Aumentando las Prestaciones en la Predicción de Flujo de Instrucciones

Departamento de Arquitectura de
Computadores y Sistemas Operativos

Tesis presentada por Juan Carlos Moure López para optar al grado de *Doctor por la Universitat Autònoma de Barcelona*.

Barcelona, Abril 2006

Aumentando las Prestaciones en la Predicción de Flujo de Instrucciones

Tesis presentada por Juan Carlos Moure López para optar al grado de *Doctor por la Universitat Autònoma de Barcelona*. El trabajo ha sido realizado en el Departamento de Arquitectura de Computadores y Sistemas Operativos de la Universidad Autónoma de Barcelona.

Bellaterra, Abril de 2006

Dirección de la Tesis:

Dr. Emilio Luque Fadón

A Milagros e Isauro, mis padres, les debo todo

A Silvia, mi compañera de viaje, este trabajo es también suyo

A David y Laura, mis hijos, a quienes darles todo

AGRADECIMIENTOS

Muchas personas han hecho posible este trabajo. Todas me han ayudado en muchos aspectos. Como es imposible que sea justo con ellas en tan poco espacio, sin poder matizar lo suficiente, tampoco lo intentaré, y a cambio seleccionaré para cada una de ellas el primer pensamiento que llegue a mi mente.

Primero de todo, al Dr. Emilio Luque, por la libertad y confianza que me ha dado. En muchas ocasiones él ha confiado en mí más que yo mismo.

A Dolores Rexachs por su constante preocupación y cuidado en que lograra acabar esta tesis.

A Domingo, que ha corrido junto a mí para saltar muchos de los obstáculos que han supuesto este trabajo.

Con Dani empecé, y de nuevo el destino ha vuelto a establecer un enlace de interconexión.

El trabajo no sería lo mismo sin compartir las penas y alegrías de la investigación durante los cafés de media mañana, y a veces incluso de alguna tarde. Y éstos no serían nada sin discusiones sobre temas transcendentales como fútbol, cine, política, ... ¿Qué hace uno solo en un bar? Gracias a Eduardo, Fernando, Josep, Carlos, Xiao, y a todos los que alguna vez hemos intercambiado tiquets-comedor.

Estoy también en deuda con todos los compañeros del Departamento de Arquitectura de Computadores y Sistemas Operativos. Como no quiero dejarme a ninguno haré una visualización mental de los despachos: Tomàs, Anna, Miquel, Elisa, Ana, Joan, Porfi, Remo. Y por supuesto todos los sufridores doctorandos que tenemos y hemos tenido: entiendo vuestras fatigas, y espero no olvidarlas. Y a los responsables de Condor y del cluster del departamento, Dani, Jordi y Enol. Y un recuerdo especial a Tomás Díez.

Pero es a mi familia a quien más le debo. Mis padres hicieron un gran esfuerzo y tenían mucha ilusión para que yo pudiera llegar hasta aquí. Todo lo que se construye necesita apoyarse en algo previo: ellos fueron ese primer apoyo. Mi hermana también me ha ayudado a recorrer este camino, con su alegría y cuidado. Esta tesis también es para vosotros.

Por fin podré entregar a mis hijos lo que llevan tiempo preguntándose ¿qué será? Cuando la vean, seguro dirán: “Ah! ¿Esto era la tesis? Pues si sólo son letras, y además hay pocos dibujos. Podríamos haberte ayudado y habríamos acabado antes”. Seguramente tendrán razón.

Silvia, mi mujer, es sin duda la que más ha sufrido. Los continuos retrasos, mi mal humor cada vez más frecuente, mis ausencias, físicas pero sobre todo mentales. Espero que el final de este viaje, casi siempre solitario, sea el principio de muchos otros viajes, pero ahora en compañía.

Por último, al considerar todo el tiempo que se ha quedado por el camino, uno no puede dejar de hacer una reflexión sobre la importancia del tiempo y de la planificación de éste. Sea poco o mucho, es de cada uno.

“Solo tú puedes decidir qué hacer con el tiempo que te ha sido dado”

J.R.R. Tolkien. “El Señor de los Anillos”

Y como corolario final, lo que realmente importa (que cada uno substituya las letras a su manera):

$$\text{Felicidad} = \frac{E\,(M + B + P)}{R + C}$$

Eduard Punset. “El Viaje a la Felicidad”

ÍNDICE

1. Introducción

Resumen

Este capítulo presenta la motivación al problema que se plantea resolver con esta tesis. Sirve para indicar los objetivos básicos, para describir de forma resumida las ideas utilizadas, y para guiar la lectura del resto de la memoria.

1.1. Introducción

Los microprocesadores actuales obtienen altas prestaciones mediante la ejecución de múltiples instrucciones en paralelo. El número de instrucciones que se pueden ejecutar de forma simultánea viene limitado por dependencias de datos y de control. El efecto negativo de las dependencias de control se puede reducir mediante la predicción de las instrucciones de control y la ejecución especulativa de instrucciones. Esta tarea supone identificar las instrucciones de control y predecir para cada una de ellas qué instrucción será la que se ejecutará a continuación. Esta tarea recae sobre una unidad que denominamos *predictor de flujo de instrucciones*, y a la cual nos referiremos a lo largo de esta memoria, como *predictor de flujo* o, simplemente, *predictor*.

1.1.1 El problema de la Predicción del Flujo de Control

Los procesadores consiguen altas prestaciones aumentando su frecuencia de reloj y su anchura de ejecución. Estas características requieren predictores a la vez muy precisos y muy rápidos. Estos dos requisitos son bastante incompatibles, ya que para obtener mayor precisión se ha de almacenar mayor cantidad de información, y entonces el acceso a esta información se hace más lento, y se alarga durante varios ciclos de reloj del procesador.

A esta incompatibilidad de objetivos se une la dificultad provocada por unos diseños del procesador cada vez más agresivos. Las mejoras en la tecnología logran una reducción en el tamaño de los transistores que permite reducir su retardo de funcionamiento, pero provoca que el retardo de propagación de las señales sea cada vez relativamente mayor [AHK+00]. De este modo, las memorias, que contienen una muy alta cantidad de conexionado, ven exacerbada la dicotomía entre capacidad y tiempo de

acceso. Si además, con objeto de aumentar la frecuencia del procesador, se aumenta la profundidad de segmentación, se está contribuyendo a que la cantidad de información accesible en un ciclo de reloj sea cada vez menor. A mayor número de etapas y mayor anchura del procesador mayor será la penalización relativa de los fallos de predicción, y la reducción de rendimiento en el procesador debida a la falta de precisión del predictor será mayor.

Por otro lado, estas mejoras tecnológicas proporcionan cada vez más cantidad de transistores, que se pueden utilizar para diseñar predictores con mejores prestaciones. Este diseño supone compromisos que logren un equilibrio entre la precisión en las predicciones, la velocidad de predicción, y la dedicación que se hace del área y energía disponible para alcanzar esas prestaciones. A continuación describiremos la organización y el modelo de rendimiento del predictor, para posteriormente enunciar y desgranar los objetivos de esta tesis.

1.1.2 Rendimiento del Predictor de Flujo de Control

La siguiente figura muestra un esquema muy básico de un predictor de flujo de control, que se utilizará para definir los parámetros que determinan sus prestaciones.

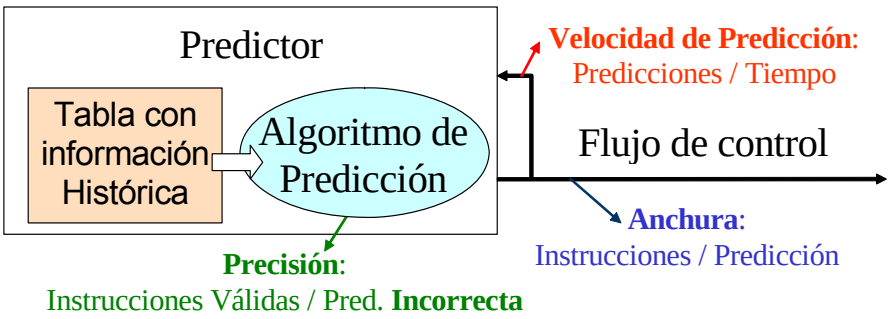


Figura 1.1. Modelo de rendimiento del predictor de flujo de control.

La *precisión* del predictor, que se puede medir como el número medio de instrucciones correctamente ejecutadas entre cada ocurrencia de un fallo de predicción, afecta enormemente al rendimiento del procesador. Un predictor más preciso no sólo aumenta el desempeño del procesador, sino que reduce la cantidad de trabajo malgastado ejecutando instrucciones en el camino erróneo, reduciendo el consumo de energía y la potencia disipada.

El *ancho de banda* del predictor, que se puede medir como el número de instrucciones válidas que proporciona el predictor por unidad de tiempo, es su otro parámetro fundamental de rendimiento. Se pueden identificar dos componentes que determinan al ancho de banda: la anchura de la predicción y la velocidad de predicción. La *anchura* del predictor es el número medio de instrucciones proporcionado en cada predicción. La *velocidad* del predictor se mide en predicciones por unidad de tiempo. El ancho de banda se calcula multiplicando la velocidad por la anchura, y descontando las instrucciones generadas de forma incorrecta debido a fallos de predicción. Si la precisión fuese perfecta se cumpliría la siguiente fórmula:

$$\textbf{\textit{Ancho de Banda}} = \textbf{\textit{Anchura}} \cdot \textbf{\textit{Velocidad}} \qquad (1)$$

Como cada cierto número de predicciones se producen errores, el ancho de banda efectivo se ve reducido respecto al ancho de banda potencial. La reducción se puede determinar a partir de la precisión del predictor y del tiempo medio transcurrido desde que se produce la predicción errónea hasta que el procesador detecta el error y lo notifica al predictor, es decir, a partir de la *penalización* media de los errores o fallos de predicción.

Proporcionar al procesador un ancho de banda de instrucciones suficiente es un requisito previo imprescindible para utilizar el paralelismo a nivel de instrucción con el objetivo de ejecutar un único programa más rápidamente. Esto implica que el predictor necesita proporcionar, al menos, el ancho de banda que el núcleo de ejecución de instrucciones sea capaz de consumir de forma sostenida. Sin embargo, hay tres factores que hacen que esto no sea suficiente: la latencia en el acceso a las instrucciones (fallos en la caché de

instrucciones), la ocurrencia de saltos en el flujo secuencial de las instrucciones, y la ocurrencia de fallos de predicción.

Los dos primeros factores ralentizan la operación de leer las instrucciones de memoria. Disponer con una cierta anticipación de la predicción de las instrucciones que se deben traer de memoria, permite no sólo realizar la búsqueda de forma más rápida, sino también de forma más eficiente.

Los fallos de predicción interrumpen el funcionamiento sostenido del procesador y generan una situación en la que disponer de un gran ancho de banda de predicción es beneficioso. Esta situación se puede apreciar en la siguiente figura, que muestra el porcentaje de ocupación de la ventana de instrucciones del núcleo de ejecución de un procesador durante el periodo de tiempo anterior y posterior a la ocurrencia de un fallo de predicción.

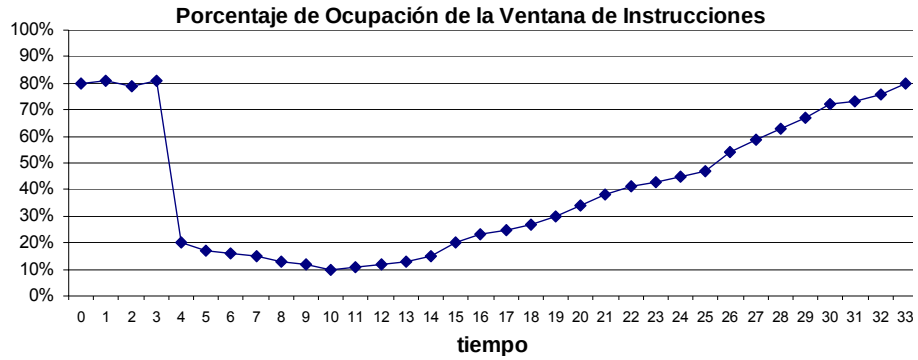


Figura 1.2. Efecto de un fallo de predicción en la ocupación de la ventana de instrucciones del núcleo de ejecución. La gráfica se ha obtenido de una ejecución real.

Al final del ciclo 3 se detecta un fallo de predicción que provoca que en el ciclo 4 muchas instrucciones desaparezcan de la ventana de instrucciones. En ciertos casos, la ventana podría llegar a vaciarse completamente. A partir de entonces, el núcleo va consumiendo las pocas instrucciones que le quedan. Es necesario que el predictor y la unidad de búsqueda reemprendan el camino correcto y que proporcionen “suficientes” instrucciones al núcleo de ejecución para volver al estado previo al fallo, para continuar como si el fallo

no se hubiera producido. El valor cuantitativo que corresponde a “suficientes” dependerá del porcentaje de la ventana que se vacía al detectar el fallo y del grado de paralelismo del procesador y del programa en ejecución.

En el periodo que sigue a la detección del fallo de predicción hay dos factores críticos para el rendimiento. En primer lugar, el tiempo mínimo de penalización por fallo queda determinado por el número de etapas entre el predictor y el núcleo de ejecución. En la Figura 1.2 no empiezan a llegar nuevas instrucciones hasta el ciclo 11, y por tanto se penalizan 7 ciclos.

Pero a partir del momento en que se entrega la primera instrucción al núcleo de ejecución, el parámetro crítico es el ancho de banda del predictor y el ancho de banda de búsqueda y decodificación de instrucciones. Es sobre todo en este periodo cuando es crítico que el predictor genere referencias a instrucciones a un ritmo superior al ritmo en que las consume el núcleo de ejecución. El ancho de banda del predictor limita la pendiente con la que se incrementa la ocupación de la ventana hasta alcanzar el valor “sostenido” que se tendría de no haberse producido el fallo.

El efecto en el rendimiento de disponer de menos instrucciones en la ventana de ejecución es difícil de prever, aunque ciertos trabajos indican que el rendimiento es aproximadamente proporcional al logaritmo del número medio de instrucciones en la ventana. En nuestra experimentación, basada en la simulación, se han obtenido unos resultados que se representan de forma cualitativa en la siguiente figura.

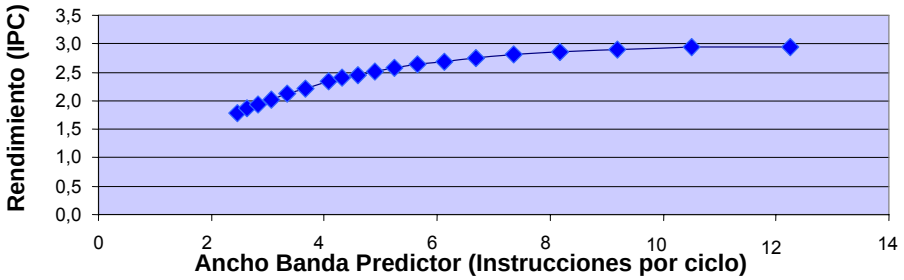


Figura 1.3. Efecto del ancho de banda del predictor en el rendimiento del procesador.

Aumentar la precisión del predictor es muy importante, ya que reduce el número de casos en que se vacía parte de la ventana de instrucciones. La ley de Amdahl advierte, no obstante, de que para un cierto nivel de precisión, el ancho de banda se convierte también en un cuello de botella importante, tal y como se puede apreciar en la figura anterior. Incrementar la precisión a cualquier coste no es eficiente, pues se necesita dedicar más memoria para almacenar la historia de los saltos, y utilizar algoritmos de predicción más sofisticados, lo cual supone una mayor complejidad que se traduce en una disminución de la velocidad del predictor y, por tanto, de su ancho de banda.

Disponer de un ancho de banda de predicción mayor del que la unidad de búsqueda de instrucciones es capaz de consumir es también beneficioso. Al disponer de la predicción con suficiente antelación, se puede organizar el acceso a las instrucciones de forma más eficiente, tanto para reducir la latencia media de acceso a las instrucciones, iniciando antes los accesos a las memorias más lentas, como para reducir el consumo energético, evitando los accesos simultáneos a múltiples bancos de memoria.

1.1.3 Organización Básica del Predictor de Flujo de Control

Aunque en los capítulos siguientes se presentará con todo detalle, es necesario apuntar aquí dos características principales de la organización del predictor que se va a adoptar como referencia en esta tesis: (1) el predictor está desacoplado de la búsqueda de instrucciones, y (2) se utiliza el bloque básico (*basic block, BB*) de instrucciones como unidad de predicción. Bajo estos supuestos, el predictor debe realizar tres tipos de tareas: (1) identificar qué instrucciones son saltos y diferenciar el tipo de salto, (2) proporcionar las direcciones de memoria a donde saltan, y (3) predecir, a partir de la historia de los saltos previos, el comportamiento futuro de los saltos.

Mientras que la predicción del sentido de los saltos condicionales se puede acelerar de forma relativamente simple con la técnica de predicción anticipada, que se describirá más adelante, el manejo coordinado de todos

los casos de predicción, y la generación de la dirección de las siguientes instrucciones, es una tarea compleja, y requiere de mecanismos que serán objeto del análisis de este trabajo.

Permitir que la predicción se realice de forma desacoplada a la búsqueda de instrucciones otorga flexibilidad para organizar el predictor de forma más eficiente. Permite también aprovechar de forma más efectiva un mayor ancho de banda en el predictor, por ejemplo, para implementar una mejor política de prebúsqueda de instrucciones y de datos. El paralelismo potencial disponible gracias a la arquitectura desacoplada y a un predictor con un elevado ancho de banda también se puede dedicar a reducir el consumo energético.

1.1.4 Objetivo de la Tesis

Esta tesis propone y analiza diferentes organizaciones y diferentes técnicas para aumentar el rendimiento del predictor de flujo de control, orientadas a su uso en un procesador de altas prestaciones. El trabajo considera el predictor en su totalidad, y la coordinación de toda la información para generar la predicción final como el elemento más problemático. Las propuestas están orientadas a aumentar la eficiencia en el uso de la memoria del predictor, y persiguen aumentar el ancho de banda del predictor, tanto aumentando la velocidad como aumentando la anchura de las predicciones. Todo ello se debe alcanzar también con un moderado consumo energético.

Se trata de organizar y codificar la información de forma eficiente, y de dotar al predictor de flexibilidad para adaptarse a los diferentes casos de forma eficaz. Así, las predicciones más frecuentes y sencillas se realizan rápidamente y utilizando recursos mínimos, mientras que predicciones menos comunes o que requieren el uso de más información para alcanzar una alta precisión, pueden realizarse más lentamente. La flexibilidad del diseño combinada con el exceso de ancho de banda permite compensar estos casos complejos y lentos, con el resto de casos, frecuentes y rápidos.

Aunque en esta tesis no se hace ninguna propuesta dirigida explícitamente a mejorar los algoritmos de predicción para aumentar su precisión, una organización más eficiente del predictor tiene como resultado lateral disponer de más recursos para aumentar la precisión. Por un lado, se utiliza mejor la memoria para almacenar la historia pasada. Por otro lado, se proporciona mayor tolerancia al aumento de la latencia en las predicciones, lo que permite aplicar algoritmos más complejos para aumentar la precisión.

El objetivo básico de esta tesis es, por tanto, predecir el flujo de instrucciones con una baja proporción de errores y alta rapidez. En particular, se hace incidencia en tres cuestiones fundamentales:

- *Aumentar la velocidad con que se predice el flujo de instrucciones, sin disminuir la precisión en la predicción y con un aumento moderado de la memoria del predictor.*
- *Aumentar la anchura de cada predicción, sin disminuir la precisión en la predicción y con un aumento moderado de la memoria del predictor.*
- *Mejorar la gestión de los saltos indirectos para, fijado el tamaño de memoria disponible, aumentar la precisión.*

A continuación se discute cada uno de estos puntos por separado.

1.2. Predicción más Rápida

Como ya se ha comentado, la búsqueda de mayor precisión y el incremento agresivo de la frecuencia de reloj se combinan para dificultar que el predictor proporcione una predicción por ciclo. Aumentar la precisión requiere almacenar más información e indexarla y usarla de forma más compleja, lo cual reduce la velocidad del predictor. Para subir la frecuencia de reloj más allá de lo que permite el avance en la tecnología, se aumenta la profundidad de la segmentación del procesador, limitando la cantidad de memoria y puertas lógicas que puede contener la ruta crítica de cada etapa.

En esta tesis se propone una organización jerárquica para el predictor. La idea básica consiste en predecir dónde se encuentran los datos relevantes imprescindibles para completar la predicción, y evitar así el retardo de tiempo que supone tener que leer todos los datos disponibles y tener que seleccionar la predicción final utilizando toda esta información. De este modo, el predictor acelera un alto porcentaje de las predicciones, a costa de aumentar el tiempo de predicción en un reducido conjunto de los casos. Como efecto lateral de la propuesta se reduce la energía consumida en el predictor, al evitar leer información que no es utilizada en la predicción.

Las contribuciones en este apartado se pueden resumir como:

- *Aplicar la técnica de predicción de vía al predictor. De este modo se evita en muchos casos el tiempo necesario para leer todas las vías de forma simultánea y tener que escoger la información de una de ellas.*
- *Aplicar la técnica de predicción de índice al predictor. Se dispone de una predicción por defecto que se utiliza para iniciar inmediatamente la siguiente predicción, mientras que el resto de información dedicada a la predicción, como el tipo de la instrucción de salto o la predicción de la dirección de los saltos condicionales, se utiliza fuera de la ruta crítica, y en ocasiones sirve para corregir la predicción por defecto.*
- *Analizar el espacio de diseño en el contexto de una jerarquía de dos niveles, que aúna la alta velocidad de predicción del primer nivel de la jerarquía con la alta capacidad para mantener información del segundo nivel de la jerarquía.*
- *Analizar el uso de la predicción anticipada para segmentar el predictor y acelerar su funcionamiento. La técnica se adopta para la predicción del sentido de los saltos condicionales, y se considera como alternativa para obtener las direcciones de salto.*

1.3. Predicción más Ancha

La alternativa al aumento de la velocidad del predictor es aumentar la anchura de cada predicción, es decir, predecir bloques que contengan más instrucciones. Incrementar la frecuencia del predictor es una opción limitada. En primer lugar, el núcleo básico del predictor debe tener un tamaño mínimo para ser efectivo, y por tanto una latencia mínima que no es posible reducir. En segundo lugar, la ley del rendimiento decreciente nos previene del alto coste que puede suponer, tanto en complejidad de diseño como en consumo energético, el aumentar la frecuencia más allá de un cierto valor.

Aumentar la anchura de las predicciones, en lugar de la velocidad de las predicciones, es análogo a aumentar la anchura del procesador en lugar de aumentar su frecuencia, y por tanto comparte sus ventajas e inconvenientes. La anchura del procesador (o del predictor) se aumenta explotando el paralelismo inherente a los programas, y por tanto requiere analizar las dependencias. En el caso del predictor, es necesario predecir múltiples instrucciones de transferencia de control (ITCs) en cada operación. Los problemas a los que se debe enfrentar una propuesta de este tipo son:

- evitar dependencias al predecir múltiples ITCs dentro del bloque de instrucciones, para que la latencia de predecir un bloque no crezca junto con la anchura del bloque.
- evitar la duplicidad de información en las tablas de predicción, o se desaprovechará un espacio de almacenamiento que podría ser utilizado para aumentar la precisión.

En esta tesis se propone un mecanismo de predicción de trazas de instrucciones que cumple los dos criterios anteriores. Las predicciones se hacen en forma de trazas de instrucciones, que se considera la unidad de predicción básica. De este modo, la predicción de las ITC internas a la traza se hace de forma implícita, y no hay que esperar a obtener el resultado de la predicción de una ITC para predecir la siguiente. La duplicidad de información

se reduce considerando un límite para el número de trazas que pueden comenzar en una misma instrucción.

Las contribuciones de esta parte del trabajo se pueden resumir como:

- *Analizar el predictor de trazas descrito en la literatura, y mostrar su falta de escalabilidad, en comparación con un predictor de bloques básicos, respecto a la relación entre precisión y memoria requerida.*
- *Demostrar que la predicción de trazas que incluyen múltiples ITCs no es significativamente menos precisa que la predicción de bloques básicos, que incluyen una única ITC. El corolario es que se puede aumentar la anchura del predictor sin reducir su precisión.*
- *Proponer y analizar el diseño detallado de un predictor de trazas de dos niveles, que codifica de forma local las predicciones para lograr alcanzar una relación entre precisión y tamaño del predictor similar a la del predictor de bloques básicos que se usa de referencia.*
- *El nuevo mecanismo de predicción de trazas requiere limitar el número de trazas que se consideran dentro de cada clase de trazas. Se proponen y se evalúan algoritmos que manejan los casos en que se produce desbordamiento dentro de una clase de trazas.*

1.4. Saltos Indirectos

La predicción de saltos indirectos es un factor que limita las prestaciones en ciertas aplicaciones. Los saltos indirectos suelen calcular la dirección de salto en base a sus datos de entrada, y la predicción precisa de los saltos indirectos con múltiples direcciones destino, o *polimórficos*, impone mayores requerimientos que la predicción de los saltos condicionales.

Las estrategias más recientes propuestas en la literatura utilizan la historia global de los saltos más recientes para acceder a una tabla específica de

predicción de saltos indirectos. En esta tesis se mostrará que estas propuestas no se escalan de forma eficiente al aumentar el tamaño de la tabla, pues almacenan múltiples veces la misma información.

Se propone una nueva organización que codifica las direcciones destino de los saltos indirectos, de forma que una primera tabla almacena la correlación con la historia previa, codificada con un identificador local, y una segunda tabla traduce el identificador local a la dirección destino completa. De este modo, se mejora la relación entre la precisión de las predicciones y la cantidad total de bits dedicados a las predicciones, lo cual permitirá aumentar la precisión del predictor o reducir sus requerimientos de memoria.

Las contribuciones en el tema de la predicción de saltos indirectos son, en resumen, las siguientes:

- *Analizar el predictor de saltos indirectos descrito en la literatura, y mostrar sus problemas de eficiencia y un mecanismo para evitarlos.*
- *Proponer y analizar el diseño detallado de un predictor de saltos indirectos de dos niveles, que codifica de forma local las predicciones para mejorar la relación entre precisión y tamaño del predictor.*
- *Mostrar el efecto muy reducido en el rendimiento del procesador que supone el pequeño incremento en la latencia de la predicción de saltos indirectos.*

1.5. Organización de la Memoria

Esta memoria de tesis se organiza tal como sigue:

- *En el capítulo 2 se presentan los conceptos básicos y el estado del arte sobre el tema de predicción del flujo de control en el procesador.*
- *En el tercer capítulo se describe la metodología experimental y se presenta con mucho detalle el diseño del predictor de bloques básicos*

que se utilizará como referencia en el resto del trabajo. En el capítulo se revisan y evalúan algunas aportaciones muy recientes al tema de la predicción de saltos que se han incorporado en el trabajo.

- *Las estrategias dedicadas a aumentar la velocidad del predictor se describen y se evalúan cuantitativamente en el capitulo 4. La exposición de las propuestas se hace de forma incremental, para facilitar la lectura. Al final del capítulo se hace una comparación en profundidad con el trabajo más directamente relacionado, que se utiliza para remarcar las contribuciones originales.*
- *El quinto capítulo analiza la problemática de aumentar la anchura del predictor, y propone y evalúa una nueva organización para predecir trazas de instrucciones, basada en un estudio previo de las propuestas de la literatura.*
- *El capítulo 6 propone un método más eficiente para predecir saltos indirectos que los que se han encontrado en la literatura. Los resultados finales muestran que el aumento de precisión, o la reducción de requerimientos de memoria, pueden ser utilizados para mejorar las prestaciones del procesador.*
- *La memoria de tesis concluye con un resumen de las conclusiones presentadas en cada uno de los capítulos, que sirve también para resaltar las aportaciones más importantes e indicar las publicaciones a las que han dado lugar. También se presentan las nuevas vías de trabajo para continuar y extender la investigación. Finalmente, se lista la biografía referenciada durante la memoria y utilizada en la investigación.*

2. Conceptos Previos y Estado del Arte

Resumen

Se presentan los conceptos previos y se revisa el estado del arte sobre el tema de predicción del flujo de control en el procesador.

2.1. Introducción

Describiremos los elementos que componen un procesador superescalar actual y los diferentes mecanismos de predicción de saltos que se han propuesto en la literatura. El capítulo comienza presentando el procesador superescalar y sus elementos básicos, describiendo el contexto del predictor de saltos, y analizando los problemas que más afectan a su rendimiento.

Posteriormente se hace un análisis histórico de las diferentes propuestas relativas a la predicción de saltos, desde trabajos pioneros influyentes y representativos, hasta las propuestas más recientes. Muchas de ellas constituyen los fundamentos de nuestra investigación, ya sea sobre aspectos más generales o más específicos. Estas referencias se agruparán por temas, en función de la relación que tengan con las propuestas de esta tesis. Otros de los trabajos referenciados atacan problemas complementarios al nuestro.

2.2. El Procesador y su Rendimiento

El procesador es un dispositivo electrónico que busca, decodifica y ejecuta instrucciones muy sencillas almacenadas en una memoria. La organización interna de un procesador actual es extremadamente compleja, ya que contiene muchos mecanismos que no están diseñados para que el procesador funcione correctamente, sino para acelerar su funcionamiento y para optimizar el consumo energético. Se pueden destacar cuatro ideas fundamentales para incrementar las prestaciones:

- *Memoria caché.* Para reducir el tiempo de acceso a la memoria del sistema y su consumo energético, se incluye dentro del procesador un subconjunto de la memoria principal, de acceso mucho más rápido, manejada de forma transparente al programa.

- *Ejecución segmentada.* Las instrucciones se ejecutan como una secuencia de ***n*** pasos, o etapas. Para acelerar la ejecución de los programas, se realizan simultáneamente la etapa ***r*** de la instrucción *i* y la etapa ***r-1*** de la instrucción *i+1*. Debe haber controles que aseguren que la funcionalidad del programa no varía.

- *Ejecución superescalar.* Múltiples instrucciones pueden realizar la misma etapa de segmentación simultáneamente.

- *Reordenación dinámica de instrucciones.* Se ejecutan las instrucciones en el orden en que se dispone de sus datos de entrada –lo antes posible–, que puede diferir del orden secuencial del programa.

La Figura 2.1 muestra el diagrama de bloques de un procesador, en el que se aprecian tres partes con funciones diferenciadas: el sistema de memoria, la unidad de búsqueda de instrucciones y el núcleo de ejecución.

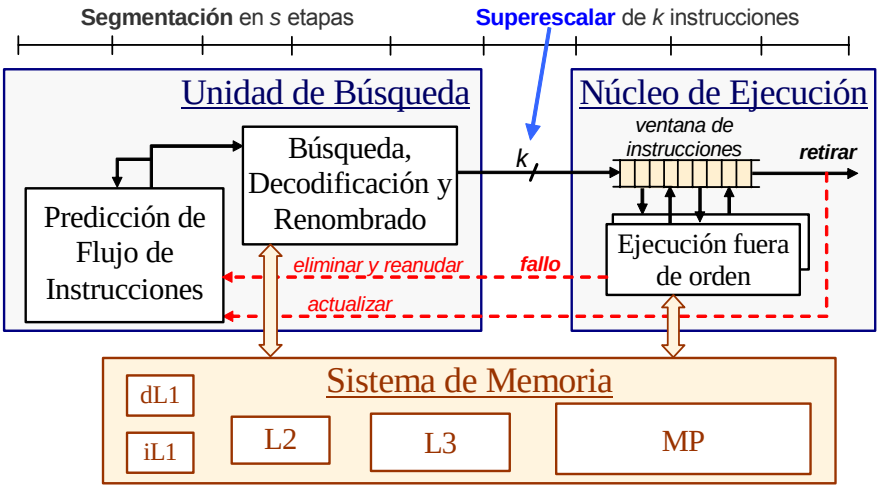


Figura 2.1. Diagrama de Bloques de un procesador, con ***s*** etapas de segmentación y una organización superescalar de ***k*** instrucciones por ciclo.

El *Sistema de Memoria* almacena datos e instrucciones. Se organiza como una jerarquía para así proporcionar una alta capacidad de almacenamiento pero con una latencia y consumo energético promedios de valor reducido.

La *Unidad de Búsqueda de Instrucciones* gestiona el flujo de control de los programas, obteniendo las instrucciones de memoria, analizándolas, y preparándolas para la ejecución. Para aumentar su rendimiento, se utiliza un predictor de flujo de control, que es el objeto de estudio de esta tesis.

El *Núcleo de Ejecución* ejecuta instrucciones según el orden dinámico que dictan sus dependencias de datos y recursos, orquestando el flujo de datos de entrada y salida entre varias unidades funcionales, lo que permite ejecutar múltiples instrucciones a la vez y en un orden diferente al original. Para asegurar el correcto funcionamiento del procesador ante eventos inesperados (interrupciones, faltas, fallos de predicción, ...), el efecto de cada instrucción en el estado del procesador se debe hacer irreversible de forma ordenada, es decir, se debe *retirar* las instrucciones en orden.

El Sistema de Memoria actúa como servidor de las peticiones realizadas por la Unidad de Búsqueda y por el Núcleo de Ejecución (los clientes), según determina la dinámica del programa. Se puede convertir en un elemento activo, y tratar de anticiparse a las peticiones, moviendo información a través de la jerarquía para minimizar el tiempo de respuesta y consumo energético.

Entre la Unidad de Búsqueda y el Núcleo de Ejecución se establece una relación de tipo productor-consumidor. El rendimiento de la Unidad de Búsqueda, por tanto, limita el máximo rendimiento del Núcleo de Ejecución.

2.2.1 La unidad de Búsqueda de Instrucciones

La unidad de búsqueda obtiene las instrucciones de memoria y las pre-procesa (decodifica instrucciones y renombra registros) antes de enviarlas al núcleo de ejecución. Para proporcionar instrucciones al núcleo de ejecución de forma ininterrumpida es necesario que el *predictor de flujo* prediga tanto el sentido como la dirección de los saltos. Las instrucciones se buscan y envían a ejecutar de forma especulativa. Muchas dependencias de control se resuelven en el núcleo de ejecución. Si la predicción de control resulta ser

incorrecta, se deben eliminar las instrucciones generadas en el camino erróneo, se han de revertir los efectos que han provocado, y se debe reanudar la operación de predicción y búsqueda de instrucciones en el punto adecuado. Al retirar una instrucción de salto se procede a la actualización de la información interna del predictor.

El ancho de banda de la unidad de búsqueda –instrucciones producidas por segundo– viene determinado por su anchura máxima y por su tiempo de ciclo. La *anchura* máxima es el máximo número de instrucciones que puede producir en una operación, y el *tiempo de ciclo* es el tiempo mínimo entre la generación de un bloque de instrucciones y el siguiente.

En último término, el ancho de banda de la unidad de búsqueda viene limitado por la anchura y velocidad del predictor. Los fallos en la caché de instrucciones pueden aumentar puntualmente el tiempo de respuesta de la unidad de búsqueda, y reducir su ancho de banda. Si no hubiera fallos de predicción, bastaría con usar las predicciones con suficiente antelación para leer las instrucciones de memoria en el momento adecuado, sin penalización. La ejecución especulativa, con los fallos de predicción potenciales, requiere de una fase de realimentación que indique al predictor el resultado de la predicción para que éste “aprenda” y, en caso de error, corrija su operación.

Se pueden distinguir dos tipos de errores de predicción. Al primer tipo lo denominamos *fallo de decodificación*, se produce cuando el predictor no identifica correctamente una instrucción de control, y los detecta la propia unidad de búsqueda. Se produce bien porque la instrucción de control aparece por primera vez (fallo en frío) o bien por falta de capacidad de almacenamiento (fallo de capacidad del predictor). El segundo tipo de fallo, que denominamos *fallo de predicción*, se debe a la incapacidad del algoritmo de predicción para determinar el comportamiento de un salto utilizando la información disponible. Estos errores se detectan en el núcleo de ejecución.

El número de etapas de segmentación de la unidad de búsqueda aumenta el tiempo necesario para notificar al predictor el resultado, erróneo o no, de

una predicción, y por tanto aumenta el valor de la penalización de los fallos de predicción.

2.2.2 El Núcleo de Ejecución

El núcleo de ejecución envía las instrucciones a las unidades funcionales según el orden dinámico que dictan las dependencias de datos y de recursos. Sus parámetros básicos son la anchura de entrada y la anchura de ejecución, el número de etapas de segmentación, la latencia de las instrucciones, y el tamaño de la ventana de instrucciones. La *latencia* de una instrucción se define como el número de ciclos necesarios para que la instrucción, una vez que ha recibido todos sus datos de entrada, proporcione su resultado a una instrucción que necesita ese dato.

Las anchuras de entrada y ejecución determinan el máximo ancho de banda del núcleo. El número de etapas de segmentación aumenta el tiempo necesario para notificar al predictor el resultado de la predicción. El *tamaño de la ventana* de instrucciones indica el máximo número de instrucciones "en vuelo", incluyendo instrucciones que esperan sus datos de entrada o un recurso, que están en ejecución, o que esperan a ser retiradas. Se suele definir un tamaño de ventana específico para los accesos a memoria. El tamaño de la ventana limita la distancia máxima entre instrucciones que se pueden ejecutar en paralelo: cuanto mayor es, mayor es el grado de paralelismo a nivel de instrucción que se puede utilizar para aumentar el rendimiento. Tal como se muestra en la introducción, aprovechar la amplia ventana requiere que *la relación entre la frecuencia de fallos de predicción y la velocidad a la que se llena la ventana sea baja*.

La latencia de las instrucciones es un factor crítico para el rendimiento. La latencia de las instrucciones sólo puede ocultarse si se dispone de suficiente paralelismo a nivel de instrucción. Cuanto mayor es la latencia, más difícil será encontrar instrucciones independientes. Las instrucciones de acceso a memoria suponen quizás el caso más crítico respecto a la latencia.

2.2.3 El Sistema de Memoria

La Figura 2.1 muestra la jerarquía de cuatro niveles que se utiliza en los experimentos realizados en este trabajo. Si los datos se encuentran en la caché de primer nivel, los accesos se resuelven en 2-3 ciclos. Sin embargo, para ciertas aplicaciones es frecuente tener que buscar los datos en los niveles superiores de la jerarquía. Esta elevada latencia es difícil de ocultar ejecutando instrucciones independientes, pues muchas de las instrucciones en la ventana pueden depender del dato buscado en la memoria, o incluso peor, se ha realizado una predicción de saltos incorrecta dependiente de este dato. Por tanto, para aprovechar una gran ventana de instrucciones se requiere de unas predicciones precisas.

Los parámetros que definen la jerarquía de memorias son su capacidad, su organización (asociatividad, línea, política de reemplazamiento), la latencia y el ancho de banda de cada nivel. La máxima cantidad de fallos intermedios que pueden estar pendientes entre cada uno de los niveles determina el grado de paralelismo del sistema de memoria. Finalmente, se puede usar un mecanismo de prebúsqueda dirigido por los accesos y fallos previos.

2.2.4 El Rendimiento del Procesador

La fórmula que determina el tiempo de ejecución de un determinado programa es la siguiente [HePa03]:

$$T_{ejecución} = N \cdot CPI \cdot T_{ciclo}$$

N es el número total de instrucciones del programa y depende del algoritmo utilizado, de la codificación del programa, del compilador y del repertorio de instrucciones. *CPI* es el promedio del número de ciclos necesarios para ejecutar cada instrucción. La métrica inversa, *IPC*, se define como el promedio de instrucciones ejecutadas por ciclo. Este valor depende, por un lado, de la microarquitectura del procesador (número y latencia de las unidades

funcionales, tamaño y latencia de la memoria caché, ...) y por otro lado de características del programa (dependencias de datos, patrón de accesos a memoria, proporción de cada tipo de instrucción, …). T_{ciclo} es el tiempo de ciclo del reloj, y determina la velocidad del procesador. La tecnología de implementación del procesador y la forma en que la microarquitectura se traslada a un diseño físico determinan el mínimo T_{ciclo} del procesador, y también su consumo energético y el área de chip requerida.

En esta tesis se proponen modificaciones en la microarquitectura que tienen incidencia en *CPI* y en T_{ciclo}. La evaluación de estas propuestas se hará utilizando programas de evaluación codificados siempre de la misma manera, y se ignorarán los factores involucrados en la generación de ese código. Por tanto, el factor *N* de la ecuación no variará en ningún caso. Para evaluar el efecto de modificar la microarquitectura en T_{ciclo} (y en el consumo energético y área de chip) se requiere un modelo detallado del procesador a nivel de implementación. Esta tarea es muy compleja, y se ha optado por realizar estimaciones indirectas por medio de herramientas que modelan los detalles de implementación de los elementos de memoria (que se describirán en el próximo capítulo). El modelo de la microarquitectura sí se simula con suficiente detalle para obtener valores precisos de *CPI*.

Cada problema encontrado durante la ejecución hace que el *CPI* crezca por encima del máximo valor teórico alcanzable por la microarquitectura. La siguiente fórmula muestra que el incremento de *CPI* depende del número de ciclos de penalización producidos por el problema, y de la frecuencia con la que se produce respecto al número total de instrucciones ejecutadas:

$$CPI_{total} = CPI_{base} + penalización \cdot frecuencia$$

Por tanto, el impacto de un problema en el rendimiento del procesador es importante tanto para problemas moderadamente frecuentes con una alta penalización, como para problemas muy frecuentes, aunque tengan una pequeña penalización. Además, el porcentaje de reducción en el rendimiento debido a un problema dependerá del rendimiento teórico máximo del que es

capaz una microarquitectura. Este resultado se conoce como *ley de los rendimientos decrecientes*, que es una ley general de la naturaleza, y que se puede formular de este modo: la mejora del rendimiento en un sistema (un procesador) decrece a medida que se aplican más recursos al sistema.

Otra ley importante que describe el comportamiento del rendimiento de un sistema es la *ley de Amdahl*. Una posible formulación de esta ley es que al añadir recursos a un sistema para reducir el efecto del problema más importante en el rendimiento, algún o algunos problemas que tenían menos importancia relativa, se convierten en el nuevo cuello de botella del sistema. En otras palabras, que toda mejora en un sistema (un procesador) debe realizarse de forma equilibrada.

2.2.5 Resumen

Los parámetros de la microarquitectura del procesador que influyen en su rendimiento se pueden clasificar como **latencias** de operaciones, **anchuras** de operación, y **capacidades** de almacenamiento. La anchura y la capacidad es relativamente fácil de escalar, aprovechando el mayor número de transistores que la tecnología permite integrar en un chip. Reducir la latencia de una operación es una tarea mucho más compleja. Las mejoras tecnológicas reducen el tiempo de operación de los transistores, pero la dificultad de reducir el tiempo de transmisión de señales entre transistores y el calor generado por un circuito que funciona a muy alta frecuencia son un claro ejemplo de la ley de rendimiento decreciente [AHK+00].

Aumentar la anchura y la capacidad del procesador permite realizar operaciones en paralelo para poder solapar las latencias, aunque también suponga aumentar estas latencias, por tener que comunicar entre sí más elementos. Además se necesita encontrar el paralelismo en las aplicaciones. Esta tesis trata de proporcionar ese paralelismo a nivel de instrucción mediante la predicción rápida y precisa del flujo de instrucciones de un programa.

Las latencias que afectan al rendimiento de forma más significativa se muestran en la siguiente figura. t_{pred} es la latencia del predictor y determina el ancho de banda de predicción, que supone el primer límite al rendimiento del procesador. t_{pipe} es el tiempo mínimo que transcurre desde que el predictor proporciona la referencia a las instrucciones hasta que éstas están listas para ser ejecutadas. t_{ejec} es la latencia para ejecutar las instrucciones. t_{mem} es la latencia de acceso a memoria en caso de que los datos o instrucciones no se encuentren en el primer nivel de caché. t_{ret} es el tiempo mínimo desde que una instrucción ha sido ejecutada hasta que se informa al predictor de que ha sido retirada correctamente.

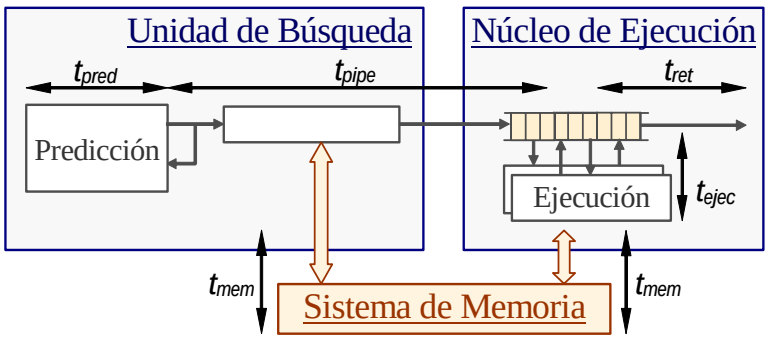


Figura 2.2. Latencias de los elementos del procesador que afectan al rendimiento.

El tiempo de penalización debido a un fallo de predicción es $t_{pipe}+t_{ejec}$ más el tiempo de espera en colas intermedias, que depende de t_{mem} y de las dependencias de datos entre las instrucciones. Los fallos de decodificación no necesitan ejecutar la instrucción para ser detectados, ni por tanto respetar las dependencias de datos, y tienen una penalización mucho menor.

Mientras que informar de un error de predicción lo antes posible es muy importante para mejorar el rendimiento, actualizar el predictor no es una tarea crítica. Es preferible esperar a que las instrucciones de salto sean retiradas, aunque ello suponga retardar la actualización de las tablas de predicción, y así evitar la complejidad de tener que deshacer las modificaciones realizadas especulativamente.

2.3. Trabajo Relacionado

La tarea del predictor de flujo de control consiste en realizar, a partir de la dirección de la instrucción actual, las siguientes operaciones:

- *Identificar la posición y tipo de las instrucciones de salto*
- *Predecir los saltos condicionales: saltar o no saltar*
- *Predecir los saltos indirectos: predicción múltiple*
- *Proporcionar las direcciones de memoria destino de los saltos*

Cada una de estas cuatro operaciones se puede realizar de diferentes maneras. Denominamos arquitectura del predictor de flujo de instrucciones a la organización general utilizada para llevar a cabo estas cuatro tareas. Comenzaremos el repaso al estado del arte con los mecanismos de predicción de saltos condicionales, los más frecuentes y que suelen penalizar más el rendimiento. Posteriormente se describen mecanismos de predicción de direcciones de salto, incluyendo los saltos indirectos. A continuación se muestran diferentes arquitecturas del predictor. Finalmente se repasan estrategias para aumentar la anchura y la velocidad del predictor. Se concluye con alguna referencia a los estudios sobre el consumo energético.

2.3.1 Predicción de Saltos Condicionales

La precisión en la predicción de saltos condicionales es la que suele tener más influencia en el rendimiento del procesador. En este apartado sólo se considera la predicción del sentido de los saltos, una predicción binaria, y no la forma de obtener la dirección de memoria destino del salto.

La forma más simple de predecir un salto condicional es ignorarlo, predecir implícitamente que no salta. Se puede sofisticar el predictor usando características estáticas del programa. Por ejemplo, algunos modelos del IBM System 360/370 diferencian entre saltos que implementan bucles, que se predice que siempre saltan, y saltos para sentencias de tipo `IF/THEN/ELSE`

que se predice que nunca saltan. El signo del desplazamiento del salto se puede usar para tomar la decisión, apostando por saltar sólo si es negativo, es decir, si la dirección donde saltar es anterior a la dirección de la instrucción de salto. Este esquema se basa en la observación de que estos saltos suelen corresponder a bucles en los que el caso más frecuente es saltar.

El problema del enfoque estático es que las decisiones no tienen en cuenta el comportamiento particular de cada salto. Se puede incluir un bit explícito dentro del formato de la instrucción, tal y como hace el Motorota 88110 [DiAl92], para realizar una predicción estática con cada instrucción de salto, independientemente de su tipo. Esta predicción se puede basar en un análisis del código, o en la ejecución del programa con datos de muestra para obtener perfiles que capturen parte del dinamismo de la ejecución [FiFr92].

Los mecanismos dinámicos de predicción no requieren de la ayuda del compilador y utilizan la información producida directamente por el programa. J. Smith propuso un predictor dinámico que utiliza un contador de 2 bits saturado asociado a cada salto [Smit81]. El contador se incrementa (como mucho hasta 3) cada vez que el sentido del salto es saltar, y se decrementa (como mucho hasta 0) cuando el sentido es no saltar. El bit más significativo del contador predice el sentido del salto la siguiente vez que aparece.

Un importante avance en la predicción de saltos condicionales se produjo cuando Yeh y Patt, [YePa91] [YePa92a], por una parte, y Pan *et al.* [PaSR92], por otra, observaron que frecuentemente el resultado de un salto está correlacionado con el resultado de otros saltos recientes. La historia de los saltos más recientes proporciona un patrón que puede utilizarse para proporcionar un contexto dinámico a la predicción. Cada vez que se conoce el resultado de un salto se actualiza la historia de los últimos N saltos (1 indica salto tomado y 0 indica salto no tomado). Esta historia se combina con la dirección del salto que se quiere predecir para indexar una tabla de contadores de 2 bits saturados (*Branch History Table, BHT*) que proporciona la predicción. Una gran parte de los predictores actuales se basan en este esquema de dos niveles, que se describirá con detalle en el capítulo 3.

Mucha investigación posterior ha tratado de mejorar la precisión refinando el esquema del predictor de 2 niveles. Uno de los problemas importantes que se ha atacado es evitar que dos casos de predicción sean emplazados en la misma entrada de *BHT*, produciendo pérdida de información y afectando negativamente al comportamiento del predictor (interferencia destructiva). La utilización de etiquetas (*tag*s) no hace un uso eficiente de la memoria, y a cambio se han propuesto numerosos mecanismos de indexación y selección de la información para reducir este efecto de interferencia [ChEP97].

Uno de los mecanismos más populares se conoce como *gshare*, y combina la historia de saltos con la dirección del salto utilizando una operación o-exclusiva [McFa93]. El efecto de esta operación es distribuir de forma más homogénea los accesos a *BHT*, reduciendo la probabilidad de que dos saltos interfieran uno con el otro en la tabla. Una forma más elaborada de distribuir los bits del índice a la tabla fue propuesta en [JBSS97] y [JaRS97], y se basa en aplicar la operación o-exclusiva de forma más exhaustiva. En el capítulo 3 mostraremos como se utiliza este esquema.

Otros ejemplos de estrategias destinadas a reducir el problema de alias en *BHT* son, entre otros, *Agree* [SCAP97], *Bi-Mode* [LeCM97], *YAGS* [EdMu98], y *Skew* [MiSu97]. En el capítulo 5 se describirá este último método, que ha sido utilizado en uno de los experimentos descritos en este trabajo, y que fue seleccionado para el diseño del Alpha EV8 [SFKS02].

La precisión de los predictores se puede mejorar combinando varios mecanismos en forma de un predictor híbrido. McFarling combinó *gshare* y un predictor *bimodal* (propuesta de Smith), con una tabla de selección, o predictor de torneo, compuesta de contadores de 2 bits [McFa93]. Se decide de forma dinámica cual es el predictor más adecuado para cada tipo de salto, o incluso para cada caso de predicción. El esquema de McFarling fue extendido por Evers *et al*. a una organización que permite combinar un número cualquiera de predictores mediante el uso de pequeñas máquinas de estados usadas como mecanismo de selección [EvCP96].

Stark *et al.* proponen un mecanismo dinámico para adaptar la longitud de la historia de saltos de forma dinámica a la predicción de cada salto particular [StEP98]. Tarlescu *et al*. proponen un esquema estático para seleccionar la longitud de historia utilizada por cada salto [TaTG96]. Una alternativa complementaria es adaptar la longitud de la historia a las diferentes fases de ejecución de un programa [JuSN98].

El uso de historias de saltos de gran longitud ofrece más oportunidades de correlacionar las predicciones, pero su uso con predictores de dos niveles hace necesario aumentar los requerimientos de memoria y aumenta el tiempo de aprendizaje del predictor. El predictor *perceptrón* asigna una única entrada a cada salto (y no a cada pareja salto-historia), que consiste en el estado de un tipo de red neuronal muy sencilla [JiLi02] [Jime03b]. El *perceptrón* puede aprender funciones booleanas sencillas de forma relativamente más rápida que el esquema de correlación de dos niveles. Otro enfoque para buscar saltos correlacionados en historias de gran longitud es el análisis explícito de las dependencias de los registros [TFWS03].

2.3.2 Predicción de Direcciones de Salto

La dirección destino de un salto predicho como tomado debe calcularse. Si la dirección destino se representa en forma relativa a PC, entonces se calcula sumando a la dirección de la instrucción de salto una constante contenida en la propia instrucción. En procesadores segmentados, el retardo de calcular la dirección supone una importante penalización. Calcular la dirección destino de un salto indirecto es aún más problemático, y se comentará en breve.

Para evitar la penalización de tener que calcular las direcciones destino, éstas se almacenan en una tabla que se suele denominar *BTB* (*Branch Target Buffer*), descrita por Lee y Smith [LeSm84]. En ocasiones la tabla también se utiliza para identificar los saltos dentro de la secuencia de instrucciones y proporcionar su tipo.

Johnson [John91] propuso un diseño donde la *BTB* se integra en la caché de instrucciones. Cada línea de la caché contiene un índice que apunta a la siguiente línea y a la primera instrucción dentro de este bloque. El problema del diseño es que si una línea contiene dos o más instrucciones de salto, se crea contención en la predicción. Bray y Flynn [BrFl91] analizaron el efecto de añadir información para dos o más saltos por línea, y concluyeron que es más eficiente el uso de una *BTB* separada para almacenar las direcciones (o índices) destino, ya que es un esquema más flexible que permite optimizar por separado cada una de las organizaciones.

Calder y Grunwald [CaGr94a] llegan a la misma conclusión de separar caché de instrucciones y *BTB*, pero además proponen separar la información de predicción dinámica de saltos condicionales (*BHT*) de la *BTB*, para que las instrucciones de salto condicional que fallan en *BTB* puedan seguir usando predicciones dinámicas. En [CaGr95], Calder y Grunwald también proponen sustituir las direcciones de salto en *BTB* por punteros a la caché de instrucciones. Este esquema, y otras ampliaciones, se describirán con más detalle en los capítulos 3 y 4.

2.3.3 Predicción de Saltos Indirectos

Los saltos indirectos son saltos incondicionales que utilizan como dirección destino una dirección de salto almacenada en un registro o en memoria. Potencialmente, una instrucción de salto indirecto puede llegar a tener un conjunto enorme de direcciones destino durante la ejecución de un programa. Esta característica complica la predicción de estos saltos.

Un uso típico de las instrucciones de salto indirecto es implementar el retorno de una subrutina. Cuando se invoca una subrutina, la dirección de retorno se almacena en un registro o en memoria, y cuando ha de devolver el control al programa que la invocó, utiliza esa dirección de retorno para realizar un salto indirecto. El esquema anidado de las llamadas a subrutina facilita su predicción. Kaeli y Emma [KaEm91] propusieron usar una estructura

de datos de tipo pila (*return address stack*, RAS) para predecir estos saltos, replicando el comportamiento del programa. Cuando se produce un fallo de predicción, la pila *RAS* se actualiza con datos erróneos. Recuperar el estado correcto de la pila al detectar el error previo no es sencillo. En [SAMC98], se propone un mecanismo para reparar la pila *RAS* basado en almacenar la información sobrescrita durante la actualización especulativa de la pila.

Las llamadas a funciones de bibliotecas vinculadas dinámicamente se implementan con saltos indirectos para permitir que las funciones se instalen, antes o después del inicio de la ejecución del programa, en posiciones de memoria que no se conocen en el momento de compilar el programa principal. Una vez establecida la correspondencia del salto indirecto con su dirección de salto, esta correspondencia se mantiene durante toda la ejecución (o al menos durante un tiempo considerable). Estos saltos, u otros cuya característica principal es que la dirección destino del salto varía muy infrecuentemente, pueden ser predichos usando *BTB*.

Calder y Grunwald [CaGr94b] muestran en su trabajo que el uso de la programación orientada a objetos aumenta la influencia de la predicción de saltos indirectos en el rendimiento del procesador. En estas aplicaciones se usan los saltos indirectos para invocar el método adecuado de una clase (llamadas virtuales), en función de su herencia y del tipo de la clase. Se analizan algunas técnicas estáticas y dinámicas para mejorar la precisión de las predicciones de los saltos indirectos. Entre ellas, proponen usar un bit de histéresis en *BTB* para permitir la actualización de la dirección destino de un salto indirecto sólo después de la aparición dos veces consecutivas de una dirección destino diferente.

El otro ejemplo clásico de uso de los saltos indirectos son las estructuras de decisión múltiple (sentencias `switch` en lenguaje C), que determinan la ejecución de un conjunto específico de instrucciones para cada uno de los diferentes valores o rangos de valores tomados por una variable. Su uso es muy frecuente en algoritmos de análisis de texto, por ejemplo en un

compilador, o en un intérprete. Los saltos indirectos permiten sustituir una larga cadena de comparaciones y saltos condicionales. El valor a comparar se usa para generar el índice a una tabla, que contiene direcciones de salto a las instrucciones asociadas a cada valor.

Nair [Nair95] propone codificar la historia de los saltos previos usando las direcciones destino, en lugar de usar un bit por salto que indique si el salto ha saltado o no. A este tipo de codificación la denomina historia del camino de los saltos (*path history*), en contraposición a la historia del patrón de saltos (*pattern history*). Sus experimentos muestran que la mayor información contenida en la historia del camino permite obtener una mayor precisión. El trabajo de Nair trata sólo los saltos condicionales, pero se usó posteriormente para la predicción de saltos indirectos.

Chang, Hao y Patt [ChHP97] propusieron utilizar una tabla específica para almacenar direcciones destino de saltos indirectos (*target cache*) y aplicar a la predicción de los saltos indirectos el mismo método de predicción de dos niveles propuesto anteriormente para los saltos condicionales [YePa91]. En su trabajo analizan diferentes variantes a la codificación de la historia propuesta por Nair, diferentes mecanismos de indexación, y el uso de etiquetas.

Driesen y Hölzle [DrHo98a] exploran un gran número de variaciones a la propuesta de Chang *et al*. En un trabajo posterior, [DrHo98b], proponen un nuevo esquema de predicción utilizando dos tablas, que denominan predicción *en cascada*. Aunque su trabajo hace referencia únicamente a la predicción de saltos indirectos, su idea ha sido recogida posteriormente y aplicada en la predicción de saltos condicionales. La idea consiste en mantener en la primera tabla la información sobre los saltos fáciles de predecir, aquellos que no necesitan usar correlación con la historia previa, y usaar la segunda tabla para la información sobre los saltos difíciles de predecir, y que se benefician de la correlación con la historia previa. La clasificación de los saltos se hace de forma dinámica. En [DrHo99], los autores presentan una generalización a su propuesta de predicción en cascada, en la que utilizan múltiples tablas en lugar de sólo dos.

Hay dos diseños comerciales, el Intel Pentium IV [BBH+04] y el Intel Pentium M [GRB+03], que utilizan la organización en cascada de *BTB* y de una tabla específica para la predicción de saltos indirectos. Analizaremos la organización en cascada con mucho mayor detalle a lo largo de esta memoria, ya que se usa como referencia y como punto de partida para alguna de las propuestas de esta tesis.

Stark, Evers y Patt [StEP98] presentaron un mecanismo similar a la *Target Cache* en el que proponen variar de forma dinámica el número de saltos incluidos dentro de la historia. Kalamatianos y Kaeli [KaKa98] proponen aplicar el algoritmo de coincidencias parciales (*partial matching*), usado en el campo de la compresión de datos, a la predicción de los saltos indirectos. Su propuesta consiste en utilizar varias tablas a las que se accede en paralelo, cada una de ellas indexada usando la historia de camino con diferentes longitudes. La tabla que proporciona una predicción válida y correspondiente a la historia de mayor longitud es seleccionada.

2.3.4 Arquitecturas de la Unidad de Búsqueda

Buscar múltiples instrucciones por ciclo no es una simple cuestión de replicar recursos, debido sobre todo a los saltos tomados. A continuación se describen varias arquitecturas de la unidad de búsqueda, que se distinguen por el elemento básico que se usa como unidad de predicción, y por el diferente grado de acoplamiento entre la predicción y la búsqueda.

Organización Acoplada

Si el predictor y la caché de instrucciones funcionan fuertemente acoplados, la máxima anchura de predicción la determina el tamaño del bloque de caché leído (w). La Figura 2.3 muestra un ejemplo de este esquema. Se accede a la vez a un bloque de la caché y a los predictores específicos. La decodificación de las instrucciones permite identificar la posición y el tipo de los saltos. Esta

información se combina con las w predicciones de saltos condicionales, para generar la máscara con la que seleccionar las instrucciones válidas del bloque leído. La información contenida en las instrucciones y la que proporcionan los predictores de saltos indirectos permite generar la dirección del siguiente bloque de instrucciones (Dir_{i+1}).

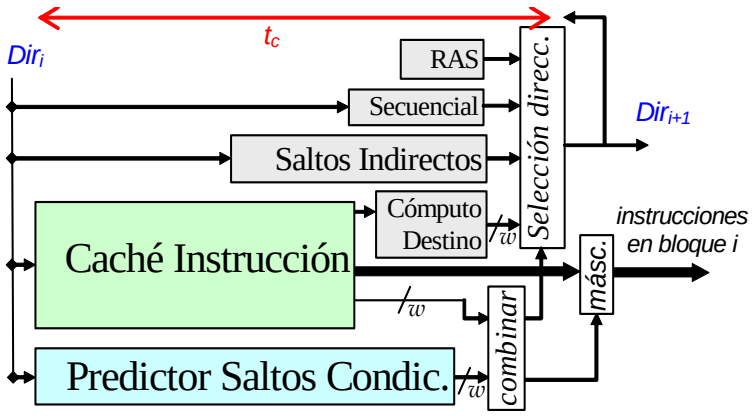


Figura 2.3. Diagrama de Bloques de una Unidad de Búsqueda Acoplada que permite generar bloques de w instrucciones.

El funcionamiento de la unidad de búsqueda se puede acelerar con la predicción de la línea de la caché que contiene las siguientes instrucciones, que supone una predicción implícita de vía, y permite reducir el consumo energético, [CaGr95] [Kess99] [Yung96]. En esta tesis se presentará un esquema análogo para acelerar la unidad de predicción en una unidad de búsqueda desacoplada, que se describirá a continuación.

El problema de los diseños precedentes es que sólo se puede predecir un salto tomado por ciclo, y que los bloques de instrucciones deben estar alineados en la caché. Es posible entrelazar la caché de instrucciones para leer dos bloques de instrucciones por ciclo y obtener una secuencia que atraviese los límites de una línea de caché [RoBS96]. Para obtener una instrucción de salto y la instrucción destino del salto en el mismo ciclo es necesario generar más de una dirección por ciclo, y requiere el uso de una

BTB con predicción múltiple, que se describirá en el apartado siguiente. El buffer de colapso [CMMP95] utiliza una red de alineación e intercambio que permite detectar saltos dentro de una misma línea de caché y generar una secuencia de instrucciones correspondiente a varias iteraciones de un bucle

En lugar de usar el buffer de colapso en la fase crítica de búsqueda de instrucciones, se puede usar al retirar las instrucciones y almacenarlas en el orden dinámico de ejecución dentro de una caché de trazas, [PeWe94] [RoBS96] [RoBS99]. Una unidad de búsqueda basada en trazas (ver Figura 2.4) captura el comportamiento dinámico del programa para simplificar la búsqueda. Aunque las primeras propuestas usaban predictores de múltiples saltos, descritos en la sección siguiente, un mecanismo más económico y efectivo es la predicción directa de trazas, considerándolas unidades básicas de predicción [JaRS97]. Este esquema se analizará extensamente en el capítulo 5, donde se usa de base para nuevas propuestas.

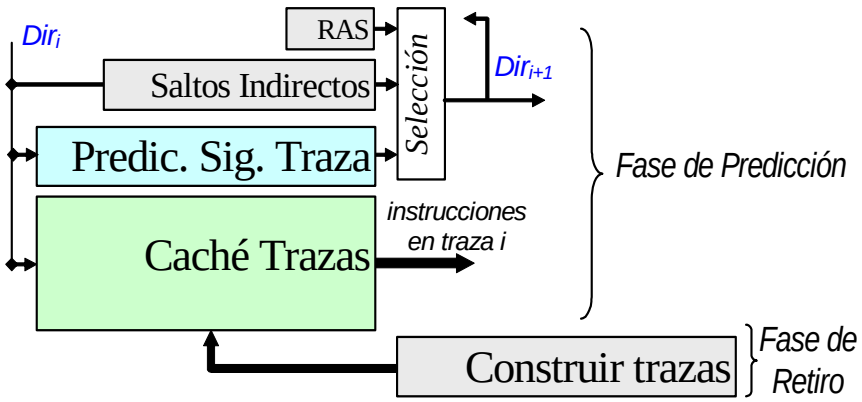


Figura 2.4. Diagrama de una Unidad de Búsqueda Acoplada basada en Trazas.

