UNIVERSIDAD DE ZARAGOZA

Tesis Doctoral

Zaragoza, Abril de 2005

Alternativas de Diseño en Memoria Cache de Primer Nivel Multibanco

D. Enrique F. Torres Moreno

Directores:

Dr. Pablo Ibáñez Marín

Dr. Víctor Viñals Yúfera

DEPARTAMENTO DE INFORMÁTICA E INGENIERÍA DE SISTEMAS

Alternativas de Diseño en Memoria Cache de Primer Nivel Multibanco

Memoria presentada por

D. Enrique F. Torres Moreno

para obtener el título de Doctor Ingeniero en Informática

Departamento de Informática e Ingeniería de Sistemas

Zaragoza, Abril de 2005

Dirigida por:

Dr. Pablo Ibáñez Marín

Dr. Víctor Viñals Yúfera

UNIVERSIDAD DE ZARAGOZA
http://www.unizar.es

Dept. de Informática e Ingeniería de Sistemas
http://diis.unizar.es

Resumen de la Tesis

Una solución prometedora para la distribución del primer nivel de cache es la separación del camino de datos a memoria en secciones. Gracias a la predicción de banco previa a iniciar las instrucciones de acceso a memoria y agrupando en cada camino de datos a memoria las unidades generadoras de dirección, el banco de memoria y las unidades consumidoras se consigue el ansiado ancho de banda de baja latencia necesario para conseguir alto rendimiento. En este trabajo nos centramos en la distribución tanto de los bancos de cache como de las estructuras de acceso a memoria situadas en el camino crítico del suministro load-uso. En concreto, plantearemos predictores de banco fácilmente implementables y que permitan realizar varias predicciones por ciclo. Propondremos mecanismos de recuperación en caso de error de predicción de latencia que funcionen adecuadamente en sistemas con altas tasas de ejecución especulativa de instrucciones. Analizaremos politicas conservadoras para iniciar la ejecución de un load a varios bancos para reducir la latencia load-uso en caso de error de predicción de banco. Estudiaremos técnicas de partición escalable con estrategias de distribución y replicación de contenidos que disminuyan conflictos sin degradar otras tasas de error. Y por último, abordaremos la distribucción del Store Buffer con el objetivo de suministrar datos de stores en vuelo a loads dependientes a latencia del primer nivel de cache mientras se mantiene un número elevado de stores en vuelo.

Palabras Clave- Microarquitectura, Jerarquía de Memoria, Primer Nivel de Cache, Caches Multibanco, Cache Seccionada, Predictor de Banco, Predicción de Latencia, Store Buffer.

pa' Anabel

Agradecimientos

"*Esta costumbre "literaria" de expresar agradecimientos al terminar un trabajo escrito no se entiende bien hasta que se ha atravesado el trance de la Tesis doctoral. La felicidad por la obra terminada, y el alivio por poner punto y final a una empresa que hipoteca la vida entera, resultan en agradecimientos mucho más que retóricos hacia quienes la han hecho posible. Pero la felicidad y el alivio son estados de ánimo, efímeros, y podrían arrojar dudas sobre la durabilidad del sentimiento de gratitud que generan.*" (frases extraídas de *google*)

Una página no alcanzaría para citar a todos los que directa o indirectamente contribuyeron para obtener los resultados que se presentan, sin embargo, y corriendo el riesgo de caer en odiosas omisiones, se referencia a continuación a los pilares de esta investigación. La deuda que tengo con las personas que enumeraré a continuación rebasa sobradamente su relación puntual con esta Tesis doctoral.

Mi gratitud a Jose María Llabería sólo puede expresarse con pobreza en unas líneas de texto. Gracias por tu ejemplo y estímulo para seguir creciendo. Me he beneficiado de tu agudeza intelectual, tus críticas, sugerencias, rigurosidad y pragmatismo científicos, y te agradezco la confianza y franqueza que has demostrado conmigo.

A mis directores Pablo y Víctor, por haber sido capaces de transmitirme su pasión por la docencia y la investigación.

A mis padres, quienes me infundieron la ética y el rigor que guían mi transitar por la vida. A mis hermanos por estar siempre ahí.

A mi mujer, por su comprensión y el amparo incondicional durante los muchos años que le dediqué a este trabajo de Tesis. A mis hijos Enrique y Diana, por ser mi estímulo diario. Esto es también vuestro premio.

Al *gaZ* y en especial a Luisma y a las chicas de *al'lado*. A todos y cada uno de los profesores, compañeros y amigos del departamento de Informática e Ingeniería de Sistemas.

La deuda personal se complementa en este caso con la material:

El trabajo desarrollado en esta Tesis se ha visto apoyado con las financiaciones y recursos computacionales que se listan a continuación.

- Proyectos de I + D financiados en convocatorias públicas nacionales del Ministerio de Ciencia y Tecnología, Plan Nacional de I + D + I, Programa PRONTIC. Son los siguientes tres proyectos coordinados con el DAC-UPC:
  - *Computación de Altas Prestaciones II. Ocultación de Latencia.* CICYT-TIC-1998-0511-C02-02.
  - *Computación de Altas Prestaciones III. Jerarquía de Memoria de Altas Prestaciones.* CICYT-TIC-2001-0995-C02-02.
  - *Computación de Altas Prestaciones IV. Jerarquía de Memoria de Altas Prestaciones.* CICYT-TIN2004-07739-C02-02
- *gaZ*: Reconocimiento Grupo Emergente, Diputación General de Aragón 2003.
- *gaZ*: Reconocimiento Grupo Consolidado, Diputación General de Aragón 2004.
- HiPEAC: European N.E. on High-Performance Embedded Architecture and Compilation
- Recursos computacionales que se listan a continuación:
  - CEPBA: Centro Europeo de Paralelismo de Barcelona
  - DIIS: Dept. de Informática e Ingeniería de Sistemas y
  - gaZ: Grupo de Arquitectura de Zaragoza

Contenido

CAPÍTULO 1

Introducción

En este capítulo de introducción centramos la memoria. En primer lugar expondremos el contexto tecnológico en el que se ha desarrollado la Tesis y motivaremos el trabajo realizado. A continuación, describiremos los trabajos previos relacionados con caches de primer nivel multibanco y las colas de loads / stores. Y finalmente resumiremos las contribuciones desarrolladas en la Tesis y comentaremos la organización de la memoria.

1.1 Introducción

Esta Tesis se centra en la Computación de Altas Prestaciones. Los avances en la alta escala de integración de circuitos y las mejoras tecnológicas han permitido incrementar el número de transistores y la frecuencia de reloj de los procesadores. Estos factores imponen nuevo desafíos. Se necesitan diseños que eviten los efectos negativos de los retrasos dentro del chip y simultáneamente mantengan la simplicidad que ayude a verificar y comprobar el funcionamiento correcto en chips con centenares de millones de transistores.

Los procesadores actuales son capaces de extraer mayor rendimiento implementando ejecución fuera de orden, segmentaciones con muchas etapas, mayores anchos de inicio de instrucciones y grandes ventanas de instrucciones donde buscar paralelismo a nivel de instrucción (ILP - *Instruction Level Parallelism*). Numerosos estudios muestran que en futuras tecnologías el retardo de las interconexiones disminuye mucho más lentamente que el de las

puertas lógicas, y previsiblemente este fenómeno tendrá un gran impacto negativo en la latencia de las cada vez mayores estructuras hardware [AHKB00][Matz97]. Las altas frecuencias de reloj de los procesadores y la latencia de las señales dentro del chip dificultan el acceso a estructuras centralizadas.

Una condición necesaria para conseguir rendimientos elevados es el suministro de datos a la velocidad de proceso. La velocidad de los procesadores es cada vez mayor que la de las memorias. En sistemas de altas prestaciones, la latencia y ancho de banda de las jerarquías actuales serán un verdadero freno para las arquitecturas del futuro. Para evitarlo se requiere de múltiples accesos concurrentes a una memoria cache rápida y no bloqueante [SoFr91]. Esto es especialmente crítico en la ejecución de códigos enteros (por ejemplo en SPECint), sin demasiado paralelismo del que sacar partido, y donde el rendimiento es muy sensible al retardo *load-uso*; en particular al retardo del primer nivel de la jerarquía (L1) compuesto por la memoria cache de primer nivel y las *colas de load*/*store*.

Se han propuesto y usado diversas alternativas en procesadores comerciales para alcanzar estos requerimientos; caches con más de un puerto a cada celda (*True Multiported* [NaHa02]), caches replicadas (*Mirror Caches* [EdRR95][SoFr91]) y/o caches virtualmente multipuerto (*Virtual Multiported* [Gwen93][MBB+98]).

La Figura 1.1 muestra el camino de datos y el diagrama de ciclos de un camino a memoria multipuerto.

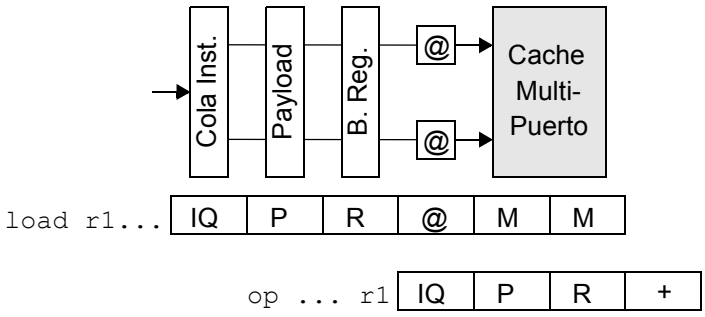

Figura 1.1 Camino de datos y diagrama de ciclos de un camino a memoria MultiPuerto.

Dos referencias a memoria cualesquiera pueden ser servidas cada ciclo. Las instrucciones son selecionadas por la *Cola de Instrucciones* (IQ) para iniciar su ejecución. Las instrucciones leen toda la información necesaria para la

ejecución (P - *Payload*) y leen sus operandos fuentes en el Banco de Registros (R). Las instrucciones calculan la dirección efectiva de memoria (@) y despiertan de forma especulativa a sus instrucciones dependientes a tiempo para que puedan coger el dato en la red de cortocircuitos. Los *loads* acceden al primer nivel de cache (M) y suministran el dato (M).

Otras opciones que buscan ampliar el ancho de banda del puerto de acceso son: *load all* y *line buffer* propuestas en el 1996 por Wilson y Olukotun [WiOl01] y la *Locality-Based Interleaved Cache* (LBIC) propuesta por Rivers y otros en [RTDA97]. Aunque con ligeras diferencias, la idea fundamental en ambos trabajos es servir simultaneamente todas las peticiones pendientes a una determinada línea de cache, basándose en el incremento del ancho de lectura del puerto.

Todas estas alternativas han sido ampliamente estudiadas y es conocido que cada una de ellas tiene sus propias restricciones en términos de área, latencia y sobre todo de escalabilidad en el futuro.

1.2 Motivación

Para tener una idea general de los compromisos existentes a la hora de conseguir una L1 eficiente, hemos realizado una serie de experimentos sobre el procesador fuera de orden *8-inicio*[1] con un primer nivel de cache multipuerto. La Figura 1.2 muestra el rendimiento en instrucciones consolidadas por ciclo (IPC-*Instructions Per Cycle*) al variar tres parámetros importantes de la cache L1 como son: capacidad (*eje X*), latencia (*lat*) y número de puertos (*P*). Los números pertenecen a la ejecución de los códigos enteros de SPEC 2K. La línea superior muestra el resultado de una cache L1 de dos ciclos de latencia y cuatro puertos de acceso (*2lat-4P*). La línea inferior pertenece a una cache L1 de 4 ciclos de latencia y 4 puertos de acceso (*4lat-4P*). Entre ambas líneas existe una importante diferencia de IPC comprendida entre el 1,98 y el 2,37. Se puede ver que la capacidad de L1 es un parámetro importante, (aproximadamente un incremento del 2% en IPC cada vez que la duplicamos), sin embargo no es el más importante.

1. El modelo de procesador se presentará ampliamente en el Capítulo 2.

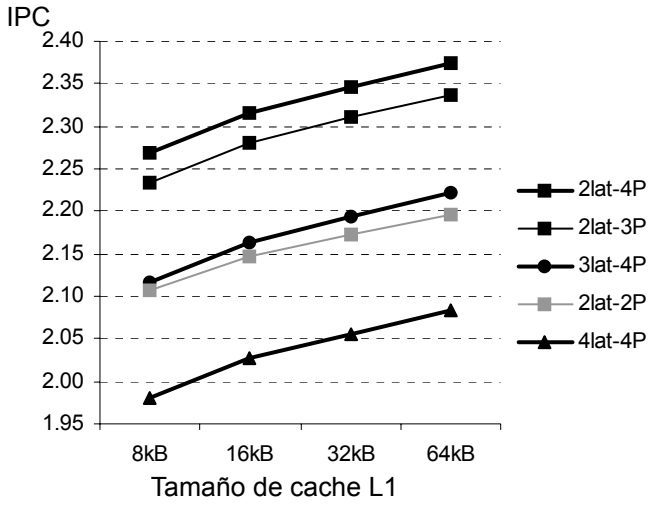

Figura 1.2 IPC al ir variando capacidad, latencia y número de puertos de una cache L1 multipuerto sobre el procesador *8-inicio* descrito en el Capítulo 2.

El número de puertos disponibles incrementa su importancia conforme los limitamos: al pasar de *2lat-4P* a *2lat-3P* y de esta a *2lat-2P* vemos como el IPC decrece un 1,5% y un 6%, respectivamente, para cualquier capacidad de banco.

Por otro lado, incrementar la latencia en un ciclo acarrea una pérdida del 7%, tanto de *2lat-4P* a *3lat-4P* como desde *3lat-4P* a *4lat-4P*. Si vamos un poco más lejos, en el entorno analizado, una cache con 3 puertos y latencia 2 ciclos de tan solo 8 Kbytes de capacidad (*2lat-3P*) siempre funciona mejor que cualquier cache de 3 ciclos de latencia (*3lat-4P*). Lo mismo sucede entre caches L1 con 3 ciclos de latencia y cualquiera de 4 ciclos.

Es obvio que nos gustaría incluir el diseño de 64 Kbytes con cuatro puertos y dos ciclos de latencia como cache de primer nivel. Desgraciadamente dicha L1 no puede ser construida con esa latencia tan baja si planeamos utilizar tecnologías presentes o futuras y diseños orientados a altas frecuencias de reloj.

1.3 Caches L1 Multibanco

Recientemente, las Caches Multibanco [BaDA03][Case93][ChYL01][Hsu94] [KMAC94][Kuma97][NeVB00][RaPa03][RTDA97][SoFr91][YMRJ99][ZyKo 01] han sido propuestas como una alternativa interesante frente a las monolíticas caches multipuerto. Multibanco distribuye los datos (líneas o palabras de cache) entre varios bancos pequeños y simples, que almacenan contenidos disjuntos y son accedidos con caminos segmentados independientes. Por todo ello, los diseños multibanco mantienen la latencia baja. Una desventaja que presenta es la aparición de conflictos cuando varias peticiones quieren acceder al mismo banco en el mismo ciclo [HSU+01]

1.3.1 Opciones de Diseño Multibanco

Multibanco necesita recursos de encaminamiento. Una opción, para enlazar las unidades de cálculo de dirección (*AGU*) y los bancos, es a través de una red de interconexión (*crossbar*) [Case93] [HSU+01][Kuma97] tal y como se muestra en la Figura 1.3. Multibanco permite escalar el ancho de banda, pero la red de interconexión puede incrementar la latencia de los *loads* (ciclo marcado con "*X*") y podría provocar nuevos conflictos.

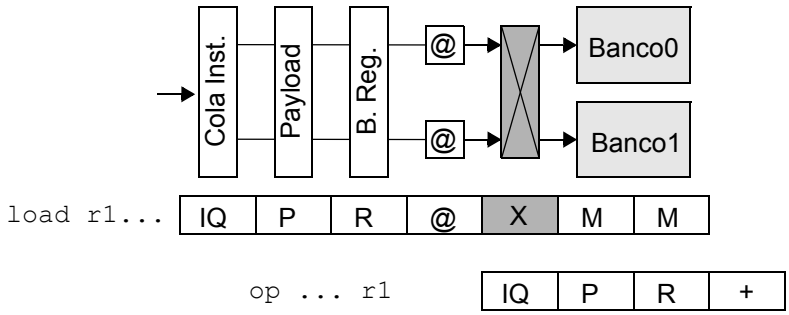

Figura 1.3 Camino de datos y diagrama de ciclos de una organización Multibanco con red de interconexión (*crossbar -X-*).

Para reducir conflictos y aumentar la productividad (*throughput*) se puede utilizar un planificador de memoria que arbitre el acceso a los bancos y ofrezca cierta capacidad de almacenamiento para peticiones pendientes [Case93] [Kuma97]. Este planificador se ubica después de calcular la dirección y antes de acceder a los bancos.

Esta opción (Figura 1.4), ampliamente estudiada ([AlKo97] [JuNT97] [NeVB00] [RTDA97] [SoFr91] [WiOl01]), incrementa y hace variable la latencia de los *loads*. Las instrucciones de memoria, tras el cálculo de la

dirección efectiva de memoria, acceden a una segunda cola de instrucciones (LSQ) que planifica el uso de los puertos de acceso a los bancos. Las instrucciones dependientes no pueden ser despertadas hasta que la notificación de selección del *load* por parte de la LSQ para acceder al banco alcance la IQ. El arbitraje en la LSQ y la notificación pueden incrementar la latencia. Además, como el tiempo de estancia en la LSQ es variable (en función de los conflictos), la latencia del *load* es variable.

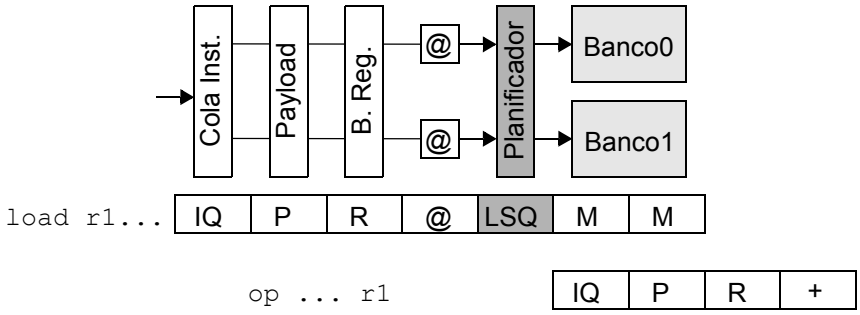

Figura 1.4 Camino de datos y temporización de multibanco con planificador de memoria (LSQ).

Otra opción consiste en encaminar las peticiones antes de calcular la dirección. El camino a memoria seccionado (*The Sliced Memory Pipeline* [YMRJ99]) elimina la red de interconexión del camino de datos, usando la cola de inicio de instrucciones (IQ-*Instruction Queue*) para encaminar las peticiones (Figura 1.5). Así mismo, minimiza el retardo distribuyendo los bancos físicamente cerca de las *AGU* y de las unidades funcionales consumidoras.

Como la IQ no conoce la dirección de memoria ni el banco destino, y hay un sólo puerto de inicio a memoria a cada banco, se necesita un predictor de banco para planificar el inicio de instrucciones. Se establece un lazo hardware entre la IQ y la etapa de cálculo de dirección responsable de la comprobación de la predicción de banco (un *loose loop* según Borch y otros [BTME02]).

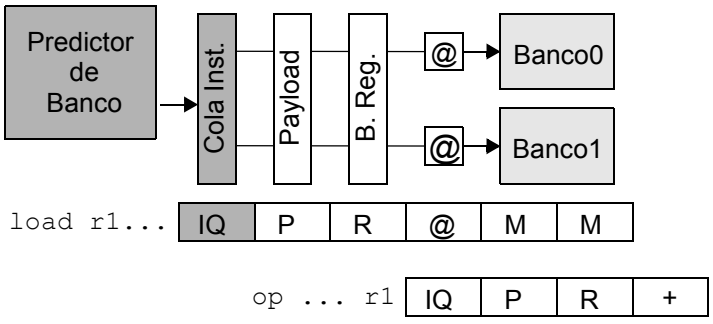

Figura 1.5 *Sliced Memory Pipeline*, reduce el retardo del camino eliminando la interconexión y distribuyendo los bancos físicamente cerca de las Unidades Funcionales.

