a) **ESCO-1**

Instrucciones prebuscadas	LEX	NROFF	PCC	YACC
1	0.500	0.388	0.516	0.175
2	0.310	0.279	0.249	0.254
3	0.095	0.143	0.106	0.281
4	0.055	0.095	0.066	0.159
5	0.024	0.048	0.038	0.081
6	0.007	0.020	0.014	0.034
>6	0.009	0.027	0.021	0.026

a) **ESCO-2**

Tabla 2.3: Función de densidad de probabilidad del número de instrucciones prebuscadas al iniciarse la ejecución de un BBD (PI).

utilizará unicamente una traza de los BBD ejecutados. Para cada uno de ellos, a partir de su longitud y del número de instrucciones prebuscadas a la entrada, se calculará el coste del salto asociado y el número de instrucciones prebuscadas del siguiente BBD. Por lo tanto la simulación se hará a nivel de BBD. De esta forma podemos obtener fácilmente el rendimiento del mecanismo propuesto que será función del número de ciclos de retardo ocasionados por el conjunto de instrucciones de salto.

Otra utilidad de estos modelos es que ponen de manifiesto de una forma concisa el comportamiento del mecanismo propuesto. Ello nos ha permitido realizar una estimación del comportamiento de los esquemas ESCO-1 y ESCO-2 sin necesidad de evaluar su comportamiento con ningún programa de prueba.

Una última ventaja de estos modelos es que en ellos quedan perfectamente determinados en que bloques básicos se encuentran ubicados los saltos candidatos a ser ejecutados con un coste mayor que cero. Por ejemplo, para ESCO-2 tenemos que sólo los saltos pertenecientes a bloques básicos de longitud menor que 3 pueden producir ciclos de retardo. Esta información puede ser utilizada por el compilador para mejorar el rendimiento del sistema mediante reordenaciones en el código del programa que consigan agrandar el tamaño de estos bloques.

2.5. ARQUITECTURAS SIN CODIGOS DE CONDICION

Para el secuenciamiento de un programa, las arquitecturas sin códigos de condición disponen de instrucciones de comparación-y-salto, de forma que tanto la evaluación de la condición como el cálculo de la dirección destino lo realiza una misma instrucción.

El mecanismo presentado en este capítulo podría utilizarse en este tipo de arquitecturas. La UI sería la unidad responsable del secuenciamiento del programa con la salvedad que, al igual que antes, la evaluación de la condición se realizaría en la UE. Por lo tanto, las instrucciones de comparación-y-salto serían utilizadas por la UI para evaluar la dirección destino y prebuscar instrucciones por ambas ramas a fin de reducir la penalización del salto y además deberían ser enviadas a la UE para evaluar la condición de salto. Este modelo de ejecución se ilustra en la figura 2.18.

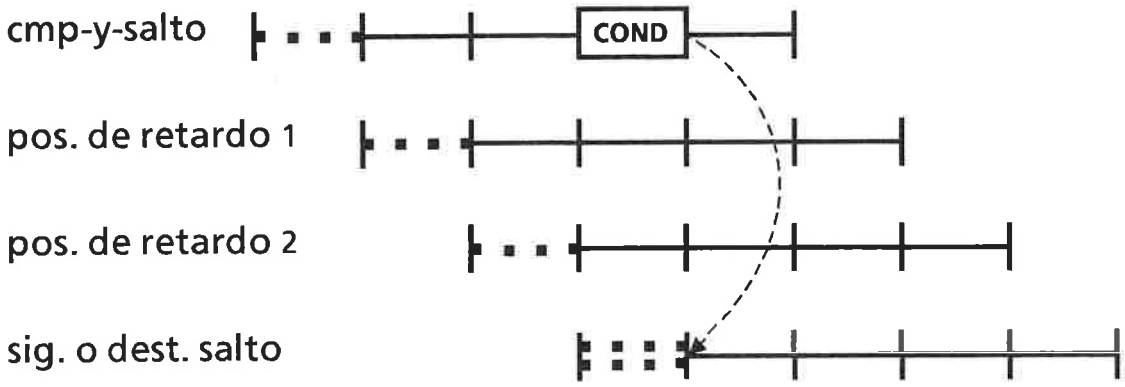


Figura 2.18: Ejecución de una instrucción de salto mediante el mecanismo ESCO en arquitecturas sin códigos de condición.

Puede observarse que con este esquema el coste de una instrucción de salto es el mismo que en arquitecturas con códigos de condición. Ahora bién, hay que tener presente que en algunas ocasiones, en arquitecturas con códigos de condición no es necesario utilizar una instrucción de forma exclusiva para realizar la comparación, sino que el test del salto puede realizarse sobre el resultado de alguna operación aritmética que realiza algún cálculo útil para el algoritmo.

Un inconveniente adicional de este esquema puede ser la dificultad de codificar en una única instrucción toda la información necesaria para realizar la evaluación de la condición y el cálculo de la dirección destino utilizando el mismo número de bits que en el resto de instrucciones.

Por otra parte, con este esquema se requiere un ancho de banda de memoria menor que en el caso de tener instrucciones distintas para la comparación y el cálculo de la dirección destino.

En [Kate85] Katevenis se propuso una técnica para ejecutar las instrucciones de salto basada en una memoria de instrucciones de dos puertos ("Fast compare-and-branch"). Con ella se conseguía ejecutar las instruciones de comparación y salto con el mismo coste que con el esquema presentado en este apartado.

El mecanismo ESCO tiene varias ventajas sobre el esquema de Katevenis. La primera es que nuestro mecanismo puede utilizarse con una memoria de un único puerto lo que simplifica su implementación. Este hecho puede ser importante cuando la memoria cache de instrucciones se implementa en el mismo chip que la CPU, ya que en principio, una memoria de un puerto requiere un área menor, o lo que es lo mismo, dada un área determinada, la memoria de un puerto puede ser mayor con lo que se consigue una mayor tasa de aciertos.

Por otra parte, ambos esquemas disponen de un ancho de banda de dos instrucciones por ciclo. Sin embargo, en el esquema de Katevenis este ancho de banda sólo es utilizado de forma efectiva un ciclo durante la ejecución de un BBD. El resto de ciclos, aunque accede a dos instrucciones, una de ellas es la misma que había accedido en el ciclo anterior. Con nuestro esquema, el ancho de banda de memoria se utiliza al máximo, lo que nos permite hacer una prebúsqueda mucho más profunda por ambas ramas del salto. Esto puede tener ventajas considerables cuando consideremos el efecto de la latencia de memoria externa en caso de producirse un fallo en memoria cache.

En tercer lugar, el mecanismo propuesto por Katevenis no es fácilmente generalizable a aquitecturas con códigos de condición.

El coste de una instrucción de comparación-y-salto sería todavía menor si el mecanismo ESCO permitiera ejecutar la comparación en paralelo con el resto de instrucciones. Ello podría conseguirse utilizando la técnica propuesta en [CLlG87].

CAPITULO 3

Decisiones de diseño

Los objetivos en el diseño de un procesador son obtener el máximo rendimiento con el mínimo coste. Generalmente, coste y rendimiento no son variables independientes, sino que para conseguir un aumento del segundo es preciso incrementar el primero. En este capítulo se evalúan diversas alternativas en el diseño del mecanismo ESCO. Para cada una de ellas se mide su rendimiento y se analiza el coste que supondría su implementación.

En primer lugar se describen los programs de prueba utilizados para evaluar el rendimiento del sistema así como la metodología utilizada para la obtención de medidas. A continuación se evalúa el mecanismo ESCO con varias alternativas para los siguientes parámetros de diseño:

- Longitud de las colas de instrucciones.
- Instante de la decisión de salto.
- Organización de la memoria de un puerto.
- Tratamiento de los saltos incondicionales.
- Tratamiento de las instrucciones de llamada y retorno de subrutina.
- Organización de la prebúsqueda en ESCO-1.
- Cálculo de las direcciones de prebúsqueda en ESCO-2.

3.1. PROGRAMAS DE PRUEBA

Las medidas de rendimiento presentadas en este trabajo se han obtenido utilizando un conjunto de programas de prueba. Como lenguaje máquina hemos escogido el definido en la arquitectura del procesador RISC II. Por lo tanto, estas aplicaciones inicialmente escritas en lenguaje C, han sido compiladas para obtener el código correspondiente a este lenguaje máquina. Los programas de prueba utilizados son los siguientes.

LEX: Generador de analizadores léxicos. 37 Kbytes de código, 9 millones de instrucciones ejecutadas.

NROFF: Formateador de textos. 56 Kbyte de código. 12 millones de instrucciones ejecutadas.

PCC: Compilador de C. 99 Kbyte de código. 21 millones de instrucciones ejecutadas.

YACC: Generador de analizadores sintácticos y semánticos de gramáticas LR(1). 33 Kbytes de código, 42 millones de instrucciones ejecutadas.

Todas estos programas está orientados a procesamiento simbólico. Normalmente, la frecuencia de saltos en este tipo de programas es mayor que en las aplicaciones de cálculo, por lo tanto, las penalizaciones debidas a los saltos tendrán un peso mayor en el rendimiento del procesador. En consecuencia, para la mayoría de aplicaciones númericas la eficiencia del mecanismo ESCO, así como la de otros muchos mecanismos, sería mayor que la obtenida para estos programas de prueba.

3.2. METODOLOGIA PARA LA OBTENCION DE MEDIDAS

Nuestro objetivo en la evaluación de diferentes esquemas para el tratamiento de los saltos es obtener el rendimiento del procesador y una estimación del coste que supondría la implementación de dicho mecanismo.

El rendimiento del procesador para un determinado programa es función de las características del propio programa además del tipo de procesador en cuestión.

El tipo de procesador que asumimos en este trabajo ejecuta un instrucción por ciclo si no se produce ningún conflicto entre instrucciones. Por lo tanto para calcular su eficiencia deberemos medir el número medio de conflictos por instrucción ejecutada. Del conjunto de conflictos que ocurren durante la ejecución de un programa nos interesan únicamente aquellos que se deben a las instrucciones de salto. Por lo tanto vamos a suponer que todo el resto de conflictos son resueltos mediante algún determinado mecanismo.

Para obtener el número de conflictos ocurridos durante la ejecución de un programa tenemos básicamente dos alternativas, que han sido introducidas en el capítulo anterior. La primera es modelar el sistema mediante un proceso de Markov. La segunda es simular el comportamiento del mecanismo. En el caso del mecanismo ESCO, dicha simulación, tal como vimos en el anterior capítulo es muy sencilla de realizar ya que, utilizando los modelos desarrollados en el capítulo 2, puede llevarse a cabo a nivel de BBD con una cantidad mínima de cálculo. En la figura 3.1 se muestra el algoritmo básico utilizado para realizar dicha simulación.

Ambos métodos de obtención de medidas necesitan conocer cierta información sobre las características de los programas evaluados. Esta información, en el caso de la simulación consiste en una traza de los bloques básicos ejecutados, junto con el tamaño de cada bloque y el tipo de salto que contiene. Si utilizamos un proceso de Markov para modelizar el comportamiento del sistema, necesitamos conocer la función de densisdad de probabilidad de la longitud de un BBD, la probabilidad de que un salto sea efectivo y la probabilidad de que un salto sea relativo al PC. Estos tres parámetros se pueden obtener fácilmente a partir de una traza de bloques básicos ejecutados, por lo tanto, para ambos mecanismos necesitamos la misma información acerca de los programas que pretendemos evaluar. Esta información ha sido obtenida mediante simulación utilizando las técnicas propuestas en [CoLl87a].

3.3. EVALUACION DEL MODELO BASICO

En este apartado vamos a evaluar el rendimiento del mecanismo ESCO con las características descritas en el capítulo anterior. Estos resultados serán comparados posteriormente con los obtenidos para otras alternativas de diseño. Para disponer de resultados que nos sirvan como punto de referencia con que

```
Número_de_saltos : = 0
Coste_total : = 0
PI : = 1
Obtener_BBD (Longitud, Tipo_salto)

mientras quedan BBD hacer
    Modelo_ESCO (Longitud, Tipo_salto, PI, Coste_del_salto_actual, PF)
    Coste_total : = Coste_total + Coste_del_salto_actual
    Número_de_saltos : = Número_de_saltos + 1
    PI : = PF
    Obtener_BBD (Longitud, Tipo_salto)
f.mientras

Coste_medio_de_un_salto : = Coste_total / Número_de_saltos


Modelo_ESCO (Longitud, Tipo_salto, PI, Coste_del_salto_actual, PF)

    Calcula Coste_del_salto_actual y PF en función de Longitud, Tipo_salto y PI
    utilizando los modelos de las figuras 2.16 y 2.17
```

Figura 3.1: Algoritmo para al obtención de medidas de rendimiento mediante simulación

comparar la eficiencia del mecanismo ESCO, vamos a evaluar también el rendimiento de otros mecanismos presentados en el capítulo 1. El coste hardware necesario para conseguir el aumento de rendimiento que aporta el mecanismo ESCO será analizado en detalle en un capítulo posterior, donde se describe la implementación del mecanismo.

A lo largo de este trabajo vamos a utilizar dos formas de expresar el rendimiento de un determinado mecanismo. Una de ellas es mediante el número

medio de ciclos necesarios para ejecutar una instrucción de salto. Alternativamente, expresaremos el rendimiento como el número de *instrucciones útiles* ejecutadas por unidad de tiempo. Consideramos instrucciones útiles a todas las instrucciones con excepción de los saltos y las instrucciones de no operación.

3.3.1. Medidas obtenidas a partir del modelo

El rendimiento del mecanismo ESCO puede calcularse de forma inmediata a partir de los resultados obtenidos mediante el modelo de Markov (tabla 2.3) y utilizando los modelos analíticos de las figuras 2.16 y 2.17. El coste medio de una instrucción de salto es igual a las siguientes expresiones.

ESCO-1

$$N_{ciclos/salto} = [f_L(1) + f_L(1)Prob(PI=1)(1\text{-}PE) + f_L(2)\,Prob(PI=1) + f_L(2)\,Prob(PI=2)(1\text{-}PE) + f_L(3)\,Prob(PI=1)(1\text{-}PE)]\,PREL + 2(1 - PREL)$$

ESCO-2

$$N_{ciclos/salto} = [f_L(1) + f_L(2)\,Prob(PI=1)]\,PREL + 2(1 - PREL)$$

Por lo tanto, el número total de ciclos perdidos debidos a las instrucciones de salto es

$$N_{ciclos} = N_{ciclos/salto}\,N_{saltos}$$

N_{saltos}: *Número de instrucciones de salto en el programa = Número de BBD*

La eficiencia del mecanismo medida como el número de instrucciones útiles ejecutadas por ciclo de procesador será:

$$Eficiencia = (N_{inst} - N_{saltos}) / (N_{inst} - N_{saltos} + N_{ciclos})$$

N_{inst} : *Número de instrucciones ejecutadas (incluyendo saltos)*

Mediante estas expresiones se ha medido la eficiencia de ESCO-1 y ESCO-2 para cada uno de los programas de prueba. Los resultados se muestran en la tabla 3.1.

	$f_L(1)$	$f_L(2)$	$f_L(3)$	PE	PREL
LEX	0.005	0.059	0.395	0.887	0.9996
NROFF	0.038	0.221	0.277	0.686	0.998
PCC	0.067	0.349	0.198	0.706	0.971
YACC	0.006	0.147	0.083	0.748	1.0

	ESCO-1			ESCO-2	
	Pr(PI = 1)	Pr(PI = 2)	eficiencia (Inst. útil /ciclo)	Pr(PI = 1)	eficiencia (Inst. útil /ciclo)
LEX	0.396	0.410	0.985	0.500	0.989
NROFF	0.326	0.370	0.952	0.388	0.964
PCC	0.375	0.405	0.900	0.516	0.906
YACC	0.219	0.272	0.987	0.175	0.993

$f_L(i)$: Probabilidad de que un BBD tenga *i* instrucciones

PE: Probabilidad de que un salto sea efectivo

PREL: Probabilidad de que un salto sea relativo al PC (los retornos de subrutina se contabilizan como relativos al PC, ya que la UI conoce la dirección destino del salto)

Pr(PI = x): Probabilidad de que al iniciarse la ejecución de un BBD el número de instrucciones prebuscadas sea ***x***.

Tabla 3.1: Eficiencia de los mecanismos ESCO-1 y ESCO-2 medida en instrucciones útiles ejecutadas por ciclo de procesador. Los resultados han sido obtenidos a partir de una cadena de Markov.

De las expresiones que determinan el rendimiento del mecanismo ESCO podemos deducir que el coste de un salto depende básicamente de la frecuencia de ejecución de dichas instrucciones, o lo que es lo mismo, de la longitud de los BBD. Ahora bién, esto no implica que dados dos programas, el mecanismo va a ser necesariamente más eficiente en aquel que tenga BBD mayores.

Tomemos como ejemplo ESCO-2. Cuanto mayor son los BBD podemos asegurar que mayor será el número de instrucciones prebuscadas al inicio de un BBD, lo que implica que el valor de $Pr(PI=1)$ va a ser menor. Sin embargo, puede ocurrir que en media los BBD tengan un tamaño mayor pero que exista una proporción más elevada de BBD con menos de tres instrucciones. Ya que $f_L(1)$ y $f_L(2)$ tienen un peso considerable en el coste medio de un salto, puede ocurrir que el rendimiento del mecanismo ESCO sea menor para el programa que tiene BBD mayores.

Un ejemplo de esto podemos encontrarlo analizando los resultados de los programas LEX y NROFF en la tabla 3.1. Los BBD del primero son algo menores que los del segundo (4.22 frente a 4.37 instrucciones en media), por lo que el valor de $Pr(PI=1)$ es mayor para el programa LEX. Sin embargo dado que el programa NROFF tiene una mayor proporción de BBD con tamaño menor que tres instrucciones su rendimiento es menor.

La información acerca del comportamiento del mecanismo ESCO que hemos obtenido a partir de los modelos analíticos puede ser de gran utilidad para el compilador. Este puede mejorar el rendimiento del mecanismo ESCO si mediante una reordenación de código consigue agrandar el tamaño de los BBD. Ahora bién, este aumento de tamaño nunca debe llevarse a cabo si con ello se incrementa el número de BBD con muy pocas instrucciones. Por otra parte, mediante los modelos análiticos también se pone de manifiesto que donde más efectivo va a ser un aumento de tamaño es en los BBD con muy pocas instrucciones.

El rendimiento peor se obtiene para el programa PCC, ya es el que tiene BBD menores y además una mayor proporción de BBD con muy pocas instrucciones.

3.3.2. Validación del modelo

Para demostrar la validez de los modelos desarrollados para caracterizar el mecanismo ESCO, hemos evaluado los mismos programas de prueba utilizados en el apartado anterior mediante técnicas de simulación. Los resultados se muestran en la tabla 3.2.

inst. útiles / ciclo	LEX	NROFF	PCC	YACC	Media
ESCO-1	0.987	0.950	0.896	0.984	0.954
ESCO-2	0.991	0.961	0.907	0.990	0.962

Tabla 3.2: Eficiencia de ESCO-1 y ESCO-2 obenida mediante simulación

Puede observarse que la discrepancia entre los resultados de la tabla 3.2 y los obtenidos mediante el modelo global (tabla 3.1) es menor que el 0.5% en todos los programas de prueba, por lo que podemos concluir que el modelo basado en un proceso de Markov es un buena representación de la realidad.

3.3.3. Rendimiento de otros mecanismos

En la tabla 3.3 puede observarse el rendimiento de otros mecanismos utilizados con frecuencia en procesadores segmentados. Estos mecanismos han sido descritos en el capítulo 1, a excepción del denominado "Salto sin retardo", que es un esquema hipotético en el que se ha supuesto que el coste de cualquier salto es un ciclo. La eficiencia de este mecanismo es una cota superior para los mecanismos que resuelven los saltos en la unidad de ejecución. En la figura 3.2 se representa gráficamente el coste de una instrucción de salto para cada uno de estos mecanismos junto con el coste obtenido para ESCO-1 y ESCO-2.

inst. útiles / ciclo	LEX	NROFF	PCC	YACC	Media
SR	0.689	0.713	0.674	0.748	0.706
SRA	0.745	0.708	0.678	0.773	0.726
SRAP	0.756	0.747	0.711	0.793	0.752
MIDS	0.763	0.756	0.713	0.806	0.760
SSR	0.763	0.771	0.741	0.807	0.770

SR: Salto retardado
SRA: Salto retardado con anulación
SRAP: Salto retardado con anulación y preejecución
SSR: Salto sin retardo
MIDS: Memoria de instrucciones destino de salto de 1Kbyte (256 entradas) + espacio para etiquetas

Tabla 3.3: Eficiencia de varios mecanismos frecuentemente utilizados en procesadores segmentados

En el mecanismo denominado "Memoria de instrucciones destino de salto" (MIDS) hemos supuesto que el procesador dispone de una memoria de 256 entradas, y que cada una de ellas contiene la primera instrucción destino de un salto (4 bytes). El tipo de mapeo es directo, y el acceso a ella se realiza mediante la dirección de la instrucción de salto. Para guardar las etiquetas será preciso una memoria adicional de tamaño cercano a 1 Kbyte.

El esquema de ejecución de un salto mediante el mecanismo MIDS se representa en la figura 3.3. En el mismo ciclo en que se va a buscar a la memoria de instrucciones la instrucción de salto, se accede a la MIDS para conseguir la primera instrucción del destino de este salto. Paralelamente se está calculando la dirección de la instrucción que sigue a la de salto. En este mismo ciclo la ALU está

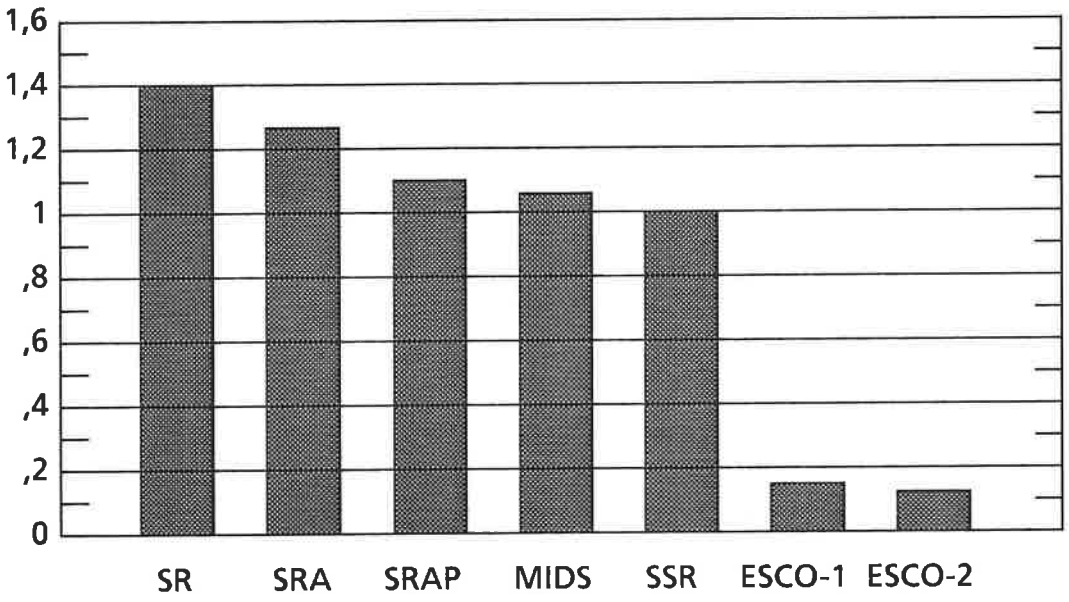
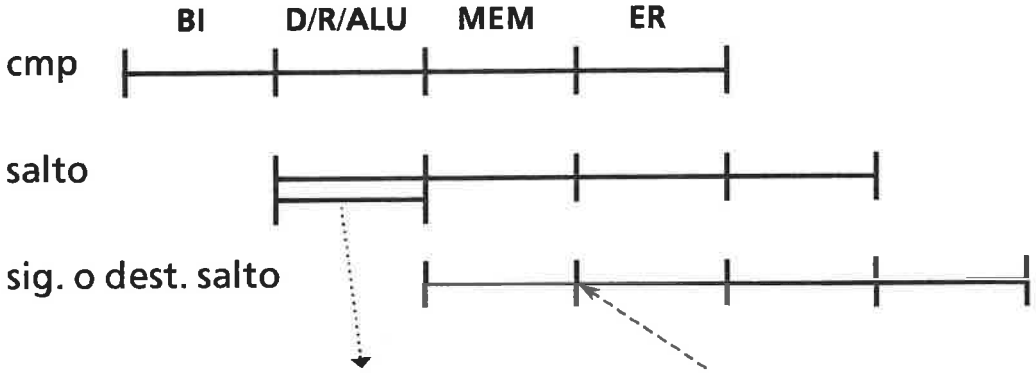


Figura 3.2: Coste de una instrucción de salto en ESCO-1, ESCO-2 y otros mecanismos frecuentemente utilizados en procesadores segmentados

evaluando la condición de salto. En el ciclo siguiente, dependiendo si en el acceso a la MIDS ha habido acierto o fracaso, la instrucción de salto calcula en la ALU la dirección de salto+1 o la dirección de salto. Paralelalamente se está buscando la instrucción que sigue al salto en secuencia. En este mismo ciclo se realiza el test sobre los códigos de condición y se determina si el salto debe ser efectivo o no. Al final de este ciclo, si el salto no es efectivo se continúa ejecutando la instrucción que sigue al salto en secuencia. Si el salto es efectivo y ha habido acierto en la MIDS se continúa ejecutando la instrucción obtenida de esta memoria. Si el salto es efectivo y ha habido un fallo en el acceso a la MIDS, se envía a ejecutar una NOP mientras se va a buscar la primera instrucción del destino del salto.

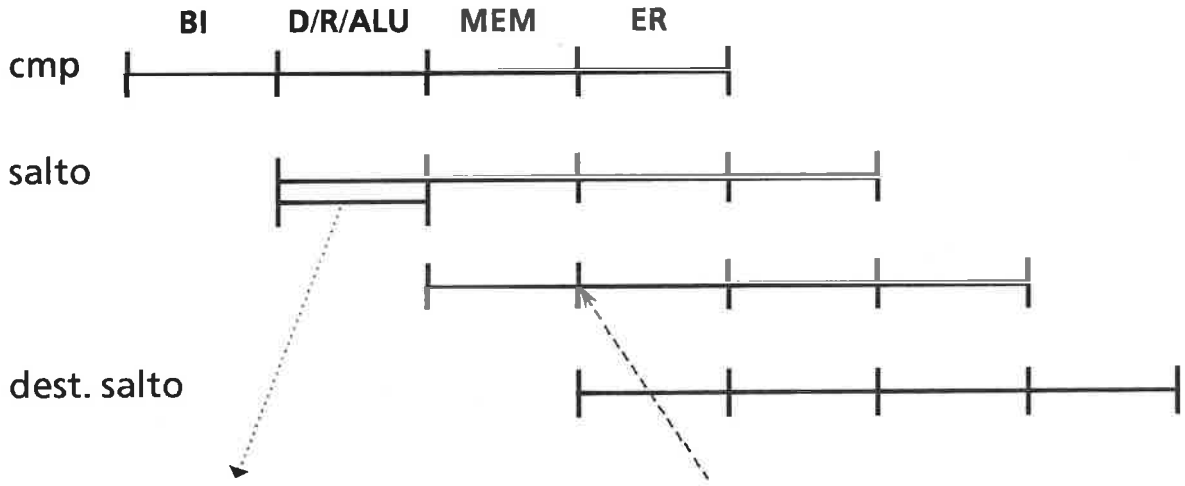
El coste de una instrucción de salto depende de si su instrucción destino se encuentra en la MIDS, y será igual a



Acceso a la MIDS para obtener la primera instrucción del destino del salto

Selección entre inst. sig. o inst. dest del salto

a) Salto no efectivo o salto efectivo con acierto en la MIDS



Acceso a la MIDS para obtener la primera instrucción del destino del salto

Se envía a ejecutar una NOP

b) Salto efectivo con fallo en la MIDS

Figura 3.3: Ejecución de una instrucción de salto mediante el mecanismo "Memoria de instrucciones destino de salto"

- 2 ciclos si el salto es efectivo y se produce un fallo en el acceso a la MIDS.
- 1 ciclo en cualquier otro caso.