Para mejorar la eficiencia de la caché de trazas se ha propuesto la detección de coincidencias parciales [FrPP98]. Para reducir la redundancia debida a que las instrucciones se pueden almacenar múltiples veces, se ha propuesto utilizar un nivel extra de indirección, y almacenar las trazas como apuntadores a bloques secuenciales, accedidos en paralelo, y de donde se

seleccionan las instrucciones con un mecanismo mucho más simple que el buffer de colapso [BIRS99]. Se han analizado técnicas para incrementar la longitud de las trazas [PaEP98], para filtrar los accesos a la caché y reducir los fallos de caché y el consumo energético [RAM+04], y se han estudiado diferentes políticas de selección de trazas [RMSR03].

Organización Desacoplada

Ha habido varios pasos en el desacoplo de los elementos de la unidad de búsqueda. Calder y Grunwald [CaGr94a] proponen separar *BTB* y caché de instrucciones, y también separar *BHT* y *BTB*. El predictor de bloques básicos de Yeh y Patt [YePa92b] usa un buffer entre la caché de instrucciones y la etapa de decodificación para compensar el diferente tamaño de los bloques: los que tienen menos instrucciones que la capacidad de decodificación se compensan con bloques que contienen más instrucciones. Reinman *et al.* van aún más allá, y desacoplan el predictor de bloques básicos y la caché de instrucciones, con una cola de predicciones que denominan *FTQ* (*Fetch Target Queue*) en [ReAC99]. La Figura 2.5 muestra el diagrama de bloques.

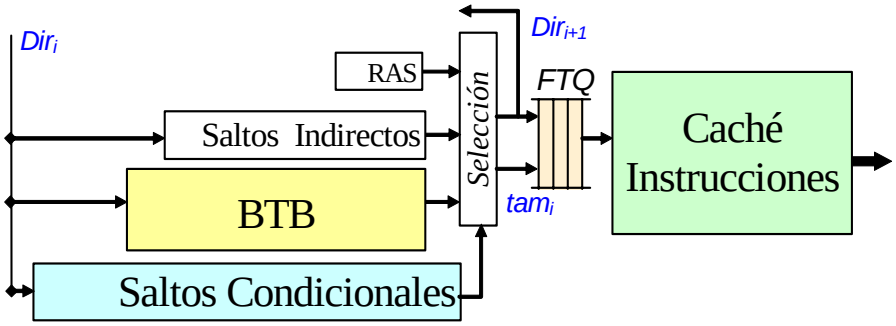


Figura 2.5. Diagrama de Bloques de una Unidad de Búsqueda Desacoplada.

El predictor genera referencias a bloques básicos, mientras que la caché consume las predicciones, de modo que una predicción puede suponer varios accesos a la caché. El tamaño de la cola de predicciones determina cuánto se puede distanciar la operación del predictor, de los accesos a la caché. Se

independizan las latencias del predictor y de la caché de instrucciones, lo cual permite segmentar el acceso a la caché para que la unidad de búsqueda funcione a la frecuencia del predictor. Otra ventaja es que se toleran retardos puntuales en el predictor, o que se pueden paralelizar las operaciones en la unidad de búsqueda [ObSo03]. En [ReCA01] y [ReCA02] se analizan la prebúsqueda de instrucciones dirigida por el predictor de saltos y la reducción del consumo energético con una caché de acceso serie.

El predictor de bloques básicos se puede mejorar ignorando los saltos que casi nunca son tomados, creando una unidad de predicción (el bloque de búsqueda) con un tamaño medio mayor que el de los bloques básicos, y así aumentar la anchura del predictor [ReAC99]. Se puede utilizar una política aún más flexible y permitir que la unidad de predicción contenga múltiples saltos que no saltan (*stream* de instrucciones), tal y como proponen Ramírez *et al*. [RSLV02]. Así se aumenta la anchura del predictor, manteniendo un esquema sencillo de acceso a las instrucciones. El mecanismo de predicción múltiple que se utiliza es una adaptación del predictor de trazas de [JaRS97]. En esta tesis se da un paso más allá en el aumento de la anchura del predictor y se propone usar un predictor de trazas desacoplado de la caché.

2.3.5 Aumentando la Anchura del Predictor

En el apartado anterior se han descrito propuestas que aumentan la anchura del predictor integrando múltiples saltos en una misma unidad de predicción (bloque de búsqueda, *stream* o traza de instrucciones). En este apartado se describen otras alternativas para predecir múltiples saltos por ciclo.

Yeh *et al.* [YeMP93] proponen incluir dos puertos de acceso en un predictor de dos niveles basado en historia global, de forma que el resultado de predecir un primer salto se utilice para seleccionar la predicción de un segundo salto, durante el mismo ciclo de predicción. Dutta y Franklin [DuFr99] organizan el predictor de dos niveles en forma de árbol, y cada entrada contiene la historia local, la dirección destino, y el tamaño del bloque básico

de cada uno de los 2^K caminos posibles. Friendly *et al.* [FrPP97] también proponen el uso de una tabla de predicción de saltos condicionales con 2^K-1 contadores saturados, que se usan en cascada para generar la predicción de *K* saltos. Una propuesta posterior de Patel *et al.* [PaFP99] reemplaza el árbol de contadores por K contadores, para dar lugar a un diseño más eficiente.

Finalmente, el predictor propuesto por Menezes *et al.* [MeSC97] codifica los diferentes caminos de ejecución con tantos bits como instrucciones de salto puede contener el camino. La idea guarda cierta similitud con la propuesta de codificación local de trazas descrita en el capítulo 5. La principal diferencia es que nuestra propuesta permite predecir trazas que contienen muchos más saltos que el número de bits que se utilizan para codificarlos.

Ninguna de las propuestas anteriores ataca de forma eficiente el problema de generar las direcciones destino de los saltos condicionales. La propuesta de Yeh *et al.* [YeMP93] utiliza una caché de direcciones de salto que proporciona dos direcciones destino por ciclo, pero que contiene direcciones duplicadas y una alta cantidad de campos sin utilizar. Séznec *et al.* [SJSM96] mejoran esta propuesta con una *BTB* de doble puerto de lectura, en la que se usa la información asociada al bloque de instrucciones actual para predecir la dirección del bloque posterior al bloque siguiente. Esta idea se analizará en el capítulo 4 para acelerar la velocidad del predictor. Wallace y Bagherzadeh [WaBa97] propusieron una modificación para evitar la comparación encadenada de etiquetas necesaria para asegurar los aciertos en *BTB*.

2.3.6 Acelerando la velocidad de Predicción

La idea de acelerar la velocidad de un predictor utilizando una jerarquía de predictores ha sido discutida por Jiménez *et al.* [JiKL00]. Un predictor pequeño y simple proporciona predicciones a alta velocidad, mientras que un segundo predictor, más lento y grande, pero a la vez más preciso, corrige las predicciones del primero. Aunque las correcciones suponen una penalización, ésta es mucho menor que la que produciría un fallo de predicción.

El predictor desacoplado de Reinman *et al.* [ReAC99] también usa dos niveles de predictores para acelerar la velocidad, pero para almacenar las direcciones de salto en lugar de las predicciones del sentido de los saltos. La predicción de línea y conjunto usada por el procesador Alpha 21264 también es una estructura jerárquica de predicción [Kess99]. La verificación de la predicción de línea se hace con un ciclo de retardo, y se corrige con un ciclo de penalización, menos los fallos de predicción de vía, que requieren dos.

El método de predicción anticipada (*ahead*) propuesto por Sèznec y Fraboulet [SeFr03] permite segmentar los accesos al predictor de saltos, y aumentar la velocidad del predictor, sin reducir su precisión. Se trata de comenzar el acceso al predictor generando un índice con la información disponible en ese momento, e ir utilizando el resto de la información a medida que se va obteniendo para determinar de forma cada vez más precisa los datos que se usarán finalmente. La idea de segmentar el predictor también se puede aplicar al perceptrón, tal y como muestra Jiménez en [Jime03a]. Esta estrategia se usará ampliamente en esta tesis, y será descrita con detalle.

2.3.7 El Consumo Energético

Un factor tecnológico importante en el diseño del predictor es el consumo energético. No es sólo de interés para sistemas móviles y empotrados, sino incluso para sistemas de altas prestaciones, donde es necesario disipar el calor para asegurar la suficiente fiabilidad. En un predictor, el parámetro que más influye en el consumo es la precisión, pues una baja precisión lleva a que se pierda energía decodificando y ejecutando inútilmente instrucciones a la largo de un camino erróneo [PSZS04]. Así, puede ser preferible que el predictor contenga más memoria y consuma más energía, si así se aumenta la precisión. Manteniendo la precisión, sí que es conveniente usar la memoria del predictor de una forma energéticamente eficiente.

El uso eficiente de la energía suele ser contrario a la velocidad. Por ejemplo, una caché de acceso serie reduce el consumo al acceder primero a

las etiquetas para comprobar la vía en que están los datos, y luego acceder a la vía adecuada, en lugar de acceder a todas las vías en paralelo. Se puede aprovechar la arquitectura desacoplada del predictor para reducir el consumo energético con una caché serie, pero reduciendo el efecto de la mayor latencia de los accesos [ReCA02]. Kessler *et al.* [KJLH89] habían propuesto el acceso serie a cada una de las vías de la tabla hasta encontrar el dato buscado, y también el uso de comparaciones parciales de etiquetas.

La predicción de vía es otro método para reducir el consumo energético, que ha sido analizado por Powell et al. [PAV+01]. Aunque la predicción de vía es especialmente efectiva para la caché de instrucciones, también se ha analizado su uso para la caché de datos por parte de Calder y Grunwald [CaGr96]. Como la precisión de las predicciones de vía en una caché de datos no es muy alta, Batson y Vijaykumar [BaVi01] proponen un mejor algoritmo de emplazamiento de los datos para mejorar las prestaciones.

2.4. Conclusiones

Se han mostrado los avances en los mecanismos de predicción de instrucciones de salto y las diferentes formas de integrar la predicción dentro de la unidad de búsqueda de instrucciones. Todos los aspectos del diseño de los predictores son claves para las prestaciones de los procesadores y han sido, son, y seguramente continuarán siendo sujeto de numerosas investigaciones. El presente trabajo, cuya descripción abordaremos a continuación, pretende contribuir en cierta medida a avanzar en este camino, y aportar soluciones a los problemas que se plantean. En concreto, se analizan las propuestas más recientes y se estudian mecanismos para hacerlas más eficientes. Se pretende así lograr un mayor equilibrio en los cuatro factores básicos que afectan a su rendimiento, a saber: precisión, velocidad, anchura y consumo de energía.

3. Método Experimental. Predictor de Referencia

Resumen

Se describe el método experimental, las herramientas y las técnicas de simulación. A continuación se describe la microarquitectura del procesador base, haciendo especial énfasis en el predictor de bloques básicos, que servirá de punto de partida para construir las propuestas de esta tesis, y como referencia para evaluar el potencial de rendimiento de estas propuestas.

3.1. Introducción

Para poder definir y evaluar las propuestas sobre el *predictor del flujo de control* (en breve, predictor) que se hacen en esta tesis es necesario definir con precisión tanto la microarquitectura del predictor sobre el cual se construirán estas propuestas, como su interfaz con el resto del procesador. Esta descripción minuciosa permitirá identificar con claridad los elementos que interactúan con el predictor y, a nivel interno, provocará que una gran parte de los detalles de implementación surjan a la luz y puedan ser analizados, y así corroborar que las propuestas son realizables.

Además de una descripción funcional, es necesario realizar una evaluación cuantitativa de las propuestas. La evaluación se ha realizado, como es habitual en la disciplina que nos ocupa, mediante simulación. El capítulo comienza presentando el método experimental, comentando las herramientas de simulación utilizadas, mencionando los programas de evaluación y definiendo las métricas empleadas.

Una parte de la evaluación se puede realizar con un modelo aislado del predictor. Pero para medir el rendimiento final –el número de instrucciones ejecutadas por ciclo– es necesario modelar el procesador completo. La elección de este modelo es importante, pues limita la mejora del rendimiento alcanzable por las nuevas propuestas. Tomando como punto de partida la microarquitectura de los procesadores comerciales de altas prestaciones del momento, se ha simulado un procesador un poco más agresivo para intentar reflejar las posibles mejoras incluidas en un futuro más o menos inmediato.

La sección 3.3 describe los parámetros utilizados para modelar el sistema de memoria y el núcleo de ejecución. La sección 3.4 detalla la unidad de búsqueda de instrucciones, que tiene una organización desacoplada.

La sección 3.5 presenta de forma general la microarquitectura interna del predictor de referencia. Se trata de un predictor de bloques básicos (*Basic*

Block Predictor, BBP). El resto de secciones analizan de forma muy minuciosa esta microarquitectura. Muchos de los detalles presentados aquí serán utilizados en los capítulos posteriores para describir las propuestas realizadas en esta tesis. Algunas de las decisiones tomadas para determinar el diseño del predictor de referencia están argumentadas en base a resultados obtenidos experimentalmente por medio de la simulación.

3.2. Método Experimental y Herramientas

El método experimental comúnmente aceptado consiste en definir con detalle un modelo de la microarquitectura del procesador, para a continuación realizar y validar la implementación S/W de un simulador de este modelo. Se debe, además, seleccionar un conjunto de programas de evaluación para ser ejecutados sobre éste modelo simulado. La definición de las métricas de rendimiento y del rango de parámetros que se considerarán en el espacio de diseño del procesador, es el paso previo a la toma de medidas durante la simulación y al posterior análisis y presentación de resultados, que permitirán cuantificar el comportamiento de las propuestas.

3.2.1 Herramientas de Simulación de la Microarquitectura

El punto de partida para simular las propuestas de esta tesis ha sido *SimpleScalar* 3.0 [AuLE02][BuAu97]. Las razones básicas para escogerlo han sido su flexibilidad, su accesibilidad, su popularidad, y su capacidad para simular la ejecución especulativa. *SimpleScalar* consiste en un conjunto de herramientas de simulación, constituido por diferentes módulos —intérprete de instrucciones, predicción de saltos, memoria caché, captura de estadísticas— que se pueden usar para construir un simulador a medida. El núcleo interno del simulador define una máquina virtual capaz de emular la ejecución de un programa objeto, instrucción a instrucción, a nivel de la

arquitectura del repertorio de instrucciones. Esta característica, en contraposición a la simulación de trazas de la ejecución del programa, es la que permite simular correctamente la ejecución especulativa a lo largo de una rama incorrecta.

Existen implementaciones para varios repertorios de instrucciones. Nuestra experimentación ha sido realizada usando el repertorio Alpha EV6 [Kess99]. Es necesario que el propio fichero objeto incluya el código de las librerías que se invocan dentro del programa (no se permite la inclusión dinámica de código), y por lo tanto las funciones dentro de librerías sí que se simulan. En cambio, no se modela el comportamiento del código del sistema operativo, ni los cambios de contexto, sino que las llamadas al sistema operativo se emulan invocando directamente al sistema operativo sobre el cual se ejecuta el simulador.

Partiendo de la versión más compleja proporcionada con *SimpleScalar* (sim-outorder), se realizaron cambios importantes para modelar por separado el buffer de reordenación y la cola de instrucciones, para modelar con mayor detalle la ocupación de los buses, el ancho de banda y la segmentación del sistema de memoria, y para modelar la arquitectura desacoplada de la unidad de búsqueda y el mecanismo de prebúsqueda de instrucciones. Parte de estas modificaciones fueron utilizadas para implementar *KScalar*, una herramienta docente disponible en la red [MoRL02a].

La versión final de la herramienta permite simular los elementos del procesador de forma independiente, a distinto nivel de detalle. Es posible, por ejemplo, simular con todo detalle el predictor de saltos y la caché de instrucciones, definiendo el número total de etapas de segmentación, pero utilizar un modelo ideal para la ejecución de instrucciones y los accesos a los datos. Aunque, en este caso, no se consideren ni las dependencias entre instrucciones ni los retardos debidos a los accesos a memoria, sí que se modela la predicción especulativa y el acceso especulativo a las instrucciones. Esta configuración aumenta más de un orden de magnitud la

velocidad de las simulaciones. Es especialmente útil para validar el modelo del predictor y para hacer una exploración rápida del espacio de diseño.

3.2.2 Herramientas de Estimación de Tiempo y Consumo

CACTI 3.0 [ShJo01] [ReJo99] es un programa que modela el tiempo de acceso, el área de chip y el consumo energético de memorias caché implementadas dentro del procesador para una tecnología de 800 nm. El programa escala los resultados para tecnologías de hasta 100 nm.

eCACTI [MaDu04] (*enhanced CACTI*) mejora el modelo de las memorias caché para proporcionar una mejor estimación de consumo energético. Este programa considera de forma automática la partición en bancos separados de los datos y etiquetas de la memoria caché, y permite seleccionar la organización que minimiza el tiempo de acceso, o el consumo energético, o el área. Con esta herramienta se pueden obtener resultados de tecnologías menores de 100 nm. En este trabajo se ha utilizado eCACTI para generar resultados para una tecnología de 70 nm.

En el trabajo presentado en [AHK+00] se muestra que la forma que tiene CACTI (y eCACTI) de escalar resultados para tecnologías de implementación más pequeñas no tiene en cuenta que el retardo de los hilos de conexión entre los transistores no se escala de forma lineal. En esta tesis se han utilizado los resultados de este trabajo para corregir los datos obtenidos con eCACTI y estimar el efecto negativo del escalado de los elementos de interconexión.

3.2.3 Métricas

Las métricas que se han escogido en la evaluación de las propuestas se comportan de forma que cuanto mayor sea su valor, mejores serán las prestaciones. El rendimiento se mide en instrucciones retiradas por ciclo de

reloj (IPC). La precisión del predictor se mide en instrucciones retiradas por fallo de predicción. La anchura del predictor se mide en instrucciones retiradas por predicción (no se incluyen las predicciones en el camino erróneo). La latencia del predictor se mide en ciclos de reloj del procesador. Los fallos en la memoria caché o en las tablas del predictor se miden en instrucciones retiradas por fallo (no se incluyen los fallos producidos en el camino erróneo).

En todos los casos anteriores, las instrucciones de no operación (NOP), generadas por el compilador para optimizar la colocación del código, no se incluyen al presentar los resultados (representan alrededor de un 15% de media el las aplicaciones utilizadas).

3.2.4 Programas de Evaluación

El uso tan extendido de los programas SPEC CPU [Henn00] en los estudios de propuestas a nivel de la microarquitectura del procesador los convierte en un referente obligado. Se han convertido en un estándar de hecho para medir el rendimiento de procesadores de propósito general. Desde 1989 ha habido varias versiones. En esta tesis se han utilizado las aplicaciones de SPEC CPU2000, y dos programas de SPEC CPU95. Los programas han sido compilados para el repertorio Alpha EV6, en un sistema operativo Digital UNIX V4.0F, con el compilador cc DEC C V5.9-008 y la opción de optimización –O4. Los programas en lenguaje Fortran se han convertido a lenguaje C con el compilador f90 Compaq Fortran V5.3-915.

La Tabla 3-1 muestra cada uno de los programas de evaluación, con los datos de entrada de forma explícita en los casos en que no quedan perfectamente definidos por la configuración de referencia. Se han ejecutado las primeras 100 mil millones de instrucciones de todos los programas (excepto en los casos en que el programa acababa antes).

Explorar con detalle el espacio de diseño de las diferentes propuestas descritas en esta tesis, con la simulación entera de todos los programas es prohibitivo en tiempo. Para no renunciar a un análisis detallado del espacio de diseño se ha optado por la técnica de simular únicamente los trozos representativos de los programas. Se trata de extraer un conjunto de ventanas de ejecución –secuencias de ejecución que comienzan en diferentes puntos del programa– cuyo comportamiento coincida con el comportamiento del programa completo. Estas ventanas se muestran en la Tabla 3-2.

Tabla 3-1 Programas de evaluación con sus parámetros de entrada y el número total de instrucciones simuladas (en millones).

Programa	Distrib.	Entrada	M. Instr.
m88ksim	CPU95	referencia	29.620
li	CPU95	referencia	5.500
bzip	INT00	graphic	100.000
crafty	INT00	referencia	100.000
eon	INT00	cook	75.970
gap	INT00	referencia	100.000
gcc	INT00	166.i	90.862
gzip	INT00	random	76.307
mcf	INT00	referencia	59.993
parser	INT00	referencia	100.000
perlbmk	INT00	splitmail	63.400
twolf	INT00	referencia	100.000
vortex	INT00	vortex1	100.000
vpr	INT00	referencia	80.531

Programa	Distrib.	Entrada	M. Instr.
ammp	FP00	referencia	100.000
applu	FP00	”	100.000
apsi	FP00	“	100.000
art	FP00	“	40.968
equake	FP00	“	100.000
facerec	FP00	“	100.000
fma3d	FP00	“	100.000
galgel	FP00	“	100.000
lucas	FP00	“	100.000
mesa	FP00	“	100.000
mgrid	FP00	“	100.000
sixtrack	FP00	“	100.000
swim	FP00	“	100.000
wupwise	FP00	“	100.000

Se han utilizado ventanas de entre 100 y 200 millones de instrucciones. En todos los casos se utiliza un fichero (*chekpoint*) para recuperar rápidamente el estado del programa y se comienza la simulación 2 millones de instrucciones antes de la ventana. Se realiza la simulación de estos 2 millones de instrucciones para que el contenido de las memorias del procesador (memoria caché, tablas de predicción, …) pueda inicializarse (*warm-up*), y se toman las medidas a partir de este punto de inicio de la ventana hasta el final.

En primer lugar se hizo una simulación completa, generando medidas de prestaciones (fallos de caché, fallos de predicción e IPC) para cada intervalo

de 1 millón de instrucciones. Para seleccionar estas ventanas se analizó la gráfica de resultados. En muchos casos, el comportamiento de la aplicación era muy repetitivo, así que era sencillo escoger una o varias ventanas que capturasen este comportamiento. En los programas donde el comportamiento era muy heterogéneo se escogieron las ventanas intentando representar los casos más frecuentes. Una vez seleccionadas las ventanas se asignó un porcentaje de peso a cada una de ellas de forma que los parámetros de prestaciones que se obtuvieran con las ventanas fueran muy similares a los que se obtenían con la simulación completa.

Tabla 3-2 Ventanas de simulación para cada programa. Se definen como ***Inicio*** (***tamaño***) ***Porcentaje***, donde ***Inicio*** y ***tamaño*** indican el número de instrucciones (en millones) hasta llegar al inicio de la ventana y el tamaño de la ventana. ***Porcentaje*** indica el peso que se le da a la simulación de cada ventana.

Programa	Ventana nº 1	Ventana nº 2	Ventana nº 3	Ventana nº 4
m88ksim	*1000 (200) 50%*	*27000 (200) 50%*		
li	*1000 (200)*			
bzip	*760 (200) 30%*	*1020 (200) 50%*	*50100 (200) 20%*	
crafty	*2650 (200) 60%*	*71760 (100) 20%*	*96980 (150) 20%*	
eon	*17930 (200) 30%*	*37520 (200) 40%*	*72240 (200) 30%*	
gap	*6800 (150) 85%*	*8200 (150) 15%*		
gcc	*2260 (200) 35%*	*5290 (200) 15%*	*60610 (200) 35%*	*79300 (200) 15%*
gzip	*300 (200) 77%*	*12000 (200) 23%*		
mcf	*5820 (200) 15%*	*25000 (200) 30%*	*29500 (200) 55%*	
parser	*6360 (200) 80%*	*16040 (200) 20%*		
perlbmk	*1700 (200) 28%*	*8600 (200) 20%*	*18500 (200) 30%*	*20000 (200) 22%*
twolf	*1650 (200)*			
vortex	*3000 (200) 32%*	*6000 (200) 68%*		
vpr	*10000 (200) 50%*	*61000 (200) 50%*		
ammp	*2900 (200) 30%*	*4600 (200) 40%*	*5200 (200) 30%*	
applu	*1230 (200)*			
apsi	*1100 (200) 40%*	*1300 (200) 60%*		
art	*1400 (200) 15%*	*8100 (200) 85%*		
equake	*5800 (200) 30%*	*17800 (200) 70%*		
facerec	*3200 (200) 50%*	*35100 (200) 50%*		
fma3d	*10000 (200) 25%*	*21550 (200) 75%*		
galgel	*4600 (200) 45%*	*6800 (200) 45%*	*8200 (200) 10%*	
lucas	*2000 (200) 65%*	*2400 (200) 35%*		
mesa	*9700 (200)*			
mgrid	*2100 (200) 50%*	*2300 (200) 50%*		
sixtrack	*2500 (200) 5%*	*9000 (200) 95%*		
swim	*1400 (100) 30%*	*1500 (100) 35%*	*1600 (100) 35%*	
wupwise	*3700 (150) 45%*	*3900 (150) 45%*	*5300 (150) 10%*	

Posteriormente a esta selección de ventanas se utilizó *SimPoint* [ShSC03], un mecanismo automático de selección de ventanas, para comparar los resultados, y se encontró que los resultados diferían muy poco.

Cabe reseñar que las ventanas de simulación sólo se han utilizado al simular el predictor de saltos por separado. Los resultados obtenidos con el modelo completo del procesador, que permiten obtener medidas del IPC, se han obtenido simulando el número de instrucciones indicado en la Tabla 3-1. Los resultados de precisión del predictor con estas simulaciones más completas se han comparado con los resultados utilizando las ventanas, y han diferido menos de un 20% para todos los programas, y menos de un 5% para el 80% de los programas. Es importante entender que esta diferencia inferior al 20% no indica un margen de error en los resultados, sino un margen de diferencia entre simular unas ventanas de ejecución u otras.

3.2.5 Método Experimental

El proceso de simulación es, en general, una tarea muy compleja y que se ha de abordar con una metodología apropiada y rigurosa. Se trata de un proceso cíclico que involucra diseño, programación, verificación de funcionalidad, depuración de errores, ejecución de la simulación, recolección y formateo de resultados y análisis posterior. Este proceso se puede resumir en las siguientes fases, que se repiten de forma iterativa en caso de errores o de encontrar problemas con las prestaciones obtenidas:

1. Implementar modificación en el modelo de una parte del procesador.
2. Ejecutar paso a paso un modelo aislado de esa parte del procesador con pequeños fragmentos de alguna aplicación, para identificar errores de funcionalidad o casos relevantes para el rendimiento del sistema. El análisis a menudo saca a la luz detalles no previstos que ayudan no sólo a corregir errores de funcionamiento, sino a mejorar el modelo para aumentar sus prestaciones.

3. Simular el modelo de procesador con diferentes parámetros y sobre diferentes ventanas de programas. El proceso se hace bajo la gestión de *Condor* [LLM88], capaz de aprovechar de forma oportunística los momentos en que ordenadores de uso no exclusivo quedan ociosos, proporcionando una muy elevada potencia de cómputo.

4. Recolectar resultados en formato de texto, importarlos en una hoja de cálculo, extraer la información relevante en forma gráfica, y analizarla.

Durante el estudio experimental no se utiliza siempre el mismo modelo de procesador. Es más productivo utilizar un modelo muy simple y rápido al principio, para analizar rápidamente las diferentes posibilidades de diseño, e ir añadiendo detalles al modelo a medida que se van restringiendo estas posibilidades. En concreto, se pueden considerar tres tipos de modelos:

1. Predictor de saltos aislado con actualización inmediata de las predicciones. No se simula el comportamiento especulativo. Permite simular usando la traza de ejecución y obtener resultados de precisión, de anchura de las predicciones, y de fallos en las tablas del predictor.

2. Predictor de saltos integrado en un esquema segmentado con un número de etapas parametrizable. El sistema de memoria es ideal y la ejecución de instrucciones es ideal. Las diferentes fases del predictor -predicción, detección y corrección de errores, y actualización- se realizan en momentos diferentes, y se modela la predicción y corrección de errores en ramas erróneas de la ejecución. Permite comprobar el efecto de la actualización retardada del predictor.

3. Procesador completo. Se modelan los dos elementos más significativos en cuanto al rendimiento: (1) en la unidad de ejecución se consideran las latencias de las instrucciones, las dependencias de datos, y la anchura de ejecución; (2) en la jerarquía de memoria se consideran los fallos en los diferentes niveles de memoria caché, y la latencia y el ancho de banda de las memorias y los buses. Permite simular el rendimiento del procesador.

3.3. Microarquitectura del Procesador

La elección de la *microarquitectura* de referencia es importante, pues, por un lado, determina los detalles que comporta la implementación de nuevas propuestas y, por otro lado, limita la mejora del rendimiento alcanzable por estas propuestas. En las secciones posteriores se describirá el modelo de la unidad de búsqueda. En esta sección se presentará el modelo de los otros dos elementos del procesador: la memoria y el núcleo de de ejecución.

3.3.1 El Sistema de Memoria

La Tabla 3-3 muestra los parámetros que definen la microarquitectura del sistema de memoria, que se organiza en forma de una jerarquía de tres niveles. El modelo del procesador incluye técnicas habituales para evitar o reducir la penalización de las largas latencias a las memorias más profundas dentro de la jerarquía.

Tabla 3-3 Parámetros que definen la organización de la memoria

Capacidad:		
iL1-Cache: (8KB) 128 cjtos x 4 vías x 16 Bytes		**L2-Cache**: (512KB) 1K cjtos. x 8 vías x 64 Bytes
dL1-Cache: (16KB) 256 cjtos x 4 vías x 16 Bytes		**L3-Cache**: (8 MB) 8K cjtos x 8 vías x 128 Bytes
Ancho de Banda:		
dL1-cache : *K*/2 accesos por ciclo (*K* = anchura ejecución)	**L2 cache :** 1 acceso / ciclo (Memoria on-chip)	**Memorias off-chip**. **L3 cache :** 12,8 GB/s **Memoria**: 6,4 GB/s
Desambigüamiento perfecto de las direcciones de memoria (*memory disambiguation*) siempre se conoce la instrucción de tipo *store* de la que depende cada instrucción de tipo *load*		
Realimentación de las instrucciones tipo store a las instrucciones tipo Load (*store forwarding***)** para cualquier tamaño de datos		
Fallos de cache pendientes de ser resueltos: hasta 16 en L1 y hasta 32 en L2 y L3		
Prebúsqueda automática en la cache: las cuatro líneas siguientes en L2 y en L3		

Se modela un mecanismo perfecto para detectar las dependencias en memoria. Se realiza una realimentación directa entre las instrucciones de escritura en memoria pendientes de ser resueltas y las instrucciones de lectura de memoria que necesitan el dato. El sistema permite gestionar en

paralelo múltiples peticiones entre las memorias de la jerarquía. Por último, se modela un sencillo mecanismo de prebúsqueda de líneas de caché, en el que cada vez que se accede a una línea en los niveles L2 y L3 de la jerarquía, se comprueba que las 4 líneas siguientes también estén presentes en la caché. Si no es así, se inician las peticiones al siguiente nivel, pero dando siempre prioridad a las peticiones "normales" frente a las peticiones de prebúsqueda.

3.3.2 El Núcleo de Ejecución

La Tabla 3-4 muestra los parámetros que se han considerado para el núcleo de ejecución. Se ha utilizado una anchura de ejecución de 6 instrucciones por ciclo, que representa una posible proyección de futuro respecto a los procesadores superescalares actuales (aunque sí que es típica de algunos procesadores con paralelismo explícito a nivel de instrucción –VLIW o EPIC-). El tamaño de la ventana de instrucciones y de las colas de instrucciones se ha escogido de forma proporcional a la anchura del procesador. Se ha verificado previamente que tamaños más grandes de estas colas producen una ganancia marginal con el modelo de memoria utilizado.

Tabla 3-4 Parámetros del núcleo de ejecución que definen la anchura de cómputo y la capacidad de la ventana de ejecución.

Anchura del Procesador	Ventana de Ejecución
Decodificación: 6 instr. /ciclo	**Buffer de Reordenación**: 240 entradas **Cola Instrucciones**: 120 entradas
Ejecución: 6 instr. / ciclo	**Cola de Instr. de Lectura a Memoria**: 120 entradas
Retirar: 8 instr. / ciclo	**Cola de Instr. de Escritura en Memoria:** 90 entradas

Para determinar la profundidad de segmentación del procesador, y las latencias de los accesos a memoria y de las operaciones de cómputo hemos usado como referencia un procesador comercial Intel Pentium 4 [HSU+01], bastante agresivo en cuanto a la frecuencia. La Tabla 3-5 muestra los valores de estas latencias.

Tabla 3-5 Latencias básicas en el procesador: operaciones de cómputo, accesos en el sistema de memoria, y latencias para atravesar las etapas del procesador.

Latencias de las Operaciones	Latencias de la Memoria
CÓMPUTO ENTERO **Sumar/comparar: 1 ciclo** **Desplazar: 2 ciclos** **Multiplicar: 5|10 ciclos** **Dividir: 30|60 ciclos**	**L1-Cache:** 2 ciclos **L2-Cache**: + 6 ciclos **L3-Cache**: + 32 ciclos **Memoria**: + 120 ciclos
CÓMPUTO Punto Flotante **Sumar/comparar: 3 ciclos** **Multiplicar: 6 ciclos** **Dividir: 16|32 ciclos**	**Latencias mínimas en el Pipeline** **Penalización Fallo de Decodificación:** 4 ciclos **Penalización Fallo de Predicción:** 16 ciclos **Tiempo desde Predicción a Retirar:** 20 ciclos

Tal como se propone para el Pentium 4, las operaciones de suma y resta de números enteros están segmentadas en dos medios ciclos, de modo que se pueden ejecutar dos operaciones dependientes por ciclo (una instrucción que depende de una instrucción de suma entera se puede lanzar a ejecutar medio ciclo después de haber comenzado la ejecución de la suma). El resto de latencias de las unidades funcionales se definen respecto a esta latencia base en función de la complejidad relativa de cada operación.

3.4. Arquitectura Desacoplada de la Unidad de Búsqueda

La microaquitectura de la unidad de búsqueda de instrucciones que se utiliza en esta tesis desacopla el predictor de la memoria caché de instrucciones. La propuesta original de Reinman et al. aparece en [ReAC99]. En [ReCA01] y [ReCA02] se analizan varias optimizaciones posibilitadas por esta organización, entre ellas la prebúsqueda de instrucciones dirigida por el predictor de saltos y la reducción del consumo energético con una muy pequeña reducción de prestaciones mediante una caché de acceso serie.

La Figura 3.1 muestra el diagrama de bloques de la organización desacoplada básica. El predictor genera referencias a bloques básicos, que consisten en la dirección inicial y el tamaño del bloque básico. Acceder a las instrucciones de un bloque básico puede suponer múltiples accesos a la memoria caché de instrucciones, bien porque el bloque básico contiene más

instrucciones que un bloque de memoria caché o bien porque las instrucciones están desalineadas respecto al bloque de memoria. Si la referencia al bloque básico no se consume en un acceso a la memoria caché, entonces se guarda en una cola de predicciones, denominada *FTQ* (*Fetch Target Queue*) en [ReAC99]. A partir de ese momento, las predicciones se almacenan en la cola de predicciones, y la memoria caché consume predicciones de la cola. Una vez usada la predicción se almacena en una cola para permitir la recuperación en caso de un error, y para realizar la actualización del predictor una vez que las instrucciones del bloque básico predicho se hayan retirado. El tamaño de la cola de predicciones determina cuánto se puede distanciar el predictor de la etapa de búsqueda de instrucciones. El tamaño de la cola de recuperación y actualización determina el número de bloques básicos que pueden estar "en vuelo" (*in-flight*), es decir, que pueden estar en el procesador sin haber sido aún retirados.

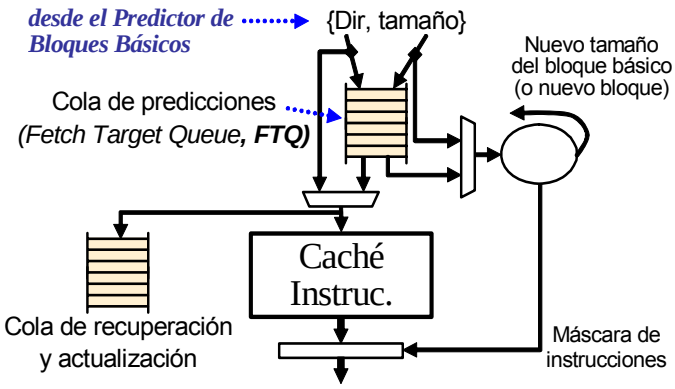


Figura 3.1. Unidad de Búsqueda de instrucciones con una organización desacoplada.

Una de las grandes ventajas del diseño desacoplado es que se independizan las latencias del predictor y de la caché de instrucciones. Si la caché tiene una latencia superior a la del predictor, se puede segmentar el acceso a la caché para permitir así que la unidad de búsqueda funcione a la misma frecuencia del predictor. La cola de predicciones (*FTQ*) posibilita incluso que el predictor funcione a una frecuencia diferente a la de la

frecuencia de la memoria caché y del resto del procesador. Un esquema en el que la latencia de caché sea superior a la del predictor implica que la información sobre el tipo de las instrucciones y sobre las direcciones de salto no se puede obtener de la memoria de instrucciones, sino que se debe almacenar en tablas asociadas al predictor.

Otra ventaja importante del esquema desacoplado es tolerar retardos puntuales en el predictor, por ejemplo, debidos a que una predicción lenta corrija a una predicción rápida, o debidos a que cierta información del predictor se haya tenido que ir a buscar a una memoria más lenta.

3.4.1 Prebúsqueda de instrucciones guiada por el Predictor

Los fallos en la memoria caché frenan el consumo de información de la cola de predicciones, pero el predictor puede seguir avanzando e ir llenando la cola. Esto permitiría parar el predictor y reducir su consumo energético. Sin embargo, una mejor solución para aumentar las prestaciones y reducir el consumo energético es la propuesta de [ReCA02], que denomina *Serial Prefetch Architecture*. Este esquema se muestra en la siguiente figura.

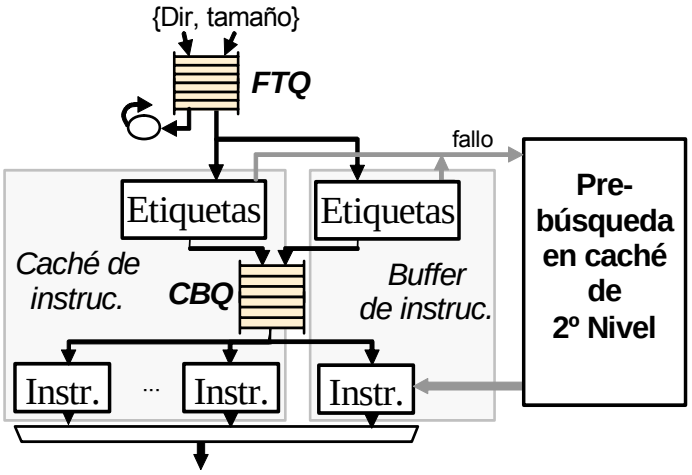


Figura 3.2. Organización desacoplada con prebúsqueda de instrucciones dirigida por el predictor y con acceso serie a la caché de instrucciones.

La memoria de instrucciones de primer nivel se divide lógicamente en dos partes: una memoria caché típica (*iCache*) y un buffer de prebúsqueda de instrucciones (*Prefetch Buffer*, *PB*), que comentaremos en breve. Ambas memorias están organizadas de forma asociativa por conjuntos, y almacenan las etiquetas y las instrucciones en bancos de memoria separados. Los accesos a las instrucciones se serializan. En primer lugar se accede a las etiquetas de *iCache* y *PB* para probar si las instrucciones se encuentran en la memoria de primer nivel y en dónde se encuentran. A continuación se accede a las instrucciones en el banco adecuado, evitando tener que leer varias vías simultáneamente y reduciendo el consumo energético.

Separar etiquetas y datos reduce el consumo energético, pero un cambio adicional permite aumentar considerablemente las prestaciones de la unidad de búsqueda. Se trata de aumentar el número de puertos de los bancos de etiquetas (al menos dos) para permitir que las comparaciones de etiquetas se avancen considerablemente a los accesos a las instrucciones. De este modo, con la ayuda de un predictor con suficiente ancho de banda, se pueden anticipar los fallos en la memoria de primer nivel y acelerar la resolución en el segundo nivel.

Una nueva cola, denominada *Cache Block Queue (CBQ*), almacena índices a bloques pendientes de ser leídos de los bancos de instrucciones. Cuando se produce un fallo, se envía la petición a la memoria caché de 2º nivel y el bloque de instrucciones que se recibe se guarda en *PB*. De este modo se evita polucionar *iCache* con instrucciones que pueden formar parte del camino erróneo. Las instrucciones se moverán de *PB* a *iCache* sólo después de ser utilizadas. El manejo de *iCache* y *PB* requiere de complejos mecanismos que aseguren consistencia, que se pueden integrar con la política de reemplazo para mejorar las prestaciones de la memoria caché, tal y como se explica con detalle en [ReCA02]. En este trabajo se ha simulado un modelo aproximado que una vez que realiza la comprobación de la etiqueta considera que las instrucciones siempre estarán en el banco de instrucciones.

3.5. Descripción General del Predictor de Referencia

La Figura 3.3 muestra el diagrama de bloques funcional del predictor que se usará como referencia. Se trata de un predictor de bloques básicos (*Basic Block Predictor*, *BBP*), es decir de secuencias de instrucciones con una única instrucción de transferencia de control (ITCs) al final. Primero describiremos su funcionamiento para saltos condicionales, y después retomaremos el resto de casos.

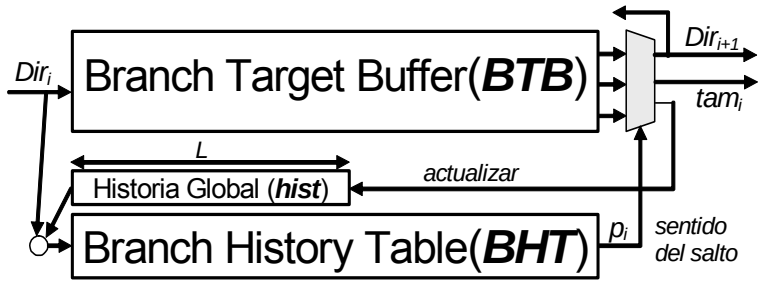


Figura 3.3. Diagrama de bloques funcional de un predictor basado en historia global.

El predictor almacena y utiliza la historia global de los saltos para correlacionar su comportamiento pasado con su comportamiento futuro. La historia global (***hist***) consiste en la codificación del sentido de los *L* saltos condicionales anteriores (0 indica salto no tomado y 1 indica salto tomado). Cuanto más larga sea la historia (cuanto mayor sea el valor de *L*) mayor será la probabilidad de encontrar correlación con un salto previo que permita aumentar la precisión en la predicción del flujo de control. Sin embargo, usar una historia mayor supone un mayor número de casos posibles a considerar y requiere almacenar más datos en el predictor.

Tal como fue propuesto en [CaGr94a], el uso de la memoria del predictor es más eficiente si se divide la información en dos tablas diferentes. En la primera tabla, la tabla de historia de saltos (*Branch History Table*, *BHT*), se almacena la predicción del sentido de los saltos (saltar o no saltar). La segunda tabla (*Branch Target Buffer*, *BTB*) contiene las direcciones de

memoria a donde saltar y el resto de información relativa al bloque básico a predecir. La Figura 3.4 muestra el contenido de ambas tablas. En las siguientes secciones se describirá la utilización de cada campo.

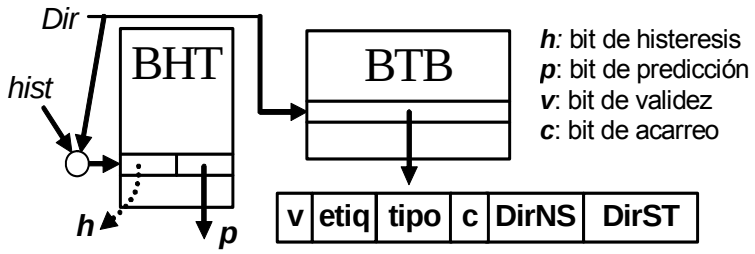


Figura 3.4. Campos de las tablas *BHT* y *BTB*.

El predictor se usa en tres fases diferentes. En la *fase de predicción* se utiliza la dirección del bloque básico actual y la historia global para predecir la dirección del siguiente bloque básico y actualizar especulativamente la historia con el sentido predicho para el salto. Ciertos datos obtenidos durante la fase de predicción se almacenan en la cola de recuperación y actualización para ser utilizados en las dos fases siguientes.

En la *fase de actualización* se utiliza la dirección del bloque básico recién retirado y la historia global de los últimos *L* saltos condicionales retirados (historia no especulativa) para actualizar, si es conveniente, el contenido de *BHT* y *BTB*. En esta fase se utiliza la información almacenada en la cola de predicciones y posteriormente, al no ser ya útil, se elimina de la cola.

Mientras las dos fases anteriores se aplican a todos los bloques básicos, la *fase de recuperación* sólo se lleva a cabo al detectar una predicción incorrecta, bien al decodificar, o bien al ejecutar una instrucción de salto. En esta fase, se recupera el valor adecuado de la historia global y se reinicia la fase de predicción con los valores de entrada corregidos. De nuevo se utiliza la información almacenada en la cola de predicciones, pero en este caso la información no se elimina de la cola sino que se actualiza.

3.5.1 Predicción de Saltos Condicionales

Cada entrada de *BHT* almacena una predicción de salto para una pareja de valores {*Dir*, *hist*} (ver Figura 3.4). Cada entrada lógica contiene dos bits, el bit de predicción (*p*) indica el sentido predicho para el salto, y el bit de histéresis (*h*) se usa para decidir cómo actualizar el bit de predicción. Puesto que el bit de histéresis sólo se utiliza en la fase de actualización, en una implementación real los bits de predicción y de histéresis se suelen almacenar en tablas separadas [SFKS02].

BHT es de acceso directo y no utiliza una etiqueta (*tag*) para verificar que una entrada corresponda exactamente a una determinada pareja {*Dir*, *hist*}. Así el mecanismo de selección de datos es muy simple y rápido. Sin embargo, no se pueden detectar los casos de *error de alias*, es decir, que dos parejas de entrada se correspondan con el mismo índice en la tabla y sobrescriban la información. Para reducir el efecto de estos casos, se propone usar como mecanismo base de indexación un esquema similar a *gshare*. En un capítulo posterior se presentará un esquema de indexación más elaborado.

El objetivo de un esquema de indexación para *BHT* es hacer corresponder cada posible pareja de datos de entrada {*Dir*, *hist*} con ***n*** bits, de forma que los datos de entrada se distribuyan de la forma más uniforme que sea posible entre los 2^{n} posibles valores de salida. Al aprovechar al máximo el espacio disponible en *BHT* es posible usar una historia global más larga para así aumentar la precisión de las predicciones.

La Figura 3.5.a muestra la función ***xor-fold-n*** utilizada para “doblar” un valor ***v*** de más de ***n*** bits en un valor de ***n*** bits, usando la operación “o-exclusivo” y basada en el esquema propuesto en [JBSS97] y [JaRS97]. La Figura 3.5.b muestra la forma en que se generan los índices de tamaño *I* bits para *BHT*. La dirección del bloque básico y la historia global se convierten usando la función ***xor-fold-n*** en varias secuencias de bits cuyo tamaño es un número primo (7, 13 ó 19), que se juntan en una larga cadena de 59 bits, que

a su vez se vuelve a convertir usando la función ***xor-fold-n*** en el índice final de *I* bits. De este modo se distribuyen los bits de forma más uniforme.

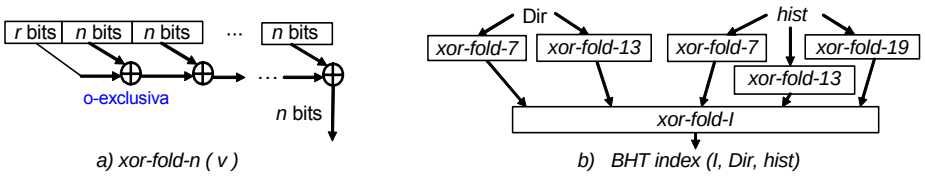


Figura 3.5. Esquema de generación de índices usando la operación "o-exclusivo".

La función final ha sido obtenida tras una serie de pruebas con varias alternativas. Se ha escogido la función que daba una mejor precisión final en las predicciones para diferentes tamaños de *BHT* y para todos los programas considerados. Esta función no pretende ser una propuesta de implementación final, sino que se trata de una alternativa de referencia que obtiene resultados bastante homogéneos para todas las configuraciones y aplicaciones analizadas. Un estudio en mayor profundidad debería permitir encontrar alternativas más simples y rápidas, que "mezclasen" menos bits, y que proporcionasen unos resultados similares.

El esquema de indexación propuesto requiere que previamente se calcule cuál es el tamaño adecuado de la historia global (*L*). Es decir, el algoritmo no proporciona ningún control para aproximarse dinámicamente a este tamaño. Se han realizado pruebas para estimar el tamaño óptimo de la historia global para tablas *BHT* de 16K, 64K y 256K entradas (4KBytes, 16KBytes y 64KBytes respectivamente). La siguiente figura muestra estos resultados. Como puede apreciarse, la precisión máxima para tablas de 16K, 64K y 256K entradas se obtiene con los valores *L*=20, 22 y 28, respectivamente. A partir de ahora, si no se dice lo contrario, consideraremos que para cada tamaño de *BHT* siempre se usan estos valores de *L* que proporcionan la máxima precisión.

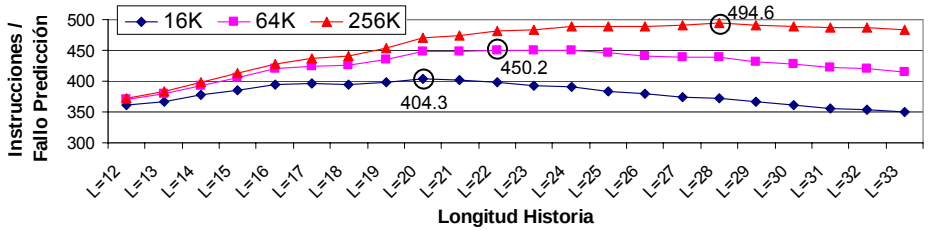


Figura 3.6. Precisión del predictor en función de la longitud de la historia (*L*= 12-33) y del tamaño de *BHT* (16K, 64K, ó 256K entradas).

3.5.2 Predicción de Direcciones de Salto

BTB proporciona la información necesaria para calcular el tamaño de un bloque básico y la dirección del siguiente bloque básico (ver Figura 3.4). La tabla es asociativa por conjuntos y se indexa usando los bits bajos de la dirección de la instrucción inicial del bloque básico. Los bits altos de la dirección constituyen la etiqueta del bloque básico y se almacenan en *BTB*. La etiqueta y el bit de validez (***v***) determinan de forma inequívoca a qué bloque básico corresponde una entrada en *BTB*, es decir, eliminan completamente el problema de los errores de *alias*.

El campo ***tipo*** codifica el tipo de instrucción de salto que finaliza el bloque básico. De momento supondremos que todos los bloques básicos finalizan en instrucciones de salto condicional. Si *BHT* predice que el salto se ha de tomar, la dirección del siguiente bloque básico se obtiene del campo ***DirST*** *(Dirección de SalTo)*. Supondremos que se usan ***w*** bits para especificar la dirección de una instrucción. Si la predicción de *BHT* es no saltar, entonces la dirección del siguiente bloque básico se puede obtener sumando el tamaño del bloque básico actual con su dirección inicial. La Figura 3.7 muestra otro procedimiento para acelerar el cálculo de esta dirección sin tener que incrementar significativamente la cantidad de bits en *BTB* (tal como fue propuesto previamente en [ReCA01] y [CaGr94a]).

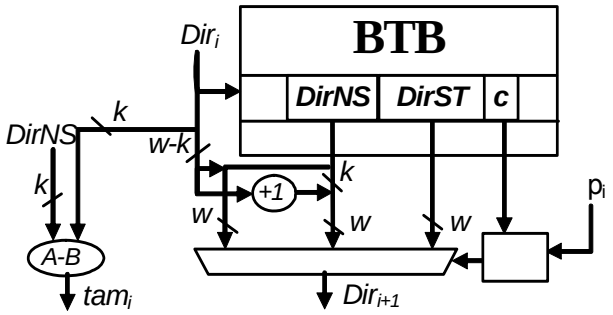


Figura 3.7. Cómputo de la dirección del bloque básico contiguo y del tamaño del bloque básico actual.

En el campo ***DirNX*** se almacenan los ***k*** bits menos significativos de la dirección del bloque básico consecutivo. El resto de bits de la dirección, los ***w-k*** bits más significativos, pueden ser o bien exactamente los mismos que los de la dirección del bloque básico actual, o bien la suma de estos bits con el valor 1. El campo ***c*** (*carry* = acarreo) indica si hay que sumar 1 o no. Cabe observar que la suma del valor 1 se hace en paralelo con el acceso a *BTB*, y por tanto no debería afectar a la ruta crítica del circuito para *BTBs* de tamaño realista. El único elemento susceptible a aumentar la ruta crítica del circuito es que el valor de ***c*** debe ser considerado al calcular cuál de las tres posibles direcciones se debe escoger en el multiplexor final. Veremos en breve que aún será necesaria al menos una cuarta entrada al multiplexor para considerar los saltos indirectos y la corrección de los fallos de predicción. La alternativa a este diseño es almacenar en *BTB* los ***w*** bits completos de la dirección del bloque contiguo. Esto supondría aumentar el tamaño total de *BTB* al menos un 50%, y el incremento correspondiente del tiempo de acceso podría fácilmente sobrepasar el retardo del multiplexor.

El valor ***k*** determina que el tamaño máximo de un bloque básico sea de 2^k instrucciones. En pruebas preliminares hemos determinado que ***k***=5, es decir, permitir bloques básicos de hasta 32 instrucciones, es un adecuado compromiso entre la anchura del predictor y el tamaño de *BTB*.

La obtención del tamaño del bloque básico (tam_i en la Figura 3.7) requiere restar al campo ***DirNS*** los ***k*** bits bajos de la dirección inicial del bloque básico. Este cómputo es necesariamente posterior al acceso a *BTB*, pero no afecta a la ruta crítica del predictor, únicamente retrasa el momento en que la etapa siguiente, encargada de buscar las instrucciones, conoce el tamaño exacto del bloque de instrucciones que se ha de buscar.

3.5.3 Saltos Incondicionales y Saltos Indirectos

El campo ***tipo*** en *BTB* permite identificar el tipo de instrucción de salto en la que acaba cada bloque básico. *BHT* sólo se usa para saltos condicionales. La predicción para los saltos incondicionales consiste simplemente en utilizar la dirección de salto almacenada en el campo ***DirST*** de *BTB*. Para predecir con mayor precisión los saltos indirectos es necesario usar dos algoritmos diferentes, uno para los saltos indirectos en general y otro más específico para las instrucciones de retorno de subrutina. Estos dos elementos se muestran en la siguiente figura.

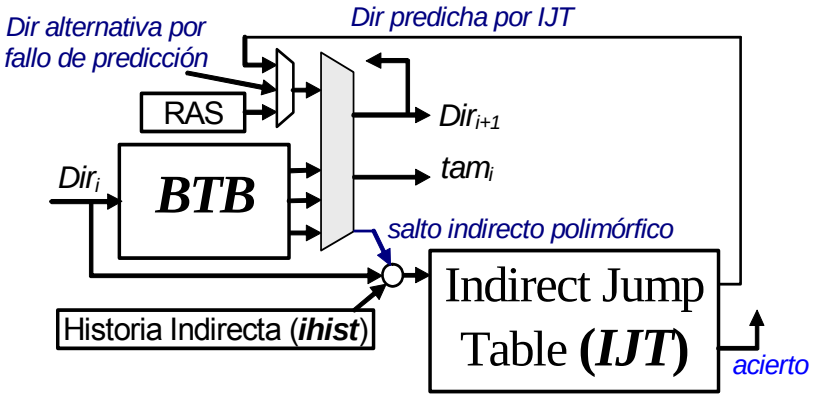


Figura 3.8. Esquema del predictor donde se incluyen mecanismos específicos para saltos indirectos polimórficos (*IJT*) e instrucciones de retorno de subrutina (*RAS*).

Al diseño básico previo se le añade una tabla específica para predecir los saltos indirectos que no son de retorno de subrutina, y que denominamos *IJT*

(*Indirect Jump Table*). El predictor utiliza la historia de los saltos previos de una forma similar a como se hace para los saltos condicionales, y es similar a la propuesta de Chang et al. en [ChHP97]. También se le añade una pila *RAS* (*return address stack*) para predecir los saltos de retorno de subrutina [KaEm91].

3.5.4 Predicción de Saltos de Retorno de Subrutina

Consideramos una pila *RAS* con espacio para almacenar las direcciones de retorno de los últimos 64 saltos a subrutina. Se incluye también la capacidad para corregir hasta 4 sobre-escrituras en la pila debido a la actualización especulativa de la pila, tal y como se propone en [SAMC98]. Se ha comprobado experimentalmente que un tamaño mayor de pila o una capacidad mayor de recuperación proporciona mejoras insignificantes.

Cuando se encuentra un bloque básico finalizado en una instrucción de salto a subrutina se inserta en la pila la dirección del bloque básico consecutivo, que será posteriormente la dirección de retorno de la subrutina. Como suponemos un repertorio de instrucciones en el que todos los saltos a subrutina son incondicionales (bien directos o indirectos), la dirección del bloque contiguo se puede almacenar en *BTB* de la misma manera que se almacena para los bloques básicos terminados en una instrucción de salto condicional (codificada en el campo ***DirNS***).

En cada iteración del predictor el contenido de la cima de la pila *RAS* se envía al multiplexor como posible dirección del bloque siguiente. Como se ve en la Figura 3.8, esta dirección “compite” en un multiplexor previo con la dirección que proviene de un fallo del predictor o con la dirección que proviene de *IJT* (discutida en breve). En caso de que alguna de estas dos direcciones sea válida, toma prioridad respecto al contenido de la pila *RAS*. Si no es así, cuando el bloque básico actual resulte ser un retorno de subrutina, se configurará el multiplexor final para que la dirección de la pila

RAS sea la que genere la dirección del siguiente bloque básico. Además, se actualizará la pila *RAS* para extraer la dirección recién usada de la cima.

Cuando un bloque básico de llamada a subrutina va seguido directamente de un bloque básico de retorno de subrutina puede ser complejo que dé tiempo a escribir la dirección de retorno en la pila y luego a leerla. En este caso se puede utilizar un sencillo circuito de realimentación para lograr el funcionamiento correcto.

3.5.5 Predicción de Saltos Indirectos

IJT se conecta en cascada con *BTB*, tal y como propone Diesen y Hölzle en [DrHo98b]. De esta forma, la predicción proporcionada por *BTB*, puede ser corregida por una predicción más específica producida por *IJT*. En el capítulo 6 se analizará una nueva propuesta para implementar la predicción de saltos indirectos y se discutirá el trabajo relacionado. En este apartado se define el modelo de predictor que se usará como referencia en aquel capítulo.

Para aumentar la eficacia del esquema en cascada se usan dos filtros que limitan el uso de *IJT*. El primero tiene como objetivo mejorar la precisión del predictor y el segundo reducir el consumo energético. Los saltos indirectos se clasifican en monomórficos y polimórficos. Los saltos *monomórficos* son aquellos que hasta el momento sólo han saltado a una única dirección destino durante la ejecución del programa. La primera vez que se decodifica un salto indirecto se clasifica como monomórfico, y la dirección a la que salta se almacena en *BTB*. El primer filtro consiste en tratar los saltos indirectos monomórficos como si fueran saltos incondicionales directos. Es decir, se predicen usando la dirección contenida en *BTB* y no se usa *IJT*. Del mismo modo, en la fase de actualización, los saltos indirectos monomórficos no realizan ninguna acción sobre *IJT*. No tiene sentido que estos saltos utilicen parte de la memoria de *IJT*, ya que *BTB* los predice de forma exacta.

En el momento en que en la fase de actualización se detecta que un salto indirecto considerado previamente como monomórfico salta a una dirección diferente a la que saltó la última vez, se le considera *polimórfico*. Es a partir de este momento cuando se usa *IJT* para este salto, tanto en la fase de predicción como en la de actualización. La predicción de *BTB* para ese salto indirecto no se modifica. De este modo, cuando la predicción de *BTB* es correcta, no es necesario actualizar *IJT*, y así se evita polucionar la tabla con información que no ayuda a mejorar la precisión del predictor. Si la información del salto indirecto desapareciera completamente de *BTB* debido a un reemplazamiento, el proceso de identificación del salto indirecto debería comenzar de nuevo.

El segundo filtro consiste en retardar el acceso a *IJT* hasta el momento en que se haya comprobado, a partir de la información leída de *BTB*, que el bloque básico a predecir finaliza en un salto indirecto polimórfico. Este esquema penaliza la latencia de predicción de los saltos indirectos a cambio de reducir el consumo energético del predictor.

En la Figura 3.9 se observa un ejemplo en el que el acceso a *IJT* se realiza en los dos ciclos posteriores a que *BTB* proporcione su predicción. Si existe una predicción en *IJT*, y no coincide con la predicción de *BTB*, se recupera el predictor usando la predicción de *IJT*, que se supone que tiene más precisión. En este ejemplo, cada vez que los dos predictores disienten, la operación del predictor se retrasa dos ciclos.

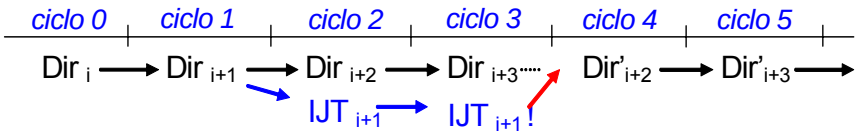


Figura 3.9. La predicción de los bloques i+2 e i+3 por parte de *BTB* se hace en paralelo con la predicción del bloque i+1 por parte de *IJT*.

La Figura 3.10 muestra un diagrama del mecanismo de indexación de *IJT* y del contenido de sus entradas. El índice se construye utilizando la dirección

del bloque básico y utilizando dos historias globales, la de los saltos condicionales previos (*hist*) y la de los saltos indirectos previos (*ihist*). La tabla es de acceso directo pero contiene una etiqueta (*tag*) de 4 bits para verificar que el contenido de la tabla corresponde con cierta probabilidad a la combinación de entrada { *Dir*, *hist, ihist* } requerida.

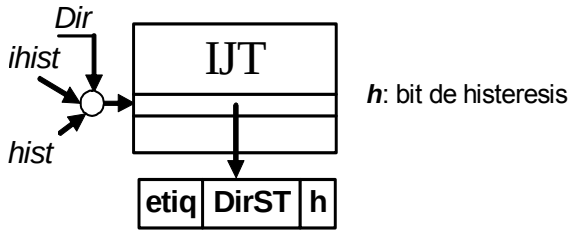


Figura 3.10. Campos de la tabla *IJT*.

El bit de histéresis se utiliza para evitar reemplazar demasiado rápido las entradas que proporcionan predicciones correctas. El bit se pone a cero cuando se asigna la entrada por primera vez. Se pone a uno en cada predicción correcta y se pone a cero en cada predicción incorrecta. Sólo se modifica la predicción si el bit se encuentra a cero. Para evitar que nuevas entradas reemplacen a entradas útiles, cuando una entrada es candidata a un reemplazamiento, sólo se permite el reemplazo si el bit de histéresis está a cero. En caso contrario se pone el bit a cero.

La Figura 3.11 ilustra cómo se genera y se actualiza la historia de saltos indirectos. Puesto que un salto indirecto puede tener múltiples destinos, su comportamiento no se puede codificar simplemente con un bit, como los saltos condicionales, sino que se deben utilizar las direcciones de salto. El resultado se denomina la historia del camino de los saltos (*path history*), en lugar de la historia del patrón de los saltos. En nuestra propuesta se combinan los bits de bajo peso de la dirección del salto indirecto y de la dirección del bloque básico a donde salta.

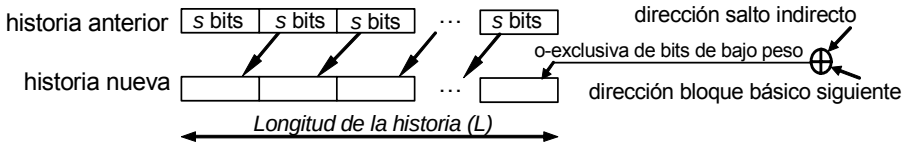


Figura 3.11. Generación y actualización de la historia de saltos indirectos (*ihist*).