1.3.2 Predicción de Latencia

Este lazo hardware entre la IQ (etapa de recuperación) y la etapa de cálculo de dirección (comprobación de banco) se puede gestionar usando predicción de latencia, al igual que se realiza al presuponer tiempo de acierto en cache.

La Figura 1.6 muestra la ejecucción de un *load* y el despertar especulativo de sus instrucciones dependientes. Por claridad se muestra únicamente el inicio de una instrucción por ciclo.

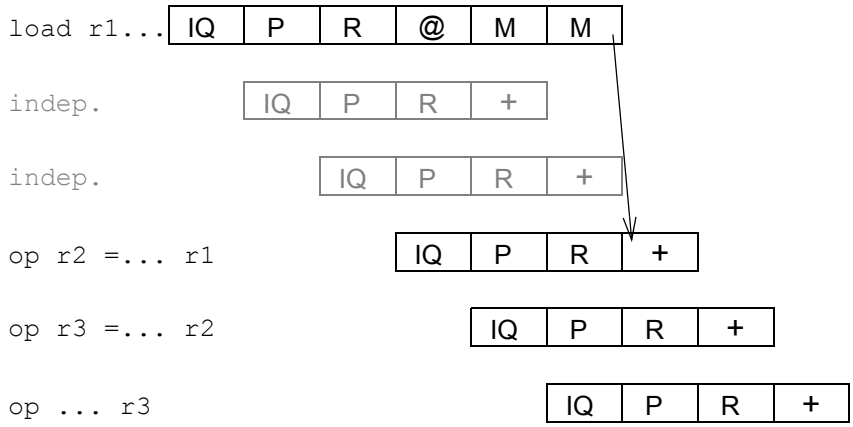

Figura 1.6 Predicción de latencia, diagrama de tiempos asumiendo 2 ciclos entre IQ y EX. Por claridad mostramos el inicio de una única instrucción por ciclo.

La IQ presupone acierto en la predicción de banco y acierto en cache, y de forma especulativa planifica el inicio de ejecución, en el momento adecuado, de las instrucciones dependientes del *load*. Si la predicción de latencia es mayoritariamente correcta, el rendimiento aumenta. Sin embargo, con cada error de predicción se pierde todo el trabajo especulativo y además se necesitan acciones correctoras [MoLO01]. El rendimiento obtenible con la predicción de latencia es directamente proporcional a los aciertos del predictor de banco.

1.3.3 Distribución de Contenidos

Los errores de predicción de banco y la contención en los puertos de inicio a memoria de la IQ producidas por ráfagas de instrucciones que quieren acceder al mismo banco, son los principales problemas del camino de datos a memoria seccionado. Las primeras propuestas [SoFr91][Case93][Kuma97][JuNT97] distribuían los datos mediante una *función de hash* de su dirección efectiva, pudiendo distinguir *entrelazado por línea* y *entrelazado por palabra*.

En los tres últimos años (coincidiendo con el trabajo en esta Tesis) han aparecido políticas dinámicas para distribuir datos entre bancos. Racunas y otros [RaPa03] proponen distribuir los datos en función del conjunto de trabajo (*working set*), mejorando la predicción de banco pero incrementando la contención. Limaye y otros [LiRS01] sugieren replicar datos bajo demanda para minimizar la contención y al hacerlo aumentan los errores de banco y los fallos de cache. [ChYL01] propone aislar las referencias a pila en un módulo de cache con una gestión adaptable a ese patrón concreto de referencia.

1.3.4 *Colas de Loads / Stores*

Los procesadores actuales, para acelerar la ejecución del programa, permiten la ejecución de instrucciones *load* y *store* fuera de orden utilizando un predictor de independencia. Cuando se ejecuta un *store*, la *cola de loads* (LDB-*LoaD Buffer*) es accedida por si hubiera un *load* más joven ejecutado erróneamente, esto es, el *load* ha obtenido un dato de la cache cuando debería haber obtenido el dato escrito por el *store* previo en orden de programa a la misma dirección de memoria. Para mantener la consistencia de memoria, los *loads* deben también acceder al LDB. Estos errores de predicción de independencia de memoria, diferente a los errores de predicción de latencia, implican habitualmente vaciar el camino de datos y reiniciar la ejecución desde la etapa de búsqueda a partir del *load* ejecutado erróneamente.

Al ejecutar un *load*, la *cola de stores* (STB-*STore Buffer*) es accedida para buscar el dato más reciente sin consolidar en orden de programa. La Figura 1.7 muestra las estructuras del primer nivel de cache.

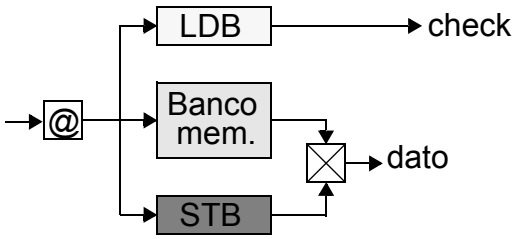

Figura 1.7 Acceso a estructuras del primer nivel de cache.

Las *colas de loads*/*stores* deben tener bastante capacidad si el número de instrucciones en vuelo es grande, y necesitan soportar más accesos por ciclo al aumentar el ancho de inicio de ejecución.

Muchos trabajos se han centrado en otras partes del procesador, como la cola de instrucciones, el banco de registros físico, los cortocircuitos, pero pocos se han centrado en las *colas de loads*/*stores*. Tanto los bancos de cache como la *cola de stores* deben mantener latencias similares y escalarse adecuadamente al resto del procesador para no limitar el rendimiento. La implementación distribuida del primer nivel debe llevar consigo la descentralización del resto de las estructuras. De no ser así, el tiempo de acceso puede ser muy superior al de la cache de primer nivel, degradando las prestaciones.

Las *colas de loads*/*stores* se suelen construir usando estructuras completamente asociativas (CAM-*Content Addresable Memory*). Este tipo de memorias disponen de un mecanismo que permite comparar las direcciones de memoria para garantizar el orden correcto de dependencias entre instrucciones *load* y *store* (Figura 1.8). El tiempo de acceso a las estructuras CAM crece de forma logarítmica con el número de entradas [CaLi04]. Para prevenir que este tiempo de acceso afecte negativamente al tiempo de ciclo del procesador, trabajos recientes han explorado variaciones de las *colas load*/*store* convencionales que permiten reducir el tamaño de la *cola de loads* o filtrar el número de búsquedas en ambas colas. Todos estos trabajos se han centrado en caches monolíticas.

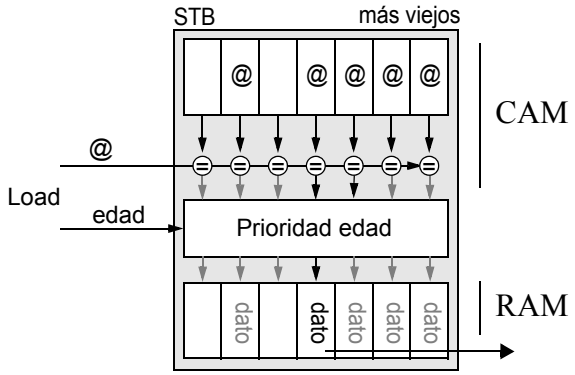

Figura 1.8 Lógica de suministro. Comparar dirección del *load* contra la de todos los *stores* del STB (CAM). De los que coinciden, seleccionar el dato del *store* más joven de entre los más viejos que el *load* (RAM).

No existen trabajos que descentralicen las colas y escalen su tamaño. En [RaPa03][YMRJ99][ZyKo01] las colas se distribuyen pero no se escalan los tamaños. La reserva de las entradas en los STBs se realiza en *emisión*, cuando todavía no se conoce el camino que va a tomar el *store*. Para evitar problemas de *Deadlock*, se reserva espacio en los STBs de todos los caminos. Además, los *stores* deben permanecer en las colas hasta su consolidación, por lo que el STB de cada camino debe tener espacio suficiente para albergar a todos los *stores* en vuelo. Un número insuficiente de entradas en los STBs provoca el bloqueo de la

etapa de emisión. Así que según estas propuestas, el tamaño de cada STB debe ser igual al que tendríamos en un sistema centralizado pero con menos puertos. En [YMRJ99] se replican los *stores* a cada camino en ejecución, perdiendo ancho de banda efectivo de inicio de ejecución a memoria. En [ZyKo01] se inician por un solo camino, pero una vez conocida la dirección se debe realizar una comunicación entre caminos para informar del resultado. En [RaPa03] sólo en caso de error de banco se necesita comunicación entre STBs para intercambiar la información.

Sethumadhaven y otros [SDB+03] reducen el número de búsquedas que deben mirar la cola completa y filtran la inserción de los *loads* ejecutados en orden. Park y otros [PaOV03] filtran las búsquedas en el STB de aquellos *loads* que se predice como independientes de *stores* previos. También segmentan las búsquedas en el STB a costa de sufrir una latencia variable. Ponomorev y otros [PoKG01] proponen, por cuestiones de consumo, desactivar partes de las colas cuando la carga es baja, pero no enfocan el problema de escalabilidad. Cain y Lipasti [CaLi04] eliminan la CAM del LDB convirtiéndola en una simple FIFO sin necesidad de realizar búsquedas, pero incrementan el número de accesos a la cache. Akkary y otros proponen una organización jerárquica del STB en dos niveles, donde el primer nivel rápido y pequeño alberga los n-últimos *stores* emitidos, mientras el resto se almacena en un gran segundo nivel más lento [AkRS03].

Estas propuestas ofrecen una cierta solución al problema de escalabilidad de las colas con menos búsquedas y más rápidas, a costa de incrementar la complejidad (y previsiblemente la latencia) de esta parte del procesador ya bastante complicada.

En resumen:

Multibanco se plantea como una alternativa viable en el primer nivel de la jerarquía de memoria. Sin embargo quedan diversos problemas pendientes de solución. El camino a memoria seccionado es muy sensible a la precisión del predictor de banco y al mecanismo de recuperación en caso de error de predicción. En la literatura se pueden encontrar diversos predictores de dirección [BJR+99][GoVB01] adaptables a multibanco, o la adaptación de predictores de saltos a multibanco [YMRJ99]. No obstante, sus resultados no han sido contrastados en el ámbito de los procesadores superescalares. También existen propuestas para la recuperación en caso de error de predicción de latencia, en [MoLO01] se pueden encontrar además de sus propias propuestas un compendio de las preexistentes. Previo a multibanco, la predicción de

latencia se empleaba para recuperarse en caso de fallo de cache (L1), y la recuperación se producía mientras los datos llegaban de los niveles superiores de la jerarquía. En multibanco, los datos en la mayoría de los casos ya están presentes en el primer nivel, además, la tasa de error del predictor de banco suele ser mayor que la tasa de fallos del primer nivel de cache por lo que la penalización por recuperación debe ser reducida.

Otro de los grandes problemas de multibanco es la contención en los puertos de inicio de ejecución de la cola de instrucciones (IQ). Esta contención se produce cuando ráfagas de instrucciones quieren acceder al mismo banco. Por ello, se necesitan mecanismos que permitan distribuir y gestionar adecuadamente los contenidos del primer nivel de cache para minimizar el efecto de las ráfagas.

La *cola de stores* (STB) es la encargada de suministrar el dato proveniente de un *store* más viejo no consolidado a los *loads* más jóvenes que acceden a la misma dirección de memoria. Por tanto, el STB se encuentra en el camino crítico de suministro de datos de los *loads*. Esta estructura debe permanecer pequeña para mantener los tiempos de acceso acotados al del primer nivel de cache. Los diversos trabajos previos no han reducido su tamaño, aunque han disminuido el número de puertos de acceso.

1.4 Contribuciones de la Tesis

El objetivo de esta Tesis es diseñar y estudiar distintas organizaciones distribuidas de memoria cache que permitan múltiples caminos de acceso sin penalizar la temporización del procesador.

Se estudian y se exponen predictores de banco fácilmente implementables y que permiten varias predicciones por ciclo. Se proponen mecanismos de recuperación de errores de predicción de orden *store-load* (*Recuperación desde el RIB*) y de error de predicción de latencia (*Recuperación en Cadena*) que funcionan adecuadamente en sistemas con ejecución especulativa de instrucciones. Se experimentan y se plantean propuestas de replicación selectiva de *loads* marcados como de baja confianza para reducir la penalización por error de banco.

Las anteriores propuestas se presentaron en el ***9th*** *International* ***Euro-Par*** *Conference* celebrado en Austria en agosto de 2003 y recogido en *Lecture Notes in Computer Science* 2790 [TIVL03].

Se presenta una taxonomía de distribución de contenidos en primeros niveles de cache con caminos de acceso a memoria seccionados atendiendo a las políticas de distribución de contenidos y al nivel de replicado de los datos para conseguir mayor ancho de banda. Se estudia el compromiso entre capacidad del primer nivel, la contención en los puertos de inicio de la IQ y la precisión del predictor de banco.

La taxonomía y el estudio fue presentado en el ***10th*** *International* ***Euro-Par*** *Conference* celebrado en Pisa en agosto de 2004 y recogido en *Lecture Notes in Computer Science* 3149 [TIVL04].

Por último, se han realizado estudios de implementaciones distribuidas de las estructuras de datos hardware relacionadas con el control de memoria situadas en el camino crítico, en especial el *Store Buffer* (STB). Se formula un diseño del STB capaz de suministrar datos a latencia del primer nivel y de mantener la información de un gran número de *stores* en vuelo.

Este trabajo se presentará en el *32º Annual* ***I****nternational* ***S****ymposium on* ***C****omputer* ***A****rchitecture* a celebrar en Madison, Wisconsin USA en junio de 2005 [TIVL05].

1.5 Organización de la memoria

Esta memoria se ha estructurado en seis capítulos además de éste, dos apéndices y el listado de referencias.

El Capítulo 2 presenta la metodología y el modelo detallado de procesador base que se va a utilizar a lo largo de esta Tesis. También se introduce el simulador empleado y las modificaciones realizadas junto con la metodología necesaria para su validación. Se finaliza el capítulo enunciando la carga de trabajo y la plataforma de simulación.

El Capítulo 3 se centra en el camino a memoria seccionado y la predictabilidad del acceso a bancos según la función de distribución de contenidos.

En el Capítulo 4 veremos la repercusión en el rendimiento de los errores de predicción de banco. Propondremos la *Recuperación en Cadena* para reducir la penalización por error de predicción de banco y la *Replicación selectiva en emisión* de *loads* para reducir la tasa de error.

El Capítulo 5 presenta una taxonomía de la gestión de contenidos en caches multibanco y estudia el compromiso entre capacidad del primer nivel, contención en los puertos de inicio de la IQ y precisión de la predicción de banco en los distintos sistemas propuestos.

En el Capítulo 6 se estudian diseños del *Store Buffer* adecuados para caminos a memoria seccionados. El objetivo es suministrar datos de *stores* en vuelo a *loads* dependientes a latencia de L1 mientras se mantiene un número elevado de *stores* en vuelo.

Por último, en el Capítulo 7 expondremos las conclusiones globales de esta Tesis y mostraremos las líneas abiertas.

En el Apéndice A se comenta brevemente la aplicación VisualPtrace diseñada como herramienta de validación del simulador y que ha permitido analizar el comportamiento interno del segmentado fuera de orden. En el Apéndice B se completan los resultados experimentales obtenidos a lo largo de la Tesis.

Concluye la memoria con el listado de las referencias.

CAPÍTULO 2

Metodología y Entorno de Evaluación

Este capítulo comenta la metodología empleada y detalla el entorno de evaluación utilizado para dar soporte a los experimentos mostrados en la Tesis.

2.1 Introducción

Todas las propuestas presentadas en esta Tesis han sido analizadas basándonos en herramientas de evaluación, concretamente en la simulación detallada de la ejecución de un conjunto de programas de prueba sobre un determinado modelo de procesador.

Un simulador de un procesador modela su temporización interna a nivel de ciclos. El simulador combina la ejecución ciclo a ciclo (lenta) con eventos discretos. Los simuladores permiten centrarse en aspectos concretos modelando distintos niveles de abstracción según el detalle requerido y eliminando detalles no tan importantes de los sistemas reales. Al trabajar a este nivel, el simulador puede modelar el comportamiento de las instrucciones a ejecutar en el procesador modelado (paso por etapas del segmentado, ocupación de componentes, proceder ante eventos simultáneos, etc.). A partir de la simulación se puede evaluar el rendimiento y cuantificar otra serie de factores importantes.

La simulación permite inferir el comportamiento de procesadores actuales y futuros y es el medio habitual dentro de la investigación y el desarrollo en arquitectura de computadores. Permiten una gran flexibilidad y son

relativamente fáciles de modificar y adaptar a un amplio abanico de posibilidades dentro del espacio de diseño.

En este capítulo describiremos en detalle el modelo de procesador. En la sección 2.2. Se comentarán tanto las etapas del segmentado como los recursos generales para pasar a continuación a describir el modelo de desambiguación de memoria y predicción de latencia. La sección 2.3 la dedicaremos a especificar la jerarquía de memoria y el camino de datos de las instrucciones de acceso a memoria.

En la sección 2.4 se comenta el simulador, enunciando brevemente las modificaciones efectuadas y el método de validación de resultados. La sección 2.5 describe los programas y juego de datos de prueba. Por último la sección 2.6 comenta la plataforma hardware / software empleada durante el desarrollo de esta Tesis.

2.2 Procesador Base

En esta sección presentamos las características principales del procesador modelado a lo largo de esta Tesis.

2.2.1 Descripción del Procesador

Modelamos un procesador super-escalar super-segmentado similar a los existentes en el mercado. La organización en bloques se muestra en la Figura 2.1.

El procesador se compone principalmente de un frontal (*front-end*) en orden y un cuerpo (*back-end*) fuera de orden. El frontal es el encargado de alimentar las

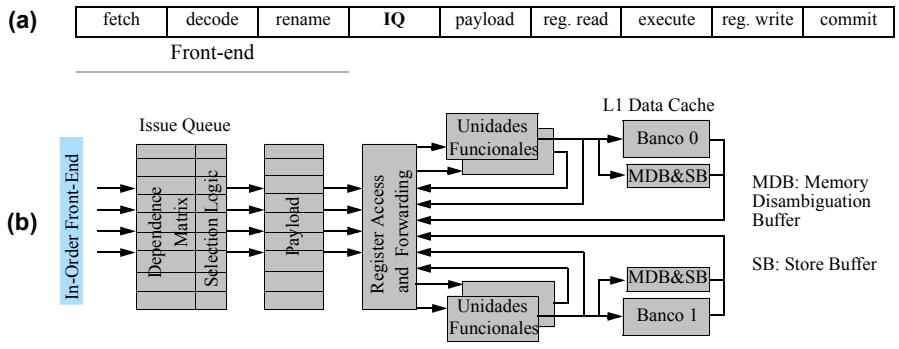

Figura 2.1 (a) Diagrama de tiempo. (b) Cuerpo fuera de orden y camino de datos simplificado mostrando dos bancos

instrucciones a ejecutar al cuerpo del procesador. Modelamos un frontal de 8 etapas entre búsqueda e IQ necesarios para *buscar*, acceder a la cache de instrucciones, seleccionar, alinear, *decodificar*, *renombrar* y *emitir* las instrucciones a las colas del cuerpo fuera de orden (*back-end)*. En un procesador segmentado estas ocho etapas sólo afectan a las prestaciones al recuperarnos de un error de predicción de salto.

El cuerpo del procesador es el responsable de la ejecución fuera de orden de las instrucciones y de *consolidar* en orden. Las instrucciones una vez emitidas esperan en la IQ al inicio de la ejecución. La IQ contiene una matriz de dependencias y una lógica de selección, (ver Figura 2.1). La matriz de dependencias aglutina la información de dependencias entre instrucciones y permite determinar qué instrucciones están listas al tener disponibles todos sus registros fuente y no estar serializadas por dependencias a través de memoria (fase de *Wake-up*). La lógica de selección inicia la ejecución de las instrucciones más viejas entre las ya listas (fase de *Selección*).

Tras el inicio de ejecución las instrucciones leen el resto de la información necesaria para ejecutar la instrucción (*payload*), acceden al banco de registros o a la red de cortocircuitos para leer sus operandos fuentes y comienzan la ejecución en las unidades funcionales apropiadas. Una vez ejecutadas, las instrucciones escriben los resultados en el banco de registros. Finalmente, las instrucciones que han completado su ejecución son retiradas y su estado se consolida en orden de programa.