El rendimiento del mecanismo MIDS depende de la localidad de los programas y del tamaño de la memoria de instrucciones destino de salto. Dado que hemos supuesto un número considerable de entradas, la tasa de aciertos de esta memoria es bastante elevada con lo que se consigue reducir el coste de las instrucciones de salto a un valor cercano a un ciclo.

3.4. TAMAÑO DE LAS COLAS DE INSTRUCCIONES

En el estudio sobre el comportamiento del mecanismo ESCO hemos asumido que la UI tenía capacidad ilimitada para almacenar cualquier número de instrucciones prebuscadas. En este apartado vamos a analizar el efecto de limitar esta capacidad de almacenamiento.

Para evaluar el comportamiento de ESCO vamos a utilizar el modelo global anteriormente descrito. La única modificación a realizar será en la matriz de probabilidades de transición. Si P es la matriz anteriormente caculada, y M es el número máximo de instrucciones que podemos almacenar de cada una de las dos ramas del salto, la nueva matriz de probabilidades de transición P' podemos calcularla de la manera siguiente.

$$p'_{i,j} = p_{i,j} \qquad si\ i \leq M\ y\ j < M$$

$$p'_{i,j} = 0 \qquad si\ i > M\ o\ j > M$$

$$p'_{i,M} = \sum_{k=M}^{\infty} p_{i,k} = 1 - \sum_{k=1}^{M-1} p_{i,k} \qquad si\ i \leq M$$

A partir de la matriz P' hemos evaluado el rendimiento de ESCO-1 y ESCO-2 para varios valores de M. Los resultados se muestras en la tabla 3.4.

Observando los resultados de la tabla 3.4 se pone de manifiesto que limitando a 4 el número de instrucciones que puede almacenar la UI de cada una de las ramas, se consigue una eficiencia prácticamente igual que teniendo una

inst. útiles/ciclo	ESCO-1			
	M = 2	**M = 4**	**M = 6**	**M = ∞**
LEX	0.983	0.985	0.985	0.985
NROFF	0.944	0.951	0.952	0.952
PCC	0.892	0.899	0.900	0.900
YACC	0.981	0.987	0.987	0.987

inst. útiles/ciclo	ESCO-2			
	M = 2	**M = 4**	**M = 6**	**M = ∞**
LEX	0.987	0.989	0.989	0.989
NROFF	0.958	0.964	0.964	0.964
PCC	0.900	0.906	0.906	0.906
YACC	0.990	0.992	0.993	0.993

Tabla 3.4: Eficiencia de ESCO-1 y ESCO-2 limitando a M el número de instrucciones prebuscadas por cada rama

capacidad infinita de almacenamiento, y que limitando dicho número a 2 la pérdida de eficiencia es inferior al 1%.

3.5. INSTANTE DE LA DECISION DE SALTO

En los dos esquemas del mecanismo ESCO, hemos supuesto que la decisión de salto, que implica la realización de un test sobre los códigos de condición y la selección de una de las dos ramas prebuscadas, podía llevarse a cabo como muy pronto el ciclo siguiente al de acceso a dicho salto.

Una alternativa de funcionamiento sería que al final del mismo ciclo en que la instrucción de salto es buscada la UI pudiera tomar la decisión de este salto, si en ese mismo ciclo se había evaluado la condición de salto.

La implementación de esta alternativa llevaría asociada pequeñas modificaciones en la parte de control del procesador, sin embargo, su mayor inconveniente es que puede causar un aumento del tiempo de ciclo y por consiguiente, aunque sea mayor el número de instrucciones útiles ejecutadas por ciclo, puede resultar que el tiempo total de ejecución de un programa no disminuya, e incluso que llegue a aumentar.

El tiempo de ciclo de procesador viene determinado por la operación más lenta que se desea realizar en un ciclo. En los casos en que el tiempo de ciclo viene determinado por el tiempo de acceso a memoria (lo que es bastante frecuente) la alternativa que se propone en este apartado significaría un aumento del tiempo de ciclo. Esto implica un aumento del tiempo de ejecución de todas las instrucciones y por lo tanto, aunque los saltos ocasionaran menos ciclos de retardo, el rendimiento del procesador podría llegar a ser menor.

En las figuras 3.4 y 3.5 se muestran los modelos de ejecución de un BBD correspondientes a esta alternativa, tanto para ESCO-1 como ESCO-2.

A partir de estos modelos se ha evaluado la eficiencia del mecanismo ESCO. Los resultados se muestran en la tabla 3.5. Los resultados de esta tabla muestran que el incremento de instrucciones útiles procesadas por ciclo es en media del orden del 0.8%. Dado que aproximadamente una de cada 4.5 instrucciones ejecutadas es un salto, si esta alternativa supone un aumento del tiempo de ciclo mayor que el 0.23% ($0.23 = 0.8 / (4.5 - 1)$) el resultado será una disminución de la eficiencia del mecanismo. Por lo tanto sería preciso realizar un análisis detallado de los tiempos de respuesta de las diferentes partes del procesador para

	SALTO EFECTIVO			
	L = 1	L = 2 y PI = 1	(L = 2 y PI > 1) ó L > 2	
			PI < L	PI ≧ L
PF	2-IMPAR	2-IMPAR	(L + PI)/4x2-IMPAR	L/2x2-IMPAR
Coste del salto	1	1	0	0

	SALTO NO EFECTIVO					
	L = 1 y PI = 1	L = 1 y PI > 1	L = 2 y PI = 2	L = 3 y PI = 1	(L = 2 y PI ≠ 2) ó (L = 3 y PI > 1) ó L > 3	
					PI < L	PI ≧ L
PF	2	PI-1	2	2	(L + PI-2)/4x2 + (L-PI) mod 2	PI-2 + L mod 2
Coste del salto	2	1	1	1	0	0

Figura 3.4: Modelo de la ejecución de un BBD con el mecanismo ESCO-1. La UI puede tomar un decisión de salto en el mismo ciclo en que éste es accedido (las expresiones deben evaluarse utilizando aritmética entera).

determinar si realmente es beneficiosa esta alternativa. En cualquier caso este beneficio será en media como máximo del 0.8%, lo que representa una ganancia poco significativa.

3.6. ORGANIZACION DE LA MEMORIA DE UN PUERTO

Una memoria de un puerto es un sistema cuya entrada es una dirección y cuya salida es una determinada cantidad de información almacenada a partir de esta dirección. A esta información leida en un acceso la denominaremos *secuencia*.

	SALTO EFECTIVO				SALTO NO EFECTIVO		
	L = 1	L = 2 y PI = 1	(L = 2 y PI > 1) ó L > 2		L = 1	L > 1	
			PI < L	PI ≧ L		PI < L	PI ≧ L
PF	1	1	(L + PI)/2-1	L-1	PI	(L+PI+1)/2-1	PI-1
Coste del salto	1	1	0	0	1	0	0

Figura 3.5: Modelo de la ejecución de un BBD con el mecanismo ESCO-2. La UI puede tomar un decisión de salto en el mismo ciclo en que éste es accedido (las expresiones deben evaluarse utilizando aritmética entera).

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%aumen.
ESCO-1	0.988	0.961	0.911	0.988	0.962	0.797
ESCO-2	0.994	0.969	0.920	0.991	0.968	0.627

Tabla 3.5: Eficiencia del mecanismo ESCO en el caso de que la decisión de salto pueda tomarse en el mismo ciclo en que éste es accedido

Por lo que concierne a nuestro mecanismo, desde el punto de vista funcional existen dos organizaciones diferentes de una memoria de un puerto: a) que sea direccionable a nivel de instrucción o b) que sea direccionable a nivel de bloque de instrucciones, siendo el tamaño de un bloque igual al número de instrucciones de una secuencia.

En el primer caso podremos acceder a cualquier secuencia de instrucciones, mientras que en el segundo únicamente a aquellas secuencias en las que la dirección de la primera instrucción sea múltiplo del tamaño de un bloque.

El coste de implementación es sensiblemente inferior en la alternativa (b). Sin embargo, la alternativa (a) se adapta mejor a las necesidades de la UI ya que cada vez que se inicia el acceso a una nueva rama, la dirección de inicio de esta rama no tiene por que ser múltiplo del tamaño de un bloque.

La alternativa (b) ha sido evaluada anteriormente en el modelo básico del mecanismo ESCO-1, por lo que en este apartado vamos a obtener medidas acerca de la eficiencia de la alternativa (a).

El modelo de ejecución de un BBD en ESCO-1 con este tipo de memoria será el mismo que el expuesto en la figura 2.17 dando a la variable "es impar" (IMPAR) el valor cero para todos los casos. La tabla 3.6 muestra el rendimiento del sistema con esta organización alternativa de la memoria. Como puede apreciarse, la ganancia en eficiencia es muy poca y teniendo en cuenta que ello supone un coste adicional en el diseño e implementación del sistema de memoria, creemos que el mejor compromiso entre eficiencia y coste es utilizar una memoria con accesos únicamente a secuencias que forman un bloque.

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%aumen.
ESCO-1	0.993	0.973	0.916	0.994	0.969	1.550

Tabla 3.6: Eficiencia de ESCO-1 con una memoria de un puerto capaz de servir en cada acceso dos instrucciones consecutivas cualesquiera

3.7. SALTOS INCONDICIONALES

En el modelo básico, los saltos incondicionales (incluyendo llamadas y retornos de subrutina) son tratados de la misma forma que los saltos condicionales. Ello implica que cuando la UI encuentra una de estas instrucciones, prebusca instrucciones de ambas ramas del salto. Evidentemente el salto va a ser efectivo, por lo que se sabe de antemano que no tiene ninguna utilidad prebuscar las instrucciones que siguen al salto en secuencia.

La UI podría tratar los salto incondicionales de forma diferente al resto de saltos para que la prebúsqueda de instrucciones se realizara únicamente por la rama destino del salto. Esto lleva asociado consigo dos ventajas: a) Un incremento del número de instrucciones prebuscadas del siguiente BBD, y b) una menor polución de la memoria cache, ya que disminuye el número de instrucciones prebuscadas que no van a ser utilizadas de forma inmediata.

Ya que todavía no hemos analizado el comportamiento del mecanismo ESCO con una memoria cache de instrucciones, en este apartado nos limitaremos a evaluar el rendimiento del procesador sin tener en cuenta la memoria cache. Más adelante se evaluará el efecto de esta alternativa con una memoria cache de instrucciones.

En la figura 3.6 se muestra el modelo correspondiente a la ejecución de un salto incondicional con este nuevo esquema. El resto de saltos son tratados de la misma forma que en el modelo básico, por lo tanto su modelo de ejecución será el que desarrollamos en el capítulo anterior (figura 2.17).

En la tabla 3.7 se muestra la eficiencia de ESCO-1 y ESCO-2 con esta nueva alternativa. En esta tabla se pone de manifiesto que el aumento de rendimiento es prácticamente despreciable, sin embargo, hay que tener en cuenta las ganancias adicionales que puede traer consigo cuando consideremos un sistema con una memoria cache de instrucciones.

	ESCO-1				ESCO-2			
	L = 1	L = 2 y PI = 1	(L = 2 y PI > 1) ó L > 2		L = 1	L = 2 y PI = 1	(L = 2 y PI > 1) ó L > 2	
			PI < L	PI ≧ L			PI < L	PI ≧ L
PF	2-IMPAR	2-IMPAR	(L + PI)/2x2-2-IMPAR	2L-2-IMPAR	1	1	(L+PI)/2x2-3	2L-3
Coste del salto	1	1	0	0	1	1	0	0

Figura 3.6: Modelo de ejecución de un salto incondicional con el mecanismo ESCO limitando la prebúsqueda a la rama destino del salto (las expresiones deben evaluarse utilizando aritmética entera).

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%aumen.
ESCO-1	0.987	0.953	0.899	0.984	0.956	0.189
ESCO-2	0.991	0.962	0.908	0.990	0.963	0.040

Tabla 3.7: Eficiencia del mecanismo ESCO limitando la prebúsqueda a la rama destino en los saltos incondicionales

3.8. LLAMADAS Y RETORNOS DE SUBRUTINA

Como ya se ha comentado en el capítulo 2, tenemos básicamente dos posibilidades para tratar las instrucciones de llamada y retorno de subrutina: a) que las gestione completamente la UI y por lo tanto que sean transparentes a la UE o b) que la UE se encargue de gestionar la dirección de retorno. La primera alternativa ha sido ya evaluada por lo que en este apartado vamos a obtener resultados de la segunda.

Estas instrucciones son una parte significativa del total de saltos de algunos programas. En el caso de nuestros programas de prueba, constituyen entre un 1.6% y un 21.2% del total de saltos (tabla 2.1), por lo que es de esperar un aumento significativo de eficiencia cuando estas instrucciones son tratadas por completo en la UI. Sin embargo, esta ganancia tiene un coste considerable. Por una parte, la UI necesita implementar una estructura tipo pila para gestionar las direcciones de retorno de subrutina. Además, debe existir algún mecanismo que permita salvar y restaurar el contenido de esta pila en los cambios de contexto y en los casos en que se exeeceda su capacidad.

Si es la UE la que gestiona las direcciones de retorno, cuando la UI se encuentra con una llamada a subrutina, al igual que antes, prebusca instrucciones del siguiente BBD. Sin embargo, a su debido tiempo, envía esta instrucción a la UE para que sea ésta la que guarde la dirección de retorno, por ejemplo, vamos a suponer en el banco de registros, como es el caso del RISC II. Esto supondrá un ciclo adicional para ejecutar esta instrucción.

Con este nuevo esquema, cuando la UI encuentra un retorno de subrutina no puede prebuscar instrucciones de la rama destino del salto, ya que no conoce la dirección destino. Esta instrucción será enviada a la UE y en este caso su coste será de dos ciclos. En el primero se calcula la dirección de retorno, y en el segundo se accede a alguna instrucción del nuevo BBD a ejecutar (suponemos que los retornos son incondicionales).

Para evaluar esta alternativa debemos modificar los modelos analíticos para que contemplen las instrucciones de llamada y retorno de subrutina como casos

particulares. La modelización de este tipo de intrucciones es la siguiente (las expresiones deben evaluarse con aritmética entera).

ESCO-1

- *Llamada a subrutina*

  Coste del salto = 1 ciclo
  Si PI < L entonces PF = (PI + L + 2) / 4 x 2 - IMPAR
  sino PF = (L + 1) / 2 x 2 - IMPAR

- *Retorno de subrutina*

  Coste del salto = 2 ciclos
  PF = 2-IMPAR

ESCO-2

- *Llamada a subrutina*

  Coste del salto = 1 ciclo
  Si PI < L entonces PF = (PI + L) / 2
  sino PF = L

- *Retorno de subrutina*

  Coste del salto = 2 ciclos
  PF = 2

En la tabla 3.8 puede observarse la eficiencia del sistema al implementar esta nueva alternativa. Como era de esperar, el rendimiento del procesador es menor, sobre todo en los programas NROFF y PCC, en los que la proporción de llamadas y retornos de subrutina respecto al total de saltos es mucho mayor (tabla 2.1). Sin embargo, las ventajas anteriormente comentadas de esta alternativa podrían justificar su implementación y por lo tanto creemos que es una opción a tener en cuenta.

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%pérdida
ESCO-1	0.981	0.888	0.847	0.977	0.923	3.229
ESCO-2	0.985	0.895	0.859	0.983	0.930	3.335

Tabla 3.8: Eficiencia de ESCO-1 Y ESCO-2 cuando la direcciones de retorno en llamadas y retornos de subrutina es gestionada por la UE

3.9. ESCO-1: ORGANIZACION DE LA PREBUSQUEDA

Cuando la UI dispone de una memoria de un único puerto, la prebúsqueda por ambas ramas del salto no puede realizarse en paralelo, por lo tanto, habrá que decidir a cual de las dos ramas se le da más prioridad.

Cuando en un BBD hay un número considerable de instrucciones, cualquier alternativa es igual de válida, ya que existe tiempo suficiente para prebuscar varias instrucciones de cada una de las ramas. Sin embargo, en un BBD con muy pocas instrucciones, esta decisión puede tener un impacto considerable en el rendimiento del mecanismo.

Por ejemplo, en un BBD con tres instrucciones al que se llega con PI=1, la UE tarda dos ciclos en ejecutar sus dos instrucciones útiles. Durante el primer ciclo, la UI accede a las dos instrucciones del propio BBD que faltaban por buscar. En el segundo ciclo tenemos dos opciones: a) acceder a las dos primeras instrucciones situadas a continuación del salto o b) acceder a las dos primeras de la rama destino del salto. Si la decisión tomada coincide con el posterior resultado del salto, el coste del salto será cero. En caso contrario, el coste del salto será un ciclo.

Necesitamos, por lo tanto, predecir el resultado de los saltos. Para ello podemos utilizar cualquiera de las técnicas descritas en el capítulo 1.

La predicción más simple de implementar es la utilizada en el modelo básico, a la que vamos a denominar *predicción estática constante*, donde para todo salto la predicción es de salto efectivo. Esta predicción se corresponde con el análisis presentado en la tabla 2.2 donde se observa que el número de saltos efectivos es bastante mayor al de no efectivos.

Una alternativa sería utilizar *predicción estática variable*, en la que es el compilador el que realiza la predicción mediante un bit en la instrucción de salto. Ello supone un incremento del coste hardware de la UI ya que habrá que implementar los circuitos necesarios para analizar este bit y en función de su valor dar prioridad a una u otra rama.

Para estimar el aumento de rendimiento respecto a la alternativa anterior podemos remitirnos a los resultados presentados en [DiMc87], donde se obtiene que mediante predicción estática variable se consigue predecir correctamente entre el 85 y 90% del total de saltos. Con predicción estática constante se consigue realizar una predicción correcta en el 76% de los casos. En consecuencia la diferencia entre ambos métodos es del orden de un 14% en la proporción de aciertos. Ahora bién, de este 14% de casos, no en todos ellos se consigue disminuir el coste del salto, ya que como se ha comentado anteriormente, incluso con una predicción errónea el salto puede llegarse a ejecutar con coste cero. Los saltos para los que se consigue disminuir su coste pueden obtenerse de la tabla 2.17. Son los siguientes.

- $L=1$ y $PI=1$
- $L=2$ y $PI=2$
- $L=3$ y $PI=1$

La diferencia entre el coste de un salto al utilizar predicción estática constante o variable es aproximadamente igual a

$$0.14\,[\,f_L(1)\,Pr(PI=1) + f_L(2)\,Pr(PI=2) + f_L(3)\,Pr(PI=1)\,] =$$

$$0.14\,[\,0.029 \times 0.329 + 0.194 \times 0.364 + 0.238 \times 0.329\,] = 0.022 \text{ ciclos}$$

Los valores de $f_L(L)$ y $Pr(PI=x)$ se corresponden con los valores medios de estas funciones para los programas LEX, NROFF, PCC, YACC, y han sido obtenidos a partir de la tabla 3.1.

Puede observarse que el beneficio aportado por esta técnica es apenas significativo por lo que no creemos conveniente complicar el diseño de la UI para implementar la técnica de predicción estática variable.

Con técnicas de predicción dinámica, podríamos realizar una predicción todavía más certera. Si nos remitimos a los resultados presentados en [DiMc87], observamos que la ganancia obtenida con predicción dinámica es poco mayor que la conseguida con predicción estática. Por lo tanto, el rendimiento aportado no justifica su implementación dentro del mecanismo ESCO.

3.10. ESCO-2: CALCULO DE LAS DIRECCIONES DE PREBUSQUEDA

En la implementación del mecanismo ESCO vamos a necesitar de algún circuito que en cada ciclo calcule la dirección o direcciones de los accesos a realizar en el ciclo siguiente. En el caso de ESCO-2 habrá que calcular dos direcciones, por lo tanto necesitamos dos de estos circuitos. Cada uno de ellos, debe ser capaz de sumar una o dos unidades al dato de entrada. Necesitamos sumar una unidad cuando la UI está realizando la prebúsqueda en paralelo por las dos ramas del salto, y dos unidades cuando está accediendo a dos intrucciones consecutivas del propio BBD.

Ahora bién, este tipo de circuito puede no ser suficiente en determinados casos. En concreto, estos casos serán aquellos ciclos en que la UI debe tomar una decisión de salto. Tomemos como ejemplo el ciclo $i+3$ de la figura 2.11. En este ciclo se evalúa la condición de salto. Si al final de éste resulta que el salto debe ser efectivo, las instrucciones a buscar en el ciclo siguiente son $b+2$ y $b+3$. Por el contrario, si el salto resulta no ser efectivo, en el ciclo siguiente habrá que acceder a las instrucciones $a+7$ y $a+8$. Por lo tanto, cada uno de los dos circuitos que calculan las direcciones de las siguientes instrucciones debe sumar una y dos unidades al dato de entrada en cada ciclo. Ello supone que cada uno de estos circuitos consta de dos incrementadores.

Dado que los dos incrementadores sólo se necesitan en los ciclos en que se toma una decisión de salto, podríamos prescindir de uno de ellos y disponer en cada circuito con un único incrementador que dependiendo de una señal de control sumase una o dos unidades al dato de entrada. Ello implica que en el ciclo siguiente al de toma de decisión de un salto, la UI sólo podría acceder a una instrucción del nuevo BBD. Durante este ciclo, un incrementador sumaría a dicha dirección una unidad y el otro dos unidades, de forma que en el ciclo siguiente y sucesivos se pudiera acceder a dos instrucciones.

El modelo analítico correspondiente a esta alternativa se muestra en la figura 3.7 y los resultados de evaluar este esquema se presentan en la tabla 3.9.

	SALTO EFECTIVO					SALTO NO EFECTIVO				
	L = 1	L = 2 y PI = 1	L = 3 y PI = 1	(L = 2 y PI > 1) ó (L = 3 y PI > 1) ó L > 3		L = 1	L = 2 y PI = 1	L = 3 y PI = 1	(L = 2 y PI > 1) ó (L = 3 y PI > 1) ó L > 3	
				PI < L	PI ≧ L				PI < L	PI ≧ L
PF	1	1	1	(L + PI-1)/2-1	L-1	PI	1	2	(L+PI)/2-1	PI-1
Coste del salto	1	1	1	0	0	1	1	1	0	0

Figura 3.7: Modelo de ejecución de un BBD con el mecanismo ESCO-2, simplificando los circuitos de cálculo de las direcciones de las instrucciones a prebuscar (las expresiones deben evaluarse utilizando aritmética entera).

La diferencia de rendimiento respecto al modelo básico de ESCO-2 es considerable, por lo que esta alternativa únicamente tendrá sentido cuando las restricciones de implementación nos impidan disponer de los circuitos con dos incrementadores.

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%pérdida
ESCO-2	0.894	0.915	0.856	0.973	0.910	5.405

Tabla 3.9: Eficiencia de ESCO-2 simplificando los circuitos de cálculo de las direcciones de las instrucciones a prebuscar

3.11. CONCLUSIONES

En este capítulo se han evaluado diversas alternativas de diseño del mecanismo ESCO. Las conclusiones obtenidas del estudio realizado son las siguientes.

1) Longitud de las colas de instrucciones: Con muy poca capacidad (del orden de 4 instruciones) el mecanismo ESCO consigue prácticamente su máximo rendimiento.

2) Instante de la decisión de salto: Una decisión de salto podrá tomarse como muy pronto en el ciclo siguiente al de acceso a dicha instrucción.

3) Organización de la memoria de un puerto: Será direccionable a nivel de bloque de instrucciones.

4) Saltos incondicionales: A menos que su efecto sobre la memoria cache de instrucciones demuestre un beneficio adicional significativo, estas instrucciones serán tratadas de la misma forma que los saltos condicionales.

5) Llamadas y retornos de subrutina: Desde el punto de vista de eficiencia, aunque las ventajas de gestionar estas instrucciones en la UI son significativas, el rendimiento del mecanismo ESCO cuando estas instrucciones son tratadas en la UE podríamos calificarlo de aceptable. Por lo tanto, podemos concluir que si podemos disponer del hardware necesario

para gestionar estas instrucciones en la UI, optaremos por esta alternativa. En caso contrario, el rendimiento del mecanismo sería menor, aunque seguiría estando dentro de niveles aceptables.

6) Organización de la prebúsqueda en ESCO-1: En la prebúsqueda por ambas ramas de un salto daremos prioridad a la rama destino del salto.

7) Cálculo de las direcciones de prebúsqueda en ESCO-2: Si podemos disponer del harware necesario, utilizaremos dos incrementadores en cada uno de los dos circuitos para el cálculo de las direcciones de prebúsqueda.

Los resultados obtenidos en los diferentes apartados de este capítulo muestran que las ventajas a nivel de rendimiento de disponer de una memoria de dos puertos son mínimas. Dado que el coste de una memoria de un único puerto es menor, nuestra propuesta es implementar el esquema basado en una memoria de un puerto (ESCO-1).

CAPITULO 4

Organización de memoria

Las características del subsistema de memoria tienen una influencia notable en el rendimiento de cualquier procesador. En este capítulo se presenta una descripción de los parámetros de memoria y se analiza su influencia en el rendimiento del procesador en general y en la eficiencia del mecanismo ESCO en particular. Este capítulo sirve de introducción a los dos capítulos siguientes donde se presentan y evalúan dos implementaciones alternativas del mecanismo ESCO con diferentes organizaciones de memoria.

4.1. CARACTERISTICAS DEL SUBSISTEMA DE MEMORIA

En este apartado comentamos los principales parámetros que caracterizan el subsistema de memoria. Para una descripción más detallada el lector puede remitirse a [Smit82], [Smit86], [ViRO86].

4.1.1. Jerarquía de memoria

La memoria de un computador está organizada de forma jerárquica. El nivel más cercano a la UE de esta jerarquía lo constituyen los registros. En el siguiente nivel está la memoria que directamente alimenta a la UE. Esta memoria recibe el nombre de *memoria cache*.

El diseño de las memorias cache ha ido evolucionando a lo largo de los años debido, entre otros factores, a que los avances tecnológicos permiten cada día una más alta densidad de integración. Actualmente existen procesadores cuya

memoria cache se implementa en varios niveles dando lugar a las denominadas *memorias cache multinivel* [BaWa87]. Como ejemplo podemos citar el procesador MIPS-X [Horo87] que dispone de una memoria cache de instrucciones interna de 2 Kbytes y una memoria cache externa de 64 Kpalabras que a su vez está conectada con memoria principal.

El disponer de una memoria cache de instrucciones implementada en el mismo chip que la CPU es una característica cada vez más frecuente en el diseño de procesadores. Sus principales ventajas son: a) reducir el tiempo de acceso, ya que las comunicaciones dentro del chip son mucho más rápidas que con el exterior y b) proporcionar un camino adicional para acceder a las instrucciones, lo que evita posibles conflictos con los accesos a datos ubicados en el nivel de memoria externa al procesador.

4.1.2. Localidad espacial y temporal

Una memoria cache es un sistema de capacidad reducida y alta velocidad para el almacenamiento de información. La eficiencia de esta memoria se basa en sacar provecho de dos características implícitas a la mayoría de programas: la *localidad temporal* y la *localidad espacial*.

Entendemos por *localidad temporal* el hecho de que es bastante probable que las referencias que se van a realizar en un futuro inmediato sean a las mismas posiciones que las recientemente realizadas. *Localidad espacial* se refiere al hecho de que es bastante probable que las referencias futuras sean a posiciones cercanas a las referencias recientemente realizadas.

Estas dos propiedades hacen que el procesador vea la memoria cache como una memoria de gran capacidad con un tiempo de acceso muy corto. La buena relación coste-efectividad de las memorias cache ha hecho que su uso sea generalizado en toda clase de computadores.

4.1.3. Latencia de memoria

Entendemos por *latencia de memoria* de un determinado nivel de la jeraquía el tiempo de acceso a la memoria de dicho nivel.

Aunque generalmente el diseño de un procesador se lleva a cabo teniendo en cuenta que los accesos a memoria van a realizarse en una memoria cache cuya latencia es relativamente pequeña, no hay que olvidar que en algunas ocasiones esta memoria no es capaz de servir determinados accesos y por lo tanto es preciso acceder al siguiente nivel de la jerarquía. En estos casos el acceso es mucho más costoso ya que generalmente la latencia de memoria es mayor cuanto más alejado del procesador está el nivel de la jerarquía al que se accede.

4.1.4. Correspondencia entre niveles

El contenido de un determinado nivel de la jerarquía de memoria es generalmente un subconjunto del contenido del nivel superior. El contenido de cada nivel se modifica dinámicamente en función de las necesidades del procesador. Por lo tanto existe un intercambio de información entre un nivel y sus niveles vecinos.