Para acomodar la información de un nuevo salto indirecto se desplazan los bits de la historia *s* posiciones a la izquierda y se insertan por la derecha los *s* bits resultantes de combinar con una operación o-exclusivo los bits de menor peso de las direcciones de entrada. Tras una serie de experimentos preliminares, y aunque el valor óptimo de *s* depende del tamaño de *IJT* y de la longitud total de la historia (*L*), se ha optado por fijar el valor *s* a 3, que es el que proporciona buenos resultados de forma más homogénea.

La Figura 3.12 muestra la forma en que se generan los índices de *I* bits para *IJT*. El esquema es muy similar al usado para *BHT* e ilustrado en la Figura 3.5.b. La dirección del bloque básico (*Dir*) y las dos historias globales (*hist* e *ihist*) se convierten usando la función ***xor-fold-n*** en varias secuencias de bits, que se juntan en una larga cadena para finalmente convertirse el índice final de *I* bits.

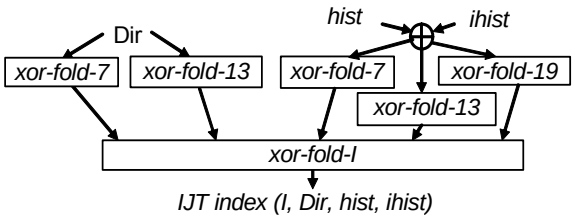


Figura 3.12. Generación del índice a *IJT*.

3.6. Predicción Anticipada de Saltos Condicionales

La función que genera el índice a *BHT* para obtener la predicción del sentido del salto, tiene la forma que se muestra en la siguiente fórmula:

$$p_i = \text{BHT} (Dir_i, hist_i)$$

El índice a BTB se genera a partir de Dir_i. La información obtenida de *BTB* junto con el valor p_i se utilizan para generar la dirección inicial del siguiente bloque básico y para actualizar la historia global de saltos condicionales (que sólo se modifica si la instrucción final del bloque básico es un salto condicional). Las siguientes fórmulas determinan las dependencias entre los valores de entrada al predictor y los valores producidos por el predictor en cada ciclo de predicción:

$$Dir_{i+1} = \boldsymbol{f_1} (\text{BTB} (Dir_i), \text{ BHT} (Dir_i, hist_i))$$

$$hist_{i+1} = \boldsymbol{f_2} (hist_i, \text{BTB} (Dir_i), \text{ BHT} (Dir_i, hist_i))$$

Tal como se puede deducir de las formulas, el retardo para generar una nueva predicción será el retardo para leer *BTB* y *BHT* (en paralelo, e incluyendo el tiempo para generar los índices correspondientes) más el retardo para ejecutar las funciones $\boldsymbol{f_1}$ y $\boldsymbol{f_2}$ (también en paralelo).

El acceso a *BHT* es potencialmente más crítico que el acceso a *BTB*. En primer lugar, el tamaño de *BHT* conviene que sea lo más grande posible, pues así permite almacenar mayor información de correlación entre saltos y aumentar la precisión del predictor. *BTB* sólo necesita ser lo suficientemente grande como para contener los bloques básicos relevantes del programa.

En segundo lugar, *BHT* suele usar un mecanismo de indexación más complejo y que utiliza más información. Por ejemplo, el diseño del procesador alpha EV8 [SFKS02] considera 4 fases en el acceso a *BHT*: selección de banco, selección de línea dentro del banco, selección de palabra dentro de la línea, y finalmente reordenación de los bits de la palabra. La última fase es utilizada para implementar la mayor parte de la complejidad del mecanismo de indexación, ya que en las tres primeras fases no hay tiempo suficiente para realizar las operaciones que requiere la indexación.

Finalmente, como cada valor leído de *BHT* es un único bit, para un tamaño en bits fijado, el número de líneas de decodificación necesarias para seleccionar el dato es mayor, lo cual supone un mayor retardo.

El método de predicción anticipada (*ahead*) permite segmentar los accesos a *BHT*, y evitar así que sea el elemento crítico, sin reducir la precisión en las predicciones [SeFr03]. La idea básica consiste en comenzar el acceso a la tabla generando un índice inicial con la información disponible en ese momento, e ir utilizando el resto de la información a medida que se va obteniendo para generar índices parciales que determinen de forma cada vez más precisa los datos que finalmente serán utilizados.

La Figura 3.13 muestra un esquema sencillo en el que *BHT* se accede en dos ciclos mientras que *BTB* se accede en un único ciclo. El esquema se puede generalizar a cualquier número de etapas de segmentación, aunque un poco más adelante indicaremos los problemas que ello comportaría.

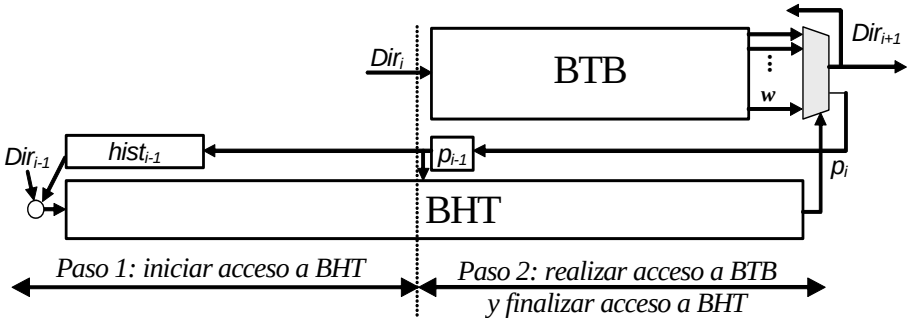


Figura 3.13. El acceso anticipado (*ahead*) a *BHT* permite reducir la latencia de las predicciones.

Para predecir el sentido del salto del bloque básico *i* (p_i) y la dirección del bloque básico *i+1* (Dir_{i+1}), en el paso 1 sólo se dispone de las predicciones de la dirección del bloque básico *i-1* (Dir_{i-1}) y del sentido de los saltos desde el bloque *i-L* hasta el bloque *i-2* ($hist_{i-1}$). Al comenzar el paso 2, después de un ciclo, ya se dispone de la predicción del sentido del salto del bloque *i-1* (p_{i-1}) y de la dirección del bloque básico *i* (Dir_i). En la figura anterior se muestra

como se “inyecta” el bit de predicción p_{i-1} en el acceso a *BHT* al inicio del paso 2. Si el bloque básico *i-1* resulta que no acaba en una instrucción de salto condicional, entonces no se utiliza este bit para actualizar la historia, y en lugar de usarlo para el acceso a *BHT* se usaría el valor p_{i-L-1}, para así mantener la misma longitud de historia en cada acceso.

Experimentalmente se ha comprobado que, tal como se indica en [SeFr03], la pérdida de precisión debida a usar el método de anticipación es, en general, bastante pequeña. La Figura 3.14 muestra la pérdida de precisión debida a la anticipación de un ciclo comparando con los valores de la Figura 3.6. Esta pérdida oscila entre el 0,3% y el 0,6% para el rango analizado de longitudes de la historia global.

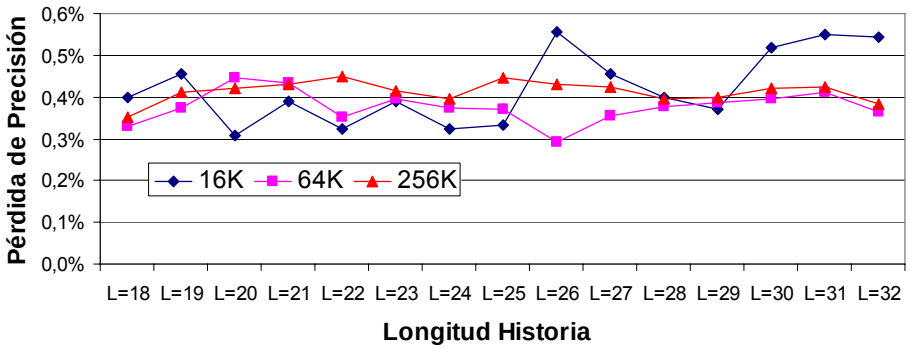


Figura 3.14. Pérdida de precisión del predictor debido al acceso anticipado de un ciclo para longitudes de la historia *L*=18-32 y *BHT* de 16K, 64K y 256K entradas

En la Figura 3.15 se muestra la pérdida de precisión al aumentar el número de ciclos de anticipación para los tres tamaños de *BHT*, y con la historia global de longitud óptima. Al aumentar el grado de anticipación, la pérdida de precisión crece ligeramente, pero no llega a sobrepasar el 1%. El argumento que justifica esta baja pérdida de precisión es que resulta muy similar usar como entradas al predictor los valores {Dir_{i-1}, $hist_{i-1}$, p_{i-1}} ó los valores {Dir_i , $hist_i$ }, pues simplemente se está substituyendo Dir_i por { Dir_{i-1}, p_{i-1} }, y es evidente que este último par de valores determina Dir_i.

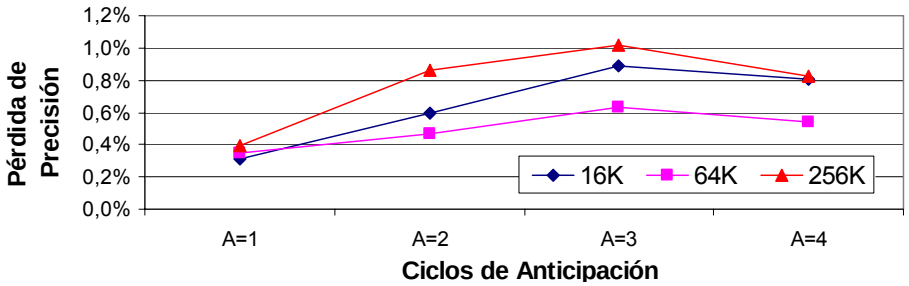


Figura 3.15. Pérdida de precisión del predictor debido al acceso anticipado a *BHT* de uno a cuatro ciclos para la longitud de historia óptima y tamaños de 16K, 64K y 256K

Aunque segmentar *BHT* permite aumentar la frecuencia del predictor, el retardo total del acceso a la tabla sí que afecta al tiempo de recuperación de un fallo de predicción. Supongamos que se erró al predecir el sentido del salto del bloque básico *i*, y que ahora se dispone del sentido correcto (p_i) y de la dirección correcta del siguiente bloque básico (Dir_{i+1}). Para calcular el sentido de salto del siguiente bloque básico (p_{i+1}) es necesario iniciar el paso 1 del predictor con los valores $hist_i$ y Dir_i, para luego en el paso 2 utilizar los valores Dir_{i+1} y p_i. Esto supone un ciclo adicional para recuperarse del fallo de predicción comparado con el caso de un predictor no segmentado.

Una forma de evitar este ciclo de penalización adicional es salvar, para cada predicción, el estado de las etapas intermedias de *BHT* en una memoria de recuperación de fallos. En el esquema de la Figura 3.13 sería necesario recuperar el estado en la mitad del acceso a *BHT*.

Una solución mucho más simple consiste en leer cada vez que se acceda a *BHT*, además de la predicción del sentido del salto, las predicciones alternativas en caso de fallo. En la Figura 3.16 se muestra como en cada acceso a *BHT* se leen dos predicciones para los valores de entrada $hist_{i-1}$ y Dir_{i-1}. La predicción del bloque anterior (p_{i-1}) se utiliza al final del paso 2 para seleccionar la nueva predicción p_i de entre las dos leídas. Tanto la predicción final como la predicción alternativa (p_i'), se almacenan en la cola de recuperación y actualización, para poder así recuperar un ciclo antes el funcionamiento del predictor en caso de un error posterior.

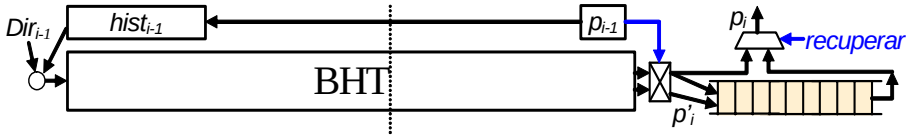


Figura 3.16. Al leer la predicción "normal" (p_i) se lee también la predicción alternativa (p_i') y se almacena en una cola, para acelerar el caso en que la predicción p_{i-1} fuese errónea. Para recuperar el predictor se usaría la predicción alternativa de la cola.

La segmentación en dos o tres etapas coincide de forma bastante natural con las etapas del acceso a la tabla (selección de línea, palabra y bit). Aumentar el número de etapas más allá de cuatro requeriría un diseño más complejo para el acceso segmentado, pero sobre todo un diseño más complejo para recuperar el predictor sin ningún ciclo adicional. Por un lado, la información que hay que almacenar para recuperar el predictor se multiplica por dos por cada nueva etapa. Además, se limita la flexibilidad del diseño, que no puede usar la información de entrada para indexar la tabla de una forma arbitraria, sino que debe usar los bits que gobiernan el multiplexor final de forma pura, sin mezclar. Finalmente, la fase de recuperación de un fallo se alarga durante más etapas. Toda esta complejidad no sólo consume área del chip, sino que también se consume mayor energía para leer y almacenar toda la información alternativa, y posteriormente usarla en la fase de recuperación.

3.7. Funcionamiento Detallado de la fase de Predicción

Al comenzar la fase de predicción se dispone de la dirección de memoria del bloque de instrucciones que se quiere predecir, y de la historia global de los últimos *L* saltos condicionales, que consiste en una secuencia de bits correspondiente a las predicciones p_{i-L}, p_{i-L+1} … p_{i-1}. *BTB* proporcionará el tipo de bloque básico de que se trata, es decir, el tipo de instrucción de transferencia de control con la que finaliza, y que determinará las operaciones que deberán hacerse y cómo interpretar los datos. La Tabla 3-6 muestra una forma de codificar ***tipo*** para todos los casos considerados.

Tabla 3-6. Codificación del campo *tipo*

tipo	descripción	qué campo de *BTB* usar; qué más hacer
0 0 0	salto condicional	usar *BHT* para decidir si se usa *DirST* o *DirNS*
0 0 1	retorno de subrutina	usar dirección de *RAS*
0 1 0	salto incondicional o indirecto **monomórfico**	usar *DirST*
0 1 1	salto incondicional o indirecto **monomórfico** a subrutina	usar *DirST*; insertar *DirNS* en *RAS*
1 0 0	no usado	----------
1 0 1	no usado	----------
1 1 0	salto indirecto **polimórfico**	usar *DirST* en primera instancia; acceder a *JPT*
1 1 1	salto indirecto **polimórfico** a subrutina	usar *DirST* en primera instancia; acceder a *JPT*; insertar *DirNS* en *RAS*

Suponiendo que en el ciclo anterior no ocurrió ninguna condición extraordinaria, las entradas a cada elemento del predictor en la fase de predicción son las que se muestran en la Figura 3.17. La flecha punteada indica que el acceso a una tabla o una cierta operación sólo se llevan a cabo para un determinado tipo de instrucciones de control.

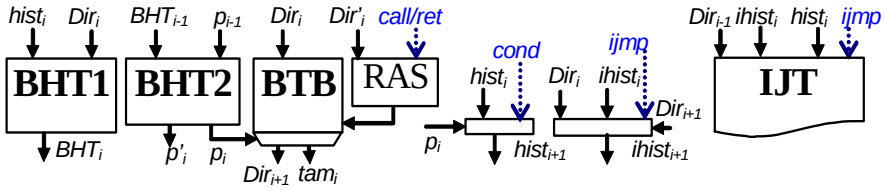


Figura 3.17. Accesos en paralelo a las diferentes tablas en cada ciclo. Se supone que el acceso a *BHT* está segmentado en dos etapas (*BHT1* y *BHT2*)

BTB produce una predicción para la dirección del siguiente bloque básico y para el tamaño del bloque actual. Si se detecta un fallo de *BTB*, entonces se utiliza una predicción por defecto que consiste en suponer que el bloque tiene un tamaño de 2^K instrucciones (el máximo posible) y que no contiene ninguna instrucción de salto. En la etapa de decodificación se descubrirá el posible error, se realizará una nueva predicción para el bloque básico, y se recuperará el ciclo de predicción. En la fase de actualización se construirá el bloque básico y se insertará en *BTB*.

En la Figura 3.17 se supone que *BHT* se segmenta en dos etapas (*BHT1* y *BHT2*). El acceso a la etapa *BHT1* produce información intermedia para la etapa *BHT2*, que se representa en la figura como PHT_i. *BHT2* siempre produce las predicciones p_i y p_i' Si en *BTB* se determina que el bloque básico actual finaliza en un salto condicional, la predicción p_i sirve para seleccionar la dirección del bloque siguiente y para actualizar especulativamente la historia global. El funcionamiento de la fase de predicción con una tabla *BHT* segmentada en más etapas se puede extrapolar fácilmente.

La pila *RAS* se modificará especulativamente si en el ciclo anterior se encontró una instrucción de salto a subrutina o de retorno de subrutina.

Si en el ciclo anterior se encontró una instrucción de salto indirecto, entonces en el ciclo actual se actualiza la historia global de saltos indirectos. Si además el salto indirecto estaba catalogado como polimórfico, entonces también se inicia el acceso a *IJT* usando la historia global antes de actualizar ($ihist_i$). Si finalmente la predicción de *IJT* resulta ser la misma que la predicción hecha por *BTB* anteriormente, entonces se ignora el resultado de *IJT*. En caso contrario, se produce una condición de *fallo* que se analizará en el apartado siguiente.

3.8. Funcionamiento Detallado de la fase de Recuperación

Para poderse recuperar de forma rápida de los fallos de predicción, y sobre todo recuperar el estado correcto, para evitar que un fallo pueda reducir la precisión del predictor, es necesario almacenar información en la fase de predicción, que se almacena en forma de cola y se va eliminando a medida que se retiran los saltos, [JSHP97]. Supongamos que se erró al realizar la predicción del bloque básico *i*. La instrucción de salto que finaliza el bloque *i*, y que ha dado lugar al error acarrea algún tipo de identificador que permite encontrar la posición de la cola de predicciones en donde encontrar la información para recuperarse del error. La información consiste en:

- la dirección del bloque básico (Dir_i)
- las historias de las que se disponía al predecir el bloque *i* (o mejor, la forma de obtenerlas) $hist_i$ e $ihist_i$
- el estado de la pila *RAS* en el momento de predecir el bloque *i* (o mejor, la forma de recuperar ese estado)
- las predicciones de salto obtenidas de *BHT* para decidir el sentido de salto de los bloques *i* e *i+1* (p_i y p_{i+1})
- las predicciones de salto alternativas obtenidas para decidir el sentido de salto del bloque *i+1* (p_{i+1}' …)

Según la causa del error y el elemento del procesador que detecta el error, la fase de recuperación deberá usar la información disponible de una forma u otra. Si la detección del error se ha dado en el núcleo de ejecución, se dispondrá de información adicional obtenida de la ejecución del salto. En breve se describirá con detalle los casos de error posibles. Primero se mostrará un esquema general común.

En todos los casos se dispondrá de la predicción de la dirección Dir_{i+1} para reiniciar la predicción en el bloque *i+1*. En caso de que el bloque i acabe en un salto condicional, también se dispondrá de la predicción del sentido de este salto, p_i. Se recuperarán las dos historias globales y la pila RAS.

Primero se presentará un análisis suponiendo que *BHT* no está segmentada. Luego se indicará como actuar para recuperar el predictor cuando *BHT* está segmentada.

3.8.1 Errores detectados en la etapa de Decodificación

Estos errores se producen cuando en *BTB* no se encuentra la información relativa al bloque *i* y se supone que es un bloque sin instrucción de salto al final, o cuando en *BTB* se encuentra una información errónea que es dada por válida. En ambos casos, la etapa de decodificación se encuentra una

instrucción de salto de un tipo diferente al esperado, y proporciona el tipo correcto y la dirección de salto (para saltos directos). Es necesario un rápido procesamiento previo antes de recuperar el ciclo de predicción. Supondremos que la detección del error en la etapa de decodificación se hace con suficiente antelación para dar tiempo a este procesamiento antes del final del ciclo de reloj.

Si el salto encontrado es **condicional**, se usa la predicción p_i de la cola de predicciones para decidir si se ha de usar la dirección de salto o la dirección de la siguiente instrucción como nuevo valor para Dir_{i+1}. Con p_i se actualiza también la historia global $hist_i$, obtenida de la cola de predicciones, para generar $hist_{i+1}$. El ciclo de recuperación comienza usando Dir_{i+1} como entrada a *BTB* y Dir_{i+1} combinado con $hist_{i+1}$ como entrada a *BHT*. *IJT* no se accede y la pila *RAS* no es actualizada.

Si el salto encontrado es **incondicional directo**, se usa la dirección de salto calculada en la etapa de decodificación como nuevo valor para Dir_{i+1}. $hist_i$ se convierte directamente en $hist_{i+1}$. El ciclo de recuperación se inicia igual que para los saltos condicionales. Si el salto es a subrutina, se inicia el ciclo actualizando la pila *RAS*

Si el salto encontrado es **indirecto de retorno de subrutina**, se usa la cima de la pila *RAS* recién recuperada como nuevo valor para Dir_{i+1}. $hist_i$ se convierte en $hist_{i+1}$ y el ciclo de recuperación se inicia igual que para los saltos condicionales. Además, se ha de actualizar la historia $ihist_i$ con Dir_{i+1} y la dirección del salto indirecto para generar $ihist_{i+1}$.

Si el salto encontrado es **indirecto**, se usa la dirección siguiente como predicción por defecto para Dir_{i+1} (que probablemente fallará y generará un error más tarde). $hist_i$ se convierte en $hist_{i+1}$, $ihist_i$ se actualiza con Dir_{i+1} y la dirección del salto indirecto para dar lugar a $ihist_{i+1,}$ y el ciclo de recuperación se inicia igual que para los saltos condicionales. No se inicia el acceso a *IJT* puesto que el salto inicialmente se considera monomórfico. Si el salto es a subrutina, se inicia el ciclo actualizando la pila *RAS*

3.8.2 Errores detectados en el Núcleo de Ejecución

Este tipo de errores se detectan una vez que se ha ejecutado la instrucción de salto cuya predicción es errónea. Por tanto, se dispondrá del sentido correcto del salto (p_i), y de la dirección correcta de salto (PC_{i+1}).

Si se detecta que el sentido de un salto **condicional** ha sido mal predicho, se actualiza la historia global $hist_i$, obtenida de la cola de predicciones, con el valor p_i para generar $hist_{i+1}$. El ciclo de recuperación comienza usando Dir_{i+1} como entrada *BTB* y Dir_{i+1} combinado con $hist_{i+1}$ como entrada a *BHT*. *IJT* no se accede y la pila *RAS* no es actualizada.

Si se detecta que la dirección de un salto **indirecto** ha sido mal predicha, se convierte $hist_i$ directamente en $hist_{i+1}$, la historia $ihist_i$ se actualiza a $ihist_{i+1}$ con las direcciones adecuadas, y se inicia la fase de recuperación igual que para los saltos condicionales. No se accede a *IJT* (no hace falta predicción pues el resultado conocido ya es seguro) y sólo si la instrucción es de salto o retorno de subrutina se necesita iniciar la actualización de la pila *RAS*.

En el caso de que el error se produzca por un salto indirecto catalogado como polimórfico, debido a que la predicción de *IJT* no coincide con la de *BTB*, el manejo es el mismo que el comentado en el párrafo anterior.

3.8.3 Recuperar la predicción Anticipada

Con *BHT* segmentada en ***k*** etapas, para recuperar el predictor en el bloque *i* sería necesario haber iniciado ***k***-1 ciclos antes el acceso a la primera etapa de la tabla segmentada. Para no perder estos ***k***-1 ciclos durante la fase de recuperación, es necesario utilizar las ***k***-1 predicciones de la cola de predicciones (y haberlas almacenado previamente).

Cuando el fallo se ha producido por predecir erróneamente el sentido del salto condicional del bloque i, entonces se usa la predicción para el bloque *i+1* considerada en su momento como alternativa (p'_{i+1}). Si el problema se ha

producido por predecir erróneamente la dirección de memoria donde saltar, entonces se usa la misma predicción que se usó anteriormente.

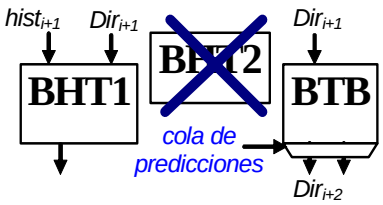


Figura 3.18. La información de la cola de predicciones se usa para sustituir a las etapas vacías de *BHT*.

3.9. Funcionamiento Detallado de la Fase de Actualización

En la fase de actualización se utiliza la información correspondiente al bloque básico recién retirado, y almacenada en la cola de predicciones, para actualizar, si es conveniente, el contenido de *BHT* y *BTB*. La historia global que se obtiene de la cola corresponde a los últimos *L* saltos condicionales retirados, y por tanto no es especulativa. Una vez realizadas las actualizaciones, la información de la cola de predicciones ya no es necesaria, y se puede sobrescribir.

Cabe observar que la actualización de las tablas de predicción se hace con cierto retraso, de forma que el predictor usa información menos reciente al tomar sus decisiones. Tal como se comentó anteriormente, el efecto que tiene este retraso en la precisión del predictor se ha demostrado que es escaso, especialmente cuando se usan tablas de predicción de tamaño relativamente grande.

La Figura 3.19 muestra las posibles operaciones de actualización de las tablas y las condiciones bajo las cuales estas actualizaciones se pueden llevar a cabo. Generalmente se realiza un primer acceso a datos no utilizados

en la fase de predicción, que sirven para determinar cómo actualizar los datos que sí se usan en la fase de predicción.

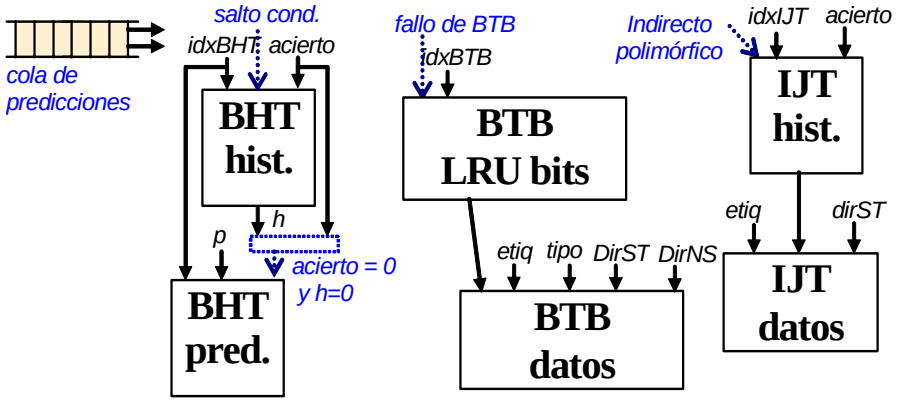


Figura 3.19. Actualización de las diferentes tablas usando la información de la cola de predicciones.

3.9.1 Actualización para Saltos Condicionales

BHT se actualiza sólo para saltos condicionales. El índice a *BHT* con el que se accedió en la fase de predicción se obtiene de la cola de recuperación. También se obtiene la información sobre si la predicción falló o no durante la ejecución. Con el índice se accede al bit de histéresis en una tabla diferente a la que contiene los bits de predicción, para así no disturbar las operaciones de predicción. En caso de que la predicción de la tabla acertara, se pone a 1 el bit de histéresis, mientras que en caso de fallo, se lee el bit de histéresis y se pone a cero.

Sólo se debe modificar el bit de predicción de *BHT* si la predicción falló y el bit de histéresis se encuentra a cero. El acceso para escribir el bit de predicción puede colisionar con una lectura de *BHT* en la fase de predicción. Se puede evitar el problema de las colisiones con una organización de dos puertos, uno de lectura y uno de escritura, o se puede limitar el problema con una organización de múltiples bancos. Como las actualizaciones son

bastante infrecuentes y la organización con dos puertos consume más área y es más lenta, la segunda opción es la más eficiente. Nuestro modelo ignora las colisiones, como si la organización fuera de dos puertos, pero asumimos un tiempo de acceso y área como una organización de múltiples bancos.

3.9.2 Actualización para Direcciones de Salto

BTB sólo se actualiza si se detectó un fallo en la fase de predicción. Tal como se comentó, si un bloque básico tiene una longitud mayor de 2^K instrucciones, se divide en trozos de 2^K instrucciones y sólo se almacena en *BTB* el último trozo, que tiene entre 1 y 2^K instrucciones y es el único que acaba en una instrucción de salto. Dejar de almacenar los primeros trozos no afecta al predictor, ya que al producirse un fallo de *BTB* se predicen correctamente.

La actualización de *BTB* requiere buscar una entrada donde emplazar la información del nuevo bloque. Tal como muestra la Figura 3.19, primero se lee la información de reemplazo, y luego se actualiza la tabla. Usamos una política de re-emplazamiento *LRU*, en la que la entrada usada menos recientemente es la que se escoge para ser sobre-escrita. Igual que en el caso de *BHT*, se pueden dar colisiones con el acceso durante la fase de predicción. Dada la baja frecuencia de los fallos de *BTB*, también ignoraremos las colisiones.

Cabe observar que entre el tiempo de predicción y el de actualización, la entrada en la tabla ha podido aparecer o desaparecer. En el primer caso, al intentar la actualización se detectará que la entrada ya existe y se anulará la actualización. En el segundo caso, se deja la tabla sin actualizar, dejando que el fallo sea detectado más adelante durante la fase de predicción.

3.9.3 Actualización para Saltos Indirectos

IJT se actualiza si el bloque básico finaliza en un salto indirecto calificado como polimórfico por *BTB*, y si la dirección del salto indirecto fue diferente a la dirección almacenada en *BTB*. De la cola de recuperación se obtiene el índice de acceso a *IJT* (acceso directo) y con él se accede al bit de histéresis, en una tabla diferente a la que contiene las direcciones. En caso de que la predicción fuese correcta, se pone a 1 el bit de histéresis, mientras que en caso de fallo, se lee el bit de histéresis y se pone a cero. Si en este último caso el bit de histéresis ya estaba a cero, entonces se modifica la predicción de *IJT*. El contenido de la etiqueta (*tag*) es ignorado en esta fase.

Igual que para *BHT* y *BTB*, se pueden dar colisiones en el acceso a *IJT* durante la fase de predicción, aunque en este caso las probabilidades de colisión son extremadamente más bajas.

3.10. Resumen

Se han presentado todos los elementos que determinan el método experimental, desde las herramientas al método de simulación, pasando por los programas de evaluación. También se ha descrito la microarquitectura del procesador que se va a utilizar como referencia en los siguientes capítulos.

Comenzando por el sistema de memoria y el núcleo de ejecución, y acabando por la organización más interna del predictor de bloques básicos (*Basic Block Predictor*, *BBP*), se han ido proporcionando detalles de forma creciente. Se han mostrado todos los elementos que influyen en el diseño de un predictor, dejando patente su elevada complejidad, y se ha establecido la base para poder argumentar las mejoras que se presentarán a continuación. Partir de un modelo tan detallado, tal como se ha dicho ya, permite aumentar la seguridad en que la mayor parte de los problemas planteados por las propuestas de esta tesis han sido considerados.

El predictor de bloques básicos, *BBP*, propuesto como punto de partida, utiliza algunas de las aportaciones realizadas durante estos últimos años en el ámbito de la arquitectura de computadores, como la organización desacoplada y la técnica de la predicción anticipada. Se trata de propuestas muy recientes, que aún no han sido adoptadas en el diseño de procesadores comerciales, pero que avanzan en el mismo camino que el trabajo de esta tesis, que es aumentar el ancho de banda del predictor. Para poder evaluar las aportaciones de este trabajo era de justicia que estas dos propuestas fueran consideradas. Para ello, ha sido necesario no sólo implementarlas en el modelo de simulación, sino también realizar múltiples experimentos para sintonizar los parámetros de su configuración.

4. Aumentando la Velocidad del Predictor

Resumen

Se proponen técnicas de predicción de vía y de predicción de índice que aceleran la velocidad del predictor de flujo de control del procesador. Estas técnicas generan retardos internos al propio predictor, pero son pequeños y poco frecuentes. El resultado final es un mayor ancho de banda de predicción, que permite mejorar de forma significativa el rendimiento del procesador.

4.1. Introducción

El aumento de la frecuencia y de la anchura de ejecución de los procesadores hace necesarios predictores a la vez muy precisos y rápidos. Este objetivo es problemático, puesto que para obtener mayor precisión se ha de almacenar más cantidad de información de la historia previa de los saltos, lo cual ralentiza el acceso a esa información. En otras palabras, el diseñador se enfrenta a la decisión de limitar la cantidad de memoria del predictor, y hacerlo más impreciso, o de disminuir su velocidad.

En el capítulo anterior se mostró que la tabla de historia de saltos (*Branch History Table*, *BHT*) se puede segmentar en varias etapas, usando la predicción anticipada, sin afectar de forma significativa a la funcionalidad del predictor, y así extraer *BHT* de la ruta crítica del predictor. De este modo, es el acceso a la tabla de direcciones de salto (*Branch Target Buffer*, *BTB*) el que determina la velocidad máxima del predictor.

En este capítulo se proponen y se analizan con detalle dos estrategias, la *predicción de vía* y la *predicción de índice*, para aumentar la velocidad del predictor de bloques básicos de referencia, *BBP*, desacoplado de la memoria caché de instrucciones.

Ambas estrategias añaden un nivel extra de predicción, que permite acceder rápidamente a los datos relevantes necesarios para predecir el siguiente bloque básico. Este primer nivel del predictor proporciona la parte crítica de la información, necesaria para iniciar una nueva predicción. Los datos restantes se obtienen más tarde. En concreto:

- se predice en cuál de las vías de una tabla asociativa por conjuntos se encuentra la información relativa al bloque de instrucciones del que se quiere predecir su sucesor. Así se evita, en muchos casos, el tiempo (y el gasto energético) de tener que leer todas las vías simultáneamente y de tener que seleccionar posteriormente los datos adecuados.

- se hace una predicción muy rápida del índice que apunta a la información del siguiente bloque básico de instrucciones, mientras en paralelo se realiza la predicción completa, más precisa y lenta.

En el diseño original de *BBP*, todas las predicciones tienen la misma latencia, el nuevo diseño "apuesta" por acelerar una gran parte de los casos de predicción. Pero si la "apuesta" falla, es necesaria una corrección que supone una latencia total mayor que la que se tenía en el diseño original. Supongamos, por ejemplo, que el nuevo diseño reduce el tiempo de predicción en la mayoría de los casos a un 75% del tiempo anterior, mientras que el resto de casos necesitan el 150% del tiempo anterior. Es necesario que la proporción de casos "rápidos" respecto a casos "lentos" sea superior al 67% para que el nuevo diseño tenga una velocidad media superior al diseño anterior. En este capítulo se demostrará que la predicción de vía y la predicción de índice capturan una proporción de casos lo suficientemente alta como para justificar la complejidad añadida al diseño.

Aplicar la técnica de predicción de vía y de índice a un predictor de bloques básicos, en lugar de a un predictor de líneas de caché como en [John91], [CaGr95] y [Yung96], simplifica la casuística y la gestión de la memoria dedicada a las predicciones de vía e índice. Mientras que una línea de caché puede tener varias instrucciones de salto, y requerir por tanto varios predictores de vía e índice, un bloque básico, por definición, contiene un único salto. Si además *BBP* está desacoplado de la memoria caché, se pueden ocultar algunos de los retardos producidos por los fallos de predicción de vía e índice. Si las predicciones se generan a un ritmo mayor del que se consumen, se almacenan temporalmente en la cola *FTQ*, proporcionando margen para tolerar los retardos producidos en los casos "lentos".

Es muy importante resaltar que los cambios propuestos apenas varían la funcionalidad del predictor. Al aumentar la velocidad media, el orden de los accesos al predictor y de las actualizaciones varía ligeramente, pero provoca cambios muy poco significativos en la precisión.

Una ventaja adicional de la predicción de vía es que al acceder a una única vía en lugar de a varias vías, se reduce el consumo energético del propio predictor. En lugar de usar la estrategia de predicción de vía para aumentar la frecuencia del predictor, se puede mantener la frecuencia que tenía el predictor original, y obtener el mismo ancho de banda de predicción, pero optimizando el diseño para reducir el consumo energético.

4.1.1 Esquema del Capítulo

El esquema de este capítulo es el siguiente. En primer lugar se estudian los factores que determinan la velocidad del predictor. Se analiza la ruta crítica y se presentan estimaciones de latencias, que sirven para motivar y presentar de forma muy general las propuestas del capítulo.

A continuación se presenta la técnica de predicción de vía en dos pasos: primero con un diseño preliminar, para simplificar su comprensión, y finalmente con un diseño exhaustivo, basado en una jerarquía de dos niveles. Tras describir con minucioso detalle la última propuesta, se presentan resultados sobre el porcentaje de aciertos de predicción de vía.

La predicción de índice se describe el la sección 4.5 como un paso adicional al diseño anterior con dos niveles. Los resultados muestran que los nuevos fallos de predicción no son excesivos. A continuación se analiza el uso de la predicción anticipada con *BTB*, y se analizan sus ventajas e inconvenientes.

Las secciones 4.7 y 4.8 muestran resultados de las simulaciones que permiten estimar la reducción de la latencia media del predictor, y el incremento en el rendimiento del procesador obtenido gracias a la mayor velocidad del predictor. Las dos últimas secciones repasan el trabajo previo que está relacionado de forma más directa con las propuestas del capítulo, y presentan las conclusiones, subrayando las contribuciones originales.

4.2. Motivación y Presentación de las Propuestas

En el capítulo anterior se describió con detalle la microarquitectura del predictor de flujo de control que se usa como referencia en este capítulo y los siguientes. Sus características principales son que está desacoplado de la búsqueda de instrucciones, que usa un mecanismo de predicción de saltos condicionales basado en la historia global, y que utiliza el bloque básico (*basic block, BB*) de instrucciones como unidad de predicción.

También se mostró, con resultados de simulación, que la tabla de historia de saltos (*BHT*) se puede segmentar en varias etapas con la técnica de predicción anticipada ("*ahead prediction*") sin afectar de forma significativa a la precisión del predictor, y sin alargar la penalización de los fallos de predicción. De este modo, podemos considerar *BHT* fuera de la ruta crítica del predictor, siendo entonces el acceso a la tabla de direcciones de salto, *BTB*, el que determina la velocidad máxima del predictor. La Figura 4.1 muestra la ruta crítica de una *BTB* organizada en 4 vías.

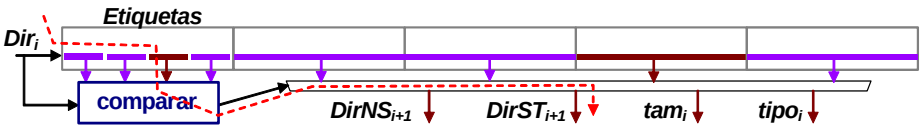


Figura 4.1. Esquema de una *BTB* asociativa por conjuntos (4 vías). La ruta crítica se muestra con una raya discontinua, e incluye la comparación de etiquetas.

Sólo se muestra el circuito para generar los datos de *BTB*, y no se muestra la lógica para convertir estos datos en una nueva predicción. El valor de entrada al circuito es la dirección del bloque básico actual (Dir_i), y la salida son los datos asociados a este bloque básico, incluyendo el tamaño del bloque, el tipo del bloque, y dos posibles direcciones para el bloque básico siguiente. Con el tipo del bloque básico actual y la predicción del sentido de salto proporcionada por *BHT*, se selecciona la dirección del siguiente bloque básico, Dir_{i+1}, entre las dos obtenidas de *BTB*, la cima de la pila *RAS*, y la

dirección obtenida tras recuperar el predictor de un error (tal vez por tener que usar el predictor específico de saltos indirectos, *IJT*).

Puesto que suponemos que el acceso a *BHT* no es crítico, pues se puede segmentar, la ruta crítica del predictor, que determina su máxima frecuencia de operación, se compondrá de tres partes: (1) generar el índice a *BTB* usando Dir_i, (2) acceder a *BTB* y leer el contenido de la vía adecuada, y (3) usar la información leída para seleccionar el valor predicho para Dir_{i+1}. El paso (1), la generación del índice a *BTB*, consiste en usar directamente los bits de bajo peso de Dir_i, y es inmediato. El paso (3) requiere combinar el tipo de bloque básico con la predicción que proviene de *BHT*. Este circuito consiste en una lógica combinacional simple combinada con un de-multiplexor final de 4 entradas (ver sección 3.5.2). La mayor parte del retardo del circuito viene determinada por el paso (2), el acceso a *BTB*.

4.2.1 Rendimiento en función de la Organización de *BTB*

La Figura 4.2 muestra medidas experimentales del número de instrucciones ejecutadas (y retiradas) por cada fallo de *BTB*, en función de su capacidad (nº de entradas) y del número de vías. Las medidas se han realizado con el modelo del predictor de saltos aislado y con actualización inmediata.

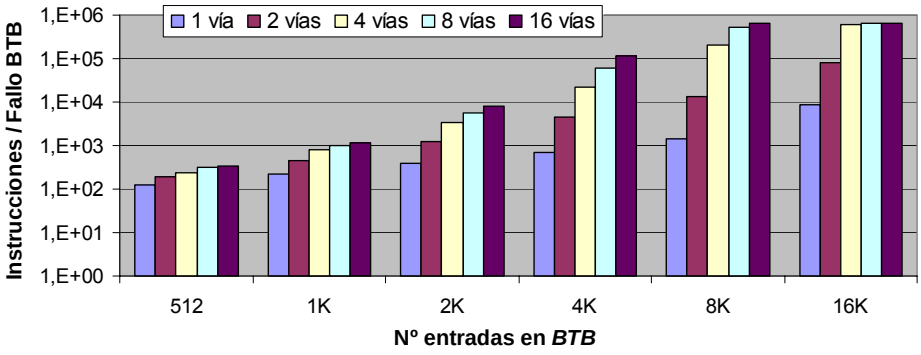


Figura 4.2. Instrucciones retiradas por cada fallo en *BTB* en función de su capacidad (entre 512 y 16K entradas) y número de vías (entre 1 y 16).

Una tabla de 8K entradas y 16 vías, o una tabla de 16K entradas y 4 vías, evitan los fallos de capacidad y de conflicto, y sólo se producen los inevitables fallos en frío. El tamaño o la asociatividad insuficientes causan fallos de *BTB* que suponen perder la información obtenida dinámicamente sobre los bloques básicos. Debido a ello, se dejan de identificar instrucciones de control en la fase de predicción, que serán detectadas posteriormente en la etapa de decodificación y provocarán un *fallo de decodificación* dentro de la propia unidad de búsqueda de instrucciones.

La Figura 4.3 muestra la cantidad de fallos de decodificación en función de la capacidad y del número de vías de *BTB*. Aunque este tipo de fallos penaliza menos ciclos que los fallos detectados en la etapa de ejecución, que analizaremos en breve, no puede despreciarse su efecto en el rendimiento del procesador.

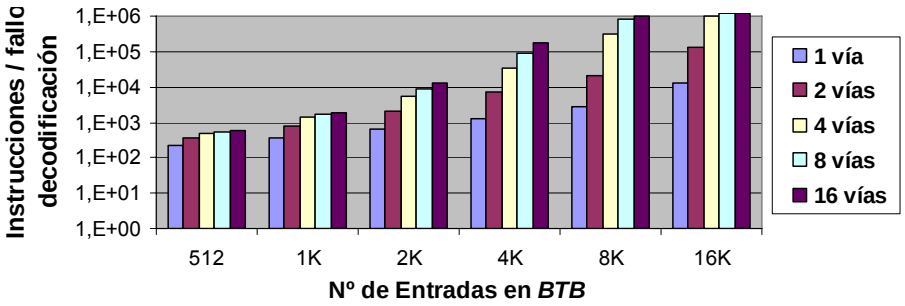


Figura 4.3. Instrucciones retiradas por cada fallo de decodificación en función de la capacidad (entre 512 y 16K entradas) y número de vías (entre 1 y 16) de *BTB*

Los fallos de *BTB* no sólo provocan fallos de decodificación, sino que modifican el uso y la actualización de la información de *BHT*, que depende de la información obtenida de *BTB*. La Figura 4.4 muestra, para tablas *BHT* de tres tamaños, el porcentaje de reducción en la precisión de los saltos condicionales debida a usar una *BTB* con una configuración realista y no de capacidad ilimitada. La Figura 4.5 muestra el porcentaje de reducción en la precisión para los saltos indirectos.

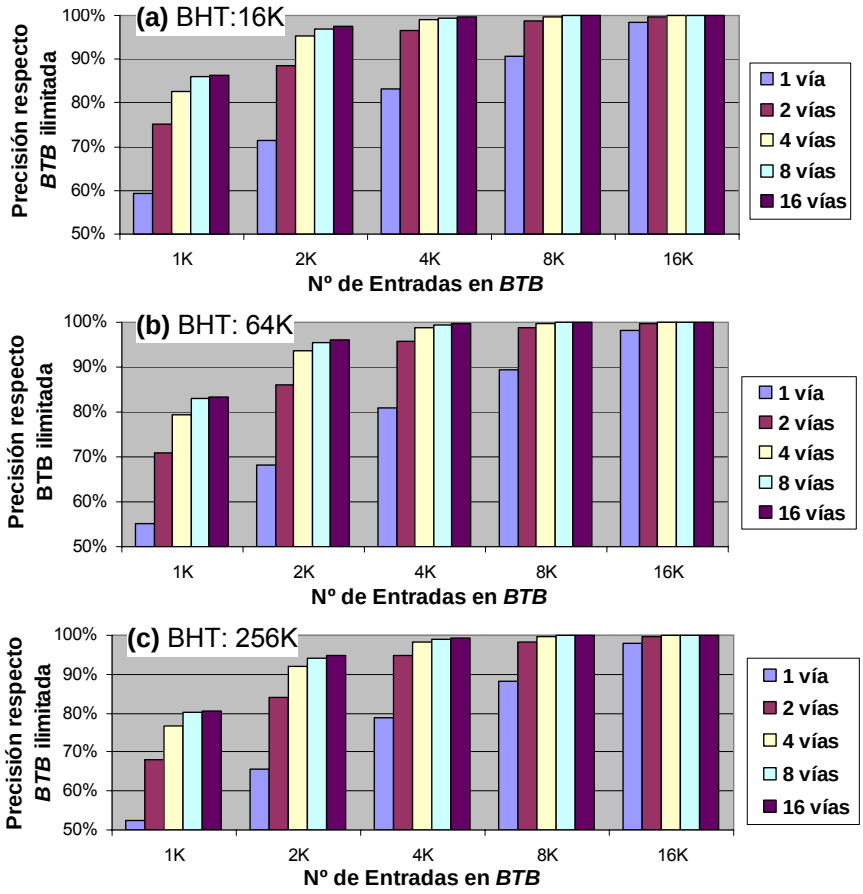


Figura 4.4. Porcentaje de la precisión del predictor para los **saltos condicionales** comparada con el uso de una *BTB* ilimitada, para diversas capacidades y nº de vías. El tamaño de *BHT* es de (a) 16K entradas (b) 64K entradas, ó (c) 256K entradas.

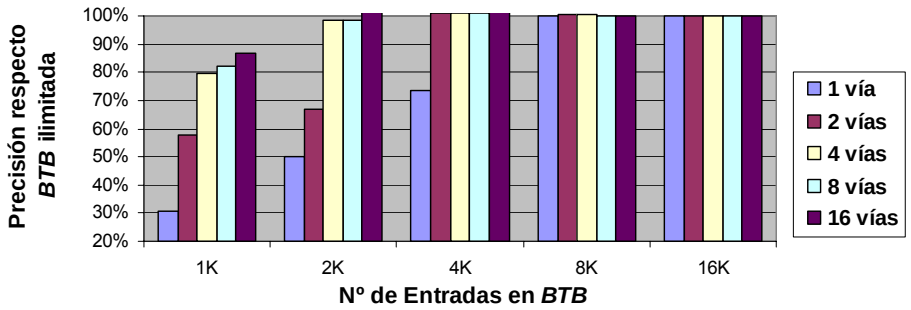


Figura 4.5. Porcentaje de la precisión del predictor para los **saltos indirectos** comparada con el uso de una *BTB* ilimitada, para diversas capacidades y nº de vías.

Fijada la capacidad de *BTB*, un mayor número de vías siempre reduce la razón de fallos. En concreto, una organización de acceso directo produce resultados claramente inferiores a los de una organización asociativa de dos vías con la mitad de capacidad. Considerando únicamente la razón de fallos, una tabla de 16 vías con una capacidad de 4K entradas sería suficiente para lograr prácticamente la máxima precisión. Pero en todo diseño se debe considerar también la latencia y el consumo energético.

4.2.2 Latencia y Consumo Energético en *BTB*

La Figura 4.6 muestra estimaciones de tiempos de acceso para *BTB*, en función de su asociatividad y tamaño. Suponemos que cada entrada en *BTB* contiene 4 bytes de datos y 2 bytes para la etiqueta (*tag*). Los valores han sido obtenidos con eCACTI [MaDu04] para una tecnología de 70 nm.

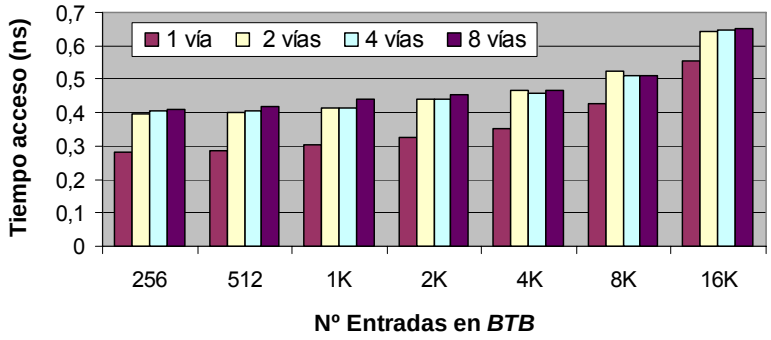


Figura 4.6. Tiempo de acceso en función de la capacidad y nº de vías de *BTB*.

El tiempo de acceso a una organización asociativa respecto al acceso directo es un 50% mayor para tablas pequeñas, y disminuye progresivamente hasta ser un 18% mayor para una tabla de 16K entradas. Asimismo, con menos de 2K entradas, doblar la capacidad aumenta el tiempo de acceso entre un 5.5% y un 7%. En cambio, pasar de 4K a 8K entradas (es decir, de 16KBytes a 32KBytes) aumenta el tiempo de acceso un 20%, y pasar de 8K a 16K entradas (de 32KB a 64KB) aumenta el tiempo de acceso un 30%.

La Figura 4.7 muestra estimaciones de la potencia dinámica consumida por *BTB*, en función de su asociatividad y capacidad. La potencia no crece apreciablemente con la capacidad hasta llegar a tener 4K entradas o más. En cambio, incrementar la asociatividad sí que aumenta considerablemente el consumo energético, que con 8 vías se llega casi a triplicar.

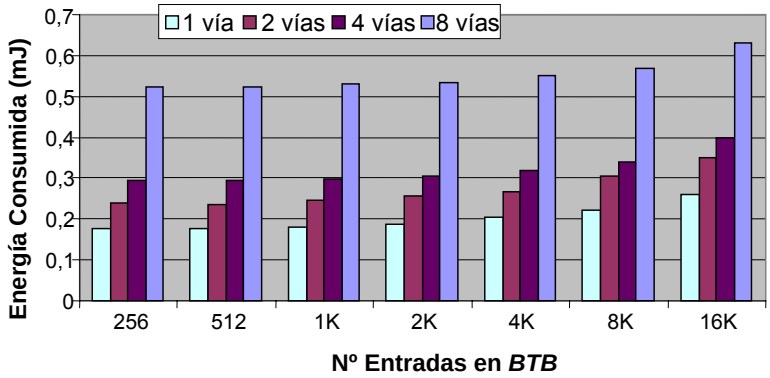


Figura 4.7. Energía dinámica consumida en función de la organización de *BTB*.

Por tanto, la reducción en el número de fallos debida a la mejor gestión de la memoria que hace una organización con alto grado de asociatividad, se ve degradada por un mayor tiempo de acceso y un mayor consumo energético. Las propuestas presentadas en los próximos capítulos pretenden aunar la rapidez y eficiencia de un esquema de acceso directo con el buen manejo de la información de una organización con múltiples vías. A continuación presentaremos brevemente la idea básica en cada uno de los dos casos.

4.2.3 Predicción de Vía

A diferencia del diseño de la Figura 4.1, el de la figura siguiente predice en que vía de una *BTB* asociativa por conjuntos se encuentran los datos de un bloque básico. La predicción de vía permite un acceso directo a *BTB*, y por tanto entre un 18% y un 50% más rápido (Figura 4.6) y reduciendo el consumo energético entre el 30% y el 60% (Figura 4.7). La contrapartida a

estas mejores prestaciones son los posibles fallos de predicción de vía, que demostraremos que son infrecuentes, y por lo tanto aceptables.

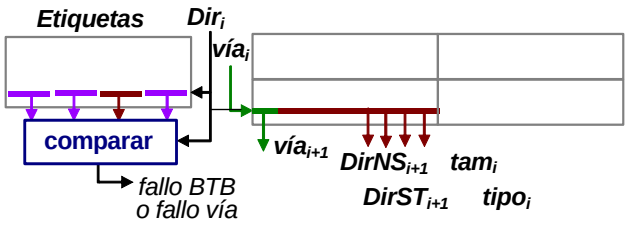


Figura 4.8. Predicción de vía para acelerar el acceso a los datos de *BTB*.

La predicción de vía del ciclo previo (via_i) permite acceder directamente a una entrada en *BTB* y obtener la predicción de vía para el siguiente bloque básico (via_{i+1}). Los fallos de vía se detectan comparando la dirección buscada con las etiquetas correspondientes a todas las vías del mismo conjunto. Si el bloque básico buscado está en una vía diferente a la predicha, entonces se necesita un nuevo acceso a *BTB*, lo cual comporta una penalización de tiempo y de energía.

4.2.4 Predicción de Índice

Puesto que la ruta crítica del predictor corresponde con la generación de una referencia al siguiente bloque básico, una forma de acelerarlo es utilizar una tabla de índices a bloques básicos.

En la siguiente figura se muestra el esquema general de la propuesta. La predicción de índice del ciclo previo (ind_i) permite acceder directamente a una tabla de índices (*index Table, iTbl*), que proporciona el índice al siguiente bloque básico (ind_{i+1}). *iTbl* es de acceso directo y muy pequeña, y por tanto muy rápida. Mostraremos que la selección del índice es también muy rápida, pues sólo hay que escoger entre dos posibilidades, y esta decisión no depende de datos leídos de *iTbl*.

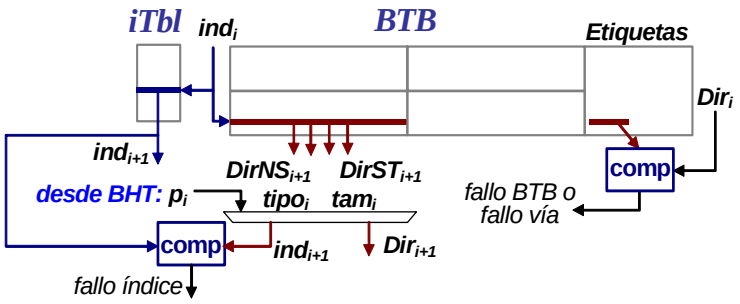


Figura 4.9. Predicción de índices utilizando una tabla de índices (*index Table*, *iTbl*) para aumentar la frecuencia del predictor.

La predicción de índice supone predicciones implícitas tanto referentes al resultado de una instrucción de transferencia de control como a la vía en *BTB*. La predicción del índice permite acceder a mayor velocidad a los datos del predictor, pero estos datos se han de obtener igualmente, y se combinan del mismo modo que en el predictor de referencia. El resultado, producido con un cierto retardo respecto a *iTbl*, se contrasta con la predicción de índice.

Si no hay coincidencia entre el índice predicho por *iTbl*, y el predicho por el mecanismo original, entonces se debe recuperar el predictor con el segundo resultado, lo cual comporta una penalización de tiempo y de energía. A los fallos de *BTB* y de predicción de vía, hay que añadir la posibilidad de un fallo de predicción de índice, que se produce cuando la predicción implícita del salto condicional o del salto indirecto hecha por *iTbl* no coincide con la que se realiza por medio de la combinación de *BHT*, *RAS* e *IJT*. Mostraremos que este último caso es infrecuente.

4.3. Predicción de Vía con Comparación en Paralelo

En este apartado se presenta un diseño sencillo que consigue acelerar el predictor y reducir su consumo energético mediante la predicción de vías. El diseño nos servirá para ejemplificar la idea y para argumentar las ventajas de un segundo diseño más complejo, que se presentará en la sección siguiente.

4.3.1 Predicción de Vía para Saltos Condicionales

La Figura 4.10 muestra los campos que tiene la tabla *BTB* en un diseño con predicción de vía. En comparación al diseño de referencia, se añaden dos campos que predicen la vía en la que se encuentra el bloque básico que viene a continuación, uno en caso de que la instrucción de salto se tome (***DirST***), y otro en caso de que no se tome (***DirNS***). Todas las etiquetas que sirven para identificar los bloques básicos de un mismo conjunto (***etiq$_0$ … etiq$_{w-1}$***) se agrupan en la misma entrada, ya que se leen simultáneamente para comprobar en cual de las vías se encuentra el bloque básico.

Figura 4.10. Campos de *BTB* divididos en datos y etiquetas (*tags*).

La Figura 4.11 muestra el diseño de *BTB* con predicción de vía y con comparación de etiquetas en paralelo. A diferencia del diseño de la Figura 4.1, la ruta que proporciona los datos de *BTB* y la ruta que comprueba en qué vía se encuentran los datos están separadas. La primera ruta, crítica, es más rápida que la segunda.

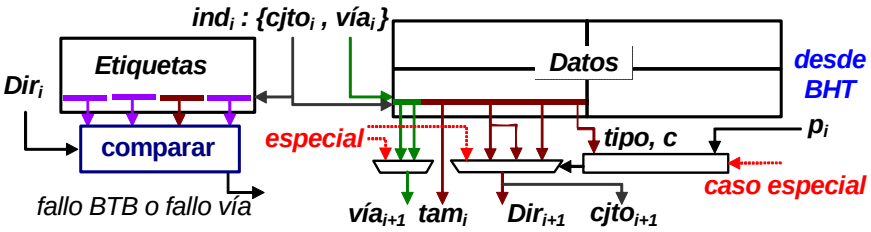


Figura 4.11. Organización de *BTB* con predicción de vía y con comparación en paralelo de las etiquetas (*tags*) correspondientes a cada una de las vías.

Los bits de menor peso de la dirección del bloque básico actual (Dir_i) proporcionan la parte del índice que determina el **conjunto** en que debe estar la información del bloque básico ($cjto_i$). El resto de bits del índice codifican en

qué **vía** dentro del conjunto ($vía_i$). Los identificadores de conjunto y de vía forman el índice completo con el que acceder a la tabla de datos (ind_i). La tabla de etiquetas se indexa únicamente con el identificador de conjunto.

En la figura anterior se puede comprobar que la dirección del siguiente bloque básico (Dir_{i+1}) se obtiene de la selección de entre cuatro posibles direcciones. Tres de las direcciones proceden de *BTB* y hay una cuarta dirección *especial*. Una de las direcciones que proceden de *BTB* corresponde al caso de saltar y las otras dos corresponden al caso de no saltar, seleccionadas por el campo ***c*** y calculadas como se mostró en la Figura 3.7. La dirección especial puede usarse para varios casos. Si el bloque básico a predecir finaliza en un salto de retorno de subrutina, esta dirección corresponde con la que proporciona la pila *RAS*. Si el predictor se encuentra en la fase de recuperación de un fallo (tal vez de predicción de vía), entonces la dirección es la de recuperación. Esta dirección puede haber sido obtenida de *IJT*, en caso de que se tratara de un salto indirecto polimórfico. Para la predicción de vía ($vía_{i+1}$) se debe seleccionar entre las dos predicciones de *BTB* (saltar o no saltar) y la predicción especial.

Tal como se ha dicho, el camino de datos sólo necesita leer los datos de un bloque básico, y no necesita esperar a que el camino de comprobación (lento) indique en qué vía se encuentran. Este cambio acelera el predictor entre un 20% y un 50%, dependiendo del tamaño de la tabla. Sin embargo, cuando el camino de comprobación detecta que la predicción de vía es errónea, ya se ha iniciado la siguiente predicción, y por tanto se debe esperar un ciclo de predicción completo para poder “recuperar” el predictor.

En la Figura 4.12 se observa cómo la detección de los fallos de predicción de vía se realiza en el ciclo siguiente al acceso a la vía equivocada. Como las etiquetas de todas las vías del mismo conjunto se comparan en paralelo, al detectar el fallo de vía también se dispone del identificador de la vía en la que encontrar el bloque básico requerido. Por tanto, cada fallo de predicción de vía retrasa dos ciclos la operación del predictor.

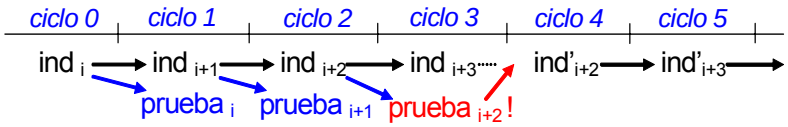


Figura 4.12. El acceso a la información del bloque *i+3* se hace en paralelo con la detección de fallo de predicción de vía para el bloque *i+2*

Los campos de predicción de vía se actualizan a la vez que los campos de predicción de dirección que les corresponden. Para ello, en la fase de actualización se debe retrasar la escritura en *BTB* hasta conocer la vía del bloque básico que viene a continuación del que se quiere actualizar.

4.3.2 Predicción de Vía para Saltos Indirectos

La dirección de algunos saltos indirectos no se obtiene de *BTB*, y en estos casos no se debería usar la predicción de vía almacenada en *BTB*. Tal como muestra la siguiente figura, los saltos indirectos que se predicen con *IJT* y los saltos indirectos de retorno a subrutina que se predicen con RAS requieren que se almacene un campo de predicción de vía en la propia tabla o pila.

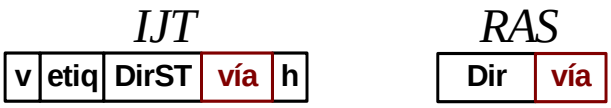


Figura 4.13. Campos de *IJT* y de *RAS*, incluyendo la predicción de vía.

La gestión para *IJT* es relativamente simple: cada vez que se almacena en la tabla una dirección de salto también se almacena la vía dentro de *BTB* donde se encuentra el bloque básico siguiente.

Para manejar correctamente la pila RAS es necesario que todos los bloques básicos de salto a subrutina almacenen en *BTB* tanto la predicción de vía para el bloque donde saltar (***víaST***) como la predicción para el bloque consecutivo en memoria (***víaNS***), donde debe saltar la instrucción de retorno de subrutina que le corresponde. Así, al predecir un bloque básico de salto a

subrutina, se usa la predicción de vía ***víaST*** para ir a buscar el nuevo bloque a *BTB*, pero se almacena en la pila *RAS* la otra predicción de vía, ***víaNS***, junto con la dirección de retorno, ***DirNS***, para recuperarla al volver de la subrutina. Veremos en breve que la actualización del campo ***víaNS*** es problemática y que requiere el uso de una pila adicional.

4.3.3 Fase de Predicción

Respecto a la microarquitectura de referencia, la fase de predicción sólo varía en que se utilizan el campo de dirección y el campo de predicción de vía para formar el índice a *BTB*. Además, en la cola de predicciones se almacena la vía donde se ha encontrado cada bloque básico, para usarla posteriormente en la fase de actualización.

4.3.4 Fase de Recuperación

Al producirse un fallo se proporciona al predictor la dirección del bloque básico donde se debería haber saltado (Dir_{i+1}). Con esta dirección se puede obtener la parte del índice a *BTB* que indica el conjunto ($cjto_{i+1}$) donde acceder. Pero no se dispone de información de en qué vía se debe acceder ($vía_{i+1}$). La solución más sencilla para evitar una búsqueda en *BTB* de la vía correcta es, para cada predicción, almacenar la predicción de vía alternativa en la cola de predicciones. Por tanto, a la información que debe contener esta cola, ya especificada en el capítulo anterior, habría que añadir:

- la predicción de vía tanto para el caso de saltar (*víaST*), como para el de no saltar (*víaNS*)

Según la causa del error, se usará la información disponible de una forma u otra. Los errores detectados en la fase de decodificación se deben a la falta de información en *BTB* (o a usar información errónea) relativa al bloque de instrucciones. Al encontrar una instrucción no esperada de salto, o de un tipo

diferente al esperado, se genera el fallo y se proporciona al predictor el tipo correcto del salto y la dirección de salto para los saltos de tipo directo. Si la instrucción de salto es de **retorno de subrutina**, se usa la cima de la pila *RAS* para obtener una predicción de vía. En cualquier otro caso, no se dispone de información para predecir la vía, así que se usa el valor 0 (que es siempre la primera vía que se utiliza al emplazar bloques básicos, y por tanto la que tiene más probabilidad de ser la correcta).