Si se mantiene la tendencia de aumento de frecuencia del procesador, el número de etapas entre IQ y ejecución continuará aumentando [AHKB00][BTME02] [HSU+01]. Por ello tomaremos este número como parámetro y lo iremos variando en los experimentos para analizar su impacto.

2.2.2 Parámetros del procesador

Modelamos dos procesadores con distinto grado de escalaridad (anchura). La mayor parte de los experimentos se basan en un procesador agresivo de grado ocho (*8-inicio*), capaz de ir a buscar 8 instrucciones por ciclo utilizando dos direcciones (PC) distintas, decodificar, renombrar y emitir ocho instrucciones por ciclo. Permite iniciar la ejecución fuera de orden de hasta 4 instrucciones de coma flotante más 8 instrucciones de enteros por ciclo, incluidos 4 accesos a memoria (compartiendo con enteros los puertos de inicio desde la IQ). El segundo procesador asemeja a un procesador más actual con la mitad de grado

de escalaridad (*4-inicio*) capaz de ir a buscar cuatro instrucciones consecutivas. El resto de parámetros y latencia de las unidades funcionales se muestra en la Tabla 2.1.

Tabla 2.1 Principales parámetros de los procesadores simulados

	4-inicio	**8-inicio**
Ancho de búsqueda, decodificación y emisión	4	8
Predictor de saltos: híbrido	(*bimodal*+*gshare*) 16 bits	
Entradas en el Reorder Buffer	128	256
Nº de *Loads* y *Stores* en vuelo	64	128
Nº max. de *Stores* en Vuelo	64	128
Entradas en IQ enteros / coma flotante	25 / 15	64 / 32
Unidades funcionales enteros / c.f.	4 / 2	8 / 4

en ciclos	**Latencia**	**Tasa Inicio**
enteros ALU	1	1
enteros MUL	7	1
enteros DIV	67	67
c.f. ADD, MUL, CMP y CVT	4	1
c.f. DIV	16	16
c.f. SQRT	35	35

Centrando el comentario en el procesador *8-inicio*, se pueden tener hasta 256 instrucciones en vuelo entre las etapas de emisión y consolidación. En total puede haber hasta 128 *Loads* o *Stores* en vuelo (pudiendo llegar al máximo cualquiera de los dos). Modelamos dos colas de inicio de ejecución de instrucciones (IQ), una para instrucciones de enteros y memoria y la otra para coma flotante. El número de unidades funcionales y sus latencias asemejan a las del Digital Alpha 21264 con el escalado en número adecuado en el *8-inicio*.

2.2.3 Desambiguación

Asumimos un modelo de desambiguación de memoria (dependencias de datos a través de memoria) basado en el adoptado por el procesador Digital Alpha 21264 [KeMW98][Kess99]. Las instrucciones *load* guardan sus direcciones en una *cola de loads* (LDB - *Load Buffer*) y las instrucciones *store* depositan la dirección y el dato a la vez en una *cola de stores* (STB - *STore Buffer*). En el modelo base de procesador, las entradas en las colas se asignan en orden al emitir, se rellenan en ejecución, y finalmente se liberan al consolidar.

Los *stores* no se inician hasta que los registros necesarios para calcular la dirección y el dato estén disponibles. Utilizamos un predictor de dependencias a través de memoria basado en *Store Sets*, tal y como se describe en [ChEm98], para ejecutar instrucciones *load* antes de conocer las direcciones de todos los *stores* precedentes. El orden predicho es gestionado desde la IQ, retardando el inicio de un *load* hasta el inicio del *store* previo, si una dependencia entre ellos ha sido creada por el predictor. Los errores de predicción de orden son descubiertos al ejecutar el *store*, posiblemente muchos ciclos después de que el *load* y sus instrucciones dependientes hayan abandonado la IQ.

Habitualmente, para recuperarse de un error de predicción de orden, el *load* y todas las instrucciones más jóvenes (dependientes o no) son eliminadas del segmentado. Las instrucciones vuelven a ser de nuevo leídas desde la cache de instrucciones. Sin embargo, las instrucciones a recargar son las mismas que fueron eliminadas. Para reducir la penalización de recuperación por error de predicción de orden, la recuperación se puede realizar desde una estructura que almacena todas las instrucciones renombradas que denominaremos *Buffer de Instrucciones Renombradas* (**RIB**-*Renamed Intruction Buffer*) mostrado en la Figura 2.2.

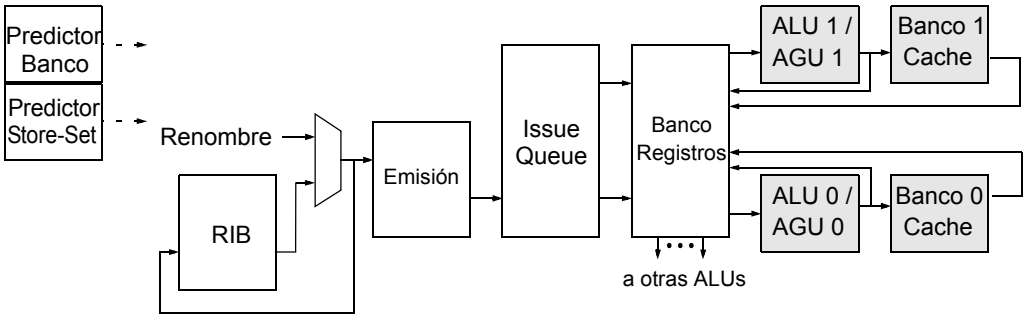

Figura 2.2 Recuperación desde el RIB y camino de datos simplificado mostrando 2 bancos

El RIB se encuentra después de las etapas de decodificación y de renombre y antes de la etapa de emisión. El RIB se rellena continuamente y mantiene en orden de programa todas las instrucciones en vuelo ya renombradas. La recuperación consiste en volver a emitir secuencialmente las instrucciones desde el RIB a partir del *load* ejecutado erróneamente. La recuperación no es selectiva ya que se volverán a emitir todas las instrucciones en orden de programa desde el *load*. Como la recuperación no se lleva a cabo desde la etapa de búsqueda, no se tiene que realizar una copia de seguridad de la tabla de mapeo de registros tal y como se hace con cada salto. Un mecanismo similar fue

propuesto por Lebeck y otros [LKL+02] para tolerar fallos de caches de latencia grande. Ellos realizaban una recuperación selectiva y por ello el RIB es mucho más simple que la estructura de datos que ellos proponían.

El RIB es un *buffer* circular. Una vez decodificadas y renombradas, el RIB guarda todas las instrucciones en vuelo. Tiene un puerto de lectura y uno de escritura. Se pueden escribir hasta 8 instrucciones en orden de programa por ciclo. En la recuperación, se pueden leer hasta 8 instrucciones consecutivas cada ciclo.

2.2.4 Predicción de latencia

Las instrucciones *load* tienen latencia variable dependiendo, por ejemplo, de si aciertan o fallan en cache. El 90% aproximadamente acierta en cache, por ello, y para acelerar la ejecución, se puede presuponer latencia de acierto en cache y despertar especulativamente a las instrucciones dependientes del *load* sin tener que esperar la confirmación del acierto.

El procesador puede iniciar instrucciones mientras espera la resolución de todas las predicciones de latencia, dígase predicción de banco o predicción de acierto en cache. Cada predicción tiene su propio retardo de resolución. La IQ asume la latencia de acierto en cache para los *loads*, y de forma especulativa despierta a las instrucciones dependientes del *load* durante un conjunto de ciclos denominados ***Ventana Especulativa*** (*Speculative Window*). La *ventana especulativa* comienza con el primer ciclo en el que las instrucciones dependientes directamente del *load* pueden ser iniciadas, y termina o bien cuando llega el reconocimiento positivo de la última predicción, o cuando un error de predicción llega a la etapa de recuperación.

La Figura 2.3 muestra la ejecución de una secuencia de instrucciones. Por claridad se muestra únicamente el inicio de una instrucción por ciclo. Durante los dos ciclos siguientes al inicio del *load* sólo se pueden iniciar instrucciones independientes de él. La IQ asume latencia de acierto en cache para el *load* y se despierta especulativamente a las instrucciones dependientes directamente del *load*. Durante el cuarto, quinto y sexto ciclo se pueden iniciar de forma especulativa instrucciones dependientes del *load*.

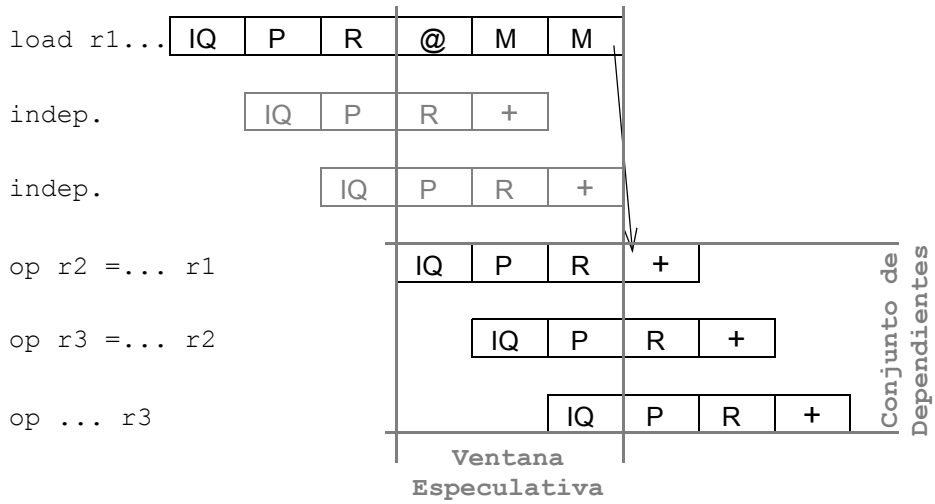

Figura 2.3 Predicción de latencia, diagrama de tiempos asumiendo 2 ciclos entre IQ y EX. Por claridad mostramos el inicio de una única instrucción por ciclo.

Denominamos ***Conjunto de Dependientes*** (*Dependent Set*) a las instrucciones dependientes iniciadas durante la *ventana especulativa*. Utilizamos la IQ como etapa de recuperación de los errores de predicción de latencia. El mecanismo de recuperación base fue descrito por Morancho y otros en [MoLO01].

Después de cada ciclo de inicio, se identifica a las instrucciones dependientes y pasan a formar parte del *conjunto de dependientes*. Las instrucciones no dependientes se desalojan de la IQ. Por el contrario, el *load* y el *conjunto de dependientes* se mantienen en la IQ hasta el último ciclo de la *ventana especulativa*.

La recuperación comienza tan pronto como la IQ es notificada del error de predicción. Es selectivo, de forma que tan sólo las instrucciones dependientes del *load* ya iniciadas serán reejecutadas.

2.3 Jerarquía de Memoria

Describiremos los distintos niveles de la jerarquía de memoria, comenzando con la descripción del camino de datos al primer nivel de memoria cache seccionado. A continuación comentaremos la estructura y temporización de los distintos niveles de la jerarquía.

2.3.1 Camino de datos a memoria seccionado

El camino de datos a memoria del primer nivel de cache está dividido en secciones. Cada sección está unida a una única unidad funcional de cálculo de dirección (ALU/AGU). Suponemos que la IQ dispone de un cierto número de puertos de inicio compartidos entre las unidades de enteros y las secciones de memoria. Las instrucciones *load* y *store* se emiten a la IQ llevando consigo la predicción de sección de cache destino (predictor de banco). Con esta predicción la IQ planifica el inicio de la ejecución a la sección de memoria determinada.

Para recuperarnos de un error de predicción de banco, que es un caso particular de error de predicción de latencia, las instrucciones se mantienen en la IQ hasta que se verifica la predicción.

La comprobación de banco se realiza concurrentemente con el cálculo de dirección evaluando la expresión A+B=K sin propagación de acarreo [CoLl92]. En caso de error de banco, la IQ es notificada durante el ciclo siguiente a la comprobación de banco (ver Figura 2.4). Junto a la notificación del error la IQ recibe el identificador de banco correcto. La instrucción mal-predicha pasa a estar lista un ciclo más tarde y puede ser iniciada, esta vez al banco correcto.

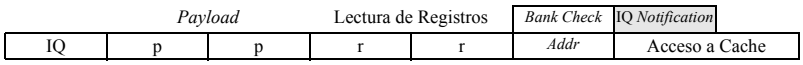

Figura 2.4 Retardo de la comprobación de banco. El banco se comprueba a la vez que se cálcula la drección efectiva. El resultado es notificado a la IQ durante el ciclo siguiente.

2.3.2 Niveles de Cache

Modelamos tres niveles de cache dentro del chip. El primer y segundo nivel de la jerarquía son multibanco con L2 y L3 centralizados. La Figura 2.5 muestra el camino de datos de la jerarquía. Cada banco de cache dispone de un único puerto de acceso de lectura o escritura compartido por *loads*, *stores* tras la

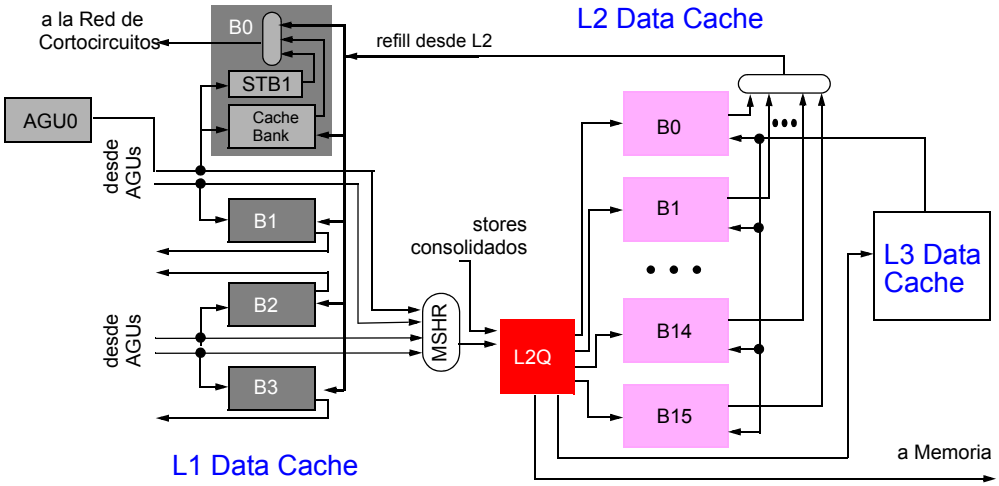

Figura 2.5 Camino de datos simplificado mostrando la iteración entre los dos primeros niveles multibanco de la jerarquía.

consolidación y rellenado de la cache (*refill*). Los bancos de L1 están unidos a L2 por un sólo puerto de rellenado de 32 Bytes (tamaño de línea de L1). De cada banco de cache L1 parte un sólo camino de datos a la red de cortocircuitos compartido por el banco de cache de L1, el STB y el suministro de datos coincidente con el *refill* desde L2..

Los fallos de cache en L1 son gestionados utilizando MSHR (*Miss Status Hold Registers*). Las peticiones a la L2 son gestionadas por una cola de peticiones pendientes a L2 (L2Q - *Level 2 Queue*) después de pasar por las etiquetas de L2 al igual que ocurre en el procesador Itanium II de Intel [NaHa02].

La L2Q puede enviar cada ciclo hasta 4 peticiones libres de conflicto a los 16 bancos entrelazados de L2 (16 Bytes). Los bancos de L2 tienen un acceso no segmentado de dos ciclos. Los fallos de L2 se envían a L3 . El rellenado de L2 desde L3 ocupa 8 bancos. El sistema permite hasta ocho fallos de L2 pendientes.

El segundo nivel unificado de cache consta de 256 Kbytes de capacidad total, con un tamaño de línea de 128 Bytes y asociatividad 8. El tercer nivel unificado es de 4 MBytes con tamaño de línea de 128 Bytes y asociatividad 16. La Tabla 2.2 resume los parámetros principales de la jerarquía de memoria.

Tabla 2.2 Parámetros principales de la jerarquía de memoria

L1 I-cache	64KB, 4-way	*L2 Cache Unificada* 256 KB	16 bancos, 8-way, 7 ciclos
L1 D-cache	2 cycles		
bancos	2 / 4	tamaño de línea	128 Bytes
puertos/banco	1 r/w	L2 MHSR	8 entradas
capacidad por banco	8 KB, 4-way	*L3 Cache Unified* 4MB	16-way, 19 ciclos
tamaño de línea	32 Bytes		
MHSR	16 entradas	tamaño de línea	128 Bytes
Store-Set Pred.	4K-entradas *Tabla SSI*	Bus L3-main mem.	8 ciclos/chunk
	128-entradas *Tabla LFS*	main mem. lat.	200 ciclos

2.3.3 Camino de datos de los *loads* y *stores*

Instrucciones *Load*. El acceso se realiza en paralelo a los bancos de cache y al STB. La Figura 2.6 muestra la temporización del acceso a memoria de las instrucciones *load*.

Se accede al banco durante un ciclo, necesitando un ciclo extra para alcanzar la red de cortocircuitos y distribuir el dato leído (línea 2-3, ciclos 2 y 3). La línea 4 en la Figura 2.6 muestra la temporización en fallo del primer nivel y acierto en el segundo. La línea 5 en la Figura 2.6 muestra la temporización de un fallo en L2 y acierto en L3.

Las instrucciones pendientes de fallos en el primer nivel son reiniciadas desde la IQ a tiempo de coger el dato coincidiendo con el rellenado desde el segundo nivel (línea 2 ciclos 10-11 en acierto L2 y ciclos 22-23 en acierto en L3).

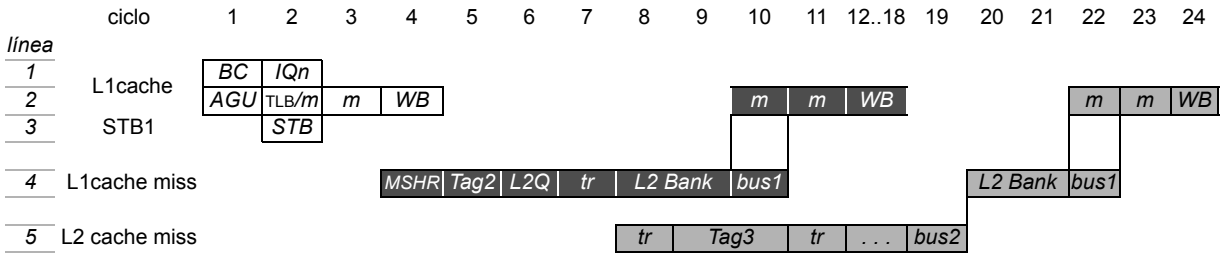

Figura 2.6 Diagrama de tiempo. BC, IQn y tr, significan comprobación de banco, notificación a la IQ, y ciclo de tránsito, respectivamente. busX y TagX son uso del bus y acceso a etiquetas, respectivamente. TLB y m denotan acceso al Translation Look-aside Buffer y acceso a L1, respectivamente.

Instrucciones *Store*. El primer nivel de la cache es "escritura inmediata" y "no-asignación en fallo de escritura" (*Write Through* y *Write Around*). Las instrucciones *store* son consolidadas sobre L2 (*Write Back*), y si los datos estaban en L1 (filtrado por el directorio de L2) se colocarán en un *buffer de escritura* con compactación de 8 entradas siendo éste local a cada banco de cache (no mostrado). Los *buffers de escritura* almacenan *stores* consolidados y actualizarán los bancos de cache en ciclos libres.

2.4 Simulador dirigido por ejecución

Los experimentos realizados en esta Tesis se han basado en SimpleScalar 3.0c [BuAu97]. Simplescalar es una infraestructura de simulación dirigida por ejecución compuesta por un conjunto de herramientas y de varios simuladores base. El código fuente está disponible de forma abierta, por lo que resulta fácilmente modificable para que los investigadores puedan adaptarlo a sus necesidades. Simplescalar permite la simulación de múltiples Arquitecturas del Lenguaje Máquina, nosotros simulamos código Alpha [Alph99].

En concreto sobre Simplescalar hemos modificado dos simuladores preexistentes:

Sim-Safe: simulador funcional que interpreta y ejecuta las instrucciones de los programas de prueba. Este simulador ha sido ligeramente modificado para generar el flujo de instrucciones *load* y *store* con sus direcciones efectivas de memoria. Sobre él, hemos implementado los distintos predictores de banco para obtener estadísticas.