Este intercambio de información se realiza de acuerdo a una función que indica la correspondencia entre niveles vecinos. A esta función se le denomina *función de mapeo.*

La función de mapeo se define sobre una determinada unidad de información a la que denominaremos *unidad de mapeo.*

La cantidad de información transferida en cada transacción se denomina *unidad de transferencia.*

4.1.5. Unidad de mapeo de memoria cache

Básicamente existen dos unidades de mapeo alternativas que conducen a diferentes organizaciones de una memoria cache de instrucciones.

La primera alternativa es que la unidad de mapeo sea una cantidad de información de tamaño fijo a la que denominaremos *bloque* (figura 4.1). A una memoria cache con esta unidad de mapeo la denominaremos *memoria cache convencional.*

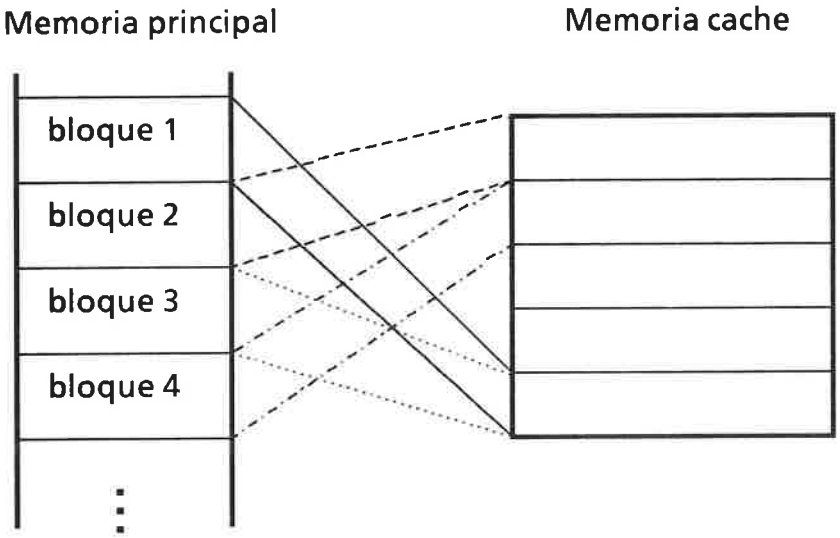


Figura 4.1: Unidad de mapeo en una memoria cache convencional.

La otra posibilidad es que la unidad de mapeo sea la información existente entre dos saltos efectivos consecutivos. En este caso la unidad de mapeo es de tamaño variable y se define en tiempo de ejecución. Nos referiremos a ella como *super bloque básico dinámico* (SBBD). A una memoria cache con esta unidad de mapeo la denominaremos *memoria de instrucciones destino de salto* (MIDS) (figura 4.2).

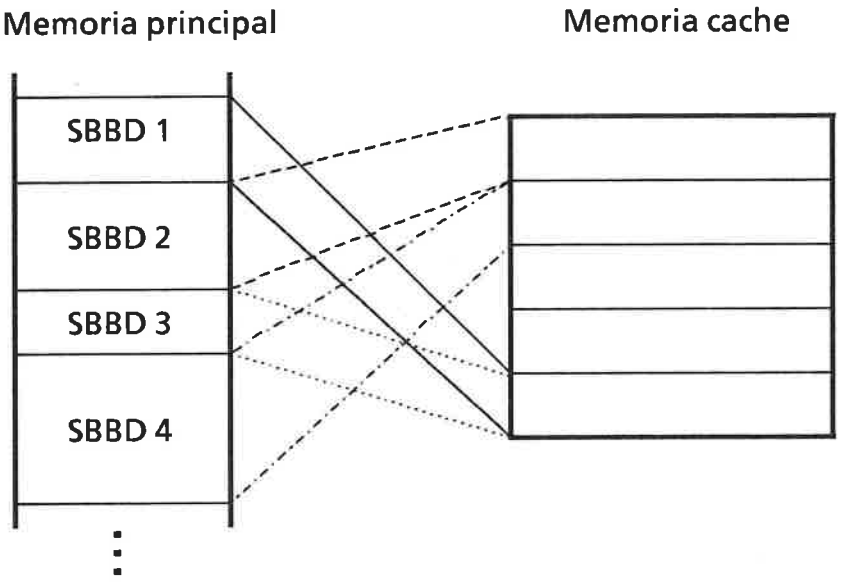


Figura 4.2: Unidad de mapeo en una MIDS.

Para evitar la complejidad que supone gestionar unidades de información de tamaño variable, una implementación usual de una MIDS consiste en limitar el número de instrucciones almacenadas de cualquier SBBD a una cantidad fija. En caso de que el SBBD sea mayor, el resto de instrucciones se obtienen del siguiente nivel de memoria. Una implementación de este tipo se encuentra en el procesador Am29000 [John87].

Denominaremos *línea* a cada una de las entradas de la memoria cache, tanto si su contenido es un bloque como si es un SBBD.

4.1.6. Función de mapeo

En el diseño de una memoria cache, básicamente podemos elegir entre tres tipos posibles de mapeo [ViRO86]: *directo, asociativo por conjuntos* y *completamente asociativo.*

Para un mismo tamaño de memoria, el mapeo completamente asociativo es el que produce una tasa de acieros más elevada mientras que el mapeo directo es el que peor se comporta desde el punto de vista de este parámetro.

Por otro lado, el mapeo directo tiene un coste de implementación menor que el asociativo por conjuntos y éste a su vez menor que el completamente asociativo.

Además, el tiempo de realizar un acceso en caso de acierto es menor si el mapeo es directo que si es asociativo por conjuntos, y este a su vez es menor que cuando el mapeo es completamente asociativo. En [Hill87] se demuestra como para tasas de acierto elevadas este último factor tiene una importancia notoria, de forma que puede llegar a ocurrir que el tiempo medio de acceso a una instrucción sea menor con mapeo directo que con mapeo asociativo por conjuntos o completamente asociativo. Esto es más probable que ocurra en el acceso a instrucciones ya que estos accesos generalmente tienen una localidad mayor que el acceso a datos.

4.1.7. Unidad de transferencia

Básicamente existen dos alternativas: a) La unidad de transferencia es igual a la unidad de mapeo y b) La unidad de transferencia es un subconjunto de la unidad de mapeo.

Con la alternativa b) se consigue aumentar el tamaño de línea que proporciona el máximo rendimiento [Smit87] con lo que se consigue reducir el área destinada a almacenar las etiquetas que implementan la función de mapeo. Sin embargo, hay que tener también en cuenta el coste adicional de la circuitería necesaria para controlar que parte de la línea está en cache y cual no. Esta alternativa permite acortar la prebúsqueda espacial que se realizaría al traer una línea completa. Para tamaños grandes de línea, esta prebúsqueda requiere un tiempo

considerable y además, la probabilidad de que todas las instrucciones prebuscadas vayan a utilizarse en un futuro próximo es bastante pequeña.

4.1.8. Protocolo de bus

El protocolo de bus indica de qué manera un determinado nivel debe realizar los accesos al nivel superior. En particular, vamos a analizar cómo el nivel más bajo de la memoria cache debe acceder al siguiente nivel de la jerarquía de memoria. Vamos a distinguir entre dos tipos de protocolos representados de forma gráfica en la figura 4.3.

Con un *protocolo simple* la UI, tras detectar un fallo en la memoria cache, envía la correspondiente dirección al siguiente nivel de memoria, y tras R ciclos llegan a la UI las instrucciones que componen la línea que ha producido el fallo a razón de una instrucción por ciclo. En caso de que al acceder a la siguiente línea se produzca un nuevo fallo de memoria cache, este protocolo debe repetirse por completo, incluyendo los R ciclos debidos a la latencia de la memoria.

Con un *protocolo modo ráfaga*, las transacciones no son de una longitud fija. Tras servir un fallo a una determinada línea, la memoria puede seguir enviando instrucciones de las líneas que siguen en secuencia, a razón de una por ciclo y sin ningún retardo adicional, hasta que el procesador o la memoria deciden terminar la transacción. De esta forma, cuando un conjunto de líneas sucesivas causan un fallo de cache, se aumenta la eficiencia de memoria, ya que la latencia de memoria (R ciclos) sólo se produce en el acceso a la primera de estas líneas.

Para reducir el tiempo medio de acceso a una instrucción, en ambos protocolos utilizaremos la técnica conocida como *continuación anticipada* ("early continuation"). Esta técnica consiste en pasar a la UE la instrucción que ha producido el fallo tan pronto como llega de memoria, sin esperar a completar la línea en cuestión.

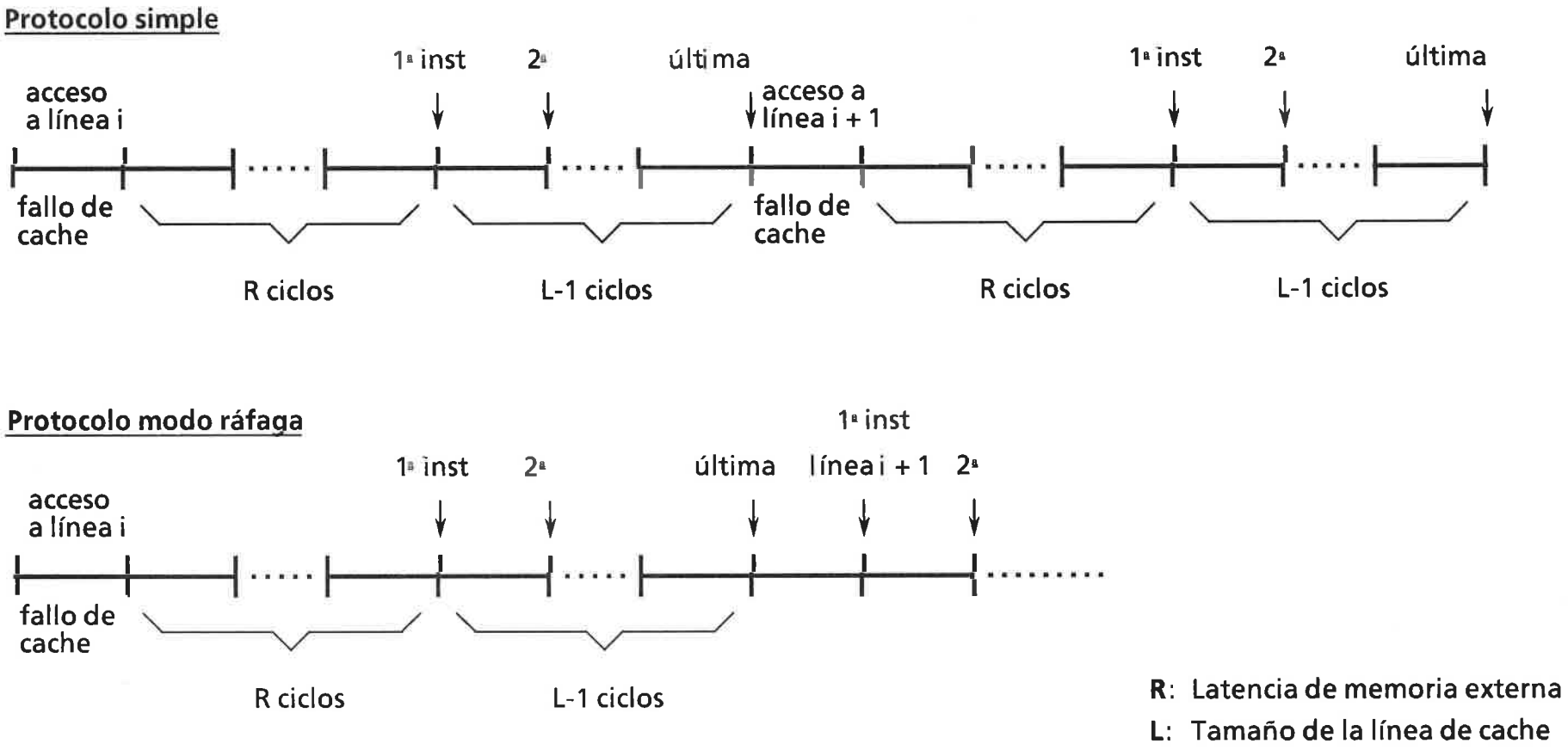


Figura 4.3: Protocolos de acceso a la memoria externa en caso de fallo de memoria cache.

4.2. INTERRELACIONES ENTRE LOS PARAMETROS DE CACHE

En este apartado se analiza la influencia de los parámetros descritos en el apartado anterior en la eficiencia del sistema de memoria lo que a su vez repercute en el rendimiento del procesador.

Una línea de cache almacena instrucciones con direcciones consecutivas. Por lo tanto, cuanto mayor sea el tamaño de una línea más provecho sacaremos de la localidad espacial de los programas. Es interesante observar que utilizando un protocolo modo ráfaga junto con técnicas de prebúsqueda espacial puede conseguirse un suministro ininterrumpido de instrucciones situadas en posiciones consecutivas. Esto significa que a nivel práctico, con esta técnica puede conseguirse un efecto similar al obtenido mediante un aumento del tamaño de línea.

Tal como vimos en el capítulo 1, una forma de disminuir el efecto negativo que produce la latencia de memoria al acceder al nivel superior de memoria es utilizar técnicas de prebúsqueda. Ya que en caso de acierto la memoria cache puede proporcionar una línea por acceso, el tamaño de la línea influye en la anticipación con que son detectados los fallos de memoria cache. Esta anticipación determina que parte de la latencia de memoria puede solaparse con la ejecución de instrucciones previamente prebuscadas. No obstante, hay que tener en cuenta que de la prebúsqueda espacial que realizamos al acceder a una línea sólo será útil aquella parte consistente en las instrucciones que van a ejecutarse inmediatamente. La elevada frecuencia de las instrucciones de salto implica que esta prebúsqueda dejará de ser útil a partir de valores bastante pequeños del tamaño de línea.

El tamaño de línea junto con la latencia del siguiente nivel de memoria influyen en la penalización en caso de fallo de memoria cache. El tiempo de acceso a la instrucción que ha producido el fallo dependerá de la posición de esta instrucción en la línea. Su tiempo medio de acceso será $R + (L-1) / 2$. Dada la elevada frecuencia de las instrucciones de salto, este factor tendrá un peso considerable en el tiempo medio de acceso a una instrucción. Este tiempo puede

reducirse haciendo que la memoria del nivel superior envíe las instrucciones que forman la línea empezando por la que ha producido el fallo.

El tamaño de línea y la latencia de memoria determinan el tiempo necesario para reemplazar una línea en memoria cache. Este tiempo de reemplazo puede disminuirse si la unidad de transferencia es un subconjunto de la unidad de mapeo.

Al aumentar el número de líneas de cache favorecemos en mayor grado a aquellos programas con una menor localidad temporal. Por otra parte, el área que debe dedicarse para almacenar las etiquetas es directamente proporcional al número de líneas

El número de líneas y el tamaño de cada línea determina la capacidad de la memoria cache. Esta capacidad más el área necesaria para las etiquetas determina el coste hardware que supone su implementación.

4.3. MEMORIA CACHE CONVENCIONAL versus MIDS

La localidad espacial que exhibe un programa viene principalmente determinada por el número de instrucciones entre dos saltos efectivos consecutivos. Ello implica que una MIDS hace un mejor uso de esta localidad ya que su unidad de mapeo es el SBBD.

Debido a la complejidad y coste de gestionar líneas de tamaño variable, generalmete una MIDS se implementa con líneas de tamaño fijo. Si un SBBD es mayor que una línea, el resto de instrucciones del SBBD se obtienen del siguiente nivel de memoria. Ello supone un incremento del tráfico respecto al generado por una cache convencional, ya que hay determinadas partes de un programa que nunca se almacenarán en la memoria cache, lo que no ocurre en la cache convencional.

Cada línea de una MIDS se dedica a almacenar un SBBD distinto, mientras que en una cache convencional puede haber varias líneas que almacenen diferentes partes del mismo SBBD. Ello supone que si ambas organizaciones tienen el mismo número de líneas, la MIDS va a tener capacidad para almacenar

un mayor número de SBBD y por lo tanto se comportará mejor que la cache convencional por lo que se refiere a la localidad temporal.

Respecto a la localidad espacial, ya que la MIDS sólo dispone de las primeras instrucciones de cada SBBD, esta memoria va a necesitar de algún mecanismo eficiente para poder obtener el resto del SBBD del nivel superior de memoria. Por ejemplo este mecanismo puede consistir en un protocolo de bus modo ráfaga y el uso de técnicas de prebúsqueda. Lógicamente estas técnicas también mejoran el rendimiento de una cache convencional, pero su impacto es mucho mayor en la MIDS. En resumen, una cache convencional puede almacenar por completo un SBBD independientemente de su tamaño, y por lo tanto sacar un provecho máximo de la localidad espacial. Una MIDS con la implementación descrita anteriormente, necesita de técnicas adicionales para poder sacar provecho de la localida espacial de los SBBD con tamaño mayor al de la línea de cache.

4.4. MECANISMO ESCO Y MEMORIA CACHE

El rendimiento del mecanismo ESCO viene determinado por la anticipación con que los saltos son detectados. Dado que en un acceso leemos una línea completa de cache la anticipación máxima la conseguiremos si todas las instrucciones de la línea se analizan en paralelo para la detección de los saltos. Por lo tanto, el tamaño de línea repercute en la anticipación máxima con que los saltos pueden ser detectados. Sin embargo, en el capítulo anterior vimos como analizando únicamente dos instrucciones por ciclo se conseguía ejecutar una gran mayoría de los saltos con coste cero. Por lo tanto, la reducción del coste de los saltos que podemos conseguir con un aumento del tamaño de línea no es demasiado significativa.

En este trabajo no pretendemos evaluar el rendimiento de las diversas alternativas en el diseño de un sistema de memoria. En cualquier caso estos resultados pueden encontrase en [ACHA87], [Hill87], [PrHH88]. Nuestro objetivo es estudiar el comportamiento del mecanismo ESCO y analizar aquellos parámetros de memoria cache que de una forma directa repercuten en el rendimiento e implementación del mecanismo.

En los dos capítulos posteriores se analiza detalladamente la implementación del mecanismo ESCO con dos organizaciones alternativas de memoria: a) cache convencional y b) MIDS.

El rendimiento del sistema es evaluado para dos protocolos de bus: a) simple y b) modo ráfaga. Para la MIDS sólo se evalúa el protocolo modo ráfaga ya que con esta organización tiene poco sentido utilizar un protocolo simple debido a la poca eficiencia que se conseguiría.

En estas evaluaciones supondremos que el tipo de mapeo es directo ya que es el más sencillo de implementar y además, dada la localidad de los accesos a instrucciones, su rendimiento es bastante bueno.

Como latencia de memoria del siguiente nivel de la jerarquía generalmente asumiremos valores en torno a cuatro ciclos. La latencia de memoria varía considerablemente de una implementación a otra. Un valor en torno a cuatro ciclos es comparable con la memoria del procesador Am29000 [John87].

Supondremos que la unidad de transferencia es igual a la unidad de mapeo, comentando aquellos casos en que pueda conseguirse una mejora considerable utilizando como unidad de transferencia un subconjunto de la unidad de mapeo.

Utilizaremos la técnica denominada continuación anticipada y supondremos que la memoria del nivel superior sirve las peticiones hechas por la memoria cache empezando por la primera instrucción de la línea.

En vistas a que la memoria cache se implemente en el mismo chip que la CPU, las evaluaciones serán llevadas a cabo asumiendo tamaños de cache entre 256 bytes y 4 Kbytes.

CAPITULO 5

Diseño y evaluación de la UI con una memoria cache convencional

En este capítulo se presenta y evalúa una implementación del mecanismo ESCO basada en una UI que utiliza una memoria cache de instrucciones convencional. [GLlC88b]. En primer lugar se presenta un diseño de la UI que implementa este mecanismo. Seguidamente se evalúa el rendimiento del procesador para los dos protocolos de comunicación entre la memoria cache de instrucciones y el siguiente nivel de la jerarquía de memoria presentados en el capítulo anterior. En esta evaluación se analizan diversas alternativas de diseño no analizadas anteriormente debido a que están relacionadas con el comportamiento de la memoria cache.

5.1. DISEÑO DE LA UNIDAD DE INSTRUCCIONES

Como consecuencia del estudio realizado en el capítulo 3, el diseño de la UI que proponemos para implementar el mecanismo ESCO es el de la figura 5.1.

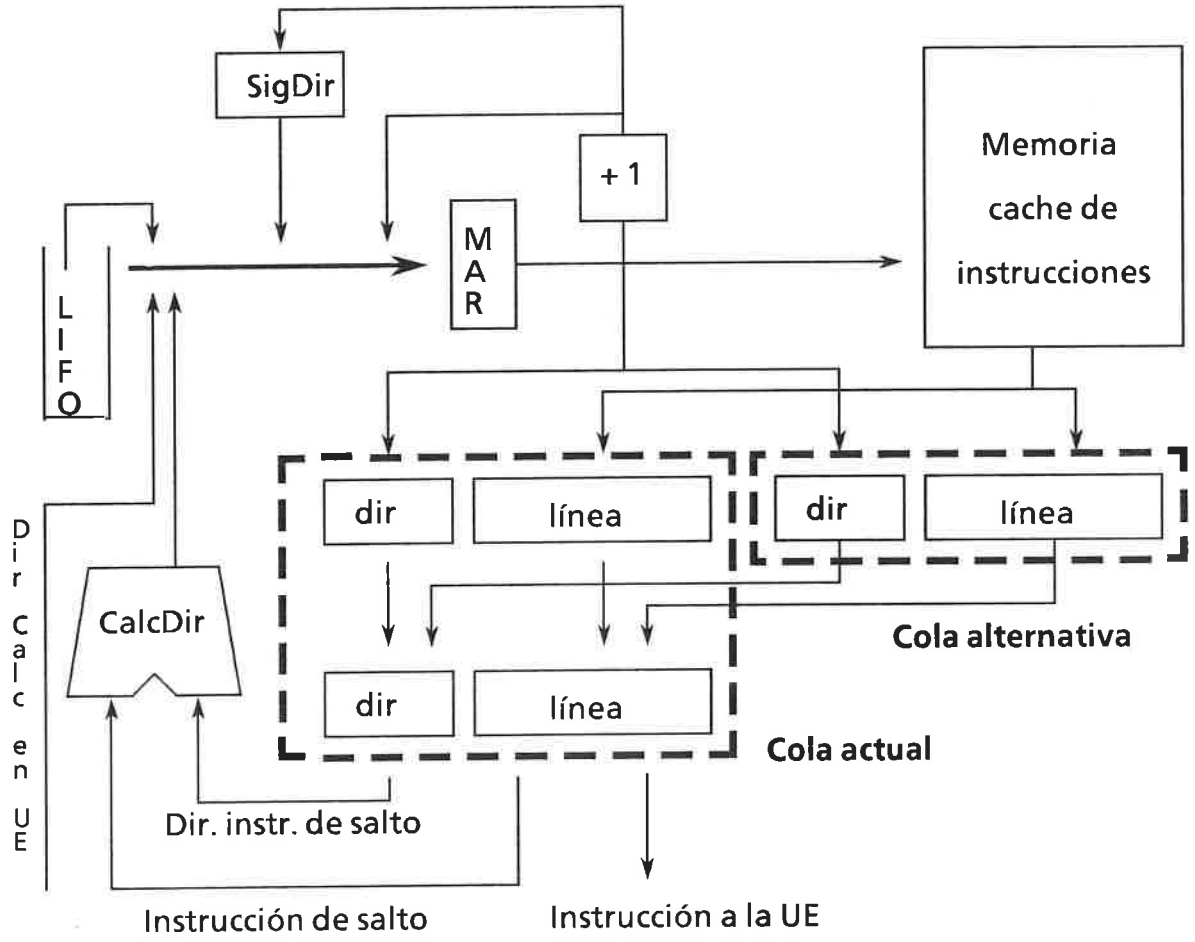


Figura 5.1: Diseño de la unidad de instrucciones.

La memoria de instrucciones es una memoria cache convencional de un único puerto que en cada acceso lee una línea completa formada por varias

instrucciones. Un parámetro que analizaremos en detalle en un apartado posterior es el tamaño de estas líneas de cache.

La UI dispone de dos colas de instrucciones en las que almacena las instrucciones prebuscadas. La *cola actual* se utiliza para almacenar las instrucciones del actual BBD en ejecución y las instrucciones que le siguen en secuencia. Por lo tanto esta cola es la que alimenta a la UE. La *cola alternativa* almacena las primeras instrucciones del destino del próximo salto a ejecutar.

El tamaño de estas colas es de dos y una líneas respectivamente. La elección de estos tamaños es consecuencia de los resultados presentados en el capítulo 3. Estos resultados indicaban que limitando la capacidad de almacenamiento de la UI a muy pocas instrucciones se conseguía obtener prácticamente el máximo rendimiento del mecanismo ESCO. En cualquier caso, mediante simulación se evaluó el comportamiento del mecanismo ESCO con una memoria cache de instrucciones para diferentes tamaños de estas colas. Los resultados mostraron una ganancia prácticamente despreciable para tamaños mayores que los utilizados en el diseño de la figura 5.1.

Durante la ejecución de un BBD, la UI prebusca instrucciones de este BBD, a razón de una línea por ciclo, y las almacena en la *cola actual*. Cada línea prebuscada es analizada para detectar si alguna de sus instrucciones es un salto. Como es lógico, una línea será prebuscada únicamente cuando haya espacio libre en la cola para que pueda ser almacenada. En cada ciclo, una instrucción útil de esta cola, o una NOP en caso de no disponer de ninguna, es enviada a la UE. Tras leer el salto situado al final del BBD, en el siguiente ciclo se accede a la primera línea del destino del salto y se almacena en la *cola alternativa*. Como comentamos en el capítulo 2, utilizando la codificación propuesta por Katevenis [Kate85]este acceso puede llevarse a cabo en paralelo con el cálculo de la parte más significativa de la dirección destino. Este cálculo se realiza en *CalcDir*. A continuación, siempre que haya un espacio libre en la *cola actual*, se prebuscará una línea de instrucciones que siguen en secuencia al salto.

Cuando la instrucción que precede al salto finaliza su operación en la ALU, el valor de los códigos de condición determina si el salto debe ser efectivo o no. Si el salto no es efectivo se anula el contenido de la *cola alternativa*. Si por el contrario el salto es efectivo, se anula el contenido de la *cola actual* y el contenido de la *cola alternativa* se copia sobre la *cola actual*.

El registro *SigDir* se utiliza para almacenar la dirección de la siguiente línea a buscar de la rama que actualmente no está siendo prebuscada. Es decir, mientras la UI está buscando la primera línea del destino de un salto, este registro almacena la dirección de la línea que sigue en secuencia a la que contiene el salto, y que por lo tanto debe ser prebuscada a continuación. De igual forma, tras prebuscar la primera línea del destino de un salto, en este registro se almacena la dirección de la línea siguiente. Esta información será necesaria para reanudar la prebúsqueda en caso de que el salto sea efectivo.

Para soportar llamadas y retornos de subrutina y hacerlos transparentes a la UE, la UI implementa una estructura tipo LIFO donde almacena la dirección de retorno de cada llamada a subrutina ejecutada. Cuando el próximo salto que debe ser ejecutado es un retorno de subrutina, la dirección destino de este salto se obtiene de la cabeza de esta pila.

Para tratar los saltos no relativos al PC, en los que la dirección destino debe calcularse en la UE y enviarse posteriormente a la UE, existe un camino entre la UI y la UE utilizado para tal fin.

5.1.1. Accesos pendientes en el instante de salto

Con el modo de operación descrito para la UI, ocurrirá en determinados casos que en el mismo ciclo en que se toma una decisión de salto se está sirviendo un fallo de memoria cache de alguna de sus dos ramas. Si esta rama resulta ser justamente la que no va a seguir el salto, debemos decidir que se hace con este acceso a memoria externa ya iniciado pero todavía no finalizado. En este apartado vamos analizar diversas formas alternativas de tratar estos accesos, para finalmente escoger aquella que proporcione una mayor eficiencia.

Básicamente existen tres formas alternativas de tratar estas transacciones. La primera es cancelar dicha transacción y, por lo tanto, no incluir dicha línea en la memoria cache. La segunda es terminar la transacción y actualizar la memoria cache incluyendo dicha línea. La tercera es una solución intermedia entre la primera y la segunda. Consiste en terminar la transacción en el caso de que hubiese llegado como mínimo una instrucción de dicha línea, y cancelarla en caso contrario.