Si se trata de un error detectado en el núcleo de ejecución para un salto **condicional**, se utiliza la cola de predicciones para recuperar la predicción de vía del camino alternativo al de la predicción original. Si el error se produjo para un salto **indirecto**, entonces tampoco se dispone de información para predecir la vía, a no ser que la predicción correcta corresponda con la predicción por defecto de *BTB*. En lugar de hacer una predicción de vía arbitraria, se recupera la predicción de vía de la cola de predicciones, que proporciona mejor resultado que otras alternativas.

Si la recuperación se debe a que la predicción de *IJT* debe reemplazar a la de *BTB* para un salto indirecto polimórfico, entonces la predicción de vía se obtiene de *IJT* junto a la nueva dirección del bloque básico.

4.3.5 Fase de Actualización

Cuando un bloque básico ha sido retirado ya se conoce con seguridad la dirección del bloque básico siguiente en la secuencia dinámica de ejecución. Es posible que en la cola de predicciones ya exista la información sobre la vía en la que se encuentra ese bloque básico siguiente. Si es así, el bloque básico actual puede comparar la predicción actual de vía (almacenada en la cola) con la posición en la que realmente se encontró el bloque básico siguiente al ser accedido en la fase de predicción. Si no coinciden, lo cual es poco frecuente, se debe actualizar el campo de predicción de vía.

Puede ocurrir que el bloque básico siguiente fallara en *BTB* en la fase de predicción. En este caso se debe esperar a retirar el bloque básico siguiente, y a asignarle una vía en *BTB*. En ese momento ya se puede actualizar la predicción de vía del bloque actual, y pasar al siguiente bloque básico.

Al insertar por primera vez en *BTB* la información de un bloque básico, se conocen tanto la dirección del bloque donde saltar (en caso de saltos directos) como la dirección del bloque consecutivo en memoria. Sin embargo, como sólo uno de los dos bloques básicos es el siguiente en la secuencia dinámica de ejecución, sólo se podrá inicializar correctamente uno de los campos de predicción de vía. El otro campo de predicción de vía se inicializa con un valor arbitrario (el valor 0 da buen resultado), que puede tener que corregirse al retirar posteriormente el bloque correspondiente a ese campo.

Para poder actualizar el campo ***víaNS*** de un bloque básico de salto a subrutina, hay que acceder primero al bloque básico consecutivo y así conocer en qué vía se encuentra. El problema es que el segundo bloque se retira justo después de volver de la subrutina, mucho más tarde que el bloque que realiza el salto a subrutina, ya que entre medio han de retirarse todos los bloques básicos que componen la subrutina (y los bloques básicos de las posibles subrutinas internas). La solución a este problema es utilizar una pila (similar a *RAS*) en la fase de actualización. Al retirar un bloque básico de salto a subrutina, se almacena en la pila el índice en *BTB* de este bloque. Al retirar el bloque básico siguiente a un salto a subrutina, se extrae de esta pila el índice del bloque básico de llamada a subrutina y se actualiza su campo de predicción de vía.

4.3.6 Resultados: Fallos de Predicción de Vía

Hay dos factores básicos que, a priori, han de determinar la razón de fallos de predicción de vía. En primer lugar, los fallos de vía deberían estar muy correlacionados con los fallos de *BTB*. Esto es debido a que cada fallo de *BTB* supone el re-emplazamiento de un bloque básico, que cuando se vuelve

a traer a *BTB*, probablemente irá a una vía diferente. En segundo lugar, cuanta menor asociatividad tenga la *BTB*, mayor será la probabilidad de acertar la vía simplemente por azar.

La siguiente figura muestra datos experimentales (modelando el predictor aislado) del número de instrucciones retiradas por fallo de predicción de vía para capacidades de *BTB* de entre 512 y 16K entradas, y número de vías entre 2 y 16. Si se cruzan los datos de la Figura 4.14 con los datos sobre porcentaje de fallos de *BTB* de la Figura 4.2, se puede comprobar que la correlación entre ellos es, como era de prever, muy grande.

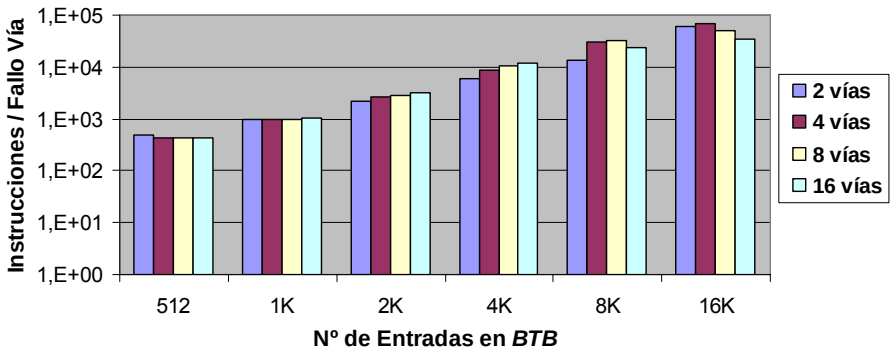


Figura 4.14. Instrucciones retiradas por fallo de predicción de vía en función de la capacidad y número de vías de *BTB*. Se utiliza *RAS* e *IJT* para la predicción de vía.

Con una *BTB* de 16K entradas, el número de fallos de predicción de vía es alrededor de 10 por cada fallo de *BTB*. El 85% de los fallos son consecuencia del fallo de predicción previo de un salto indirecto, que sólo en ciertos casos puede preverse con precisión, mientras que sólo el 15% de los fallos son consecuencia de un fallo de *BTB* previo. Asimismo, el número de fallos de predicción de vía es menor a medida que la tabla tiene menos vías, tal y como podría esperarse por razones puramente probabilísticas.

Al disminuir la capacidad en *BTB*, se incrementa el número de fallos de *BTB* y por tanto el número de fallos de predicción de vía. Pero con menos de 8K entradas, el número de fallos de vía es menor cuanto mayor es el número de vías en *BTB*. Esto es debido a que, aunque tener menos vías da más

oportunidades a la suerte, también supone una peor gestión de la tabla y provoca más fallos de *BTB*, que anulan la ventaja del primer factor.

Para ver el efecto de los mecanismos de predicción de vía específicos para los saltos indirectos, la Figura 4.15 muestra resultados en función de cuáles de estos mecanismos se incluyen o no. Se puede comprobar la gran importancia de ambos mecanismos, ya que entre los dos aumentan la precisión de las predicciones de vía más de un orden de magnitud.

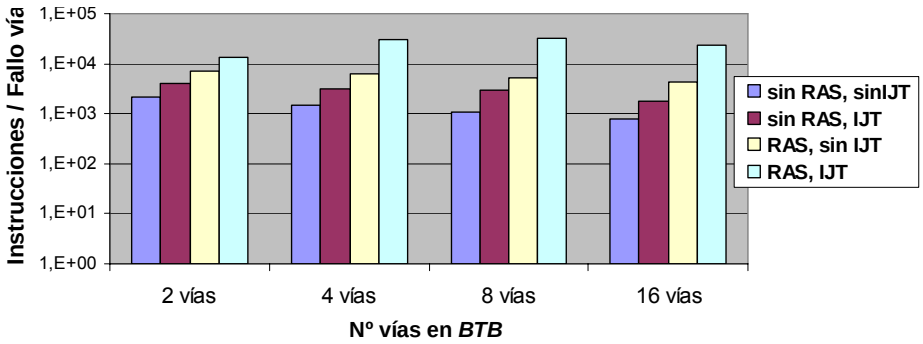


Figura 4.15. Instrucciones retiradas por fallo de predicción de vía en una *BTB* de 8K entradas, en función del número de vías y de si se utiliza *RAS* e *IJT*.

4.3.7 Conclusión

El mecanismo de predicción de vías genera menos de un fallo cada 10.000 instrucciones con tablas *BTB* de 8K entradas o más, o con una tabla de 4K entradas y 8 vías o más. Con una tabla de 1K entradas, se produce de media un fallo de predicción de vía cada 1000 instrucciones, que multiplicados por 2 ciclos de penalización supondrían incrementar en 0,002 ciclos el tiempo medio de predicción por instrucción. Esta cantidad se compensaría con reducciones en la latencia del predictor superiores al 1,6% (suponiendo bloques básicos de 8 instrucciones).

Por tanto, el problema de una *BTB* de 1K entradas no es la predicción de vía, sino que, debido a los frecuentes fallos de *BTB*, la precisión del predictor

disminuye hasta el 80%, incluso con una organización de 8 vías (ver Figura 4.4). Pero tener una *BTB* de alta capacidad aumenta la latencia del predictor y aumenta el consumo. Además, la comparación en paralelo de todas las etiquetas del conjunto, aunque simplifica la detección de fallos de predicción de vía y de *BTB*, supone un mayor consumo energético.

Para disponer de una *BTB* de gran capacidad y a la vez mantener un moderado tiempo medio de acceso y un moderado consumo energético, es muy apropiada una organización jerárquica, usada comúnmente para las instrucciones y datos del programa. Reinmann et al. [ReAC99] proponen el uso de un predictor de dos niveles, denominado *FTB* (*Fetch Target Buffer*). El predictor es muy similar al predictor de bloques básicos que se está usando como referencia, y también se utiliza en una arquitectura desacoplada. La contribución del siguiente apartado es aplicar la técnica de predicción de vía.

4.4. Jerarquía de 2 Niveles con Predicción Cruzada de Vía

En este apartado mostraremos un diseño más complejo que el presentado en el apartado anterior, y que aúna las sinergias de la predicción de vía con una organización de dos niveles, para así aumentar la velocidad del predictor y reducir su consumo energético. Al final de esta sección se describirá de forma detallada el funcionamiento del diseño, diferenciando entre las fases de predicción, de actualización, y de recuperación de un fallo de predicción.

4.4.1 Descripción General

La siguiente figura muestra el diagrama de bloques de un predictor con una organización de *BTB* en dos niveles. Consideramos que la tabla de 2º nivel (*L2-BTB*) contiene toda la información de la tabla de 1er nivel (*L1-BTB*). Para reducir el consumo energético, el acceso a la tabla de 2º nivel comienza una vez que se detecta el fallo en la tabla de 1er nivel.

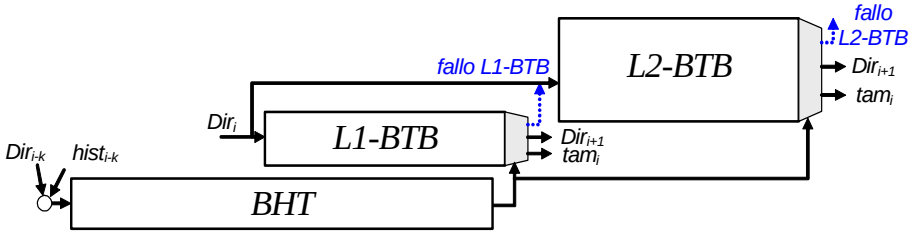


Figura 4.16. Organización genérica para almacenar la información de bloques básicos en dos niveles.

Supongamos que se aplica la técnica de predicción de vía a *L1-BTB* y que se detecta que los datos leídos no corresponden al bloque básico requerido. En la propuesta del apartado anterior se hacía una búsqueda asociativa en todas las vías de *L1-BTB* para determinar en qué otra vía se encuentra el bloque básico o para confirmar que no está en la tabla de 1er nivel. Tal como se ha comentado al final de la sección anterior, si el número de vías de *L1-BTB* es grande, ésta será una operación costosa en consumo energético y de área de chip. Una posible alternativa es realizar la búsqueda de forma secuencial, pero se aumenta la latencia de la operación e incluso se puede aumentar la complejidad del diseño y el consumo energético.

En el diseño de dos niveles, al detectar el fallo en *L1-BTB* se opta por renunciar a encontrar el bloque básico, e ir a buscarlo directamente al 2º nivel. La forma más rápida y eficiente de hacerlo es almacenar en *L1-BTB* predicciones de vía a *L2-BTB.* Si *L2-BTB* es grande, el número de re-emplazamientos que sufrirá será pequeño, y la posibilidad de un nuevo fallo de predicción de vía, ahora en *L2-BTB*, será muy pequeña. El predictor obtendrá del 2º nivel la información del bloque básico, pero además obtendrá la predicción de vía necesaria para seguir sus operaciones utilizando de nuevo *L1-BTB*.

La nueva propuesta presenta las siguientes ventajas respecto a la anterior:

- Acceso directo, muy rápido, a una tabla *L1-BTB* que puede ser pequeña. Los fallos en esta tabla están “protegidos” por la tabla del segundo nivel para no afectar a la precisión del predictor.

- El control de fallos en *L1-BTB* es más simple y no depende del grado de asociatividad de la tabla: se compara una única etiqueta y, si la comparación falla, siempre se procede del mismo modo, que es acceder a la tabla de 2º nivel.
- *L2-BTB* puede ser tan grande como permitan las restricciones del diseño, y así tener un grado de asociatividad más reducido. Las búsquedas asociativas en *L2-BTB* se necesitan muy esporádicamente, y pueden eliminarse de la fase de predicción y relegarse a casos poco frecuentes en la fase de actualización.

4.4.2 Descripción Detallada del 1er Nivel

La Figura 4.17 muestra los campos de *L1-BTB* en una organización de dos niveles con predicción de vía. Igual que en el diseño de la Figura 4.10, los campos de la tabla de primer nivel también se dividen entre dos bancos diferentes. A parte de esta similitud, los cambios son numerosos. El banco de datos de *L1-BTB* almacena los apuntadores a los dos bloques básicos que pueden ir a continuación del bloque básico actual (***indNS*** e ***indST***), con parte de los bits que son los identificadores de conjunto (***cjtoNS*** y ***cjtoST***) y el resto de bits que son las predicciones de vía (***víaNS*** y ***víaST***).

Figura 4.17. Campos de *L1-BTB* divididos en banco de datos y de etiquetas.

El banco de etiquetas de *L1-BTB,* tal como se ha dicho, sólo permite leer la información de un único bloque básico en cada acceso. Se puede corroborar que la información leída en el banco de datos es la correcta, pero en caso de que no lo sea, se necesitarían más accesos para averiguar si se trata de un

fallo de predicción de vía o de un fallo de *BTB*. Además de realizar esta comprobación, el banco de etiquetas genera la dirección del siguiente bloque básico y las predicciones de vía a *L2-BTB*. Cada entrada almacena la etiqueta del bloque básico actual y del bloque básico donde saltar (***etiq*** y ***etiqST***). También contiene los dos campos que predicen la vía en el 2º nivel en la que se encuentran ambos bloques básicos (***víaNS2*** y ***víaST2***).

La figura siguiente muestra el circuito que genera el tamaño del bloque básico actual y el índice en *L1-BTB* para el bloque básico predicho como siguiente. *L1-BTB* contiene las dos posibles predicciones para el índice (saltar o no saltar) y el tipo del bloque básico, que sirve para tomar la decisión de selección. En la selección final del índice del siguiente bloque básico hay que incluir el índice predicho por la pila *RAS* para las instrucciones de retorno de subrutina, y el índice utilizado al recuperar el predictor tras un fallo.

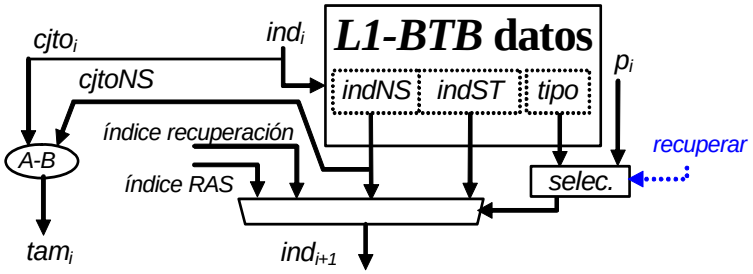


Figura 4.18. Circuito de generación del índice del siguiente bloque básico y del tamaño del bloque básico actual por parte de *L1-BTB*

En comparación con el diseño del apartado anterior (Figura 4.11), el banco de datos de *L1-BTB* no contiene la parte de la etiqueta de la dirección de memoria destino del salto, que ahora se encuentra en el banco de etiquetas. Dependiendo del número de entradas en *L1-BTB*, que determina la longitud de los índices, las entradas de *L1-BTB* pueden llegar a reducirse hasta el 50% de su tamaño anterior, y reducir así la latencia de las predicciones.

El tamaño del bloque actual (tam_i) se obtiene restando a la parte del índice correspondiente al bloque básico contiguo (*indNS*) que representa el conjunto

(*cjtoNS*), los bits bajos de la dirección del bloque actual, que también representan el conjunto dentro de BTB ($cjto_i$).

La siguiente figura muestra el circuito para generar la etiqueta del siguiente bloque básico ($etiq_{i+1}$) y la predicción de vía en *L2-BTB* para el siguiente bloque básico ($víaL2_{i+1}$). La dirección completa del bloque siguiente (Dir_{i+1}) se obtiene juntando la etiqueta del siguiente bloque básico ($etiq_{i+1}$) con el conjunto obtenido del banco de datos de *L1-BTB* ($cjto_{i+1}$).

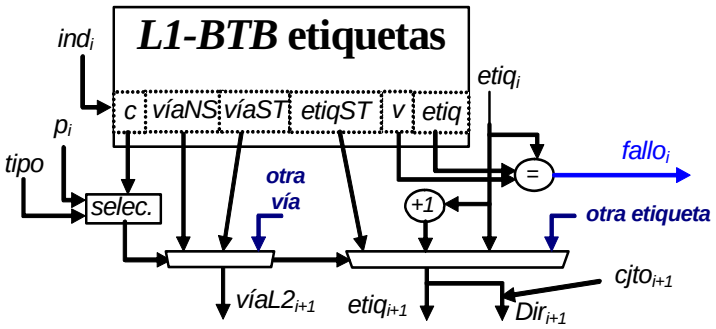


Figura 4.19. Circuito de generación de la etiqueta y de la dirección del siguiente bloque básico, y de detección de fallos en *L1-BTB*.

Tal como se mostró para el predictor de referencia, si la predicción para el bloque básico es no saltar, la etiqueta del siguiente bloque básico se puede obtener sumando 1 a la etiqueta del bloque básico actual ($etiq_i$). Los casos menos frecuentes, como el uso de la pila *RAS* o la recuperación de un fallo de predicción, quedan englobados dentro de un único caso en el multiplexor final del circuito de la figura (referenciado como “otra etiqueta” y “otra vía”). H

La Figura 4.19 también muestra el control de fallos en el acceso a *L1-BTB*. Cuando la etiqueta leída de la tabla (*etiq*) no corresponde con la etiqueta del bloque básico buscado ($etiq_i$), esto implica que, o bien el bloque básico buscado se encuentra en una vía diferente a la predicha, o bien no se encuentra en *L1-BTB*. En ambos casos, tal como se ha explicado, se inicia el acceso a *L2-BTB* usando el predictor de vía obtenido en el ciclo anterior ($víaL2_i$). Aunque el fallo se deba a una predicción de vía errónea, es más

simple y en ocasiones más rápido ir a buscar la información a *L2-BTB* que buscar en todas las posibles vías de *L1-BTB* hasta encontrarla (o quizás no).

Tanto *IJT* como *RAS* deben acomodar un nuevo campo que contenga la predicción de vía para el 2º nivel (ver figura siguiente).

Figura 4.20. Campos de *IJT* y RAS incluyendo las predicciones de vía de 2º nivel.

4.4.3 Descripción Detallada del 2º Nivel

La tabla *L2-BTB* sólo es accedida cuando se detecta un fallo en *L1-BTB*. La Figura 4.21 muestra los campos de *L2-BTB*, divididos en los habituales bancos de datos y de etiquetas. En el banco de datos se guarda la información necesaria para generar las direcciones completas de los dos bloques básicos que pueden seguir al bloque básico actual, y que es la misma que contenía *BTB* en el diseño de referencia descrito en el capítulo anterior. También se incluye la predicción de vía para estos dos bloques, tanto en *L1-BTB* como en *L2-BTB*.

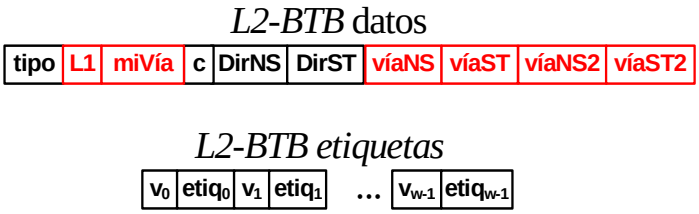


Figura 4.21. Campos de *L2-BTB* divididos en datos y etiquetas (*tags*).

El campo ***L1*** es un bit que indica si el bloque básico se encuentra almacenado actualmente o no en *L1-BTB*. En caso de que sea así, el campo ***miVía*** indica en qué vía de *L1-BTB* se encuentra el bloque. Al acceder a *L2-*

BTB debido a un fallo en *L1-BTB*, ***L1*** permite saber rápidamente si el problema fue un fallo de predicción de vía o un fallo real. Si fue un fallo de predicción de vía, el valor ***miVía*** se guarda en la cola de recuperación para ayudar a la fase de actualización, y evitar que se intente insertar en *L1-BTB* un bloque que ya reside en ella.

La figura siguiente muestra como se genera con *L2-BTB* la dirección del bloque básico siguiente y la predicción de vía en *L1-BTB*. Es un circuito muy similar al diseño de referencia sin predicción de vía presentado en el capítulo anterior. Si el tipo de bloque básico es de retorno de subrutina, se utilizará la pila *RAS* para generar la dirección. Si el tipo es un salto indirecto polimórfico, se iniciará el acceso a *IJT*, aunque el uso de la dirección que proporcione se controlará posteriormente con el multiplexor de *L1-BTB* (Figura 4.19).

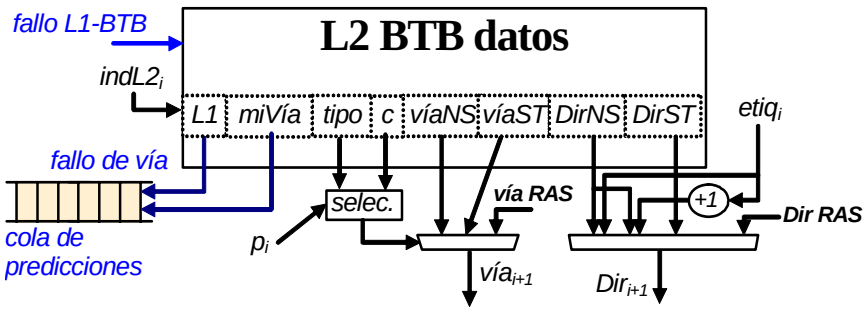


Figura 4.22. Circuito de generación de la dirección y de la predicción de vía en *L1-BTB* del siguiente bloque básico. No se verifican fallos en *L2-BTB*.

Una importante decisión de diseño consiste en no verificar los accesos a *L2-BTB* en la fase de predicción. De este modo, el circuito de predicción se simplifica y se reduce el consumo energético. A cambio, se renuncia a detectar los fallos en *L2-BTB*, tanto los fallos de predicción de vía como los producidos por no encontrarse los datos. La detección de estos fallos se retrasa hasta la etapa de decodificación, con un cierto número de ciclos de penalización, pero la precisión del predictor no queda afectada, ya que la información sigue estando en *L2-BTB* y en la fase de actualización se modificará el predictor como corresponde.

4.4.4 Fase de Predicción

La fase de predicción consiste en lecturas sucesivas a los bancos de datos y de etiquetas de *L1-BTB*. Los accesos al banco de datos marcan la latencia del predictor, mientras que los accesos al banco de etiquetas pueden ser más lentos. Así, el ciclo del predictor ya ha finalizado con la obtención del índice al siguiente bloque básico mientras que aún no se ha obtenido ni la dirección completa del siguiente bloque básico ni se ha comprobado el fallo en *L1-BTB*.

La figura siguiente muestra como la última parte de la generación de Dir_{i+1} y tam_{i+1}, y la comprobación de fallo en *L1-BTB*, se realizan mientras ya se está accediendo al supuesto bloque *i+1* usando ind_{i+1}. La figura también muestra que mientras no se detectan fallos en *L1-BTB* no se accede a *L2-BTB*. Al detectar un fallo en *L1-BTB* (que en la figura se ilustra con el bloque cuya dirección inicial es Dir_{i+1}), se anula el acceso en curso en *L1-BTB* y se inicia el acceso a *L2-BTB* usando el predictor de vía obtenido de *L1-BTB* en el ciclo de predicción anterior ($indL2_{i+1}$). Como en *L2-BTB* no se comprueban los fallos, los datos leídos siempre se utilizarán, tanto si son correctos como si no, y con estos datos se reinician los accesos a *L1-BTB*, tal como muestra el camino marcado en la figura con (1). En el esquema de la figura, el fallo de *L1-BTB* provoca una penalización de 2 ciclos. La penalización en un diseño concreto dependerá de la relación entre los tiempos de acceso a *L1* y *L2*.

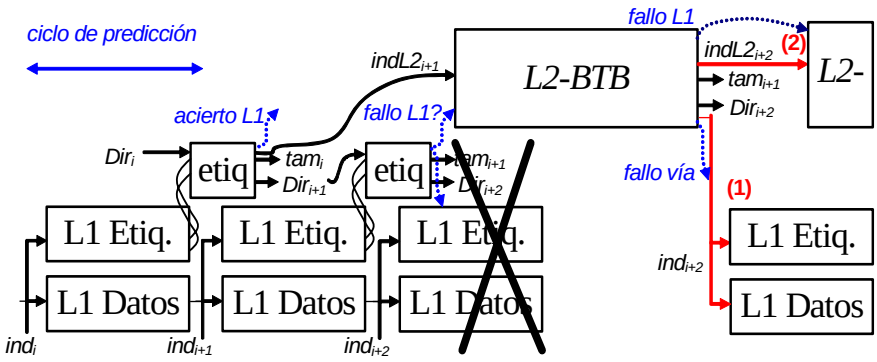


Figura 4.23. Predictor con *BTB* organizada en dos niveles. En el ejemplo se ve como un fallo en la tabla L1 para el bloque *i+1* provoca una penalización de 2 ciclos.

El acceso a *L2-BTB* proporciona el contenido del campo ***L1***, que permite discernir correctamente entre fallos reales y fallos de predicción de vía en *L1-BTB*. Si se detecta que el problema ha sido un fallo de predicción de vía, entonces es muy probable que la nueva predicción de vía para *L1-BTB*, leída ahora de *L2-BTB,* sea correcta, y se rompa una posible cadena de fallos de predicción de vía. En cambio, si se detecta que el problema ha sido un fallo real en *L1-BTB*, entonces es bastante probable que se produzca una cadena de fallos reales en *L1-BTB*. En este caso, se toma la decisión de diseño de adelantar el camino paralelo marcado en la figura con el número (2), para así reducir la penalización de un posible nuevo fallo en *L1-BTB* a un único ciclo, en lugar de dos.

La siguiente figura muestra de forma esquemática el mismo ejemplo de la figura anterior. Cada ciclo se obtiene un nuevo índice a *L1-BTB* (ind_i, ind_{i+1}, ind_{i+2}, …), y la comprobación de que el índice usado era correcto finaliza un poco después de haber comenzado el acceso con el siguiente índice. Por ejemplo el acceso a *L1-BTB* para el bloque *i+1* se inicia en el ciclo 1 con ind_{i+1}, y el error se detecta ya iniciado el ciclo 2. Al detectar un error, se inicia inmediatamente el acceso a *L2-BTB*, en el ejemplo usando $indL2_{i+1}$. Al comenzar el ciclo 4 ya se dispone de los índices para el bloque *i+2*, tanto en *L1* como en *L2*. Además, ya se sabe si el error en *L1* fue provocado por un fallo real o por un fallo de predicción de vía. En el primer caso, tal como se ha comentado, se adelanta el acceso a *L2-BTB* para prevenir un posible nuevo fallo en *L1-BTB* (camino (2) en la figura).

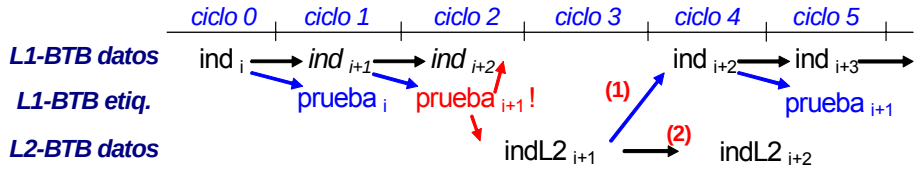


Figura 4.24. Al final del ciclo 2 se detecta el fallo en *L1-BTB* al acceder al bloque *i+1* mientras ya se accedía al bloque *i+2*. Tras el acceso a *L2* se conoce la causa del fallo en *L1*. El camino (2) usa si el fallo fue real, y no debido a una mala predicción de vía.

4.4.5 Fase de Recuperación

Al recuperarse del fallo de predicción de un salto condicional, la cola de predicciones o la pila *RAS* proporcionan la predicción de la vía alternativa para los dos niveles de *BTB*. La otra diferencia respecto a la organización con un único nivel es la gestión de los fallos detectados en la etapa de decodificación o debidos a un salto indirecto.

Tras un fallo detectado en la etapa de decodificación o tras un fallo de predicción de saltos indirectos, no se dispone de una predicción de vía adecuada, y es fácil que se produzca un nuevo fallo de predicción de vía tanto en *L1-BTB* como en *L2-BTB.* El nuevo fallo de predicción de vía en *L2-BTB* puede volver a generar un fallo de decodificación, y producir una cadena de fallos que no finaliza hasta que, por azar, se realice una predicción de vía correcta. Para evitar estas cadenas de fallos, se tratan los dos casos anteriores de forma especial.

Tras la recuperación del fallo, a la vez que se inicia el acceso a *L1-BTB*, también se accede al banco de etiquetas de *L2-BTB*, para leer en paralelo todas las etiquetas del conjunto y encontrar la vía en *L2* correspondiente al bloque que se comienza a predecir. Si se detecta fallo en *L1-BTB*, se puede entonces acceder a la vía adecuada de *L2-BTB*, rompiendo el círculo de fallos de predicción de vía.

4.4.6 Fase de Actualización

La penalización de los fallos en *L1-BTB* y la no detección de los fallos en *L2-BTB*, hacen que la organización de 2 niveles sea más eficiente si, al revés que en los dos esquemas previos, también se guardan los bloques básicos que tienen longitud máxima y no finalizan en ninguna instrucción de salto.

Cuando en la etapa de predicción se produce un acierto en *L1-BTB*, no es necesario actualizar ningún campo de *L2-BTB*. Los cambios en los campos de predicción de vía de *L1-BTB* se hacen del mismo modo que en el diseño

básico. Igualmente, al registrar por primera vez un bloque básico en *L1-BTB*, sólo se conoce con certeza la predicción de vía de uno de los dos bloques básicos siguientes, y el otro campo de predicción de vía se inicializa con el valor 0. La actualización correcta se hará cuando se detecte posteriormente un error de predicción de vía.

Al retirar un bloque de salto a subrutina, se deben almacenar los índices a los dos niveles de *BTB* en la pila especial de actualización. Al retirar el bloque siguiente a un bloque de retorno de subrutina, se extraen de la pila estos dos índices al bloque de llamada a subrutina y, si es necesario, se actualizan sus campos de predicción de vía.

La Figura 4.25 muestra las operaciones de actualización que pueden ser necesarias. Para reducir el número de accesos a las tablas, sólo se realiza la actualización correspondiente al bloque básico *i,* una vez que se ha analizado el resultado del bloque básico siguiente, *i+1*.

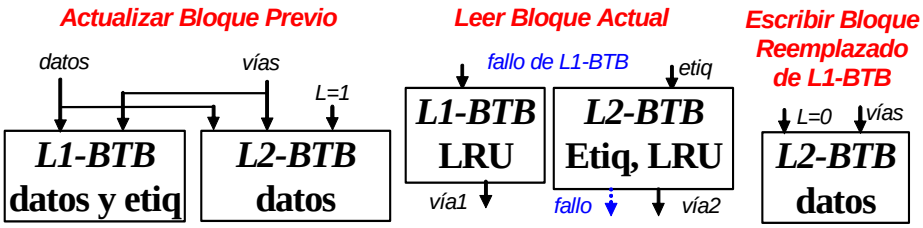


Figura 4.25. Actualización de la organización de dos niveles.

Sólo se comprueban fallos si se detectó un error en la fase de predicción. La detección previa de un fallo en *L1-BTB* hace que se lea la información de reemplazo y que se invoque la política de reemplazamiento *LRU* (la entrada usada menos recientemente). Como el fallo en *L1-BTB* provocó que se usara la información de *L2-BTB*, también se comprueban las etiquetas de *L2-BTB* para detectar otro posible fallo, y realizar, si es necesario, el emplazamiento. Usando el campo ***L1***, se asegura que sólo se reemplaza de *L2-BTB* un bloque básico si no está almacenado en *L1-BTB*.

Cuando un bloque básico es desplazado del 1er al 2º nivel se hacen dos actualizaciones muy importantes en *L2-BTB*. Se actualiza el campo ***L1*** para así poder discernir posteriormente entre fallos reales en *L1-BTB* y fallos de predicción de vía. También se copian en el 2º nivel los campos de predicción de vía del 1er nivel, que proporcionan una muy buena predicción cuando el bloque básico ha de volver a ser transferido a *L1-BTB.*

Lo que no es viable al reemplazar un bloque básico de *L1-BTB*, es actualizar los campos de predicción de vía de todos los bloques básicos que apuntan a él. Se requeriría una costosa operación de búsqueda, que sólo serviría para detectar un ciclo antes algunos de los fallos en *L1-BTB*. Nuestra propuesta de diseño consiste en que tras un caso potencialmente peligroso, por ejemplo un fallo en *L1-BTB* o un fallo de predicción de salto indirecto, se inicie el acceso a *L2-BTB* en paralelo con el acceso a *L1-BTB* para así reducir en un ciclo la penalización de un posible fallo.

4.4.7 Resultados: Fallos de Predicción de Vía y Fallos en *L1-BTB*

Con el modelo de predictor aislado, se han realizado simulaciones para corroborar que la organización de dos niveles funciona de forma similar a la de una única *BTB* del tamaño de la tabla de segundo nivel.

Para la experimentación se han escogido dos configuraciones de la tabla *L2-BTB*. La configuración de 4K entradas y 8 vías sufre un número de fallos aceptable (1 fallo en *L2-BTB* cada 60000 instrucciones, ver Figura 4.2), y mucho más importante, sólo reduce la precisión de los saltos condicionales e indirectos un 1% respecto a una tabla ilimitada. La contrapartida de esta configuración es su relativamente gran tamaño. La Tabla 4-1 muestra el tamaño de *L2-BTB*, distribuido entre la parte de datos, etiquetas y predicciones de vía. Una configuración de 2K entradas y 4 vías reduce los requerimientos de memoria hasta menos de la mitad (16,75 KB frente a 36 KB), pero genera 1 fallo en *L2-BTB* cada 3400 instrucciones, y reduce la precisión para los saltos condicionales un 8%.

Tabla 4-1 Tamaño del predictor para diferentes organizaciones de *L1-BTB* y *L2-BTB,* mostrando por separado la memoria dedicada a datos, etiquetas, y predicciones de vía. Se suponen direcciones de 28 bits y etiquetas de 16 bits.

Configuración	Nº Entradas	Nº Vías	Datos y Etiquetas	Predicciones Vía
L2-BTB	*4K*	*8*	*20+8 KBytes*	*8 KBytes*
L2-BTB	*2 K*	*4*	*10+4 KBytes*	*2,75 KBytes*
L1-BTB	*1 K*	*4*	*5+2 KBytes*	*1 KBytes*
L1-BTB	*512*	*8*	*2,5+1 KBytes*	*0,75 KBytes*

Para *L1-BTB* se han analizado configuraciones de 1K y 512 entradas. Con 1K entradas se usan 4 vías en lugar de 8 vías porque la mejora es mínima, y no compensa el incremento en el tamaño del predictor debido a usar un bit más en cada predicción de vía. La Figura 4.26 muestra la cantidad de fallos de diferente tipo, para 4 combinaciones de las configuraciones de la tabla anterior. En comparación con la organización de un único nivel, la cantidad de fallos de predicción de vía en *L1-BTB* se ha visto reducida a casi un 45%. Este efecto es debido a salvar y recuperar las predicciones de vía entre el nivel *L1* y el nivel *L2*. La proporción para el resto de tipos de fallo no varía respecto a la que se daba con cada *BTB* funcionando por separado. La gran ventaja de la organización de 2 niveles es que ahora la penalización debida a fallos en *L1* es muy pequeña, pues la mayoría se resuelven en *L2*.

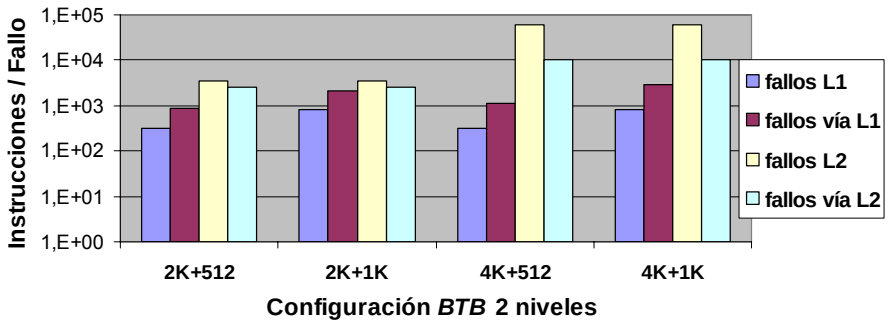


Figura 4.26. Fallos en el primer y segundo nivel de *BTB* para cuatro combinaciones de las configuraciones mostradas en la Tabla 4-1.

El otro resultado obtenido de la simulación es que los dos mecanismos de "prevención" de fallos funcionan de forma muy efectiva. Tras cada fallo real

en *L1-BTB* se inicia un acceso "preventivo" a *L2-BTB*, que se produce en paralelo con el acceso normal a *L1-BTB*, para reducir la penalización de un posible nuevo fallo en *L1*. La Figura 4.27 indica que este acceso "preventivo" es útil, es decir, sirve para reducir la penalización de 2 ciclos a 1 ciclo, entre un 63% y un 77% de los casos.

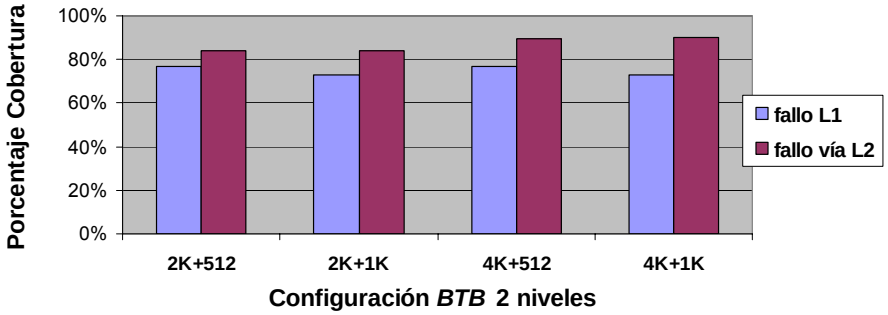


Figura 4.27. Cobertura (porcentaje) de los mecanismos de prevención para fallos en *L1-BTB* y fallos de predicción de vía en *L2-BTB*.

La "prevención" de fallos de vía en *L2-BTB* se activa después de un fallo de decodificación o de salto indirecto, y es aún más efectiva, ya que entre un 82% y un 90% de las veces se consigue evitar un nuevo fallo en la etapa de decodificación. Suponemos que el acceso a las etiquetas de *L2-BTB* es lento y no logra solaparse completamente con el acceso a *L1-BTB*, así que la penalización en estos casos "preventivos" aún sería de 1 ciclo.

Con *L1-BTB* de 512 entradas se produce, de media, un fallo real cada 324 instrucciones, y de éstos, el 77% penalizan un ciclo y el resto penalizan 2 ciclos. El número de fallos de predicción de vía es de 1 cada 880 instrucciones, y penalizan 2 ciclos. Al aumento de velocidad del predictor debido al uso de esta pequeña tabla de 512 entradas habría, por tanto, que añadir una media de 0,006 ciclos extra por instrucción debidos a los dos tipos de fallo en *L1*. Si *L1-BTB* tiene 1K entradas, hay una media de un fallo real cada 800 predicciones, de los que sólo el 27% penalizarían 2 ciclos, y una media de 1 fallo de vía cada 2100 predicciones. Esto supone añadir 0,0025 ciclos por instrucción predicha debidos a fallos en *L1*.

Con respecto a *L2-BTB*, una tabla de 4K entradas genera un fallo de predicción de vía cada 10300 instrucciones, de los cuales un 90% (1 cada 11.450) son detectados y corregidos en 1 ciclo de penalización. El 10% restante, un fallo de cada 103.000, igual que los fallos reales de *L2-BTB* (uno cada 59.000 instrucciones) se detectan en la etapa de decodificación, y tienen una penalización variable. De todos modos, cabe volver a señalar que en la fase de actualización se "descubren" los fallos "falsos", y que entonces el efecto que tienen estos fallos en la precisión de las predicciones de saltos condicionales e incondicionales es muy pequeño.

Si *L2-BTB* tiene 2K entradas y 4 vías, entonces los fallos de vía detectados y corregidos en un ciclo son uno cada 3100 instrucciones, los "falsos" fallos de *L2-BTB* son uno cada 16400 instrucciones, y los fallos reales de *L2-BTB* son uno de cada 3400 instrucciones. Hay que recordar que los fallos reales reducen un 8% la precisión para los saltos condicionales.

4.4.8 Conclusión

Las simulaciones han corroborado que la organización de dos niveles:

- con un tamaño total un poco mayor que el de una *BTB* de un único nivel, proporciona un comportamiento global casi idéntico en cuanto a la precisión de saltos condicionales e indirectos, aunque incrementa un poco el número de fallos que se detectan en la etapa de decodificación.
- proporciona una velocidad media de predicción (y un consumo energético medio) incluso más favorable al de una *BTB* de un único nivel con el tamaño de *L1-BTB*, pues se logra reducir el número de fallos de predicción de vía en *L1-BTB.*

En una sección posterior se analizará con detalle cual es el incremento en la velocidad del predictor. De momento, se puede concluir que aunque el diseño es complejo en cuanto al número de casos que pueden ocurrir, la mayoría de la complejidad del circuito está situada fuera de la ruta crítica. El

núcleo crítico del predictor funciona de forma autónoma, mientras que otros elementos adicionales van comprobando si genera resultados correctos, tratando de afectar lo menos posible a su funcionamiento.

4.5. Predicción de Índice

En este apartado se propone acelerar aún más la generación de la referencia al siguiente bloque básico con una tabla de índices a bloques básicos. En la siguiente figura se muestra el diagrama de bloques de la propuesta, un poco más detallada que en la Figura 4.9 previa. En ella se observa la tabla de índices (*index Table, iTbl*) y la tabla *L1-BTB*, ambas tablas indexadas simultáneamente por el mismo valor (ind_i).

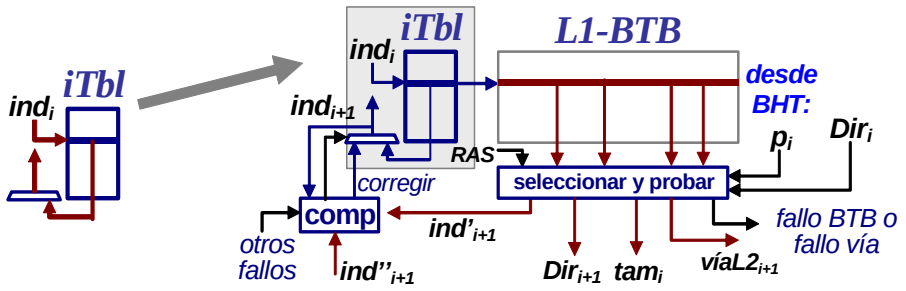


Figura 4.28. Aceleración del predictor con una tabla de predicción de índices (*iTbl*). A la izquierda se muestra el predictor de índices aislado, y a la derecha se muestra conectado con *L1-BTB* y con la lógica de corrección de la predicción de índice.

En este diseño se acentúa aún más el funcionamiento independiente del núcleo crítico del predictor, que tal como se ve en la parte izquierda de la figura, no puede ser más simple. *L1-BTB* está completamente fuera de la ruta crítica del predictor, y en este caso se decide que un mismo banco de memoria contenga tanto los datos como las etiquetas. *L1-BTB* ha de estar segmentada en dos etapas para que el predictor pueda funcionar a la mayor velocidad que posibilita la tabla de índices. El funcionamiento de *L1-BTB* es exactamente el mismo que en la organización de dos niveles descrita en la

sección anterior, y genera la dirección completa y el índice en *L1-BTB* del siguiente bloque básico, y el tamaño del bloque básico actual. En caso de que se detecte fallo en *L1-BTB*, tal vez por una predicción errónea de vía, se pasa la petición al segundo nivel de *BTB*.

El valor actual de índice (ind_i) permite acceder directamente a *iTbl* para obtener la predicción del índice al siguiente bloque básico (ind_{i+1}). *iTbl* es de acceso directo y muy pequeña, y por tanto muy rápida. Además, la selección del índice es muy rápida, pues hay que escoger entre sólo dos opciones: el índice leído de *iTbl* y el índice calculado tras detectar un error. La decisión entre las dos alternativas se debe tomar con suficiente antelación para que el caso más frecuente, que es seleccionar el contenido de *iTbl*, sufra el mínimo retardo de atravesar una puerta tri-estado (o similar).

La predicción de índice supone la predicción implícita (1) del resultado de una instrucción de transferencia de control (saltos condicionales e indirectos) y (2) de una vía en *L1-BTB*. Al predecir usando *iTbl* se usa sólo la referencia (índice) al bloque básico actual, y por tanto no se utiliza ninguna historia de saltos. En cambio, la predicción de *L1-BTB* (o *L2-BTB*), obtenida uno o varios ciclos más tarde, sí utiliza la historia previa de los saltos condicionales (por medio de *BHT*) y predictores específicos para saltos indirectos (*IJT* y *RAS*). La predicción de *L1-BTB* se compara con cada predicción realizada por *iTbl*. Si no hay coincidencia se debe corregir al predictor de índices, pues la predicción de *L1-BTB* tiene más probabilidades de ser correcta, lo cual comporta una penalización de tiempo y energía. Mostraremos que estos casos de predicción “lenta” son lo suficientemente infrecuentes como para que su penalización quede compensada por el aumento de velocidad de los casos “rápidos”.

La predicción de vía realizada por *L1-BTB* (implícita dentro de ind'_{i+1}, en la Figura 4.28) puede ser diferente a la predicción de vía realizada por *iTbl*, y en este caso también se le da prioridad a *L1-BTB*. La ocurrencia de estos casos, de todos modos, es bastante infrecuente.

4.5.1 Descripción de la Tabla de Índices

La Figura 4.17 muestra la simplicidad de la tabla de índices: cada entrada en la tabla es un apuntador a una nueva entrada. Se generan predicciones a bloques básicos con continuas lecturas a esta tabla y un mínimo mecanismo de selección. El índice almacenado en la tabla se puede interpretar también como un predictor de conjunto y un predictor de vía en *L1-BTB*.

Figura 4.29. Campo en la tabla de índices (*iTbl*) y nuevo campo en *L1-BTB*.

El otro cambio en el diseño, que mejora ligeramente las prestaciones, es añadir en *L1-BTB* un bit histéresis, ***h***, para controlar la actualización de las entradas de *iTbl*. Mostraremos su utilización en la sección sobre la actualización del predictor. Como el bit sólo es accedido en esta última fase, puede almacenarse en una tabla aparte, la misma que guarde la información de reemplazamiento de *L1-BTB* (bits LRU). Se ha analizado la posibilidad de guardar el bit en *L2-BTB*, pero los resultados mostraron poco beneficio.

4.5.2 Fase de Predicción

La fase de predicción consiste en lecturas sucesivas a *iTbl* a la máxima velocidad que permite su latencia de acceso y el mínimo retardo de la lógica de selección. El resto del funcionamiento del predictor es idéntico al de la organización de dos niveles con predicción de vía, excepto que ahora *L1-BTB* está segmentada en 2 etapas, y que se deben comprobar, y corregir si es necesario, las predicciones de índice. La siguiente figura muestra un ejemplo de forma esquemática.

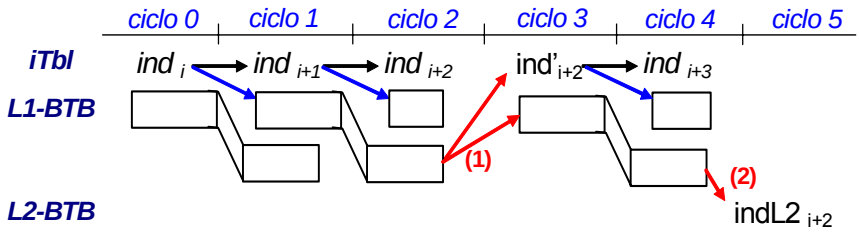


Figura 4.30. Se detecta el error de índice al comparar en el ciclo 2 el resultado de *L1-BTB* (ind'_{i+2}) con la predicción de *iTbl* al final del ciclo 1 (ind'_{i+2}). Inmediatamente se repara la operación de predicción para usar el resultado de *L1-BTB*.

En cada ciclo se genera un índice a *iTbl* y *L1-BTB* (ind_i, ind_{i+1}, ind_{i+2}, …). La comprobación de que el índice usado era el adecuado finaliza después de haber comenzado un acceso con el nuevo índice, a mitad del siguiente ciclo. Por ejemplo, el acceso a *iTbl* y *L1-BTB* para el bloque *i+1* se inicia en el ciclo 1 con ind_{i+1}, y el error de índice (de vía o de predicción) se detecta ya iniciado el ciclo 2 (flecha marcada con (1)). Al detectar el error, se utiliza como nuevo índice el proporcionado por *L1-BTB*, ind'_{i+2}, que reinicia inmediatamente la fase de predicción. Si se detectara un fallo en *L1-BTB*, como en el caso marcado con (2) al finalizar el ciclo 4, entonces se anulan los accesos a *iTbl* y *L1-BTB*, y se accede a *L2-BTB*, del modo explicado en la sección anterior.

4.5.3 Fase de Recuperación

La dirección del bloque básico a predecir proporciona el identificador de conjunto en *iTbl* y *L1-BTB*. El índice para reiniciar la operación del predictor se obtiene añadiendo la predicción de vía en *iTbl* y *L1-BTB*, calculada como se explica en la sección anterior para el diseño de dos niveles.

4.5.4 Fase de Actualización

Únicamente explicaremos cómo actualizar *iTbl*, ya que la actualización del resto de elementos no varía respecto al diseño de dos niveles. En cada predicción correcta de *iTbl* correspondiente a un salto condicional, se pone a

uno el bit de histéresis. En cada predicción incorrecta se lee el bit de histéresis y se pone a cero. Si el bit leído es un cero, entonces se modifica la predicción de *iTbl* para que apunte al índice alternativo. Para ello es necesario, o bien haberlo almacenado en la cola de predicciones, o bien obtenerlo en la fase de actualización de *L1-BTB* o de *L2-BTB*.

4.5.5 Resultados: Fallos de Predicción de Índice

En primer lugar, se ha simulado el predictor *iTbl* sin mecanismos específicos para saltos condicionales (*BHT* indexado con gshare), ni para saltos indirectos (*RAS* e *IJT*). La siguiente figura muestra la precisión de este predictor, con 512 y 4K entradas, indexado sin utilizar ninguna historia de saltos, sino sólo la dirección del bloque básico. A modo de comparación, se muestra también la precisión del predictor de saltos condicionales *gshare*, con 64K y 256K entradas, y con la longitud óptima para la historia de saltos. También se muestra la precisión para saltos de retorno de subrutina cuando no se usa la pila *RAS*, y del resto de saltos indirectos cuando no se usa y cuando se usa el predictor específico *IJT*.

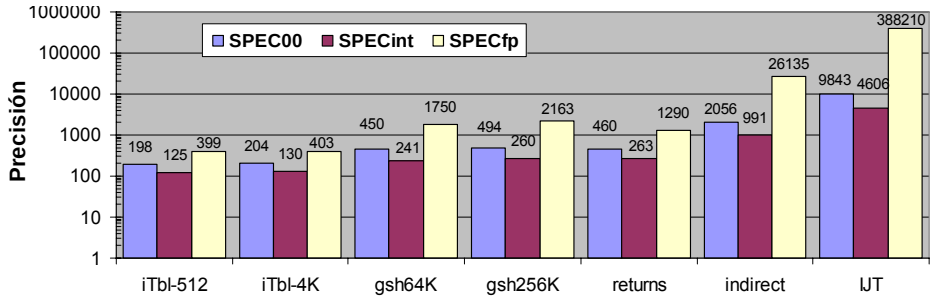


Figura 4.31. Precisión para saltos condicionales de un predictor *iTbl* sin historia (512 y 4K entradas), y de un predictor *gshare* con historia de longitud óptima (de 64K y 256K entradas). Precisión para retornos de subrutina sin pila *RAS*, y para el resto de saltos indirectos sin y con tabla *IJT*.

El predictor *iTbl* de 512 entradas, sin historia, genera, para SPEC00, un fallo de predicción aproximadamente cada 200 instrucciones. En cambio, el

predictor *gshare* de 256K entradas, con historia, falla más o menos cada 500 instrucciones. La diferencia es importante, ya que *iTbl* genera 1,5 fallos adicionales cada 500 instrucciones, y cada fallo de este tipo puede suponer una penalización de más de 20 ciclos. Es decir, se puede llegar a incrementar el CPI del procesador en más de 1,5·20/500 = 0,06.

El número de veces que los predictores *iTbl*-512 y *gshare*-256K difieren, y que darían lugar a un fallo de índice, debe oscilar entre 1,5 y 3,5 veces cada 500 instrucciones. Sin embargo, cada fallo de índice sólo comporta un retardo de un ciclo, que supone aumentar el tiempo medio por instrucción predicha en un máximo de 3,5/500 = 0,007 ciclos (y un valor medio cercano a 0,005). El mismo razonamiento sólo para SPECint00 da lugar a una penalización adicional por instrucción de 0,008. De nuevo, hay que tener en cuenta que la penalización afecta al ancho de banda del predictor, y que parte de esta reducción queda oculta por el esquema desacoplado del predictor.

Si las predicciones de *iTbl*-512 se salvan en *L2-BTB*, de 4K entradas, para preservarlas en los reemplazamientos, la precisión de *iTbl* mejorará muy poco (1 fallo cada 204 en lugar de cada 198 instrucciones). Por esta razón, se decide no dedicar dos bits adicionales en *L2-BTB* para esta función. Por la misma razón, usar una tabla *iTbl* de 1K entradas no disminuye prácticamente el número de fallos de predicción de índice (sólo el número de fallos reales y de predicción de vía en *L1-BTB*).

La precisión de las predicciones de saltos de retorno a subrutina sin utilizar *RAS* es de 1 fallo cada 263 instrucciones en SPECint00 (uno cada 1290 en SPECfp00), de modo que el número de errores de índice por esta causa causarían un incremento absoluto de menos de 0,004 ciclos por instrucción (menos de 0,0008 para SPECfp00).

Para la predicción del resto de saltos indirectos, *iTbl* siempre coincide con *L1-BTB*, ya que sólo hay una alternativa para escoger. Si se opta finalmente por usar la predicción de *IJT*, la penalización se produce tanto con tabla de índices como sin ella. La penalización en nanosegundos será muy similar a la

de antes, aunque medida en ciclos será algo mayor. Supondremos tres ciclos de penalización, que suponen incrementar el tiempo medio de predicción por instrucción en 0,003 ciclos.

La simulación de *iTbl* junto con *L1-BTB, L2-BTB, BHT*, *RAS* e *IJT* ha dado resultados mejores a los calculados anteriormente, debido a que algunos de los fallos de índice coinciden con fallos de *BTB* o con fallos de predicción de vía. Con *iTbl*-512, *L2-BTB*-4K, *gshare-256K*, *RAS* de 64 entradas, e *IJT-1K*, se ha detectado para SPECint00 un fallo de índice para saltos condicionales cada 143 instrucciones, un fallo para retornos de subrutina cada 297 instrucciones, y un fallo para saltos indirectos cada 1103 instrucciones.

Como la penalización de un fallo es de 1 ciclo, se tendría que añadir una penalización 0,01 ciclos por cada instrucción predicha. Para compensar esta penalización, como la media de tamaño de los bloques básicos es de menos de 7,6 instrucciones (para SPECint00), sería necesario aumentar la velocidad del predictor más de un 7,6% respecto a la propuesta de la sección anterior.

4.5.6 Conclusión

La tabla de índices contiene una parte del grafo de las transiciones predichas entre bloques básicos. Este grafo se recorre a alta velocidad, ya que la tabla de índices es muy pequeña, de acceso directo, y con un sencillo mecanismo de selección. Mientras se recorre este grafo, en paralelo, el resto de elementos del predictor, organizados de forma segmentada, proporcionan una predicción que usa más información (la historia de los saltos previos) para confirmar o para modificar la predicción de la tabla de índices.

La bondad de la propuesta radica en que la proporción de veces en que difieren las predicciones de *iTbl* y del predictor completo es pequeña, alrededor de una vez cada 100 instrucciones en SPECint00 (una vez cada 340 instrucciones en SPECfp00). Para compensar esta penalización, sería necesario aumentar la velocidad del predictor más de un 7,6% respecto a la

propuesta de la sección anterior. En breve presentaremos un análisis de los tiempos de funcionamiento de cada predictor. Antes, mostraremos resultados que analizan el uso con *BTB* de la técnica de anticipación que se usó con *BHT* (descrito en la sección 3.6).

4.6. Predicción Anticipada (*Ahead*) en *BTB*

La técnica de la predicción anticipada, propuesta para *BHT* en la sección 3.6, consiste en comenzar el acceso a la tabla antes de tener toda la información que en principio sería necesaria para generar el índice. Así que se construye un índice inicial con la información disponible en ese momento, y a medida que se obtiene el resto de información, se incluye en el mecanismo de selección del resultado final. Se mostró en la sección 3.6 que la aplicación de la predicción anticipada a *BHT* era muy efectiva, al menos anticipando hasta 4 ciclos, y que sólo se reducía la precisión alrededor de un 1%.

4.6.1 Diseño simple

Aplicar la misma técnica a *BTB* comporta más problemas. En el diseño sin anticipación, la dirección de un bloque básico *i* se utiliza para indexar *BTB* y encontrar la información del bloque básico *i*. Anticipar el acceso un ciclo supone usar la dirección del bloque básico *i-1* para acceder a *BTB* y, a mitad del acceso, utilizar la dirección del bloque básico *i*, recién obtenida, para seleccionar la información final. Es decir, para cada dirección de bloque básico se ha de almacenar en *BTB* los datos de sus bloques básicos sucesores. Si un bloque básico finaliza en un salto condicional tiene como máximo dos sucesores, pero si finaliza en un salto indirecto pueden tener múltiples sucesores (hasta 170 en algún caso puntual).

La forma más simple de tratar el problema de los saltos indirectos es detectarlos y anular el acceso anticipado, con el consiguiente retardo de

penalización. En realidad, sólo es necesario anular el acceso anticipado para saltos indirectos polimórficos.

El esquema más sencillo de anticipación consiste entonces en el que se muestra en la siguiente figura. Se inicia el acceso usando la dirección del bloque básico *i-1*, y si este bloque finaliza en un salto condicional se utiliza su predicción, p_{i-1}, para seleccionar la información del bloque básico *i* (BB_i). Para acelerar la recuperación del predictor en caso de error, se podría almacenar la predicción alternativa en la cola, pero el tamaño de los datos almacenados es demasiado grande (al menos 4 bytes por predicción) y no resulta eficiente hacerlo. Por tanto, cada vez que se produce un fallo de predicción de un salto condicional hay que añadir un ciclo extra para recuperar el predictor.

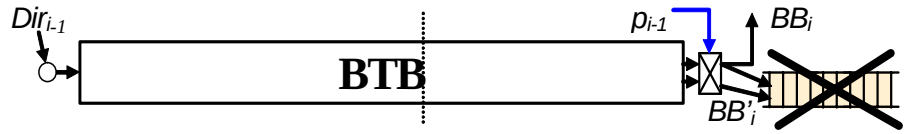


Figura 4.32. El acceso al bloque básico i se inicia usando la dirección del bloque básico i-1. La predicción del salto condicional del bloque básico *i-1* (p_{i-1}) se usa para seleccionar la información del bloque básico *i*.

Este diseño se puede generalizar para anticipar 2 o más ciclos. Sin embargo, hay que tener en cuenta que los requerimientos de memoria crecen exponencialmente. Anticiparse 1 ciclo requiere, para mantener la misma razón de fallos, una *BTB* con el doble de entradas que en el caso original. Anticiparse 2 ciclos requeriría el cuádruplo de entradas. Lo mismo se podría decir del consumo energético. Anticiparse 1 ciclo requiere leer el doble de información de la tabla en cada acceso, para finalmente quedarse con la mitad. Anticiparse 2 ciclos requeriría leer el cuádruplo de entradas.

4.6.2 Diseño complejo

Se ha comprobado experimentalmente que cada bloque básico tiene una media de 1,43 sucesores, o lo que es lo mismo, que alrededor del 57% de los

bloques básicos tienen un único sucesor. Es decir, no sería estrictamente necesario duplicar el tamaño de *BTB*, sino que habría que multiplicarlo por 1,43 para poder anticipar un ciclo los accesos y mantener la tasa de fallos. En ese caso se debe usar un esquema de indexación diferente, utilizando la predicción del salto condicional p_{i-1} como parte de la comparación de etiquetas en *BTB*. Se han realizado simulaciones de este último diseño, con anticipación de un ciclo, con p_{i-1} formando parte de la etiqueta, y una *BTB* de organización asociativa de 4 y 8 vías. Los resultados se muestran a continuación.

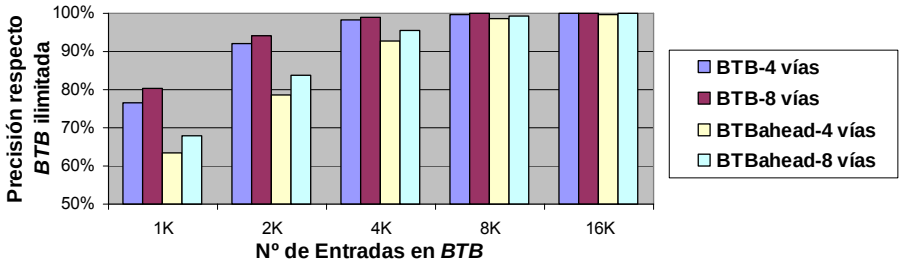


Figura 4.33. Porcentaje de la precisión para saltos condicionales comparada con el uso de una *BTB* ilimitada, para diversas capacidades de la tabla *BTB*, con 4 y 8 vías, y con anticipación (*BTBahead*) o sin ella.

Un resultado importante es que la asociatividad de 4 vías resulta insuficiente comparada con la asociatividad de 8 vías. La razón es que hay más conflictos debidos a usar p_{i-1} en la parte de la comparación de etiquetas. Para organizaciones de 8 vías, se puede corroborar que la anticipación de un ciclo supone aproximadamente unas prestaciones que corresponderían a una *BTB* sin anticipación de alrededor de un 70% del tamaño.

4.6.3 Conclusiones

La técnica de anticipación aplicada a *BTB* consigue aumentar la velocidad del predictor pero a costa de aumentar los requerimientos de memoria, el consumo energético y la complejidad del diseño. Además, se alarga el tiempo

de recuperación en caso de fallo de predicción y no se puede aplicar de forma eficiente a la predicción de saltos indirectos polimórficos.

La predicción de vía y de índice, en cambio, aumenta moderadamente los requerimientos de memoria, pero disminuye notablemente el consumo. La complejidad del diseño es también alta, pero no se penaliza la latencia de recuperación de los fallos de predicción.

Se han hecho pruebas preliminares mezclando anticipación y predicción de vía y se ha constatado la dificultad del diseño. Entre otras dificultades, las actualizaciones de predicción de vía se han de hacer con un ciclo (o varios) de retraso y se han de tener en cuenta los saltos indirectos que no utilizan predicción anticipada. En todo caso, la combinación de ambas ideas se puede tratar en una futura línea de estudio.