Sim-Outorder: simulador temporal complejo de un procesador superscalar fuera de orden. De simulación más lenta que el anterior, `sim-outorder` modela la temporización interna de un procesador ciclo a ciclo.

Hemos modificado ampliamente `sim-outorder` para modelar el procesador descrito, en concreto:

- Separar la funcionalidad entre *ReOrder Buffer* (ROB) y colas de inicio de ejecución (IQ). IQ separadas para enteros y coma flotante y de menor tamaño que el ROB.

- Predicción de latencia. El inicio de instrucciones especulativo y la recuperación han sido modeladas con cuidado, simulando los ciclos

necesarios para leer el *payload* y acceder al banco de registros. El camino de datos de las instrucciones *load* y *store* se ha modelado con sumo detalle.

- Predictor de orden *store-load*. Se han modelado tanto los mecanismos de predicción de orden como la gestión de dependencias predichas en la IQ. Así mismo se ha implementado la recuperación desde el RIB en caso de error.

- Jerarquía de memoria. Se ha reescrito completamente el interface con memoria de Simplescalar modelando la jerarquía en tres niveles comentada tanto a nivel de ciclos como de contención en las colas y buses. En particular se ha creado el primer nivel multibanco distribuido y el segundo nivel multibanco centralizado de la jerarquía de memoria.

La validación de un simulador ciclo a ciclo es una tarea difícil debido a las complejas relaciones que se establecen entre los distintos componentes, más cuando se simula un procesador tan completo como el descrito. Para validar los resultados obtenidos con el simulador se han empleado diversas estrategias.

- Los resultados obtenidos en la simulación de los *benchmarks* se han comparado con los resultados de la ejecución nativa de los mismos y con los resultados obtenidos por otros investigadores.

- Depuración de los fragmentos modificados del código fuente del simulador paso a paso y comparación de los resultados con los previos a la modificación.

- Comprobaciones redundantes a lo largo del código para revisar el correcto funcionamiento del segmentado.

- Multitud de chequeos de consistencia a lo largo de todas las etapas para detectar errores (*panic*).

Con todo ello y aunque la simulación paso a paso sea correcta, resulta difícil validar la corrección temporal de la ejecución de las instrucciones. Esta tarea se complica enormemente al modelar mecanismos complejos de recuperación.

Durante la Tesis se desarrolló una aplicación visual que permite ver y seguir el estado del segmentado. Con la aplicación se puede tanto validar la corrección temporal de la ejecución como profundizar en el conocimiento de lo que ocurre en el motor fuera de orden ante distintas situaciones. En el Apéndice A se muestran algunas capturas de pantalla y se detalla algo más la aplicación.

2.5 Carga de Trabajo (Benchmarks)

Estamos interesados en programas de código entero, ya que éstos disponen de flujos de direcciones difíciles de predecir. Como carga de trabajo seleccionamos todo el conjunto de programas de prueba de SPECint2K (ver Tabla 2.3).

Tabla 2.3 Programas de prueba junto al conjunto de entradas.

Benchmark	Datos	Benchmark	Datos
bzip2	program-ref	parser	ref
crafty	ref	perl	diffmail-ref
eon	rushmeier-ref	twolf	ref
gcc	166-ref	vortex	one-ref
gzip	program-ref	vpr	route-ref
mcf *	ref		

** NOTA: El grueso de los números presentados en esta Tesis (por ejemplo IPC) no tiene en cuenta el peso del programa mcf. MCF, aún siendo parte de los SPECs, tiene el rendimiento extremadamente limitado por memoria y un IPC muy bajo y constante independientemente del modelo estudiado. Mostraremos los resultados del MCF y la media con él en las gráficas por programa.*

Utilizamos los binarios para Digital Alpha 21264 compilados por Chris Weaver y puestos a disposición pública a través de las páginas web de SimpleScalar.

`http://www.simplescalar.com/benchmarks.html`

`http://www.eecs.umich.edu/~chriswea/benchmarks/spec2000.html`

Los binarios han sido compilados con las máximas optimizaciones y se ha validado su ejecución sobre SimpleScalar cumpliendo los requisitos de SPEC. A continuación se muestran los principales parámetros del entorno de compilación (Figura 2.7). La especificación completa y los flags de compilación por programa pueden ser obtenidos desde la página web.

```
Digital UNIX V4.0F
cc DEC C V5.9-008 on Digital UNIX V4.0 (Rev. 1229)
cxx Compaq C++ V6.2-024 for Digital UNIX V4.0F (Rev. 1229)
f90 Compaq Fortran V5.3-915
f77 Compaq Fortran V5.3-915
Processor 21264
```

Figura 2.7 Entorno de compilación de los programas de prueba (parámetros principales).

Simulamos una ejecución de 100 millones de instrucciones contiguas desde los puntos sugeridos por Sherwood y otros [SPHC02] tras un calentamiento de caches y predictores de 200 Millones de instrucciones. Los *Simpoints* fueron propuestos en 2002, definen los puntos de simulación siguiendo técnicas de análisis de fases de ejecución en programas con Miles de Millones de instrucciones. Para los SPECint2K se basan en los binarios ya mencionados. Al ser utilizados ampliamente por la comunidad permiten una cierta consistencia entre trabajos. Información sobre los Simpoints puede ser obtenida de la URL:

`http://www.cse.ucsd.edu/~calder/simpoint/index.htm`

Binarios y puntos de simulación están disponibles abiertamente. Sin embargo los juegos de datos de entrada a los programas pertenecen a SPEC bajo licencia. Los *Simpoint* definen los puntos de simulación muy avanzada la ejecución del programa [SPHC02]. Los resultados son consistentes en la simulación sobre el sistema operativo Digital Unix (llamadas al sistema).

SimpleScalar permite almacenar en una traza la interacción con el sistema operativo (*trazas eio* - *External Input / Output*). Una *traza eio* es un fichero que aglutina la ejecución de un programa. Por un lado dispone de un volcado del estado de la memoria del programa y de los registros lógicos en un punto concreto de la ejecución (*checkpoint*). Por otro, y a partir de este punto de ejecución, almacena la traza de interación con el sistema operativo. Para cada llamada al sistema operativo que realiza el programa simulado se anota el número en orden de programa y el contador de programa de la instrucción que realiza la llamada al sistema, los valores de los registros lógicos antes de llamar, los valores de los registros después de la llamada y la modificación de la memoria del programa realizada por el S.O. en la llamada.

Con las *trazas eio* se puede comenzar la simulación del programa a partir del *checkpoint* sin tener que re-ejecutar desde el principio cada vez. La simulación se vuelve independiente del sistema operativo al no tener que emular las llamadas al sistema pudiendolas leer del disco. Además, se dispone de un mecanismo de detección de errores en la simulación al poder contrastar los valores de los registros en el sistema original con los valores actuales en cada llamada al sistema.

En el transcurso de esta Tesis se han generado las *trazas eio* de todos los SPEC2K. En el Apéndice B, en la sección b.1 se puede obtener más información sobre la carga de trabajo.

2.6 Plataforma de Simulación

Para la realización de la Tesis se ha necesitado gran potencia de cálculo. Los primeros pasos se realizaron gracias a las facilidades computacionales del Centro Europeo de Paralelismo de Barcelona -CEPBA- sobre dos máquinas Digital Alpha (`Kemet` y `Kerma`). Posteriormente el grueso de la simulación lo realicé en Zaragoza, primero gracias a la instalación de Condor en los laboratorios de prácticas de DIIS. Condor es un sistema de gestión de colas de trabajos y de recursos, corre bajo Microsoft Windows NT y varios Unix. El lector interesado puede buscar más información sobre él en:

`http://www.cs.wisc.edu/condor/`

Conforme el GAZ comenzó a tener recursos financieros se vio la conveniencia de disponer de una plataforma de simulación dedicada. Se llevaron a cabo muchas pruebas sobre distintas configuraciones máquina-sistema operativo-compilador para elegir la mejor opción coste-prestaciones en la ejecución del *benchmark* SimpleScalar (*sim-safe* + *sim-outorder*). La conclusión básica fue que el rendimiento del simulador dependía sobre todo de la latencia *load-uso* (como buen candidato a entrar en SPECint) y agradecía en el *back-to-back* frecuencias de reloj elevadas. Por todo ello la plataforma elegida fue un cluster tipo *Beowolf* con PCs Intel Pentium 4 bajo Linux y el compilador de C++ de Intel para Linux.

Comenzamos con unas pocas máquinas y en la actualidad disponemos de 36 PCs de sobremesa de los que se pueden comprar en cualquier tienda, con configuraciones que van desde los 1,4 a los 3,2Ghz todos ellos con 512 MBytes de *RAM*. Cada equipo dispone de un disco duro *IDE* donde reside un sistema operativo mínimo y una réplica de las *trazas eio* de los programas de prueba para mínimizar el tráfico de red.

CAPÍTULO 3

Predicción de Banco

El camino de datos a memoria seccionado en bancos encamina las peticiones a memoria desde la IQ antes de conocer la dirección de memoria. Para ello necesita un predictor de banco y el rendimiento obtenible depende de los aciertos del predictor.

En este capítulo nos centraremos en el análisis del predictor de banco.

3.1 Introducción

En una organización de cache de primer nivel multibanco, se distribuyen los contenidos entre los bancos. Cuando se produce un fallo de cache los datos son trasladados desde los niveles superiores de la jerarquía y colocados en el banco adecuado de cache de primer nivel. Las instrucciones de acceso a memoria deben encaminarse al banco que almacena el bloque.

Existen varias opciones para enlazar las unidades de cálculo (AGU) con los bancos de cache. Una vez calculada la dirección efectiva de memoria, los *loads* acceden al banco que alojaba los datos o bien pasando por una red de interconexión (*crossbar* en Figura 3.1.a) o bien a través de un planificador de memoria (*LSQ*) que arbitra los accesos a los bancos (Figura 3.1.b).

El camino a memoria seccionado (*Sliced Memory Pipeline*) propuesto por [YMRJ99], distribuye los bancos físicamente cerca de las unidades de cálculo

de dirección (AGU) y de las unidades funcionales consumidoras y con ello minimiza el retardo del camino de datos a memoria (Figura 3.1.c). La cola de inicio de ejecución de instrucciones (IQ) es la encargada de encaminar las peticiones a los bancos antes de calcular la dirección de memoria. Como cada puerto de inicio de ejecución esta acoplado a un sólo banco, necesitaremos un predictor de banco para planificar el inicio de ejecución de las instrucciones de memoria por el puerto adecuado.

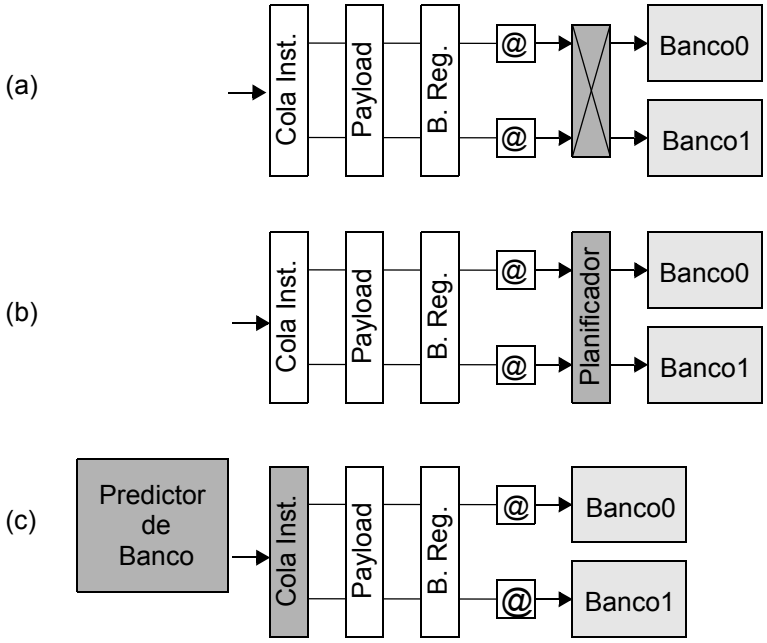

Figura 3.1 Sistemas de encaminamiento de accesos, (a) crossbar, (b) doble planificador y (c) *Slice Memory Pipeline*.

Las interconexiones son muy costosas, y su complejidad aumenta con el número de bancos. Además, como ya vimos en la introducción, el paso por la red de interconexión o el segundo planificador puede incrementar la latencia y/o hacerla variable. El camino a memoria seccionado es más corto y simple. Así pues se reduce la latencia de los *loads* y por tanto se incrementa el rendimiento. Esto es cierto, siempre y cuando el banco sea predicho con mucha precisión.

3.2 Predicción de banco

Como veremos en el Capítulo 5, existen diversas políticas de distribución de contenidos. La política de distribución determina a qué banco o bancos deben ir los datos una vez servido el fallo de cache. La forma de predecir el banco, y con ello el puerto de inicio, dependerá de la gestión de contenidos llevada a cabo.

Las primeras propuestas multibanco distribuyen los contenidos de la cache entre los bancos aplicando una *función de hash* a la dirección de memoria. La memoria se divide en trozos que se reparten entre los bancos.

Según la unidad de reparto, se distinguen dos políticas de distribución de contenidos, entrelazado por palabra y entrelazado por línea. En la primera se toma como unidad de reparto la unidad mayor de acceso (en nuestro caso una palabra son 64 bits). La segunda trocea la memoria en bloques mayores conteniendo varias palabras (32 bytes, o lo que es lo mismo 4 palabras de 8 bytes) y que suele coincidir con el tamaño de línea de la cache.

La Figura 3.2 muestra un esquema de la asignación de los datos a los bancos siguiendo las políticas de distribución mencionadas. Dada una secuencia de palabras de memoria (A0..A7), en un entrelazado por línea, el bloque comprendido por las palabras (A0..A3) es asignado al banco 0 mientras el bloque (A4..A7) lo sería al banco 1. O lo que es lo mismo, las líneas pares son asignadas al banco 0 y las impares al banco 1. En un entrelazado por palabra, las palabras pares se asignan al banco 0 y las impares al banco 1.

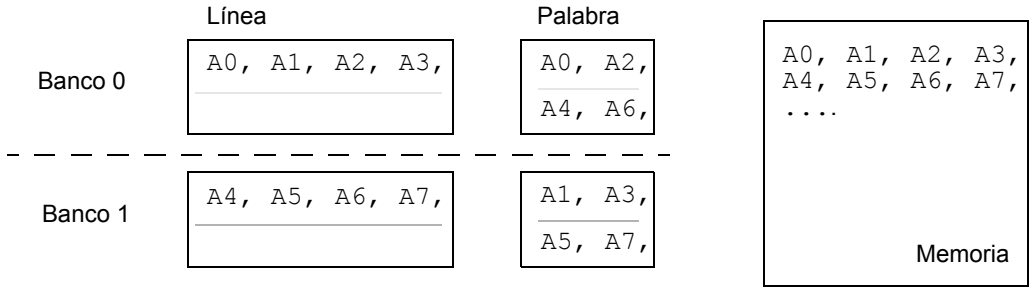

Figura 3.2 Reparto de contenidos, contenido de los bancos.

En modelos con políticas de distribución de contenidos por dirección de memoria, predecir el banco equivale a predecir la dirección efectiva de las referencias a memoria. Más concretamente, se debe predecir un subconjunto de bits de la dirección efectiva.

En un entrelazado por línea y con una configuración de dos bancos se debe predecir el bit 5 de la dirección efectiva. En un sistema con cuatro bancos deberemos predecir los bits 5 y 6. En un entrelazado por palabra (8 bytes) y con dos bancos se predecirá el bit 3, mientras con cuatro bancos necesitaremos los bits 3 y 4. La Figura 3.3 muestra el reparto en función de los bits de la dirección efectiva.

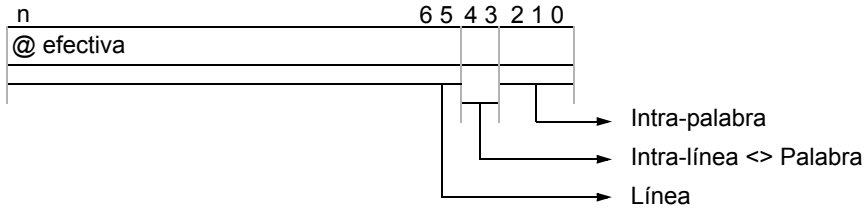

Figura 3.3 Función de *hash* aplicada a la dirección de memoria.

Yoaz y otros [YMRJ99] proponen dos tipos de predictores para predecir banco: predictores de dirección y predictores binarios empleados habitualmente para predecir saltos. En el primer caso predeciremos la dirección y nos quedaremos con los bits necesarios para determinar el banco. En el segundo, necesitaremos un predictor binario por cada bit a predecir. Se han estudiado ambos tipos de predictores pero enseguida nos decantamos por predictores binarios.

Existen multitud de predictores de dirección, pudiéndolos clasificar en dos grandes grupos; predictores de stride y predictores contextuales. El *Differential Finite Context Method* (*DFCM*) es un predictor contextual que soluciona el problema de este tipo de predictores en recorridos stride. El *DFCM* fue propuesto por Goeman y otros [GoVB01] y en precisión supera al *FCM* tradicional entre un 15 y un 33% según sus autores.

Un FCM utiliza la historia de los valores recientes, denominada contexto, para predecir el siguiente valor. Esta estrategia se implementa con dos niveles de tablas. La primera tabla almacena el contexto reciente para cada instrucción. La segunda tabla guarda, para cada posible contexto, el valor siguiente más probable.

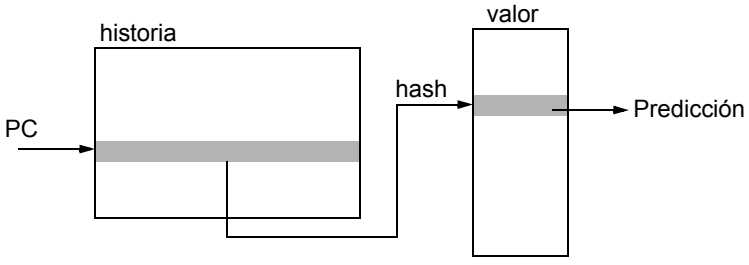

Figura 3.4 *Finite Context Method* (FCM).

Tal y como se muestra en la Figura 3.4, para predecir una instrucción se accede a la primera tabla indexando con el contador de programa (PC) para encontrar la

historia. La historia es empleada para indexar la segunda tabla donde se localiza el valor.

En cuanto a los predictores binarios, podemos clasificarlos en *locales* y *globales*.

Los predictores *locales* requieren dos tablas. Acceden a la primera tabla (información local) para leer la historia y después acceden a la segunda tabla (predicción). Tanto en el *DFCM* como en el *binario local*, en el peor de los casos, nuestro procesador *8-inicio* deberia poder leer 8 historias de dos posibles bloques de búsqueda (dos PCs no consecutivos). Con un puerto de acceso por bloque de búsqueda es suficiente al poder leer en parallelo la historia de las instrucciones consecutivas (varias entradas consecutivas). Sin embargo el acceso a la segunda tabla requiere 8 puertos. Notar que ambas tablas deben ser consultadas y actualizadas en cada ciclo.

Sin embargo, los predictores *globales* utilizan la historia global de las últimas referencias a memoria para acceder a unasóla tabla. En [SFKS02] se muestra como organizar la tabla en bancos con un único puerto sin que se produzcan conflictos de acceso cuando se va a buscar dos bloques secuenciales de instrucciones (caso *8-inicio*). Por ello centraremos los resultados en un predictor binario global.

3.3 Predictor Binario: eGSKEW

Entre los predictores binarios globales, hemos centrado el análisis sobre el predictor global *enhanced Skewed Branch Predictor* (*egskew*) [MiSU97]. El predictor *egskew* es una mejora sobre otros predictores como *gshare* o *gselect*.

Este predictor usa tres tablas. Cada tabla es indexada con una *función de hash* distinta e independiente de las otras. En concreto, todas las *funciones de hash* son calculadas a partir del mismo vector (V). La idea es que al formar los tres índices de forma distinta se minimicen los conflictos. En concreto, si dos vectores V y W tienen un conflicto en una de las tablas, no lo tengan en las otras dos. La acción destructora del conflicto afectará a una tabla, pero la predictabilidad de V no se verá afectada salvo que tenga conflicto con otro vector en otra de las tablas.