En la primera alternativa, si se cancela un acceso durante la ejecución de un bucle, existe el inconveniente de que el próximo y posiblemente sucesivos accesos a dicha línea volverán a ocasionar un fallo de cache, y por lo tanto, en cada iteración del bucle la UI dedicará un número determinado de ciclos para empezar a servir la transacción y luego cancelarla.

Este acceso cancelado puede corresponder a una instrucción de salto situada en el interior de un bucle (por ejemplo, caso de una sentencia IF-THEN-ELSE). Si este acceso cancelado es a la rama destino del salto, los ciclos que la UI está ocupada accediendo a esta rama impiden que la UI prebusque la línea que sigue al salto en secuencia, que justamente es la rama que va a seguir el salto.

El problema puede ser todavía mayor si el acceso cancelado es a la rama correspondiente al salto no efectivo. En este caso, el acceso cancelado puede ser a una línea que contiene instrucciones del actual BBD y del siguiente. Si esta línea nunca la introducimos en la memoria cache, cada nueva ejecución de este BBD será retardada debido a que hay que ir a buscar el principio de esta línea a memoria externa.

La desventaja de la segunda alternativa es que generalmente estas transacciones son de instrucciones que no se van a utilizar en un futuro inmediato y por lo tanto polucionan la memoria cache y pueden ocasionar una disminución de la tasa de aciertos.

La tercera alternativa poluciona la cache pero en menos casos que la segunda y tiene el mismo inconveniente que la primera alternativa, pero el número de transacciones canceladas es menor y por lo tanto su efecto negativo se reduce.

En la tabla 5.1 se muestra el rendimiento para cada una de las tres alternativas suponiendo una memoria cache de mapeo directo con capacidad de 2 Kbytes, tamaño de línea de 4 instrucciones, latencia de memoria externa igual a 4 ciclos (R igual a 4 en la figura 4.3) y protocolo simple para acceder a esta memoria.

Los resultados de la tabla 5.1 muestra que la alternativa 3 es la que proporciona un mayor rendimiento, por lo tanto, es ésta la opción que proponemos implementar en la UI. La alternativa 1 es la menos eficiente de las tres. Ello se debe al efecto comentado anteriormente para determinados BBD en los que la

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media
ALT 1	0.926	0.537	0.448	0.839	0.687
ALT 2	0.969	0.545	0.487	0.952	0.738
ALT 3	0.971	0.558	0.512	0.956	0.749

Tabla 5.1: Eficiencia del mecanismo ESCO para tres formas alternativas de tratar las transacciones pendientes cuando se toma una decisión de salto:

ALT 1: Son canceladas.
ALT 2: Son completadas
ALT 3: Si ha llegado alguna instrucción se completan y se cancelan en caso contrario.

En los tres casos la memoria cache es de mapeo directo, con capacidad de 2 Kbytes, tamaño de línea de 4 instrucciones, 4 ciclos de retardo en caso de fallo y acceso a memoria externa mediante protocolo simple.

última línea contiene instrucciones del actual BBD y del siguiente. Puede ocurrir que esta línea, pese a ejecutarse repetidas veces, nunca esté en memoria cahe, y por lo tanto haya que ir buscarla a memoria externa en cada ejecución del BBD.

5.1.2. Tamaño de la línea de cache

En el capítulo anterior (apartado 4.2) analizamos la relación entre el tamaño de línea y el rendimiento del procesador. En este apartado vamos a evaluar su influencia para finalmente obtener el tamaño que optimiza la eficiencia del procesador con el mecanismo ESCO.

Para ello hemos evaluado el comportamiento de los programas de prueba con diferentes parámetros del sistema de memoria. Los resultados se muestran en la

figura 5.2. Puede observarse que el mejor rendimiento se consigue con líneas de 4 instrucciones. Este tamaño es bastante parecido a la longitud media de los BBD. Como se ha comentado en el capítulo anterior, a partir de este tamaño, parte de las instrucciones prebuscadas en un acceso tienen una alta probabilidad de no ser utilizadas en un futuro inmediato. Estos resultados están en consonancia con los obtenidos por Przybylski et al. en [PrHH88].

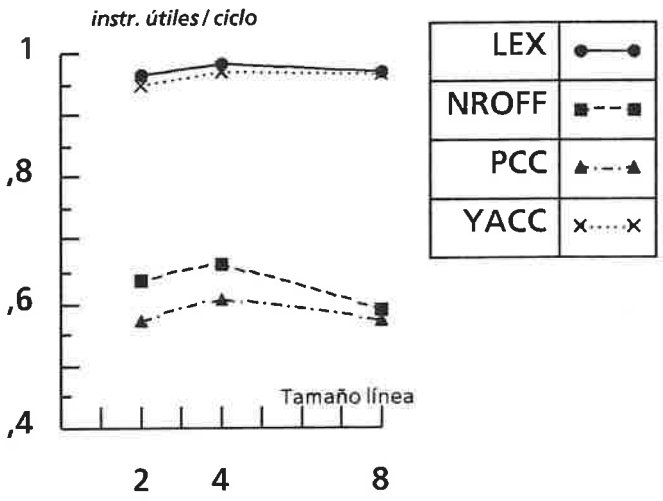
5.2. EVALUACION DEL MECANISMO ESCO

En este apartado presentamos medidas acerca del rendimiento del mecanismo ESCO. Estos resultados son comparados con el rendimiento obtenido mediante otros mecanismos de uso bastante generalizado. En primer lugar, esta evaluación se lleva a cabo asumiendo un protocolo simple para la comunicación con la memoria externa. Seguidamente se evalúa una alternativa de diseño consistente en limitar la prebúsqueda en los saltos incondicionales a la rama destino del salto. Finalmente se evalúa el rendimiento del sistema al utilizar un protocolo modo ráfaga en la conexión con memoria externa.

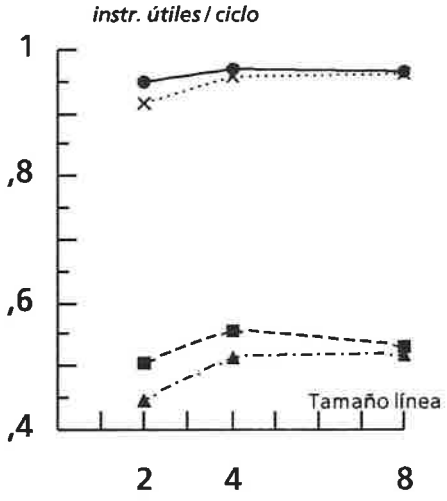
5.2.1. Protocolo simple

En la tabla 5.2 se muestra la eficiencia del mecanismo ESCO junto con la de otros mecanismos descritos en el capítulo 1. Hemos asumido una memoria cache de mapeo directo con un tamaño de línea de 4 instrucciones (16 bytes) y una capacidad de 2 ó 4 Kbytes (128 ó 256 líneas). Hemos supuesto que la latencia de memoria externa es igual a 4 ciclos (R en figura 4.3) y que la conexión con la memoria externa es mediante un protocolo simple.

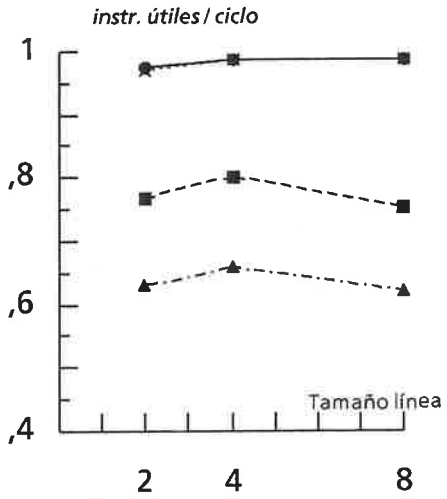
Observando los resultados de la tabla 5.2 (columna %aumen.), puede apreciarse que el rendimiento del mecanismo ESCO medido en instrucciones útiles por ciclo es entre un 25 y un 38% superior al de los mecanismos basados en el esquema de salto retardado. Respecto al mecanismo *Memoria de instrucciones destino de salto* (MIDS), la mejora es entre un 18 y un 22%. Hay que destacar que este mecanismo, tal como se ha evaluado tendría un coste de implementación considerable en cuanto a área de chip se refiere, ya que su capacidad es de 1 Kbyte, a lo que hay que añadir el espacio necesario para almacenar las etiquetas.



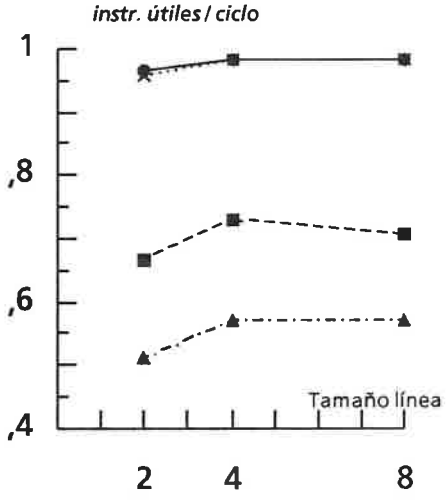
Tamaño cache: 2 Kbytes
Latencia mem. ext: 2 ciclos



Tamaño cache: 2 Kbytes
Latencia mem. ext: 4 ciclos



Tamaño cache: 4 Kbytes
Latencia mem. ext: 2 ciclos



Tamaño cache: 4 Kbytes
Latencia mem. ext: 4 ciclos

Figura 5.2: Eficiencia del mecanismo ESCO en instrucciones útiles ejecutadas por ciclo de procesador para diferentes parámetros de memoria cache.

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%aumen.
SR	0.676	0.404	0.422	0.723	0.556	35
SRA	0.729	0.408	0.433	0.743	0.578	30
SRAP	0.742	0.456	0.430	0.765	0.598	25
SSR	0.750	0.495	0.462	0.782	0.622	20
MIDS	0.751	0.533	0.466	0.784	0.633	18
ESCO	0.971	0.558	0.512	0.956	0.749	-

a) Capacidad: 2 Kbytes

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%aumen.
SR	0.681	0.490	0.466	0.735	0.593	38
SRA	0.734	0.482	0.468	0.757	0.610	34
SRAP	0.747	0.603	0.466	0.781	0.649	26
SSR	0.757	0.616	0.503	0.798	0.668	22
MIDS	0.757	0.625	0.503	0.798	0.671	22
ESCO	0.983	0.729	0.570	0.984	0.817	-

b) Capacidad: 4 Kbytes

Tabla 5.2: Eficiencia con una memoria cache de instrucciones. La columna *%aumen.* representa la diferencia de rendimiento entre el mecanismo ESCO y el mecanismo en cuestión respecto a este último.

Parámetros de la cache:

- Mapeo directo
- Capacidad: 2 ó 4 Kbytes
- Tamaño de línea: 4 instrucciones
- Latencia de memoria externa: 4 ciclos
- Protocolo simple

Una característica importante a destacar es que el rendimiento de ESCO no tiene una relación directa con la localidad temporal del programa ejecutado. Tal como vimos al modelizar el comportamiento de este mecacnismo, su rendimiento es únicamente función de la longitud de los BBD, es decir, de la frecuencia de las instrucciones de salto. Por ejemplo, si suponemos que la memoria es capaz de servir todas las peticiones de la UI en el mismo ciclo en que son realizadas, el rendimiento de ESCO sería el mismo para un programa que ejecutase 1000 BBD de longitud L, cada uno de ellos una sola vez, que para un programa formado por un único BBD de longitud L que se ejecutase 1000 veces. En el primer caso la localidad temporal es nula, mientras que en el segundo es muy elevada.

Por el contrario, otros mecanismos hardware como el MIDS, tienen un rendimiento estrechamente ligado a la localidad temporal de los programas. Es decir, en el mecanismo MIDS el coste de un salto depende de cuanto tiempo hace que se ha ejecutado ese mismo salto, ya que en función de ello puede que la información asociada a él esté o no esté en la memoria destinada a tal fin.

Lógicamente, el rendimiento del procesador sí va a depender de la localidad del programa ya que ésta determina la tasa de aciertos de la memoria cache. Cuando se produce un fallo en memoria cache hay que ir a buscar la línea correspondiente al siguiente nivel de la jerarquía. Ya que la latencia de este nivel es mayor que la de memoria cache, este fallo supondrá un determinado retardo en el acceso a estas instrucciones. Este retardo supondrá una disminución de la anticipación con que los saltos son accedidos y por lo tanto un aumento del coste de los saltos. Además este retardo también puede ocasionar que la UI pierda algunos ciclos a la espera de que llegue alguna instrucción útil para ejecutar.

De los 4 programas de prueba, LEX y YACC son los dos más locales. Puede observarse que para estos programas el rendimiento obtenido en la tabla 5.2 es prácticamente el mismo que el obtenido en el capítulo 3 donde no se tenía en cuenta el efecto de los fallos de memoria cache (tabla 3.2).

En la tabla 5.2 puede también apreciarse como al disminuir el tamaño de la memoria cache, la diferencia de rendimiento entre ESCO y el resto de mecanismos es menor, sobre todo para los programas menos locales (NROFF y PCC). Como hemos comentado anteriormente, ello se debe a que el rendimiento de ESCO se basa en la utilización de técnicas de prebúsqueda para conseguir

resolver los saltos con cierta anticipación. Al disminuir el tamaño de la memoria cache disminuye la tasa de aciertos, y por lo tanto, esta anticipación es menor.

Un inconveniente de las técnicas de prebúsqueda es que pueden aumentar el tráfico entre la memoria cache y la memoria externa. Si este tráfico se realiza a través de un bus compartido con otros procesadores o controladores de entrada y salida, puede tener una repercusión directa sobre el rendimiento global del sistema. Por este motivo hemos medido el tráfico generado por ESCO y por el resto de mecanismos. Los resultados se muestran en la tabla 5.3. En la figura 5.3 se representa gráficamente el tráfico medio generado por los cuatro programas de prueba. Como medida de tráfico hemos utilizado la proporción del tiempo que dura la ejecución de un programa en que el bus de instrucciones se encuentra ocupado transmitiendo instrucciones.

Como puede observarse en estos resultados, el tráfico generado por ESCO es algo mayor que el generado por los mecanismos MIDS y SSR. Ello se debe a la prebúsqueda de instrucciones por ambas ramas de los saltos. Esta prebúsqueda, además de aumentar el tráfico por sí misma, poluciona en cierta medida la memoria cache, lo que a su vez produce un aumento adicional de tráfico. Además, el esquema MIDS tiene la ventaja adicional de disponer de una memoria de 1 Kbyte donde se almacenan las instrucciones destino de salto. Esta memoria puede verse en cierto modo como una ampliación de la memoria cache, lo que ocasiona una disminución adicional del tráfico generado.

Comparando el tráfico de ESCO con el de los mecanismos basados en el esquema de salto retardado (SR, SRA, SRAP), puede observarse que los resultados son muy similares.

Por una parte ESCO tiene la desventaja de que prebusca instrucciones por las dos ramas de los saltos, algunas de las cuales son desechadas cuando se conoce el resultado del salto. En cambio, los mecanismos de salto retardado, al no utilizar técnicas de prebúsqueda, únicamente acceden a las instrucciones que van a ser ejecutadas.

Sin embargo, el número de instrucciones desechadas en el mecanismo ESCO y que ocasionen tráfico adicional es bastante pequeño. Debido al tamaño de las colas de instrucciones como máximo se desechará una línea por cada salto. Para que esto ocurra y además esta línea genere un tráfico adicional, debe ocurrir que

ocupación bus de instr.	LEX	NROFF	PCC	YACC	Media
SR	0.015	0.324	0.288	0.028	0.164
SRA	0.017	0.331	0.279	0.032	0.165
SRAP	0.015	0.296	0.301	0.029	0.160
SSR	0.014	0.269	0.288	0.026	0.149
MIDS	0.013	0.239	0.279	0.024	0.139
ESCO	0.019	0.303	0.320	0.032	0.169

a) Capacidad: 2 Kbytes

ocupación bus de instr.	LEX	NROFF	PCC	YACC	Media
SR	0.010	0.235	0.237	0.015	0.124
SRA	0.011	0.253	0.239	0.017	0.130
SRAP	0.009	0.146	0.264	0.013	0.108
SSR	0.007	0.156	0.244	0.009	0.104
MIDS	0.007	0.141	0.235	0.008	0.098
ESCO	0.009	0.185	0.277	0.011	0.121

b) Capacidad: 4 Kbytes

Tabla 5.3: Tráfico entre memoria cache y memoria externa medido en porcentaje de tiempo que el bus de instrucciones está ocupado.

Parámetros de la cache:

- Mapeo directo
- Capacidad: 2 ó 4 Kbytes
- Tamaño de línea: 4 instrucciones
- Latencia de memoria externa: 4 ciclos
- Protocolo simple

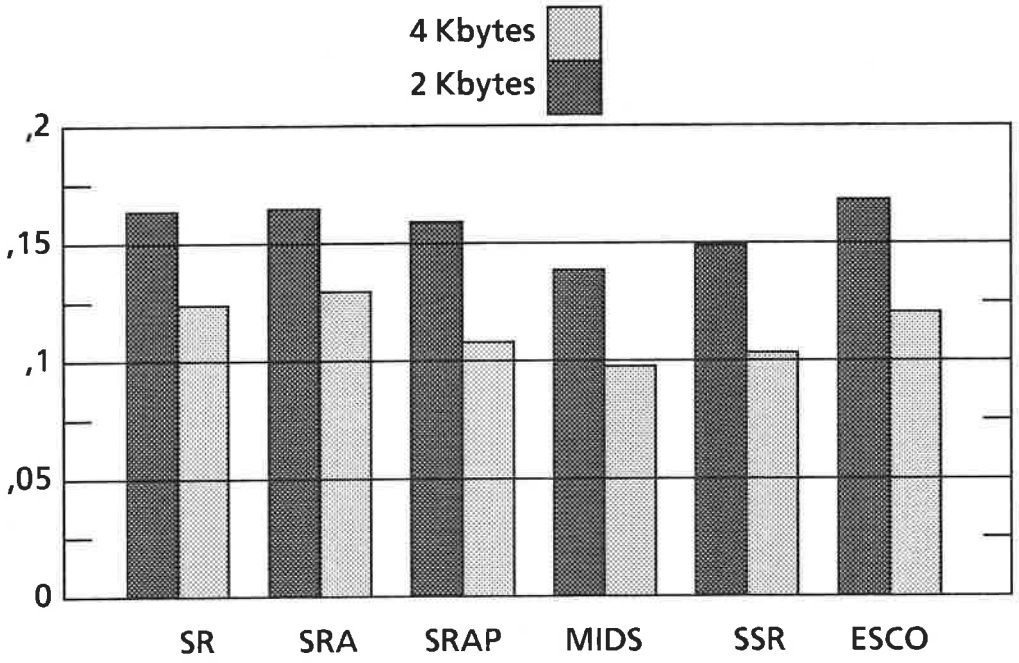


Figura 5.3: Tráfico entre memoria cache y memoria externa medido en porcentaje de tiempo que el bus de instrucciones está ocupado.

Parámetros de la cache:

- Mapeo directo
- Capacidad: 2 (tabla a) ó 4 Kbytes (tabla b)
- Tamaño de línea: 4 instrucciones
- Latencia de memoria externa: 4 ciclos
- Protocolo simple

el acceso a la primera línea de la rama abandonada ocasione un fallo de cache y que este acceso se realice un número de ciclos antes de producirse el fallo igual a la latencia de memoria externa (ya que de lo contrario el salto ocurrirá antes de que llegue la primera instrucción y por lo tanto la transacción será abortada). Ello implica que el salto debe haberse localizado con una anticipación mayor que la latencia de memoria.

Por otra parte, las técnicas basadas en el salto retardado tienen la desventaja de incrementar el tamaño del código de los programas, ya que en determinados casos, tras un salto, se requiere añadir instrucciones de NOP o duplicar algunas instrucciones del programa, y por lo tanto, para ejecutar el mismo programa es preciso acceder a un mayor número de instrucciones.

5.2.2. Saltos incondicionales

En el capítulo 3 analizamos una alternativa de diseño que consistía en tratar los saltos incondicionales de forma que la UI se limitara a prebuscar únicamente instrucciones de la rama destino. Vimos que esta alternativa apenas afectaba al rendimiento del mecanismo ESCO. Al evaluar el comportamiento del sistema teniendo en cuenta la existencia de una memoria cache, esta alternativa puede afectar al comportamiento de esta memoria debido a que en determinados momentos evitamos prebuscar líneas de instrucciones que no van a ser utilizadas en un futuro inmediato.

Los resultados de evaluar esta alternativa para varios parámetros de cache muestran que la diferencia, tanto en rendimiento como en tráfico entre memoria cache y memoria externa, es menor que el 1% en todos los casos, por lo que no creemos conveniente complicar el diseño de la UI para obtener un beneficio tan reducido.

El motivo de esta diferencia tan reducida se debe a que la polución de memoria cache en ambos casos es prácticamente la misma. Para que esta alternativa produzca algún beneficio debe ocurrir que desde el ciclo en que la UI encuentra un salto, hasta el ciclo en que este debe ser efectivo, la UI haya tenido tiempo suficiente para prebuscar la primera línea de la rama destino del salto y alguna instrucción de la primera línea que sigue en secuencia al salto, y además que esta última línea produzca un fallo de cache, de lo contrario si ya está en cache no varía el contenido de ésta. La elevada frecuencia de las instrucciones de salto implica que la anticipación con que generalmente estas instrucciones son encontradas es muy pequeña, por lo que raramente se dan todo el cúmulo de circunstancias anteriormente citado.

5.2.3. Protocolo modo ráfaga

Con un protocolo modo ráfaga, cuando se produce un fallo de memoria cache, la memoria externa envía la línea correspondiente a razón de una instrucción por ciclo. Una vez terminada de enviar dicha línea, la memoria externa sigue enviando instrucciones de las líneas siguientes, a razón de una por ciclo, hasta que por algún motivo se decide terminar la transacción.

Con este protocolo es de esperar un aumento del rendimiento del sistema para todos los mecanismos, ya que debido a la localidad espacial de los programas, es bastante probable que tras acceder a la línea i se acceda a la $i+1$. En caso de que ambas línea no estén en memoria cache, el tiempo total de acceso es menor utilizando el protocolo modo ráfaga que utilizando el protocolo simple.

En nuestro caso, una transacción entre memoria cache y la memoria externa será abortada cuando se detecte que la siguiente línea ya está en memoria cache, o bién cuando se termine de acceder a una línea que contenga una instrucción de salto (ya que el siguiente acceso será a la primera línea del destino del salto), o bién cuando se termine de acceder a la primera línea del destino de un salto (ya que el siguiente acceso será a la primera línea que sigue en secuencia al salto).

En la tabla 5.4 se muestra el rendimiento para los diversos mecanismos utilizando un protocolo modo ráfaga. El rendimiento de ESCO medido en instrucciones útiles ejecutadas por ciclo es entre un 18 y un 36% mayor que el del resto de los mecanismos, lo que implica una ganancia similar a la obtenida con el protocolo simple.

Para cualquier mecanismo, la diferencia de rendimiento entre el protocolo simple y el protocolo modo ráfaga es tanto mayor cuanto menor es la tasa de aciertos del programa. En las tablas 5.3 y 5.4 puede observarse que la diferencia de rendimiento para los programas LEX y YACC es mínima dado que estos programas son bastante locales. Sin embargo para los programs NROFF y PCC las diferencias entre ambos protocolos son sustanciales.

En cuanto al tráfico entre memoria cache y la memoria externa, puede comprobarse que si el tamaño de la línea de cache es igual a la latencia de la memoria externa, el número de líneas que se escriben de memoria externa a

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%aumen.
SR	0.682	0.481	0.495	0.736	0.598	32
SRA	0.736	0.481	0.505	0.758	0.620	28
SRAP	0.748	0.523	0.509	0.780	0.640	24
SSR	0.756	0.561	0.540	0.794	0.663	19
MIDS	0.757	0.590	0.533	0.795	0.670	18
ESCO	0.980	0.628	0.587	0.969	0.791	-

a) Capacidad: 2 Kbytes

inst. útiles por ciclo	LEX	NROFF	PCC	YACC	Media	%aumen.
SR	0.684	0.544	0.528	0.741	0.624	36
SRA	0.739	0.548	0.530	0.765	0.645	31
SRAP	0.751	0.640	0.536	0.787	0.678	25
SSR	0.759	0.662	0.567	0.802	0.698	21
MIDS	0.759	0.663	0.563	0.802	0.697	22
ESCO	0.987	0.783	0.635	0.988	0.848	-

b) Capacidad: 4 Kbytes

Tabla 5.4: Eficiencia con una memoria cache de instrucciones. La columna *%aumen.* representa la diferencia de rendimiento entre el mecanismo ESCO y el mecanismo en cuestión respecto a este último.

Parámetros de la cache:

- Mapeo directo
- Capacidad: 2 ó 4 Kbytes
- Tamaño de línea: 4 instrucciones
- Latencia de memoria externa: 4 ciclos
- Protocolo modo ráfaga

memoria cache es el mismo tanto en el protocolo simple como en el protocolo modo ráfaga. Ello es debido a que la secuencia de referencias a memoria cache realizadas por la UI es exactamente igual en ambos casos. Ya que la eficiencia del procesador con protocolo modo ráfaga es mayor que con protocolo simple, esto supone que con el primer protocolo la misma cantidad de tráfico se realiza en menos tiempo y por lo tanto, el tráfico resultante será mayor para el protocolo modo ráfaga. Los reultados referentes al tráfico se listan en la tabla 5.5 y figura 5.4.

ocupación bus de instr.	LEX	NROFF	PCC	YACC	Media
SR	0.015	0.386	0.337	0.028	0.192
SRA	0.017	0.391	0.325	0.033	0.191
SRAP	0.015	0.340	0.356	0.030	0.185
SSR	0.015	0.305	0.337	0.026	0.171
MIDS	0.014	0.265	0.323	0.024	0.156
ESCO	0.019	0.341	0.367	0.032	0.190

a) Capacidad: 2 Kbytes

ocupación bus de instr.	LEX	NROFF	PCC	YACC	Media
SR	0.010	0.261	0.268	0.015	0.139
SRA	0.011	0.287	0.271	0.018	0.147
SRAP	0.009	0.155	0.303	0.013	0.120
SSR	0.007	0.168	0.276	0.009	0.115
MIDS	0.007	0.150	0.264	0.008	0.107
ESCO	0.009	0.198	0.309	0.011	0.132

b) Capacidad: 4 Kbytes

Tabla 5.5: Tráfico entre memoria cache y memoria externa medido en porcentaje de tiempo que el bus de instrucciones está ocupado.

Parámetros de la cache:

- Mapeo directo
- Capacidad: 2 ó 4 Kbytes
- Tamaño de línea: 4 instrucciones
- Latencia de memoria externa: 4 ciclos
- Protocolo modo ráfaga

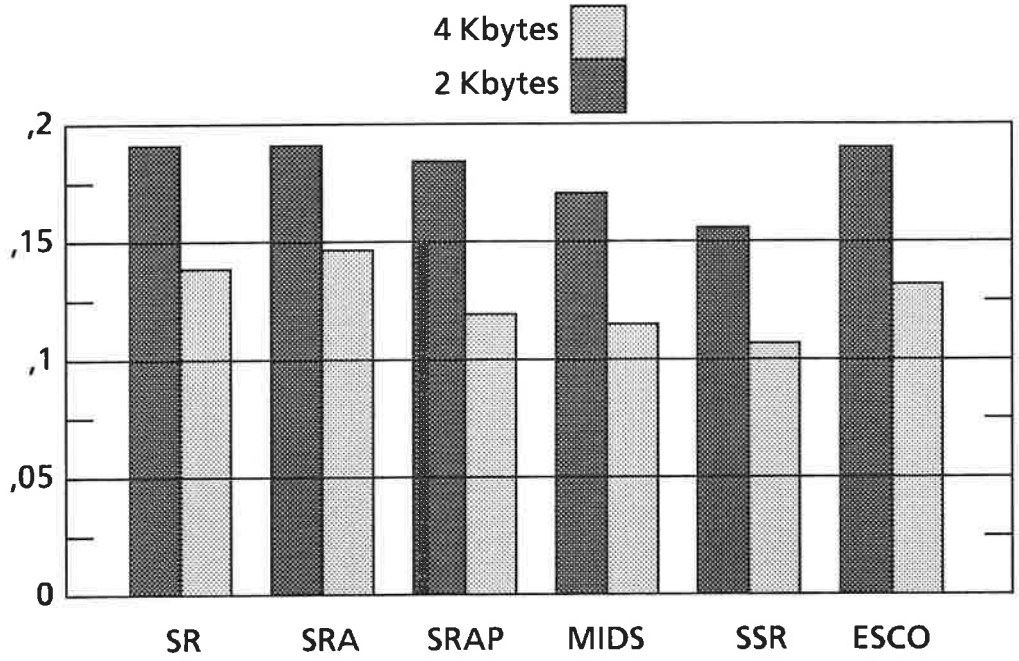


Figura 5.4: Tráfico entre memoria cache y memoria externa medido en porcentaje de tiempo que el bus de instrucciones está ocupado.