4.7. Latencia del Predictor

Se ha utilizado eCACTI para estimar el tiempo de acceso y el consumo energético de organizaciones de memoria de diferentes tamaños y número de vías. Esta herramienta no considera el problema de escalabilidad en el retardo de los hilos, y usa un simple factor de escala para generar, a partir de una tecnología de 800 nm, el tiempo de acceso de todas las configuraciones con una tecnología de 70 nm. Usando los datos presentados por Agarwal et al. en [AHK+00], se han estimado los tiempos de acceso considerando un escalado no ideal del retardo de transmisión por los hilos. La Figura 4.34 y la Figura 4.35 muestran estos tiempos de acceso. La Figura 4.36 muestra el consumo energético estimado por eCACTI.

La estimación de tiempos debe considerarse con reservas, debido al margen de error de eCACTI y del modelo de escalado de los retardos. Estos datos servirán de guía en las conclusiones, configurando una banda de valores máximos y mínimos, para que las conclusiones dependan lo menos posible de posibles errores en estas estimaciones.

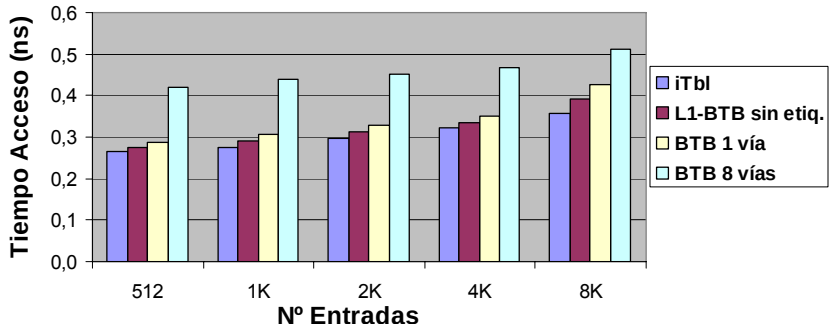


Figura 4.34. Tiempo de Acceso calculado con eCACTI (tecnología de 70 nm) para las organizaciones de *BTB* consideradas, en función del número de entradas.

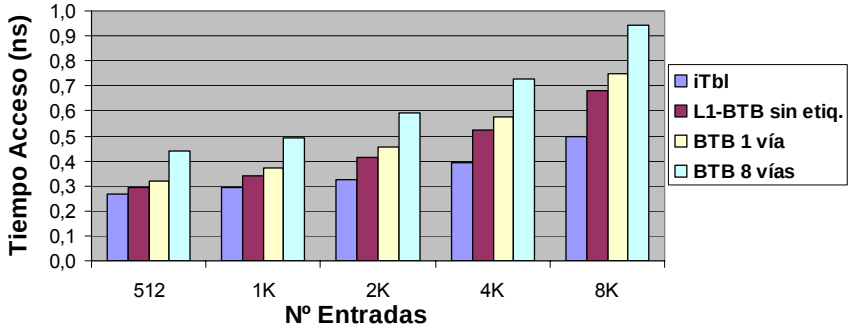


Figura 4.35. Tiempo de Acceso calculado con eCACTI (70 nm) y extrapolado para considerar la no escalabilidad de los retardos de los hilos de comunicación.

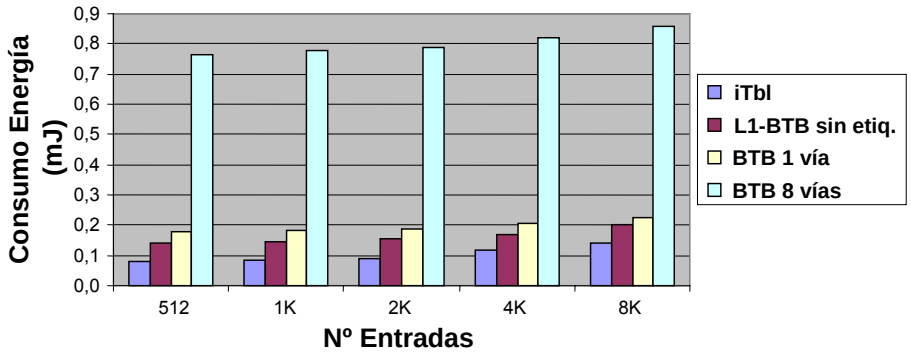


Figura 4.36. Energía consumida obtenida con eCACTI (70 nm) para las organizaciones de *BTB* consideradas, en función del número de entradas.

Con los resultados mostrados en las figuras anteriores se puede estimar la latencia de los predictores analizados en las secciones anteriores. La Tabla 4-2 proporciona la estimación del tiempo de acceso de la tabla que constituye la ruta crítica de cada predictor. La información se muestra separada entre predictores con capacidad para 4K bloques básicos y una asociatividad de 8 vías, y predictores capaces de almacenar 2K bloques básicos y con 4 vías.

Tabla 4-2 Tiempo de ciclo, consumo energético, tamaño en KBytes y frecuencia de casos de penalización por instrucción ejecutada para varias organizaciones de *BTB: (1) organización sin predicción de vía, (2) predicción de vía con un nivel de BTB, (3) predicción de vía con 2 niveles de BTB, (4) predicción de índice con 2 niveles BTB*

(A) Configuraciones con 4 K entradas y 8 vías

configuración	**Tiempo eCACTI**	**Tiempo Escalado**	**Consumo Energía**	**Tamaño**	**Fallos Índice**	**Fallos Vía**	**Fallos Decodif.**	**Δ Fallos Cond.**
(1) BTB 4K, 8 vías	*0.468*	*0,727*	*0,82*	*28 KB*	---	---	*1 cada 90000*	*1 cada 49000*
(2) 1-Nivel BTB(4K, 8 vías)	*0.332*	*0,551*	*0,204*	*31 KB*	---	*1 cada 10300*	*1 cada 90000*	*1 cada 49000*
(3) L1(1K, 4vías) L2(4K, 8vías)	*0,291*	*0,339*	*0,16*	*44 KB*	---	*1 cada 790*	*1 cada 76000*	*1 cada 49000*
(3) L1(512, 8vías) L2(4K, 8vías)	*0,276*	*0,293*	*0,147*	*40,3 KB*	---	*1 cada 340*	*1 cada 76000*	*1 cada 49000*
(4) iTbl(1K) L1(1K) L2(4K)	*0,275*	*0,293*	*0,231*	*45,2 KB*	*1 cada 158*	*1 cada 790*	*1 cada 76000*	*1 cada 49000*
(4) iTbl(512) L1(512) L2(4K)	*0,265*	*0,266*	*0,223*	*41 KB*	*1 cada 154*	*1 cada 340*	*1 cada 76000*	*1 cada 49000*

(B) Configuraciones con 2 K entradas y 4 vías

configuración	**Tiempo eCACTI**	**Tiempo Escalado**	**Consumo Energía**	**Tamaño**	**Fallos Índice**	**Fallos Vía**	**Fallos Decodif.**	**Δ Fallos Cond.**
(1) BTB 2K, 4 vías	*0.468*	*0,727*	*0,82*	*14 KB*	---	---	*1 cada 5500*	*1 cada 7700*
(2) 1-Nivel BTB(2K, 4 vías)	*0.317*	*0,465*	*0,188*	*15,2 KB*	---	*1 cada 2625*	*1 cada 5500*	*1 cada 7700*
(3) L1(512, 8vías) L2(2K, 4vías)	*0,276*	*0,293*	*0,147*	*21 KB*	---	*1 cada 340*	*1 cada 4200*	*1 cada 7700*
(4) iTbl(512) L1(512) L2(2K)	*0,265*	*0,266*	*0,223*	*21,7 KB*	*1 cada 154*	*1 cada 340*	*1 cada 4200*	*1 cada 7700*

Las tablas muestran el tiempo obtenido con eCACTI y el que considera un escalado no lineal de la latencia de las líneas de interconexión. Estos tiempos no incluyen el retardo de la lógica de generación de índices ni de selección

final. A modo comparativo, se muestra el consumo energético debido a los accesos al primer nivel de las tablas durante la fase de predicción, sin considerar el consumo de fallos en *L1* ni de la fase de actualización.

La contrapartida a la mayor velocidad del predictor de las configuraciones más complejas es la ocurrencia de casos que producen retardos internos (ciclos de penalización). La tabla anterior muestra la frecuencia de estos casos. Los fallos de predicción de índice provocan un ciclo de retardo. Aunque los fallos de predicción de vía provocan 1 ó 2 ciclos de retardo, ambos casos se promedian y se muestra una frecuencia de fallos que supone que siempre se penalizan 2 ciclos. Finalmente, se muestra la frecuencia de fallos de decodificación y de predicción de salto, aunque estos valores no se usarán hasta una sección posterior.

Los tiempos indicados en la tabla anterior no incluyen el retardo de la lógica que selecciona el índice final. La siguiente figura muestra los dos casos diferentes a considerar. En el caso (a), la información sobre el tipo del bloque básico, leída de *BTB*, sirve para seleccionar un índice entre 4 (ó 3) posibles. A los casos de la Tabla 4-2 etiquetados con los números (1), (2) y (3) se les debería añadir el retardo de esta lógica. En el caso (b) de la figura, la decisión para seleccionar entre dos posibles índices no depende de los datos leídos de la tabla. Por tanto, a los casos de la Tabla 4-2 etiquetados con el número (4) se les debería añadir únicamente el retardo de atravesar un demultiplexor simple.

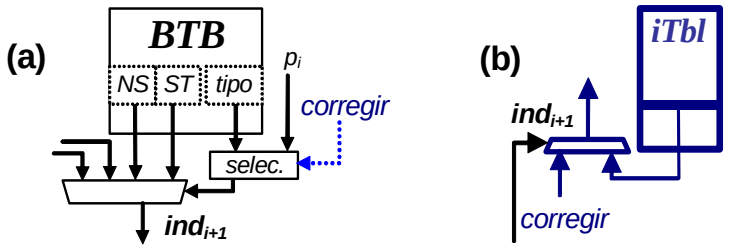


Figura 4.37. Circuito de generación del índice siguiente en los esquemas: (a) sin y con predicción de vía (1 y 2 niveles), y (b) con predicción de índice

Es complejo evaluar cuantitativamente el valor absoluto y la diferencia entre las latencias de los dos diseños de la figura anterior. En el análisis consideraremos dos casos: ignorar el retardo de la lógica de selección, o suponer que el retardo de la selección es de 0,05 ns en las configuraciones (1-3) y 0,02 ns en las configuraciones (4). Estos valores suponen un 20% y un 7% respectivamente del tiempo de acceso a una tabla de 512 bytes.

La siguiente figura muestra el porcentaje de incremento de la latencia de las configuraciones de predictor de 4K entradas, respecto a la latencia de la configuración más rápida, es decir, ***iTbl*(512), *L1*(512), *L2*(4K)**.

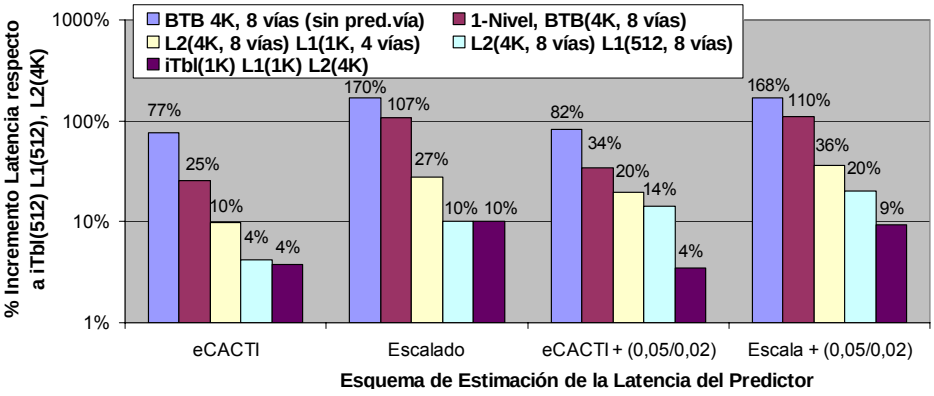


Figura 4.38. Porcentaje de incremento de la latencia de ciertas configuraciones del predictor (Tabla 4-2), respecto a configuración más rápida. La latencia se calcula con cuatro esquemas diferentes.

Los resultados muestran que si se considera la mala escalabilidad de la latencia de los hilos, las diferencias en la latencia de una configuración a otra aumentan de forma considerable. Considerar el tiempo de la lógica de selección es importante para decidir si el esquema de predicción de índice mejora la velocidad entre un 4% y un 10% respecto al esquema de predicción de vía con dos niveles, o si la mejora oscila entre el 14% y el 20%.

En resumen, las estimaciones aproximadas realizadas en este apartado muestran que la reducción de la latencia al usar la predicción de vía es importante: entre el 22% y el 30% con el esquema de un nivel, y entre el 40%

y el 60% con dos niveles. La reducción adicional de la latencia al predecir índices es dependiente de la proporción de tiempo que consuma la lógica de selección, y se ha estimado entre el 4% y el 17%. A continuación se analizará la influencia que tienen las penalizaciones producidas por los fallos de predicción de vía y de índice, y el efecto en el rendimiento del procesador.

4.8. Resultados

Primero se analiza el ancho de banda potencial del predictor y luego se estudia el rendimiento obtenido en función de ese ancho de banda potencial.

Los resultados de ancho de banda del predictor se han obtenido simulando cada predictor de forma aislada, es decir, suponiendo que las predicciones se consumen y los predictores se actualizan de forma inmediata. El cómputo final del ancho de banda considera tanto la latencia del predictor, estimada en la sección anterior, como la ocurrencia de fallos de predicción de vía y de índice, que suponen 1 ó 2 ciclos de penalización. También se modela una penalización de 2 ciclos cada vez que el predictor de saltos indirectos corrige a la predicción de *BTB*. En cambio, los fallos de decodificación y los fallos de predicción de saltos no afectan al ancho de banda. Así, el ancho de banda medido es un límite superior. En el capítulo 2 se analizó la relación entre el ancho de banda potencial y el rendimiento final del procesador.

Los resultados de rendimiento se han obtenido simulando los predictores integrados dentro de una microarquitectura realista del procesador. La Tabla 4-3 muestra las características más importantes del procesador simulado. Los datos no mostrados en la tabla se pueden encontrar en la sección 3.3. Los fallos de decodificación de saltos penalizan un mínimo de 4 ciclos, y los fallos de predicción de saltos un mínimo de 16 ciclos. El predictor está desacoplado de la caché de instrucciones y se utiliza un mecanismo de prebúsqueda de instrucciones dirigido por el predictor (ver sección 3.4). La caché proporciona hasta 2 bloques de 4 instrucciones por ciclo.

Tabla 4-3 Características más importantes de la microarquitectura del procesador.

Unidad de Búsqueda	Núcleo de Ejecución	Sistema de Memoria
Desacoplada: FTQ: 16 entradas **iCache**: 2 bloques x 4 instr. x ciclo **BHT:** *gshare* 256K entr., 3 ciclos anticipación, **IJT**: 1K entradas, **RAS**: 64 entradas **Penalización mínima fallo**: decod. (4 ciclos), ejec. (16 ciclos)	**Decodificación**: 6 instr. /ciclo **Ejecución**: 6 instr. / ciclo **Reorder Buffer**: 240 entradas **Cola Instrucc.**: 120 entradas **Cola de Load's**: 120 entradas **Cola de Store's**: 90 entradas	**iCache**: 8KB, 2 ciclos (**Prebúsqueda**) **dCache:** 16KB, 2 ciclos **L2-Cache**: 512KB, + 6 ciclos **L3-Cache**: 16 MB, + 32 ciclos **Memoria**: + 120 ciclos

4.8.1 Ancho de Banda del Predictor

La Figura 4.39.a muestra el ancho de banda de predicción para cada configuración de predictor, y la Figura 4.39.b muestra la latencia media por predicción, es decir, el valor inverso al ancho de banda, pero indicando la parte de incremento de esta latencia media debida a los fallos de predicción de índice y a los fallos de predicción de vía.

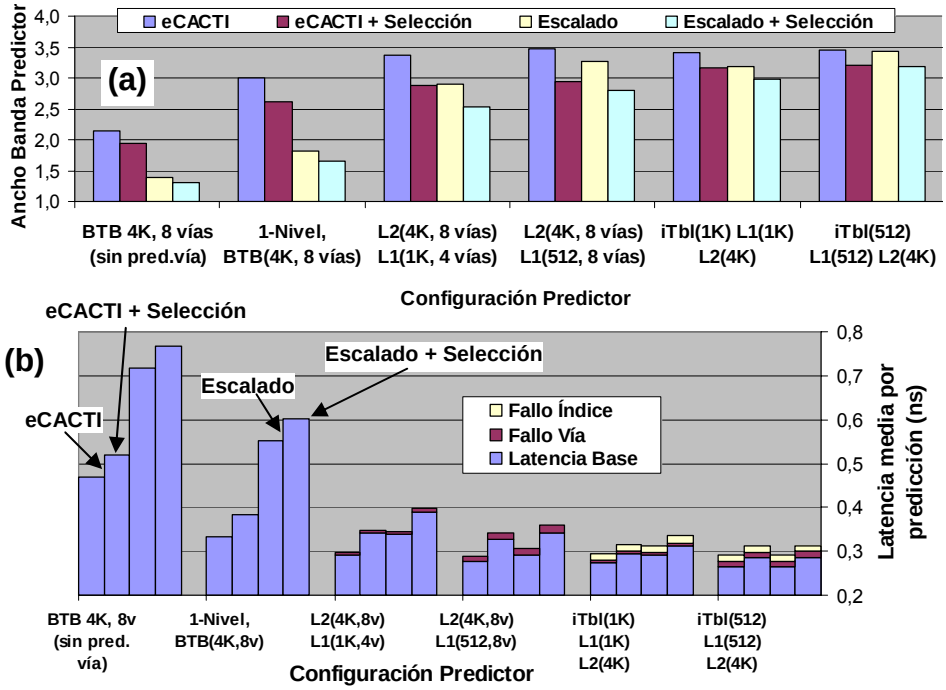


Figura 4.39. (a) Ancho de banda del predictor (predicciones por nanosegundo). (b) Latencia media del predictor mostrando el efecto de los fallos de predicción de vía e índice. Para 6 configuraciones (Tabla 4-2), y 4 esquemas de cálculo de la latencia

Un resultado importante es que la configuración con predicción de vía de 2 niveles mejora de forma sustancial, sea cual sea el esquema utilizado para estimar la latencia de los predictores, a la configuración base sin predicción de vía (entre un 50% y un 134% de incremento en el ancho de banda). El esquema de dos niveles también mejora a la configuración con un solo nivel (entre un 12% y un 80% de incremento en el ancho de banda). Cabe recordar que el tamaño de la configuración de dos niveles (sin contar *BHT*, *IJT* y *RAS*) es alrededor de un 40% mayor que el tamaño de la configuración base (pero consume menos de la cuarta parte de energía en cada acceso), y que es alrededor del 30% mayor que el de una configuración de un único nivel.

En cambio, la configuración de dos niveles con predicción de índice no siempre supera a la configuración que sólo realiza predicción de vía. El resultado depende del peso que tenga la lógica de selección en el tiempo de ciclo del predictor, y de la mayor o menor diferencia entre el tiempo de acceso a una tabla de 512 entradas de 9 bits por entrada y a una tabla de 512 entradas de 21 bits por entrada. Por ejemplo, si no se considera el tiempo de la lógica de selección, y usando los resultados de tiempo de acceso proporcionados por eCACTI, el esquema de predicción de índice es alrededor de un 0,6% peor que el esquema sin predicción de índice. En el lado contrario, la ganancia sería algo superior al 14%.

En todos los casos, usar una tabla de primer nivel de 512 entradas da mejor resultado que una tabla de 1K entradas. No se presentan resultados para una tabla de 256 entradas, porque provoca un importante incremento en la ocurrencia de fallos reales y fallos de vía, que no se compensa con la pequeñísima reducción en el tiempo de acceso que predice eCACTI.

4.8.2 Rendimiento del Procesador

Las latencias de los predictores estimadas por eCACTI en la sección anterior son útiles para comparar los predictores entre sí. Sin embargo, convertir el valor en nanosegundos a un valor en ciclos de reloj del procesador depende

del grado de profundidad de la segmentación, y requiere un análisis del diseño en relación a la tecnología de implementación. En lugar de asignar a cada configuración del predictor una latencia fija en ciclos de reloj, se presentan resultados para un amplio rango de latencias.

La Figura 4.40 muestra el rendimiento del procesador utilizando el predictor de referencia con *BTB* sin predicción de vía. Las cinco curvas de la figura, que corresponden a diferentes casos de simulación, están numeradas para facilitar su identificación.

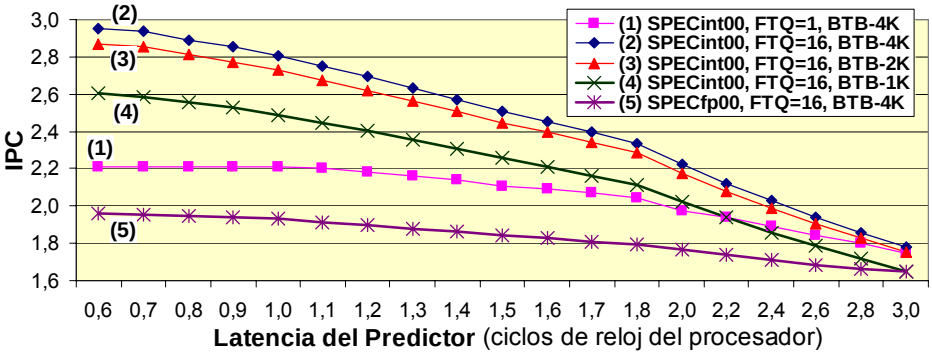


Figura 4.40. IPC para la microarquitectura descrita en la Tabla 4-3 con el predictor de referencia usando una *BTB* de 4K, 2K ó 1K entradas sin predicción de vía, para diferentes latencias de predicción.

El predictor funciona con un reloj diferente al del resto del procesador, y la frecuencia del primero no necesita ser un múltiplo o divisor exacto de la frecuencia del segundo. La cola *FTQ* permite desacoplar estas diferentes velocidades. Se simula una cola *FTQ* de 16 entradas y también una cola con capacidad para una única predicción, con objeto de mostrar el efecto de desacoplar el predictor. Se presentan resultados con *BTBs* de capacidad para 4K, 2K y 1K bloques básicos. Las curvas 1-4 de la figura muestran el promedio de resultados para los programas SPECint00, mientras que la curva 5 muestra resultados para los programas SPECfp00.

Las tres conclusiones fundamentales que se extraen de los resultados son que (1) la organización desacoplada es clave para aprovechar la mayor

velocidad del predictor, que (2) reducir el número de entradas en *BTB* para aumentar la velocidad del predictor también incrementa el número de fallos de predicción y entonces reduce de forma importante el rendimiento, y que (3) disminuir la latencia del predictor aumenta claramente el rendimiento del procesador. A continuación discutiremos con más detalle estos tres resultados, en el mismo orden en que se han nombrado.

Evaluación de la Organización Desacoplada

Comparando las dos primeras curvas de la Figura 4.40 se puede observar el efecto de usar una organización desacoplada. Por ejemplo, fijando la latencia del predictor a un ciclo de reloj, el esquema desacoplado incrementa el IPC un 27% (para SPECint00). Si se fija la latencia a dos ciclos, el incremento es del 12,5%. La razón de esta mejora es que la organización desacoplada amortigua el problema creado por bloques básicos demasiado pequeños y por fallos en la caché de instrucciones. El beneficio es mayor cuanto mayor es el ancho de banda del predictor.

En el sistema acoplado, cada vez que se predice un bloque básico de menos de 6 instrucciones, la etapa de decodificación queda desaprovechada parcialmente. Además, al leer los bloques básicos en parejas de líneas de caché de como máximo 4 instrucciones, se produce fragmentación si una de las líneas de caché contiene una única instrucción válida. Para evitar más problemas de fragmentación, si en un ciclo se obtienen 7 ó 8 instrucciones válidas, las que no se pueden enviar a decodificar inmediatamente se almacenan para el siguiente ciclo.

La organización desacoplada permite compensar los bloques pequeños con bloques de tamaño mayor, leyendo simultáneamente dos líneas de caché en direcciones que pueden ser no consecutivas. Permite además aprovechar la reducción de la latencia del predictor a menos de un ciclo de reloj.

El tamaño de la caché de instrucciones es relativamente pequeño (8KBytes), y en algunas aplicaciones genera un porcentaje significativo de

fallos. Se ha escogido este tamaño para evaluar la capacidad que tiene la organización desacoplada junto con el mecanismo de prebúsqueda para reducir el problema de los fallos de caché. Una alta velocidad de predicción permite anticipar los fallos de caché, y pedir las instrucciones al segundo nivel de memoria antes de que la etapa de decodificación consuma las instrucciones de las predicciones pendientes. Esta capacidad de anticiparse al fallo le permite ocultar parte de la latencia del acceso a la caché de segundo nivel. Pero quizás aún más importante, en caso de encontrar una secuencia de fallos consecutivos (un caso relativamente frecuente), como el predictor no se bloquea ante un fallo de caché, se pueden enviar las peticiones a la caché de segundo nivel (que debe estar segmentada) a un ritmo suficiente para que proporcione un bloque de hasta 8 instrucciones consecutivas cada ciclo.

Efecto del Tamaño del Predictor

Las curvas 2, 3 y 4 de la Figura 4.40 muestran que reducir el número de entradas en *BTB* reduce de forma considerable el rendimiento, ya que incrementa la proporción de fallos de decodificación y, más importante, de fallos de predicción de saltos. Por ejemplo, si la latencia de un predictor de 4K entradas en *BTB* es de 1,4 ciclos, habría que reducir la latencia a menos de 0,7 ciclos para que el beneficio de un predictor más rápido se compensase con el incremento del número de fallos.

Efecto de la Latencia en el Rendimiento

Reducir la latencia del predictor sin aumentar el número de fallos de predicción, y aumentando muy poco el número de fallos de decodificación, como consiguen las propuestas de este capítulo, sí que es claramente beneficioso para el rendimiento. Las cinco curvas de la Figura 4.40, en especial la segunda curva, muestran que reducir la latencia del predictor, al menos hasta 0,5 ciclos, aumenta de forma considerable el rendimiento. Esta

observación ya fue apreciada en el trabajo de Jiménez et al. [JiKL00]. La mejora en el rendimiento depende de la aplicación y del valor de la latencia del que se parta originalmente. En la curva nº 2, reducir la latencia de 3 a 2 ciclos (un 33%) incrementa el rendimiento para SPECint00 un 25%. Reducir la latencia de 1,5 a 1 ciclo (también un 33%) o reducirla de 1 a 0,6 ciclos (un 40%) incrementa el rendimiento un 12% o un 5%, respectivamente.

Por otro lado, la curva nº 5 muestra los resultados del mismo experimento que el de la curva nº 2, pero para SPECfp00. Puesto que estos programas contienen bloques básicos mucho más largos, el predictor es capaz de proporcionar una media de 16,6 instrucciones por predicción (en lugar de 7,6 en el caso de SPECint00). La mayor anchura de las predicciones compensa una menor velocidad de predicción. Adicionalmente, la alta precisión de la predicción de saltos en los programas SPECfp00 y el menor rendimiento que se obtiene con ellos (ya que muchos están limitados por la latencia a los datos de memoria) hace que la latencia del predictor tenga un efecto menor en el rendimiento. En definitiva, sólo se obtienen ganancias importantes al acelerar el predictor si se parte de una alta latencia inicial.

4.9. Trabajo relacionado

Este trabajo se construye sobre la propuesta de arquitectura desacoplada de la unidad de búsqueda [ReAC99], y sobre los esquemas de predicción de vía y de línea aplicados a la caché de instrucciones [John91], [CaGr95], [Yung96]. Una idea inherente a la predicción de línea y vía es la jerarquía de predictores, discutida en [JiKL00].

4.9.1 Arquitectura Desacoplada de la Unidad de Búsqueda

Reinmann et al. [ReAC99] presentan, junto a su propuesta de arquitectura desacoplada, un predictor de dos niveles que denominan *FTB* (*Fetch Target*

Buffer), y que es muy similar al predictor de bloques básicos, *BBP*, que se ha usado como referencia. La contribución de este capítulo es aplicar a *BBP* la técnica de predicción de vía y de predicción de línea.

FTB ignora los saltos condicionales hasta que saltan por primera vez, de modo que los saltos que no han saltado nunca están incluidos dentro de la unidad de predicción que denominan *Fetch Block*. Para simplificar el análisis, hemos optado por no aplicar esta política. En el capítulo siguiente se propone un esquema más ambicioso para aumentar la anchura de las predicciones.

4.9.2 Predicción de Línea para la Caché de Instrucciones

En [John91], [CaGr95] y [Yung96] se propone sustituir las direcciones de memoria destino de los saltos, contenidas en *BTB*, por apuntadores a líneas de la caché de instrucciones. Las diferencias entre las tres propuestas residen en si los apuntadores se guardan en la caché o no, y en cómo se hacen corresponder los apuntadores con las instrucciones de salto que los utilizan. Antes de analizar estas diferencias, describiremos las similitudes.

El apuntador que sustituye a una dirección de salto identifica la línea de caché donde se encuentra la instrucción a la que saltar. Predice tanto el conjunto como la vía en la que se encuentra la línea, y permite iniciar un acceso a la caché antes de decodificar las instrucciones que contenían la línea de caché actual (y de calcular las posibles direcciones de salto de estas instrucciones). Aunque las líneas de caché se organicen con un esquema asociativo por conjuntos, el apuntador permite realizar un acceso directo, rápido y de poco consumo energético.

El gran problema de tener el predictor acoplado a la caché es que cada predicción de flujo de control se asocia a una línea de caché que, al contrario que los bloques básicos, no tienen una correspondencia natural con las instrucciones de transferencia de control. Es frecuente que la instrucción destino de un salto no sea la primera dentro de la línea de caché, y por tanto

el apuntador deberá indicar su posición relativa dentro de la línea, y ocupará más bits. Un problema más grave es que una línea puede contener varias instrucciones de salto, incluso en el peor caso todas las instrucciones de la línea podrían ser de transferencia de control. Esto complica enormemente la casuística de la selección del apuntador, y requiere almacenar múltiples tipos de salto y apuntadores de salto ligados a la misma línea de caché.

Un sistema desacoplado en el que el predictor genera direcciones completas de bloques básicos (en lugar de apuntadores a caché) facilita que la caché de instrucciones tenga una latencia mayor que el predictor y aún así se puedan verificar y anticipar los fallos en la caché. Generar las direcciones completas permite también verificar los fallos de *BTB*, de predicción de vía y de predicción de índice, reduciendo considerablemente su penalización, y permite implementar una jerarquía de dos niveles de *BTB*.

El diseño propuesto por Johnson [John91] integra la *BTB* dentro de la propia caché de instrucciones, de forma que cada línea de caché contiene un apuntador a la siguiente línea. La ventaja del diseño es que se aprovechan las mismas etiquetas para la *BTB* y la caché. El problema es que si una línea contiene dos o más saltos, habrá polución al actualizar el único apuntador.

Calder y Grunwald [CaGr95] analizan el uso de varios apuntadores por línea de caché. Sin embargo, reservar una cantidad fija de memoria en cada línea no es eficiente (a veces es excesiva y otras insuficiente) y proponen usar una tabla de apuntadores separada, a la que denominan *NLS* (*Next Line and Set*). *NLS* es de acceso directo y no contiene etiquetas, lo que permite que tenga muchas entradas, distribuidas entre todos los saltos del programa, y aún así ser muy rápida. Debe, eso sí, contener el tipo del salto para decidir, por ejemplo, si usar la pila *RAS* o la línea consecutiva (*fall-through*), cuya predicción de vía se puede almacenar en la propia caché de instrucciones.

Yung [Yung96] analiza el uso de varios predictores de vía por línea de caché y concluye que el uso de un predictor por instrucción es una opción adecuada en una caché con dos vías (predictores de vía de 1 bit). También

utiliza la estrategia de predecir el índice a la siguiente línea de instrucciones sin verificar el tipo de las instrucciones ni utilizar la historia de saltos, lo cual correspondería con el predictor de índice de este capítulo.

4.9.3 Jerarquía de Predictores

La idea de acelerar la velocidad del predictor utilizando una jerarquía de predictores ha sido discutida en [JiKL00], aunque había sido implementada previamente en varios diseños comerciales como el Alpha 21264 [Kess99], el UltraSPARC [Yung96], o el Intel Itanium [ShAr00]. Un predictor pequeño y simple proporciona predicciones a alta velocidad, mientras que un segundo predictor, más lento y grande, pero a la vez más preciso, corrige las predicciones del primero. Aunque las correcciones suponen una penalización, ésta es mucho menor que la que produciría un fallo de predicción. Así, el aumento de la velocidad de predicción junto a la menor incidencia de fallos de predicción, que tienen una alta penalización, sobrepasa con creces a la frecuencia de las correcciones multiplicada por su pequeña penalización.

4.10. Conclusiones

En este capítulo se han analizado dos estrategias para acelerar la velocidad de un predictor de bloques básicos en una arquitectura desacoplada de la unidad de búsqueda de instrucciones. La latencia del predictor depende de varios aspectos, pero uno de los más importantes es el tamaño y su esquema de asociatividad, que también determinan su precisión. Reducir la latencia limitando el tamaño total del predictor supone también reducir su precisión, y por tanto aumentar las penalizaciones debidas a fallos de predicción. Las técnicas analizadas utilizan una moderada cantidad de memoria para acelerar la velocidad de predictor sin disminuir su precisión. Aunque generan retardos internos al propio predictor, éstos son pequeños y poco frecuentes.

El resultado final es un mayor ancho de banda de predicción, que permite mejorar de forma significativa el rendimiento del procesador.

La técnica de predicción de vía aplicada a una jerarquía de dos niveles aumenta la velocidad de predicción entre un 70% y un 145%. Considerando los retardos por fallo de predicción de vía, la velocidad media del predictor aumenta entre un 50% y un 130%. El coste de esta mejora es una mayor complejidad de diseño y un moderado aumento en la memoria, pero no hay una reducción significativa en la precisión final del predictor.

La predicción de vía con dos niveles permite simplificar mucho el núcleo del predictor, aunque necesita aún ser “alimentado” con la información del tipo del bloque básico y con la predicción de saltos condicionales de una *BHT* segmentada, con predicción anticipada. La técnica de predicción de índice es un paso adicional que reduce al mínimo la complejidad del núcleo del predictor: una tabla que proporciona un índice que se usa directamente para volver a acceder a la misma tabla. Funciona de forma autónoma, sin necesidad de disponer ni del tipo de bloque básico ni de la predicción de saltos condicionales. El grafo con las transiciones más frecuentes entre bloques básicos se recorre a gran velocidad sin usar información adicional. En paralelo, se usa toda la información disponible (historia de saltos previos, tipo del bloque básico, …) para generar predicciones bastante más precisas, pero más lentamente. Si la predicción “lenta” no coincide con la predicción “rápida” se corrige al predictor. Aunque el diseño es complejo en cuanto al número de casos que pueden ocurrir, toda la complejidad del circuito está situada fuera de la ruta crítica.

Estimaciones conservadoras sobre la reducción de latencia del predictor comparada con el esquema sin predicción de índice indican una ganancia en el ancho de banda del predictor de entre un 0% y un 15%. En todo caso, la precisión del predictor no disminuye, los requerimientos de memoria y de consumo energético aumentan moderadamente, y se reduce la presión sobre el predictor de saltos condicionales, que ya no requiere un esquema de predicción anticipada.

Se han analizado los inconvenientes de aplicar la predicción anticipada a *BTB*, básicamente una mayor ineficiencia en el uso de memoria y en el consumo energético, y una mayor complejidad, sobre todo para poder manejar la predicción de saltos indirectos. Se aumenta además la latencia para recuperarse de un fallo de predicción.

La contribución básica del capítulo es aplicar las técnicas de predicción de vía y de índice a un predictor de bloques básicos, en lugar de a un predictor de líneas de caché, lo cual simplifica la casuística y la gestión de la memoria dedicada a las predicciones de vía e índice, pues sólo se ha de considerar un único salto por predicción. La sinergia entre la predicción de vía y la jerarquía de dos niveles de *BTB* permite que mediante el uso de apuntadores cruzados se reduzca al mínimo la penalización por los fallos en el primer nivel sin tener que verificar etiquetas. Al salvar la información de predicción de vía entre *L1* y *L2*, también se consigue aumentar la efectividad de esta estrategia.

Entre los detalles del diseño que suponen aportaciones originales cabe resaltar el mecanismo de actualización de las predicciones de vía en la pila *RAS*, la propuesta para recuperar la predicción de vía alternativa tras un fallo de predicción de salto condicional, y la política de “protección” ante fallos encadenados tras un fallo de *BTB* o de salto indirecto.

Una ventaja adicional de la predicción de vía es la reducción del consumo energético del propio predictor. Las técnicas descritas en este capítulo se pueden utilizar para optimizar el consumo energético en lugar de para aumentar la frecuencia del predictor, aprovechando que se reduce la complejidad en la ruta crítica.

Para las aplicaciones SPECfp00, la reducción de la latencia del predictor tiene menor efecto, puesto que en cada predicción se proporcionan bloques básicos de un tamaño medio de más de 16 instrucciones. En el siguiente capítulo se analiza la posibilidad de aumentar la anchura de las predicciones mediante la predicción de trazas de instrucciones, una unidad que puede contener varias instrucciones de salto, y por tanto, múltiples bloques básicos.

5. Aumentando la Anchura del Predictor

Resumen

Se describe y analiza una nueva organización para predecir trazas de instrucciones. Sin disminuir la relación entre la precisión del predictor y la cantidad de memoria dedicada a las tablas de predicción, se consigue generar más instrucciones en cada predicción, es decir, aumentar la anchura del predictor. Esta mejora se puede traducir en un incremento importante en el rendimiento del procesador.

5.1. Introducción

Se ha argumentado a lo largo de esta tesis cómo los diseños cada vez más agresivos requieren una mayor precisión en el predictor de flujo de control para alcanzar las cotas de rendimiento esperadas, pero a la vez dificultan conseguir que el predictor realice una predicción por ciclo. A lo largo del capítulo previo se describieron propuestas para reducir la latencia media de las predicciones sin renunciar a una alta precisión.

Sin embargo, la alternativa de aumentar la frecuencia del predictor está limitada. En primer lugar, es necesario un tamaño mínimo en el núcleo del predictor. En segundo lugar, escalar la jerarquía de predictores añadiendo más y más niveles no es una solución eficiente, ya que existe duplicación de la información y crece la complejidad de gestionar las prioridades entre los predictores. Si además el esquema de predicción es híbrido (combina diferentes estrategias de predicción o utiliza diferentes tipos de información), entonces llegar un diseño equilibrado supone un elevado esfuerzo de diseño. Segmentar el predictor también implica mayor dificultad y mayor ineficiencia en el uso de los recursos, debido a la gestión de las etapas intermedias.

En este capítulo se ahondará de nuevo en la problemática del ancho de banda del predictor, pero ahora desde la perspectiva de incrementar la anchura de cada predicción, es decir, el número medio de instrucciones válidas por predicción. Se trata de, fijados los recursos de memoria del predictor, aumentar la anchura de las predicciones sin reducir la precisión. En otras palabras, el incremento en la anchura del predictor mantiene, o reduce muy poco, la relación entre precisión y memoria dedicada al predictor.

Aumentar la anchura de las predicciones no descarta, si no que es una alternativa a, aumentar la frecuencia del predictor. En una arquitectura desacoplada de la unidad de búsqueda, se compensa una baja frecuencia de predicción con una alta anchura de predicción, y viceversa. De este modo, si

una nueva estrategia de predicción es capaz de aumentar la precisión a costa de una mayor latencia, aumentar la anchura del predictor permite tolerar mejor este incremento de latencia. En general, se posibilitan nuevos métodos que aprovechen el mayor ancho de banda del predictor para incrementar el rendimiento del procesador o reducir el consumo de energía.

Para aumentar la anchura de las predicciones es necesario predecir múltiples instrucciones de transferencia de control (ITC) en cada operación del predictor. Como se comentó en la sección 2.3, existen varias propuestas, y una de las más simples y eficientes es organizar el predictor en base a trazas de instrucciones. En este trabajo se parte de la propuesta de Jacobson et al. denominada *Next-Trace Predictor* (*NTP*) [JaRS97]. Se comparará *NTP* con un predictor de bloques básicos (*Basic Block Predictor*, *BBP*) analizando cómo varía la relación entre precisión y requerimientos de memoria, y cómo aumenta la latencia al escalar el predictor.

Mostraremos que *NTP* ofrece una peor escalabilidad que *BBP*, debido a un uso ineficiente de la memoria. Para tamaños de *NTP* superiores a 12 KBytes, la relación entre precisión y tamaño se reduce de forma drástica. Así, la mayor anchura de las predicciones proporcionada por *NTP* se obtiene a costa de una menor precisión, lo cual no es aceptable según los objetivos que se habían indicado. El problema que se plantea es diseñar una nueva organización para predecir trazas de instrucciones, y aumentar la anchura del predictor, pero que mantenga una relación entre precisión y requerimientos de memoria similar a la de un predictor de bloques básicos.

La propuesta descrita en este capítulo se basa en dos ideas básicas:

- codificar las predicciones con un *identificador local* de trazas reduce el tamaño dedicado a la tabla de predicciones
- como el número de trazas que comienzan en una misma instrucción es reducido, se puede usar un *identificador local* de muy pocos bits

El resultado es un predictor llamado *Local Trace Predictor* (*LTP*). La primera idea es adaptar el esquema de un predictor de bloques básicos, *BBP,* a un predictor de trazas. Mientras *BBP* codifica si un salto debe saltar o no con 1 bit, *LTP* codificará cada una de las trazas que empiezan en la misma instrucción con un identificador. Utilizando tablas diferentes para las predicciones codificadas y para la información relativa a cada traza, se conseguirá un uso más eficiente de la memoria.

La segunda idea es reducir el número de bits dedicados a la codificación de una traza. Si se limitan a dos las trazas que pueden comenzar en una misma instrucción, el diseño de *LTP* adquiere el mismo tamaño y complejidad que *BBP*, pero con una anchura de predicciones alrededor de un 80% mayor. Si el límite se aumenta a 4 trazas, se consigue aumentar la anchura de las predicciones hasta un 200%, pero a cambio se reduce la relación entre precisión y tamaño del predictor alrededor de un 35%.

Limitar el número de trazas es efectivo porque los programas, en general, recorren sólo un reducido subconjunto de los 2^{B} caminos que se pueden dar en una secuencia de ***B*** saltos condicionales. O, al menos, sólo algunos de estos caminos se transitan con una frecuencia significativa.

5.1.1 Esquema del Capítulo

El esquema de este capítulo es el siguiente. La sección 5.2 analizará con profundidad el problema de escalabilidad del predictor *NTP*. Este análisis servirá para motivar, presentar y defender la nueva propuesta de codificación local de trazas. En la siguiente sección se detallará el funcionamiento de esta nueva propuesta (*LTP*), comenzando por una descripción general, y acabando con una explicación minuciosa de las fases de predicción, de recuperación de fallos y de actualización del predictor.

La sección 5.4 presenta resultados para las tres propuestas analizadas, *BBP*, *NTP* y *LTP*. Se muestran, en orden, resultados sobre anchura y

precisión de las predicciones, sobre requerimientos de memoria y latencia de los predictores, y finalmente resultados de rendimiento. Se concluye el capítulo resumiendo las contribuciones presentadas.

5.2. Motivación y Presentación de la Propuesta

En primer lugar, se darán una serie de definiciones previas, necesarias o útiles para proseguir con las explicaciones posteriores. A continuación se describirá con cierto detalle el esquema del predictor de trazas denominado *Next-Trace Predictor* (*NTP*) [JaRS97], y que se ha adoptado como referencia de partida en este trabajo. Con un ejemplo se analizará que, en comparación con un predictor de bloques básicos, el esquema no escala de forma eficiente al tratar de aumentar su precisión. Este análisis servirá para introducir la nueva propuesta y las ideas básicas en las que se basa. En la sección siguiente se describirá la nueva propuesta con todo detalle.

5.2.1 Definición, Identificación y Política de Selección de Trazas

Una *traza de instrucciones* consiste en una secuencia de instrucciones en el mismo orden en que se deben interpretar durante la ejecución dinámica de un programa. La traza puede contener varias instrucciones de transferencia de control (ITCs), y por tanto instrucciones no consecutivas dentro del espacio de direcciones virtuales de la memoria.

Al conjunto de trazas que comienzan en la misma instrucción lo denominaremos una *clase de trazas*. Una clase de trazas se identifica por la dirección de esta primera instrucción. Denominaremos *instancias de trazas*, o simplemente *trazas*, a los elementos del conjunto de una clase. En la Figura 5.1 se muestran dos clases diferentes, representadas por los cuadros con línea discontinua, la primera con tres instancias de traza y la segunda con cuatro. Cada traza dentro de una clase recibe un número de traza local (*Local*

Trace Number, ó *LTN*) que la identifica dentro de la clase. El patrón de los saltos condicionales de una traza se puede codificar de forma binaria y también permite diferenciar las trazas entre sí. Un parámetro básico al definir una traza es el número máximo de instrucciones e ITCs que puede contener.

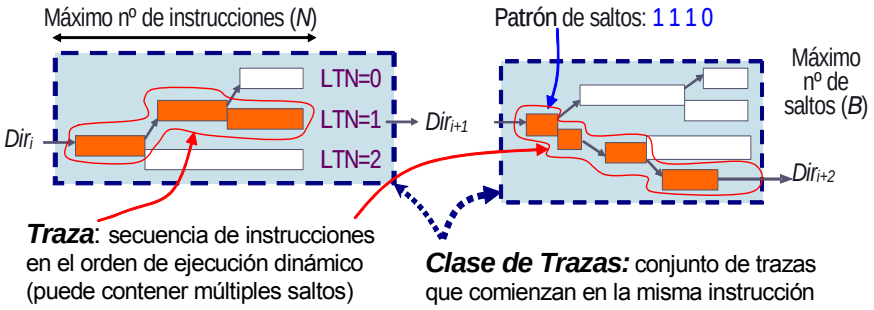


Figura 5.1. Clase de Trazas e Instancia de Traza.

Al diseñar un predictor que construye dinámicamente referencias a trazas de instrucciones y las utiliza como la unidad básica, es necesario definir:

- la *política de selección de trazas,* o reglas que, a partir de la secuencia dinámica de instrucciones, determinan la construcción de una traza
- el esquema de *identificación de las trazas*, o la forma de nombrar una traza que permita diferenciarla en todo momento de otras trazas

Política de Selección de Trazas

La política de selección de trazas es enormemente importante. Con las reglas de selección se pueden definir casi todos los modelos de predictor que han sido utilizados en la literatura (instrucción a instrucción, predictor de *bloques básicos* [YeMP93], de *fetch blocks* [ReAC99], de *instruction streams* [RSLV02], …). La política de selección de trazas determina tanto la forma de identificar las trazas, como la forma de predecirlas, y afecta directamente a la precisión y a la anchura de las predicciones, e indirectamente a la implementación del predictor y a su latencia. El trabajo presentado en este capítulo propone y

analiza una política de selección de trazas que permite aumentar la anchura del predictor, pero a la vez mantener la precisión y la latencia.

Una regla muy conveniente es forzar a que las trazas finalicen siempre que se encuentre una instrucción de salto indirecto (incluidos retornos de subrutina), o que se encuentre una instrucción que genere excepciones, como una llamada al modo supervisor. Veremos en la siguiente sección que esta regla simplifica la identificación de las trazas. Hay otra razón para imponer esta regla de finalización de trazas, y es que se simplifica el uso de predictores especiales para los saltos indirectos. Del mismo modo, finalizar una traza al encontrar un salto a subrutina, aunque sea un salto directo, también facilita la predicción de los saltos de retorno de subrutina. De este modo, sólo permitiremos que una traza incluya en su interior ITCs cuya dirección de salto se conozca de forma estática (saltos directos), ya sean saltos condicionales o incondicionales, excluyendo entre estos últimos a los saltos a subrutina.

En la Tabla 5-1 se muestra una clasificación de cinco tipos de predictores de trazas que se van a analizar en este capítulo, determinados cada uno de ellos por una diferente política de selección de trazas.

Tabla 5-1 Diferentes Predictores de Trazas, definidos por su Política de Selección. (ITC: Instrucción de Transferencia de Control)

Nombre	Una instrucción finaliza la traza si es …
BASIC BLOCK PREDICTOR *BBP* (***N***)	una ITC, o la $N^{ésima}$ instrucción de la traza
NEXT TRACE PREDICTOR *NTP*(***N***,***B***)	el $B^{ésimo}$ salto condicional, un salto indirecto o a subrutina, o la $N^{ésima}$ instrucción de la traza
LOCAL TRACE PREDICTOR *LTP*(***N,B,K***)	como *NTP*(***N,B***), pero sólo 2^K trazas pueden comenzar en la misma instrucción. Si aparece la traza $(2^K+1)^{ésima}$, se aplica un algoritmo especial de desbordamiento de clase
ALIGNED NTP *A-NTP*(***N***,***B***)	como *NTP*(***N,B***), pero si una traza contiene una ITC, entonces debe acabar en una ITC
ALIGNED LTP *A-LTP*(***N,B,K***)	como *LTP*(***N,B,K***), pero si una traza contiene una ITC, entonces debe acabar en una ITC

En todos los casos, los predictores (o las políticas de selección que los definen) tienen ciertos parámetros que permiten especificarlos con más detalle. Un parámetro común es el número máximo de instrucciones por traza (***N***). Cuanto más largas sean las trazas, mayor será la anchura de cada predicción. Trazas más largas contendrán más ITCs, que debido a la regla comentada con anterioridad sólo podrán ser saltos directos incondicionales o condicionales. Como se deberá reservar un espacio fijo en la memoria del predictor para el patrón de saltos condicionales de una traza, es conveniente establecer alguna limitación a la cantidad de saltos condicionales que puede contener una traza, que denominaremos ***B***.

NTP queda definido únicamente por los parámetros ***N*** y ***B***. *BBP* es similar al caso particular de *NTP* en el que ***B***=1, que podemos representar como *NTP*(***N***, ***B***=1). Sin embargo, la configuración *NTP*(***N***, ***B***=1) se diferencia de *BBP*(***N***) en que permite que una traza contenga en su interior varios saltos directos incondicionales.

En este capítulo se presenta una nueva estrategia de selección de trazas que consiste en limitar el número de trazas que pueden coexistir en el predictor para una misma clase. Es decir, se limita el número máximo de trazas que empiezan en la misma dirección de memoria. Definiremos el parámetro ***K*** como el logaritmo en base 2 de este límite. Dicho de otro modo, se permite que hasta 2^{K} trazas comiencen en la misma instrucción, o que cada clase de trazas contenga hasta 2^{K} instancias. Usar el parámetro ***K***, en lugar de 2^{K}, simplifica las explicaciones y se corresponde mejor con la implementación que se propondrá más adelante. Al mecanismo de predicción que se deriva de esta política de selección lo denominaremos predictor local de trazas (*Local Trace Predictor*, *LTP*).

La última característica que utilizaremos para precisar la clasificación de los métodos de selección es si las trazas están alineadas a bloques básicos o no. Una traza está alineada a un BB si finaliza en una ITC, o si es una traza de tamaño máximo (***N*** instrucciones) que no contiene ninguna ITC. La razón de alinear las trazas a BBs es provocar que comiencen en un grupo más

reducido de direcciones, y así también reducir el número total de clases de trazas y de instancias de trazas, y disminuir los requerimientos de memoria en el predictor. Esta característica permite definir los últimos dos tipos de predictores, que denominamos *A-NTP* y *A-LTP*.

Identificación de Trazas

Una traza queda perfectamente identificada por la dirección en memoria de cada una de las instrucciones que contiene. Sin embargo, es preferible usar una forma más compacta para identificar una traza. La alternativa más directa es usar la dirección de memoria de la primera instrucción de la traza (y de la clase de trazas) junto con el comportamiento de las ITCs internas de la traza.

Si las ITCs dentro de la traza sólo pueden ser saltos condicionales o saltos incondicionales directos, entonces un identificador de traza (*Trace Identifier*, *TID*) queda perfectamente determinado con la dirección de memoria de su clase (*Dir*) y con el patrón de sus saltos internos (*Ptrn*). El *patrón de saltos* codifica con un bit el resultado de cada salto condicional (saltar o no saltar), y es el mismo que se utiliza para representar la historia global de los saltos ejecutados previamente. Los saltos incondicionales directos que se encuentran en el interior de una traza tienen un comportamiento determinista, y por tanto no añaden incertidumbre a la identificación de la traza.

Se pueden también identificar las trazas de forma relativa o *local* a la clase a la que pertenecen. Cada traza recibe un número local (*Local Trace Number*, *LTN*) que la identifica dentro de la clase. Siguiendo la nomenclatura de los predictores definidos en la Tabla 5-1, el identificador local se representaría con ***K*** bits. Tal como se avanzó en la introducción del capítulo, esta propuesta es la que permite predecir trazas de una forma eficiente.

Un problema diferente al de identificar las trazas, aunque muy relacionado, es el de determinar qué instrucciones la componen para poderlas ir a buscar a memoria. En la sección 5.4.5 se describirá la organización de la memoria caché de instrucciones.

5.2.2 Predicción de Trazas con *NTP*

El esquema del predictor de trazas denominado *Next-Trace Predictor* (*NTP*) utiliza, en su versión más simple, una única tabla para predecir el flujo de control traza a traza [JaRS97]. La Figura 5.2 muestra un diagrama de bloques que describe su organización. La historia global, utilizada para correlacionar las predicciones de trazas con el comportamiento pasado de los saltos, se construye mediante los identificadores de las últimas trazas predichas (*path history*). Cada identificador de traza, *TID*, consiste en la dirección de la clase de la traza, *Dir*, y en el patrón de sus saltos internos, *Ptrn*. Los bits de la historia se combinan con una serie de operaciones o-exclusiva, realizadas con un esquema de árbol, para obtener un índice con el número de bits adecuado para acceder de forma directa a la tabla.

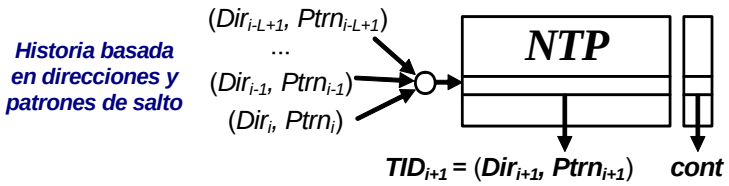


Figura 5.2. Predictor de trazas con una tabla única (*Next-Trace Predictor*, *NTP*).

El campo ***cont*** de la tabla *NTP* es un contador saturado de 2 bis que se utiliza para controlar la actualización de la tabla. El contador se incrementa de 1 en 1 en caso de predicción correcta, y se decrementa de 2 en 2 en caso de fallo. Sólo se modifica una predicción si se encuentra el contador a cero.

La definición que se ha dado en la Tabla 5-1 para el predictor *NTP* corresponde con este esquema, ya que en él no es posible, ni tiene sentido, limitar el número de trazas por clase (parámetro ***K***). Para cada clase de trazas pueden existir tantas entradas en la tabla como historias diferentes hayan sido registradas para todas las instancias de la clase.

En el esquema de tabla única es posible que la entrada leída no corresponda realmente con la historia previa actual. En ese caso, la

probabilidad de que la predicción sea correcta es bajísima. El problema se puede deber a un error de *alias* (la entrada ha sido sobrescrita por una historia diferente). Otra posibilidad, es que la historia que se está utilizando como entrada al predictor ocurra por primera vez (predicciones *en frío*). Fijado el tamaño de la tabla, cuanto más grande sea la historia (más trazas se incluyan en ella) mayor será la incidencia de este problema, hasta tal punto que se puede llegar a revertir la ventaja de correlacionar la predicción con saltos más antiguos. Utilizar etiquetas en la tabla para detectar el error tampoco ayuda, pues no se dispone de ninguna predicción alternativa.

Para solventar el problema se propone una organización híbrida que utiliza dos tablas en lugar de una, [JaRS97]. La primera tabla, de correlación, es similar a la del esquema anterior, y la segunda tabla, secundaria, se indexa utilizando únicamente el identificador de la última traza. La tabla secundaria contiene, para cada traza, una única predicción de la traza que ha de venir a continuación. La tabla de correlación incluye etiquetas para detectar casos de error de alias o de predicción en frío, y en ese caso usar la predicción de la tabla secundaria. Si la etiqueta no es completa, entonces sólo proporciona una probabilidad de error, más certera cuanto mayor sea el tamaño de la etiqueta. Este esquema se muestra en la Figura 5.3.

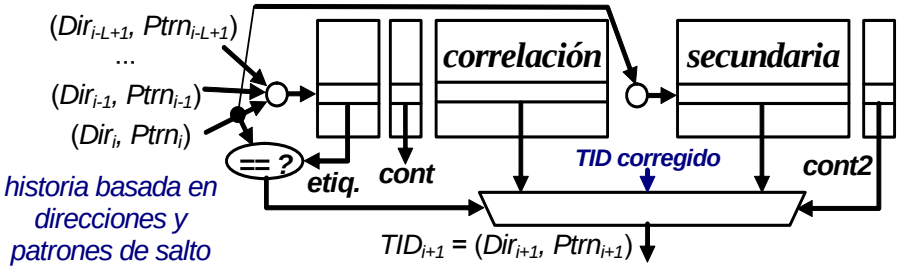


Figura 5.3. Organización híbrida de *NTP*.

El contador de la tabla secundaria (***cont2***) es de 4 bits, y se saturará cuando una traza sea seguida por un mismo sucesor en un porcentaje muy alto de los casos (> 90%). El contador saturado fuerza la utilización de la

predicción de la tabla secundaria y, para mejorar el uso de la tabla de correlación, también impide la actualización de esta última cuando la tabla secundaria contiene la predicción correcta.

En el multiplexor que selecciona la salida de una de las dos tablas se muestra otra entrada, que es necesaria para recuperarse de un fallo de predicción. De algún modo se debe poder cambiar el ciclo de predicción, y aprovechar este multiplexor para hacerlo es una opción eficiente.

Predicción de Saltos Indirectos

El análisis realizado en este capítulo se centrará exclusivamente en los saltos condicionales. En los resultados proporcionados se asume que los saltos indirectos no consumen ningún recurso de los predictores.

El predictor *NTP* usa el mismo mecanismo para predecir tanto los saltos condicionales como los saltos indirectos. En general, los saltos indirectos se predicen con bastante precisión, aunque ésta se puede mejorar de forma considerable usando un mecanismo especial para las instrucciones de retorno de subrutina, llamado la *pila de historias de retorno*. El problema de *NTP* es que al no distinguir entre saltos directos e indirectos, una aplicación con un elevado número de saltos indirectos puede crear mucha polución en las tablas y reducir la precisión para los saltos condicionales, que generalmente es más crítica.

En el próximo capítulo de esta tesis se mostrará que es posible diseñar un predictor específico de saltos indirectos, que utiliza sus recursos de memoria de forma eficiente y que consume muy pocos recursos del predictor principal.

5.2.3 Comparación con un Predictor de Bloques Básicos

El predictor de bloques básicos, *BBP*, presentado en el capítulo 3 y cuyo esquema se muestra de nuevo en la Figura 5.4, utiliza dos tablas diferentes para desacoplar la predicción del sentido del salto condicional y la predicción

de la dirección de memoria a donde saltar. Así, la tabla *BHT* almacena un único bit de predicción para escoger entre los dos posibles caminos de cada salto condicional, y la tabla *BTB* almacena direcciones de salto, tamaños de BBs y tipos de BBs. De esta manera, fijado el tamaño total del predictor, se pueden tener muchas más entradas en la tabla *BHT*, y así aumentar el tamaño de la historia para poder incrementar a su vez la precisión.

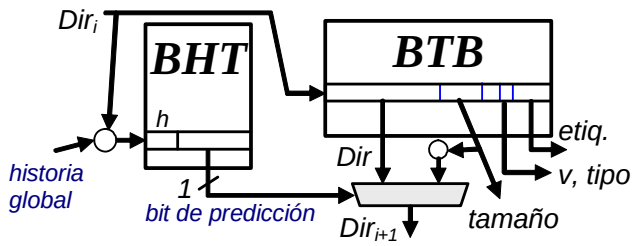


Figura 5.4. Diagrama de bloques funcional de un predictor basado en historia global.

La Tabla 5-2 muestra el tamaño de cada entrada para los predictores *BBP* y *NTP*. En *BBP*, las entradas se reparten entre las tablas *BHT* y *BTB*. Suponemos que las direcciones son de 28 bits y las etiquetas de 10 bits. Los resultados de la Figura 4.4, en el capítulo anterior, indicaban que la tabla *BTB* debía tener 4K entradas para que no se redujese la precisión de forma apreciable, sobre todo si la tabla *BHT* tenía más de 16K entradas. Para tablas *BHT* de 16K entradas o menos, un tamaño de 2K ó 1K entradas reduce un poco la precisión, pero resulta en un mejor compromiso de diseño.

Tabla 5-2 Requisitos de Memoria para *BBP* y para *NTP*

	pred	***h***					
BHT	1	1					
	cont	***Dir***	***Ptrn***	***etiq.***	***tipo***	***v***	***tam***
BTB(*N*)	---	28	---	10	3	1	$\log_2(N)$
NTP(*N*,*B*) de correlación	2	28	***B***	10	---	---	---
NTP(*N*,*B*) secundaria	4	28	***B***	---	---	---	---

La Figura 5.5 muestra cómo se escala el tamaño del predictor *BBP*, en función del tamaño de las tablas *BTB* y *BHT*, al tratar de aumentar la precisión. El pequeño tamaño de las entradas en la tabla *BHT* permite aumentar mucho el número de entradas, sin que el tamaño total del predictor crezca de forma desproporcionada. Eso sí, cuanto mayor es el tamaño de la tabla *BHT*, para poder aumentar la precisión del predictor, menor es la influencia de la tabla *BTB* en el tamaño del predictor. El efecto del parámetro ***N*** en el tamaño del predictor, variando entre 4 y 64, es muy pequeño, sólo apreciable cuando el predictor es pequeño.

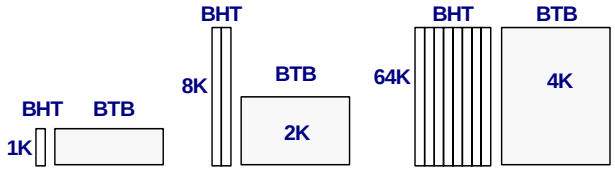


Figura 5.5. Esquema gráfico de la forma en que aumenta el tamaño de *BBP* al escalarlo para aumentar su precisión.

La Figura 5.6 muestra cómo se escala el tamaño del predictor *NTP* al tratar de aumentar la precisión. La tabla secundaria sólo es de ayuda cuando tiene 1K entradas o más, y por tanto sólo tiene sentido el esquema híbrido cuando la tabla de correlación tiene un tamaño de 4K entradas o más. El tamaño de *NTP* crece desde el principio de forma proporcional al número de entradas de la tabla de correlación, cuyo tamaño es más de 20 veces mayor que el tamaño de las entradas de la tabla *BHT*.

Figura 5.6. Esquema gráfico de la forma en que aumenta el tamaño de *NTP* al escalarlo para aumentar su precisión.

La Figura 5.7 compara los tamaños de los predictores *BBP* y *NTP* cuando se escala el número de entradas de cada una de sus tablas para aumentar la precisión. Para cada predictor se usa una proporción adecuada entre los tamaños de las dos tablas que los componen. Esta configuración es la que proporciona mejor precisión en los resultados que se presentarán más avanzado este capítulo. Se puede corroborar en la figura que *BBP* utiliza menos memoria que *NTP* al dedicar 2K entradas o más a correlacionar las predicciones con la historia global. A partir de este número de entradas, la diferencia de tamaños aumenta de forma considerable, hasta alcanzar la proporción de 1 a 20 comentada anteriormente.

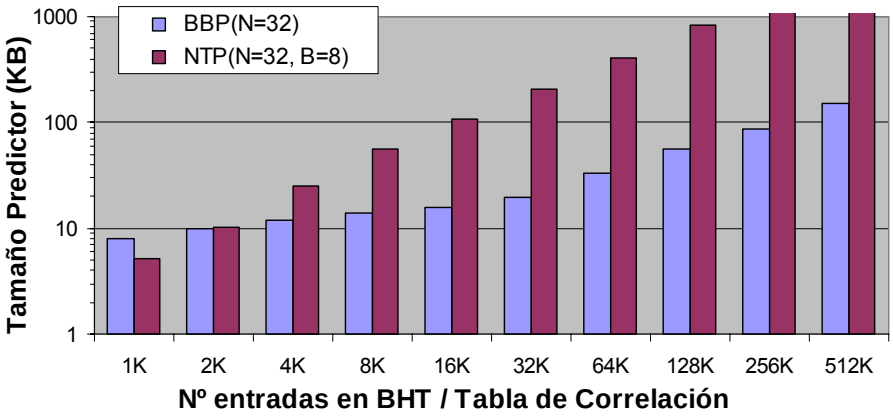


Figura 5.7. Tamaño de los predictores *BBP* y *NTP* en función del número de entradas en la tabla *BHT* o en la *Tabla de correlación*.