Para predecir, se lee cada una de las tres tablas. Una vez consultadas las tres tablas se realiza una votación por mayoría. Al menos dos tablas votarán lo mismo, y ese será el resultado del predictor.

La actualización es parcial, esto quiere decir que sólo se actualizan las tablas que han aportado a la predicción. Si la predicción es errónea se actualizan las tres tablas. Si una tabla realiza una predicción errónea y el resultado final es correcto, no se actualizará. Esta predicción errónea se considera relacionada con otro vector V (conflicto en una de las tablas).

El vector V se forma concatenando el contador de programa $PC = (a_N, .. , a_2)$ con un cierto número (k) de bits provenientes de la historia global.

- $V = (a_N, .. , a_2, h_k, .. , h_1)$

Las funciones de *hash* (f_0, f_1,f_2) se usan para acceder a las tres tablas de 2^n entradas. Descomponemos el vector V en los subvectores (V_3, V_2, V_1), con la condición de que tanto V1 como V2 tengan *n* bits.

Se define la función H como:

- $H: \{0, .. , 2^n -1\} \rightarrow \{0, .. , 2^n -1\}$. $H: (y_n, y_{n-1}, .. , y_1) = (y_n + y_1, y_n , y_{n-1}, .. ,y_2)$

Siendo "+" la or-exclusiva (XOR). Podemos definir las tres funciones de *hash* como:

- $f_0: \{N\} \rightarrow \{0, .. , 2^n -1\}$. $f_0: (a_N , .. , a_2) = (a_{n+2} , .. , a_2)$ -- Bimodal
- $f_1: V \rightarrow \{0, .. , 2^n -1\}$. $f_1: (V_3, V_2, V_1) = H (V_1) + H^{-1} (V_2) + V_1$
- $f_2: V \rightarrow \{0, .. , 2^n -1\}$. $f_2: (V_3, V_2, V_1) = H^{-1} (V_1) + H (V_2) + V_2$

Mas información acerca del predictor y de estas funciones pueden ser consultados en [MiSU97]. La Figura 3.5 muestra un esquema del predictor *egskew* implementado. Nótese que f_o no contiene infomación de historia pasada, sólo PC. La primera tabla es un Bimodal.

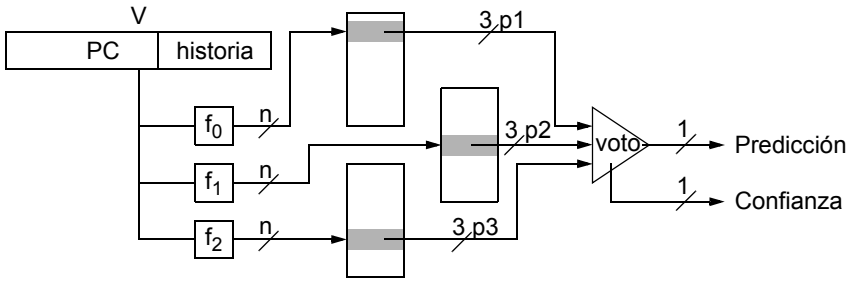

Figura 3.5 *Enhanced Skewed Binary Predictor.*

En el entorno del *camino a memoria seccionado* en dos bancos, Yoaz y otros proponen en [YMRJ99], además de utilizar un predictor de dirección basado en [BJR+99], usar un predictor híbrido compuesto por un predictor *local* más un *gshare* más un *gskew*. El *gskew* lo implementan con tablas de 2^{10} entradas y 17 bits de historia (0,75 Kbytes). No comentan como se realiza la actualización (híbrido), pero la tasa de acierto en SPECint95 no llega al 70%.

En nuestro caso, todas las referencias necesitan una predicción, sin embargo, nos interesa tener una estimación de la confianza de la predicción. Si el predictor nos da baja confianza, podremos llevar a cabo acciones conservadoras; como por ejemplo no despertar a las instrucciones dependientes del *load* hasta que no se haya validado el banco y de esta forma evitar errores de predicción de latencia.

Como estamos interesados en identificar referencias difíciles de predecir, cada tabla tiene un contador saturante de 3 bits por entrada. Los valores del contador varían de 0 a 7. Los valores 0..3 predicen el valor 0, mientras los valores 4..7 predicen el valor 1. Los contadores se alejan del centro lentamente (de uno en uno) y se acercan a él rápidamente (de dos en dos).

Con tablas de 2^{10} el tamaño total del predictor es:

```
3 tablas / bit * 3 bits / entrada * 1024 entradas = 9216 bits
```

Asignaremos confianza a la predicción, sólo si dos de las tres tablas tienen confianza máxima (dos ceros si la predicción es 0 y dos sietes si es 1). Esta decisión tiene que ver con los altos costes de recuperación. El compromiso entre la penalización por recuperación en errores con confianza y los costes de las acciones conservadoras a tomar en predicciones sin confianza se estudian en el Capítulo 4.

Cuando necesitamos predecir dos bits (configuraciones con cuatro bancos) implementaremos dos predictores binarios con historias y tablas propias. Cada uno de ellos votará y será actualizado por separado. Sólo asignaremos confianza a la predicción (de los dos bits conjunta) cuando ambos bits tengan confianza.

Una vez comentado el predictor *egskew* y su implementación, pasamos a comentar los resultados analíticos obtenidos en función del tamaño de tabla (2^n) y de la longitud de historia (k).

3.4 Resultados

Los dos parámetros fundamentales que nos quedan por variar son el tamaño de las tablas y la longitud de historia (una vez fijadas las tres funciones de *hash*, el número de bits por contador, la confianza y la actualización).

Atendiendo al resultado de la predicción (acierto o error) y de la confianza (alta o baja) podemos clasificar las estadísticas en cuatro posibles clases. Sin embargo, como ya veremos en el Capítulo 4, la clasificación la realizaremos en función de las implicaciones en el rendimiento.

La primera clase es el total de referencias dinámicas (*loads* y *stores*) clasificadas como de baja confianza. Este número representa el total de referencias en las que se pueden tomar acciones conservadoras (como por ejemplo desechar la predicción y replicar el *load* a todos los puertos como veremos en el Capítulo 4).

La segunda clase es el total de errores del predictor independientemente de la confianza asignada. Este número nos informa sobre la precisión del predictor y nos permite comparar distintos predictores entre sí.

El tercer resultado son los errores con confianza cometidos por el predictor. En estos errores no podremos tomar acciones conservadoras y por tanto cometeremos un error de especulación de latencia con las consabidas penalizaciones por recuperación.

La Figura 3.6 y la Figura 3.7 muestran los resultados en tanto por uno obtenidos al barrer el tamaño de cada tabla entre 2^{10} y 2^{17} entradas y la historia desde 4 bits hasta el *tamaño de tabla + 2* para la media del conjunto completo de SPECint2K. La gráfica superior muestra el total de referencias clasificadas como de baja confianza (*T_sin_conf*), en el centro se presenta el total de referencias con errores de predicción (*T_error*), la gráfica inferior pone de manifiesto los errores con confianza (*Error_conf*). Al estar interesados tanto en entrelazado por línea como por palabra y en configuraciones con dos y con cuatro bancos, mostraremos las estadísticas para los bits 3, 4, 5 y 6.

T_sin_conf se sitúa en torno al 12% de las referencias, *T_error* alrededor del 7% y los errores con confianza (*Error_conf*) poco más o menos en el 1,5%. Como ya hemos dejado intuir, este último factor es el que sobre todo nos interesa minimizar para reducir el número de errores de especulación de latencia, aún a costa de incrementar *T_sin_conf*.

A partir de un determinado tamaño de tabla (2^{15} .. 2^{17}) las métricas se saturan, y no se obtienen mejoras al incrementar el tamaño. Para cada tamaño de tabla hay una historia ideal, existe un punto de inflexión a partir del cuál la métrica empeora. Al incrementar la longitud de historia se incrementa el número de conflictos en las tablas.

En el análisis inicial nos centraremos en el bit 5 (*entrelazado por línea*) para a continuación comentar las diferencias observadas con los otros bits.

Con tamaños de tablas pequeñas (2^{10} y 2^{11}), la longitud de historia óptima se sitúa en torno a 6 bits. Al ir aumentando el número de entradas de las tablas, esta historia ideal se incrementa hasta 16 bits con tablas de 2^{17} entradas. Podemos concluir que con tablas pequeñas al incrementar la historia se producen demasiados conflictos y bajan todas las métricas. Para tamaños medios de tablas (por ejemplo 2^{13} entradas), la longitud ideal de historia se sitúa entorno a 9 bits.

Error_conf satura con tablas desde 2^{13} a 2^{15} e historias entre 9 y 16 bits. Las otras dos métricas siguen bajando aunque muy lentamente.

En un primer cotejo entre los diferentes bits a predecir, observamos que el bit 3 goza de mejores métricas que el resto de bits y que el peor es el bit 6. Los bits 4 y 5 se sitúan próximos uno del otro con el bit 4 ligeramente mejor que el 5.

Da la impresión que conforme intentamos predecir bits más significativos disminuye la cantidad de historia necesaria o bien se incrementan los conflictos. Hay que recordar que la historia se actualiza con el bit que se está prediciendo.

Los resultados para el bit 3 saturan con tablas de 2^{12} entradas e historias desde 9 bits. Con tablas de 2^{11} entradas, el bit 3 consigue mejor resultado que el bit 5 con tablas de 2^{17} entradas.

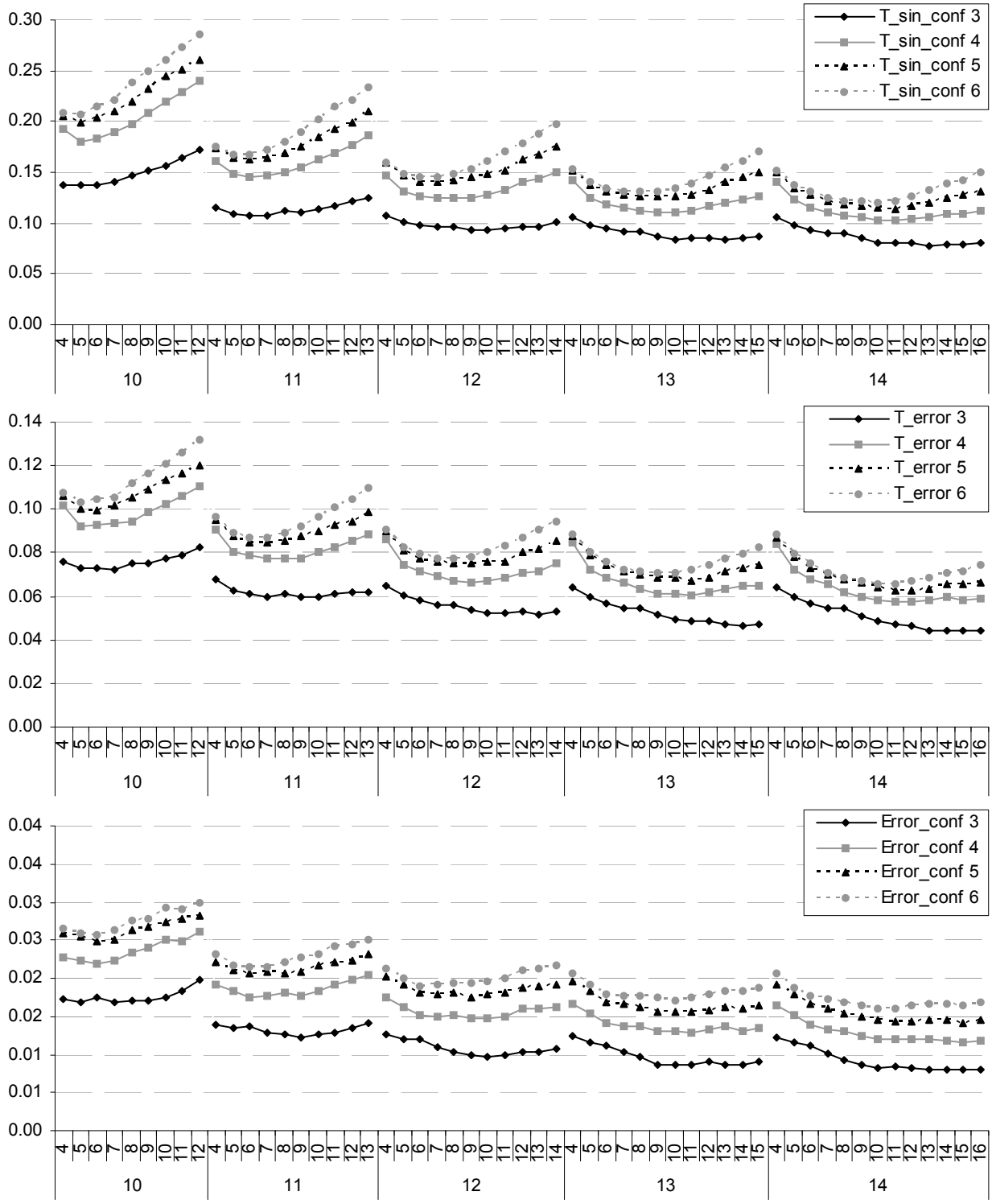

Figura 3.6 Resultados predictor de banco para la media aritmética de SPECint2K (`Mint`) con tablas de 2^{10} a 2^{14}. Arriba: total de referencias sin confianza (*T_sin_conf*) para los bits 3, 4, 5 y 6. Centro: total de errores (*T_error*). Abajo: error con confianza

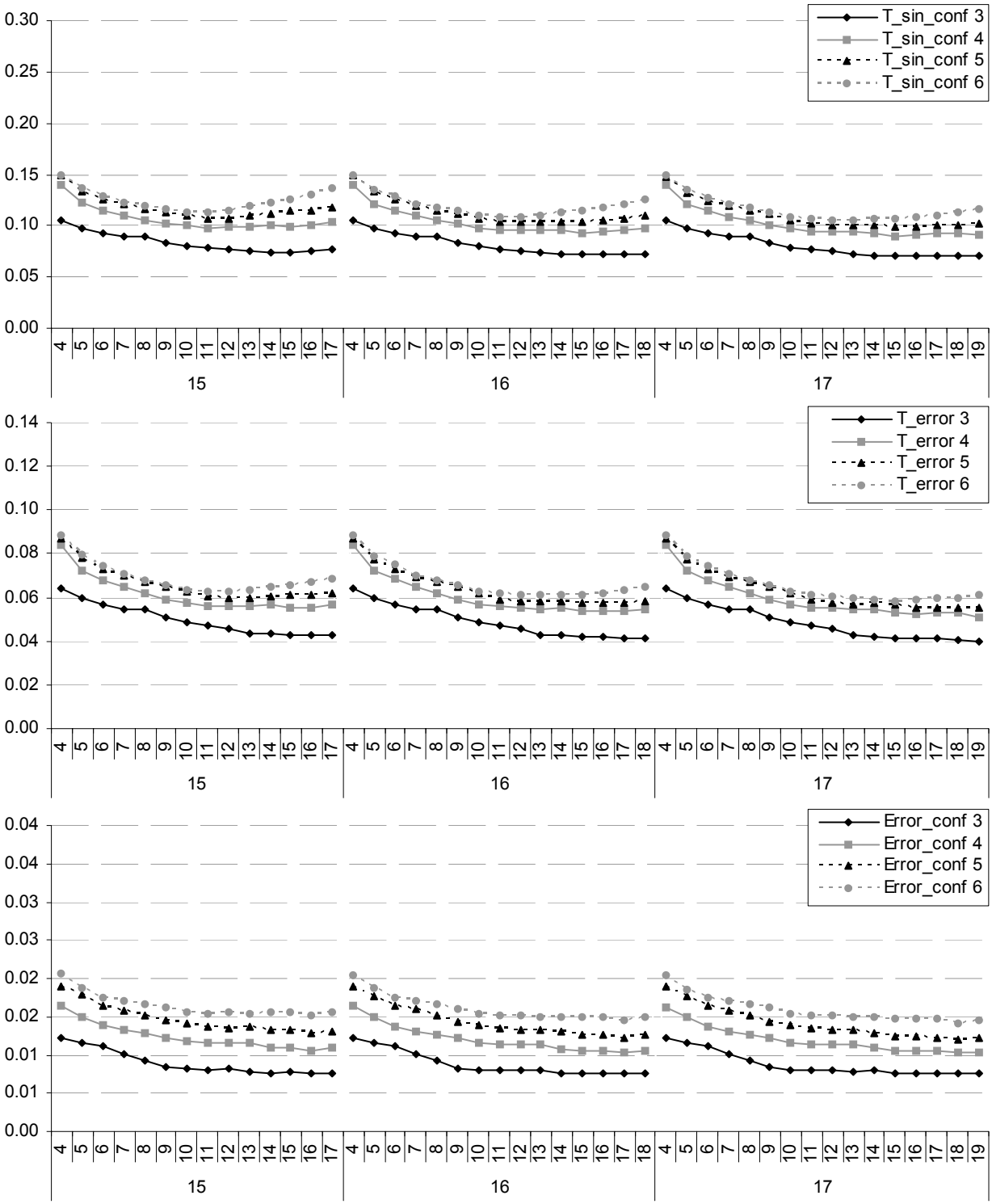

Figura 3.7 Resultados predictor de banco `Mint` (tablas desde 2^{15} a 2^{17}).

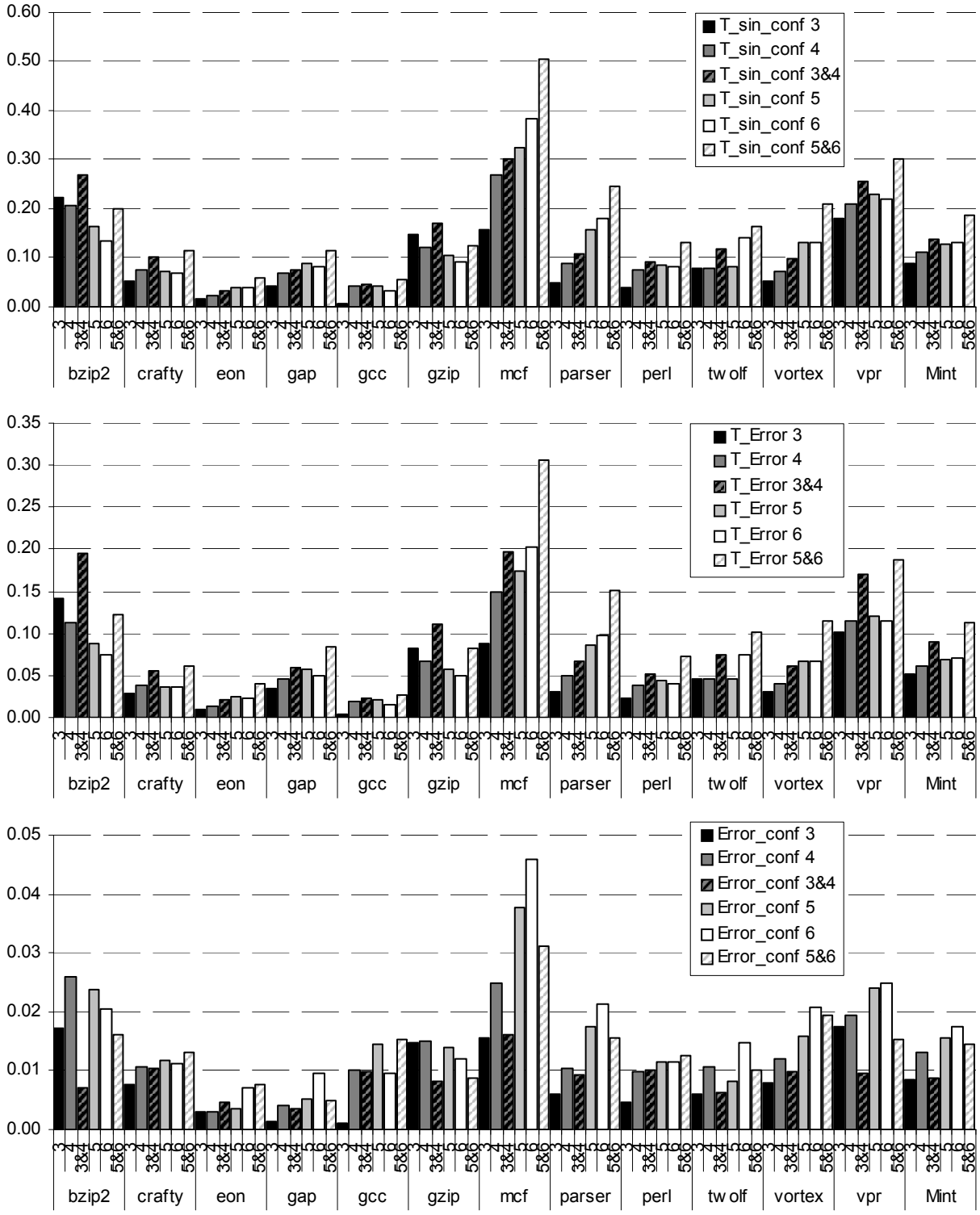

Figura 3.8 Resultados predictor de banco *egskew* con tablas de 2^{13} y 9 bits de historia, (9 Kbytes por bit a predecir). Para 2 bancos bits 3 y 5, para 4 bancos bits 3&4 y 5&6.