Parámetros de la cache:

- Mapeo directo
- Capacidad: 2 ó 4 Kbytes
- Tamaño de línea: 4 instrucciones
- Latencia de memoria externa: 4 ciclos
- Protocolo modo ráfaga

CAPITULO 6

Diseño y evaluación de la UI con una memoria de instrucciones destino de salto

En este capítulo se analiza la implementación del mecanismo ESCO utilizando como memoria cache de instrucciones una memoria de instrucciones destino de salto [GoLl89]. Para obtener los parámetros de diseño de la UI se desarrolla un modelo analítico que caracteriza el comportamiento del mecanismo ESCO con esta organización de memoria. A continuación se presenta un diseño de la UI. Seguidamente se evalúa el rendimiento de este mecanismo y se comparan los resultados con el rendimiento obtenido mediante el esquema de salto retardado junto con la misma organización de memoria supuesta para el mecanismo ESCO. Finalmente se analiza la eficiencia del mecanismo ESCO al utilizar prebúsqueda temporal en lugar de espacial.

6.1. ORGANIZACION DE MEMORIA

Una memoria de instrucciones destino de salto (MIDS) es una memoria cache cuya unidad de mapeo es un super bloque básico dinámico (SBBD: conjunto de instrucciones entre dos saltos efectivos consecutivos, incluyendo únicamente el último). Esta memoria es accedida mediante la dirección efectiva de las instrucciones de salto y contiene un número fijo de instrucciones consecutivas situadas a partir de la dirección destino del salto. Para acceder a las instrucciones que siguen en secuencia a las almacenadas en una línea de cache utilizaremos un protocolo modo ráfaga . Una organización de memoria con estas características es utilizada por el procesador Am29000 [AdMD87] y ha sido evaluada para los procesadores VAX y RISC II en [OlVR88].

Definimos la *anticipación* con que una instrucción es accedida como el número de ciclos transcurridos desde su acceso hasta el ciclo en que debería iniciarse su ejecución. Una anticipación negativa significa que la instrucción es accedida en el mismo ciclo o más tarde que el ciclo en que debería empezar su ejecución, lo que implica una cierta penalización en el rendimiento del procesador.

Dada una memoria externa cuya latencia es R y a la que se accede mediante protocolo modo ráfaga, para conseguir que esta memoria sirva una secuencia de instrucciones con una anticipación X, será preciso lanzar la petición a la primera instrucción de esta secuencia con una anticipación de $R+X$ ciclos.

La idea básica de como utilizaremos una MIDS junto con el mecanismo ESCO o el mecanismo salto retardado es la siguiente. Para conseguir acceder a una secuencia de instrucciones con anticipación X podríamos disponer de una memoria cache que nos proporcionara las $R+X$ primeras instrucciones. De esta forma, la petición que se lanza a memoria externa es para acceder a las instrucciones situadas $R+X$ posiciones siguientes al inicio de la secuencia. Tras los R ciclos de latencia, se habrán consumido R de las instrucciones proporcionadas por la memoria cache. En ese momento empezarán a llegar las instrucciones de la memoria externa, por lo que nos quedará una anticipación igual a X.

De esta forma, parte de las instrucciones entre dos saltos efectivos son suministradas por una línea de la MIDS y las restantes por la memoria externa.

Puede entenderse que la línea almacena parte de la localidad espacial del programa, que se incrementa automáticamente desde el punto de vista del procesador utilizando un protocolo de bus modo ráfaga y técnicas de prebúsqueda espacial. Este incremento de la localidad espacial almacenada en la MIDS se lleva a cabo sin penalización alguna si el retardo de acceso a la memoria externa es compensado por la anticipación con que se realiza el acceso.

A menos que se indique lo contratio supondremos que las líneas de la MIDS se rellenan completamente al ser reemplazadas.

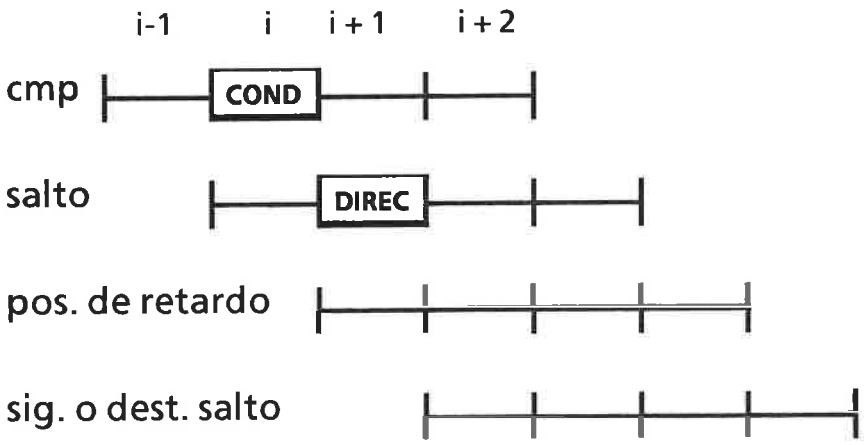
6.2. SALTO RETARDADO CON MIDS

En este apartado se analiza el funcionamiento de una procesador que implemente el mecanismo de salto retardado y haga uso de una MIDS. Este es el caso del procesador Am29000, aunque en este estudio asumimos la misma segmentación que a lo largo de este trabajo (figura 2.8), que es diferente de la de este procesador. Nuestro interés en este mecanismo reside en desarrollar un modelo analítico que caracterice su comportamiento, para más tarde obtener medidas de rendimiento que puedan ser comparadas con las del mecanismo ESCO.

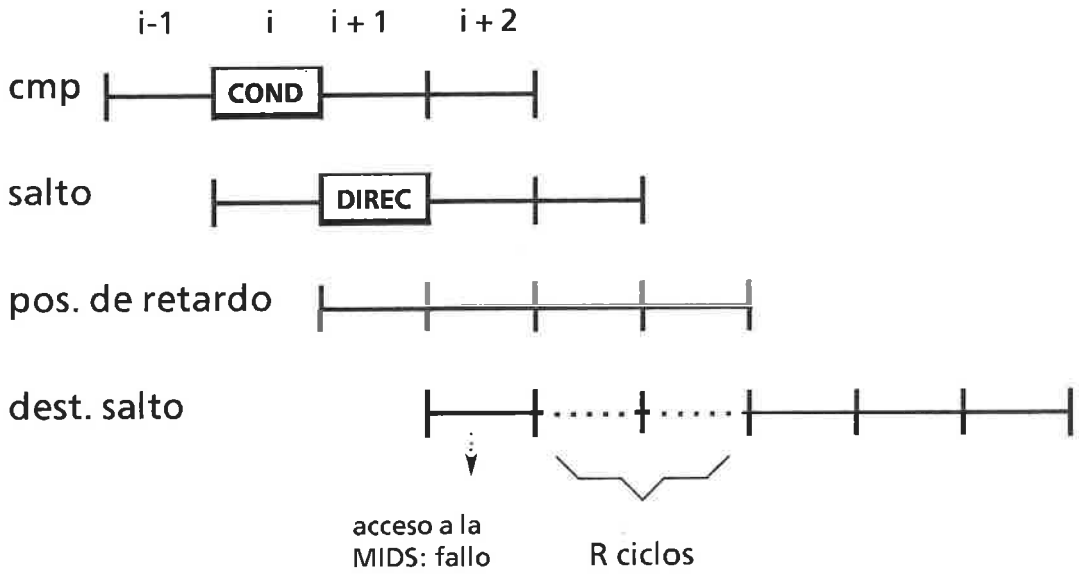
Las instrucciones de salto con el tipo de segmentación asumida tendrán una posición de retardo. Los pasos en la ejecución de un salto se detallan seguidamente y se han representado gráficamente en la figura 6.1.

Supongamos que se efectúa la búsqueda del salto en el ciclo i. En el ciclo $i+1$ se calcula la dirección destino del salto a la vez que se busca la instrucción siguiente al salto que será la situada en la posición de retardo. Al iniciarse el ciclo $i+2$ se conoce por completo el resultado del salto, tanto la dirección destino como el resultado de la condidición.

Si el salto es efectivo, en el ciclo $i+2$ se accede a la MIDS con la dirección destino del salto y si la petición requerida se encuentra en la MIDS, la UE puede empezar a ejecutar la rama destino del salto sin ningún retardo. Además, en este caso se efectúa una petición a memoria externa de las instrucciones que siguen en secuencia a la última instrucción almacenada en la línea de cache (prebúsqueda espacial). Para ello ha sido necesario calcular en paralelo con el acceso a la MIDS la dirección de la primera instrucción que sigue en secuencia a la línea de cache.



a) salto no efectivo o salto efectivo con acierto en la MIDS



b) salto efectivo con fallo en la MIDS

Figura 6.1: Ejecución de un salto mediante el mecanismo de salto retardado con una MIDS.

Si el acceso a la MIDS produce un fallo de cache, la petición que se realiza a memoria externa es para leer la secuencia de instrucciones a partir de la dirección de salto. En este caso, la UI no podrá enviar nuevas instrucciones a la UE hasta que la primera instrucción del destino del salto llegue de memoria externa.

En el caso de que el salto no sea efectivo se alimenta al procesador con la siguiente instrucción en secuencia. Esta puede proceder de la línea de cache accedida en el último salto efectivo o bien ser suministrada por la memoria externa mediante el protocolo modo ráfaga.

Con este esquema, la anticipación con que debe accederse cualquier instrucción es igual a cero. Por lo tanto, para conseguir el máximo rendimiento del mecanismo bastará que el tamaño de línea de la MIDS sea igual a la latencia de la memoria externa.

6.2.1. Definiciones previas

En el modelo analítico que vamos a desarrollar a continuación utilizaremos las definiciones siguientes:

- Sea T la probabilidad de que un salto sea efectivo.
- Sea H la tasa de aciertos a la MIDS. Definimos la tasa de aciertos a la MIDS como la probabilidad de que una determinada línea se encuentre en memoria cuando es requerida.
- Sea R la latencia de la memoria externa. El número de instrucciones de una línea es también R.
- Sea Po la probabilidad de optimizar la posición de retardo asociada a un salto.
- Sea $D(d)$ la función de densidad de probabilidad de la distancia (número de instrucciones) entre dos saltos efectivos, incluyendo el último y su posición de retardo y excluyendo el primero y su posición de retardo.

6.2.2. Modelo analítico para el salto retardado

En el mecanismo salto retardado con una MIDS tenemos que el coste medio de una instrucción de salto es la suma de los cuatro apartados siguientes:

a) Ejecución de la instrucción de salto: *1* ciclo.

b) No optimización del slot: *1-Po* ciclos. Valores típicos de *Po* están alrededor de 0.6.

c) Fallo de la MIDS en el acceso a la primera línea destino de un salto efectivo: *T (1-H) R* ciclos.

d) Salto efectivo que ocurre antes de finalizar por completo el reemplazo de la línea correspondiente al anterior fallo de cache: *T (1-H) Nc*. Donde *Nc* es el número medio de entradas de la línea que faltan por rellenar. Su valor viene dado por la siguiente expresión:

$$N_c = \sum_{d=2}^{R-2} D(d)\,(R-1-d)$$

La diferencia entre el rendimiento del procesador estimado mediante este modelo y los resultados obtenidos mediante simulación para los cuatro programas de prueba utilizados a lo largo de este trabajo y 15 conjuntos diferentes de parámetros del sistema de memoria es siempre menor que un 0.22% y el valor medio de la diferencia es de un 0.05%.

Este modelo puede ser también utilizado para evaluar el mecanismo salto retardado con anulación. La única diferencia es el valor de *Po* y un mínimo cambio en la función *D(d)* debido a la distinta optimización de las posiciones de retardo.

6.3. MECANISMO ESCO CON UNA MIDS

En este apartado se presenta un descripción del comportamiento del mecanismo ESCO con una MIDS. Seguidamente se desarrolla un modelo analítico que caracteriza este comportamiento.

La organización de la memoria cache elegida (MIDS) permite obtener el siguiente bloque básico dinámico (BBD) que hay que ejecutar en el caso de que el salto no sea efectivo de forma sencilla. Está almacenado en la línea de la MIDS obtenida en el anterior salto efectivo, o lo suministra directamente la memoria externa debido a la prebúsqueda espacial. En el caso de que el salto sea efectivo la MIDS puede tener almacenado en alguna de sus líneas el BBD necesitado.

Para ejecutar los saltos con coste cero, en caso de producirse un salto efectivo, la UI debe localizar el siguiente BBD con la anticipación suficiente para que no se interrumpa el suministro de instrucciones útiles a la UE. Para obtener este nuevo BBD la UI accede a la MIDS. Para acceder a esta memoria es necesario disponer de la dirección destino de salto. Por lo tanto, será necesario detectar las instrucciones de salto con una determinada anticipación.

Parte de las instrucciones de un SBBD son suministradas por la MIDS y parte por la memoria externa. En el caso de la MIDS, el circuito de prebúsqueda de saltos analiza en paralelo la línea de la MIDS y detecta los saltos almacenados en el orden determinado por su posible ejecución. Esto implica que la anticipación máxima es igual al tamaño de línea menos uno. Los resultados presentados posteriormente muestra que es suficiente analizar las instrucciones con una anticipación de un ciclo.

Si un salto no efectivo está almacenado en la línea de la MIDS, la anticipación en el siguiente salto será como máximo el tamaño del nuevo BBD que debe de ejecutarse menos una unidad.

Una vez es la memoria la que suministra las intrucciones éstas son analizadas de una en una para detectar los saltos. En este caso la anticipación con que los saltos serán detectados vendrá determinada por el tamaño de línea, el tiempo de acceso a memoria externa y el número de saltos no efectivos entre el último salto efectivo y el salto actual.

Cuando la memoria externa empieza a suministrar las intrucciones, la anticipación disminuye a $B\text{-}L\text{-}N_{ne}$, donde B es el tamaño de la línea de cache, L es la latencia de memoria externa y N_{ne} es el número de saltos no efectivos en la línea proporcionada por la MIDS. A partir de este instante, cada salto no efectivo disminuye en una unidad la anticipación actual.

En el modelo propuesto suponemos que un fallo en el acceso a la MIDS al efectuar la prebúsqueda de la rama destino de un salto no aborta la transacción iniciada en el anterior salto efectivo hasta que se sabe con certeza que el salto debe ser efectivo.

6.3.1. Modelo analítico

El coste de un salto es función de la anticipación con que dicho salto es detectado. Esta anticipación está determinada por el flujo de instrucciones ejecutado anteriormente. Ahora bien, para determinar el coste de un salto no es preciso analizar toda la historia anterior. Es suficiente con remontarse hasta el último salto efectivo. Este actúa como un punto de regeneración ya que en ese punto se inicia el acceso a una nueva línea de la MIDS cuyo resultado es un acierto o un fallo. En cualquiera de ambos casos, la anticipación en ese instante queda determinada y no depende de la historia anterior.

Por lo tanto, para evaluar el coste de un salto hay que analizar todos los posibles caminos que nos han podido llevar desde el anterior salto efectivo al salto que estamos analizando. Calculando la probabilidad de cada uno de estos caminos y el coste del salto debido al camino podemos determinar cual es el coste medio de un salto. El desarrollo analítico de este modelo sin efectuar ninguna relajación es bastante complejo. Este desarrollo puede encontrarse en el apéndice A. En este apartado presentamos el modelo obtenido al realizar la siguiente simplificación:

- El tamaño del SBBD al que pertenece un determinado salto es independiente del número de saltos no efectivos que le preceden en ese mismo SBBD.

Además de esta simplificación utilizaremos dos relajaciones posibles que dan lugar a modelos cuya divergencia con el modelo del apéndice A es menor del 1%. Estas relajaciones son las siguientes:

1) El número de instrucciones de salto de un SBBD es independiente de su longitud.

2) El número de instrucciones de salto de un SBBD es directamente proporcional a su longitud.

En estos dos modelos se supone además que nunca existe un salto sobre salto o seguido de otro salto. Con esta hipótesis se simplifica su desarrollo cometiendo un error mínimo, ya que la probabilidad de que esto ocurra es baja. Los modelos correspondientes a estas dos relajaciones se exponen seguidamente.

El coste medio de un salto lo evaluaremos a partir de los siguientes parámetros:

1) Probabilidad de que el salto sea efectivo (T).

2) Probabilidad de acierto en la MIDS (H).

3) Función de densidad de probabilidad de la distancia entre dos saltos efectivos, $D(d)$ (esta función da una idea de la localidad espacial del programa ejecutado).

4) Tamaño de línea de la MIDS. Para facilitar el análisis hemos descompuesto este parámetro en dos términos ($R+K$):

   - R que es igual a la latencia de memoria externa.

   - K que representa la anticipación con que se analizarán, hasta la primera instrucción de salto, las instrucciones suministradas por la memoria para la detección de una instrucción de salto.

   Analizaremos únicamente aquellos casos en que K es mayor o igual a cero. Si el tamaño de línea es menor que la latencia de memoria se producirán muy a menudo retardos en el suministro de instrucciones al procesador, a menos que el tamaño de línea sea lo suficientemente grande para que permita en la mayoría de los casos almacenar en una línea todo el bloque de instrucciones entre dos saltos efectivos.

Para evaluar el coste medio de un salto vamos a caracterizar las cuatro causas posibles (figura 6.2) que ocasionan que el coste de un salto sea diferente a cero y

evaluaremos su contribución al coste medio de un salto. Estas causas son las siguientes.

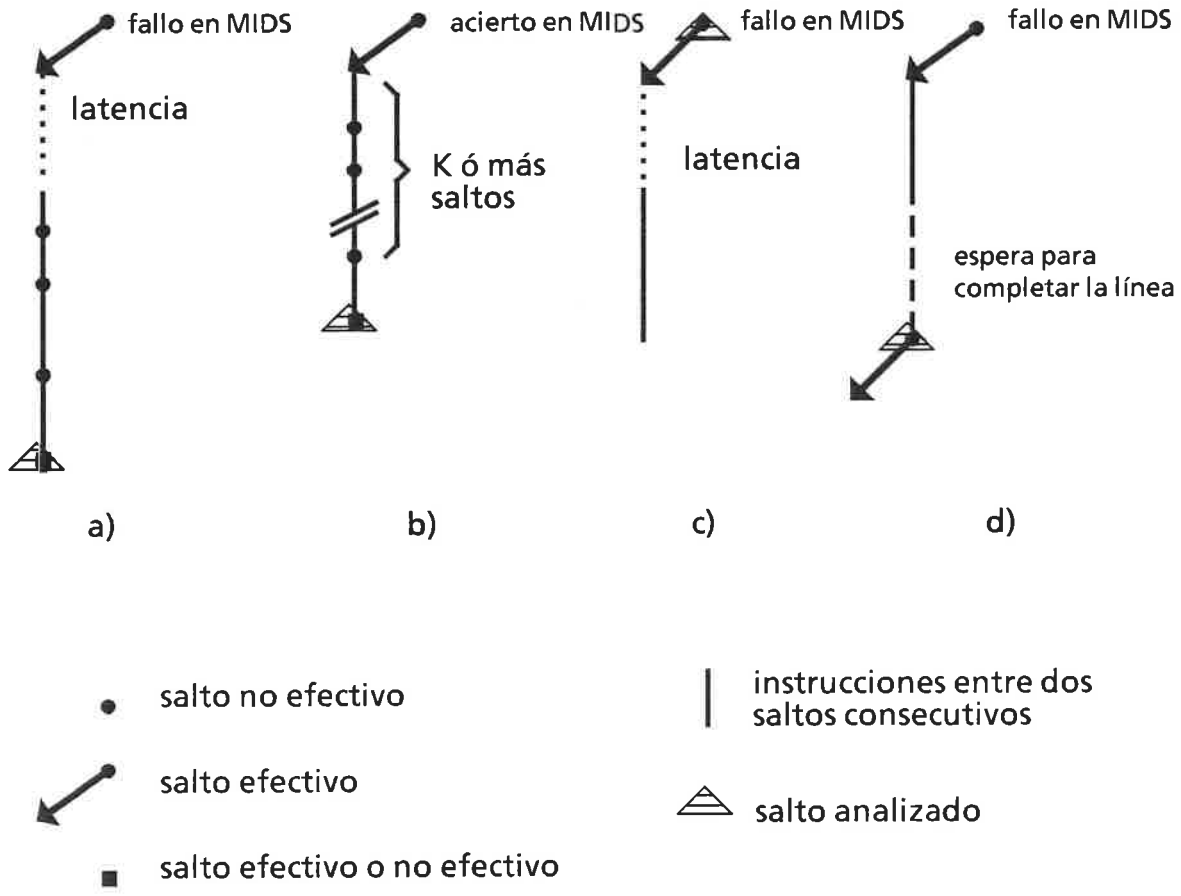


Figura 6.2: Causas que producen que el coste de un salto sea diferente a cero: a) Falta de anticipación debido a fallo en MIDS, b) pérdida de anticipación debida a los saltos no efectivos, c) latencia de memoria debida a fallo en MIDS y d) reemplazo completo de una línea de la MIDS.

a) *Falta de anticipación debido a un fallo en MIDS*: El acceso a la MIDS en el salto efectivo precedente ha producido un fallo. Entre el salto actual y el salto efectivo precedente hay cualquier número de saltos no efectivos. En este caso el coste del salto es un ciclo y por lo tanto, la contribución de esta

causa al coste medio de un salto es:

$$\sum_{j=0}^{\infty} T(1-H)(1-T)^j = 1-H$$

b) *Pérdida de anticipación debida a los saltos no efectivos*: El acceso a la MIDS en el salto efectivo precedente ha producido un acierto. Entre el salto actual y el salto efectivo precedente hay K o más saltos no efectivos. Además, el salto analizado pertenece a un SBBD cuyo tamaño es mayor que $R+K$ instrucciones ($R+K$ = tamaño de línea de la MIDS). El coste de este salto es de un ciclo.

Un caso particular es cuando tanto el salto analizado como los K o más saltos no efectivos precedentes se encuentren todos ellos almacenados en la línea obtenida de cache. En este caso se le asocia un coste nulo a los K primeros saltos no efectivos y un coste igual a un ciclo a los restantes saltos almacenados en la línea. Realmente todos estos saltos serán ejecutados con coste cero ya que la anticipación para cada uno de ellos es igual al tamaño de su BBD. Sin embargo, cada salto no efectivo ocasiona una disminución de una unidad en la anticipación con que la memoria externa va a suministrar las instrucciones que siguen a la línea de cache. Por lo tanto, cuando la UE haya consumido todas las instrucciones de la línea de cache, deberá esperarse un número determinado de ciclos a que lleguen las siguientes instrucciones de la memoria externa. Estos ciclos son los que contabilizamos como coste del salto, ya que de hecho son las instrucciones de salto las causantes de que ocurran.

La contribución de esta causa al coste medio de un salto es:

$$[1-G_{AC}(R+K)]\sum_{j=K}^{\infty} T\,H(1-T)^j = [(1-G_{AC}(R+K)]\,H(1-T)^K$$

donde

$$G_{AC}(t) = \sum_{d=1}^{t} G(d)$$

y $G(d)$ representa la probabilidad de que un salto pertenezca a un bloque

de instrucciones entre dos saltos efectivos cuyo tamaño es d instrucciones. Si utilizamos la relajación 1) tenemos que $G(d) = D(d)$ mientras que para la relajación 2)

$$G(d) = \frac{D(d)\,d}{\sum_{i=1}^{\infty} D(i)\,i}$$

c) *Latencia de memoria debida a un fallo en MIDS*: El salto analizado es efectivo y al acceder a la MIDS no se encuentra la línea deseada. El coste de de este fallo es de R ciclos y su contribución al coste medio de un salto es $RT(1\text{-}H)$ ciclos.

d) *Reemplazo completo de una línea*: En el acceso realizado en el anterior salto efectivo se ha producido un fallo en la MIDS, el salto analizado es efectivo y la distancia entre estos dos saltos (d) es menor que $R+K\text{-}1$. Los ciclos adicionales para completar el reemplazo son $R+K\text{-}1\text{-}d$ y la contribución de esta causa al coste medio de un salto es:

$$(1-H)\,T \sum_{d=1}^{R+K-2} D(d)\,(R+K-1-d)$$

6.3.2. Validación del modelo

La validez del modelo analítico ha sido comprobada comparando sus resultados con los obtenidos mediante simulación para los cuatro programas de prueba: LEX, NROFF, PCC y YACC. Esta comparación se ha realizado para diferentes valores de tamaño de cache, tamaño de línea y tiempo de acceso a memoria externa. En total se han evaluado 31 conjuntos de valores diferentes.

Los parámetros de entrada al modelo referentes a las características de los programas ejecutados (T, H, $D(d)$) han sido obtenidos mediante simulación utilizando las técnicas comentadas en el capítulo 3.

Es lógico esperar que los resultados del modelo sean algo optimistas debido a que no tienen en cuenta los saltos sobre saltos ni los saltos calculados (no relativos al PC).

Con la hipótesis $G(d) = D(d)$, la discrepancia entre el modelo y la simulación es como máximo del 4.93% y en media del 1.93%..

Con la hipótesis

$$G(d) = \frac{D(d)\,d}{\sum_{i=1}^{\infty} D(i)\,i}$$

la discrepancia máxima entre el rendimiento estimado por el modelo y el obtenido mediante simulación es del 3.76% y en media 1.36%.

6.4. ANALISIS DE LOS PARAMETROS DE DISEÑO

En esta sección vamos a utilizar el modelo desarrollado anteriormente para analizar como influyen en el rendimiento del procesador diversos parámetros del sistema.

Para presentar los resultados obtenidos hemos elegido la función $D(d)$ y la probabilidad de que un salto sea efectivo (T) correspondientes al programa NROFF. Para el resto de programas (LEX, PCC, YACC) los resultados son muy similares desde el punto de vista de la influencia de los parámetros de diseño.

En primer lugar se analiza la influencia del tamaño de la línea de cache. Este determina la anticipación con que pueden detectarse las instrucciones de salto. El tamaño de línea mínimo analizado es igual al tiempo de acceso a memoria. En el estudio presentado se ha supuesto un tiempo de acceso a memoria de tres ciclos ($R=3$) y se ha evaluado el rendimiento del sistema para diversos tamaños de línea y diferentes tasas de aciertos. Los resultados obtenidos se presentan de forma gráfica en las figuras 6.3 y 6.4.