Hasta este momento, se ha mostrado que, a partir de un cierto punto de diseño, *NTP* requiere mucha más memoria para dedicar el mismo número de entradas a correlacionar pasado con futuro. Hace falta ahora comprobar cómo afecta a la precisión de las predicciones escalar el número de entradas en cada uno de los dos predictores. En el capítulo 3 se proporcionaron los resultados de precisión para *BBP*. En una sección posterior de este capítulo se darán estos resultados para *NTP*. Se verá que predecir bloques básicos o predecir trazas no proporciona resultados muy diferentes. Los factores que más determinan la precisión son el número de entradas dedicadas a la

correlación, el esquema de indexación y selección utilizado, y que el tamaño de la historia esté bien sintonizado con ambos. Dicho de otro modo, la precisión no depende significativamente de si la unidad de predicción es un bloque básico o es una traza que contenga varias instrucciones de salto.

La Figura 5.8 utiliza todos estos datos para mostrar la relación entre precisión y memoria en ambos esquemas de predicción. Cuanto más a la izquierda se encuentra un punto en la gráfica, más favorable es su relación entre precisión y memoria. Las curvas se cruzan aproximadamente para tamaños del predictor de 12 KBytes. Para tamaños más pequeños, el predictor *NTP* proporciona mejores resultados. Para tamaños más grandes, el predictor *NTP* es claramente muy ineficiente. En las secciones siguientes mostraremos que el problema del predictor *NTP* no es intrínseco a la predicción de trazas, sino a su organización interna.

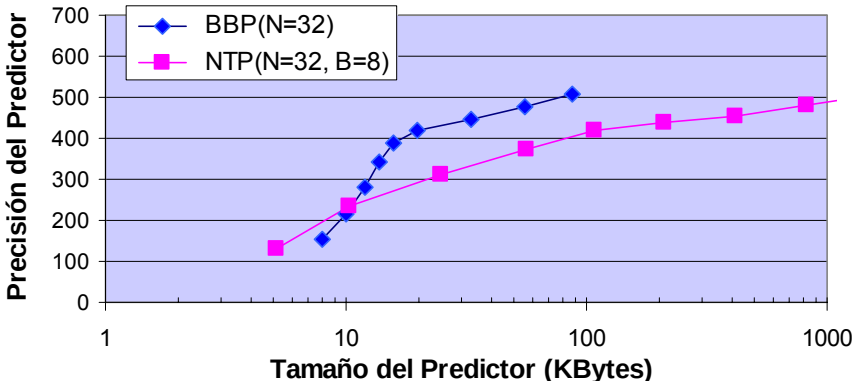


Figura 5.8. Relación entre precisión y requisitos de memoria para *BBP* y *NTP*, con diferentes tamaños de las tablas *BHT* y de correlación.

5.2.4 Propuesta de Codificación Local

Para mejorar la codificación de la información en el predictor de trazas, se propone un esquema similar al del predictor de bloques básicos. Se trata de separar en dos tablas diferentes la información para (1) predecir qué camino

entre ***n*** posibles tomará el flujo de control, y la información para (2) predecir exactamente qué instrucciones componen ese camino y en qué dirección hay que continuar el camino.

Si *BBP* usa un bit de predicción como una especie de identificador local de cada uno de los dos posibles destinos de un salto condicional, el nuevo predictor de trazas (*LTP*) usa ***K*** bits de predicción –el número de traza local (*LTN*) definido con anterioridad– para predecir cada una de las 2^K trazas que es posible recorrer a partir de la instrucción inicial. Estos LTNs se almacenan en una tabla que denominamos *THT* (*Trace History Table*).

Una nueva tabla, que denominamos *Trace Target Buffer* (*TTB*), contiene la información de qué instrucciones componen el camino de cada traza y la información de a dónde saltar tras cada traza. *TTB* es equivalente al *Branch Target Buffer* (*BTB*), que contenía la información sobre qué instrucciones componen un BB (es decir, su tamaño) y a dónde saltar.

La Figura 5.9 muestra el esquema de *LTP*. En el próximo capítulo se describirá con más detalle, y se propondrán algunas modificaciones. En la figura se muestra un ejemplo con ***K***=2 y ***B***=6, en el que la traza es codificada de forma local con el valor 01 (binario), y se corresponde con el patrón de saltos 010110 (no salta-salta-salta-no salta-salta-no salta: el orden de interpretación es de derecha a izquierda).

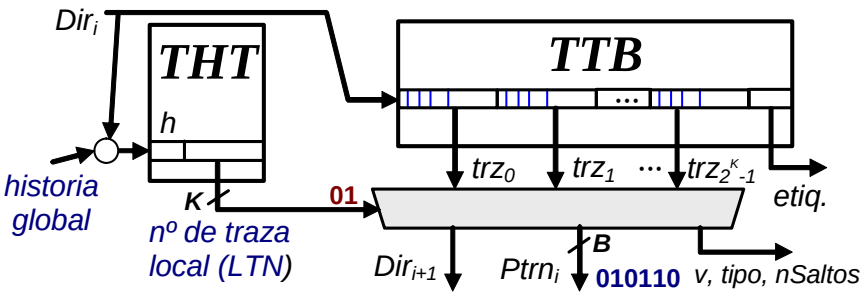


Figura 5.9. Predictor de trazas con dos tablas y codificación local de las predicciones de traza (*Local Trace Predictor*, *LTP*).

Para simular el comportamiento del esquema *NTP*, sería necesario que se pudieran representar con el valor *LTN* todas las posibles trazas de una clase, sin restricciones. Para ello sería necesario que ***K*** fuera igual a ***B*** (el número máximo de saltos condicionales por traza). En ese caso, el valor *LTN* podría coincidir con el patrón de saltos, y este último no sería necesario guardarlo en *TTB*. Mostraremos, sin embargo, que esta opción no es adecuada.

Por un lado, interesa que el valor ***K*** (nº de bits con el que se codifican las trazas) sea muy pequeño, para que también lo sea *THT*, la tabla que contiene las predicciones de ***K*** bits. Pero, por otro lado, interesa que ***B*** (el número máximo de saltos condicionales por traza) sea grande, para que las trazas también lo sean, y aumentar la anchura de las predicciones. Para aportar luz en esta decisión se muestra, en la Figura 5.10, la distribución del tamaño de las clases de trazas, a medida que se incrementa el tamaño de las trazas. La proporción se mide sobre el número total de trazas dinámicas ejecutadas.

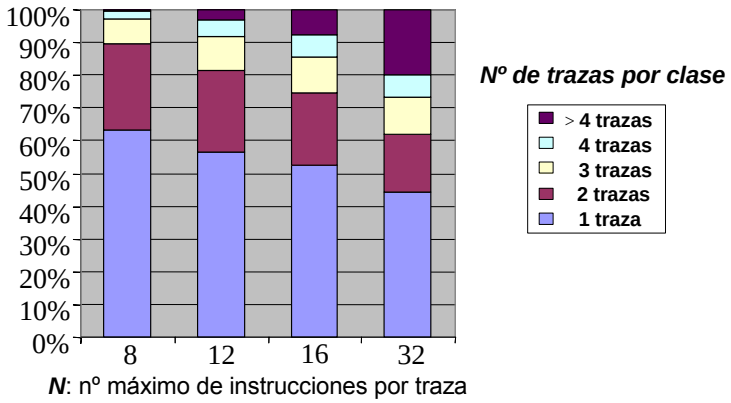


Figura 5.10. Distribución del tamaño de las clases de trazas para diferentes tamaños de las trazas (sin límite para el número total de saltos condicionales, ***B*=*N***).

Para un tamaño máximo de las trazas entre 8 y 32 instrucciones, entre un 62% y un 44% de las trazas ejecutadas pertenecen a una clase *monomórfica*, es decir, una clase en la que sólo hay una única traza. Este es un dato importante, ya que implica que esta proporción de trazas es susceptible a ser predicha de forma perfecta. Por otro lado, poner un límite de 4 trazas por

clase capturaría entre un 99% y un 80% de las trazas dinámicas, dependiendo del tamaño total que se considere para las trazas, y podría suponer un compromiso adecuado de diseño. Mostraremos más adelante que se puede poner un límite de 2 o 4 trazas por clase (***K***=1 ó ***K***=2) con unos valores altos para ***N*** y ***B*** (32 y 6, respectivamente), y aún así alcanzar una anchura en las predicciones comparable a la de *NTP*. Se describirá también un mecanismo que soluciona el problema que se produce cuando aparecen más trazas en una clase de las que se pueden representar en el predictor.

La Tabla 5-3 muestra el tamaño de cada entrada para *LTP*. *THT* utiliza un bit de histéresis y ***K*** bits para predecir una traza local. *TTB* tiene una etiqueta de 10 bits y una serie de campos que contienen la información de una traza, repetidos 2^{K} veces. Estos campos se describirán con detalle más adelante. Los campos de mayor tamaño son los que indican la dirección de la siguiente traza (***Dir***) y el patrón de saltos (***Ptrn***).

Tabla 5-3 Requisitos de Memoria para *LTP*

	LTN	***h***				
THT	*K*	1				
	etiq.	***Dir***	***Ptrn***	***nSaltos***	***tipo***	***v***
TTB(*N,B,K*)	10	$2^{K}\cdot 28$	$2^{K}\cdot B$	$2^{K}\cdot \log_2(B)$	$2^{K}\cdot 3$	$2^{K}\cdot 1$

La Figura 5.11 compara el tamaño de las configuraciones de *LTP* con ***K***=1 y ***K***=2, con los tamaños de *BBP* y *NTP* mostrados en la Figura 5.7. Para cada configuración de *LTP*, el eje horizontal indica el número de entradas en *THT*, pero para cada punto en la gráfica se debe decidir cuál es el número adecuado de entradas en *TTB*. Para tablas *THT* de entre 1K y 8K se usa una *TTB* de 512 entradas. Para tablas *THT* con más entradas, se usa una *TTB* de 1K entradas. Estos valores son los que proporcionan una mejor relación entre precisión y requisitos de memoria (usando datos de precisión que se presentarán más adelante).

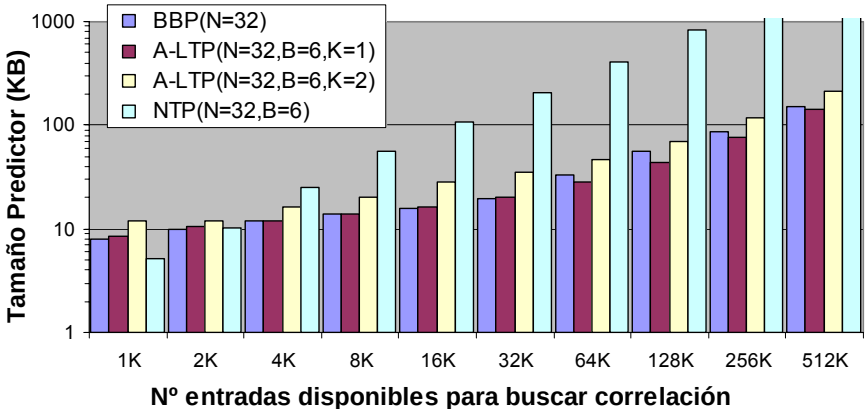


Figura 5.11. Tamaño de *BBP*, *LTP* y *NTP* en función del número de entradas dedicadas a la correlación con la historia global.

Los predictores *LTP*(***K***=1) y *BBP* utilizan aproximadamente la misma cantidad de memoria. Por un lado, *BHT* y *THT* tienen entradas de la misma longitud (2 bits). Por otro lado, para conseguir la misma precisión, *BBP* necesita, en general, el doble de entradas en *BTB* que el predictor *LTP*(***K***=1) en *TTB*, pero las entradas en *BTB* son de la mitad de longitud. *LTP*(***K***=2) requiere más memoria que los dos predictores anteriores debido a que las entradas en *TTB* son de casi el doble de tamaño, y a que las entradas en *THT* son un 50% más grandes (3 bits, en lugar de 2 bits).

En todo caso, se muestra con claridad que el nuevo esquema de predicción de trazas, para 4K entradas de correlación o más, necesita mucha menos memoria que *NTP*. Sería necesario que *NTP* obtuviera bastante mayor precisión con el mismo número de entradas para justificar sus grandes requerimientos de memoria. En realidad, los experimentos que se mostrarán más adelante demuestran que la precisión de *LTP* es prácticamente la misma que la de *NTP*, y que al menos es tan buena como la de *BBP*. Cruzando los resultados de precisión con los requerimientos de memoria se obtiene la gráfica de la figura siguiente. En ella se puede observar cómo *BBP* y *LTP*(***K***=1) tienen una relación muy similar entre precisión y memoria. *LTP*(***K***=2) tiene una peor relación precisión/memoria que los dos anteriores predictores pero, a cambio, veremos que proporciona anchuras de predicción

bastante superiores. Finalmente, *NTP* muestra, para tamaños superiores a 12 KBytes, una muy baja relación precisión/memoria con respecto a los otros tres diseños.

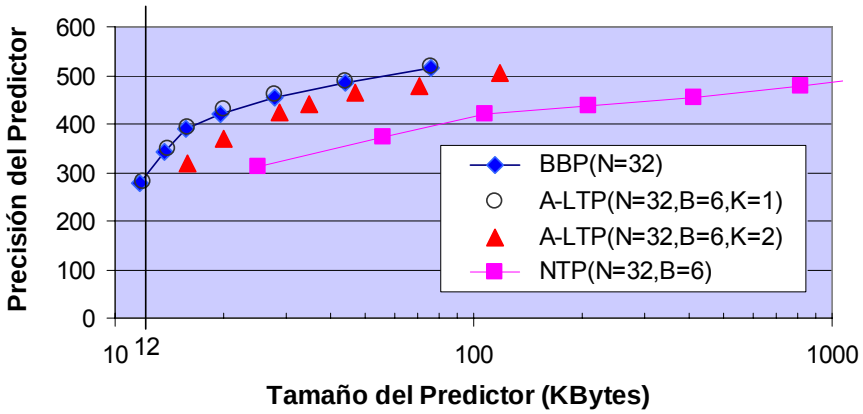


Figura 5.12. Relación entre precisión y requisitos de memoria para los predictores *BBP, LTP* y *NTP*, con diferentes tamaños de las tablas.

5.3. Predicción de Trazas con Codificación Local

En esta sección se mostrarán los detalles del funcionamiento del predictor *LTP*. En primer lugar, se explicará el modo general en que el predictor va guardando y utilizando la información sobre las trazas. A continuación se analizarán diferentes posibilidades para representar la historia global y para realizar la indexación de las tablas de predicción y la selección final de la predicción. De este modo, se obtendrá un esquema general de predictor que permitirá comparar de una forma más equitativa los efectos en el rendimiento de la política de selección de trazas utilizada. Finalmente, se detallarán las tres diferentes fases del predictor: predicción, recuperación de fallos, y actualización.

5.3.1 Descripción General

La Figura 5.13 muestra de nuevo el diagrama de bloques de *LTP*. *THT* contiene las predicciones en forma de número local de traza (*LTN)* de ***K*** bits. El *LTN* especifica uno de entre 2^{K} campos (de trz_0 a trz_{2^K-1}) de la entrada de *TTB* que corresponde a la clase de trazas que se desea predecir.

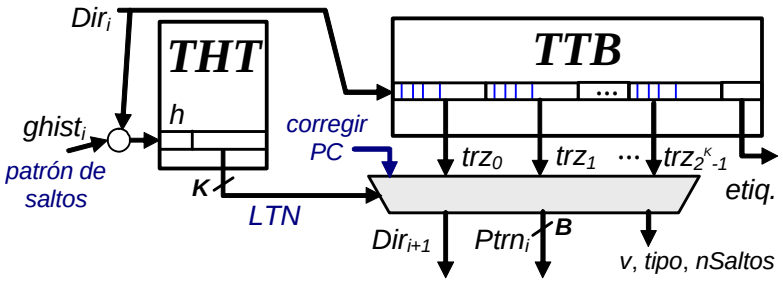


Figura 5.13. Predictor de trazas con codificación local de predicciones (*Local Trace Predictor*, *LTP*).

A medida que se retiran las instrucciones de la unidad de ejecución, se van aplicando las reglas de selección de trazas y se va obteniendo la secuencia dinámica de trazas. La primera vez que aparece una traza de una nueva clase se le asigna una entrada en *TTB*. Para cada traza, se almacena el número de instrucciones de salto condicional que contiene, el patrón de resultados de estos saltos, el tipo de instrucción con la que finaliza, y la dirección de la clase de trazas que le sigue a continuación.

Mientras hay una única traza en una clase, no se utiliza *THT* ni en las predicciones ni en las actualizaciones, ya que no es necesaria. De este modo, se reducen los errores de alias en *THT*. A medida que aparecen nuevas trazas de una misma clase, éstas se van copiando en la entrada correspondiente de *TTB*. Cuando se sobrepasa el límite del número de trazas en una clase, decimos que se produce *desbordamiento en la clase de trazas*. Es necesario buscar una solución a este problema que no reduzca la precisión del predictor y que reduzca lo menos posible la anchura de las

predicciones. Más adelante analizaremos dos estrategias básicas: hacer que las trazas “compitan” por ser almacenadas en el predictor, o “recortar” el tamaño de las trazas para que todas puedan caber en *TTB*.

Si el tamaño de *TTB* es insuficiente, el aprendizaje de una clase de trazas se pierde al reemplazar cada entrada. Por tanto, conviene tener una *TTB* grande. Por otro lado, cuantos más bloques básicos se colapsen dentro de una traza, menos entradas serán necesarias en *TTB*.

5.3.2 Codificación de la Historia Global

El uso de la historia global proporciona un contexto a las predicciones que permite aumentar su precisión. Existen en la literatura diferentes alternativas para codificar la historia de los saltos previos. Fundamentalmente, se puede utilizar el patrón de saltos (*branch pattern*), o las direcciones de saltos (*path history*). En la propuesta original de *NTP* [JaRS97] se utilizan identificadores de trazas, que contienen tanto direcciones de trazas como patrones de sus saltos internos. En el diseño del procesador alpha EV8 también se propone utilizar este tipo de combinación para la historia global [SFKS02].

Se han hecho pruebas con las dos formas de codificar la historia, para un predictor de bloques básicos y para varias configuraciones de *LTP*. Para cada tamaño de *BHT* (ó *THT*) y para cada forma de combinar los bits de la historia y generar un índice (que se describirán a continuación) es necesario probar diferentes longitudes de historia para encontrar el mejor resultado. Si la historia se genera con direcciones, existe la dificultad añadida de tener que decidir cuántos bits se utilizan de cada dirección. Es conveniente usar más bits para las direcciones de salto más recientes, y menos bits para las más lejanas, tal y como se propone en [JBSS97].

En el siguiente capítulo se verá que usar direcciones al codificar la historia mejora ligeramente la predicción de los saltos indirectos. En cambio, la predicción de saltos condicionales sólo se mejora ocasionalmente, y no es

homogénea para todos los programas. En este capítulo se ha decidido codificar la historia sólo con el patrón de saltos, y se invocan dos razones.

En primer lugar, el patrón de saltos es siempre el mismo, sea cual sea la política de selección de trazas, mientras que usar la dirección de la traza (y el patrón interno de la traza) es dependiente de la política de selección. Forzar a que la historia sea la misma para todos los predictores evita cierta incertidumbre al comparar sus resultados. Además, el patrón de saltos tampoco cambia si se modifica dinámicamente la política de selección de trazas, un caso que puede darse cuando se produce un desbordamiento de clase (tal como se verá más adelante). Para usar una historia basada en direcciones que fuera independiente de la política de selección sería necesario generar todas las direcciones de salto intermedias.

La segunda razón de codificar la historia con el patrón de saltos es simplificar el espacio de diseño para cada tamaño de la tabla de predicción, ya que el único parámetro relevante es la longitud (número de bits) de la historia. Así es posible analizar todas las posibilidades para poder comparar de forma más justa los predictores.

5.3.3 Mecanismos de Indexación y Selección

Generalmente el acceso a la tabla de correlación es directo, y no se utiliza una etiqueta (*tag*) para verificar que una entrada corresponda exactamente a una determinada historia global. Así el mecanismo de selección de datos es simple y rápido, y se evita utilizar memoria para almacenar las etiquetas. El problema son los errores de *alias*, que una entrada de la tabla sea asignada a varias combinaciones de historia diferentes, que se perjudiquen entre sí. El sistema de indexación debe tratar de reducir la probabilidad de estos errores, distribuyendo los datos de entrada de la forma más uniforme que sea posible entre los 2^n posibles valores para el índice. Al aprovechar al máximo el espacio disponible en la tabla de correlación es posible usar una historia global más larga para así aumentar la precisión.

El primer esquema de indexación que se ha utilizado es una variante de *gshare* [YePa91], basada en el esquema propuesto en [JBSS97] y [JaRS97]. Este esquema ya fue presentado en el capítulo 3 y usado en los predictores definidos en el capítulo 4. La sección 3.5.1 mostraba cómo convertir un valor ***v*** de más de ***n*** bits en un valor de ***n*** bits con una función que se denominaba ***xor-fold***, basada en la operación "o-exclusivo". En la Figura 5.14 volvemos a mostrar el esquema para convertir la historia y la dirección del bloque básico o traza en varias secuencias intermedias de bits, que se juntan en una larga cadena para dar lugar al índice final.

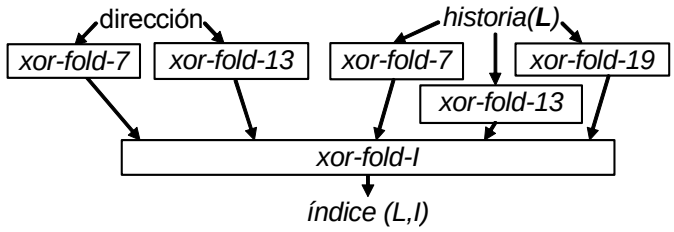


Figura 5.14. Esquema de generación de índices, en función de la longitud en bits de la historia, ***L***, y de la longitud en bits del índice generado, ***I***

En el esquema de indexación *gshare* los dos parámetros son la longitud en bits del índice a la tabla, ***I***, y la longitud en bits de la historia global, ***L***. Cada tamaño de tabla de correlación determina ***I***, pero es necesario realizar simulaciones para estimar el valor de ***L*** que proporciona mayor precisión.

El esquema de indexación denominado *e-gskew*, y propuesto en [MiSU97], divide el espacio de la tabla de correlación en tres tablas o bancos, cada uno de ellos indexado utilizando historias de diferente longitud. Las predicciones leídas de cada banco se comparan entre sí, y se escoge la predicción mayoritaria. En el caso de predecir números de traza de 2 o más bits, es posible que las tres predicciones difieran entre sí. En este caso, se escogerá la predicción de la tabla que utiliza una historia de menor longitud.

Un error de alias en uno solo de los tres bancos queda desapercibido por el esquema de selección mayoritaria. Al usar índices diferentes, se reduce

mucho la probabilidad de un error de alias en dos o más tablas a la vez. El banco que usa una menor longitud de historia aprende antes que los otros bancos, y por tanto es la mejor opción para realizar predicciones *en frío*.

5.3.4 Fase de Predicción

Las entradas en la fase de predicción son la dirección de una clase de trazas (Dir_i) y la historia global de saltos condicionales ($ghist_i$). Los dos valores se combinan para generar un índice con el que acceder a *THT*, tal como se ha descrito en la subsección anterior. Simultáneamente, se usa la dirección de la clase de trazas (Dir_i) para indexar la tabla *TTB* y leer una entrada. El número local de traza, *LTN*, leído de *THT* se utiliza para seleccionar la información de *TTB* que corresponde a una de las 2^K trazas (de trz_0 a trz_{2^K-1}).

Para cada traza se utilizan 5 campos de información: ***v***, ***Dir***, ***tipo***, ***nSaltos*** y ***Ptrn*** (que se mostraron en la Tabla 5-3). El campo ***v*** es un bit que indica si la información de la traza en la tabla es válida o no. El campo ***Dir*** contiene la dirección de la clase de trazas que sigue a la traza, y que se utiliza para generar el valor Dir_{i+1} con el que iniciar el siguiente ciclo de predicción. El campo ***tipo*** codifica el tipo de instrucción con la que finaliza la traza. Si se trata de un salto indirecto, o de una instrucción de retorno de subrutina, entonces se pueden utilizar predictores específicos para proporcionar mejor precisión que la obtenida por la dirección almacenada en ***Dir***.

El campo ***nSaltos*** indica el número de instrucciones de salto condicional que contiene la traza, y el campo ***Ptrn*** el patrón de resultados de estos saltos condicionales. Ambos valores se utilizan para actualizar de forma especulativa la historia global y para generar el identificador de la traza predicha. La Figura 5.15 muestra que, en caso de acierto en la tabla *TTB*, la historia global se actualiza desplazando la historia previa ***nSaltos*** bits hacia la izquierda y añadiendo por la derecha los ***nSaltos*** bits menos significativos del campo ***Ptrn***.

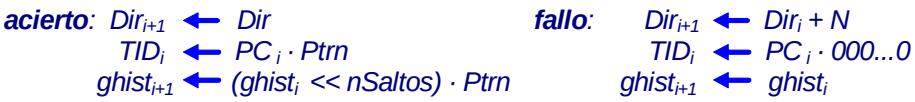


Figura 5.15. Descripción formal del funcionamiento de la fase de predicción. La operación "·" indica concatenar bits. La operación "<<" indica desplazar los bits hacia la izquierda.

El identificador de la traza (TID_i) se construye añadiendo los ***B*** bits del campo ***Ptrn*** a la derecha de la dirección de la clase de trazas (Dir_i). Cabe resaltar que *LTP* genera los patrones de saltos de las trazas con un ciclo de retardo en comparación a como lo hace *NTP*. Si el valor de entrada es la dirección de la traza *i*, *NTP* predice la dirección de la clase *i+1* y el patrón de la traza *i+1*, mientras que *LTP* predice la dirección de la clase *i+1* y el patrón de la traza *i*.

En caso de fallo en *TTB*, se predice que la traza será un bloque básico secuencial (sin saltos internos, ***nSaltos***=0) y de tamaño máximo (***N***). La dirección de la siguiente traza (Dir_{i+1}) se obtiene sumando ***N*** a la dirección de la traza actual, y la historia global de saltos condicionales se deja tal como estaba. Por otro lado, el identificador de traza se construye añadiendo ***B*** ceros a la derecha de la dirección de la traza.

La etapa del procesador posterior al predictor busca las instrucciones en memoria usando el valor *TID*. Para simplificar la interpretación de este identificador, siempre se usan ***B*** bits para representar el patrón de saltos, aunque algunos bits no proporcionen información. En cambio, al actualizar la historia global sólo se utilizan los bits que contienen información, para así maximizar la información contenida en la historia global. Para facilitar y acelerar la recuperación de un fallo de predicción se realizan múltiples copias de la historia global y, por tanto, cuanto menor sea su longitud, menor será la memoria requerida.

5.3.5 Fase de Recuperación: Fallos Parciales de Trazas

Al producirse un fallo de predicción de salto, se debe proporcionar al predictor la dirección de la traza donde se ha producido el error, el número del salto dentro de la traza en el que se ha producido el error, y la dirección de la siguiente instrucción a ese salto mal predicho. La Figura 5.13 muestra que el multiplexor que permite seleccionar entre las 2^K posibles trazas destino, también permite recuperar el camino correcto después de un fallo de predicción.

Recuperación de un fallo completo de traza

Cuando el salto en el que se ha producido el error está al final de la traza, diremos que se trata de un *fallo completo* de predicción de trazas. Este tipo de fallos se recupera tal y como se haría en el caso del predictor de bloques básicos (ver capítulo 3). Se utiliza la dirección de la siguiente instrucción como dirección inicial de la traza que hay que predecir y se combina con la historia, una vez corregida, para comenzar de nuevo la fase de predicción.

Recuperación de un fallo parcial de traza

Cuando el salto en el que se ha producido el error no está al final de la traza, diremos que se trata de un *fallo parcial* de predicción de trazas. Este tipo de fallos es bastante frecuente, más frecuente cuanto más largas sean las trazas y más instrucciones de salto puedan contener. Por tanto, utilizar una estrategia sencilla para recuperarse de este tipo de fallo puede reducir el rendimiento de forma considerable, tal y como se observa en [FrPP98].

Nuestra propuesta consiste en utilizar toda la información disponible hasta ese momento, incluida la que está almacenada en *TTB*, para realizar una nueva predicción. Llamaremos *traza errónea* a la que contiene el salto mal predicho. Llamaremos *trazas alternativas* a las trazas almacenadas en *TTB* que pertenecen a la misma clase que la traza errónea. El *patrón parcial de*

saltos es el patrón de la traza errónea tras quitarle los saltos que vienen a continuación del salto mal predicho, y tras corregir la predicción de este último salto. Es posible que el patrón parcial de saltos sea erróneo, debido a un error en algún salto anterior que todavía no haya sido ejecutado, pero es la información más fiable disponible en ese momento.

Los datos de las trazas alternativas se pueden recuperar, o bien de la cola de predicciones (donde se han debido guardar durante la fase de predicción), o bien leyendo de nuevo la tabla *TTB*. La primera opción es más rápida pero consume más energía y más memoria en la cola de predicciones.

Entre las trazas alternativas se seleccionan aquéllas que coincidan con el patrón parcial de saltos. Se ha comprobado experimentalmente que elegir la primera opción válida de la tabla *TTB* proporciona mejores resultados que escoger una al azar, o incluso que utilizar contadores de un bit para correlacionar los fallos de una traza con una traza alternativa. Una posible explicación es que las trazas se guardan en cada entrada de *TTB* en el orden en que aparecen durante la ejecución, de modo que las primeras trazas suelen ser también las más frecuentes, y parece ser que las que tienen más probabilidades de ejecutarse en caso de error de predicción.

El contenido de *TTB* va cambiando dinámicamente, y puede ocurrir que la posición de la tabla que contenía la predicción errónea haya sido modificada. La poca frecuencia de estos sucesos hace que no sea significativa en el rendimiento la forma en que se manejan, pero siempre se ha de cumplir que la nueva predicción contenga al menos una nueva instrucción diferente a la de la predicción previa, para no producir interbloqueo.

Aparece una nueva traza no almacenada en TTB

Si en *TTB* no se encuentra ninguna traza alternativa que coincida con el patrón parcial de saltos, entonces se “rellena” el trozo de traza ya ejecutado con una secuencia de instrucciones consecutivas, hasta completar el tamaño máximo de la traza.

Este caso es relativamente frecuente cuando el predictor está "aprendiendo" y aparecen trazas que todavía no habían sido ejecutadas. En este caso, es frecuente que el error se detecte en la etapa de decodificación y que no exista información sobre el resultado de la ejecución del salto (es decir, si finalmente salta o no salta). En este caso se debe utilizar una política de predicción estática para el salto. Por ejemplo, saltar si el salto es hacia atrás y no saltar si es hacia delante.

5.3.6 Fase de Actualización: Desbordamiento de la Clase de Trazas

En la fase de actualización se van formando las trazas utilizando las instrucciones recién retiradas y aplicando la política de selección de trazas que corresponda. Una vez identificada la traza retirada, se comprueba si corresponde con la traza predicha, cuya información se extrae de la cola de predicciones. Entonces se debe actualizar, si es conveniente, el contenido de *THT* y *TTB*. Cabe remarcar de nuevo que la actualización de las tablas de predicción se hace con cierto retraso.

THT se actualiza sólo para clases de trazas con más de una instancia. En caso de predicción acertada, se pone a 1 el bit de histéresis, mientras que en caso de fallo, se lee el bit de histéresis y se pone a cero. Sólo se modifica la predicción de *THT* si hubo fallo previo y el bit de histéresis leído era cero.

TTB sólo se actualiza si se produjo un error de predicción de traza. Se vuelven a leer las etiquetas de *TTB* para encontrar la posición donde se almacena la información sobre la clase. En caso de fallo (pudo haberlo o no en la fase de predicción), se selecciona una entrada para emplazar la nueva clase, usando una política de reemplazamiento *LRU*. La nueva traza se almacena en el primer campo de la entrada seleccionada, se le asigna el número cero como *LTN*, y se borran los bits de validez correspondientes al resto de trazas de la clase.

En caso de acierto en *TTB*, se comparan los patrones de salto para ver si la traza ya está almacenada. Si no es así, se emplaza la nueva traza en la primera posición vacía, se le asigna el *LTN* que le corresponda a esa posición, y se activa su bit de validez. Si no queda espacio en la entrada de *TTB* para una nueva traza, entonces se ejecuta el algoritmo descrito a continuación.

Algoritmo de desbordamiento de la clase de trazas.

Es necesario buscar una solución a este problema que no reduzca la precisión del predictor y que reduzca lo menos posible la anchura de las predicciones. En un estudio preliminar se analizaron soluciones simples. La más directa era descartar las nuevas trazas. Los resultados fueron muy insatisfactorios, especialmente cuando el límite era de 2 trazas por clase. Otra solución analizada fue hacer que las trazas "compitieran" por una posición dentro de *TTB*, utilizando contadores para "premiar" a las trazas más frecuentes, o a las trazas que se predecían correctamente de forma más frecuente. Las pruebas en este sentido tampoco fueron satisfactorias, ya que con un límite de 2 trazas por clase la precisión se reducía más de un 15%.

La razón de que el mecanismo anterior no funcionara adecuadamente es que para ciertos programas existían clases de trazas con muchas instancias y con una alta frecuencia de ejecución. Al competir por una posición en la clase, se producía un continuo trasiego que impedía que el predictor tuviese tiempo suficiente para aprender.

Una alternativa más adecuada es reducir dinámicamente el número de saltos condicionales incluidos dentro de las trazas, de forma que se reduce el número total de trazas en una clase. De este modo, se reemplazan las trazas existentes en *TTB* por versiones más cortas, fundiendo dos o más trazas en una traza más pequeña que contiene la parte común de las primeras. De este modo, algunos campos de la entrada *TTB* pueden ser invalidados, y algunas predicciones en *THT* pueden volverse obsoletas. Por tanto, el efecto en el

rendimiento del predictor es una reducción en la anchura media de las predicciones, y un posible incremento en el número de fallos de predicción, que se ha probado experimentalmente que es muy pequeño.

La precisión se reduce muy poco porque el número de desbordamientos es pequeño. Al principio de la ejecución del programa, los desbordamientos se producen con relativa frecuencia, pero una vez que se ha acomodado el tamaño de las trazas en cada clase, prácticamente dejan de producirse, y el predictor puede "aprender" a predecir las trazas en su longitud final.

Se han analizado dos algoritmos para manejar el desbordamiento de una clase de trazas:

- *Recortar todas las trazas*: se reduce el número máximo de saltos condicionales por traza para todas las trazas de la clase, hasta que se consigue acomodar la nueva traza.
- *Recortar la traza de mayor coincidencia:* entre las trazas de la clase se busca aquella que coincida con la nueva traza en un número mayor de instrucciones, y se substituye por la traza que corresponde con el trozo coincidente. Si el número de saltos condicionales de la traza final resultara ser menor que ***K***, entonces se anula la operación, y en su lugar ser utiliza la estrategia de *recortar todas las trazas*.

El primer algoritmo es más sencillo de implementar, pero requiere guardar en cada entrada de *TTB* el límite actual de saltos condicionales. El segundo algoritmo es más complejo de implementar, pero no requiere de información adicional en *TTB*. Hay que notar que este algoritmo se lleva a cabo en una fase cuyo retardo no es crítico y que además se ejecuta de forma muy poco frecuente. Por tanto, el algoritmo puede alargarse múltiples ciclos sin que tenga efectos significativos en el rendimiento.

La siguiente figura muestra un ejemplo en el que se tienen 4 trazas en una misma clase y aparece una quinta traza. Inicialmente se admiten trazas con hasta 5 saltos condicionales. El algoritmo de *recortar todas las trazas* daría

lugar a trazas con sólo 4 saltos condicionales. Afectaría a las trazas nº 2 y nº 3, que se verían recortadas. La nueva traza correspondería con la traza nº 3, y sobraría un trozo de la traza que se está retirando, que se uniría a las siguientes instrucciones retiradas del procesador para forma una nueva traza.

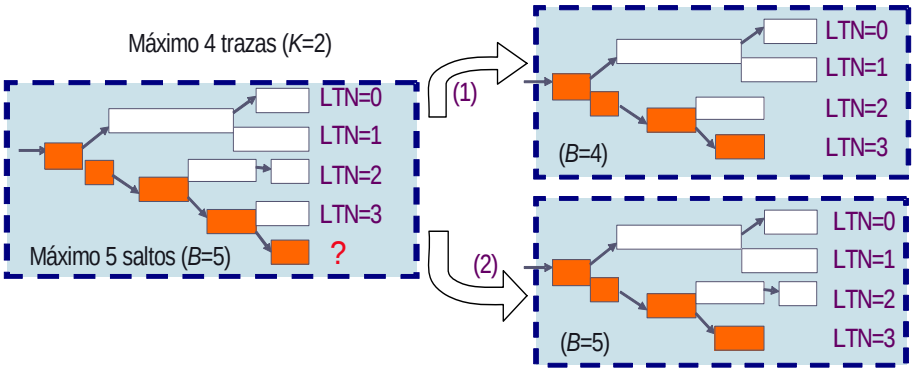


Figura 5.16. Ejemplo que ilustra los dos algoritmos para tratar un desbordamiento de clase de trazas. (1) *recortar todas las trazas* (2) *recortar la traza de mayor coincidencia*.

El algoritmo de *recortar la traza de mayor coincidencia* sólo afectaría a la traza nº 3, que sería recortada para acomodar a la nueva traza. En este ejemplo, el segundo algoritmo reduce la longitud media de las trazas menos que el primer algoritmo, y por tanto reduciría menos la anchura de las predicciones. Las simulaciones han mostrado que este caso se produce a menudo, y que la anchura de las predicciones es un 5% mayor para la estrategia de *recortar la traza de mayor coincidencia*, y que es la que se considerará a partir de ahora.

5.4. Resultados

En primer lugar se analiza la anchura y la precisión de las predicciones para las diferentes configuraciones de predictores. Para medir la precisión del predictor *NTP*(***N***,***B***) se usa la configuración *LTP*(***N***,***B***,***K*=*B***). De este modo, se

simula el mismo algoritmo de indexación en todos los predictores, y al igualar los parámetros ***K*** y ***B*** se asegura que no hay ningún límite en el número de trazas que puede haber en una clase. Para simplificar la discusión, se sobreentenderá que al nombrar las tablas *THT* y *TTB* mientras se considera el predictor *BBP*, se está refiriendo a las tablas *BHT* y *BTB*, respectivamente.

Estos datos se cruzan con el tamaño de cada predictor para encontrar la configuración más adecuada de cada tipo de predictor y para poderlos comparar de forma justa. Estos resultados se han obtenido con un modelo de procesador simplificado, que utiliza un modelo ideal de la memoria, y que no considera las dependencias entre las instrucciones ni sus latencias. El número de ciclos entre la predicción y la actualización de las tablas se ha fijado en 16 ciclos. El predictor sí que se ha modelado con todo detalle.

A continuación se analiza el efecto en el rendimiento del procesador del aumento de anchura proporcionado por *LTP*. Para obtener estos resultados se utiliza un modelo de procesador realista, con tres niveles de memoria y una organización desacoplada de la unidad de búsqueda, y con prebúsqueda de instrucciones dirigida por el predictor.

5.4.1 Anchura de las Predicciones

La anchura de las predicciones equivale a la longitud media de las trazas, que básicamente depende de la política de selección de trazas utilizada. Tal y como se definió en el capítulo 2, el cómputo de la anchura del predictor no tiene en cuenta los fallos de predicción y su penalización, aunque el efecto de los fallos en el rendimiento sí que se considera en la simulación detallada del procesador. La longitud de las trazas se mide en la fase de actualización, a medida que las instrucciones se van retirando, teniendo en cuenta la reducción en la longitud de las trazas que se produce durante la ejecución de programa a medida que se producen desbordamientos de clase.

La siguiente tabla muestra la correspondencia entre el máximo número de instrucciones permitido en una traza, ***N***, y el máximo número de saltos condicionales, ***B***.

Tabla 5-4 Correspondencia entre ***N*** y ***B***, para cada tamaño de ***N***.

	BBP				**LTP**			
***N*:**	**8**	**16**	**32**	**64**	**8**	**16**	**32**	**64**
***B*:**	**1**	**1**	**1**	**1**	**3**	**5**	**6**	**8**

La Figura 5.17 muestra la anchura de las predicciones para diferentes configuraciones de predictor y diferentes valores de ***N***, separando los datos entre los programas SPECint00 y los programas SPECfp00. Los resultados correspondientes a *BBP* indican que se alcanza un límite para la longitud de los bloques básicos cercano a 7,7 para SPECint00, y superior a 18 para SPECfp00. Estos resultados revelan el hecho bien conocido de que los programas de SPECint00 contienen bloques básicos más cortos (es decir, un mayor porcentaje de instrucciones de salto) que los programas de SPECfp00.

En claro contraste, *LTP*(***K***=***B***), que corresponde a *NTP*, alcanza anchuras de precisión crecientes a medida que se aumenta el parámetro ***N***. Si no se llega al tamaño máximo de las trazas no es debido a que el parámetro ***B*** sea muy restrictivo, ya que éste se ha escogido suficientemente grande para que no se reduzca la anchura más de un 5% comparado con haber usado un valor para ***B*** ilimitado. La reducción en la longitud de las trazas viene provocada por la restricción de finalizar las trazas en saltos indirectos o saltos a subrutina.

La configuración *LTP*(***K***=2) reduce la longitud de las trazas menos de un 10% comparado con *LTP*(***K***=***B***), excepto si ***N***=64 y con el subconjunto de programas SPECint00. Por tanto, la restricción de 4 trazas por clase afecta muy poco a la anchura del predictor. En otras palabras, los casos de desbordamiento de clase son poco frecuentes y, o bien afectan a clases poco utilizadas, o bien reducen moderadamente el tamaño de las trazas.

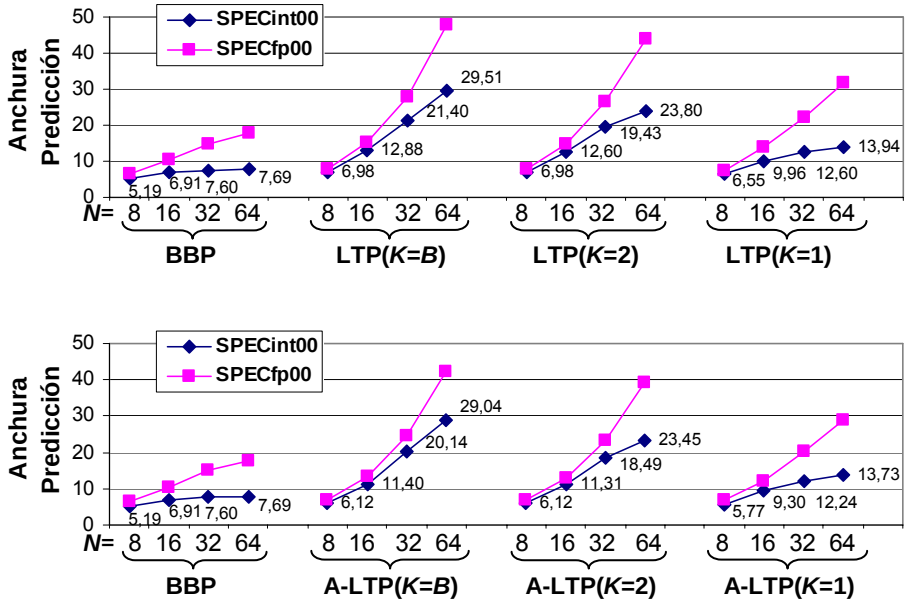


Figura 5.17. Anchura de las predicciones para diferentes políticas de selección de trazas (Tabla 5-1). Los resultados suponen una tabla *BTB* ilimitada.

La configuración *LTP*(***K***=1) satura la anchura de predicción alrededor de 14 para SPECint00, y reduce la anchura hasta un 30% en comparación con *LTP*(***K***=***B***) para SPECfp00. Por tanto, la restricción de 2 trazas por clase reduce la anchura de *LTP*, pero aún así casi dobla la anchura de *BBP*.

Para todas las configuraciones de *LTP*, alinear las trazas a bloques básicos (configuración denominada *A-LTP*) apenas reduce la anchura de las predicciones, y esta reducción es menor a medida que el valor ***N*** es mayor.

5.4.2 Precisión del Predictor

Las medidas de precisión de esta subsección sólo consideran saltos condicionales. Mostraremos que la precisión que se obtiene al predecir trazas que contienen múltiples instrucciones de salto condicional es muy similar a la que se obtiene al predecir bloques básicos. A pesar de que la configuración *LTP*(***K***=***B***) (*NTP*) tiene unos muy altos requerimientos de memoria, se

muestran los resultados de precisión como referencia para analizar el efecto de limitar el número de trazas por clase.

Se han realizado simulaciones para el mecanismo de indexación *gshare* con tablas *THT* de entre 16K y 128K entradas y para el mecanismo de indexación *e-gskew* con *THT* dividida en bancos de 32K y 64K entradas. El tamaño de *BTB* se supone ilimitado. La longitud de la historia usada para cada predictor, número de entradas en *THT* y parámetro ***N*** se muestran en la siguiente tabla. Para *gshare* se ha escogido la misma longitud *L* para todos los programas SPEC00, de forma que, de media, se obtenga la máxima precisión. La longitud de las historias para *e-gskew* se ha obtenido tras una exploración similar del espacio de diseño, pero incompleta. Se usan las mismas longitudes de historia para todos los predictores.

Tabla 5-5 Longitudes de la historia global (*L* para *gshare*, y L_0, L_1, L_2 para *e-gskew*) para cada predictor, cada número de entradas de *THT*, y cada valor de ***N***.

Entradas		**BBP**				**LTP(*K*=*B*)**				**A-LTP(*K*=*B*)**				**LTP(*K*=2)**				**A-LTP(*K*=2)**				**LTP(*K*=1)**				**A-LTP(*K*=1)**			
THT	***N*:**	**8**	**16**	**32**	**64**	**8**	**16**	**32**	**64**	**8**	**16**	**32**	**64**	**8**	**16**	**32**	**64**	**8**	**16**	**32**	**64**	**8**	**16**	**32**	**64**	**8**	**16**	**32**	**64**
16K	*gshare*	*20*	*20*	*20*	*20*	*16*	*17*	*19*	*18*	*19*	*19*	*20*	*20*	*16*	*19*	*18*	*18*	*19*	*19*	*20*	*20*	*19*	*19*	*16*	*18*	*19*	*19*	*20*	*20*
32K	*gshare*	*20*	*20*	*20*	*20*	*19*	*19*	*21*	*19*	*20*	*21*	*21*	*21*	*19*	*19*	*20*	*20*	*21*	*20*	*21*	*22*	*21*	*21*	*21*	*22*	*22*	*21*	*21*	*21*
64K	*gshare*	*22*	*22*	*23*	*21*	*24*	*22*	*23*	*24*	*24*	*24*	*24*	*24*	*21*	*22*	*21*	*23*	*24*	*24*	*24*	*23*	*23*	*22*	*23*	*22*	*23*	*24*	*23*	*22*
128K	*gshare*	*24*	*24*	*24*	*24*	*24*	*26*	*25*	*26*	*27*	*26*	*28*	*25*	*23*	*26*	*25*	*25*	*27*	*27*	*26*	*24*	*23*	*23*	*23*	*23*	*24*	*26*	*25*	*24*
3x32K	*e-gskew*	$L_0 = 10$, $L_1 = 20$, $L_2 = 40$																											
3x64K	*e-gskew*	$L_0 = 12$, $L_1 = 24$, $L_2 = 48$																											

La Figura 5.18 muestra la precisión obtenida con estas simulaciones. Al comparar los resultados entre los diferentes predictores no se pueden sacar conclusiones muy precisas, ya que en general no existen tendencias claras que se cumplan para todas las configuraciones. Las diferencias de precisión entre los predictores, fijados el resto de parámetros, son en la mayoría de los casos menores del 5%. Por tanto, predecir trazas que incluyen múltiples instrucciones de salto no es significativamente muy diferente a predecir saltos de uno en uno en forma de bloques básicos.

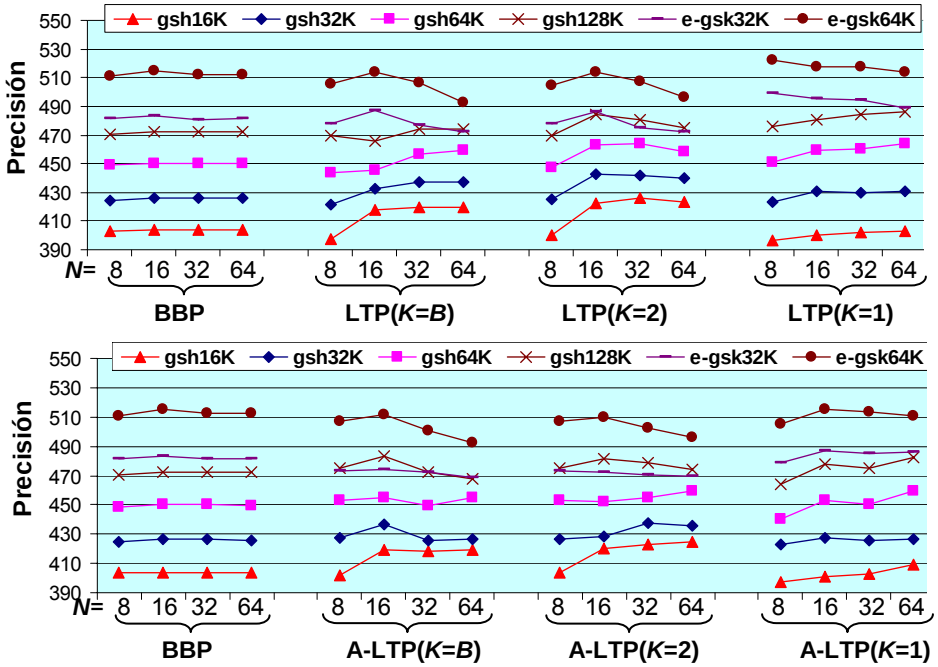


Figura 5.18. Precisión de las predicciones para diferentes configuraciones de predictor, usando las funciones de indexación *gshare* y *e-gskew*, diferentes tamaños de la tabla *THT*, y una tabla *TTB* ilimitada.

Si para cada predictor se realiza un promedio de todas las configuraciones se pueden destacar los siguientes resultados:

- Los predictores *LTP*(***K*=*B***) y *LTP*(***K***=2) proporcionan una precisión muy similar, prácticamente en todas las configuraciones, con diferencias inferiores al 2%, y una diferencia media inferior a 0,5%. Esto demuestra que limitar a 4 el número de trazas por clase afecta muy ligeramente al comportamiento del predictor.

- Los predictores *BBP* y *LTP*(***K***=1) proporcionan una precisión muy similar en todos los casos, con diferencias puntuales inferiores al 3%. De media, la configuración *LTP*(***K***=1) mejora la precisión alrededor del 0.7%.

- Los predictores *LTP*(***K*=*B***) y *LTP*(***K***=2) mejoran la precisión respecto a *BBP* alrededor del 1.5%. La mejora es importante si el tamaño del

predictor es relativamente pequeño (hasta un 7%), y decrece a medida que el predictor tiene mayor tamaño.

- Alinear trazas a bloques básicos disminuye la precisión menos del 1%.

Aunque en un primer análisis parece que la predicción de trazas llega incluso a mejorar ligeramente la precisión de la predicción de bloques básicos, un análisis más a fondo revela resultados un poco diferentes. La Figura 5.19 muestra resultados usando para cada programa la longitud de la historia que proporciona mejor precisión, con la función *gshare* y una tabla *THT* de 64 entradas. Se trata de un experimento que permite desligar los resultados de un programa de los de los otros programas. En este caso sólo se muestran datos para SPECint00, ya que los resultados para SPECfp00 eran muy similares a los de la figura anterior.

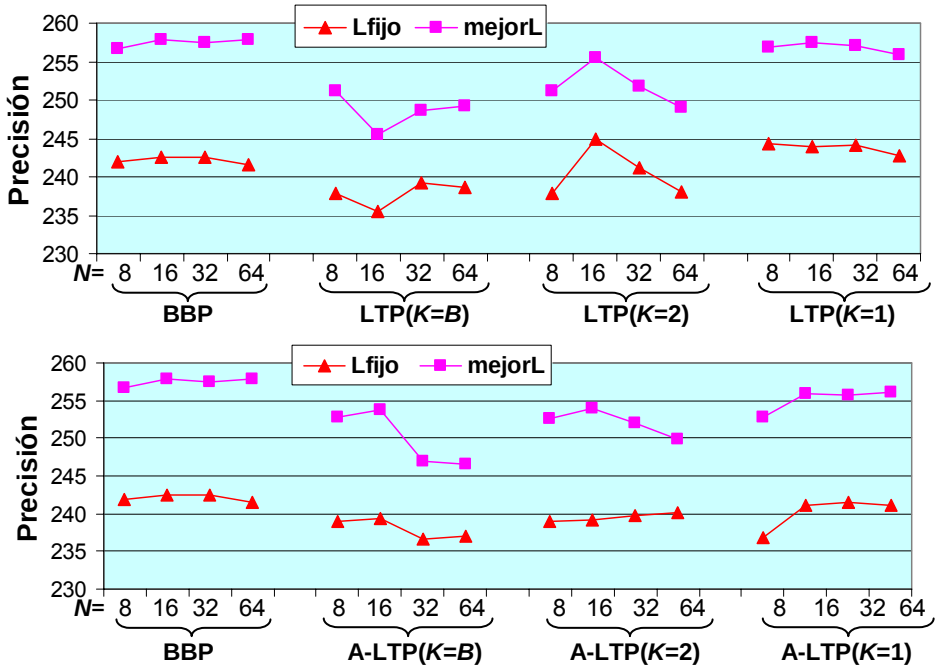


Figura 5.19. Precisión de las predicciones para diferentes predictores, con función *gshare*, tabla *THT* de 64K entradas y tabla *TTB* ilimitada. (SPECint00)
Lfijo indica que el tamaño de la historia global es el mismo para todos los programas.
mejorL indica que el tamaño de la historia es el mejor para cada programa concreto.

Resumiendo los datos anteriores:

- con la misma longitud de historia para todos los programas, *LTP*(***K*=*B***) y *LTP*(***K***=2) proporcionan una precisión para SPECint00 un 1.2% inferior a *BBP*, mientras que para SPEC00 la precisión era un 1% mejor.
- con la mejor longitud de historia para cada programa, *LTP*(***K*=*B***) y *LTP*(***K***=2) proporcionan una precisión entre el 3% y el 4% inferior a *BBP.*
- *LTP*(***K***=1) mejora la precisión de *BBP* un 0,7% con la misma longitud de historia para todos los programas, pero la precisión se iguala o es ligeramente inferior con la mejor longitud de historia para cada programa.
- Alinear trazas a BBs tiene resultados poco homogéneos, pero de media afecta sólo ligeramente a la precisión.

Finalmente, la Figura 5.20 muestra resultados para dos programas que representan dos casos extremos. El programa ***gcc*** se ve beneficiado por predecir 4 o más trazas por operación, de modo que *LTP*(***K*=*B***) y *LTP*(***K***=2) mejoran alrededor de un 15% la precisión de *BBP*. En cambio, con ***bzip***, *LTP*(***K*=*B***) y *LTP*(***K***=2) reducen alrededor de un 22% la precisión de *BBP*. Considerando cada programa en particular (todos, y no sólo ***gcc*** y ***bzip***) las diferencias de precisión entre *LTP*(***K***=*1*) y *BBP* son menores del 2%.

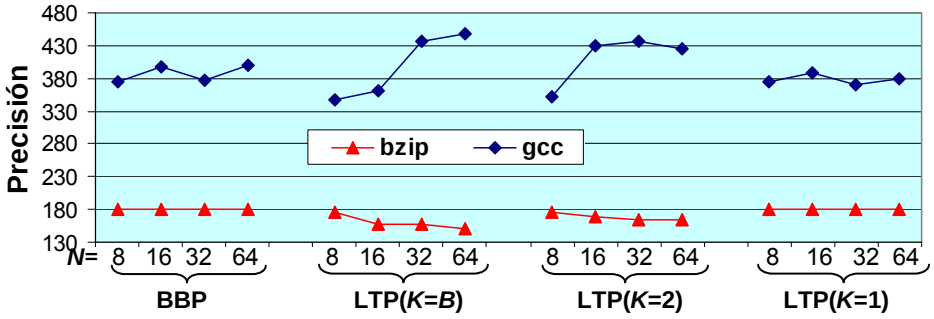


Figura 5.20. Precisión de las predicciones para diferentes predictores, con función *gshare*, tabla *THT* de 64K entradas, y tabla TTB ilimitada para los programas ***bzip*** y ***gcc***. El tamaño de la historia es el mejor para cada programa concreto.

En resumen, los resultados de predicción varían en función del programa y de los parámetros del predictor hasta un 20%. Si se consideran resultados promediados para las diferentes configuraciones y programas, entonces las variaciones son inferiores al 5%. Los predictores capaces de predecir 4 o más trazas en cada operación consiguen alcanzar una alta anchura de predicción (como se mostró en la subsección anterior) pero generalmente a costa de reducir la precisión de las predicciones entre un 1% y un 4% de media. *LTP* con un límite de 2 trazas por clase consigue casi doblar la anchura de las predicciones comparado con *BBP*, reduciendo la precisión, en promedio, un máximo del 1%.

5.4.3 Requerimientos de Memoria

Los resultados de la subsección anterior comparaban la precisión entre predictores con el mismo número de entradas en la tabla de correlación y con una tabla *TTB* ilimitada, y no consideraban explícitamente el tamaño total del predictor, en Kbytes. En esta subsección se considera el tamaño del predictor para mostrar la relación entre la precisión obtenida y los recursos de memoria utilizados. En primer lugar, se estima el número adecuado de entradas en *TTB* para cada predictor. Posteriormente, se combinan los resultados de precisión y tamaño del predictor.

Si *TTB* dispone de demasiadas pocas entradas, se produce un alto número de fallos que reduce la precisión de las predicciones de forma grave. El número adecuado de entradas es específico de cada programa, y depende de la cantidad de trazas que aparezcan en su ejecución, que además depende de las reglas de selección de trazas que se utilicen. Se ha encontrado que una tabla *TTB* de 4K entradas y con asociatividad de 8 vías proporciona prácticamente los mismos resultados que una tabla ilimitada. Así que se ha usado como referencia para comparar los resultados con tablas más pequeñas y de menor asociatividad.

La Figura 5.21 muestra el comportamiento de los fallos en *TTB* a medida que aumenta el número de entradas en la tabla. Se usa una asociatividad de 4 vías, porque proporciona resultados considerablemente mejores que el acceso directo, ligeramente mejores que una asociatividad de 2 vías, y sólo un poco inferiores a los de una asociatividad de 8 vías.

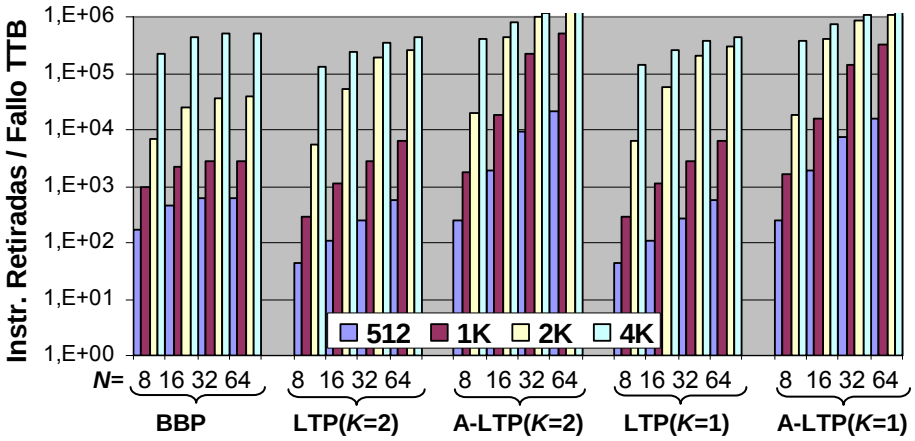


Figura 5.21. Número de instrucciones retiradas por cada fallo en *TTB* al variar el número de entradas entre 512 y 4K. *TTB* es asociativa con 4 vías.

Se pueden apreciar tres resultados importantes. En primer lugar, cuanto mayor es el tamaño de trazas y bloques básicos (***N***), menor es el número de bloques básicos y trazas diferentes, y se necesitan menos entradas en *TTB*. Segundo, representar trazas en lugar de bloques básicos reduce el número total de entradas en *TTB*. Esto se debe a que muchos bloques básicos que ocupan una entrada propia en *BTB*, al juntarse con otros bloques básicos en una traza no aparecen como punto de entrada en *TTB.* En tercer lugar, alinear las trazas a bloques básicos –forzar a que acaben en instrucciones de salto– reduce considerablemente el número de entradas necesarias. Esto se debe a que se evita un problema bien conocido que ocurre en ciertos bucles, [PaFP99]. Si las trazas siempre tienen tamaño ***N***, y el número de instrucciones del bucle, por ejemplo ***I*****,** no es un divisor exacto de ***N***, entonces aparecen dinámicamente múltiples trazas que contienen las mismas instrucciones pero

desplazadas dentro de la traza. Suponiendo que ***I*** = ***N***-1, aparecerá la traza que comienza en la instrucción 0 y finaliza en la instr. 0, la traza que comienza en la instr. 1 y finaliza en la instr. 1, … Al alinear la traza a un bloque básico, sólo habrá una traza que empiece en la instrucción 0 y acabe en la instrucción ***N***-1.

De todos modos, el dato que realmente interesa para determinar el número adecuado de entradas en TTB es la precisión final obtenida por el predictor. La Figura 5.22 muestra la precisión normalizada a la obtenida con una *TTB* de 4K entradas. La normalización se hace para cada predictor y tamaño de ***N***. De este modo se puede comparar para cada configuración el efecto de reducir el número de entradas en *TTB* a 2K, 1K o 512.

Un primer resultado es que el efecto de los fallos de ***TTB*** no se aprecia en la precisión cuando es menor de 1 por cada 500.000 instrucciones retiradas. Cuando ***N*** es 32 o mayor, alinear trazas a bloques básicos permite reducir el tamaño de *TTB* a la mitad. Así, mientras que *BBP* necesita 4K entradas para evitar reducir la precisión un 2%, *A-LTP* puede usar una *TTB* de 2K e incluso de 1K entradas sin reducir la precisión.

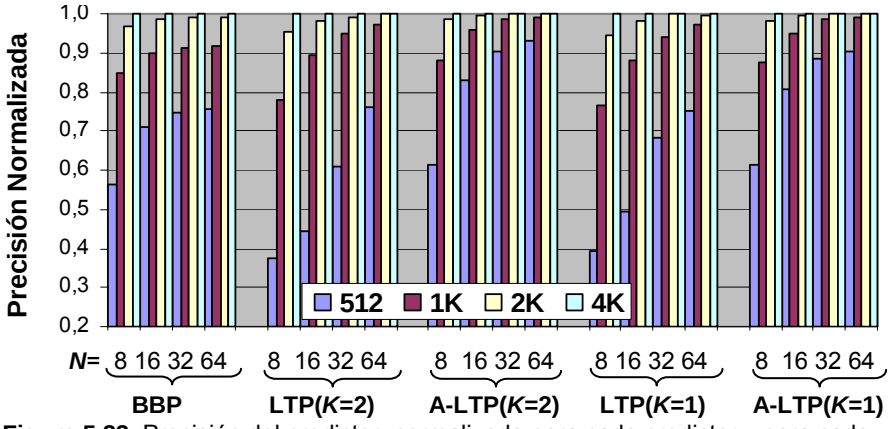


Figura 5.22. Precisión del predictor, normalizada para cada predictor y para cada tamaño máximo de las trazas (***N***), al variar el número de entradas en *TTB* entre 512 y 4K. La tabla *TTB* es asociativa con 4 vías.

Con estos resultados se calculó el tamaño adecuado de los predictores que se mostraron en la Figura 5.11. Cuando *THT* tiene pocas entradas, se usa una *TTB* también con pocas entradas, reduciendo ligeramente la precisión pero ahorrando el coste relativamente más alto de las entradas en *TTB*. De este modo se obtiene una adecuada relación entre precisión y requisitos de memoria. El resultado de cruzar los datos sobre precisión con los datos sobre el tamaño del predictor se mostró en la Figura 5.12 y se vuelve a mostrar a continuación.