Para observar el comportamiento por programa, la Figura 3.8 muestra las tres métricas para tablas de 2^{13} entradas y 9 bits de historia (9 Kbytes por bit a predecir). Mostramos los resultados de predecir los bits 3, 4, 5, 6 individualmente más el resultado de predecir los bits 3&4 y los bits 5&6 necesarios para discernir entre cuatro bancos.

Al predecir 2 bits disminuye el número de errores con confianza (*Error_conf* en la gráfica inferior). Ello es debido a que una predicción sólo se considera con confianza si las predicciones de ambos bits la tienen (regla muy restrictiva). Esto hace que un mayor número de peticiones sean clasificadas como de baja confianza (*T_sin_conf* en la gráfica superior). En la gráfica del centro podemos observar el total de errores (*T_error*), es lógico que se acumulen los errores de un bit y del otro. El total sube, sin embargo podemos constatar que hay un subconjunto importante de referencias en las que falla la predicción de ambos bits simultáneamente.

Atendiendo al comportamiento por programa, podemos constatar que existen programas como `BZIP2`, `MCF` y `VPR` con tasas mucho más altas que la media. Por contra, otros como el `EON` y el `GAP` se sitúan claramente por debajo de la media.

En el Apéndice B se muestran las métricas al variar el tamaño de tabla y la historia individualmente por programa. El comportamiento es bastante heterogéneo. Se puede observar que los programas saturan a distintos tamaños de tabla y con historias diferentes.

3.5 Conclusiones

Los predictores binarios *globales* son una buena opción de diseño ya que soportan mejor que otros el aumento en el número de predicciones por ciclo. Usamos el *egskew* porque además minimiza los conflictos.

Añadimos a este predictor confianza para poder tomar medidas conservadoras con las predicciones sin confianza. De esta forma, sólo los errores con confianza sufrirán la penalización por fallo de predicción de banco. Como veremos en los capítulos siguientes, incluso un reducido número de recuperaciones nos puede hacer perder mucho rendimiento.

Con tablas de 2^{13} entradas y el tamaño de historia apropiado se consiguen resultados muy cercanos al “óptimo”.

Los bits más bajos de la dirección son más predecibles que los altos. Por tanto, atendiendo a la predictibilidad, en multibanco sería mejor entrelazar los datos por palabra que entrelazarlos por línea.

Los primeros resultados de la Tesis (Capítulo 4) fueron obtenidos con tres tablas de 2^{10} entradas por tabla, 6 bits de historia, y con contadores saturantes de tres bits por entrada y tabla, necesitando una capacidad total de 1,1 Kbytes por bit a predecir. Este tamaño era representativo de los predictores que se estaban utilizando en esa epoca. Con los avances en la escala de integración, los predictores se han ido haciendo más grandes. Los dos últimos capítulos (Capítulo 5 y Capítulo 6) utilizan un predictor de banco de 2^{13} entradas por tabla (9 Kbytes por bit a predecir). La longitud de la historia en este caso es de 9 bits.

CAPÍTULO 4

Contrarrestando la Penalización por Error de Predicción de Banco

Los procesadores futuros con caminos de datos al primer nivel de memoria cache seccionada en bancos dependerán de la predicción de banco para planificar la ejecución de las instrucciones de memoria. En procesadores super-segmentados, incluso un pequeño número de errores de banco puede perjudicar enormemente al rendimiento.

Este capítulo pone de manifiesto las pérdidas de rendimiento debidas a la penalización en caso de error de predicción de banco. El primer objetivo es reducir la penalización por error de especulación. El segundo objetivo es reducir el número de errores.

4.1 Introducción

Como ya se comentó en la introducción, el camino de datos a memoria seccionado reduce la latencia de los esquemas multibanco al eliminar el componente de interconexión [YMRJ99]. La cola de inicio de ejecución de instrucciones (IQ) sólo dispone de un puerto a cada banco de cache y debe planificar por qué camino enviar la petición de acceso a memoria basándose en un predictor (ya que la dirección de acceso todavía no ha sido calculada (Figura 4.1).

Se establece un lazo hardware (*loose loop* según [BTME02]) entre la IQ y la etapa de cálculo de dirección responsable de comprobar la predicción de banco

realizada. La IQ utiliza la predicción de banco para iniciar la ejecucción de las instrucciones de acceso a memoria, y en el caso de los *loads*, planifica de forma especulativa la ejecución de las instrucciones dependientes.

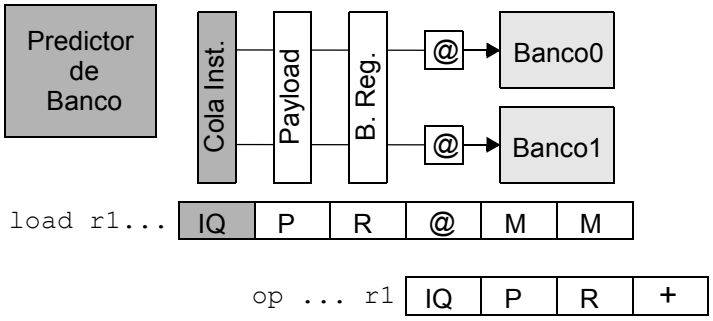

Figura 4.1 Camino de datos y diagrama de tiempo del camino a memoria seccionado.

Cuando la predicción es correcta, el rendimiento aumenta respecto a un modelo donde se espere a conocer si la predicción fue correcta para despertar a las instrucciones dependientes. Sin embargo, con cada error de predicción perderemos todo el trabajo realizado especulativamente y necesitaremos recuperar el estado del procesador.

En este capítulo pondremos de manifiesto las pérdidas de rendimiento debidas a la penalización de errores de predicción de banco. Clasificaremos la penalización entre incremento de latencia *load-uso*, trabajo perdido y penalización de recuperación (sección 4.3). Agrupamos estas dos últimas bajo el término penalización de error de especulación. Una vez clasificadas y cuantificadas las pérdidas propondremos un mecanismo de recuperación alternativo que minimiza las pérdidas debidas a la penalización de recuperación, válido para todas las recuperaciones en error de especulación de latencia (sección 4.4). Para reducir el incremento de latencia *load-uso*, en caso de error de predicción, propondremos políticas de replicación del inicio de ejecución de los *loads* a múltiples puertos utilizando la confianza del predictor (sección 4.5). Finalizaremos con las conclusiones en la sección 4.6.

4.2 Procesador Base y Sistema de memoria

Realizaremos los experimentos sobre los dos procesadores (*4-inicio* y *8-inicio*) presentados en el Capítulo 2 con la jerarquía de memoria comentada. Los bancos del primer nivel de cache son de 8 Kbyte.

El STB / LDB está distribuido siguiendo las reglas de [ZyKo01], cada STB / LDB puede albergar el total de *stores*/*loads* en vuelo, pero con un único puerto de acceso. Tanto los *loads* como los *stores* deben ser emitidos a la IQ acompañados de la predicción de puerto de inicio a memoria de la IQ por el que deben iniciar la ejecución. En caso de error de predicción de banco la IQ es notificada en el ciclo siguiente al cálculo de dirección, tal y como se comentó en el Capítulo 2 y se muestra en la Figura 4.2.

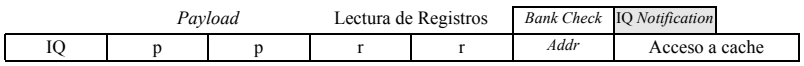

Figura 4.2 Retardo de la comprobación de banco.

Si la tendencia de incremento de las frecuencias de reloj se mantiene, el número de ciclos entre el inicio de la ejecución (IQ) y la ejecución se irá incrementando. Estamos interesados en analizar el efecto de los errores de predicción de banco sobre el rendimiento, como la longitud del lazo (*loose loop*) depende del número de ciclos destinados a leer el *payload* y acceder al banco de registros, este número lo tomaremos como parámetro en nuestras simulaciones variándolo entre 1 y 4 ciclos.

El rendimiento se expresa en media armónica de IPC sin contar el programa `MCF` salvo que sea declarado explícitamente.

4.2.1 Predictor de banco

Asumimos que los bancos están entrelazados por dirección de línea de cache. Por ello el objetivo es predecir los bits menos significativos de la dirección de la línea. Todas las instrucciones de memoria necesitan una predicción. Predecimos por separado cada bit de dirección; un bit para dos bancos (*4-inicio*), 2 bits para 4 bancos (*8-inicio*).

Como predictor de banco elegimos el *enhanded skewed binary predictor* presentado en el capítulo anterior. Cada predictor de bit tiene la misma dimensión: 1K entradas en cada una de las tres tablas con una historia de 6 bits. Como estamos interesados en reconocer *loads* difíciles de predecir, utilizamos contadores de 3 bits saturantes por entrada y tabla.

La Tabla 4.1 muestra los porcentajes medios de acierto y error alcanzados por el predictor de banco tanto para 2 bancos como para 4 bancos. Los resultados se han separado entre el total de referencias, *loads* y *stores*. Cada ejecución

individual ha sido clasificada según su resultado. Como cabía esperar la precisión decrece conforme aumenta el número de bancos, simplemente por el hecho de que aleatoriamente es más fácil acertar uno entre 2 que entre 4.

Tabla 4.1 Precisión del predictor para 2 bancos (*4-inicio*) y 4 bancos (*8-inicio*) entrelazados por línea.

Gskew 1Ke * 3bits	Referencias		Loads		Stores	
historia 6	acierto	error	acierto	error	acierto	error
2 bancos	89,76	10,26	87,93	12,07	93,34	6,66
4 bancos	82,90	17,10	80,06	19,94	88,46	11,54

La Figura 4.3 muestra la precisión del predictor de banco por programa. Se puede observar que MCF representa un caso aparte, su tasa de errores es mucho más alta que la del resto. Destaca GCC con una tasa de acierto del 93% (92%) para 2 bancos (4) mientras VPR se queda en el 82% (72%).

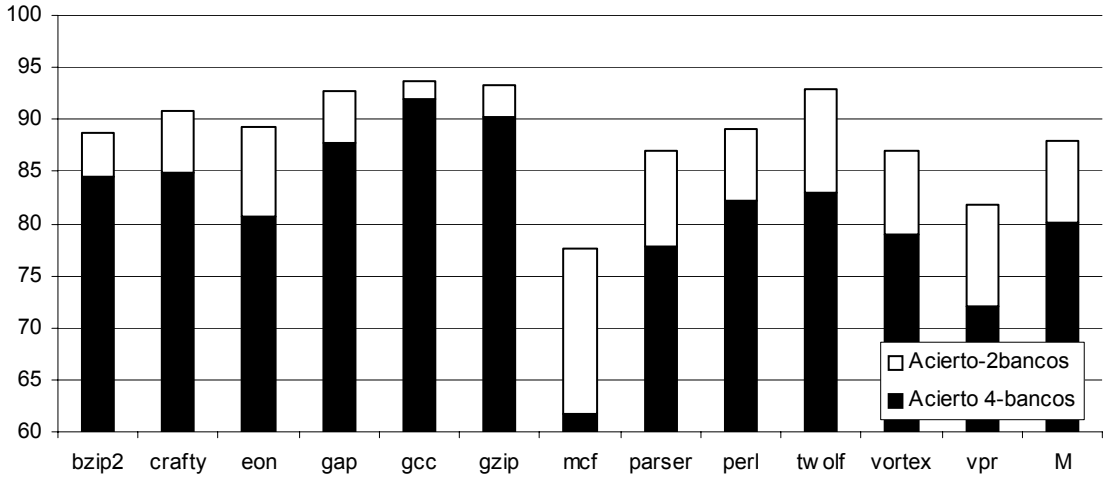

Figura 4.3 Precisión del predictor de banco por programa para 2 bancos (blanco) y 4 bancos (negro) para las instrucciones *Load*.

4.3 Repercusión de los errores de banco en el rendimiento

Para estudiar la repercusión en el rendimiento de los errores de banco, suponemos predicción de latencia y un mecanismo de recuperación BASE en errores de predicción descrito por Morancho y otros [MoLO01].

La Figura 4.4.a muestra el diagrama de tiempo de la ejecución de un *load* bien predicho y dos instrucciones dependientes (add y sub). La instrucción *load* se inicia en el ciclo 1. Modelamos 4 ciclos para leer la información de la instrucción (*p*) y acceder al banco de registros (*r*). Durante el 5º ciclo la instrucción obtendrá sus operandos o bien del banco de registros o bien a través de los cortocircuitos y podrá comenzar la ejecución. En el 6º ciclo se calcula la dirección efectiva de memoria y se comprueba la predicción de banco. La cache es accedida en el 7º ciclo, necesitando un ciclo más para distribuir el dato por los cortocircuitos. La IQ será notificada sobre la ejecución correcta del *load*. El retardo *load-uso* es de 2 ciclos, eso quiere decir que las instrucciones dependientes encontrarán el dato en el cortocircuito si son iniciadas de forma especulativa tres ciclos después de haber iniciado al *load*. Las instrucciones dependientes se pueden iniciar a partir del 4º ciclo, cuando todavía no se conoce ni si el banco predicho es correcto ni si el *load* acertará en cache (*ventana especulativa*).

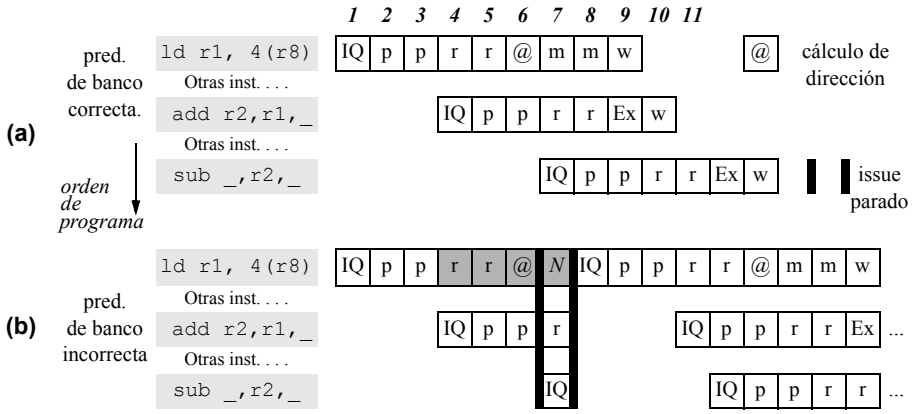

Figura 4.4 Diagrama de tiempo para predicción de banco correcta y errónea, asumiendo 4 ciclos entre IQ y EX. Por claridad mostramos el inicio de una única instrucción por ciclo.

La Figura 4.4.b muestra el mismo diagrama en caso de error de banco. En el 7º ciclo, en vez de acceder a cache, la comprobación de banco es notificada a la IQ. La recuperación tiene lugar durante ese ciclo (marcado con una *N*). La recuperación implica parar el inicio de instrucciones durante ese ciclo, ya que los resultados del *load* y el *conjunto de dependientes* deben ser marcados en la

matriz de dependencias como "no disponibles". Durante ese mismo ciclo, el *load* y el *conjunto de dependientes* pasan a ser visibles de nuevo a la lógica de selección y serán re-ejecutados de nuevo conforme sus operandos vuelvan a estar disponibles (ciclos 8º, 11º y 12º para el *load*, *add* y *sub* respectivamente).

Al comparar la temporización de ambos casos (acierto y error de predicción de banco), aparecen las siguientes diferencias; la latencia *load-uso* se ha visto incrementada en 7 ciclos (ciclo 4 al 11), el trabajo iniciado por el *conjunto de dependientes* durante la *ventana especulativa* ha sido descartado, y finalmente, la recuperación ha inhibido el inicio de instrucciones durante todo un ciclo.

Resumiendo, la penalización por error de predicción de banco es debida a varios factores; incremento de la latencia *load-uso*, trabajo perdido y penalización de recuperación. Agrupamos estos dos últimos factores bajo el término penalización por error de especulación. Destacar que tanto el incremento de latencia *load-uso*, como el trabajo perdido dependen del número de ciclos entre IQ y EX (ciclos 2:7).

Para cuantificar la pérdida, medimos el rendimiento (media armónica de IPC) en las dos configuraciones de procesador (*4* y *8-inicio*) y variando el número de etapas entre IQ y EX desde 1 a 4. La Figura 4.5 muestra los resultados obtenidos. Primero nos centraremos en las barras denominadas BASE y OBP (*Oracle Bank Predictor*).

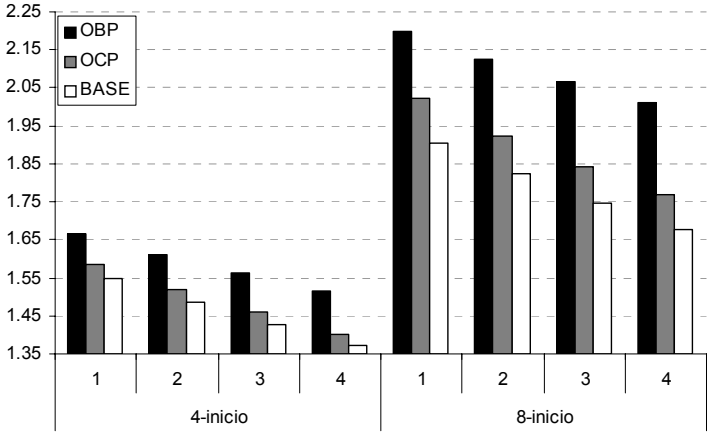

Figura 4.5 Media armónica de IPC para los procesadores *4-inicio* y *8-inicio* variando el número de etapas IQ:EX; con predictor de banco Oráculo (OBP), predictor con confianza perfecta (OCP) y el predictor real (BASE).

El procesador BASE implementa el predictor real introducido en la sección anterior. OBP dispone de un oráculo que siempre predice el banco correctamente. Así que la diferencia en rendimiento entre OBP y BASE es debida a los errores de predicción de banco del modelo BASE. El rendimiento de todos los modelos empeora al incrementar el número de etapas entre IQ y EX, ello es debido a que también se incrementa la penalización por error de salto (se nota claramente en OBP).

El procesador BASE con *4-inicio* pierde con respecto a OBP desde un 6,9 a un 9,3% en IPC al ir incrementando las etapas IQ:EX. El *8-inicio* incrementa la pérdida desde 13,3 a un significante 16,6%.

Otro punto interesante es separar la penalización por error de predicción de banco entre incremento de latencia *load-uso* y penalización por error de especulación. Para ello simulamos un nuevo modelo donde hacemos uso del predictor real pero añadimos un oráculo para saber si la predicción es correcta o no. En caso de predicción incorrecta las instrucciones dependientes no son despertadas (OCP, *Oracle Confidence Predictor*). Al no despertar eliminamos la penalización por error de especulación. La diferencia OBP y OCP es debida al incremento de latencia *load-uso* de los *loads* mal predichos, mientras la diferencia OCP-BASE es atribuible a la penalización por error de especulación.

Los resultados muestran que ambos factores son importantes. Sin embargo, el incremento de la latencia *load-uso* pesa más conforme aumenta el número de etapas IQ:EX; en concreto, su contribución en la penalización pasa de dos tercios a tres cuartos al pasar de 1 a 4 etapas entre IQ y EX. Esto se cumple en ambas configuraciones de procesador.