Para describir las gráficas presentadas efectuaremos un análisis del modelo desarrollado. El coste de los saltos debido a las causas a (falta de anticipación debido a un fallo en MIDS) y c (latencia de memoria debido a un fallo en MIDS) es independiente del tamaño de línea y disminuye de forma lineal al aumentar la tasa de aciertos. Para las otras dos causas, b (pérdida de la anticipación debido a

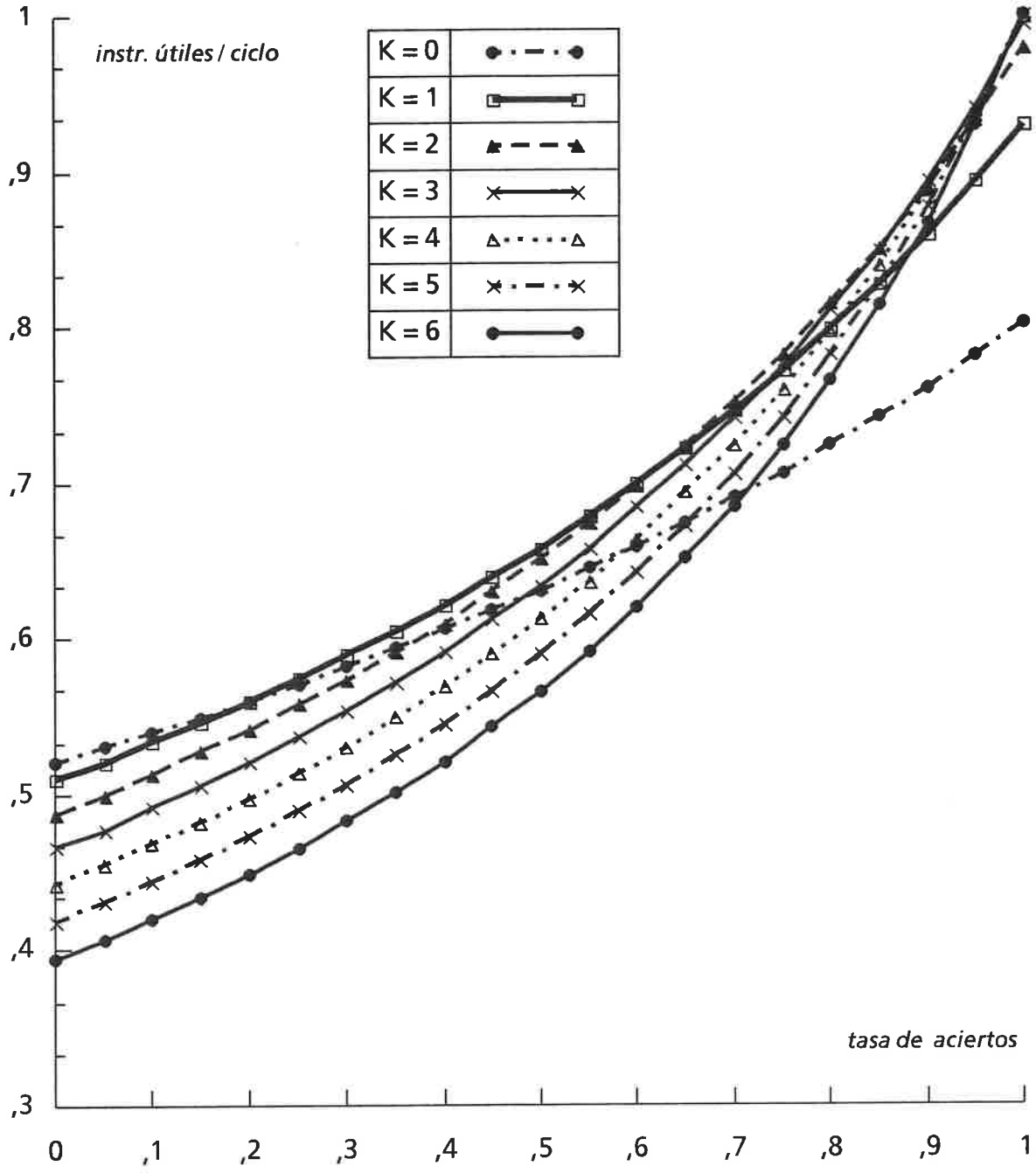


Figura 6.3: Rendimiento del procesador para diversos parámetros de la MIDS y latencia de memoria externa igual a 3 ciclos. (3 + K = Tamaño de línea de la MIDS)

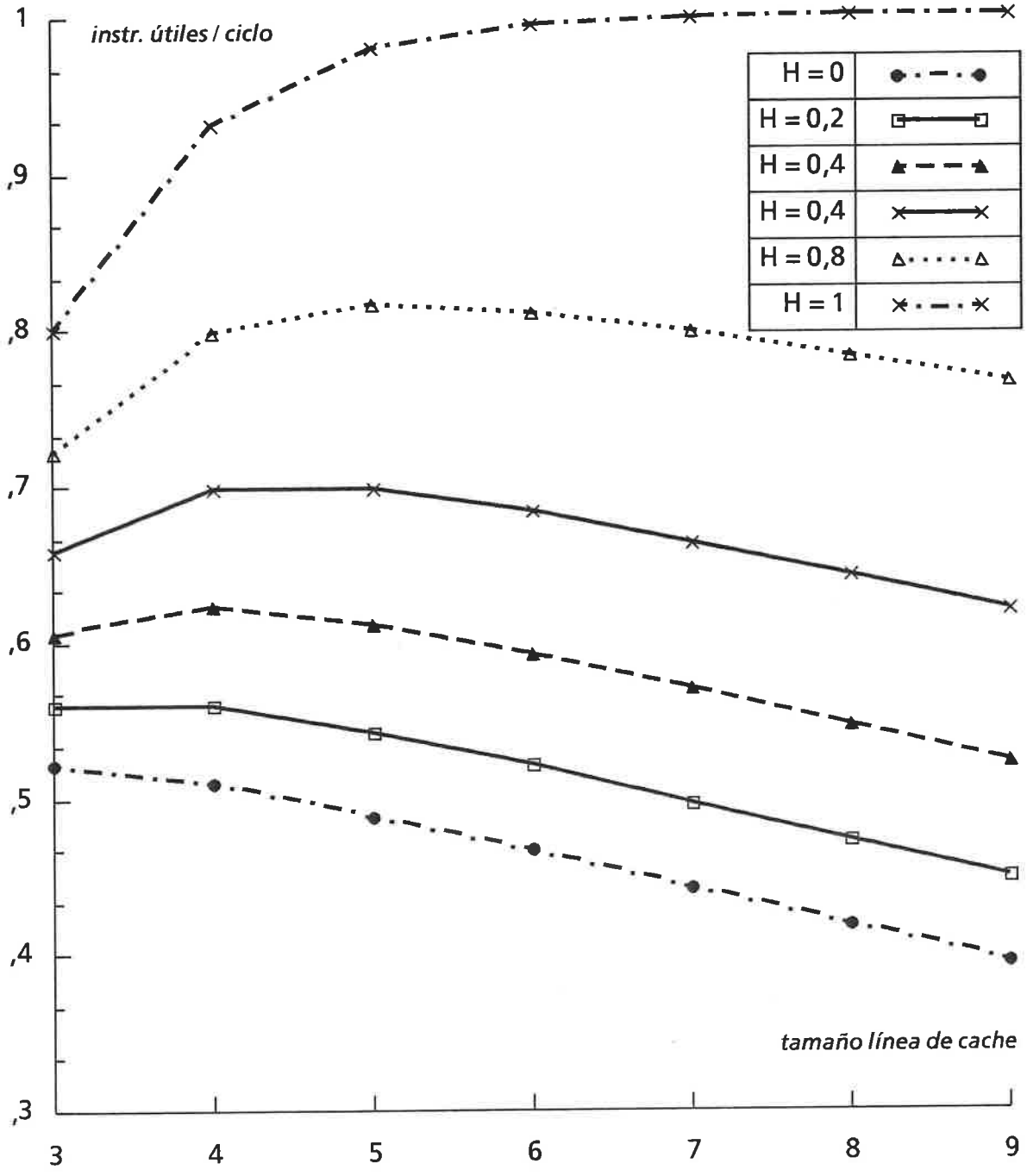


Figura 6.4: Rendimiento del procesador para diversos parámetros de la MIDS y latencia de memoria externa igual a 3 ciclos. (H: Tasa de aciertos de la MIDS)

los saltos no efectivos) y d (reemplazo completo de una línea), su contribución al coste medio de un salto depende del tamaño de línea y de la tasa de aciertos.

Ahora bien, en la causa b el coste del salto disminuye al aumentar el tamaño de línea, aumenta al incrementarse la tasa de aciertos y el rango de valores que puede tomar está entre cero y uno. En cambio, en los caminos caracterizados como d el coste del salto aumenta al aumentar el tamaño de línea, disminuye al incrementarse la tasa de aciertos y el rango de valores que puede tomar está entre cero e infinito. Sería infinito para el caso hipotético de líneas con tamaño infinito. No obstante, lo que queremos resaltar es que su valor no está acotado y puede ser mucho mayor que el de b.

En resumen, para tasas de acierto elevadas, un aumento del tamaño de línea tiene un efecto mayor en la causa b que en la d. Al contrario, para tasas de aciertos bajas, un aumento del tamaño de línea tiene un efecto mayor en d que en b. Al tener en cuenta además los rangos de valores de cada una de las causas se explica que, para una tasa de aciertos dada, el rendimiento mejora al aumentar el tamaño de línea, pero únicamente hasta un determinado tamaño. A partir de este tamaño, un incremento adicional del tamaño de línea produce una disminución del rendimiento del procesador. La segunda conclusión es que este tamaño a partir del cual el rendimiento empieza a decrecer es tanto mayor cuanto más elevada es la tasa de aciertos.

En segundo lugar analizaremos la influencia de la latencia de memoria. Para valores de latencia diferentes a tres ciclos obtendríamos curvas similares a las de la figura 6.3 pero dichas curvas se cruzarían para valores distintos de la tasa de aciertos.

Para latencias elevadas un aumento de K produce un beneficio mínimo ya que al ser el tamaño de línea $R+K$ bastante elevado el coste debido a la causa b (pérdida de anticipación debida a los saltos no efectivos) es muy bajo y por lo tanto su disminución va a ser mínima. Este coste pequeño se debe a que la probabilidad de que en una línea de cache haya un salto efectivo es muy alta ($G_{ac}(R+K) \rightarrow 1$). Por otra parte, como un aumento de K implica un aumento del tamaño de línea, esto supone un aumento del coste debido al tiempo de reemplazo (causa d). En consecuencia, para latencias mayores de tres ciclos la figura 6.3 tendría una forma similar pero los cruces entre curvas se producirían más a la derecha, es decir, para tasas de aciertos más elevadas.

Para valores de latencia muy pequeños, un aumento de K tiene un doble efecto positivo. Por una parte aumenta el número de saltos no efectivos consecutivos que pueden ejecutarse con coste cero. Por otra parte, debido al aumento del tamaño de línea ($R+K$), aumenta la probabilidad de encontrar un salto efectivo en la línea de cache. Por lo tanto, el aumento de K será tanto más beneficioso cuanto menor sea la latencia de memoria. En consecuencia para valores de latencia menores que tres ciclos obtendríamos que las curvas de la figura 6.3 se cruzarían más a la izquierda, es decir, al aumentar el K obtendríamos un incremento del rendimiento a partir de tasas de aciertos más bajas.

En la figura 6.5 puede observarse el efecto de incrementar el K para una determinada tasa de aciertos (0.8) y diferentes valores de latencia de memoria. Con K igual a 1 obtenemos siempre un rendimiento mejor que con K igual a cero, pero la diferencia entre ambos es cada vez menor al aumentar la latencia. El rendimiento con K igual a dos es mejor que con K igual a 1 pero sólo para latencias menores que 7. En resumen, el beneficio obtenido al incrementar K es tanto menor cuanto mayor es la latencia.

En la evaluación efectuada se ha supuesto que cuando al prebuscar las instrucciones de la rama destino del salto se produce un fallo en el acceso a la MIDS, no se inicia el acceso a la memoria externa de este flujo de instrucciones hasta que el resultado del salto indica que el salto debe ser efectivo. Por tanto, al detectar el fallo de cache no se aborta el protocolo de bus iniciado en el anterior salto efectivo. Es más, si el anterior salto efectivo ocasionó un fallo de cache, la búsqueda de la rama destino del salto no se inicia hasta que no se ha reemplazado completamente la línea anterior. En resumen, podemos decir que apostamos por una prebúsqueda espacial de instrucciones.

Los resultados obtenidos y el análisis del modelo indican que cuando el tamaño de la línea es bastante grande la probabilidad de que una línea contenga un salto efectivo es bastante elevada. Por tanto, la prebúsqueda espacial no aportará apenas ningún beneficio ya que en un gran número de casos encontraremos el SBBD en su totalidad almacenado en una línea de cache. En este caso podría pensarse en apostar por una prebúsqueda temporal. Es decir, cuando ocurre un acierto de cache, en lugar de prebuscar las instrucciones que siguen en secuencia a la línea, la UI detecta lo antes posible la primera instrucción de salto en la línea, pide a la MIDS que busque la línea destino de este salto y en caso de fallo de cache,

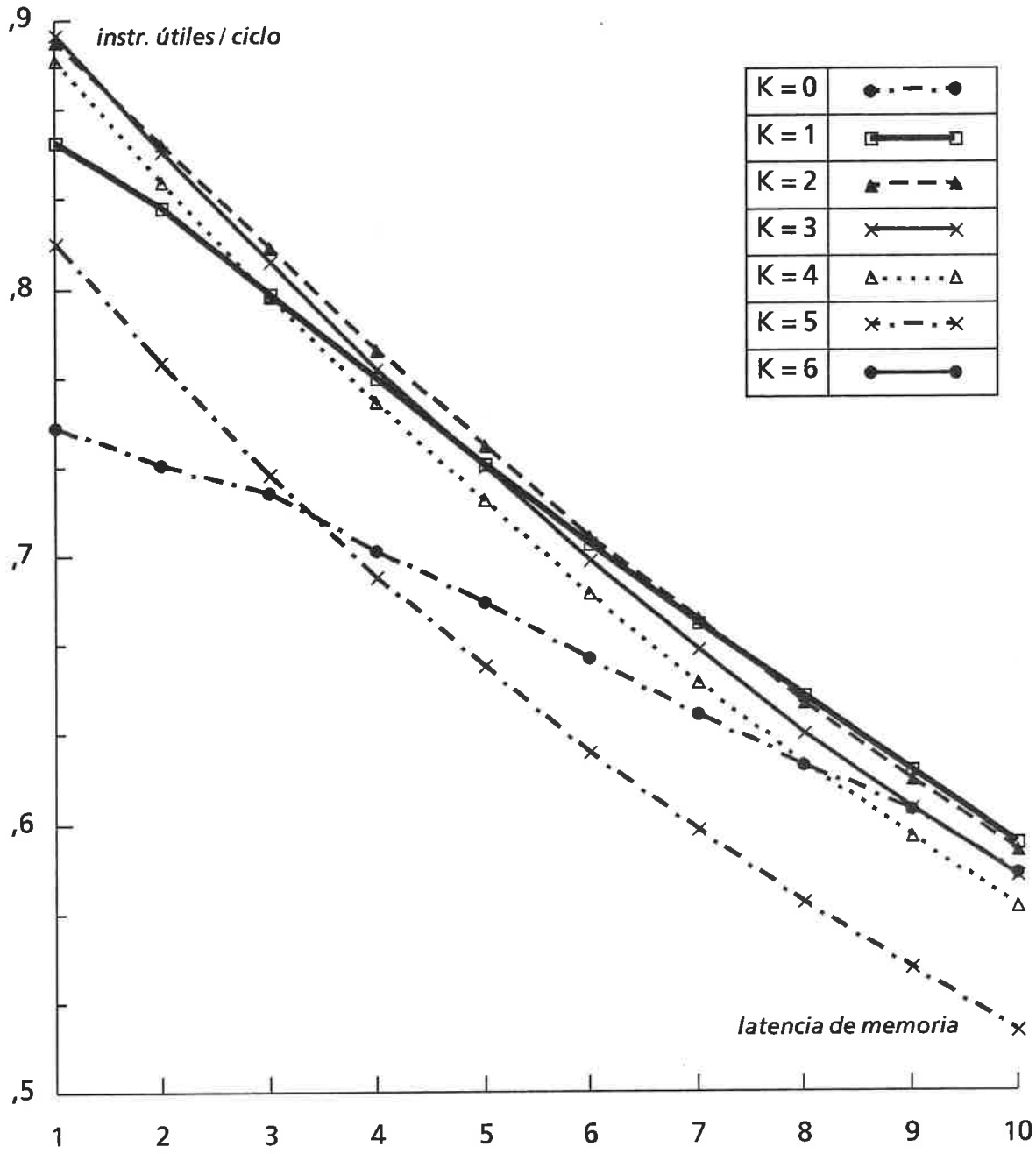


Figura 6.5: Rendimiento del procesador para diversos valores de latencia de memoria y diferentes valores de K (latencia + K = Tamaño de línea de la MIDS). La tasa de aciertos es 0.8 en todos los casos.

prebusca las instrucciones de la línea destino de este salto sin esperar a conocer el resultado del salto. Esta alternativa se analiza en un apartado posterior.

También se observa en el modelo que el coste del salto está muy influenciado por el hecho de que las líneas de cache deben ser reemplazadas en su totalidad (causa *d*). Este coste puede disminuirse permitiendo que las líneas no estén completamente llenas, aunque ello supone un mayor coste hardware para gestionar la MIDS. Una posibilidad sería añadir un bit cada X instrucciones de una línea que indicara si el contenido válido de la línea de cache finaliza o no en esa posición.

6.5. DISEÑO DE LA UI

En esta sección se presenta la UI propuesta y se justifica la elección de los parámetros de diseño.

6.5.1. Circuito de detección anticipada del salto

Las dos alternativas propuestas en la sección anterior para utilizar el bus que comunica la memoria cache de instrucciones con el siguiente nivel de la jerarquía de memoria dan lugar a diseños hardware de distinta complejidad. En el caso de prebúsqueda temporal hay que detectar la siguiente instrucción de salto lo antes posible. En cambio, en el caso de efectuar prebúsqueda espacial es suficiente detectar el salto con una anticipación de un ciclo.

En el caso de prebúsqueda temporal, el circuito de detección tiene que analizar en paralelo toda la línea obtenida de la MIDS para detectar la primera instrucción de salto correspondiente al BBD actualmente en ejecución (máxima anticipación).

En el caso de prebúsqueda espacial es suficiente analizar la instrucción que se va a enviar a la EU en el siguiente ciclo y la siguiente en secuencia. El circuito puede simplificarse sin pérdida de rendimiento si el compilador garantiza que nunca ocurrirá un salto a otro salto o un salto seguido de otro salto. Una vez detectado un salto se accede a la MIDS. Si se produce un acierto, en el siguiente ciclo se puede elegir, en función del resultado del test, entre cualquiera de los dos

flujos posibles de instrucciones, y por lo tanto el coste de ejecutar el salto será nulo.

6.5.2. Tamaño de la línea

A partir del análisis realizado en el apartado 6.3 y teniendo en cuenta que un aumento del tamaño de línea significa un incremento del área utilizada por la MIDS, consideramos que el mejor compromiso entre coste y eficiencia es elegir un valor de K igual a uno. Al comparar $K=1$ con $K=0$ se observa que el rendimiento del primero es mejor a partir de una tasa de aciertos bastante baja y que la diferencia entre ambos es considerable para valores típicos de la tasa de aciertos (0.6 - 0.7). Sin embargo, un incremento adicional en el tamaño de línea ($K \geq 2$) únicamente es beneficioso si la tasa de aciertos es bastante elevada. Además, este aumento de rendimiento es tan pequeño que no justifica el incremento de área.

6.5.3. Unidad de instrucciones

Los principales componentes de la unidad de instrucciones se muestran en la figura 6.6. La UI se compone de una MIDS junto con la circuitería necesaria para seleccionar la instrucción que debe alimentar a la unidad de instrucciones, detectar las instrucciones de salto con anticipación y para eliminarlas del flujo de instrucciones enviado a la UE.

La UI dispone de un registro para almacenar la línea suministrada por la MIDS en caso de acierto. No es preciso almacenar la primera instrucción de esta línea ya que debe ser enviada inmediatamente a la UE.

X_1 es un multiplexor que selecciona la instrucción que debe ser enviada a la UE. El multiplexor X_2 selecciona la instrucción siguiente a la seleccionada por X1. Esta instrucción es analizada por el circuito reconocedor de instrucciones de salto. En arquitecturas tipo RISC, reconocer si una instrucción es un salto puede ser tan sencillo como analizar un único bit de la instrucción. El circuito que genera las señales de control de cada multiplexor es básicamente un contador con la posibilidad de incrementar una o dos unidades en función de si se ha detectado o no un salto. Si el salto es efectivo estos contadores deberán reinicializarse para seleccionar la primera instrucción de la nueva rama.

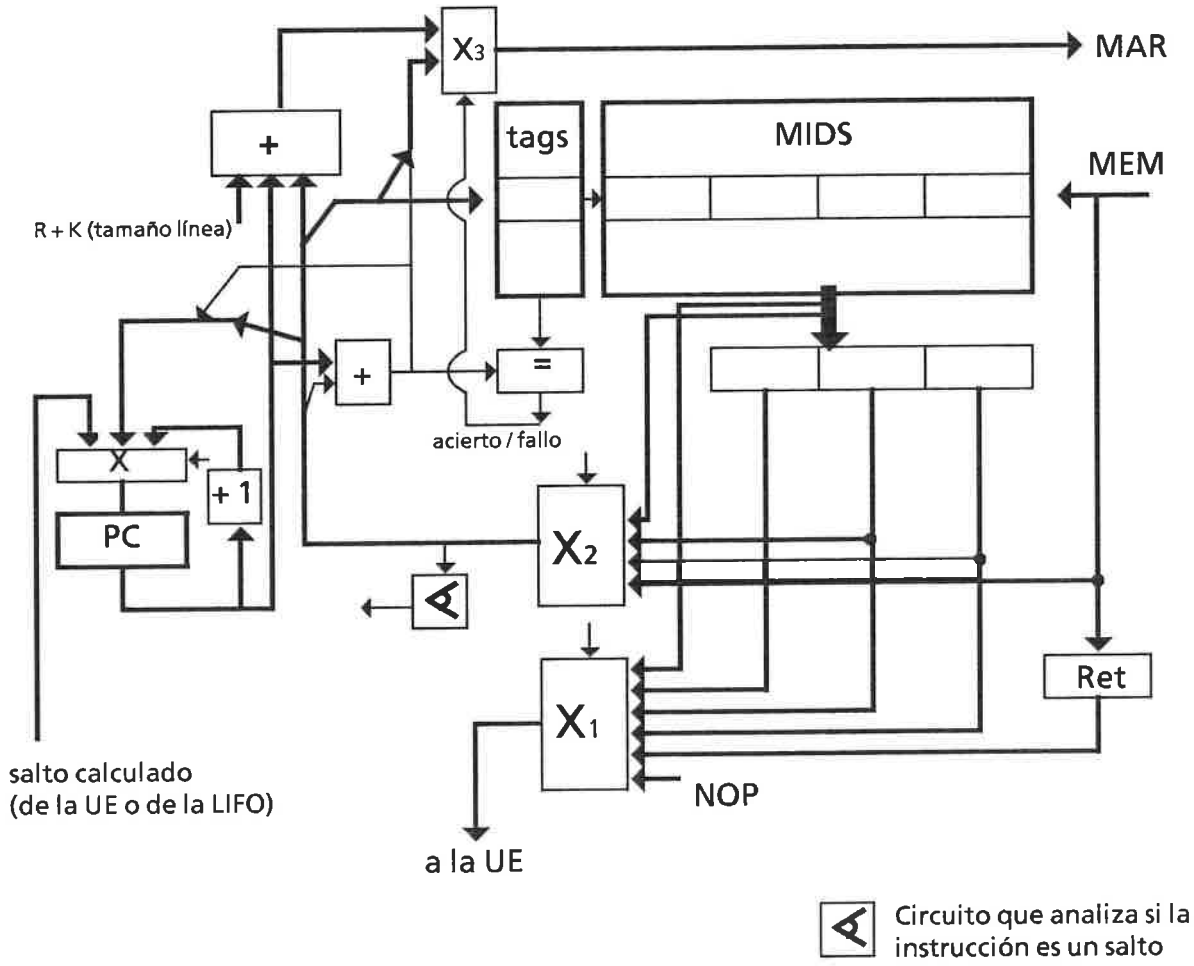


Figura 6.6: Diseño de bloques de la unidad de instrucciones

Las instrucciones suministradas por la memoria externa son analizadas por el circuito reconocedor de saltos tan pronto como llegan a la UI. El registro *Ret* tiene por finalidad retardar un ciclo su envio a la UE. De esta forma se consigue una anticipación igual a un ciclo.

Cuando se detecta una instrucción de salto, se accede a la MIDS para obtener la línea destino del salto mientras la instrucción anterior al salto está realizando la etapa de ALU. Al final de este ciclo, los códigos de condición determinan si el

salto debe ser efectivo. Si el salto es efectivo el registro PC se carga con la dirección de salto. X_3 selecciona la dirección que debe enviarse a la memoria externa. Si en el acceso a la MIDS se produce un fallo esta dirección es igual a la dirección destino del salto. En caso contrario, la dirección que se envía al exterior es la dirección de salto más el tamaño de la línea.

6.6. MEDIDAS DE RENDIMIENTO

En esta sección se presentan los resultados de evaluar el mecanismo ESCO con una MIDS. Los resultados son comparados con el rendimiento del mecanismo de salto retardado con y sin anulación suponiendo la misma organización de memoria cache.

La evaluación ha sido llevada a cabo mediante simulación de la ejecución de los programas de prueba: LEX, NROFF, PCC, YACC.

En la figura 6.7 se muestra el rendimiento del procesador para estos cuatro programas asumiendo una memoria de instruccciones destino de salto (MIDS) con mapeo directo y 32, 64, 128 o 256 entradas. Esta memoria está conectada al siguiente nivel de la jerarquía mediante un protocolo modo ráfaga cuya latencia es igual a tres ciclos. Para el mecanismo de salto retardado el tamaño de línea de cache es igual a la latencia de memoria (3 instrucciones), mientras que en el mecanismo ESCO este tamaño es igual a la latencia más una unidad (4 instrucciones). Si para cada tamaño de cache calculamos el rendimiento medio de cada mecanismo para los cuatro programas de prueba, tenemos que con el mecanismo ESCO se consigue un aumento de rendimiento respecto al salto retardado entre un 21 y un 28%. Cuanto más elevada es la tasa de aciertos mayor es esta diferencia de rendimiento.

Es interesante observar como el salto retardado con anulación en algunos casos puede tener un rendimiento peor que sin anulación. Ello se debe a que el salto retardado con anulación modifica la dirección destino de algunos saltos, incrementándola en una unidad, lo que puede ocasionar la aparición de nuevos SBBD y en consecuencia una disminución de la tasa de aciertos. Si observamos la figura 6.7 podemos ver que el rendimiento del salto retardado con anulación es superior al del salto retardado para los programas LEX y YACC, mientras que para los programas NROFF y PCC este rendimiento es del mismo orden, e incluso

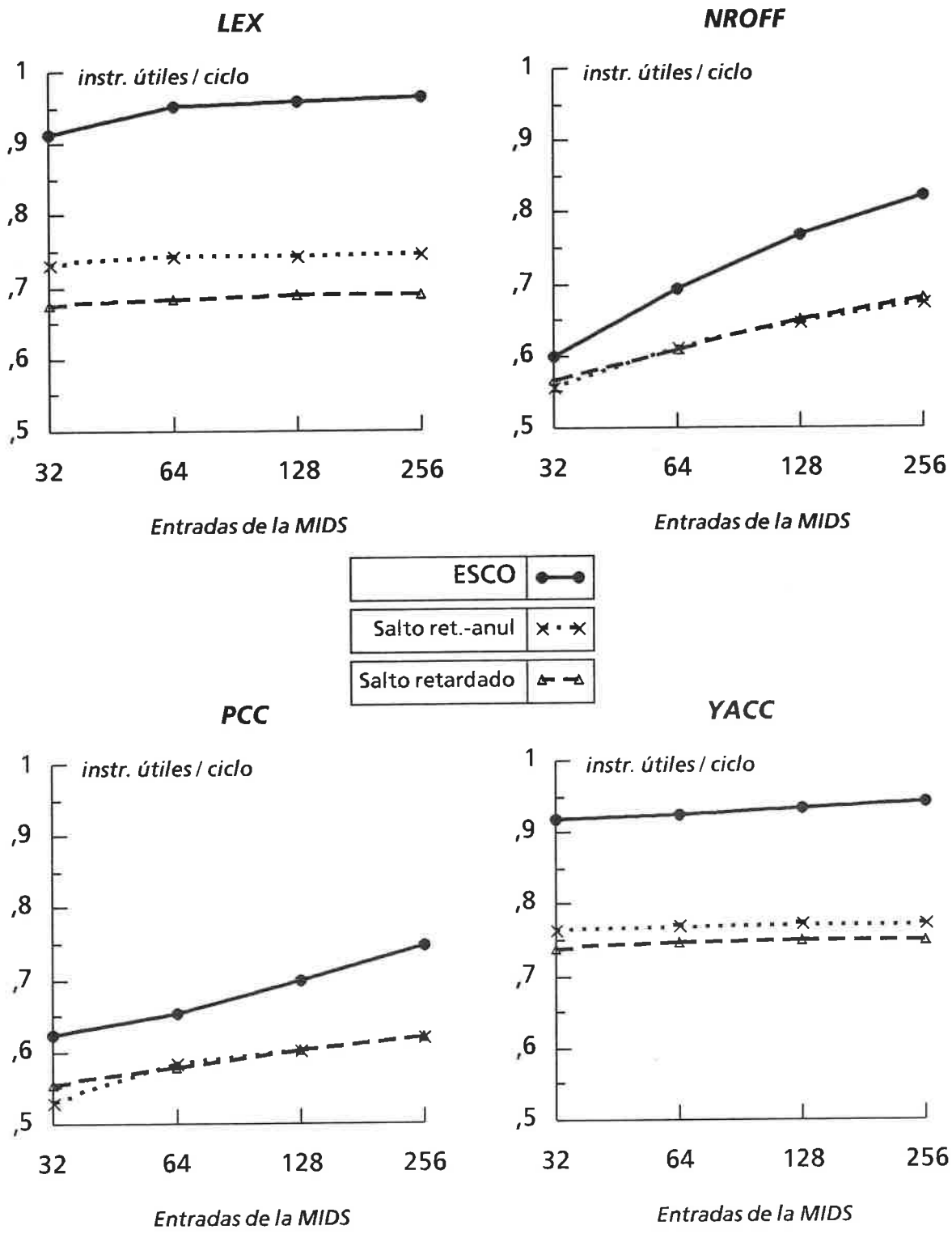


Figura 6.7: Rendimiento del procesador para los mecanismos ESCO, salto retardado y salto retardado con anulación. En todos ellos la memoria cache es una MIDS conectada mediante un protocolo modo ráfaga a la memoria externa. La latencia de esta memoria es igual a tres ciclos.

inferior para el salto retardado con anulación cuando la cache tiene pocas entradas. El motivo es que los dos primeros programas son bastante más locales, por lo que la aparición de nuevos SBBD apenas afecta a su tasa de aciertos. En cambio, en los programas NROFF y PCC que son bastante menos locales, estos nuevos SBBD tienen un efecto negativo sobre la tasa de aciertos, tanto mayor cuanto menor es el número de entradas de la cache, y por lo tanto, este efecto negativo contrarresta la ganancia obtenida por la posibilidad de anular las instrucciones de las posiciones de retardo.