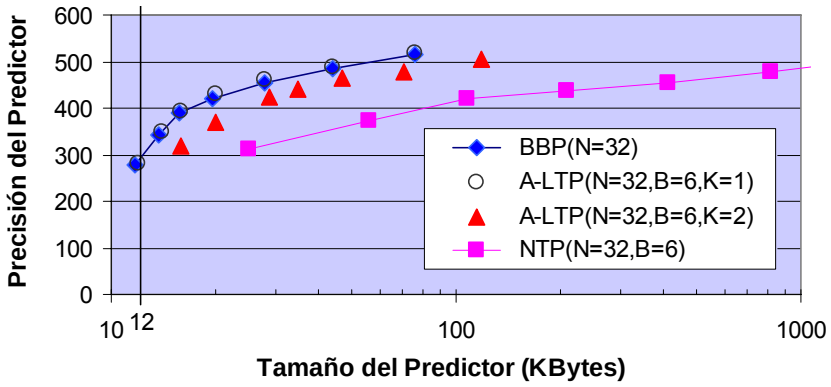


Figura 5.23. Relación entre precisión y requisitos de memoria para los predictores *BBP, LTP* y *NTP*, con diferentes tamaños de las tablas.

En resumen, *BBP* y *LTP*(***K***=1) tienen una relación muy similar entre precisión y memoria, pero *LTP*(***K***=1) proporciona un 80% más anchura que *BBP*, y por tanto aumenta el ancho de banda de las predicciones sin ningún coste adicional en precisión ni en memoria. *LTP*(***K***=2) tiene una peor relación precisión/memoria que los dos anteriores pero, a cambio, proporciona anchuras de predicción bastante superiores. Finalmente, *NTP* muestra, para tamaños superiores a 12 KBytes, una muy baja relación precisión/memoria con respecto a los otros tres diseños, de modo que la gran anchura de predicción que teóricamente puede lograr sería a costa de aumentar los recursos de memoria de forma inaceptable.

5.4.4 Latencia del Predictor

Antes de tomar medidas del rendimiento del procesador es necesario considerar la latencia de los predictores. No se trata de obtener valores absolutos de la latencia, sino de comparar la latencia de los predictores analizados en este capítulo.

La latencia de los predictores se puede estimar como el tiempo de acceso a las tabla internas más el tiempo necesario para los mecanismos de indexación y selección de información. Para hacer una comparación justa entre los predictores, tal como se ha argumentado previamente, se considera el mismo mecanismo de indexación en todos los casos, y por tanto se puede ignorar al comparar las latencias.

Respecto al mecanismo de selección, habría muy pocas diferencias entre *BBP* y *LTP*(***K***=1). En ambos casos hay que seleccionar entre tres casos: las dos trazas proporcionadas por el predictor, y la traza obtenida al recuperarse de un fallo de predicción. *LTP*(***K***=2) requiere seleccionar entre 4+1 casos, y esto podría aumentar la latencia ligeramente. El mecanismo de selección en la propuesta original de *NTP* también debía escoger entre dos trazas, una de cada tabla, y sería de una complejidad similar a *BBP* y *LTP*(***K***=1).

Considerando el tiempo de acceso a las tablas, los predictores *BBP* y *LTP*(***K***=1) vuelven a ser muy similares. Las tablas *BHT* y *THT* son idénticas, y las tablas *BTB* y *TTB* tienen el mismo tamaño total, aunque *TTB* tiene la mitad de entradas. Esta diferencia debería beneficiar ligeramente a *TTB* para ciertos tamaños de la tabla, ya que se reduce el tiempo de decodificación, que suele ser más crítico, pero consideraremos que su tiempo de acceso es el mismo. Las entradas en *THT* y *TTB* de la configuración *LTP*(***K***=2) tienen más bits que *BBP* y *LTP*(***K***=1), y por tanto el predictor necesita más memoria (alrededor de un 40%-60% más). Esta diferencia aumentaría la latencia de ***LTP***(***K***=2). Los altos requerimientos de memoria de *NTP* supondrían un incremento considerable en la latencia.

5.4.5 Caché de Trazas

El modelo de la unidad de búsqueda que se ha utilizado para analizar el efecto de los diferentes predictores está basado en la caché de trazas (*Trace Cache*, *tCache*) [PeWe94][RoBS96]. El uso de la caché de trazas simplifica la búsqueda de instrucciones y proporciona un alto ancho de banda, a pesar de la ocurrencia de frecuentes saltos tomados.

En la memoria caché de instrucciones clásica, compuesta de bloques con instrucciones en direcciones contiguas, no es suficiente con el patrón de saltos para determinar exactamente qué instrucciones componen una traza, sino que se necesitan la posición dentro de la traza y la dirección destino de los saltos intermedios. Estas direcciones intermedias se podrían obtener de alguna tabla asociada al predictor, del modo que se propone para la *Block-Based Trace Cache* [BIRS99].

Por el contrario, si las instrucciones se almacenan en memoria en forma de trazas, entonces las direcciones intermedias se pueden obtener a partir de la información contenida en la propia *tCache*, y no es necesario almacenarlas en la tabla del predictor principal. Además, incluyendo en *tCache* información adicional de control (decodificación, renombrado, …), se reduce el retardo desde la lectura de las instrucciones hasta la fase de ejecución, y se puede también reducir el consumo energético. La caché de trazas también posibilita otros tipos de optimizaciones, como las propuestas en [RMSR03] y [RAM+04].

Sin embargo, almacenar las instrucciones en forma de traza genera redundancia y fragmentación, lo que la hace menos eficiente que una memoria caché tradicional. Una forma de reducir este problema ya ha sido comentada, y es alinear las trazas a bloques básicos. Los experimentos realizados indican que si las trazas son de 8 instrucciones la redundancia es aceptable, pero con trazas de 32 o 64 instrucciones y un tamaño realista de caché, la cobertura de la memoria caché para ciertas aplicaciones es prácticamente nula.

Para reducir la fragmentación y la redundancia se ha decidido, por una parte, limitar el tamaño máximo de las trazas a 32 instrucciones, y por otra, almacenar las trazas en forma de hasta 4 bloques independientes de 8 instrucciones cada uno. De este modo, la fragmentación se limita a un peor caso de 7 instrucciones por traza. Además, los bloques de 8 instrucciones comunes a varias trazas se almacenan una sola vez. La Figura 5.24 muestra el diagrama de bloques del diseño de unidad desacoplada presentado en la Figura 3.2 adaptado a una caché de trazas. Al igual que en la unidad de búsqueda desacoplada descrita en la sección 3.4, la caché de trazas se accede de forma serie para reducir el consumo energético.

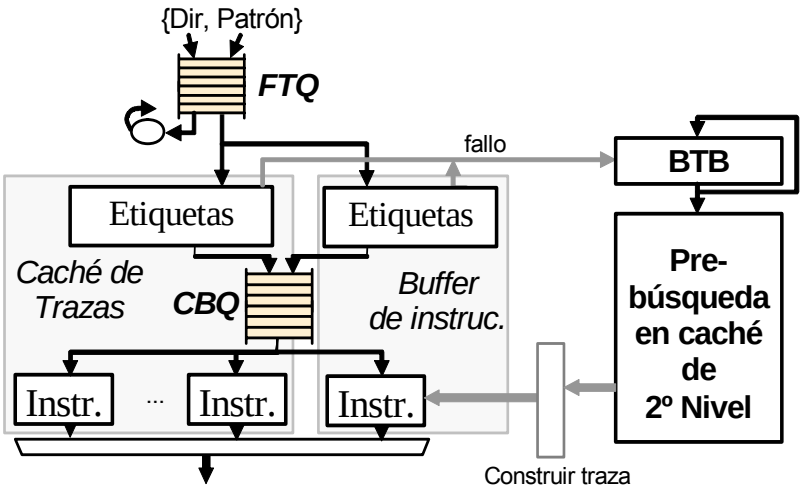


Figura 5.24. Unidad de Búsqueda de Instrucciones en una organización desacoplada basada en trazas, con prebúsqueda de instrucciones dirigida por el predictor, y con acceso serie a las trazas.

Cuando se produce un fallo en *tCache*, se envía la petición de la traza a la caché de segundo nivel, con organización tradicional. Para acelerar el proceso de la construcción de la nueva traza se utiliza una tabla *BTB* especial que contiene las direcciones destino de los saltos internos a las trazas. El patrón de saltos de la traza se utiliza para guiar la construcción de la nueva traza, que se almacena en primer lugar en el buffer de prebúsqueda.

Dividir las trazas de hasta 32 instrucciones proporcionadas por el predictor, en bloques de trazas de 8 instrucciones, crea el problema de generar las direcciones de los hasta 3 bloques de trazas intermedios. En el modelo que se ha simulado se ha supuesto que estas direcciones se obtienen de una tabla especial. Es posible un diseño, más eficiente en el uso de memoria, que substituya las direcciones de los bloques intermedios por apuntadores a los bancos de los bloques de instrucciones. La discusión de esta posibilidad queda fuera del ámbito de este trabajo.

5.4.6 Rendimiento del Procesador

La Tabla 5-6 muestra las características más importantes del procesador que se ha simulado. Los datos no mostrados en la tabla se pueden encontrar en la sección 3.3.

Tabla 5-6 Características más importantes de la microarquitectura del procesador.

Unidad de Búsqueda	Núcleo de Ejecución	Sistema de Memoria
Predicción perfecta de saltos indirectos y de retorno de subrutina **Penalización fallo**: 16 ciclos mínimo **Prebúsqueda** en tCache	**Decodificación**: 6 instr. /ciclo **Ejecución**: 6 instr. / ciclo **Reorder Buffer**: 240 entradas **Cola Instrucc.**: 120 entradas **Cola de Load's**: 120 entradas **Cola de Store's**: 90 entradas	**tCache**: 32KB, 3 ciclos (4 vías x 32B) **dCache:** 16KB, 2 ciclos **L2-Cache**: 512KB, + 6 ciclos **L3-Cache**: 16 MB, + 32 ciclos **Memoria**: + 120 ciclos

La Tabla 5-7 muestra las características básicas de los cuatro predictores que se van a utilizar para medir el rendimiento del procesador. Tal como se ha comentado, *LTP* usa las versiones con trazas alineadas a bloques básicos (*A-LTP*) y con un tamaño máximo de la traza de 32 instrucciones. Los cuatro predictores tienen un tamaño similar y por tanto tendrían una latencia similar (excepto la configuración *A-LTP*(***K***=2)).

No se considera *NTP* en estas simulaciones de rendimiento debido a su baja relación entre precisión y memoria, ya que para obtener una precisión comparable a los cuatro predictores de la tabla, necesitaría tener un tamaño mucho mayor (alrededor de 140 KBytes) y por tanto una latencia bastante

más elevada. Determinando esta latencia, los resultados de rendimiento se podrían extrapolar fácilmente.

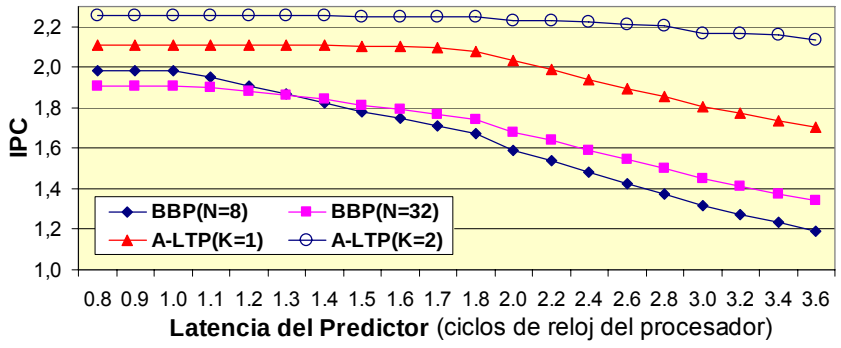
Tabla 5-7 Características del rendimiento de cuatro predictores seleccionados.

	Tamaño (KB)	***Latencia Relativa***	***Precisión (instr/fallo)***		***Anchura (instr/pred)***	
			SPECint	***SPECfp***	***SPECint***	***SPECfp***
BBP(***N***=*8*), 2K-BTB, 64K-BHT	***27.25***	***1***	***241.4***	***1770.5***	***5.2***	***6.6***
BBP(***N***=*32*), 2K-BTB, 64K-BHT	***27.75***	***1***	***241.9***	***1781.2***	***7.6***	***16.6***
A-LTP(***N***=*32*,***B***=*6*,***K***=*1*), 1K-TTB, 64K-THT	***28.00***	***1***	***242.3***	***1796.4***	***12.2***	***20.1***
A-LTP(***N***=*32*,***B***=*6*,***K***=*2*), 1K-TTB, 64K-THT	***38.75***	***>1***	***239.7***	***1809.2***	***18.5***	***23.8***

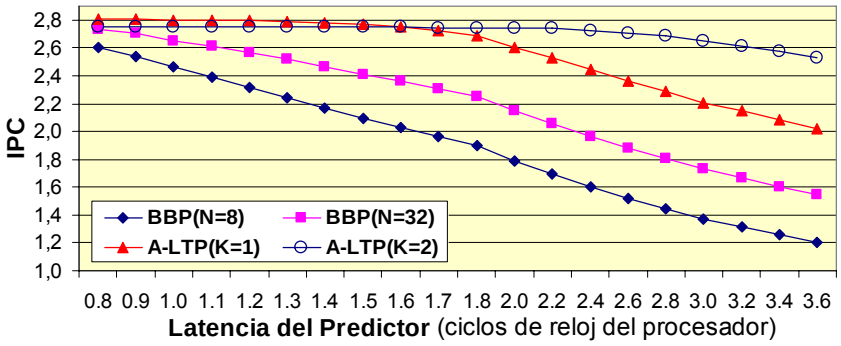
La Figura 5.25 muestra el rendimiento del procesador para los cuatro predictores, con diferentes latencias en ciclos de reloj del procesador. Se está suponiendo que el predictor está controlado por un reloj independiente, cuya frecuencia no necesita ser un múltiplo o divisor exacto de la frecuencia con que funciona el resto del procesador. Precisamente la cola *FTQ* permite desacoplar estas velocidades.

La Figura 5.25(a) simula una unidad de búsqueda acoplada al utilizar una cola *FTQ* de tamaño 1. La Figura 5.25(b) simula una cola *FTQ* de 16 entradas, que realmente desacopla el predictor. En estas dos figuras se muestran resultados promediados para los programas SPECint00. La Figura 5.25(c) muestra los resultados con una cola *FTQ* de 16 entradas para los programas SPECfp00.

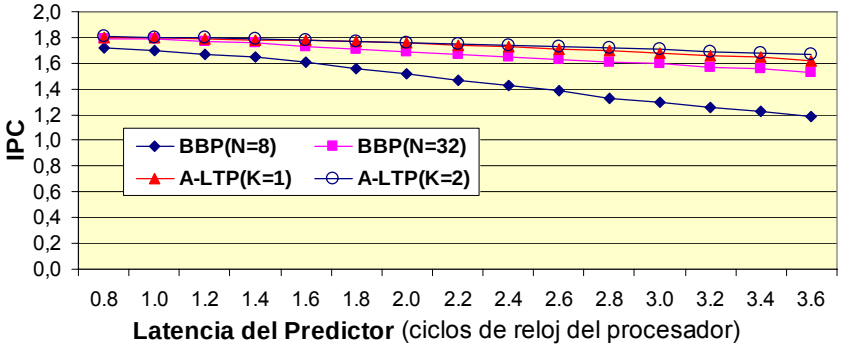
Las dos conclusiones fundamentales que se pueden extraer de estos resultados son que (1) la organización desacoplada utiliza mucho mejor la mayor anchura de los predictores que la organización acoplada, y que (2) el efecto negativo de tener predictores con una alta latencia se compensa con una mayor anchura de las predicciones. A continuación discutiremos estos dos resultados con más detalle, y en este mismo orden.



(a) front-end acoplado (tamaño FTQ = 1 entrada). SPECint00



(**b**) front-end desacoplado (tamaño FTQ=16 entradas). SPECint00



(c) front-end desacoplado (tamaño FTQ= 16 entradas). SPECfp00

Figura 5.25. IPC para la microarquitectura descrita en la Tabla 5-6 y los predictores descritos en la Tabla 5-7, suponiendo diferentes latencias de predicción.

Evaluación de la Organización Desacoplada

Las figuras (a) y (b) comparan el efecto de usar una organización desacoplada. Fijando la latencia del predictor a un ciclo de reloj, y dependiendo del predictor que se utilice, se incrementa el IPC entre un 20% y un 35%. El menor rendimiento de la organización acoplada se debe a no ser capaz de solventar de manera efectiva el problema creado por la fragmentación en los bloques de trazas y por los fallos en la caché de trazas. En la sección 4.8.2 se proporcionaron argumentos para sostener este comportamiento. A continuación comentaremos únicamente los detalles particulares del uso de un caché de trazas (*tCache*).

El tamaño de *tCache* (32KBytes) es bastante pequeño, y genera un alto porcentaje de fallos en algunas aplicaciones. La redundancia interna (una misma instrucción almacenada múltiples veces) hace que la efectividad de *tCache* sea similar al de una caché tradicional de 16KB, o en algunos casos menor. Se puede comprobar en la Figura 5.25(a) que *BBP*(***N***=8) supera a *BBP*(***N***=32) en un sistema acoplado cuando la latencia es de 1 ciclo. La razón es que los bloques básicos más cortos generan menos redundancia, y se producen menos fallos de caché. Se ha escogido *tCache* precisamente de este tamaño para evaluar la capacidad de la organización desacoplada.

El problema de la fragmentación se produce al leer un fragmento de traza que contiene menos de las 6 instrucciones que es posible decodificar cada ciclo. La organización desacoplada compensa trazas de tamaño menor a 6 con otras de tamaño mayor. Además, se puede anticipar a los fallos de caché buscando instrucciones en el segundo nivel antes de que se necesiten en la etapa de decodificación, y gestionando múltiples peticiones en paralelo.

Efecto de la Latencia y de la Anchura del Predictor en el Rendimiento

La curva descendente de la Figura 5.25(b) muestra que incrementar la latencia del predictor reduce de forma considerable el rendimiento. Esta observación ya ha sido discutida con detalle en el capítulo anterior. La

aportación de este experimento es mostrar que, aunque la degradación del rendimiento se produce con todos los predictores, es sustancialmente más acuciada para los predictores con una menor anchura de predicción. En otras palabras, una mayor anchura de predicción permite tolerar una mayor latencia de predicción.

Por ejemplo, *BBP*(***N***=8) necesita tener una latencia menor a 0,8 ciclos para alcanzar el mismo rendimiento que *A-LTP*(***K***=2) funcionando con una latencia de 3 ciclos. En otro sentido, si las latencias de los predictores son de un ciclo de reloj, entonces *BBP*(***N***=32) mejora el rendimiento un 7,9% respecto a *BBP*(***N***=8), y a su vez, *A-LTP*(***K***=1) mejora el rendimiento un 5,5% respecto a *BBP*(***N***=32). Si las latencias de los tres predictores fueran 2 ciclos, entonces las mejoras de rendimiento serían de alrededor del 20% en ambos casos.

La Figura 5.25(c) muestra los resultados del mismo experimento que el de la figura (b) pero para los programas SPECfp00. Puesto que estos programas contienen bloques básicos mucho más largos, *BBP*(***N***=32) es capaz de proporcionar una anchura de predicción de 16,5 (ver Tabla 5-7), y la ganancia que se puede obtener con *A-LTP* es mucho más limitada. Adicionalmente, la alta precisión de la predicción de saltos en los programas SPECfp00 y el bajo rendimiento que se obtiene con ellos (ya que muchos están limitados por la latencia a los datos de memoria) hace que la anchura del predictor tenga una importancia mucho menor. De todos modos, reemplazar el predictor *BBP*(***N***=32) por *A-LTP*(***K***=1) cuando ambos tienen una latencia de 2 ciclos mejora el rendimiento alrededor de un 4%. La mejora es del 15,9% si se reemplaza el predictor *BBP*(***N***=5) por *A-LTP*(***K***=1).

5.5. Conclusiones

La latencia del predictor medida en ciclos de reloj depende de varios aspectos, pero uno de los más importantes es el tamaño del predictor, que también determina la precisión del predictor. El diseñador debe decidir entre

incrementar la precisión del predictor, dedicándole más memoria y aumentando su latencia, o usar menos memoria para reducir su latencia pero a costa de una menor precisión. Las estrategias de organización jerárquica y segmentada para permitir usar más memoria en el predictor y mantener la latencia (analizadas en el capítulo previo) están limitadas por una complejidad que crece de forma exponencial. Es decir, son estrategias que no se pueden escalar indefinidamente.

En este contexto, aparece la posibilidad de aumentar el ancho de banda del predictor de una manera diferente, que es incrementando la anchura de cada predicción. Hemos demostrado que una mayor anchura de predicciones compensa una mayor latencia de las predicciones (y viceversa). También hemos comprobado que el ancho de banda total del predictor se puede convertir en cuello de botella para un procesador realista con aplicaciones de referencia estándar. El ancho de banda del predictor es también útil para utilizar estrategias que utilizan el "conocimiento del futuro" para anticipar tareas que aumenten el rendimiento, reduzcan el consumo energético, o ambas cosas a la vez. Por ejemplo, hemos probado que es especialmente útil para ocultar la penalización de los fallos de la memoria caché de trazas.

Al aumentar la anchura de las predicciones hemos de asegurar que el resto de características del rendimiento del predictor no empeoran. En concreto, hemos de mantener su precisión y su latencia. Por supuesto, siempre se han de considerar todos estos parámetros de forma relativa a la cantidad de recursos (memoria) que se dedica al predictor.

Como punto de partida de la investigación se ha utilizado el algoritmo de predicción de trazas denominado *Next-Trace Predictor* (*NTP*), propuesto por Jacobson et al. [JaRS97]. Predecir trazas tiene múltiples ventajas, quizás la más importante es que la complejidad del predictor no depende del número de instrucciones de transferencia de control que contenga una traza. Sin embargo, *NTP* no escala la precisión con los requerimientos de memoria al mismo ritmo que lo hace un algoritmo de predicción basado en bloques básicos (*BBP*). Por tanto, el aumento de la anchura de las predicciones que

proporciona *NTP* se hace a costa de, o bien disminuir la precisión, o bien aumentar los recursos de memoria y por lo tanto la latencia del predictor.

La principal aportación de este capítulo es proponer un nuevo algoritmo de predicción, que denominamos predictor de trazas locales (*Local Trace Predictor, LTP*), que, fijados los recursos de memoria, proporciona una precisión similar a *BBP*, pero con la ventaja añadida de una mayor anchura de las predicciones. Un parámetro básico de este algoritmo de predicción es el límite que se impone al número de trazas que pueden comenzar por una misma instrucción. Utilizar un límite de 2 trazas produce una configuración con unas prestaciones prácticamente idénticas a *BBP* pero que mejora la anchura de predicción alrededor de un 80% para los programas SPECint00. Aumentar el límite a 4 trazas aumenta considerablemente la anchura de las predicciones –hasta un 160%–, pero a cambio de aumentar los requerimientos de memoria algo más de un 50%.

En el capítulo final de esta memoria se discutirán las posibles líneas generales de investigación que se abren a partir de este trabajo. Una línea bastante evidente consiste en aplicar las técnicas propuestas en el capítulo anterior para reducir la latencia de *LTP*. La otra línea consiste en aprovechar el paralelismo que proporciona un predictor con un elevado ancho de banda para diseñar una caché de trazas que use de forma más eficiente su capacidad de memoria y su consumo energético.

Dos ampliaciones más especializadas a la propuesta de LTP son el estudio de políticas alternativas de selección de trazas y de políticas alternativas para el manejo de los desbordamientos de clase.

6. Predicción Eficiente de Saltos Indirectos

Resumen

Se propone un método de codificar las direcciones destino de los saltos indirectos que proporciona una mejor relación entre la precisión de las predicciones y la cantidad total de bits dedicados a las predicciones. Esto permite aumentar la precisión del predictor o reducir sus requerimientos de memoria.

6.1. Introducción

La predicción de saltos indirectos es un factor que limita las prestaciones en ciertas aplicaciones. Los saltos indirectos utilizan una dirección de salto, que a partir de ahora denominaremos *dirección destino*, que frecuentemente es calculada por el programa en base a sus datos de entrada. Durante la ejecución de un programa, cierta parte de los saltos indirectos tienen siempre la misma dirección destino, y en este caso se les denomina *monomórficos*. Este subconjunto de saltos ofrece poca dificultad al predictor, pues se comportan exactamente igual que los saltos directos incondicionales.

La dificultad aparece para los saltos indirectos con múltiples direcciones destino, a los que se denomina *polimórficos*. Las aplicaciones ***perlbmk*** y ***gcc***, por ejemplo, contienen algún salto indirecto con hasta 179 direcciones destino diferentes. La predicción precisa de los saltos indirectos polimórficos impone más requerimientos que la predicción de saltos condicionales:

- para cada instrucción de salto indirecto se han de almacenar múltiples direcciones destino.
- la resolución del salto no se puede codificar con un valor binario.

Las estrategias más recientes propuestas en la literatura combinan los bits de la dirección del salto indirecto con la historia global de los saltos más recientes para generar el índice con el que acceder a una tabla específica, que denominamos *IJT* (*Indirect Jump Table*), y obtener la dirección destino. *IJT* puede almacenar múltiples direcciones destino para un mismo salto indirecto, una dirección para cada historia previa al salto indirecto. Para aumentar la precisión de las predicciones se debe aumentar el tamaño de la historia utilizada con *IJT*, y también aumentar el tamaño de *IJT*, para así dar soporte al mayor número de combinaciones posibles. El problema de este método es que no se escala de forma eficiente al aumentar el tamaño de *IJT*, pues las mismas direcciones destino se almacenan múltiples veces.

La propuesta que se describe en este capítulo consiste en codificar la información del predictor de modo que se evite la duplicación de las direcciones destino. Para ello se utiliza un esquema de dos niveles. El primer nivel almacena identificadores que codifican cada dirección destino de forma local al salto indirecto que la utiliza. El segundo nivel del predictor decodifica esos identificadores, y los convierte en la dirección destino completa. Como los identificadores locales ocupan entre 4 y 8 veces menos bits que las direcciones destino, con el mismo número total de bits se pueden tener más entradas en la tabla del primer nivel, y esto permite aumentar la precisión. La tabla de segundo nivel contiene direcciones completas, pero sin duplicar, y de este modo el número total de entradas necesarias en la tabla es pequeño. Mostraremos datos experimentales que indican que un número razonable es entre 64 y 128 entradas.

Un posible problema de la organización de dos niveles en el predictor de saltos indirectos es incrementar la latencia. Para reducir este problema se propone un diseño similar al de la predicción de vía presentado en el capítulo 4, que permite que la tabla del segundo nivel tenga una organización con alta asociatividad (y así ser más pequeña pero con las mismas prestaciones) y con acceso directo (y así ser más rápida). De este modo, el incremento de la latencia del predictor es poco importante. Además, se mostrarán resultados indicando que incrementar esta latencia uno o dos ciclos tiene un efecto muy reducido en el rendimiento del procesador. Un gran porcentaje de este retardo se oculta gracias a la organización desacoplada del predictor y a que los saltos indirectos típicamente están relativamente alejados del salto que ha provocado el último fallo de predicción.

A lo largo del capítulo mostraremos que la propuesta reduce la redundancia en el predictor de saltos indirectos y consigue:

- obtener la misma precisión, y por tanto el mismo rendimiento, dedicando al predictor una menor cantidad de memoria.

- obtener mayor precisión, y por tanto mayor rendimiento, dedicando al predictor la misma cantidad de memoria.

El incremento en el rendimiento del procesador debido a la mejora en la precisión del predictor dependerá de la proporción de saltos polimórficos que contengan las aplicaciones ejecutadas. Las aplicaciones codificadas con lenguajes orientados a objeto, tal y como se explicó en el capítulo 2, tienen una alta proporción de saltos polimórficos, mientras que los programas de evaluación SPEC-CPU00, en general, tienen una baja proporción. En todo caso, existen dos diseños comerciales actuales, el Intel Pentium IV de 90nm [BBH+04] y el Intel Pentium M [GRB+03], que incluyen un mecanismo específico para predecir con mayor precisión estos saltos, muy similar a la propuesta de referencia usada en este trabajo. Estos dos procesadores se podrían beneficiar de la propuesta que se va a presentar en este capítulo.

La mayor precisión del predictor también se puede utilizar para reducir el consumo energético, ya que se reduce la cantidad de trabajo realizado en el camino erróneo durante las ejecuciones especulativa. Por otro lado, reducir el tamaño de memoria del predictor, manteniendo la precisión, permite reducir el consumo del predictor.

6.1.1 Esquema del Capítulo

El esquema de este capítulo es el siguiente. En primer lugar, se presenta el diseño que se toma como punto de partida, y se discute el problema con detalle, dando pié a una descripción general de la nueva propuesta. En segundo lugar se describe con precisión la nueva organización con dos niveles del predictor. A continuación se presentan los resultados de precisión del predictor, que se combinan con el tamaño del predictor para mostrar que se obtiene una mejor relación entre precisión y memoria utilizada. También se analiza el efecto en el rendimiento de esta nueva propuesta. Se finaliza el capítulo con las conclusiones del estudio y con las líneas futuras que plantea.

6.2. Motivación y Presentación de la Propuesta

En primer lugar describiremos un predictor básico de saltos indirectos, fundamento de las demás propuestas, y después una variación que permite mejorar sus prestaciones, y que hemos adoptado como referencia de partida en nuestro trabajo. Con un ejemplo analizaremos el problema que supone tener que almacenar múltiples direcciones destino, y este análisis servirá para introducir nuestra propuesta y las ideas básicas en las que se basa. En la sección siguiente se describirá la nueva propuesta con todo detalle.

6.2.1 Predicción de Saltos Indirectos con una Tabla Única

Un predictor de saltos indirectos que permite conseguir una precisión creciente para una cantidad también creciente de recursos es la *Target Cache* [ChHP97]. La Figura 6.1 muestra el esquema de este predictor. La historia global codifica el resultado de los últimos saltos. Esta historia se combina con la dirección del salto indirecto que se quiere predecir para obtener un índice con el que acceder a *IJT* (*Indirect Jump Table*). La función ***f*** trata de dispersar al máximo los posibles valores de entrada a lo largo de *IJT*, para así maximizar su utilización. La dirección destino obtenida al leer la tabla se envía al procesador segmentado, que al cabo de algunos ciclos devolverá la dirección destino correcta, para actualizar (o tal vez corregir) la predicción.

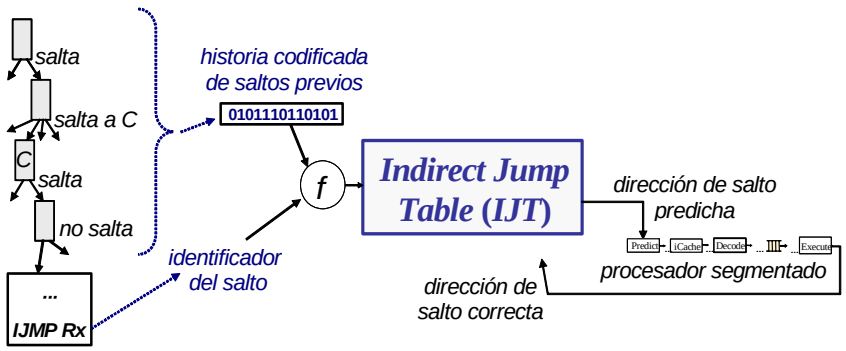


Figura 6.1. Predicción de saltos indirectos con una tabla única.

La Figura 6.2 muestra el ejemplo de una instrucción de salto indirecto con tres direcciones destino diferentes. A medida que se ejecuta la instrucción, se registra en *IJT* la dirección destino que toma un salto indirecto para cada uno de los caminos diferentes por los cuales se llega a ella. Una parte del índice a *IJT* se almacena como etiqueta (*tag*) en la propia tabla, para poder así verificar en la fase de predicción que el contenido de la tabla, con alta probabilidad, corresponde a la combinación de entrada deseada. Se puede observar en el ejemplo de la figura que *IJT* contiene las mismas direcciones destino repetidas varias veces. Esta redundancia provoca que sean necesarias muchas entradas en *IJT* para alcanzar una alta precisión.

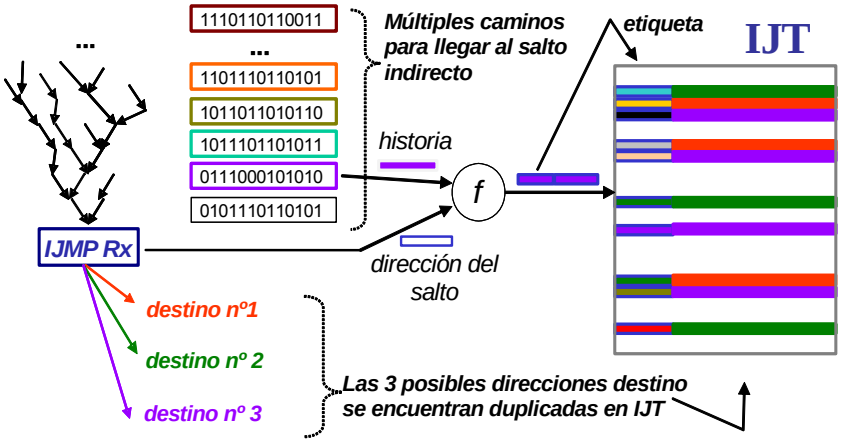


Figura 6.2. Ejemplo que muestra la redundancia existente en *IJT*.

6.2.2 Predicción de Saltos Indirectos con dos Tablas en Cascada

El esquema en cascada propuesto por Driesen y Hölzle en [DrHo98b], y descrito como predictor de referencia en el capítulo 3, tiene como objetivo mejorar el uso de la memoria del predictor y mejorar así la relación entre precisión y memoria. *BTB*, que normalmente contiene las direcciones destino de los saltos directos, se usa como filtro al predictor específico de saltos indirectos. En *BTB* se almacena la primera dirección destino que toma el

salto indirecto, que denominaremos la *dirección destino base*. Mientras no se produzca un fallo de predicción, la dirección destino de este salto indirecto siempre se predecirá usando *BTB*, evitando usar *IJT*. Así, la predicción de los saltos indirectos monomórficos siempre acertará, y no se consumirá espacio de la memoria de *IJT*.

Cuando se produzca un fallo de predicción, se marcará que el salto indirecto es ahora polimórfico, y se comenzará a usar *IJT* para este salto. De todos modos, en *IJT* sólo se almacenarán las direcciones destino diferentes a la dirección base (guardada en *BTB*). Por defecto, si un camino no se encuentra en *IJT* entonces se usa la predicción de *BTB*, y por tanto, para asociar un camino con la dirección destino base, simplemente se debe no almacenar en *IJT*.

La Figura 6.3 muestra el mismo ejemplo que la figura anterior, pero esta vez utilizando la tabla *BTB* en cascada. Se puede apreciar que la dirección destino base, almacenada una única vez en *BTB*, no aparece en *IJT*. De este modo se filtra una cantidad razonable de información redundante en *IJT*.

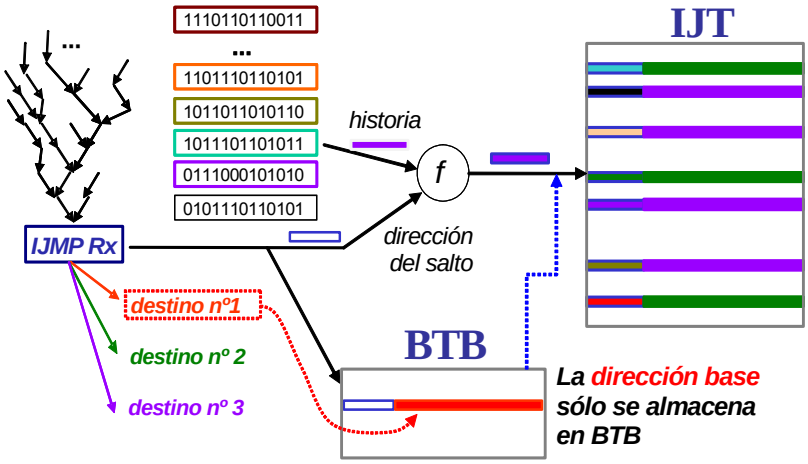


Figura 6.3. Predicción de saltos indirectos con dos tablas en cascada. La dirección destino seleccionada como base no se necesita almacenar en *IJT*.

Tal y como se propone en [DrHo99] o en [KaKa98], se pueden utilizar más de dos niveles de tablas para reducir aún más la duplicación de direcciones. Sin embargo, la pequeña mejora de prestaciones no compensa el incremento en la complejidad del diseño. A continuación se propone otra estrategia para utilizar la memoria de *IJT* de una forma más eficiente.

6.2.3 Codificación Local de Direcciones Destino

La redundancia existente en *IJT* es implícita a la intención de hacer corresponder una entrada con cada camino diferente por el cual llegar a cada salto indirecto. Esta idea es la misma que se utiliza en *BHT* para los saltos condicionales. La diferencia básica es que en *BHT* sólo se almacenan dos bits por entrada, uno codificando el sentido del salto condicional (saltar o no), y otro para establecer histéresis en el predictor. En *IJT* se almacena la dirección destino (28 bits), una etiqueta (16 bits) y un bit de histéresis.

La propuesta presentada en este capítulo consiste en sustituir la dirección destino por un identificador local (***dirID***) para esta dirección. La siguiente figura muestra el esquema de esta propuesta con el mismo ejemplo que se ha ido utilizando. En *IJT* se sustituyen direcciones por identificadores, mucho más pequeños. La conversión de ***dirID*** a la dirección destino completa se hace con una segunda tabla, que denominamos *ITT* (*Indirect Target Table*).

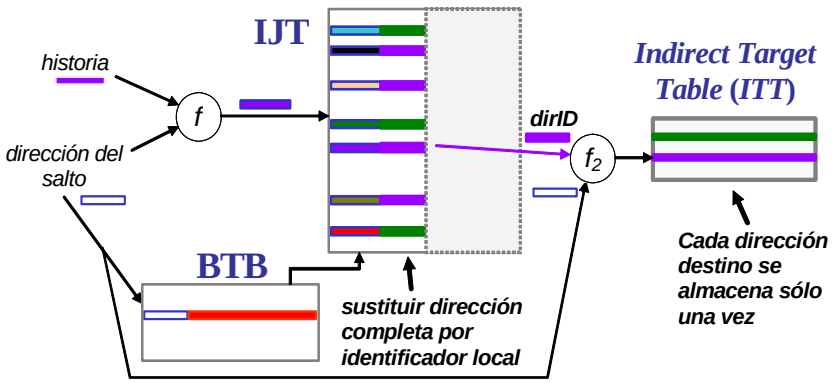


Figura 6.4. Predicción de saltos indirectos con codificación local de direcciones.

El término local indica que el valor ***dirID*** es relativo al salto indirecto al que corresponde. En otras palabras, para determinar la dirección destino completa es necesario tanto el valor ***dirID*** como la dirección del salto indirecto (o la dirección del bloque básico que lo contiene). La codificación local permite reducir el tamaño de los identificadores, que deben tener tantos bits como sean necesarios para representar las direcciones destino diferentes de un mismo salto indirecto, en lugar de los bits necesarios para representar todas las direcciones almacenadas en *ITT*.

Recuperando la comparación con la predicción de saltos condicionales, el bit de predicción usado en *BHT* es también un identificador local para cada salto condicional. El valor 0 codifica la dirección del bloque básico consecutivo (no saltar), mientras que el valor 1 codifica la dirección destino del salto. Este bit se convierte en una dirección completa usando *BTB*. Como para cada salto condicional sólo existen dos opciones, es eficiente preparar las dos posibles direcciones, y realizar la selección final entre las dos una vez que se dispone del bit que codifica el sentido del salto.

En el caso de los saltos indirectos, la nueva propuesta también separa en dos tablas diferentes la información sobre (1) cuál de los posibles caminos tomará el flujo de control, y sobre (2) exactamente en qué dirección comienza ese nuevo camino. El valor ***dirID*** identifica cuál de los caminos escoger. Sin embargo, como la decisión no es binaria sino múltiple (***dirID*** contiene varios bits), no es eficiente leer todas las posibles direcciones destino y esperar a obtener el valor de ***dirID*** para la selección final. En la sección 6.3 presentaremos un diseño que realiza esta operación de forma eficiente.

6.2.4 Discusión de la Viabilidad de la Propuesta

La Figura 6.4 muestra con claridad que para que la estrategia propuesta sea efectiva, la cantidad de memoria que se ahorra en *IJT* debe ser bastante mayor que la memoria requerida en *ITT*. Esto implica que el identificador ***dirID*** ha de ser pequeño y, por tanto, que se debe limitar el número total de

caminos para cada salto indirecto. También implica que el número total de direcciones destino no debe ser muy grande, o en caso contrario, para que los fallos en *ITT* no degraden la precisión, ésta deberá disponer de muchas entradas.

Mostraremos en los apartados siguientes que se puede obtener unas prestaciones similares a las de la organización original (Figura 6.3) dedicando 4 o 5 bits a ***dirID***, y con *ITT* de solamente 64 ó 128 entradas. El resultado final es que el esquema de dos niveles compensa cuando *IJT* contiene 128 entradas o más (≥ 0,5 KBytes).

La razón de que *ITT* pueda tener relativamente pocas entradas no es que aparezcan pocas direcciones destino para los saltos indirectos durante la ejecución del programa, sino que el conjunto de direcciones destino útiles es relativamente pequeño. Son útiles aquellas direcciones que son utilizadas con cierta frecuencia para realizar una predicción correcta. Hay direcciones a las que se salta pocas veces, y que no es eficiente guardar en *ITT*. Por otro lado, hay direcciones que aparecen con relativa frecuencia, pero que no suelen ser predichas correctamente, porque no están correlacionadas con los saltos disponibles en la historia global. En este caso tampoco es beneficioso guardarlas en *ITT*.

Para manejar de forma eficaz una *ITT* de pocas entradas y conseguir que las direcciones destino poco útiles no desplacen a las direcciones útiles, es necesario un diseño cuidadoso. En primer lugar, para reducir los fallos por conflicto, se necesita una alta asociatividad. Mostraremos hacerlo de forma eficiente. En segundo lugar se debe usar una política que considere la frecuencia de utilización de las direcciones destino.

La razón de que se pueda limitar a 16 ó 32 el número total de direcciones destino para cada salto indirecto es similar a la que permite que *ITT* tenga pocas entradas. Aunque hay saltos indirectos que tienen hasta 179 direcciones destino, muchas de ellas se usan muy raramente, y otras se predicen de forma correcta sólo muy ocasionalmente. Por supuesto, a

medida que *IJT* es más grande, y permite usar historias globales de mayor tamaño, algunas de estas direcciones destino se predicen correctamente con mayor asiduidad, y se hacen útiles. Los datos de nuestra experimentación muestran que, para si *IJT* tiene pocas entradas, el límite de 16 direcciones destino por salto es apropiado, mientras que si *IJT* tiene hasta 4K entradas, con 32 direcciones es suficiente.

6.3. Descripción Detallada de la Propuesta

El esquema de bloques para la predicción de saltos indirectos se muestra en la siguiente figura. Los elementos de la zona sombreada reemplazan a la tabla *IJT* del diseño de referencia descrito en el capítulo 3, y que se mostraba en la Figura 3.8, manteniendo las mismas entradas y salidas. Igual que antes, el acceso a *IJT* se activa cuando *BTB* detecta que se ha de predecir un salto indirecto polimórfico. También se combinan dos tipos de historias, una de saltos condicionales y otra de saltos indirectos. El método de indexación se comentará en breve.

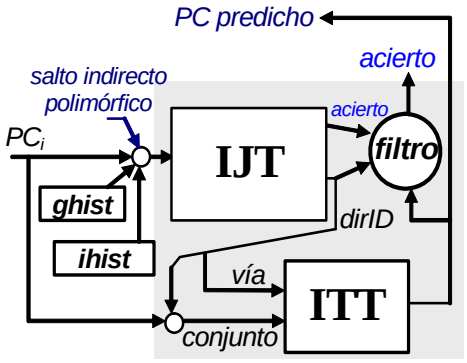


Figura 6.5. Esquema del predictor de saltos indirectos polimórficos con codificación de direcciones de salto.

Un primer acceso a la nueva tabla *IJT* proporciona, siempre que se verifique un acierto en la tabla, el identificador ***dirID*** que se usa para realizar

un segundo acceso, ahora a *ITT*, para obtener la dirección destino predicha. El método de indexación a esta segunda tabla también se comentará en breve. Tal y como se verá más adelante, la dirección destino final y el identificador ***dirID*** de esta dirección comparten información. En el diseño propuesto se valida que esta información coincida, para así establecer un mecanismo alternativo de detección de fallos.

6.3.1 Organización en Cascada con Acceso en Paralelo o en Serie

La organización en cascada para predecir los saltos indirectos polimórficos facilita la reducción de la latencia y del consumo energético del predictor. Puesto que *BTB* proporciona una primera predicción para estos saltos, el acceso a *IJT* puede estar fuera de la ruta crítica del predictor, sin tener que esperar su resultado para generar una primera predicción.

Con un esquema de acceso en paralelo a *BTB* e *IJT*, sería posible utilizar una tabla *IJT* de muchas entradas, y con una latencia mayor que la del predictor. Sin embargo, se puede reducir el consumo energético del predictor con un esquema de acceso en serie, es decir, retardando el acceso a *IJT* hasta haber verificado en *BTB* que el salto a predecir es indirecto polimórfico. Esta opción es la que se ha tomado en el diseño de referencia descrito en el capítulo 3, y en las nuevas propuestas de los dos capítulos anteriores.

6.3.2 Indexación de la Tabla *IJT*

Para codificar la historia del comportamiento de los saltos indirectos se necesita más de un bit para cada salto. En la sección 3.5.5 se describe el mecanismo utilizado para generar esta historia mezclando los bits de la dirección destino y de la dirección del salto indirecto.

Una alternativa para generar la historia de los saltos indirectos es usar los identificadores ***dirID***. El problema es que éstos pueden codificar direcciones

diferentes durante la ejecución de un mismo programa, y estas variaciones introducen ruido en el proceso de aprendizaje que reduce la precisión de las predicciones. Las variaciones en los identificadores son mayores en los programas que generan un mayor número de direcciones destino y que son los que más necesitan usar la correlación con la historia para proporcionar una alta precisión. Por esta razón se ha descartado esta alternativa.

En la sección 3.5.5 también se describe como se construye el índice a *IJT* utilizando la dirección del bloque básico y las dos historias globales, de saltos condicionales (*hist*) e indirectos (*ihist*). *IJT* es de acceso directo y contiene una etiqueta de 16 bits para verificar que el contenido de la tabla corresponde a la combinación de entrada deseada. Más adelante se presentarán resultados con tamaños más pequeños para la etiqueta.

6.3.3 Generación de Identificadores Locales

La codificación de una dirección destino de 28 bits en un identificador de 4-6 bits se realiza en la fase de actualización del predictor, cada vez que aparece una dirección destino nueva que se quiere insertar en *ITT*. En principio existe total libertad para realizar esta codificación. En nuestra propuesta tratamos de aprovechar esta flexibilidad para conseguir dos objetivos:

- simplificar el mecanismo de indexación a *ITT* que se usa durante la fase de predicción, de forma que sea rápido, con un consumo energético reducido, y con un uso eficiente de las entradas en la tabla.
- simplificar el mecanismo para asignar identificadores distintos a cada dirección destino de un salto indirecto.

Tal como se ha argumentado previamente, *ITT* ha de tener relativamente pocas entradas, y para hacer un uso eficiente de ellas es necesario que tenga una alta asociatividad. Sin embargo, una búsqueda asociativa en la fase de predicción es lenta y consume más energía. La solución propuesta es codificar dentro del propio identificador ***dirID*** la vía en la que se encuentra la

dirección destino. Supondremos que la identificación de la vía se codifica en los bits bajos de ***dirID***. Esta idea es bastante similar a la de la predicción de vía propuesta para acelerar el predictor en el capítulo 4.

Para generar los bits altos de ***dirID***, la solución más efectiva es utilizar la propia dirección destino. De nuevo se ha optado por utilizar la función ***xor-fold-n*** (Figura 3.5.a) para convertir una dirección de muchos bits en un fragmento de pocos bits. Veremos más adelante que esta solución permite detectar dinámicamente, en la fase de predicción, ciertos casos en que el identificador ***dirID*** ha dejado de apuntar correctamente a la dirección destino.

6.3.4 Indexación de la Tabla *ITT*

La figura siguiente muestra la función para generar el índice con el cual acceder a *ITT*. El índice se compone de los bits que determinan el conjunto al que acceder y de los bits que determinan la vía donde acceder. Tal como se ha comentado en el apartado anterior, la generación de la vía se hace de forma directa utilizando los bits bajos de ***dirID*** (***dirID***$_{bajo}$).

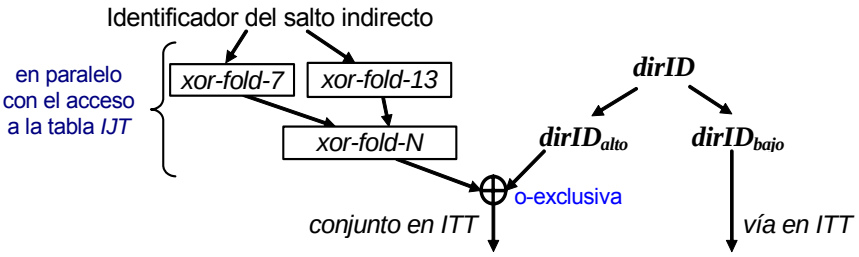


Figura 6.6. Generación del índice a *ITT* en la fase de predicción.

La generación del conjunto es más complicada, aunque cabe observar que la parte de cómputo más compleja, la que convierte la dirección del salto indirecto en un valor de pocos bits (en la parte izquierda de la figura), se hace en paralelo al acceso a *IJT*, y por tanto no debe ralentizar la operación. El único retardo para generar el índice consiste en la combinación mediante la

operación *o-exclusiva* de los bits altos de ***dirID*** (***dirID***$_{alto}$) con los bits que identifican el salto indirecto. Esta combinación es necesaria debido a usar identificadores locales.

Cabe señalar de nuevo, que toda la complejidad de la organización asociativa se desplaza de la fase de predicción a la fase de actualización, más concretamente a las ocasiones en que se debe asignar una entrada de *ITT* a una nueva dirección. En este momento puede ser necesaria una búsqueda asociativa en *ITT* para descubrir si la dirección destino realmente no reside en la tabla y para buscar una entrada donde emplazarla. Una vez asignada la entrada, se utiliza el identificador de la vía para determinar ***dirID***$_{bajo}$, y así en la fase de predicción realizar los accesos de forma directa.

6.3.5 Fase de Predicción. Detección de Fallos y Filtro

La Figura 6.7 muestra los pasos llevados a cabo en la fase de predicción para los saltos indirectos. La función ***f*** genera el índice a *IJT* (sección 6.3.2), pero el acceso no comienza hasta que se ha validado por *BTB* que el salto a predecir es indirecto y polimórfico (acceso serie). Una vez obtenido ***dirID*** de *IJT* se genera el índice a *ITT* (función f_2) de la manera mostrada por la Figura 6.6. El resultado final del segundo acceso es la dirección destino predicha.

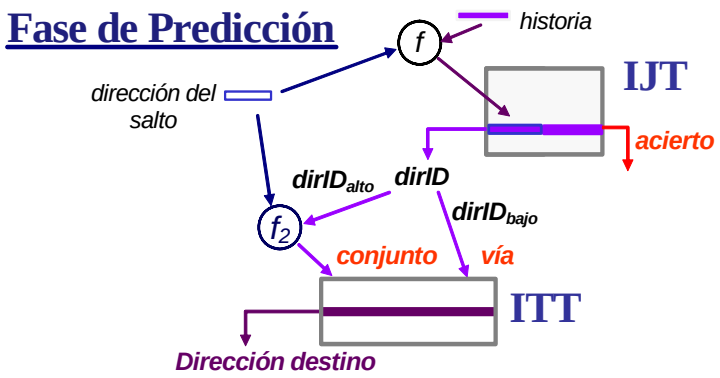


Figura 6.7. Fase de predicción para saltos indirectos polimórficos, usando una organización con codificación local de las direcciones de salto.

Una cuestión clave en el acceso al predictor es validar que la dirección destino corresponde realmente al salto indirecto y a la historia previa. Es decir, validar si el acceso es lo que se conoce generalmente como un acierto o es un fallo. Detectar el fallo en el predictor específico para saltos indirectos permite utilizar la dirección destino base que proporciona *BTB*, que tiene más probabilidades de ser correcta que la dirección obtenida por un fallo.

Los casos que pueden provocar que la dirección destino obtenida en la fase de predicción sea incorrecta son los siguientes:

1. coincidencia (error de *alias*) en el índice a *IJT* y en la etiqueta de *IJT* para valores de entrada diferentes.
2. coincidencia al generar el índice a *ITT* a partir de dos direcciones de salto diferentes con sus identificadores ***dirID.***
3. un reemplazamiento anterior en *ITT* provoca que el valor ***dirID*** ya no apunte a la dirección destino correcta.

El problema del punto 1 se puede evitar utilizando etiquetas de gran tamaño en *IJT*. Sin embargo, dedicar mucho espacio de la tabla a etiquetas también reduce el número de entradas y por tanto la precisión. En la sección de resultados presentaremos un análisis experimental para encontrar el tamaño adecuado de estas etiquetas.

El problema de los puntos 2 y 3 se puede reducir utilizando etiquetas en *ITT*. No obstante, se puede usar el propio contenido de *ITT* para comprobar si la dirección obtenida es probablemente incorrecta. Puesto que $\boldsymbol{dirID_{alto}}$ se forma utilizando la propia dirección destino (supongamos que con la función $\boldsymbol{f_3}$), se puede volver a aplicar esta función a la dirección obtenida de *ITT* para comprobar que el resultado coincide con $\boldsymbol{dirID_{alto}}$. Este mecanismo de control se muestra en la siguiente figura.

La forma más directa de reducir la ocurrencia de los casos en el punto 3 es reducir el número de reemplazamientos en *ITT.* Mostraremos una política de prioritización basada en el uso del contador de histéresis que reduce de

forma efectiva estos reemplazamientos, y es adecuada para disminuir el número de fallos en *ITT*, y aumentar así la precisión de las predicciones.

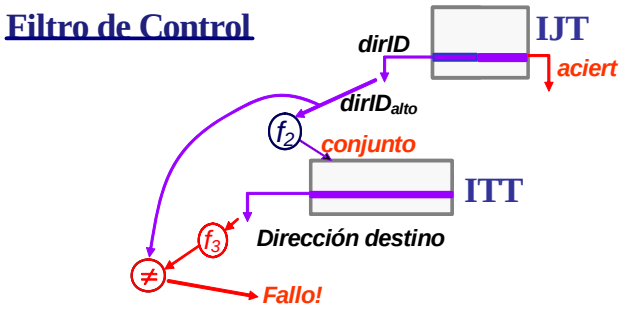


Figura 6.8. Filtro de control con codificación local de direcciones de salto.

6.3.6 Fase de Recuperación

Respecto al diseño original, los cambios propuestos en el predictor de saltos indirectos no implican cambios en la fase de recuperación de fallos.

6.3.7 Fase de Actualización

La siguiente figura muestra el organigrama detallado de la fase de actualización, que se irá comentando paso por paso. Las predicciones correctas realizadas por *BTB*, cerca de la mitad, no requieren ninguna actualización. Las predicciones correctas realizadas por *IJT*+*ITT* incrementan el contador de histéresis asociado a la entrada correspondiente de *ITT*. Este contador implementa la "resistencia" de esta dirección destino a ser reemplazada. La intención de esta política es priorizar la ocupación de *ITT*, relativamente pequeña, por direcciones destino útiles.

Si la predicción fue incorrecta, se debe comprobar que el bit de histéresis de la entrada correspondiente en *IJT* esté a cero. Si no es así, se pone el bit a cero y se finaliza la fase de actualización. Si el bit de histéresis sí que está

a cero, entonces se permite alterar la predicción de *IJT*. El razonamiento de esta política es que si la predicción había sido correcta la última vez, un único fallo no debe ser suficiente para modificar la predicción.

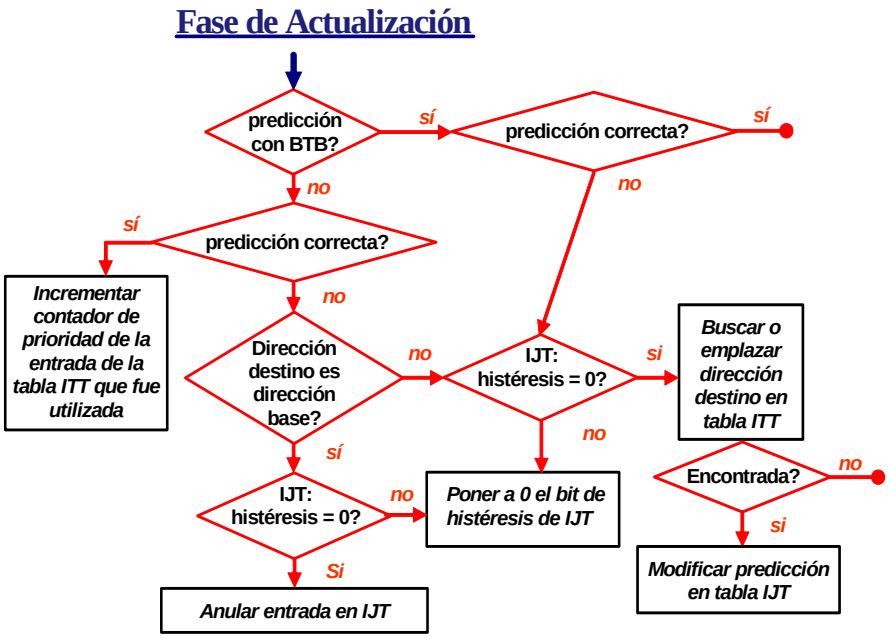


Figura 6.9. Fase de actualización con codificación local de direcciones de salto.

Si la dirección destino que no se predijo correctamente es la dirección base de *BTB*, entonces se anula la entrada correspondiente a *IJT* y se deja el bit de histéresis a 0. Así, la próxima vez que se encuentre la misma historia global, no se encontrará ninguna predicción en *IJT* y se usará la predicción de *BTB*. De este modo, se libera la entrada en *IJT* para su uso por parte de otro caso de predicción.

Si la dirección destino del salto indirecto no fue la dirección base, entonces se debe buscar en *ITT*, y si no se encuentra, buscar una entrada donde emplazarla. La siguiente figura muestra cómo realizar esta operación.

En primer lugar se debe buscar la dirección destino en *ITT*. El conjunto donde puede encontrarse queda determinado por la propia dirección destino

y por la dirección del salto indirecto. Se debe buscar en todas las vías del conjunto. Puesto que la latencia de la fase de actualización no es crítica, se puede realizar la búsqueda en las vías en forma serie, para que así los accesos a *ITT* se hagan de la misma forma en la fase de actualización que en la fase de predicción.

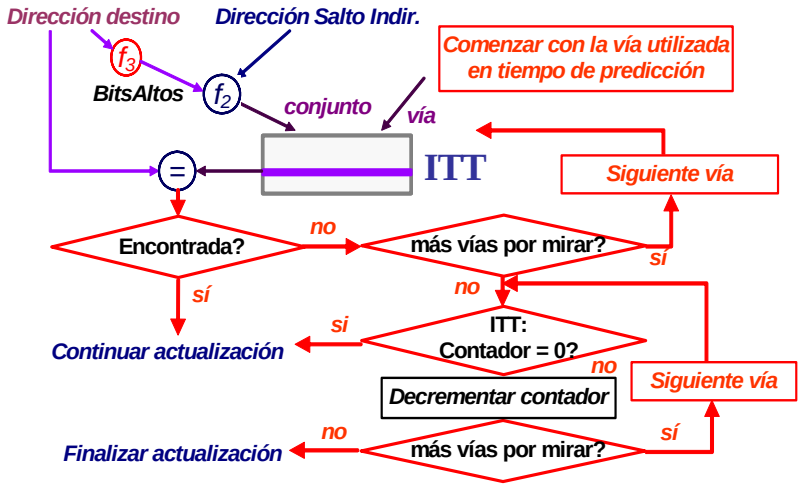


Figura 6.10. Búsqueda y emplazamiento de direcciones de salto en *ITT*. La operación se hace en serie, buscando vía a vía.

Si la dirección destino no se encuentra en la tabla, se debe buscar una entrada donde emplazarla. La propuesta consiste en ir decrementando todos los contadores de las entradas del conjunto hasta encontrar una de ellas con el contador a cero. Si tras decrementar una vez todos los contadores, no se ha encontrado ninguna entrada con el contador a cero, entonces se descarta la dirección destino y se finaliza la fase de actualización. El razonamiento de esta política es que sólo se permite entrar una nueva dirección destino en *ITT* si el número de veces que ésta se intenta almacenar es superior al número de veces que se usa en predicciones correctas alguna de las direcciones que ya está en la tabla.

Si el método anterior proporciona una vía donde almacenar la dirección destino, entonces se unen los bits que identifican la vía y los bits que

identifican el conjunto para formar ***dirID***. Entonces se modifica la entrada que corresponde en *IJT* para que contenga el nuevo valor de ***dirID***.

6.4. Resultados

En primer lugar se analizarán los programas de evaluación para determinar el porcentaje de utilización del predictor de saltos indirectos. Luego se estudiará por separado tanto el espacio de diseño de *IJT* como el de *ITT*. Todos estos resultados se combinarán con el cómputo de los requerimientos de memoria de cada alternativa, para relacionar la precisión con el tamaño del predictor. Estos resultados servirán para encontrar la configuración más adecuada de la nueva propuesta y para compararla con la propuesta original de referencia.

Los anteriores resultados se han obtenido con el modelo de procesador simplificado, que utiliza un modelo ideal de la memoria y que no considera las dependencias entre las instrucciones ni sus latencias. El número de ciclos entre la predicción y la actualización de las tablas se ha fijado en 16 ciclos. El predictor sí que se ha modelado en detalle.

A continuación se mostrarán resultados de simulaciones ciclo a ciclo del modelo completo del procesador. Con estos datos se corroborará que el efecto de actualizar las tablas del predictor con varios ciclos de retardo es insignificante, y que el incremento de un ciclo en la latencia de la predicción de los saltos indirectos sólo reduce las prestaciones ligeramente. También se probará que el incremento de precisión de las predicciones de saltos indirectos aumenta en bastantes casos las prestaciones del procesador.

6.4.1 Análisis preliminar de los programas de evaluación

Dentro del conjunto de programas de evaluación SPEC, muchos de ellos no contienen apenas saltos indirectos polimórficos. Una de las posibles razones es que ninguno de los programas SPEC está escrito con un lenguaje

orientado a objeto (C+, Java) que, tal como se comentó en el capítulo 2, son una de las fuentes de saltos indirectos polimórficos. Para evaluar la propuesta de este capítulo se han seleccionado aquellos programas de evaluación SPEC en los que la predicción de saltos indirectos puede constituir un problema.

Mientras que algún trabajo sobre el tema de la predicción de saltos indirectos también ha utilizado una selección de programas SPEC, [ChHP97], otros trabajos han incluido más programas, generalmente escritos en C+, en los que la cantidad de saltos indirectos polimórficos es más importante, [DrHo98b], [KaKa98] [CaGr94b]. Es evidente que los resultados promedio finales dependen mucho de esta selección. Una selección de este tipo, sesgada, proporciona una cota superior para las prestaciones.

Todos estos autores coinciden en reconocer que el problema de los saltos indirectos afecta sólo a algunas aplicaciones. Aún así, esto no es óbice para que dos diseños comerciales actuales, el Intel Pentium IV de 90nm [BBH+04] y el Intel Pentium M [GRB+03], incluyan un mecanismo específico para predecir con mayor precisión los saltos indirectos. De hecho, el mecanismo propuesto es similar a la propuesta de referencia usada en este trabajo.

Nuestra propuesta de diseño es conservadora, y sólo utiliza el mecanismo específico para los saltos indirecto polimórficos cuando hay una alta probabilidad de que su uso sea útil, y prioriza en todo momento la predicción de otros tipos de saltos. Por tanto, no existe penalización para los programas que no se benefician de mejoras en la predicción de saltos indirectos, y no se falsean los resultados al no incluir estos programas en el análisis.