La Figura 4.6 y la Figura 4.7 muestran el IPC por programa y la media armónica teniéndo en cuenta al programa `MCF` (MA+mcf) y la ya presentada sin tenerlo en cuenta (MA). La Figura 4.6 representa al procesador *4-inicio* mientras la Figura 4.7 muestra los resultados del procesador *8-inicio* presentados uno y otro en el Capítulo 2. En ambas figuras se muestran dos gráficas separadas en función del número de etapas IQ:EX y correspondientes a una etapa en la gráfica superior (a) y a cuatro etapas en la gráfica inferior (b).

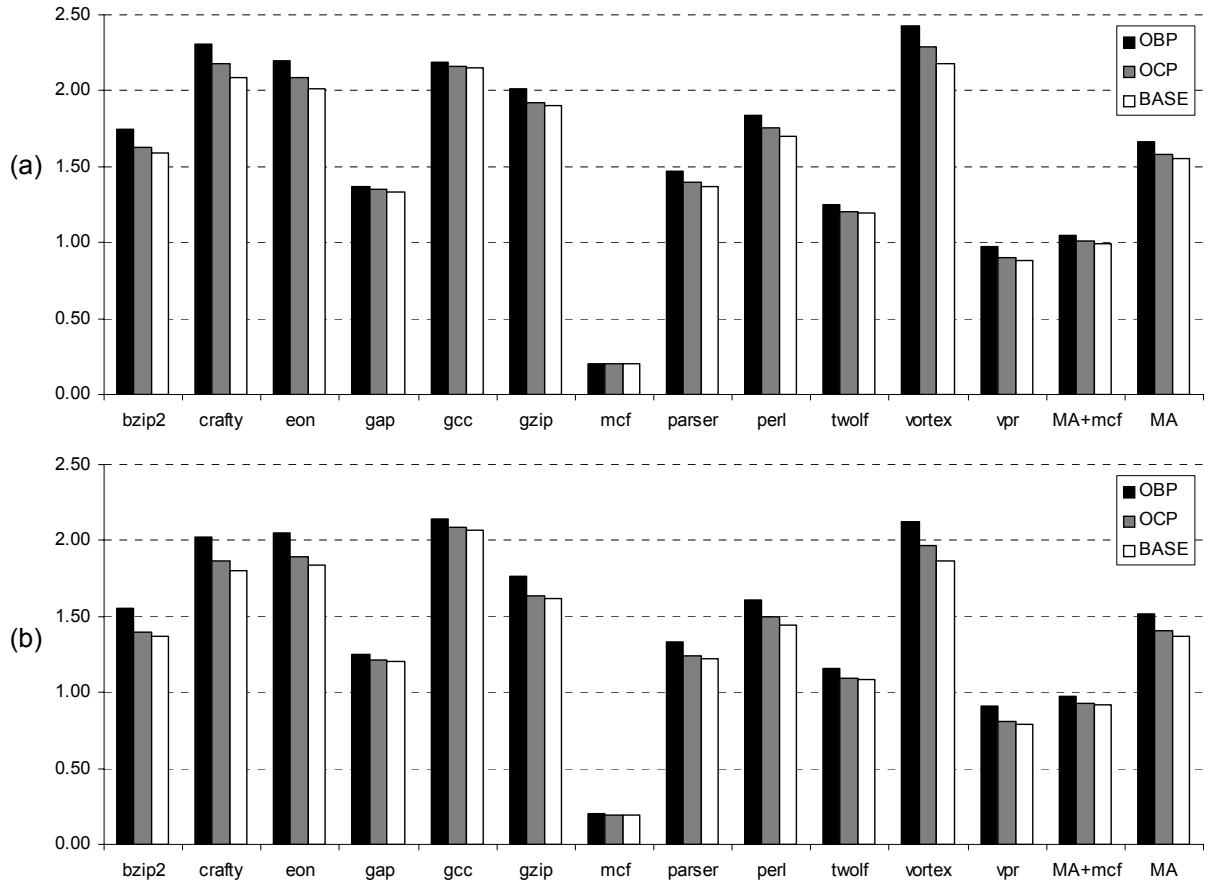

Figura 4.6 IPC para el procesador *4-inicio* y una etapa (a) o cuatro ciclos (b) entre IQ y EX; con predictor de banco Oráculo (OBP), predictor con confianza perfecta (OCP) y el predictor real (BASE).

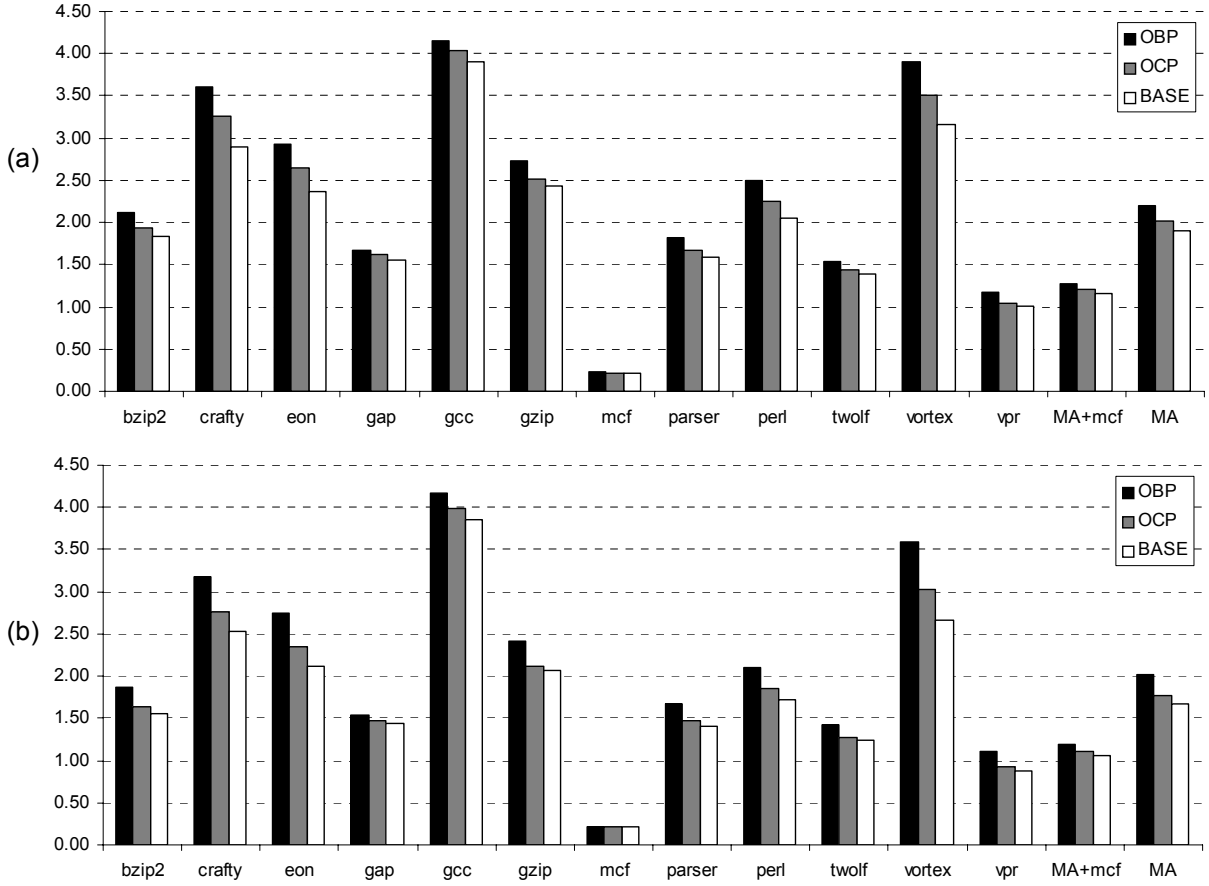

Figura 4.7 IPC para el procesador *8-inicio* y una etapa (a) o cuatro etapas (b) entre IQ y EX; con predictor de banco Oráculo (OBP), predictor con confianza perfecta (OCP) y el predictor real (BASE).

Como se puede observar, las tendencias entre programas tienen pequeñas variaciones aunque todos siguen la tónica general tanto en *4-inicio* como *8-inicio*. El `MCF`, que tenía la peor precisión prácticamente no ve alterado su rendimiento al estar muy limitado por otras razones. Programas como `GAP` y `GCC` apenas notan las diferencias ya que tenían buena predictabilidad de banco. Algunos notan más el incremento de latencia *load-uso*, mientras otros responden mal a la penalización por error de especulación.

A continuación presentaremos la propuesta para reducir la penalización en error de especulación, para luego centrarnos en la reducción de la latencia *load-uso*.

4.4 Contrarrestando la penalización por error de especulación

La Figura 4.8 muestra el diagrama de tiempo en caso de error de predicción de latencia por fallo de cache L1.

La IQ contiene una matriz de dependencias y una lógica de selección. La matriz de dependencias aglutina la información de dependencias entre instrucciones y permite determinar en cada ciclo que instrucciones están listas al tener disponibles todos sus registros fuente (fase de *Wake-upen* Figura 4.8). La lógica de selección inicia la ejecución de las instrucciones más viejas entre las ya listas (fase de *Selección* Figura 4.8). Estas dos fases se llevan a cabo dentro del mismo ciclo.

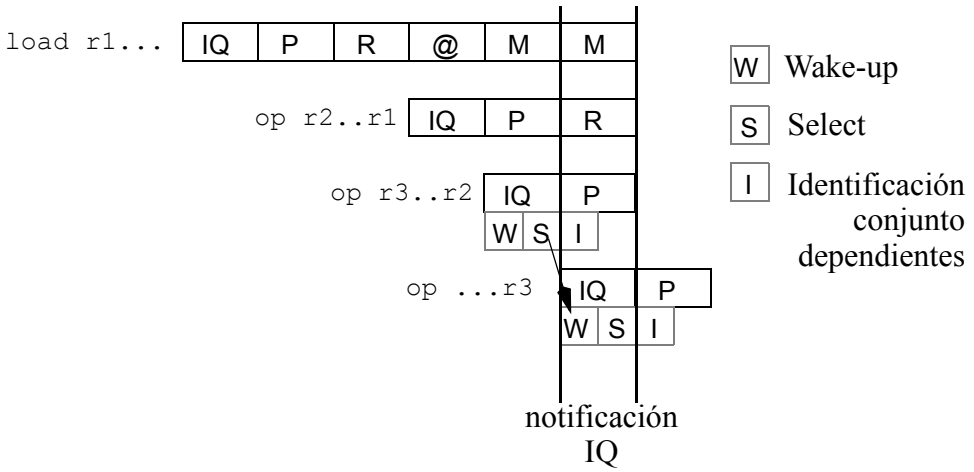

Figura 4.8 Lazo despertar-seleccionar-identificar *conjunto de dependientes*.

Las instrucciones seleccionadas marcan la disponibilidad de su destino para entrar en la nueva fase de despertar de las instrucciones dependientes (*tight loop,* lazo a distancia 1).

Las instrucciones pertenecientes al *conjunto de dependientes* se identifican tras la selección accediendo a un conjunto de vectores de bits [MoLO01] (marcado con una “I” en la Figura 4.8). Existen dos vectores de bits por instrucción con predicción de latencia (*loads*); el primero identifica y aglutina a las instrucciones pertenecientes al *conjunto de dependientes*, el segundo marca la disponibilidad de los resultados de estas instrucciones como especulativos. Resulta imposible discernir, en la fase de selección, aquellas instrucciones pertenecientes al *conjunto de dependientes* de las que no lo son. Nótese que el

número total de vectores de bits depende de la longitud de la *ventana especulativa* y del número de predicciones de latencia que se puedan hacer por ciclo.

En cada ciclo las instrucciones seleccionadas (en el ciclo anterior) consultan en el segundo vector si sus operandos fuentes son especulativos. Si no lo son, las instrucciones pueden abandonar la IQ. Si alguna de ellas lo es, marcarán su destino como especulativo en el vector correspondiente y se añadirán al *conjunto de dependientes* (primer vector) de ese *load*. Así mismo deberán permanecer en la IQ (aunque invisibles a la lógica de selección).

Si todas las predicciones acaban siendo correctas, el vector del *conjunto de dependientes* nos permitirá extraer de la IQ a todas las instrucciones dependientes iniciadas especulativamente. Si nos llega la notificación de un error de predicción de latencia (error de predicción de banco o fallo en cache), se marcarán con el segundo vector los operandos como no disponibles y el vector del *conjunto de dependientes* nos permite volve a hacer visibles a la lógica de despertar-seleccionar a las instrucciones iniciadas especulativamente.

Cuando llegue la notificación a la IQ, se estarán despertando y/o seleccionando instruciones que no sabemos si pertenecen o no al *conjunto de dependientes*. El mecanismo base propuesto por Morancho y otros [MoLO01] corta la cadena de despertar bloqueando el inicio de todas las instrucciones seleccionadas durante el ciclo en el que se está recuperando la matriz.

4.4.1 Recuperación en Cadena

La ***Recuperación en Cadena*** (*Chained Recovery*: CR) reduce la penalización de recuperación al no tener que parar el inicio de instrucciones durante todo un ciclo tal y como hace el mecanismo base de recuperación. En su lugar, después de detectar un error de predicción de latencia se sigue iniciando instrucciones. Entre ellas puede iniciarse la ejecución de instrucciones dependientes de la instrucción mal predicha. Estas instrucciones dependientes se detectan en el ciclo siguiente al de inicio y se anula su ejecución en el ciclo siguiente a la detección.

El proceso de recuperación finaliza cuando en un ciclo no se inicia ninguna instrucción dependiente de una instrucción mal predicha. En estas condiciones la duración del proceso de recuperación de un error de predicción de latencia no está acotada de antemano.

La diferencia básica entre el mecanismo de recuperación base y el mecanismo CR es que en el primero se pierde siempre un número de slots de inicio igual al ancho de inicio, mientras que en el segundo el número de slots de inicio que se pierden depende de la cadena de instrucciones dependientes. Sin embargo, en el mecanismo CR el inicio de ejecución de instrucciones más viejas que la instrucción mal predicha no se ve retardado por el error de predicción, ya que pueden iniciar su ejecución mientras se efectúa la recuperación.

Aplicamos CR a todos los errores de predicción de latencia, pero centrando el análisis en la recuperación por errores de predicción de banco.

La Figura 4.9 muestra como CR actúa en el código de la Figura 4.4. En el primer ciclo de la recuperación (ciclo 7) los resultados del *load* y del *conjunto de dependientes* son marcados como no disponibles (al igual que en la recuperación base). Sin embargo, en ese mismo ciclo la lógica de selección escoge una instrucción (*sub*) que depende de otra que está siendo anulada (*add*). Durante el ciclo 8º, CR identifica qué instrucciones iniciadas en el ciclo anterior dependen del *load* mal predicho. En el ciclo 9º, el resultado de las instrucciones identificadas por CR en el ciclo 8º son marcados como no disponibles.

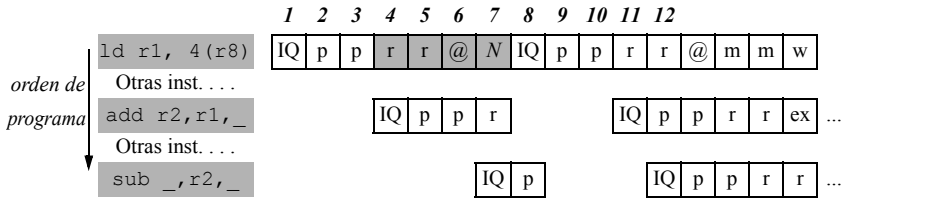

Figura 4.9 *Recuperación en Cadena* (CR) por error de especulación de banco.

De ahora en adelante, en cada ciclo (ciclos 9, 10, 11, …) las dos acciones mencionadas se siguen haciendo concurrentemente: a) los resultados de las instrucciones identificadas como dependientes se marcan como no disponibles, y b) se identifica a las instrucciones dependientes iniciadas en el ciclo anterior. El proceso siempre está en marcha. Dado un determinado *load*, su recuperación termina cuando en un ciclo no se inicia ninguna instrucción dependiente (porque se acaba la cadena de dependencias o porque se iniciaron instrucciones dependientes cuya latencia es mayor que un ciclo).

Para implementar CR se necesita únicamente añadir un nuevo vector de bits al mecanismo de recuperación base. Cada instrucción chequea en el ciclo siguiente a su inicio si alguno de sus operandos fuentes está marcado en ese vector como anulado. Si es así, marcará su destino como anulado para el ciclo

siguiente y se anulará la instrucción. Si ninguno de sus operandos fuentes está marcado, la instrucción progresará en el segmentado como hasta ahora.

4.4.2 Evaluación

La Figura 4.10 muestra los resultados de nuevo para los modelos OCP y BASE considerados en la sección anterior. Añadimos dos barras centrales donde se muestra el rendimiento obtenido por un procesador mejorado con CR. La barra OCP-CR muestra un procesador sin errores de especulación de banco, al no despertar a las dependientes en caso de error de predicción (OCP), que se recupera de los errores de predicción de latencia por fallo en cache con CR. Por ello existe una pequeña diferencia entre OCP y OCP-CR.

Los modelos OCP y CR tienen aproximadamente el mismo IPC, sin embargo no se pueden comparar. El modelo OCP sólo tiene fallos de cache y se recupera bloqueando el inicio de ejecución durante un ciclo. El modelo CR tiene tanto errores de predicción de banco como fallos de cache. El modelo CR utiliza la *recuperación en cadena* mientras que el modelo BASE pierde un ciclo en todos los errores de predicción de latencia.

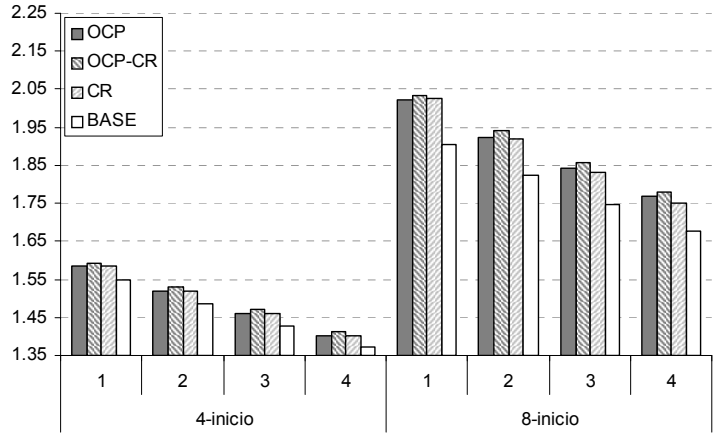

Figura 4.10 Media armónica de IPC para los procesadores de referencia *4-inicio* y *8-inicio* al aplicar la *recuperación en cadena* (*Chained Recovery*).

Como esperábamos, la pérdida de rendimiento por penalizaciones en error de especulación (diferencia entre OCP-CR y CR) es muy pequeña. Esta diferencia es debida tanto al trabajo perdido en la *ventana especulativa* como a la pérdida de slots de inicio de ejecución malgastados tras el ciclo de notificación por las instrucciones que se van escapando ciclo a ciclo.

El hecho de que la diferencia entre OCP-CR y CR sea tan pequeña nos lleva a pensar que los slots de inicio malgastados en la recuperación no impiden el inicio de una cantidad significativa de trabajo útil.

Hemos podido constatar un caso patológico muy común en el modelo BASE, debido al ciclo de parada de inicio de ejecucción un salto mal predicho ve retardada su resolucción por un errores de predicción de latencia de un *load* que pertenece al camino mal predicho por el salto. Por contra, CR no bloquea el inicio de ejecución, por lo que los slots de inicio quedan disponibles a la lógica de selección, que seleccionará entre las instrucciones listas para ejecutar a las más viejas. Por ello, no se retarda la comprobación de la predicción en una instrucción de salto más vieja que el *load* mal predicho.

La Figura 4.11 muestra en media aritmética, por cada error de predicción de latencia, el número de instrucciones dependientes del *load* mal predicho que inician la ejecución después de detectar el error de predicción. Está medida representa la longitud en instrucciones de la cadena de recuperación.

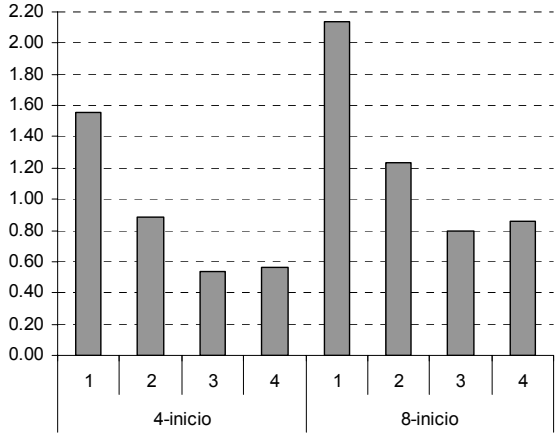

Figura 4.11 Longitud media en instrucciones de la cadena de recuperación.