En las figuras 6.8 y 6.9 se compara el comportamiento del mecanismo ESCO con una cache convencional y con una MIDS. En la figura 6.8 se representa el rendimiento del mecanismo y en la figura 6.9 puede observarse el tráfico generado con el siguiente nivel de la jerarquía de memoria.

En la figura 6.8 puede observarse que, para los parámetros de cache evaluados, el rendimiento para los programas LEX y YACC es mejor al utilizar una cache convencional mientras que para NROFF y PCC el rendimiento es superior con una MIDS. También puede observarse que para LEX y YACC, la diferencia de rendimiento entre ambas organizaciones de cache no es demasiado grande, mientras que para NROFF y PCC esta diferencia es considerablemente mayor, sobre todo para los tamaños más pequeños de cache.

Desde el punto de vista únicamente del rendimiento obtenido, la conclusión de este análisis podría ser la siguiente. Cuando la tasa de aciertos es muy elevada, el comportamiento de la cache convencional es ligeramente mejor que el de la MIDS mientras que para tasas de aciertos menos elevadas el rendimiento de la MIDS es considerablemente mejor que el de la cache convencional.

La justificación de este comportamiento es la siguiente. Cuando la tasa de aciertos no es muy elevada, la MIDS se comporta mejor ya que debido a que su unidad de mapeo es el SBBD, su contenido se adapta mejor a las peticiones de la UI. Por otra parte, cuando la tasa de aciertos es muy elevada, la organización de la cache no tiene un peso tan importante ya que en ambos casos el sistema de memoria consigue servir la mayoría de las peticiones de la UI con la anticipación requerida por ésta. La diferencia entre ambas organizaciones reside en que con la cache convencional la UI realiza una prebúsqueda más profunda de la rama que sigue al salto en secuencia. En concreto, la UI con cache convencional no tiene el problema debido a la pérdida de anticipación que en la MIDS ocasionaban los

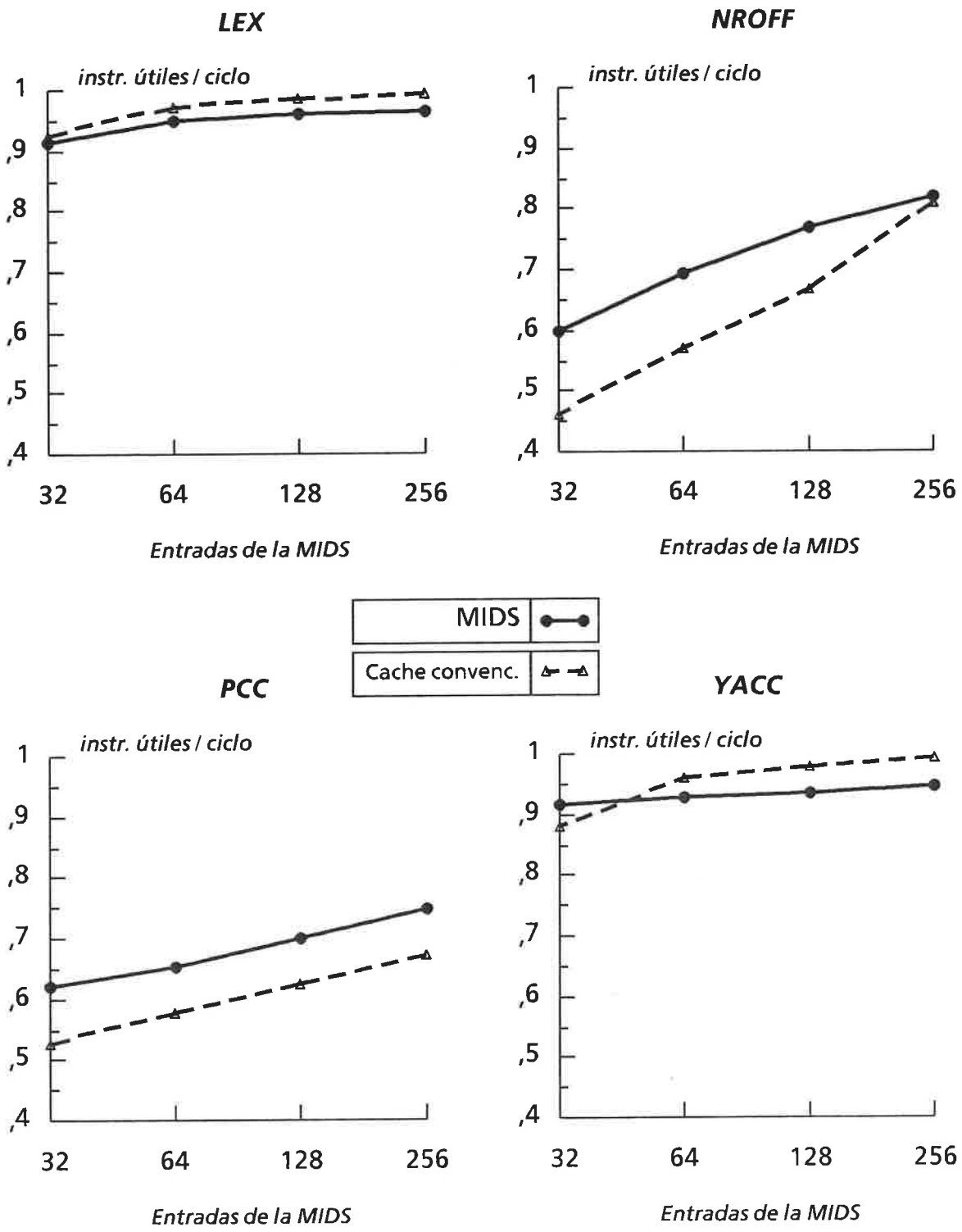


Figura 6.8: Rendimiento del mecanismo ESCO con una MIDS y con una memoria cache convencional. En ambos casos la memoria cache está conectada mediante un protocolo modo ráfaga a la memoria externa cuya latencia es igual a tres ciclos. El tamaño de línea es de 4 instrucciones.

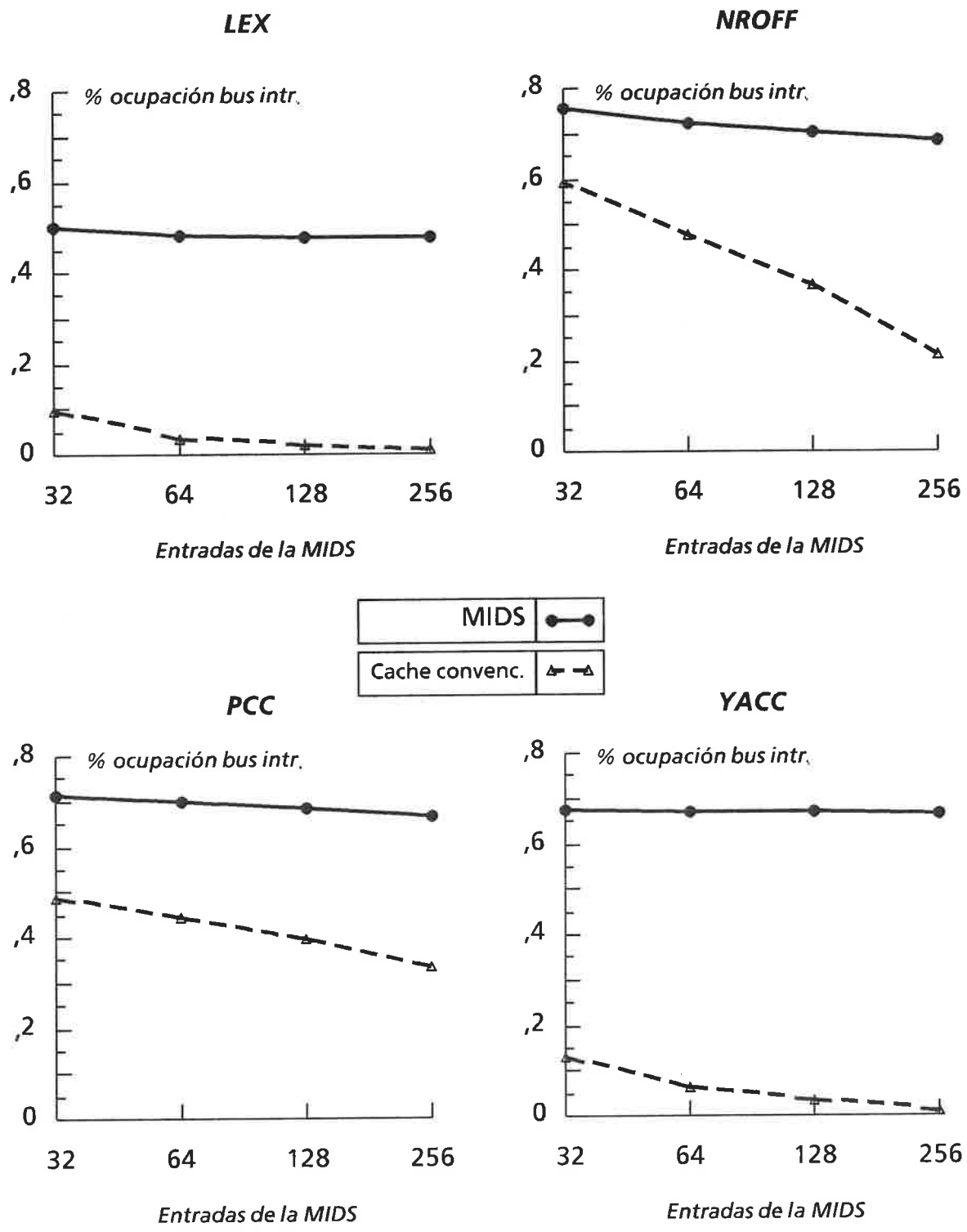


Figura 6.9: Tráfico generado por el mecanismo ESCO con una MIDS y con una memoria cache convencional. En ambos casos la memoria cache está conectada mediante un protocolo modo ráfaga a la memoria externa cuya latencia es igual a tres ciclos. El tamaño de línea es de 4 instrucciones.

saltos no efectivos. Ello es la causa de su mejor rendimiento, sin embargo, debido a que la probabilidad de que un salto no sea efectivo es pequeña (alrededor de 0.25), la ganancia de rendimiento no es demasiado significativa.

La figura 6.9 muestra que el tráfico con una MIDS es considerablemente superior que con una cache convencional. El motivo se debe a que en la MIDS existe una parte del programa que, independientemente del tamaño de cache y de la tasa de aciertos, debe ser siempre suministrada por la memoria externa. Esta parte del progama está determinada por el tamaño de los SBBD.

Al aumentar el tamaño de cache, disminuye el tráfico generado por las dos organizaciones de memoria. No obstante, esta disminución es significativamente mayor en la memoria cache convencional, debido a que en la MIDS existe esta parte de tráfico comentada en el párrafo anterior que es constante, independientemente del tamaño de cache. Ello significa que en una MIDS el tamaño de los SBBD tiene un peso en el tráfico generado que puede ser incluso mayor que el de la tasa de aciertos. Por ejemplo, en la figura 6.9 puede observarse que la MIDS genera prácticamente la misma cantidad de tráfico en los programas PCC y YACC, mientras que con la cache convencional este tráfico es considerablemente inferior en el YACC. El motivo es que la tasa de aciertos de memoria cache es bastante menor para el PCC que para el YACC. Sin embargo, el tamaño de los SBBD es mayor en el programa YACC (5.47 frente a 6.92 instrucciones en media), lo que hace que en la MIDS se compense el beneficio ocasionado por una tasa de aciertos más elevada con el tráfico adicional debido a que los SBBD son mayores.

6.7. PREBUSQUEDA TEMPORAL

Tal como apuntamos el el apartado 6.4, cuando el tamaño de la línea de cache es grande, la probabilidad de que en una línea de cache haya un salto efectivo es muy elevada por lo que podría pensarse en utilizar prebúsqueda temporal en lugar de espacial. En este apartado vamos a comparar el rendimiento obtenido con cada una de estas alternativas.

En el análisis presentado veremos en que casos la prebúsqueda temporal aporta un rendimiento mejor que la espacial y en que casos el rendimiento aportado es menor. Calcularemos el valor medio del beneficio y de la pérdida

ocasionada por la prebúsqueda temporal, lo que nos permitirá calcular la diferencia de rendimiento de ambos esquemas.

6.7.1 Hipótesis de trabajo

En el análisis que realizamos supondremos que la probabilidad de que una instrucción sea un salto es constante a lo largo de la ejecución de un programa, y que esta probabilidad es independiente de lo ocurrido en las anteriores instrucciones, es decir, de si éstas son saltos o no. Estas hipótesis implican que la función de densidad de probabilidad de la longitud de los BBD y de los SBBD sigue una ley geométrica.

El análisis presentado hace uso de las siguientes definiciones:

- Sea S la probabilidad de que una determinada instrucción sea un salto.
- Sea T la probabilidad de que un salto sea efectivo.
- Sea H la tasa de aciertos de la MIDS.
- Sea L el tamaño de la línea de la MIDS

6.7.2. Beneficio de la prebúsqueda temporal

El rendimiento de la prebúsqueda temporal será superior al de la espacial cuando una determinada línea de cache contenga un salto efectivo y el destino de este salto efectivo no se encuentre en la MIDS. Es decir, debe ocurrir que en el acceso al acual SBBD se produzca un acierto, que este SBBD quepa por completo en una línea de cache y que en el acceso al siguiente SBBD se produzca un fallo.

La ganancia obtenida será igual a la anticipación con que se inicie la prebúsqueda del siguiente SBBD respecto al instante en que se inicia cuando se utiliza prebúsqueda espacial. Esta anticipación viene determinada por el tamaño del último BBD dentro del actual SBBD y será igual, en número de ciclos, a este tamaño menos dos unidades.

Por lo tanto, el número medio de ciclos de beneficio por cada SBBD ejecutado es igual a

$$Bene = \sum_{l=3}^{L} P_l G_l H\,(1-H)$$

donde P_l representa la probabilidad de que la longitud de un SBBD sea igual a l y G_l es la ganancia obtenida en el caso de que la longidud del SBBD sea l.

Con las hipótesis realizadas tenemos que

$$P_l = (1-ST)^{l-1} ST$$

El valor de G_l depende del tamaño del último BBD. Si en el SBBD no existe ningún salto no efectivo, y por lo tanto el tamaño del último BBD es igual al del SBBD, esta ganancia será $l-2$ ciclos. Si hay algún salto no efectivo esta ganancia será igual al tamaño del último BBD menos dos unidades. En resumen, tenemos que

$$G_l = (1-S)^{l-1}(l-2) + \sum_{i=3}^{l-1} (1-S)^{i-1} S\,(i-2)$$

6.7.3. Pérdida de la prebúsqueda temporal

El rendimiento de la prebúsqueda temporal será inferior al de la espacial cuando el tamaño del actual SBBD sea superior a una línea de cache, la primera línea de instrucciones de este SBBD se encuentre en la MIDS, que en esta línea de cache haya algún salto no efectivo y que en la prebúsqueda de la rama destino de este salto no efectivo se produzca un fallo de cache. En este caso, los ciclos que dura la prebúsqueda del destino de este salto son de utilidad nula ya que el salto no será efectivo.

El número de ciclos perdidos será igual a la posición dentro de la línea del último salto no efectivo que produce un fallo de MIDS al prebuscar su rama destino.

El número medio de ciclos perdidos por cada SBBD ejecutado es igual a

$$Perd = Q_L H \sum_{i=0}^{L-1} E_i N_i$$

donde Q_L representa la probabilidad de que la longitud de un SBBD sea mayor que L, E_i es la probabilidad de que el último salto no efectivo dentro de la línea que ocasiona un fallo de cache esté situado en la posición $L-i$ y N_i es el número de ciclos perdidos en ese caso.

Estas funciones son igual a las siguientes expresiones. Q_L es la probabilidad de que en una línea de cache no haya ningún salto efectivo, por lo tanto

$$Q_L = (1-ST)^L$$

E_i sigue una ley geométrica de parámetro S (1 – T) (1 – H)

$$E_i = (1-S\ (1-T)\ (1-H))^i\ S\ \ (1-T)\ (1-H)$$

N_i es igual al número de ciclos que dura la ejecución de las instrucciones de la línea situadas antes del salto no efectivo. Ya que los saltos se ejecutan con coste cero este número es igual a

$$N_i = L-i-1-(L-i-1)S$$

6.7.4. Rendimiento neto

El rendimiento neto aportado por la prebúsqueda temporal es igual a $Bene-Perd$ ciclos por cada SBBD ejecutado. Ya que la longitud media de un SBBD es $1/ST$ y el número medio de instrucciones de salto de un SBBD es igual a 1/T, el rendimiento neto de la prebúsqueda temporal respecto a la espacial medido en ciclos por instrucción útil ejecutada es igual a

$$\Delta Rend = \frac{Bene - Perd}{\frac{1}{ST} - \frac{1}{T}}$$

6.7.5. Análisis de los resultados

Sustituyendo en las expresiones anteriormente presentadas los valores correspondientes a la tasa de aciertos (H), frecuencia de las instrucciones de salto (S), frecuencia con que los saltos son efectivos (T) y longitud de la línea de la MIDS (L), podemos hallar la diferencia de rendimiento entre la prebúsqueda temporal y la espacial.

La primera conclusión que obtenemos al realizar esta evaluación es que la diferencia de rendimiento entre ambos esquemas es mínima. En efecto, hemos evaluado esta diferencia de rendimiento para cualquier valor de H, S, T entre 0.1 y 0.9 y cualquier valor de L entre 1 y 50 obteniendo como valores máximo y mínimo de $\Delta Rend$ 0.08 y -0.16 ciclos por instrucción útil.

Para comprobar la validez de este resultado hemos evaluado mediante simulación el rendimiento del mecanismo ESCO con prebúsqueda espacial o temporal y tamaño de línea pequeño o grande para los 4 programas de prueba. Los resultados se muestran en la tabla 6.1. Puede apreciarse que el rendimiento de ambos esquemas es muy similar en todos los casos.

	Preb. Temporal		*Preb. Espacial*		*Temporal-Espacial*	
	Lin = 4	Lin = 60	Lin = 4	Lin = 60	Lin = 4	Lin = 60
LEX	0.952	0.844	0.959	0.840	-0.007	0.004
NROFF	0.759	0.154	0.767	0.153	-0.008	0.001
PCC	0.681	0.108	0.699	0.107	-0.018	0.001
YACC	0.901	0.712	0.934	0.709	-0.033	0.003

Tabla 6.1: Diferencia de rendimiento del mecanismo ESCO con prebúsqueda temporal y espacial.

En la figura 6.10 puede observarse como varía el valor de $\Delta Rend$ para diferentes valores de estos parámetros. Para ello hemos escogido 4 casos extremos. Para explicar los resultados de estas gráficas vamos en primer lugar a analizar las expresiones obtenidas anteriormente.

Si analizamos como influye el tamaño de línea en el valor de $\Delta Rend$ para valores fijos de H, S, y T podemos ver que un aumento de este tamaño aporta un incremento del valor de *Bene* (beneficio de la prebúsqueda temporal). Este aumento de rendimiento se debe básicamente a un aumento de la probabilidad de que la línea de cache contenga un salto efectivo. La magnitud de este aumento va a depender del valor de S, de forma que será mayor cuanto menor sea S.

Por otra parte un incremento del tamaño de línea tiene un efecto de doble signo sobre el valor de *Perd* (pérdida de la prebúsqueda temporal). El efecto positivo es que disminuye la probabilidad de que un SBBD no quepa en una línea de cache. Sin embargo, aumenta el número de ciclos pérdidos en el caso de que no se cumpla la anterior condición.

Para tamaños muy elevados de línea (supongamos $L \to \infty$), el rendimiento de la prebúsqueda temporal será superior al de la espacial ya que $Perd \to 0$. No obstante, la magnitud de esta diferencia no crece indefinidamente ya que cuando un determinado tamaño de línea es suficiente para poder albergar la mayoría de SBBD, un aumento adicional de este tamaño produce un incremento mínimo de rendimiento.

En las gráficas de la figura 6.10 puede observarse que las ganancias y pérdidas máximas se obtinen para tasas de aciertos intermedias. Ello se debe a que tanto en la expresión de *Bene* como de *Perd* aparece el factor $H(1-H)$ cuyo valor máximo se alcanza para valores intermedios de H.

En la primera gráfica ($S=0.1$ y $T=0.1$) el rendimiento de la prebúsqueda temporal es inferior al de la espacial para los tamaños de línea evaluados. Ello se debe a que la frecuencia de saltos efectivos ($S\ T$) es muy pequeña por lo que gran parte de los SBBD no caben en una línea de cache. Si continuaramos aumentando el tamaño de línea llegaría un punto en que la pendiente de las curvas pasaría a ser positiva para finalmente acabar siendo positivo el valor de $\Delta Rend$.

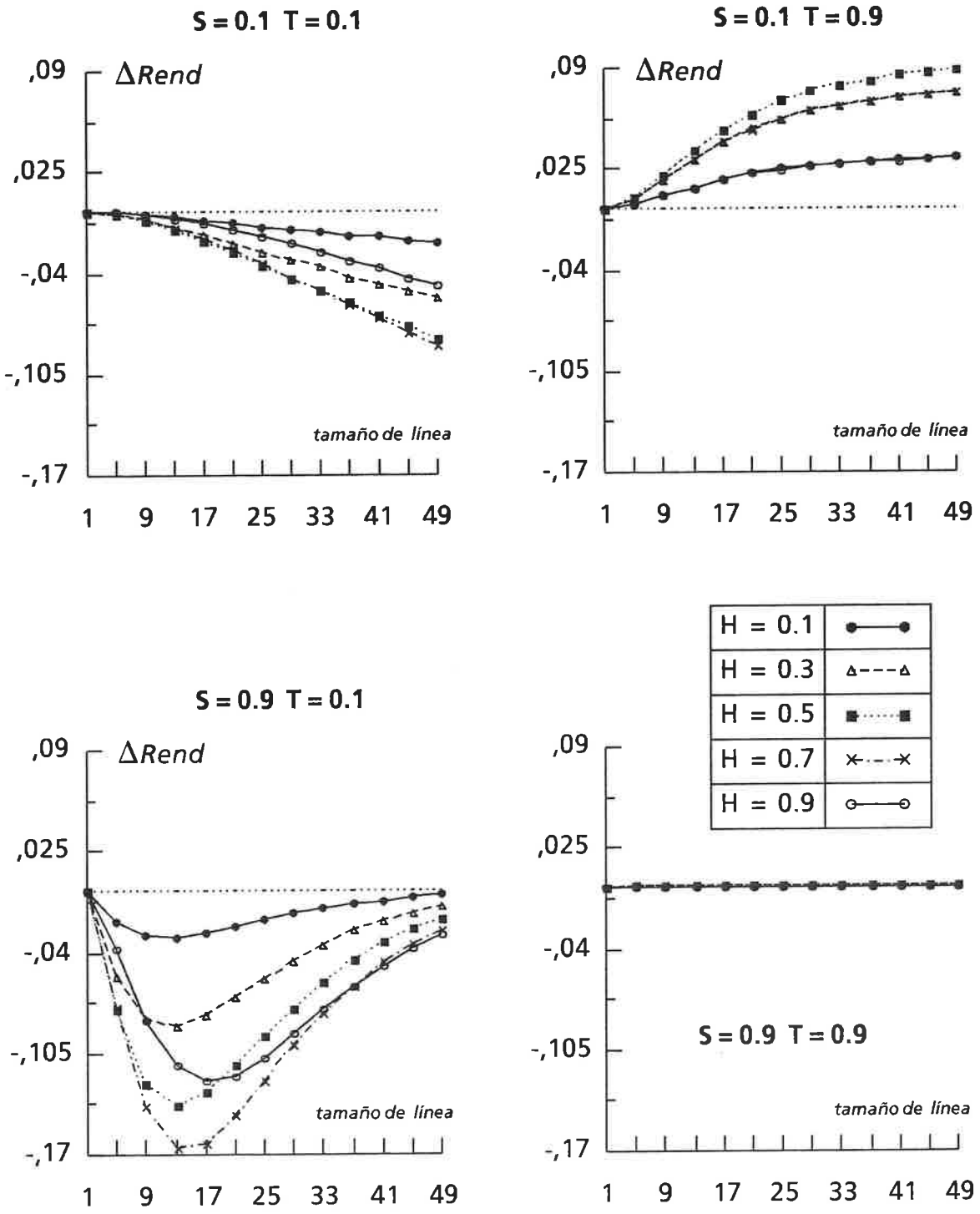


Figura 6.10: Diferencia entre el rendimiento del mecanismo ESCO con prebúsqueda temporal y espacial para diferentes valores de la tasa de aciertos a la MIDS (H), frecuencia de las instrucciones de salto (S), frecuencia con que un salto es efectivo (T) y tamaño de línea de la MIDS. Los resultados se dan en ciclos por instrucción útil ejecutada.

En la segunda ($S=0.1$ y $T=0.9$) y tercera ($S=0.9$ y $T=0.1$) la frecuencia de los saltos efectivos es idéntica en ambos casos. Sin embargo mientras en la segunda el rendimiento de la prebúsqueda temporal es siempre superior al de la espacial, no ocurre así en la tercera. El motivo se debe a que en el caso $S=0.9$, al ser muy frecuentes las intrucciones de salto, la pérdida ocasionada por un SBBD con tamaño mayor al de la línea de cache es grande debido a que aumenta la probabilidad de encontrar un salto no efectivo situado en las últimas posiciones de la línea que ocasione un fallo de cache. Por otra parte, la elevada frecuencia de saltos hace que el beneficio aportado sea mínimo ya que el tamaño de los BBD es pequeño y por lo tanto la anticipación conseguida es mínima (la anticipación respecto a la prebúsqueda espacial como mucho es igual al tamaño del último BBD menos dos unidades).

Finalmente, en la cuarta gráfica ($S=0.9$ y $T=0.9$) la diferencia de rendimiento entre la prebúsqueda espacial y la temporal es prácticamente nula para todos los valores de H y L. El motivo es que la elevada frecuencia de saltos efectivos (S T) hace que sea muy elevada la probabilidad de que en una línea de cache haya un salto efectivo. Por lo tanto el rendimiento de la prebúsqueda temporal es siempre superior al de la espacial. Sin embargo, al igual que ocurría en la gráfica tercera, la elevada frecuencia de instrucciones de salto ocasiona que la anticipación sea muy pequeña y por lo tanto el beneficio aportado es mínimo.

CAPITULO 7

Conclusiones y líneas abiertas

7.1. CONCLUSIONES

El presente trabajo se centra en el diseño y evaluación de mecanismos que permitan reducir los conflictos ocasionados por las instrucciones de salto en arquitecturas segmentadas.