La Tabla 6-1 presenta los programas seleccionados. Las columnas 3 y 4 muestran el número total de saltos indirectos polimórficos y el total de direcciones destino para todos estos saltos. Las últimas tres columnas muestran la proporción por instrucción retirada de: (1) saltos indirectos, (2) saltos indirectos polimórficos, y (3) saltos indirectos que usan la predicción de una tabla *IJT* de 2K entradas.

Tabla 6-1. Programas de evaluación e información estática y dinámica sobre los saltos indirectos polimórficos

programas	SPEC	Información estática		Información dinámica		
		polimórf.	dirección	Indir.	polimórf.	uso IJT
crafty	INT00	15	89	1.29	0,291	0,175
eon	INT00	13	47	2,70	0,375	0,244
gap	INT00	49	146	3,22	1,19	0,526
gcc	INT00	183	975	0,73	0,298	0,124
perlbmk	INT00	65	659	1,70	1,59	0,794
m88ksim	INT95	7	26	1,12	0,332	0,076
li	INT95	7	71	1,49	0,730	0,314
sixtrack	FP00	8	49	0,43	0,187	0,140
		total		***Porcentaje de instrucciones***		

Se puede observar en la tabla que generalmente el número total de saltos indirectos polimórficos de un programa es pequeño, y que de media saltan a más de 5 direcciones destino. Un dato no reflejado en la tabla es que, en general, el número de saltos indirectos polimórficos que provoca la mayoría de los fallos es aún más reducido (entre 1 y 6). Algunos de estos saltos indirectos pueden llegar a tener entre 40 y 179 direcciones destino diferentes.

Por otro lado, en los programas seleccionados los saltos indirectos polimórficos suponen menos del 1,6% del total de instrucciones ejecutadas. En menos del 50% de estos casos se utiliza la predicción proporcionada por *IJT*, bien sea porque la predicción la proporciona *BTB*, o bien porque la predicción es errónea.

6.4.2 Espacio de Diseño de *IJT*

La Figura 6.11 muestra la precisión del predictor de referencia, descrito en el capítulo 3, promediada para el conjunto de programas de evaluación. Se analiza el efecto del número de entradas de *IJT* en la precisión, y del tipo de historia global utilizada. En la tabla se reflejan los resultados considerando la longitud óptima de la historia, es decir, aquella que proporciona la mayor precisión. Cuando se combinan las historias *hist* e *ihist*, se utilizan siempre dos vectores de bits de la misma longitud. Pruebas de simulación indican que

no se consiguen mejoras apreciables variando la relación entre ambas longitudes. La longitud óptima para la combinación de ambas historias se muestra en la figura.

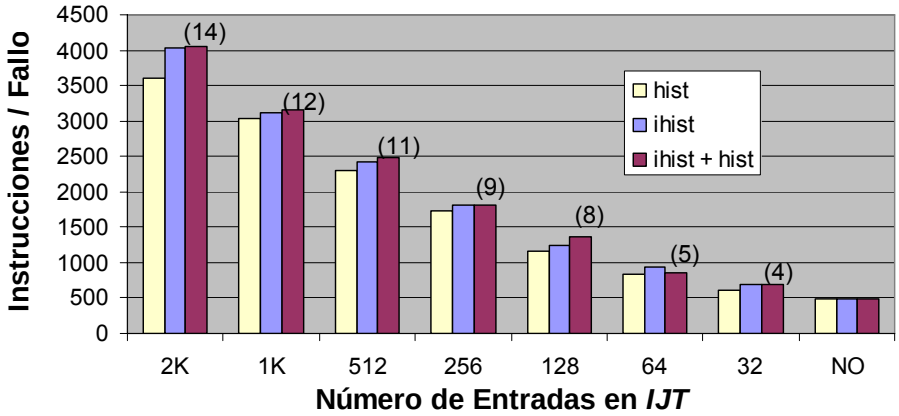


Figura 6.11. Precisión de la predicción de saltos indirectos en función del número de entradas de *IJT* y del tipo de historia global utilizada para indexar la tabla.

Se puede apreciar que la ganancia en precisión aportada por *IJT* es importante, incluso para tablas con un muy pequeño número de entradas (32 ó 64). También se demuestra que el uso de historia basada en los saltos indirectos previos proporciona mejor precisión que usar simplemente la historia de los saltos condicionales. Para la mayoría de los tamaños de *IJT*, la mejor opción consiste en combinar ambas historias.

La siguiente figura desglosa la precisión del predictor para cada uno de los programas de evaluación. En este caso, *IJT* se indexa con la combinación de *hist* e *ihist*, para los tamaños mostrados en la Figura 6.11. Para casi todos los programas seleccionados, la precisión aumenta de forma importante al utilizar *IJT*. El programa ***crafty*** es el único que no se beneficia demasiado del uso de la correlación con la historia previa.

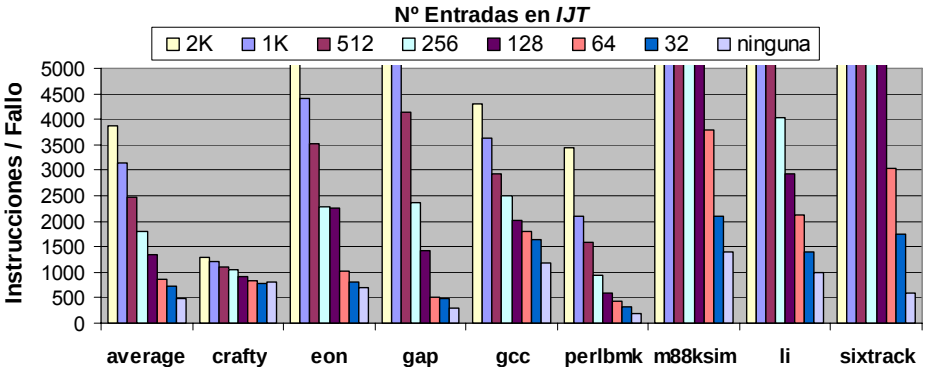


Figura 6.12. Precisión de la predicción de saltos indirectos en función del número de entradas de *IJT*. La tabla se indexa combinando *hist* e *ihist*.

6.4.3 Espacio de Diseño de *ITT*

Los parámetros fundamentales en el diseño de *ITT* son el número de entradas, el grado de asociatividad y, en menor grado, el tamaño de los contadores de histéresis. Se han tomado medidas del número de fallos en *ITT* variando estos parámetros.

Los resultados han indicado que una organización asociativa de 8 vías proporciona razones de fallo muy similares a las de una organización completamente asociativa. Se ha probado también que el uso del contador de histéresis reduce la razón total de fallos para tablas pequeñas, y que con un tamaño de 4 bits se obtiene el mismo resultado que con tamaños mayores. Ya que tienen una influencia muy pequeña en el tamaño total del predictor, se han usado contadores de 4 bits en todas las configuraciones posteriores.

La siguiente figura muestra la razón de fallos al variar el número de entradas de *ITT* (con 8 vías), y para diferente número de entradas en *IJT*. Los resultados indican que una tabla de 64 entradas tiene una razón de fallos entre el 8% y el 15%. También se puede apreciar que cuanto menor es el tamaño de *IJT* (eje horizontal) mayor es la razón de fallos en *ITT*. Puesto que una tabla *IJT* mayor produce menos fallos de predicción, se intenta actualizar menos veces *ITT*, y se producen menos fallos.

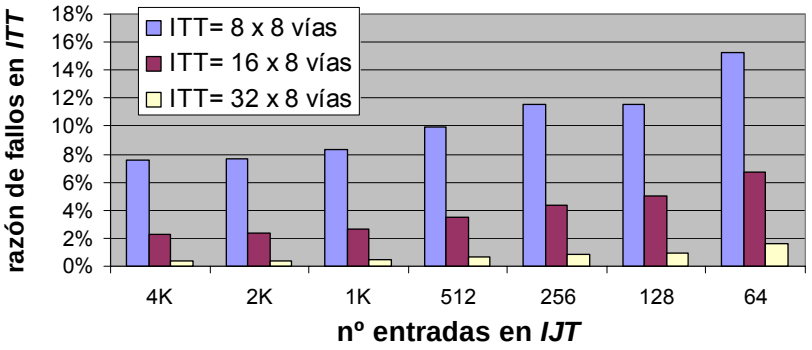


Figura 6.13. Razón de fallos en una tabla *ITT* (8 vías, contador de 4 bits) en función de su número de entradas (64, 128 y 256) y del tamaño de la tabla *IJT*.

El hecho de que la dirección destino esté en *ITT* no implica un acierto en la predicción, pues para ello es necesario que *IJT* genere la predicción correcta. La métrica de la razón de fallos sirve para explorar los valores óptimos de asociatividad y tamaño del contador, pero no hay un criterio claro que determine el efecto de la razón de fallos en la precisión del predictor. Así que es necesario explorar el espacio de diseño considerando el predictor completo, y utilizando como métricas la precisión y el tamaño del predictor.

6.4.4 Relación entre Precisión y Tamaño del Predictor

Para poder comparar de forma equitativa las dos organizaciones del predictor es necesario estimar la memoria que requieren. Puesto que todas las tablas son de acceso directo y la lógica adicional a las tablas es simple, es razonable considerar que el tamaño en bits del conjunto de tablas del predictor determina de forma bastante precisa el área de chip que ocupa.

Supondremos que las direcciones destino ocupan 28 bits, bien sea en *IJT* o en *ITT*. Para *IJT* hay que considerar, dependiendo de si la configuración es de uno o dos niveles, o bien una dirección o bien un identificador ***dirID*** (de entre 6 y 4 bits), y además una etiqueta y un bit de histéresis. Para *ITT* se debe incluir, además de la dirección de 28 bits, el contador de 4 bits.

La Figura 6.14 muestra la relación entre precisión y tamaño del predictor, tanto para el diseño base (*IJT*-base) como para tres configuraciones diferentes del diseño con direcciones codificadas (*IJT*+*ITT*). Cada una de estas tres configuraciones usa una *ITT* con un número diferente de entradas (256, 128, ó 64). Los otros dos parámetros (tamaño de ***dirID*** y longitud de etiqueta), se han obtenido tras una exploración del espacio de diseño.

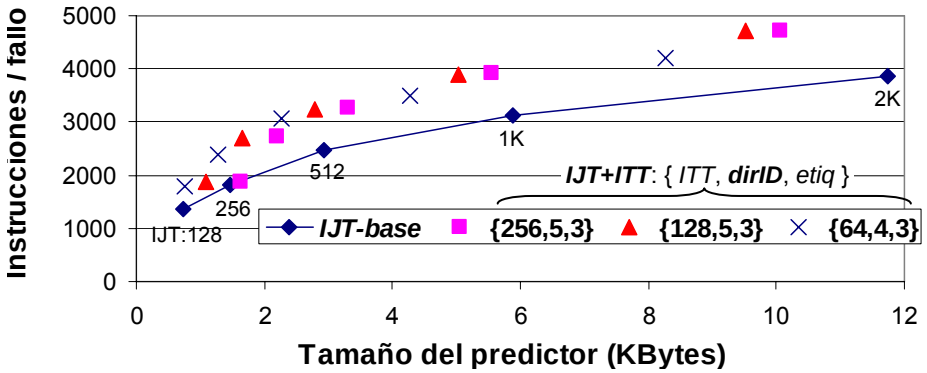


Figura 6.14. Relación entre precisión y tamaño del predictor para el esquema base, y para el esquema con codificación de direcciones. La notación {*ITT*, *dirID*, *etiq*} indica el número de entradas en *ITT*, el tamaño de ***dirID***, y la longitud de la etiqueta en *IJT*.

Los valores correspondientes al diseño base (*IJT*-base) están señalados en la figura con una línea que los une a todos entre sí, y para cada punto de la gráfica se indica el número de entradas en *IJT*. Se puede observar que los valores del diseño propuesto en este capitulo están todos por encima de esta línea. Esto indica que se consigue una mejor relación entre precisión y tamaño del predictor. Para un tamaño fijado del predictor, el incremento en la precisión está entre el 20% y el 40%.

Es importante remarcar que para un mismo número de entradas en *IJT*, el esquema de dos niveles (*IJT*+*ITT*) obtiene menor precisión que el esquema de un nivel (*IJT-base*), desde un 15% menos para 4K y 2K entradas, hasta casi un 40% menos para 128 entradas. Esta pérdida de precisión se debe a usar una tabla *ITT* relativamente pequeña. Sin embargo, esta reducción de

precisión queda compensada de sobras por la reducción en requerimientos de memoria, y convierte a las configuraciones de dos niveles con tablas *ITT* pequeñas en las opciones más eficientes. Se puede observar en la gráfica anterior que para tamaños pequeños de predictor (< 3KBytes) es preferible usar una tabla *ITT* de 64 entradas. Para predictores con menos de 128 entradas en *IJT*, la propuesta de dos niveles deja de ser eficiente, pues usar menos de 64 entradas en *ITT* reduce excesivamente la precisión. Para predictores de tamaño mayor de 3KBytes, una tabla *ITT* con 128 entradas produce los mejores resultados.

En cuanto al tamaño del identificador ***dirID***, el uso de 5 bits sólo compensa cuando *ITT* tiene al menos 128 entradas. Si tiene sólo 64 entradas, entonces la contención de las direcciones en *ITT* limita el número de direcciones destino por salto indirecto, y hace que usar sólo 4 bits para ***dirID*** se convierta en la opción más eficiente.

En el apartado 6.2.4 ya se explicó que, a pesar de que el número total de direcciones destino para algún programa es cercano a 1000, no es necesario poder representar todas las direcciones destino, pues muchas o bien son muy infrecuentes, o bien se predicen correctamente pocas veces.

Finalmente, se ha determinado que la longitud de 3 bits es la más apropiada para la etiqueta de *IJT* con un esquema de dos niveles. Aunque reducir la longitud de la etiqueta aumenta el número de colisiones no detectadas y disminuye la precisión del predictor, una etiqueta de 3 bits en lugar de 16 bits, reduce el tamaño de *IJT* aproximadamente un 60%. Además, el mecanismo de filtro descrito en el apartado 6.3.5 consigue detectar una parte importante (entre el 20% y el 70%) de los fallos no detectados por la etiqueta pequeña.

La siguiente figura muestra el efecto que tiene reducir la longitud de la etiqueta en el predictor base (con direcciones completas), tanto en su precisión como en su tamaño. Se ha vuelto a buscar la longitud apropiada de la historia para cada tamaño de tabla, para así adaptarlo a las nuevas

condiciones de detección de colisiones. Se puede ver que los resultados con longitudes de etiqueta de 9 bits o menos caen por debajo de la línea marcada para el predictor base original. Esto indica que en este caso la reducción en precisión por reducir la longitud de la etiqueta no queda compensada por la reducción, relativamente más pequeña, en el tamaño del predictor.

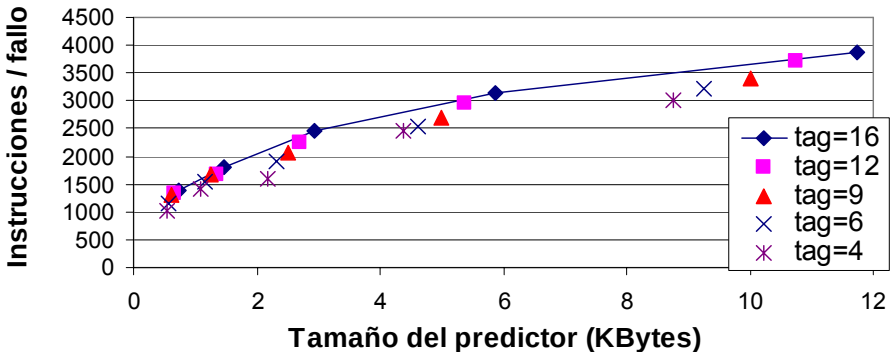


Figura 6.15. Relación entre precisión y tamaño del predictor para *IJT* de entre 256 y 2K entradas, en función de la longitud de la etiqueta.

Para finalizar esta sección, en las gráficas siguientes se mostrarán resultados para algunos programas concretos de evaluación. La primera gráfica muestra resultados para el programa ***gap*** (los programas ***eon***, ***li,*** ***m88ksim*** y **sixtrack** tiene un comportamiento muy similar). En este caso, los saltos indirectos tienen una gran correlación con la historia anterior, de modo que aumentar el tamaño de la tabla proporciona un gran incremento en la precisión. Gracias a esto, la propuesta de codificar las direcciones logra incrementar la precisión de forma espectacular para tamaños del predictor superiores a 3 KBytes. En algunos de los programas, con un predictor de alrededor de 1KBytes se reduce el número de fallos de predicción de saltos indirectos a un valor casi insignificante.

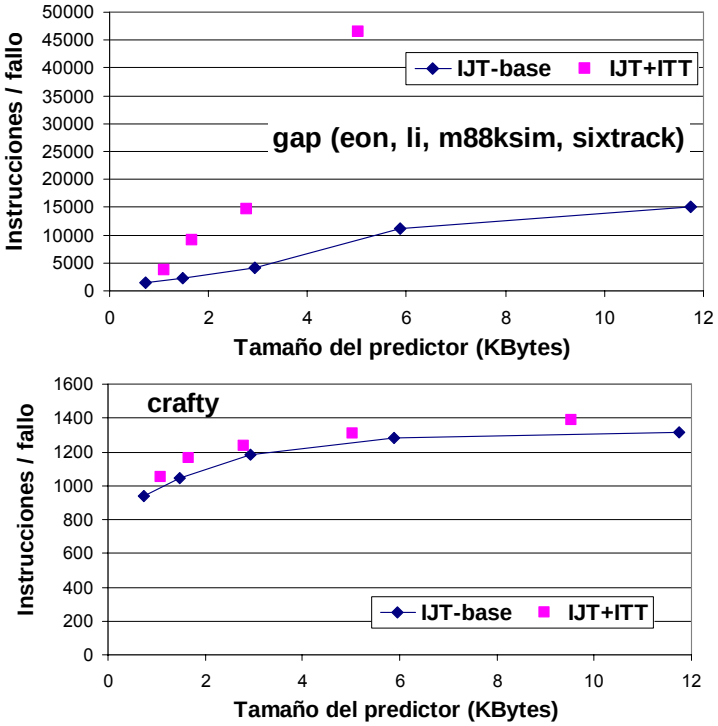


Figura 6.16. Relación entre precisión y tamaño del predictor para el esquema base, y para {*ITT=16 conjuntos x 8 vías*, *dirID: 5 bits*, *etiq IJT: 3 bits*}.

En el otro rango del espectro se encuentra la aplicación ***crafty***, mostrada en la segunda gráfica de la figura anterior, y que incrementa la precisión muy ligeramente al incrementar el tamaño del predictor más allá de 3KBytes. En este caso, la ventaja de codificar direcciones es sólo la de reducir el tamaño del predictor.

La siguiente figura agrupa dos gráficas más, las de los programas ***perlbmk*** y ***gcc***. A pesar de que contienen un gran número de saltos indirectos polimórficos y de direcciones destino, se puede comprobar que en el rango de tamaños entre 1KBytes y 10KBytes la propuesta de dos niveles consigue aumentar la relación entre precisión y tamaño del predictor alrededor del 50%.

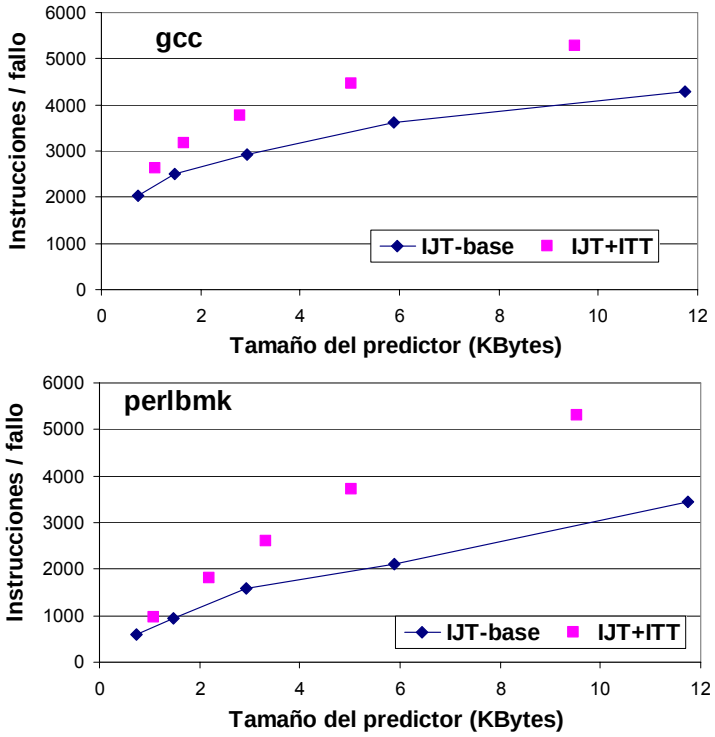


Figura 6.17. Relación entre precisión y tamaño del predictor para el esquema base, y para {*ITT=16 conjuntos x 8 vías*, *dirID: 5 bits*, *etiq IJT: 3 bits*}.

6.4.5 Modelo de la Microarquitectura del Procesador

Para obtener medidas del rendimiento del procesador se han realizado simulaciones ciclo a ciclo con un modelo completo del procesador. La Tabla 6-2 muestra las características básicas del procesador simulado. Los datos no mostrados se pueden encontrar en la sección 3.3.

Tabla 6-2. Características más importantes de la microarquitectura del procesador

Unidad de Búsqueda	Núcleo de Ejecución	Sistema de Memoria
Desacoplada: tamaño *FTQ* = 16 **Predictor**: 1 bloque básico por ciclo *Gshare* (256K entradas), **Penalización fallo**: 16 ciclos mínimo **Prebúsqueda** en iCache.	**Decodificación**: 6 instr./ciclo **Ejecución**: 6 instr./ciclo **Reorder Buffer**: 240 entradas **Cola Instrucc.**: 120 entradas **Cola de Load's**: 120 entradas **Cola de Store's**: 90 entradas	**iL1-Cache:** 8KB, 2 ciclos **dL1-Cache:** 16KB, 2 ciclos **L2-Cache**: 512KB, + 6 ciclos **L3-Cache**: 16 MB, + 32 ciclos **Memoria**: + 120 ciclos

6.4.6 Efecto de la Actualización Retardada del Predictor

El primer experimento realizado ha consistido en verificar que la precisión del predictor de saltos indirectos no varía significativamente al considerar la ejecución real. En el modelo del procesador completo el retardo producido en la actualización es variable, desde un mínimo de 16 ciclos hasta un máximo que depende de las dependencias con las instrucciones anteriores y de los retardos que afectan a estas instrucciones, los más importantes debidos a fallos en la caché de datos.

Los resultados indican variaciones en la precisión inferiores al 1% en todos los casos. Como muchos autores han argumentado al realizar experimentos similares, el mayor retardo en la actualización del predictor incrementa la histéresis del predictor, es decir, la inercia a no cambiar de estado demasiado rápido. El efecto no es siempre negativo, y en algunos programas la precisión incluso aumenta ligeramente.

6.4.7 Efecto de la latencia del Predictor

Un problema de la organización de dos niveles propuesta en este capítulo es incrementar la latencia de la predicción de saltos indirectos. Se reemplaza un único acceso a *IJT* por dos accesos, uno a los identificadores codificados (*IJT*) y otro a las direcciones (*ITT*). El diseño cuidadoso mostrado en la sección anterior permite que el acceso a *ITT* sea rápido a pesar de tener una alta asociatividad. Además, la nueva tabla *IJT* es más pequeña.

Se ha medido el efecto que tiene en el rendimiento incrementar la latencia de la predicción de saltos indirectos. La Figura 6.18 muestra la reducción en el rendimiento del procesador, para diferentes tamaños de *IJT*, al variar la penalización en ciclos debida a usar *IJT* en lugar de *BTB*.

Penalizar con un solo ciclo las predicciones que usan *IJT* tiene un efecto prácticamente insignificante en el rendimiento. La reducción al aumentar la penalización de 1 a 2 ciclos es también muy pequeña, alrededor del 0,02%.

La razón de la pequeña reducción de rendimiento es que, por un lado, el uso del predictor no es muy frecuente, y que, por otro lado, un gran porcentaje del retardo adicional es ocultado por el esquema desacoplado del predictor. Para ello, debe ocurrir que los saltos indirectos estén relativamente alejados de los saltos que provocan fallos de predicción, y de este modo el retardo adicional pueda ser "absorbido" con las predicciones anteriores al retardo, que aún no han sido consumidas por la etapa de búsqueda de instrucciones.

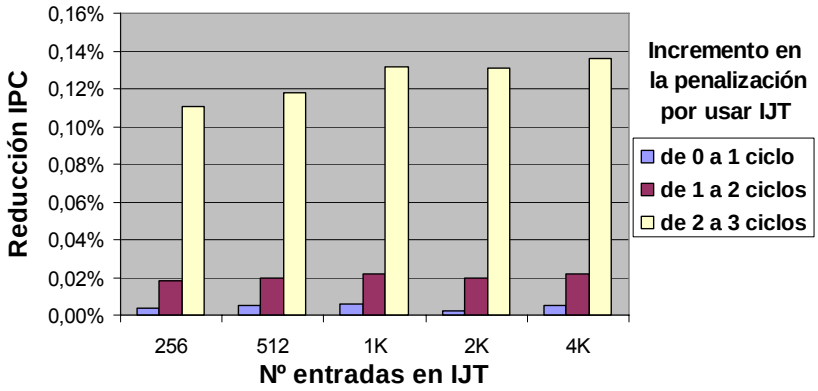


Figura 6.18. Reducción del IPC para diferentes latencias de la predicción de saltos indirectos y diferente número de entradas en *IJT*

De todos modos, tal como predice la ley de Amdahl, al ir incrementando los ciclos de penalización, los problemas poco frecuentes adquieren mayor influencia (al pasar de 2 a 3 ciclos de penalización, el IPC se degrada entre un 0,11% y un 0,13%). Si realmente la latencia de la predicción de saltos indirectos se convierte en un problema, entonces se puede iniciar el acceso a *IJT* en paralelo con el acceso a *BTB*, a pesar de que ello suponga un incremento en el consumo energético.

6.4.8 Mejora del Rendimiento

Por último, se quiere estimar el efecto en el rendimiento de la predicción de saltos indirectos, y de las propuestas de este capítulo. La Figura 6.19

muestra cómo disminuye el IPC a medida que la predicción de saltos indirectos pasa de ser perfecta a ser real, con predictores de tamaño cada vez menor, hasta llegar al esquema en que no se utiliza *IJT*, y sólo se utiliza *BTB*. Se muestra el resultado para la media geométrica de todos los programas, y también para los programas más representativos. El predictor de saltos indirectos que se ha utilizado en este caso es el predictor base de un solo nivel (*IJT-base*).

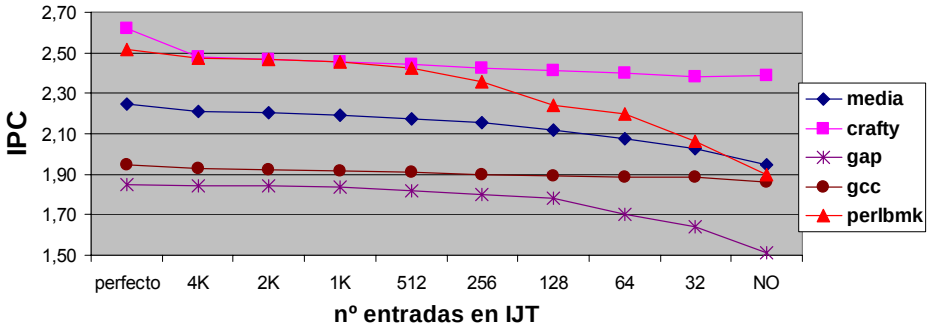


Figura 6.19. IPC obtenido con el predictor *IJT-base*, con diferente número de entradas en *IJT*.

Para la media de los programas seleccionados, un predictor perfecto de saltos indirectos comparado con usar únicamente *BTB*, mejoraría el rendimiento del procesador en un 15,6%. Un predictor *IJT-base* de 4K entradas (22 KBytes) mejora el IPC un 13,6%. Por tanto, el predictor realista obtiene un 85% de la máxima mejora posible. Usar un predictor *IJT-base* de 512 entradas (3 KBytes) mejora el IPC un 11,9%, e incluso un predictor de 64 entradas (0,4 KBytes) llega a mejorar el IPC un 6,7%.

Los dos programas que obtienen mayor beneficio del predictor de saltos indirectos son ***perlbmk*** y ***gap***, aunque el primero requiere de un predictor de mayor tamaño para acercarse al máximo rendimiento.

El predictor de dos niveles (*IJT*+*ITT*) consigue proporcionar una media de entre un 20% y un 40% mejor precisión para la misma cantidad de memoria. La Figura 6.20 muestra cómo se transforma esta mejora de precisión en

mejora de rendimiento. La mayor parte del incremento de rendimiento se logra con tablas relativamente pequeñas. Por esta razón, la mayor mejora de rendimiento proporcionada por la propuesta de este capítulo, de alrededor de un 1,3%, se da para predictores de entre 0,5 KBytes y 1 KBytes.

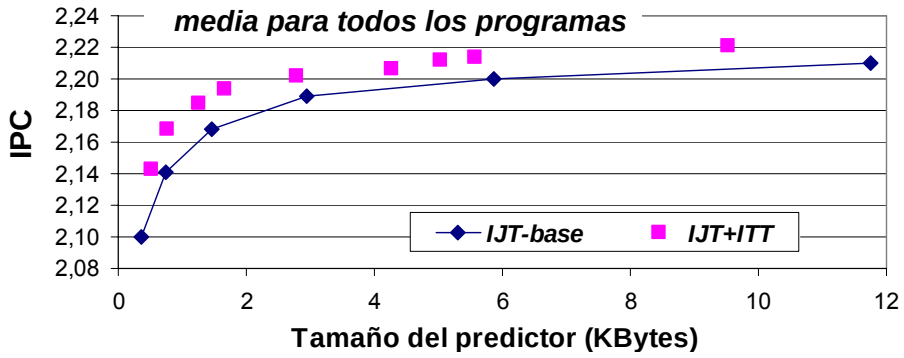


Figura 6.20. IPC para los predictores de saltos indirectos de referencia (*IJT-base*) y de dos niveles (*IJT+ITT*), para diferentes tamaños de memoria.

Ya se ha comentado que el programa ***perlbmk*** es el que más se beneficia del predictor de saltos indirectos. En la siguiente figura se muestra el rendimiento que se obtiene para este programa con los dos predictores de saltos indirectos analizados. Para un tamaño del predictor de entre 1 KByte y 3 KBytes la mejora de rendimiento alcanza el 3,1%.

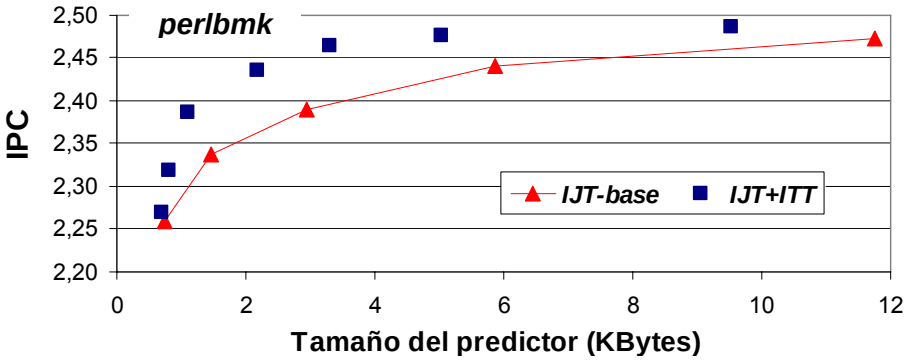


Figura 6.21. IPC para el programa ***perlbmk*** de los predictores de saltos indirectos de referencia (*IJT-base*) y de dos niveles (*IJT+ITT*), para diferentes tamaños de memoria.

6.5. Conclusiones

En este capítulo se ha presentado una nueva implementación de un predictor de saltos indirectos. Se propone un método de codificar las direcciones destino de los saltos indirectos que proporciona una mejor relación entre la precisión de las predicciones y la cantidad total de bits dedicados a las predicciones. Esto permite aumentar la precisión del predictor o reducir sus requerimientos de memoria.

El método consiste en usar una primera tabla para almacenar las relaciones entre historia previa y el camino a escoger (codificado localmente), y una segunda tabla para traducir el identificador del camino a la dirección de memoria completa. La idea funciona porque hay pocos saltos indirectos que requieren el uso de *IJT*, y que usan un número moderado de direcciones destino, pero estos pocos saltos son complicados de predecir, y requieren el uso de historias de gran longitud para obtener una precisión apropiada.

En el diseño se prioriza una buena relación entre el tamaño del predictor y su precisión, y también un moderado consumo energético, a costa de aumentar la latencia. De todos modos, el hecho de que los saltos indirectos raramente estén a poca distancia de un fallo de predicción, detectado de forma experimental, y el gran ancho de banda que se le supone al predictor (utilizando, por ejemplo, las mejoras de los capítulos 4 y 5), permiten que esta latencia tenga un impacto muy reducido en el rendimiento.

Es importante remarcar que para un mismo número de entradas en *IJT*, el esquema de dos niveles (*IJT*+*ITT*) obtiene menor precisión que el esquema de un nivel (*IJT-base*), desde un 15% menos para 4K y 2K entradas, hasta casi un 40% menos para 128 entradas. Esta pérdida de precisión se debe a usar una tabla *ITT* relativamente pequeña. Sin embargo, cuando se considera la relación entre precisión y tamaño del predictor, el esquema de dos niveles resulta claramente superior, entre un 20% y un 40%.

Otra forma de interpretar los resultados de nuestra propuesta es que se consigue obtener la misma precisión para los saltos indirectos que el predictor de referencia pero con menores requisitos de memoria, por ejemplo se mantiene la precisión con una reducción del tamaño de 12 KBytes a 5 KBytes, o reduciendo el tamaño de 6 KBytes a 2,5 KBytes. Se podría entonces argumentar que se ayuda a las aplicaciones que no se benefician del predictor, poniendo a disposición del diseñador más memoria en el chip para dedicarla a mejorar otros aspectos del funcionamiento del procesador.

Como trabajo futuro más directamente relacionado con la propuesta de este capítulo se plantea la posibilidad de combinar la predicción múltiple de trazas con la predicción múltiple de saltos indirectos, y usar una misma tabla para ambos casos. Un compromiso para realizar la predicción múltiple es la amplitud de la predicción: escoger entre 4, entre 8 … Una posibilidad interesante, basada en la idea de la jerarquía y en la de otorgar más recursos a los casos más frecuentes, es la de encadenar la selección múltiple. Por ejemplo, al seleccionar entre cuatro opciones se obtiene un valor especial (una especie de código de “escape”) que indica que hay que continuar el proceso de selección entre un grupo de casos menos frecuentes.

7. Conclusiones

Resumen

Se presentan las conclusiones de esta tesis, detallando las aportaciones originales realizadas. Se presentan nuevas vías de trabajo, posibilitadas en parte por los logros de esta tesis, y que permiten continuar y extender la investigación.

7.1. Conclusiones y Principales Aportaciones

Sin lugar a dudas, la predicción de saltos ha sido y sigue siendo uno de los temas más importantes en la investigación sobre la microarquitectura de computadores. Esta tesis presenta varias propuestas dirigidas a incrementar la eficiencia en el uso de los recursos del predictor de flujo de control. Usando un algoritmo de predicción típico de los procesadores comerciales actuales, las propuestas analizadas en este trabajo permiten acelerar la velocidad del predictor, incrementar la anchura del predictor, y reducir sus requerimientos de memoria y de consumo energético. Todo ello permite alcanzar un alto rendimiento en los procesadores superescalares actuales y en los previstos para un futuro inmediato. También se ataca el problema de que la mejora de la tecnología de construcción de chips no reduce el tiempo de acceso a los elementos de memoria –un predictor consta básicamente de memoria- en la misma proporción que el tiempo de operación de los transistores.

El punto de partida de la investigación ha sido un predictor de bloques básicos desacoplado de la caché de instrucciones. Esta organización presenta como ventajas más reseñables su simplicidad –cada predicción sólo considera una instrucción de transferencia de control-, su eficiencia –no se gasta tiempo ni recursos con instrucciones que no son de transferencia de control- y la posibilidad de permitir aprovechar el exceso de ancho de banda del predictor por parte de las etapas siguientes en el procesador. El exceso de ancho de banda de predicciones -una especie de “conocimiento del futuro”- proporciona paralelismo potencial que permite ocultar retardos en los accesos a la jerarquía de memoria y compensar una mayor latencia asociada a diseños que se optimizan para reducir el consumo energético. Si el ancho de banda excediera la capacidad que el diseño es capaz de aprovechar, se puede reducir la frecuencia de reloj del predictor para obtener el ancho de banda adecuado con un menor gasto energético.

Una de las tareas, y sin duda de las más arduas, que supone una investigación como la que se describe en esta tesis, es establecer el entorno experimental que permita evaluar las propuestas realizadas. Tal como se ha explicado en el capítulo 3, ha supuesto tanto el análisis de programas de evaluación, como el uso y la modificación de herramientas de simulación para modelar los elementos de la microarquitectura que se han descrito. En concreto, el diseño y depuración del simulador ha significado un esfuerzo considerable. Parte de ese esfuerzo fue orientado a una aplicación docente, y dio lugar a la siguiente publicación:

[MoRL02a] J. C. Moure, D. I. Rexachs, and E. Luque: The KScalar Simulator. ACM *Journal of Educational Resources in Computing* (*JERIC*), Vol 2(1), pp. 73–116, 2002.

Para aumentar la velocidad del predictor –el número de predicciones realizadas por unidad de tiempo– se ha propuesto el uso de técnicas de predicción de vía y de predicción de índice. Un efecto lateral beneficioso a esta propuesta es la reducción del consumo energético del predictor. El capítulo 4 ha presentado un diseño detallado de un predictor organizado en forma de jerarquía de dos niveles, que integra de una forma muy entrelazada tanto la predicción de vía como la predicción de índice. Se ha estimado la reducción en la latencia del predictor utilizando herramientas de uso habitual en la disciplina. También se ha realizado un análisis detallado del funcionamiento de la nueva organización, que ha permitido determinar de forma cuantitativa el impacto de retardos que aparecen como consecuencia exclusiva de las técnicas aplicadas.

Usando como referencia una configuración razonable para el predictor, obtenida tras un análisis cuantitativo previo, se ha estimado el incremento en el ancho de banda para el nuevo diseño entre un 40% y un 150%. Este incremento en el ancho de banda del predictor se traduce en una mejora en el rendimiento del procesador que puede oscilar entre un 5% y un 25%.

Las propuestas sobre la aplicación de la predicción de vía e índice a un predictor de bloques básicos desacoplado de la caché de instrucciones han sido recogidas en las publicaciones que se presentan a continuación. Cabe reseñar que la propuesta descrita en esta tesis supone una extensión importante al contenido de estas publicaciones, primero en el nivel de detalle, y segundo en que la memoria de tesis incluye el uso de mecanismos propuestos y publicados muy recientemente, como la predicción anticipada y la prebúsqueda de instrucciones guiada por el predictor.

[MoRL02b] J. C. Moure, D. I. Rexachs, and E. Luque: Speeding Up Target Address Generation Using a Self-Indexed FTB. *Lecture Notes on Computer Science* (*LNCS*) 2400, *Euro-Par 2002*, pp. 517–521, 2002.

[MoRL03] J. C. Moure, D. I. Rexachs, and E. Luque: Optimizing a decoupled front-end architecture: the Indexed Fetch Target Buffer (iFTB). *Lecture Notes on Computer Science* (*LNCS*) 2790, *Euro-Par* 2003, pp. 566-575, 2003.

Para aumentar la anchura del predictor –el número medio de instrucciones proporcionadas por cada predicción– se ha partido de la propuesta de un predictor de trazas de instrucciones existente en la literatura, y referenciado por la práctica totalidad de trabajos relacionados. La gran ventaja de predecir trazas de instrucciones es que la complejidad del predictor no depende del número de instrucciones de transferencia de control que contenga una traza. Si, además, el predictor se desacopla de la caché de trazas, el resultado es que la casuística de la fase de predicción es muy similar a la que resulta de predecir bloques básicos, bastante simple. La complejidad inherente a la predicción de múltiples instrucciones de salto en una única operación, en forma de predicción de trazas, queda relegada a la fase de actualización del predictor y de la caché de trazas, fuera de la ruta crítica. Sin embargo, el diseño propuesto en la literatura no escala la relación entre la precisión y el tamaño del predictor al mismo ritmo que lo hace un diseño basado en la predicción de bloques básicos. Por tanto, el aumento de la anchura de las

predicciones se hace a costa de, o bien disminuir la precisión, o bien aumentar los recursos de memoria y por lo tanto la latencia del predictor.

La principal aportación descrita en el capítulo 5 ha sido proponer un nuevo algoritmo de predicción de trazas, que mejora la relación entre precisión y recursos de memoria, y la hace similar a la obtenida por un diseño basado en bloques básicos, pero con la ventaja añadida de una mayor anchura de predicción. Para lograrlo, se ha impuesto un límite al número de trazas que pueden comenzar por una misma instrucción, entre 2 y 4. Cuando este límite es 2, se alcanza un incremento en la anchura de predicciones cercano al 80%, manteniendo la misma precisión y requerimientos de memoria que un predictor de bloques básicos. Ampliar el límite a 4 trazas, permite alcanzar un incremento en la anchura de predicciones de hasta un 160%, manteniendo la misma precisión de las predicciones de bloques básicos, pero con un incremento en los requerimientos de memoria que puede llegar al 50%. Estas propuestas han sido recogidas en la publicación que se muestra a continuación.

[MBRL06] J. C. Moure, D. Benítez, D. I. Rexachs, and E. Luque: Wide and Efficient Trace Prediction using the Local Trace Predictor. *Proc. of the 20th Intl. Conf. on Supercomputing (ICS-17)*, pp. 55-65, June 2006.

Finalmente, en el capítulo 6 se ha presentado una nueva organización para predecir saltos indirectos. Se trata de codificar las direcciones destino de los saltos indirectos, de forma que una primera tabla almacena la correlación con la historia de los saltos previos, codificada con un identificador local, y una segunda tabla traduce el identificador local a la dirección destino completa. De este modo, se mejora la relación entre la precisión de las predicciones y la cantidad total de bits dedicados a las predicciones, y esto permite aumentar la precisión del predictor o reducir sus requerimientos de memoria. Priorizar una mejor relación entre el tamaño del predictor y su precisión, y un moderado consumo energético, supone aumentar la latencia de la predicción de saltos indirectos. Sin embargo, el gran ancho de banda

del predictor (mediante el uso de las mejoras descritas anteriormente) permite que esta latencia tenga un impacto muy reducido en el rendimiento.

[MBRL05] J. C. Moure, D. Benítez, D. I. Rexachs, and E. Luque: Target Encoding for Efficient Indirect Jump Prediction. *Lecture Notes on Computer Science* (*LNCS*) 3648, *Euro-Par* 2005, pp. 497-507, 2005.

7.2. Trabajo Futuro

Son múltiples las extensiones que se pueden realizar al trabajo presentado en esta memoria, y también las líneas generales de investigación que se abren a partir de este trabajo.

En cuanto a las extensiones a la investigación, una vía bastante natural consiste en aplicar las técnicas de predicción de vía e índice, analizadas en el capítulo 4, a la propuesta del predictor de trazas del capítulo 5. También se ha comentado la posibilidad de combinar la predicción múltiple de trazas con la predicción múltiple de saltos indirectos, es decir, unificar los esquemas descritos en los capítulos 5 y 6. De este modo, se haría un uso más eficiente de la memoria del predictor y se simplificaría el número de casos a considerar.

Otras vías alternativas para extender el trabajo son las siguientes:

- Aplicar y analizar las mismas ideas de diseño a algoritmos de predicción con perceptrones, [Jime03b], que se han mostrado ser una vía eficiente de utilizar historias de gran longitud para aumentar el uso de la correlación con el comportamiento pasado del programa.
- El análisis más detallado de la reducción en el consumo energético que se puede lograr con las propuestas de esta tesis, no sólo en el predictor, sino también, y en especial, en la caché de instrucciones.
- Aumentar la eficiencia del predictor de trazas explorando políticas alternativas para la selección de trazas y para el manejo de los casos

de desbordamiento de clase. Seleccionar trazas usando criterios dinámicos puede permitir tanto el incremento en la precisión como la disminución de la redundancia. Una idea prometedora es terminar las trazas en saltos poco deterministas (difíciles de predecir) y en cambio permitir incluir en las trazas los saltos altamente deterministas (fáciles de predecir), tal y como se analiza en [RMSR03].

El elevado ancho de banda de predicción que posibilitan las propuestas de esta tesis, junto a la organización desacoplada, permiten disponer de predicciones sobre el flujo de control con mayor anticipación, y esto posibilita líneas de investigación adicionales:

- diseñar una organización de la caché de trazas, posiblemente con una estructura jerárquica [RAM+04], que use de forma más eficiente su capacidad de memoria y su consumo energético. Las trazas críticas se pueden almacenar en tablas de acceso muy rápido, mientras que las menos críticas se pueden organizar de una forma más eficiente, por ejemplo usando un esquema de apuntadores similar al de la caché de trazas basada en bloques [BIRS99].
- las estrategias de prebúsqueda de datos, que permiten reducir la penalización de los fallos en la caché de datos, o las estrategias de predicción de valores, que reducen las penalizaciones debidas a las dependencias de datos, se benefician de disponer de la predicción del flujo de control con mayor anticipación.
- el uso de esquemas adaptables que utilicen información sobre el comportamiento del programa extraída de forma dinámica, como el propuesto en [BMRL05], permiten un uso más eficaz de la memoria del predictor, y por tanto alcanzar una mejor relación entre ancho de banda y precisión.
- los procesadores multi-hebra (*multithreading*) se benefician de disponer de un predictor unificado de gran tamaño y de alta velocidad. La memoria unificada que almacena la correlación con el

comportamiento histórico de los saltos permite un uso eficiente tanto si se dispone de una única hebra de ejecución como si coexisten varias. El mayor ancho de banda del predictor asegura que todas las hebras reciben un ancho de banda suficiente, para atacar de forma más efectiva el problema de la latencia de acceder a las instrucciones.

Referencias

[AHK+00] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger: Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. *Proc. of the 27th Intl. Symp. on Computer Architecture (ISCA-27)*, pp. 248–259, December 2000.

[AuLE02] T. Austin, E. Larson, and D. Ernst: SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, vol. 35(2), pp. 59-67, February 2002.

[BaVi01] Batson B., and T.N. Vijaykumar: Reactive-Associative Caches. *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques, (PACT'01)*, pp. 49–60, October 2001

[BBH+04] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman: The microarchitecture of the Intel Pentium 4 processor on 90 nm Technology. *Intel Technology Journal*, Vol. 8 (1), 2004.

[BlRS99] B. Black, B. Rychlik and J.P. Shen: The block-based trace cache. *Proc. 26th Int. Symp. on Computer Architecture (ISCA-26)*, pp. 196–207, May 1999.

[BMRL05] D. Benítez, J. C. Moure, D. I. Rexachs, E. Luque: Performance and Power Evaluation of an Intelligently Adaptive Data Cache. *Lecture Notes on Computer Science* (*LNCS*) 3769, *HiPC-2005*, pp. 363–375, December 2005.

[BrFl91] B.K. Bray and M. J. Flynn: Strategies for Branch Target Buffers. *Proc. of the 24th Intl. Symp. on Microarchitecure (MICRO-24)*, pp. 42-49, December 1991.

[BuAu97] D. Burger and T. M. Austin: The SimpleScalar Tool Set. Technical Report TR-1342, *University Wisconsin-Madison, Computer Science Department*, 1997.

[CaGr94a] B. Calder and D. Grunwald: Fast and accurate instruction fetch and branch prediction. *Proc. 21st Intl. Symp. on Computer Architecture (ISCA-21),* pp. 2–11, April 1994.

[CaGr94b] B. Calder and D. Grunwald: Reducing Indirect Function Call Overhead in C++ Programs. *Proc. 21th Int. Symp. on Principles of Programming Languages* (*ISPPL-21*), pp. 397–408, August 1994.

[CaGr95] B. Calder and D. Grunwald: Next Cache Line and Set Prediction. *Proc. of the 22nd Intl. Symp. on Computer Architecture (ISCA-22)*, pp. 287–296, June 1995.

[CaGr96] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. *Proc. of the 2nd Symp. on High-Performance Computer Architecture (HPCA-2)*, pp. 244-253, February 1996.

[ChEP97] P.-Y. Chang, M. Evers and Y. N. Patt: Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. *Int. Journal of Parallel Programming*, 25(5): 339-362, 1997.

[ChHP97] P.-Y. Chang, E. Hao and Y.N. Patt: Target prediction for indirect jumps. *Proc. of the 24th Intl. Symp. on Computer Architecture (ISCA-24)*, pp. 274–283, June 1997.

[CMMP95] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. *Proc. 22nd Intl. Symp. on Computer Architecture (ISCA-22)*, pp. 333-344, June 1995.

[DiAl92] K. Diefendorf and M. Allen: Organization of the Motorola 88110 superscalar RISC processor. *IEEE Micro*, 12(2) pp. 40–63, February 1992

[DrHo98a] K. Driesen and U. Hölzle: Accurate indirect branch prediction. *Proc. of the 25th Intl. Symp. on Computer Architecture (ISCA-25),* pp. 167–178, June 1998.

[DrHo98b] K. Driesen and U. Hölzle: The cascaded predictor: economical and adaptive branch target prediction. *Proc. of the 31st Intl. Symp. on Microarchitecture (MICRO-31)*, pp. 249–258, 1998.

[DrHo99] K. Driesen and U. Hölzle: Multi-stage cascaded prediction. *Proc. EuroPar '99 Conference*, LNCS 1685, pp. 1312-1321, September 1999.

[DuFr99] S. Dutta and M. Franklin: Control flow prediction schemes for wide-issue superscalar processors. *Trans. on Parallel and Distributed Systems*, IEEE, 10(4):346–359, April 1999.

[EdMu98] A. Eden and T. Mudge: The yags branch prediction scheme. *Proc. of the 31st Intl. Symp. on Microarchitecture (MICRO-31)*, pp. 69-77, December 1998.

[EvCP96] M. Evers, P.-Y. Chang, and Y. N. Patt: Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. *Proc. of the 23th Intl. Symp. on Computer Architecture (ISCA-23),* pp. 3–11, May 1996.

[FiFr92] J. Fisher and S.M. Freudenberger. Predicting Conditional Branch Directions from Previous Runs of a Program. *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 85–95, May 1992.

[FrPP98] D. H. Friendly, S. J. Patel, and Y. N. Patt: Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism. *Proc. of the 30th Intl. Symp. on Microarchitecture (MICRO-30)*, pp. 24–33, December 1997.

[GRB+03] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine: The Intel Pentium M processor: Microarchitecture and Performance. *Intel Technology Journal,* vol. 7(2), pp. 21–36, May 2003

[HePa03] J.L. Hennessy, and D.A. Patterson: Computer Architecture, a Quantitative Approach. 3rd Edition, *Morgan Kauffmann*, 2003

[Henn00] J. Henning: SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, vol. 33, no 7, pp. 28–35, July 2000.

[HSU+01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel: The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[JaRS97] Q. Jacobson, E. Rotenberg, and J. E. Smith: Path-Based Next Trace Prediction. *Proc. of the 30th Intl. Symp. on Microarchitecture (MICRO-30)*, pp. 14–23, December 1997.

[JBSS97] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith: Control flow speculation in multiscalar processors. *Proc. 3rd Intl. Symp. on High-Performance Computer Architecture (HPCA-3)*, pp. 218-229, 1997

[JiKL00] D. A. Jimenez, S. W. Keckler, C. Lin: The Impact of Delay on the Design of Branch Predictors. *Proc. of the 33th Intl. Symp. on Microarchitecture (MICRO-33)*, pp. 67–76, December 2000.

[Jime03a] D. A. Jimenez: Reconsidering Complex Branch Predictors. *Proc. 9th Int. Symp. on High-Performance Computer Architecture (HPCA-9)*, pp. 43–52, February 2003.

[Jime03b] D. A. Jimenez: Fast Path-Based Neural Branch Prediction. *Proc. of the 36th Intl. Symp. on Microarchitecture (MICRO-36)*, pp. 243–252, December 2003.

[JiLi02] D. A. Jimenez and C. Lin: Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4), November 2002

[John91] M. Johnson: Superscalar Microprocessor Design. *Innovative Technology.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1991

[JSHP97] S. Jourdan, J. Stark, T.-H. Hsing and Y. N. Patt: Recovery Requirements of Branch Prediction and Storage Structures in the Presence of Mispredicted-Path Execution. *Int. Journal of Parallel Programming*, 25(5):363-384, 1997.

[JuSN98] T. Juan, S. Sanjeevan, and J.J. Navarro. A third level of adaptivity for branch prediction. *Proc. of the 25th Intl. Symp. on Computer Architecture* (*ISCA-25*), pp. 155-166, June 1998

[KaEm91] D. R. Kaeli, P. G. Emma: Branch History Table Prediction of Moving Target Branches due Subroutine Returns. *Proc. 18th Intl. Symp. on Computer Architecture (ISCA-18)*, pp. 34-41, June 1991.

[KaKa98] J. Kalamatianos and D. R. Kaeli: Predicting indirect branches via data compression. *Proc. of the 31th Intl. Symp. on Microarchitecture* (*MICRO-31)*, pp. 272-281, December 1998.

[KJLH89] R. E. Kessler, R. Jooss, A. Lebeck. and M. D. Hill: Inexpensive implementations of Set-Associativity. *Proc. 18th Intl. Symp. on Computer Architecture (ISCA-16)*, pp. 131-139, June 1989.

[Kess99] R. Kessler: The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24-36, 1999.

[LeCM97] C.-C. Lee, I.-C. K. Chen, and T. Mudge: The bi-mode branch predictor. *Proc. of the 32nd Intl. Symp. on Microarchitecture* (*MICRO-22)*, pp. 4-13, December 1997

[LeSm84] J. K. F. Lee, A. J. Smith: Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer* 17(2):6-22, 1984.

[LLM88] M. Litzkow, M. Livny, and M. Mutka: Condor - A Hunter of Idle Workstations. *Proc. of the 8th Intl. Conf. of Distributed Computing Systems,* pp. 104-111, June 1988

[MaDu04] M. Mamidipaka and N. Dutt: eCACTI: An Enhanced Power Estimation Model for On-chip Caches. Technical Report #04-28 Center for Embedded Computer Systems, University of California, Irvine, 2004

[McFa93] S. McFarling: Combining Branch Predictors. Technical Report TN-36, *Digital Western Research Laboratory*, June 1993.

[MeSC97] K. N. Menezes, S. W. Sathaye, and T. M. Conte. Path prediction for high issue-rate processors. *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'97)*, pp. 178-188, November 1997.

[MiSU97] P. Michaud, A. Seznec, and R. Uhlig: Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. *Proc. 24th Intl. Symp. on Computer Architecture (ISCA-24)*, pp. 292 – 303, June 1997

[MoRL02a] J. C. Moure, D. I. Rexachs, and E. Luque: The KScalar Simulator. ACM *Journal of Educational Resources in Computing* (*JERIC*), Vol 2(1), pp. 73–116, 2002.

[MoRL02b] J. C. Moure, D. I. Rexachs, and E. Luque: Speeding Up Target Address Generation Using a Self-Indexed FTB. *Lecture Notes on Computer Science* (*LNCS*) 2400, *Euro-Par 2002*, pp. 517–521, 2002.

[MoRL03] J. C. Moure, D. I. Rexachs, and E. Luque: Optimizing a decoupled front-end architecture: the Indexed Fetch Target Buffer (iFTB). *Lecture Notes on Computer Science* (*LNCS*) 2790, *Euro-Par* 2003, pp. 566-575, 2003.

[MBRL05] J. C. Moure, D. Benítez, D. I. Rexachs, and E. Luque: Target Encoding for Efficient Indirect Jump Prediction. *Lecture Notes on Computer Science* (*LNCS*) 3648, *Euro-Par* 2005, pp. 497-507, 2005.

[MBRL06] J. C. Moure, D. Benítez, D. I. Rexachs, and E. Luque: Wide and Efficient Trace Prediction using the Local Trace Predictor. *Proc. of the 20th Intl. Conf. on Supercomputing (ICS-17)*, pp. 55-65, June 2006.

[Nair95] R. Nair: Dynamic path-based branch correlation, *Proc. of the 28th Intl. Symp. on Microarchitecture* (*MICRO-28)*, pp. 15-23, December 1995.

[ObSo03] P. S. Oberoi and G. S. Sohi: Parallelism in the front-end, *Proc. 30th Int. Symp. on Computer Architecture (ISCA-30),* pp. 230-240, June 2003.

[PaSR92] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-V*), pp. 76-84, October 1992.

[PSZS04] D. Parikh, K. Skadron, Y. Zhang, and M. Stan: Power-Aware Branch Prediction: Characterization and Design. *IEEE Trans. On Computers* Vol. 53, nº 2, 168–186, 2004

[PaFP99] S. J. Patel, D. H. Friendly, and Y. N. Patt: Evaluation of Design Options for the Trace Cache Fetch Mechanism. *IEEE Transactions on Computers*, 48(2):193–204, 1999.

[PaEP98] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *Proc. 25th Int. Symp. on Computer Architecture (ISCA-25),* pp. 262–271, June 1998.

[PeWe94] A. Peleg and U. Weiser: Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line. *U.S. Patent* Number 5,381,533, Intel Corporation, 1994.

[PAV+01] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, K. Roy: Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping. *Proc. of the 34th Intl. Symp. on Microarchitecture (MICRO-34)*, pp. 54-65, 2001.

[RSLV02] A. Ramírez, O. J. Santana, J. L. Larriba-Pey, and M. Valero: Fetching Instruction Streams. *Proc. of the 35th Intl. Symp. on Microarchitecture (MICRO-35)*, pp. 371–382, December 2002.

[ReAC99] G. Reinman, T. Austin, B. Calder: A Scalable Front-End Architecture for Fast Instruction Delivery. *Proc. of the 26th Intl. Symp. on Computer Architecture (ISCA-26)*, pp. 234-245, June 1999.

[ReCA01] G. Reinman, B. Calder, T. Austin: Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE Trans. on Computers*, 50(4): 338–355, 2001

[ReCA02] G. Reinman, B. Calder, T. Austin: High Performance and Energy Efficient Serial Prefetch Architecture. *Intl. Symp. On High Performance Computing (ISHPC-4)*, pp. 146–159, October 2002.

[ReJo99] G. Reinman, N. Jouppi: An Integrated Cache Timing and Power Model. *COMPAQ Western Research Laboratory,* www.research.digital.com/wrl/people/jouppi/CACTI.html, 1999.

[RMSR03] R. Rosner, M. Moffie, Y. Sazeides, and R. Ronen: Selecting Long Atomic Traces for High Coverage. *Proc. of the 17th Intl. Conf. on Supercomputing (ICS-17)*, pp. 2-11, June 2003.

[RAM+04] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A Mendelson: Power Awareness through Selective Dynamically Optimized Traces. *Proc. of the 314th Intl. Symp. on Computer Architecture (ISCA-31)*, pp. 162-173, May 2004.

[RoBS96] E. Rotenberg, S. Bennett and J. E. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. *Proc. of the 29th Intl. Symp. on Microarchitecture (MICRO-29)*, pp. 24-34, December 1996.

[RoBS99] E. Rotenberg, S. Bennett, J. E. Smith: A Trace Cache Microarchitecture and Evaluation. *IEEE Trans. on Computers*, 48(2):111–120, 1999.

[SAMC98] K. Skadron, P. Ahuja, M. Martonosi, and D.W. Clark: Improving prediction for procedure returns with Return-Address-Stack repair mechanisms. *Proc. of the 31th Intl. Symp. on Microarchitecture (MICRO-31)*, pp. 259-271, November 1998.

[SCAP97] E. Sprangle, R. S. Chappell, M. Alsup and Y. N. Patt: The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. *Proc. of the 24th Intl. Symp. on Computer Architecture (ISCA-24)*, pp. 284 – 291, June 1997.

[SeFr03] A. Seznec and A. Fraboulet: Effective ahead pipelining of instruction block address generation. *Proc. 30th Intl. Symp. on Computer Architecture (ISCA-30),* pp. 241-252, June 2003

[SeJS96] A. Seznec, S. Jourdan, P. Sainrat and P. Michaud: Multiple-block ahead branch predictors. *Proc of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* pp. 116–127, October 1996

[SFKS02] A. Seznec, S. Felix, V. Krishnan, Y. Sazeides: Design tradeoffs for the Alpha EV8 Conditional Branch Predictor. *Proc. of the 29th Intl. Symp. on Computer Architecture (ISCA-29)*, pp. 295-306, May 2002.

[ShAr00] H. Sharangpani, and K. Arora: Itanium Processor Microarchitecture. IEEE *Micro*, 7:24-43, August 2000.

[ShSC03] T. Sherwood, S. Sair, and B. Calder: Phase Tracking and Prediction. *Proc. of the 30 Intl. Symp. on Computer Architecture (ISCA-30)*, pp. 336-349, 2003

[ShJo01] P. Shivakunar and N. Jouppi: CACTI 3.0: An Integrate Cache Timing, Power, and Area Model. WRL Research Report 2001/2, Aug. 2001

[SkMC00a] K. Skadron, M. Martonosi, and D. W. Clark: Speculative Updates of Local and Global Branch History: A Quantitative Analysis. *Journal of Instruction Level Parallelism*, vol. 2, January 2000 (http://www.jilp.org/vol2).

[SkMC00b] K. Skadron, M. Martonosi, and D.W. Clark: A taxonomy of branch mispredictions, and alloyed prediction as a robust solution to wrong-history mispredictions. *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques, (PACT'00)*, pp. 199–206, October 2000.

[Smit81] J. E. Smith, "A study of branch prediction strategies," *Proc. of the 8th Intl. Symp. on Computer Architecture (ISCA-8),* pp. 135-148, May 1981

[StEP98] J. Stark, M. Evers and Y. N. Patt: Variable Length Path Branch Prediction. *Proc. of the 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VIII), pp. 170–179, October 1998.

[StRP97] J. Stark, P. Racunas, and Y. N. Patt: Reducing the Performance Impact of Instruction Cache Misses by writing instructions into the Reservation Stations Out-of-Order. *Proceedings of the 30th Intl. Symp. on Microarchitecture (MICRO-30)*, pp. 34-43, December 1997.

[TaTG96] M.-D Tarlescu, K. B. Theobald, and G. R. Gao: Elastic history buffer: a low-cost method to improve branch prediction accuracy. *Proc. Intl. Conf. on Computer Design (ICCD '96)*, pp. 82–87, October 1996.

[TFWS03] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark. Improving Branch Prediction by Dynamic Dataflow-Based Identification of Correlated Branches from a Large Global History. *Proc. 30th Intl. Symp. on Computer Architecture (ISCA-30)*, pp. 314-323, June 2003.

[WaBa97] S. Wallace and N. Bagherzadeh: Multiple Branch and Block Prediction. *Proc. of the 3rd Intl. Symp. on High-Performance Computer Architecture*, *(HPCA-3),* pp. 94–103, February 1997.

[Wen92] Chih-Po Wen. Improving Instruction Supply Efficiency in Superscalar Architectures Using Instruction Trace Buffers. *Proc. Symp. Applied Computing*, Volume 1, pp. 28-36, March 1992.

[YePa91] T.-Y. Yeh and Y.N. Patt: Two-Level Adaptive Branch Prediction. *Proc. of the 24th Intl. Symp. on Microarchitecture (MICRO-24)*, pp. 51-61, December 1991.

[YePa92a] T.-Y. Yeh and Y.N. Patt: Alternative Implementations of Two-Level Adaptive Branch Prediction. *Proc. of the 19th Intl. Symp. on Computer Architecture (ISCA-19)*, pp. 124 – 134, May 1992.

[YePa92b] T.-Y. Yeh and Y. N. Patt: A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. *Proc. of the 25th Intl. Symp. on Microarchitecture (MICRO-25)*, pp. 129-139, December 1992.

[YeMP93] T.-Y. Yeh, D. T. Marr, Y. N. Patt: Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. *Proc. 7th Intl. Conf. on Supercomputing (ICS'93)*, pp. 67–76, July 1993.

[YePa93] T.-Y. Yeh and Y. N. Patt: Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors. *Proc. 26th Annual Intl. Symp. on Microarchitecture, (MICRO-26)*, pp. 164–175, December 1993

[Yung96] R. Yung: Design Decisions Influencing the Ultrasparc's Instruction Fetch Architecture. *Proc. of the 29th Intl. Symp. on Microarchitecture (MICRO-29)*, pp. 178-190, December 1996.