Las aportaciones de esta tesis son de dos tipos. Por una lado se presenta un mecanismo (ESCO) con el objetivo de ejecutar las instrucciones de salto en paralelo con el resto de instrucciones, de forma que su coste de ejecución (en tiempo) sea nulo. El otro tipo de aportación es referente a la técnica utilizada para la obtención de medidas. Esta técnica basada en la modelización del comportamiento de la unidad de instrucciones nos ha permitido obtener medidas de rendimiento para diferentes alternativas, tanto en el diseño del mecanismo como en el diseño del sistema de memoria, utilizando programas de prueba de tamaño considerable con un coste muy reducido. Además, mediante estos modelos hemos podido evaluar la influencia de los diferentes parámetros del sistema en el rendimiento del procesador.

En el capítulo 2 se realiza una descripción funcional del mecanismo ESCO con dos organizaciones alternativas del sistema de memoria (apartados 2.1, 2.2 y 2.3). Estas organizaciones se caracterizan por tener un único puerto de acceso o dos puertos de acceso. Seguidamente se desarrolla un modelo analítico que caracteriza el comportamiento del mecanismo ESCO durante la ejecución de un BBD (bloque básico dinámico) para ambas organizaciones de memoria. Finalmente se modeliza el comportamiento del mecanismo ESCO durante la

ejecución completa de un programa mediante un proceso de Markov (apartado 2.4).

Los modelos desarrollados en el capítulo 2 permiten obtener medidas de rendimiento con un coste mínimo de tiempo de cálculo. Estos modelos son utilizados en el capítulo 3 para analizar diferentes alternativas en el diseño del mecanismo ESCO. Las conclusiones de este análisis son las siguientes.

1) El rendimiento obtenido con la memoria de un puerto es prácticamente igual al obtenido con la memoria de dos puertos. Dado el menor coste de una memoria de un puerto, nos inclinamos por utilizar este tipo de memoria.

2) La capacidad de almacenamiento que necesita la unidad de instrucciones para obtener prácticamente el máximo rendimiento del mecanismo ESCO es mínima (del orden de 4 instrucciones).

3) El permitir que una instrucción de salto pueda ser efectiva en el mismo ciclo en que es accedida no aporta un aumento significativo del número de instrucciones ejecutadas por ciclo.

4) Los saltos incondicionales pueden ser tratados por la unidad de instrucciones de la misma forma que los saltos condicionales.

5) Es interesante que las llamadas y retornos de subrutina sean gestionados por la unidad de ejecución y por lo tanto se ejecuten en paralelo. No obstante, el redimiento del mecanismo ESCO, aunque menor, sigue siendo bastante bueno cuando estas instrucciones son ejecutadas en la unidad de ejecución.

6) La eficiencia de ESCO es mayor si en la prebúsqueda por ambas ramas de un salto damos prioridad a la rama destino del salto. No se obtiene una ganancia significativa con técnicas de predicción más sofisticadas.

En el capítulo 5 se presenta una implementación del mecanismo ESCO haciendo uso de una memoria cache de instrucciones convencional (apartado 5.1). Se pone de manifiesto que el hardware adicional que necesita la unidad de instrucciones para implementar el mecanismo ESCO es bastante reducido.

Diversos parámetros del sistema son analizados para obtener el valor que optimiza el rendimiento del sistema. En concreto, los dos resultados obtenidos son los siguientes.

1) Los accesos a memoria externa que están en curso en el instante en que una instrucción de salto tiene efecto y que pertenecen a la rama abandonada por el salto serán cancelados si y sólo si todavía no ha llegado a la unidad de instrucciones ninguna instrucción correspondiente a ese acceso.

2) El tamaño óptimo de la línea de cache es de cuatro instrucciones.

También en el capítulo 5 se evalúa el rendimiento del mecanismo ESCO y de otros mecanismos de ejecución de saltos para dos diferentes protocolos de comunicación con la memoria externa (apartado 5.2).

Para un protocolo simple el rendimiento del mecanismo ESCO es entre un 25 y un 38% superior al de los mecanismos basados en el esquema de salto retardado y entre un 18 y 20% superior al del mecanismo denominado memoria de instrucciones destino de salto. El tráfico generado por el mecanismo ESCO entre la memoria cache y la memoria externa es bastante parecido al que generan los mecanismos basados en el salto retardado y algo mayor que el generado por la memoria de instrucciones destino de salto.

Con un protocolo modo ráfaga se consigue aumentar la eficiencia de todos los mecanismos. La diferencia de rendimiento y tráfico entre el mecanismo ESCO y el resto de mecanismos es del mismo orden que con el protocolo simple.

En el capítulo 6 se presenta una implementación del mecanismo ESCO utilizando como memoria cache una memoria de instrucciones destino de salto (MIDS). Inicialmente nos centramos en el uso de técnicas de prebúsqueda espacial para finalmente evaluar el comportamiento de ESCO con técnicas de prebúsqueda temporal.

Para evaluar el rendimiento del mecanismo se desarrolla un modelo analítico que caracteriza el comportamiento del mecanismo ESCO con esta organización de memoria cache (apartado 6.3). También se desarrolla un modelo analítico que

caracteriza el comportamiento del mecanismo salto retardado con este tipo de memoria (apartado 6.2).

El modelo analítico de ESCO se utiliza para obtener los parámetros de diseño de la unidad de instrucciones (apartado 6.4). El principal resultado de este análisis es que el tamaño óptimo de la línea de cache es igual a la latencia de memoria externa más una unidad.

El diseño que se presenta de la unidad de instrucciones pone de manifiesto que el hardware necesario para implementar el mecanismo ESCO con prebúsqueda espacial es todavía menor con esta organización de memoria cache que con una cache convencional (apartado 6.5).

El rendimiento del mecanismo ESCO con una MIDS es entre un 21 y un 28% superior al del salto retardado (apartado 6.6).

La diferencia de rendimiento del mecanismo ESCO con una MIDS y con una cache convencional depende de la tasa de aciertos (apartado 6.6). Si la tasa de aciertos no es muy elevada el rendimiento de la MIDS es considerablemente superior al de la cache convencional. Para tasas de aciertos muy elevadas el rendimiento de la cache convencional es algo superior.

El tráfico generado con memoria externa es mucho mayor con una MIDS que con una cache convencional.

Al evaluar el mecanismo ESCO con prebúsqueda temporal se obtiene que el rendimiento es prácticamente igual al obtenido con prebúsqueda espacial (apartado 6.7). Ya que la prebúsqueda temporal implica que el circuito de detección anticipada del salto debe ser más complejo, será preferible utilizar prebúsqueda espacial.

7.2. LÍNEAS ABIERTAS

A continuación se describen algunas líneas de investigación que han quedado abiertas con esta tesis.

La primera de ellas es un trabajo que se está llevando a cabo en el momento de escribir esta tesis. Se refiere a la realización VLSI de la unidad de instrucciones

que implementa el mecanismo ESCO. Para ello se están utilizando las herramientas de diseño desarrolladas en las universidades de Berkeley y Washington conocidas con el nombre de MAGIC. El diseño se lleva a cabo utilizando la tecnología de dos micras de ES2. Ello nos permitirá obtener medidas más exactas sobre el área ocupada por cada una de las partes de la unidad de instrucciones así como el tiempo de respuesta y la identificación de los cáminos críticos que nos fuerzan ese tiempo de respuesta. Con estos resultados se puede analizar más detalladamente el mecanismo ESCO y estudiar determinadas modificaciones que permitan mejorar su rendimiento.

Otro campo de investigación es a nivel de compilador. Los modelos desarrollados en esta tesis ponen de manifiesto que tipos de bloques básicos causan una disminución mayor del rendimiento de ESCO. Un trabajo interesante sería analizar mecanismos software basados en la reorganización del código de programa que permitan reducir estos casos y por lo tanto aumentar el rendimiento del mecanismo ESCO.

En el capítulo 6 se pone de manifiesto las ventajas en cuanto a rendimiento que proporciona el utilizar una memoria cache cuya unidad de mapeo sea el SBBD. Este tipo de memoria, que denominamos memoria de instrucciones destino de salto (MIDS), la implementamos mediante líneas de tamaño fijo. Las principales desventajas de esta implementación son: a) El tráfico generado con memoria externa es considerable debido a aquellos SBBD cuyo tamaño es superior al de la línea y b) Se desaprovecha la capacidad de la memoria cuando hay SBBD cuyo tamaño es menor que el de una línea.

Sería interesante estudiar otras implementaciones de una MIDS que permitieran reducir estos efectos negativos. Una posibilidad sería utilizar una memoria dividida en diferentes niveles (ver figura 7.1). Cuando se lee un SBBD de memoria externa sus primeras instrucciones se almacenan en el nivel 1. Si el tamaño de este SBBD supera al de la línea del nivel 1, se coge una línea del nivel 2 para almacenar la continuación de este SBBD y así sucesivamente con niveles sucesivos. Parámetros a determinar serían el número máximo de niveles y el tamaño de la línea de cada nivel.

Finalmente, podría estudiarse la aplicación del mecanismo ESCO a arquitecturas con instrucciones de lenguaje máquina de duración variable. En este caso, al haber instrucciones con duración mayor a un ciclo podría no ser

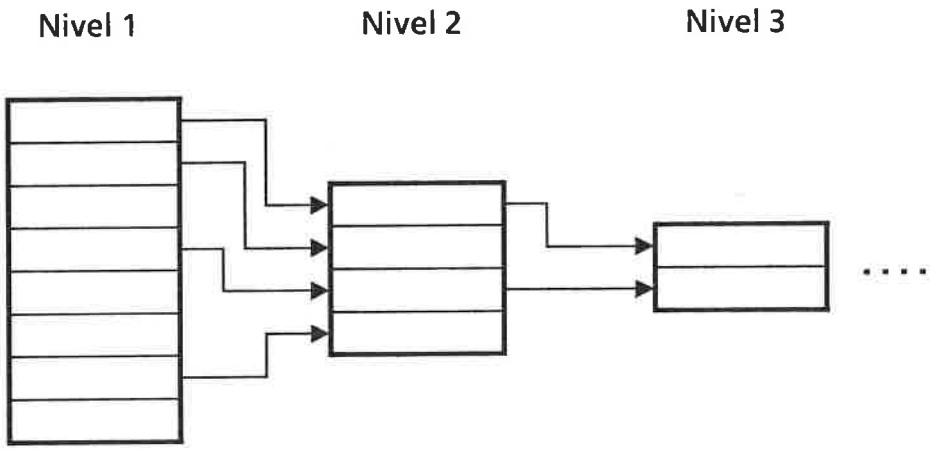


Figura 7.1: Implementación de una MIDS.

necesario el disponer de un ancho de banda mayor que una instrucción por ciclo para así poder detectar los saltos con cierta anticipación. Ello podría suponer una simplificación en la implementación de la unidad de instrucciones.

APENDICE A

Modelo analítico de ESCO con una MIDS

En este apéndice presentamos el desarrollo del modelo analítico que caracteriza el comportamiento del mecanismo ESCO con una MIDS. Seguidamente comentamos diversas relajaciones de este modelo que dan lugar a los modelos presentados en el capítulo 6.

A.1. DEFINICIONES PREVIAS

- Sea T la probabilidad de que el salto sea efectivo.
- Sea H la probabilidad de acierto en la MIDS.
- Sea $D(d)$ la función de densidad de probabilidad de la distancia entre dos saltos efectivos (longitud de los SBBD).
- Sea $F(d)$ la función de densidad de probabilidad de la distancia entre dos instrucciones de salto (longitud de los BBD).
- Sea R la latencia de la memoria externa.
- Sea $R+K$ el tamaño de línea de la MIDS ($K \geq 0$).
- Sea N la variable aleatoria que representa el número de saltos no efectivos que existen entre el anterior salto efectivo y salto analizado.

- Sea L la variable aleatoria que representa el número de instrucciones del SBBD al que pertenece el salto analizado.

A.2. MODELO

Tal como vimos en el capítulo 6, para calcular el coste de un salto será preciso analizar lo ocurrido desde el anterior salto efectivo hasta el salto analizado.

Para ello caracterizaremos las cuatro posibles causas que ocasionan que el coste del salto sea diferente a cero y evaluaremos su contribución al coste medio de un salto. Estas cuatro causas son las descritas en el capítulo 6, representadas gráficamente en la figura 6.2.

La contribución de cada una de estas causas es la presentada en aquel capítulo (apartado 6.3.1), a excepción de la causa *b*, donde se realizaron una serie de relajaciones. En este apartado vamos a calcular la contribución de esta causa sin realizar ningún tipo de relajación.

Recordemos que esta causa la etiquetamos como *"Pérdida de anticipación debida a los saltos no efectivos"*. Ello ocurre cuando el acceso a la MIDS en el anterior salto efectivo produce un acierto, entre este salto efectivo y el salto actual hay K o más saltos no efectivos y además el salto analizado pertenece a un SBBD cuyo tamaño es mayor que $R+K$.

La contribución de esta causa al coste medio de un salto es igual a

$$H\ Prob\,(N \geq K\ \cap\ L > R+K)$$

Aplicando la siguiente igualdad conocida por ley de multiplicación

$$Prob\,(A \cap B) = Prob\,(A\,|\,B)\ Prob\,(B)\ =\ Prob\,(B\,|\,A)\,Prob\,(A)$$

tenemos que la contribución de la causa *b* es igual a

$$H\ Prob\,(L > R+K\ |\ N \geq K)\ Prob\,(N \geq K)$$

A.2.1. Cálculo de Prob (N ≥ K)

Si hacemos la hipótesis de que la probabilidad de que un salto sea efectivo es independiente de lo ocurrido en los anteriores saltos, la variable aleatoria N seguirá una ley geométrica, y por lo tanto

$$Prob\,(N \geq K) = \sum_{j=K}^{\infty} T\,(1-T)^{j} = (1-T)^{K}$$

A.2.2. Cálculo de Prob (L > R+K | N ≥ K)

Para calcular esta probabilidad previamente calcularemos $Prob\ (\ L\ >\ R+K\ )$. Para ello definimos N_I como el número medio de instrucciones de salto de un SBBD que tenga I instrucciones. Tenemos que

$$Prob\,(L = I) = \frac{N_I}{\sum_{j=1}^{\infty} N_j}$$

$$Prob\,(L > I) = 1 - \frac{1}{\sum_{j=1}^{\infty} N_j} \sum_{j=1}^{I} N_j$$

N_I puede calcularse mediante la siguiente expresión

$$N_I = \sum_{j=1}^{I} j\ B_j\ C_j(I)$$

donde B_j es la probabilidad de que un SBBD esté formado por j BBD y $C_j\ (\ I\ )$ representa la probabilidad de que la longitud total de j BBD consecutivos sea igual a I.

Con las hipótesis realizadas, B_j vendrá dado por la función de densidad de probabilidad de una ley geométrica, es decir

$$B_j = T\,(1-T)^{j-1}$$

$C_j(I)$ es función de $F(d)$ y puede calcularse mediante las siguientes expresiones.

$$C_1(I) = F(I)$$

$$C_j(I) = \sum_{k=j-1}^{I-1} F(I-k)\,C_{j-1}(k) \qquad si\ j>1$$

El cálculo de $Prob\,(L > R+K \mid N \geq K)$ es semejante al de $Prob\,(L > R+K)$ con la salvedad de que en este caso sólo debemos considerar aquellos SBBD con más de K saltos y en el cálculo de la probabilidad no debemos tener en cuenta la contribución de estos K primeros saltos. Por lo tanto, tenemos que

$$Prob\,(L > R+K \mid N \geq K) = 1 - \frac{1}{\sum_{k=1}^{\infty} M_K(k)} \sum_{k=1}^{R+K} M_K(k)$$

donde $M_K(k)$ representa el número medio de saltos que quedan en un SBBD con k instrucciones (sin contabilizar los K primeros) dado que como mínimo tiene $K+1$ instrucciones de salto (BBD). Su valor es igual a

$$M_K(k) = \sum_{i=K+1}^{k} (i-K)\ B_i\ C_i(k)$$

siendo B_i y $C_i\ (k)$ las funciones anteriormente definidas.

A.3. POSIBLES SIMPLIFICACIONES

Las simplificaciones del modelo desarrollado en este apéndice que han dado lugar a los modelos presentados en el capítulo 6 son las siguientes.

1) Las variables aleatorias L y N son estadísticamente independientes. Ello implica que

$$Prob\,(N \geq K \cap L > R+K) = Prob\,(N \geq K)\; Prob\,(L > R+K)$$

con esta simplificación el cálculo de $Prob\,(N \geq K)$ y $Prob\,(L > R+K)$ se realizaría mediante las expresiones desarrolladas en el apartado anterior. El cálculo del segundo término sigue siendo bastante complejo. La siguiente relajación nos simplifica este cálculo.

2) Cálculo de $Prob\,(L > R+K)$.

2.1) El número de instrucciones de salto de un SBBD es independiente de su longitud. En este caso

$$Prob\,(L > R+K) = 1 - \sum_{d=1}^{R+K} D(d)$$

2.2) El número de instrucciones de salto de un SBBD es directamente proporcional a su longitud. En este caso

$$Prob\,(L > R+K) = 1 - \sum_{d=1}^{R+K} \frac{D(d)\,d}{\sum_{i=1}^{\infty} D(i)\,i}$$

Referencias

[AdMD87] Advanced MicroDevices
Am29000 Streamlined Instruction Processor
Publicación no. 09075, Febrero 1987, pp. 49-54.

[ACHA87] A. Agarwal, P. Chow, M. Horowitz, J. Acken, A. Salz, J. Hennessy
On-Chip Instruction Caches for High Performance Processors
en Advanced Research in VLSI (Proceedings of the 1987 Stanford Conference)
P. Losleben, editor
The MIT Press, 1987, pp. 1-24

[AnST87] D.W. Anderson, F.J. Sparacio and R.M. Tomasulo.
The IBM System/360 Model 91: Machine Philosophy and Instruction Handling
IBM Journal of Research and Development, vol. 11, no. 1, Enero 1967, pp. 8-24

[BaWa87] J.L. Baer, W.H. Wang
Architectural Choices for Multi-Level Cache Hierarchies
Proceedings of the 16th. International Conference on Parallel Processing, Agosto 1987, pp. 258-261

[Case87] B. Case
Pipelined Processor Pushes Performance
ESD: The Electronic System Design Magazine, Marzo 1987

[CGLl88] J. Cortadella, A. González, J.M. LLabería
RISC: Un nuevo enfoque en el diseño de procesadores
Mundo Electrónico, no. 180, Enero 1988, pp. 49-57

[ChHo87] P. Chow, M. Horowitz
Architectural Tradeoffs in the Design of MIPS-X
Proceedings of the 14th. Annual International Symposium on Computer Architecture, Junio 1987, pp. 300-308.

[Chow86] P. Chow
MIPS-X Instruction Set and Programmers Manual
Technical Report no. 86-289, Computer Systems Laboratory, Departament of Electrical Engineering and Computer Science, Stanford University, Mayo 1986

[CLlG87] J. Cortadella, J.M. Llabería, A. González
Keeping Control Transfer Instructions out of the Pipeline in Architectures without Condition Codes
Informe de investigación no. RR 87/11, Facultad de Informática. Universidad Politécnica de Cataluña, 1987

[CoJo88] J. Cortadella, T. Jove
Designing a Branch Target Buffer for Executing Branches with Zero Time Cost in a RISC Processor
Microprocessing and Microprogramming, vol. 24, no. 1-5, Agosto 1988

[CoLl87a] J.Cortadella, J.M.Llaberia
A Low Cost Evaluation Methodology for New Architectures
Proceedings of the IASTED International Symposium on Applied Informatics, Febrero 1987, pp. 192-195.

[CoLl87b] J. Cortadella, J.M. Llabería
An Instruction Unit for Eliminating Branch Execution in Pipelined Processors
Informe de investigación no. RR 87/10, Facultad de Informática. Universidad Politécnica de Cataluña, 1987

[DiMB87] D.R. Ditzel, H.R. McLellan, A.D. Berenbaum
The Hardware Architecture of the CRISP Microprocessor
Proceedings of the 14th. Annual International Symposium on Computer Architecture, Junio 1987, pp. 309-319

[DeLe87] A. De Rosa and H.M. Levy
An Evaluation of Branch Architectures
Proceedings of the 14th. Annual International Symposium on Computer Architecture, Junio 1987, pp. 10-16

[DiMc82] D.R. Ditzel, H.R. McLellan, A.D. Berenbaum
Register Allocation for Free: The C Machine Stack Cache
Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, Marzo 1982, pp. 48-56

[DiMc87] D.R. Ditzel and H.R. McLellan
Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero
Proceedings of the 14th. Annual International Symposium on Computer Architecture, Junio 1987, pp. 2-9

[GiMi89] C.E. Gimarc, V.M. Milutinovic
RISC Principles, Architecture and Design
en High-Level Language Computer Architecture
V.M. Milutinovic, editor
Computer Science Press, 1989, pp. 1-64

[GLlC88a] A. González, J.M. Llabería, J. Cortadella
Zero-Delay Cost Branches in RISC Architectures
Proceedings of the IASTED International Symposium on Applied Informatics, Febrero 1988, pp. 24-27

[GLlC88b] A. González, J.M. Llabería, J. Cortadella
A Mechanism for Reducing the Cost of Branches in RISC Architectures
Microprocessing and Microprogramming, vol. 24, no. 1-5, 1988, pp. 565-572.

[GoLl89] A. González, J.M. Llabería
Instruction Fetch Unit for Parallel Execution of Branch Instructions
1989 International Conference on Supercomputing, ACM SIGARCH ICS-89, Junio 1989.

[Good85] J.R. Goodman et al.
PIPE: A VLSI Decoupled Architecture
Proceedings of the 12th. Annual International Symposium on Computer Architecture, Junio 1985, pp. 20-27

[GrHe82] T.R. Gross, J.L. Hennessy
Optimizing Delayed Branches
Proceedings of the 15th. Annual Workshop on Microprogramming, ACM SIGMICRO, Octubre 1982, pp. 114-120

[Henn84] J.L. Hennessy
VLSI Processor Architecture
IEEE Transactions on Computers, vol. c-33, no. 12, Diciembre 1984, pp. 1221-1246

[Hill86] M. Hill et al.
Design Decisions in SPUR
Computer Magazine, vol. 19, no. 10, Noviembre 1986, pp. 8-22

[Hill87] M. Hill
Aspects of Cache Memory and Instruction Buffer Performance
Report no. UCB/CSD 87/381, Computer Science Division, University of California, Berkeley (Tesis Doctoral), Noviembre 1987

[HJBG82] J.Hennessy, N.Jouppi, F. Baskett, T. Gross and J. Gill
Hardware/Software Tradeoffs for Increased Performance
Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, Marzo 1982, pp. 2-11.

[Hopk85] M.E. Hopkins
A Definition of RISC
Proceedings of the International Workshop on High-Level Computer Architecture, Mayo 1984, pp. 3.8-3.11

[Horo87] M. Horowitz et al.
A 32-Bit Microprocessor with 2K-Byte On-Chip Cache
Proceedings of the IEEE International Solid-State Circuits Conference, 1987

[Ibbe82] R.N. Ibbett
Intruction Buffering
en The Architecture of High Performance Computers, capítulo 5, Mcmillan Press, 1982, pp. 72-94

[IBMC86] International Business Machines Corporation
IBM RT Personal Computer Technology
Publicación no. SA23-1057, 1986

[InCo88] Intel Corporation
80960KB Programmer's Reference Manual
Publicación no. 270567-001, 1988

[John87] M. Johnson
System Considerations in the Design of the Am29000
IEEE Micro, Agosto 1987, pp. 29-41

[Kate85] M.G.H. Katevenis
Reduced Instruction Set Computer Architecture for VLSI
The MIT Press, Cambridge, Massachussets, 1985

[Katz85] R.H. Katz, editor
Proceedings of CS292i: Implementation of VLSI Systems
Technical Report UCB/CSD 86/259, Computer Science Division, University of California, Berkeley, Septiembre 1985

[Kell75] R.M. Keller
Look-Ahead Processors
Computing Surveys, vol. 7, no. 4, Diciembre 1975, pp. 177-195

[Kogg81] P.M. Kogge
The Architecture of Pipelined Computers
Mc Graw Hill, 1981

[LeSm84] J.K. F. Lee, A.J. Smith
Branch Prediction Strategies and Branch Target Buffer Design
Computer, no. 1, Enero 1984, pp. 6-22.

[Lilj88] D.L. Lilja
Reducing the Branch Penalty in Pipelined Processors
Computer, vol. 21, no. 7, Julio 1988, pp. 47-55

[MBMH86] M.J. Mahon, R. Bel-Loh Lee, T.C. Miller, J.C. Huck, W.R. Bryg
Hewlett-Packard Precision Architecture: the Processor
Hewlett-Packard Journal, vol. 37, no. 8, Agosto 1986, pp. 4-22

[McHe86] S. McFarling and J. Hennessy
Reducing the Cost of Branches
Proceedings of the 13th. Annual International Symposium on Computer Architecture, 1986, pp. 396-403.

[MoIb79] D. Morris, R.N. Ibbett.
The MU-5 Computer System
Springer-Verlag, 1979

[MoIn88] Motorola Inc.
32-Bit Third-Generation RISC Microprocessor
Publicación no. BR588/D, 1988

[OlVR88] A. Olivé, V. Viñals and C. Rodríguez
Identifying Influencing Factors on Branch Target Cache Memory Performance
Proceedings of the ISMM International Symposium: Mini and Microcomputers and their Application, Spain, 1988, pp. 364-367

[Patt85] D.A. Patterson
Reduced Instruction Set Computers
Communications of ACM, vol. 28, no. 1, Enero 1985, pp. 8-21

[PGHJ84] S.A. Przybylski, T.R. Gross, J.L. Hennessy, N. Jouppi, C. Rowen
Organization and VLSI implementation of MIPS
Technical Report 84-259, Computer Systems Laboratory, Departament of Electrical Engineering and Computer Science, Stanford University, Abril 1984.

[PGHJ87] A.R. Pleszkun, J.R. Goodman, W.C. Hsu, R.T. Joersz, G. Bier, P. Woest, P.B. Schechter.
WISQ: A Restartable Architecture Using Queues
Proceedings of the 14th. Annual International Symposium on Computer Architecture, Junio 1987, pp. 290-299

[PlFa86] A.R. Pleszkun, M.K. Farrens
An Instruction Cache Design for use with a Delayed Branch
en Advanced Research in VLSI (Proceedings of the Fourth MIT Conference)
C.E. Leirserson, editor
The MIT Press, 1986, pp. 73-88

[PrHH88] S. Przybylski, M. Horowitz, J. Hennessy
Performance Tradeoffs in Cache Design
Proceedings of the 15th. Annual International Symposium on Computer Architecture, Junio 1988, pp. 290-298

[RiFo72] E.M. Riseman, C.C. Foster
The Inhibition of Potential Parallelism by Conditional Jumps
IEEE Transactions on Computers, vol. 21, no. 10, Diciembre 1972, pp. 1405-1441

[Ryan88] D.P. Ryan
Intel's 80960: An Architecture Optimized for Embedded Control
IEEE Micro, vol. 8, no. 3, Junio 1988, pp. 63-76

[Smit82] A.J. Smith
Cache Memories
ACM Computing Surveys, vol. 14, no. 3, Septiembre 1982, pp. 473-530

[Smit86] A.J. Smith
Bibliography and Readings on CPU Cache Memories and Related Topics
Computer Architecture News, vol 14. no. 1, Enero 1986, pp. 22-42

[Smit87] A.J. Smith
Line (Block) Size Choice for CPU Cache Memories
IEEE Transactions on Computers, c-36, no. 9, Septiembre 1987, pp. 1063-1075

[ViRO86] V. Viñals, C. Rodríguez, A. Olivé
Memorias Cache
Mundo Electrónico, no. 167, 1986, pp. 72-82

Aquesta TESI fou [illegible] la a la Facultat d'Informàtica de la Universitat Politècnica de Barcelona el dia 12 de Maig del 1989 i mereixqué la següent qualificació: APTE "CUM LAUDE" per unanimitat

La Presidència,

Signat: M. VALERO

El Secretariat del Tribunal,

Signat: Jordi Cortadella

Vocal.

Signat: F. TIRADO

Vocal.

Signat: Jose Aviz

Vocal.

Signat: Pedro de Miguel

El/La Doctorand/a,

A. Gonzalez