La longitud de la cadena de recuperación es muy corta, entre 0,5 y 2,1 instrucciones en media. El número es mayor en el procesador *8-inicio* al disponer de mayor ancho de banda de inicio. Conforme el número de etapas entre IQ y EX aumenta, disminuye la longitud de la cadena de recuperación, ya que se incrementa el número de ciclos de la *ventana especulativa* y por tanto, después de detectar el error de predicción quedan por iniciar menos instrucciones dependientes. La longitud de la cadena aumenta ligeramente al pasar a 4 etapas entre IQ:EX, debido a la latencia de las unidades funcionales,

en concreto a la latencia del acceso a memoria (cadena de dependencias *load-load-usos*).

Resumiendo, un procesador *8-inicio* mejorado con CR consigue incrementar las prestaciones del procesador BASE entre un 6,4 y un 4,5% conforme el número de etapas IQ:EX pasa de 1 a 4. El procesador *4-inicio* sin embargo consigue menores beneficios (entre 2,4 y 2,0%). CR elimina casi por completo la penalización por error de especulación alcanzando en el peor de los casos el 98,3% del IPC de la opción OCP-CR.

La Figura 4.12 y la Figura 4.13 muestran el rendimiento por programa sobre los procesadores *4-inicio* y *8-inicio*, respectivamente. De nuevo se observa que la tendencia general se mantiene. La *recuperación en cadena* mejora el rendimiento del BASE en todos los programas.

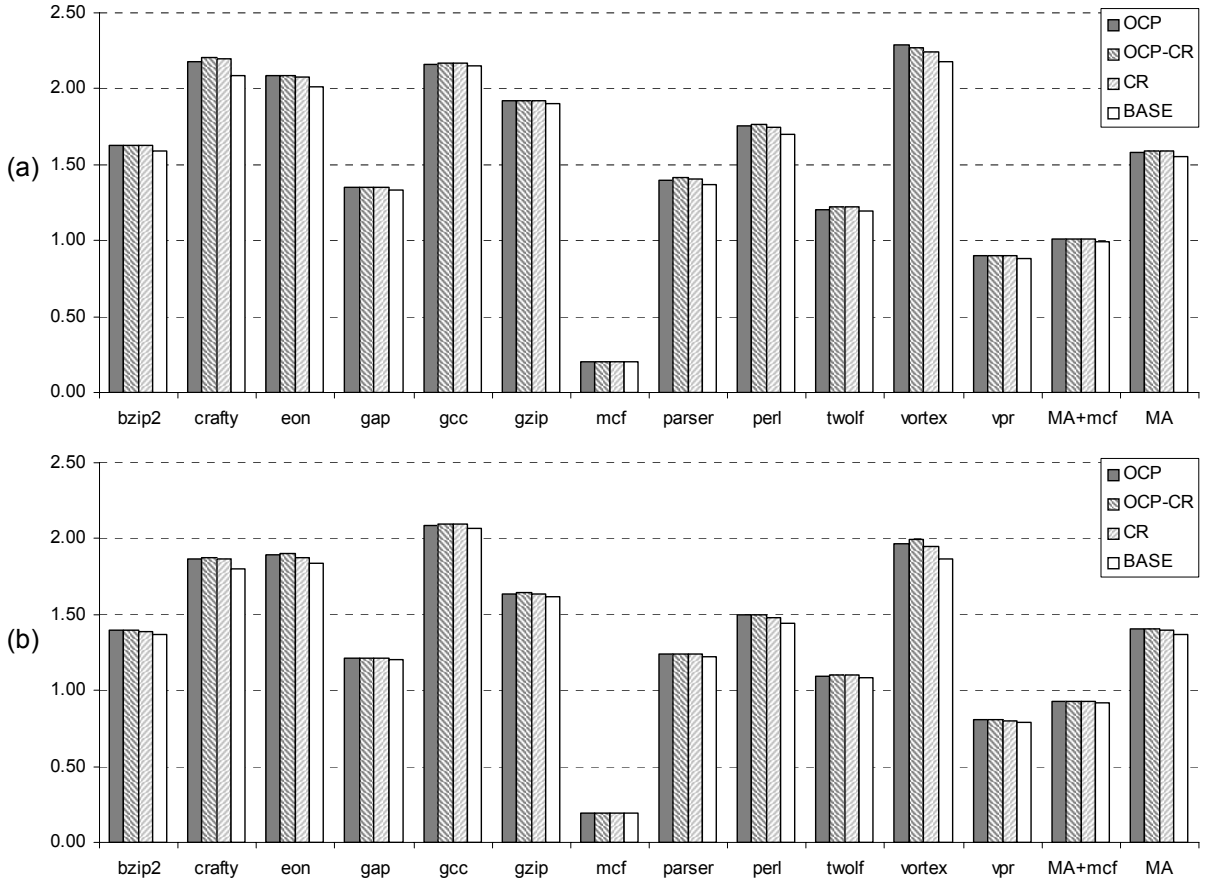

Figura 4.12 IPC para el procesador *4-inicio* y una etapa (a) o cuatro etapas (b) entre IQ y EX; al aplicar la *recuperación en cadena* (*Chained Recovery*).

Con una etapa entre IQ:EX casi todos los programas alcanzan el rendimiento del ideal (OCP-CR), consiguiendo que la recuperación por error de especulación no tenga coste en rendimiento. Esto ocurre claramente en el *4-inicio* y, aunque no tan claro, en el *8-inicio*. Con 4 etapas entre IQ:EX y sobre todo en el *8-inicio*, aparecen pequeñas pérdidas respecto al ideal, pero se mejora el rendimiento en todos los programas respecto al BASE. Resulta curioso constatar que programas como EON y VORTEX, con tasas de errores de banco alrededor de la media, sufren las mayores penalizaciones (diferencia OCP-CR y CR). Sin embargo, programas como VPR con tasas de errores muy por encima de la media, no notan en rendimiento dicha penalización.

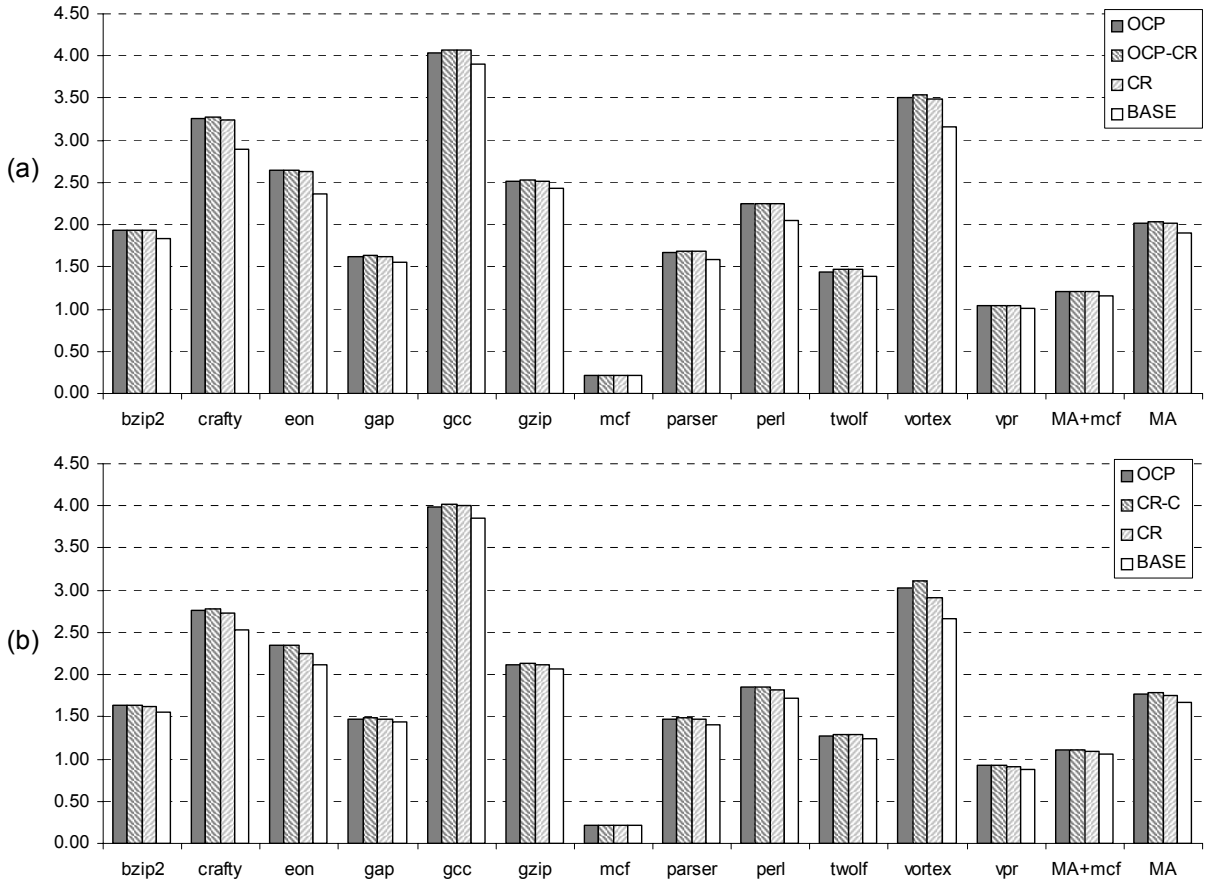

Figura 4.13 IPC para el procesador *8-inicio* y una etapa (a) o cuatro etapas (b) entre IQ y EX; al aplicar la *recuperación en cadena* (*Chained Recovery*).

4.5 Contrarrestando el incremento de latencia

Tal y como se mostró en la sección 4.3, los errores de predicción de banco incrementan la latencia *load-uso* y su efecto es más pronunciado al incrementarse el número de etapas IQ:EX. Para reducir esta penalización, en vez de re-iniciar desde IQ los *loads* mal predichos, Neefs y otros proponen utilizar una red dedicada de re-encaminamiento una vez conocido el error de predicción de banco. La red está situada fuera del camino crítico y re-encamina instrucciones mal predichas al banco correcto [NeVB00]. La Figura 4.14 compara la temporización al utilizar una red de re-encaminamiento con la opción de re-encaminar desde la IQ.

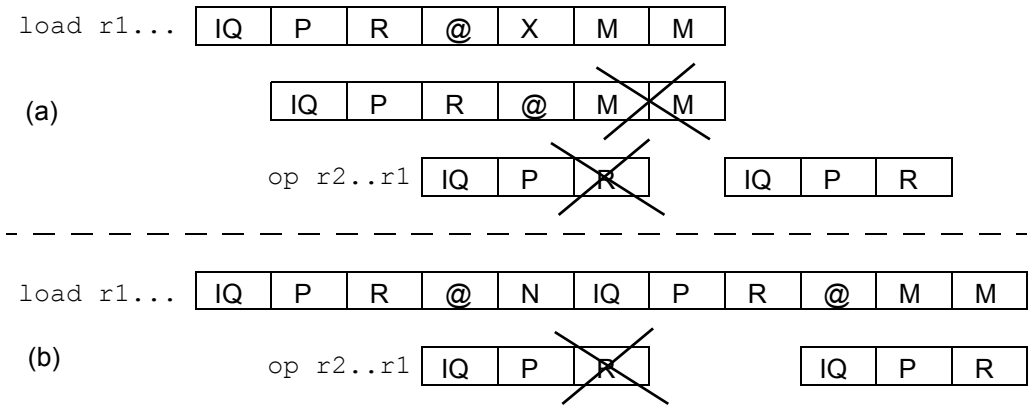

Figura 4.14 Ciclos (a) red de re-encaminamiento comparada con (b) re-iniciar desde IQ.

Sin embargo en un procesador super-escalar con predicción de latencia esta técnica no puede ser aplicada. Por un lado, al iniciar una instrucción se planifica el uso de ciertos recursos compartidos como pueden ser, la red de cortocircuitos o el puerto de escritura al banco de registros. Al re-encaminar necesitaremos acceder entre otras cosas al banco de cache correcto, que podría haber sido planificado varios ciclos antes de saber si nos hemos equivocado para ser utilizado por otra instrucción *load* o un *refill* desde L2 durante ese mismo ciclo (2ª instrucción en Figura 4.14.a). Por otro lado, las instrucciones dependientes del *load* ya han sido despertadas de forma especulativa y potencialmente iniciadas en sincronía para capturar el dato al vuelo en la red de cortocircuitos. Si variamos la latencia predicha al re-encaminar, deberemos en cualquier caso anular a las dependientes y volverlas a re-ejecutar (3ª instrucción en Figura 4.14.a). Por ello, el aumento de latencia *load-uso* sería similar a la opción de re-encaminar desde la IQ tal y como se muestra en la Figura 4.14.b.

En [YMRJ99] Yoaz y otros, proponen iniciar el *load* a todos los puertos de inicio libres sin utilizar la predicción, evitando por tanto fallar y tener que re-ejecutar el *load*. Su modelo dispone de una cola de inicio independiente de la de enteros para las instrucciones de memoria en un entorno con dos bancos. Comentan que los *stores* no están en el camino crítico, por lo que son siempre replicados a los dos bancos. En el caso de los *loads*, solo replicarán un *load* si no hay otro *load* listo para ser iniciado. Evalúan su modelo de camino de memoria seccionado sobre dos bancos de forma analítica, sin replicación de *loads* (no modelan contención) y sin mostrar rendimiento.

Estudiaremos una implementación de este modelo junto a otro que proponemos en la siguiente sección. En concreto, modelamos que después de la fase de selección en IQ se identifica al *load* más viejo de las instrucciones seleccionadas. Si quedaron puertos a memoria libres enviaremos a cada uno de ellos una instancia de ese *load* mientras no agotemos el ancho de banda de inicio a ejecución. Las instrucciones dependientes serán iniciadas especulativamente como hasta ahora. Denominamos a este sistema Replicación haciendo uso de puertos libres (*Replication Using Free Memory Slots*: **RF**). Nótese que nuestro modelo comparte la IQ tanto para instrucciones de memoria como para instrucciones de aritmética entera.

4.5.1 Replicación selectiva en Emisión

En esta sección presentamos nuestra propuesta. A semejanza de Yoaz y otros [YMRJ99] la acción que tomaremos es conservadora pero se basará en una estimación de la confianza en la predicción de banco. Además la replicación de los *loads* no dependerá de los puertos de inicio a memoria en un ciclo determinado.

Añadimos un estimador de confianza al predictor de banco como vimos en el Capítulo 3. Cuando la confianza asignada a un *load* es baja se descarta la predicción de banco y el *load* se marca al insertarlo en la IQ para que sea iniciado por todos los puertos de inicio a memoria, a esta propuesta la denominamos ***Replicación Selectiva en Emisión*** (*Replication in Dispatch*: **RD**). El objetivo es forzar, en *emisión*, la replicación de los *loads* difíciles de predecir, a costa de disminuir el ancho de banda de inicio a memoria del procesador.

Para estimar la confianza hemos añadido un tercer bit a los contadores de cada entrada del predictor de banco. La Tabla 4.2 muestra las estadísticas del predictor de banco.

Tabla 4.2 Precisión del predictor para 2 y 4 bancos con estimador de confianza.

Gskew 1Ke x 3bits historia 6	Loads			
	Alta confianza		Baja confianza	
	acierto	error	acierto	error
2 bancos	72,96	3,00	14,97	9,07
4 bancos	62,17	2,44	17,90	17,49

RD puede producir pérdidas de rendimiento en ciertas situaciones. Los *loads* replicados por RD son aproximadamente el 35,39% (24,04%) de todos los *loads* para 4 (2) bancos, el replicado de estos *loads* podría retardar la ejecución de otras instrucciones más jóvenes por falta de puertos de inicio.

Como los *loads* marcados tienen baja confianza, decidimos no despertar especulativamente a las instrucciones dependientes. Las instrucciones serán despertadas con la comprobación de banco correcto o el inicio de la última instancia del *load*. Por ello, los *loads* de baja confianza que fueron bien predichos (ej. 17,90% para 4 bancos) ven como el inicio de sus instrucciones dependientes son retardadas hasta la comprobación o el inicio de la última instancia. Por contra, las instrucciones de baja confianza mal predichas (ej. 17,49% para 4 bancos) consiguen los beneficios de la replicación: decremento de la latencia *load-uso* y eliminación de la penalización por error de especulación.

Todas las instancias de los *loads* de baja confianza no tienen por qué ser iniciadas en el mismo ciclo. Cada selector de banco hará uso de la edad de la instrucción para seleccionar, entre las instrucciones listas, la candidata a iniciar. De esta forma, RD no entorpece el inicio de instrucciones más viejas listas para ser iniciadas. Además, cuando la notificación del chequeo de banco llega a la IQ, se anulan todas las instancias pendientes de inicio de ese *load* a bancos incorrectos.

4.5.2 Evaluación

Como dijimos en la sección 4.3, contrarrestar los efectos causados por los errores de predicción de banco ofrece buenas oportunidades para mejorar el

rendimiento. Observando la diferencia entre OBP-CR y BASE en la Figura 4.15 se puede ver que las oportunidades crecen conforme aumenta el número de etapas IQ:EX. Las tres barras entre OBP-CR y BASE muestran el rendimiento de una serie de mejoras ordenadas por complejidad: RF, RD, y la combinación de RD con la *recuperación en cadena* (RD-CR). Si nos centramos en la configuración con 4 ciclos entre IQ y EX y nos movemos desde la alternativa más simple a la más completa veremos:

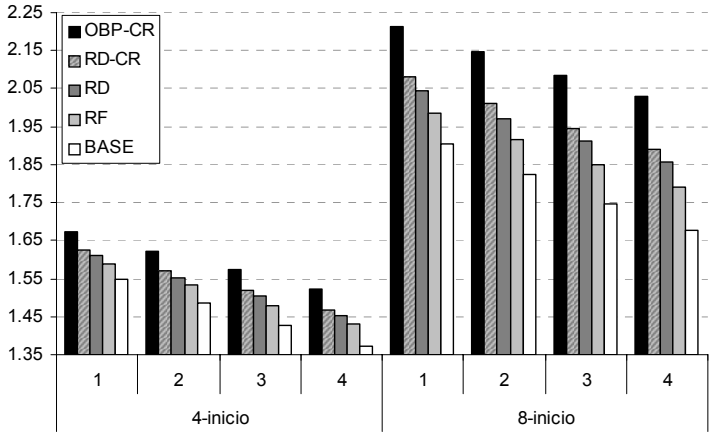

Figura 4.15 Media Armónica de IPC. OBP-CR dispone de un oráculo como predictor de banco. Los procesadores con predictores reales añadiendo mejoras son: Replicación usando slots de memoria libres (RF), Replicación en Emisión (RD) y este último añadiéndole la *recuperación en cadena* (RD-CR).

RF, mejora el IPC hasta un 6,8% (4,2%) respecto a BASE para la configuración *8-inicio* (*4*), respectivamente. Teniendo en cuenta que RF añade muy poca complejidad a la lógica de control, el resultado obtenido es muy bueno.

Añadiendo la estimación de confianza al predictor (invirtiendo 6 Kbits de memoria), el rendimiento aumenta al 10,8% (5,8%) en la configuración *8-inicio* (*4*) con la mejora RD respecto al BASE.

Si añadimos a RD la recuperación en cadena CR, estaremos simultáneamente decrementando la penalización por error de especulación y la media de retardo *load-uso*. El IPC de RD-CR se incrementa sobre el BASE un 12,7% (6,9%) para la configuración *8-inicio* (*4*).

Por último, comentar que independientemente del número de etapas IQ:EX y del ancho de inicio, RD-CR alcanza los dos tercios del rendimiento ideal dado

por OBP-CR. La solución RD-CR se muestra muy interesante para todo un conjunto amplio de parámetros arquitectónicos y por tanto un punto de diseño atractivo en el compromiso coste/rendimiento.

La Figura 4.16 y la Figura 4.17 muestran el rendimiento por programa de los distintos esquemas con y sin replicación para los procesadores 4 y 8 inicio con entre 1 y 4 etapas de IQ a EX.

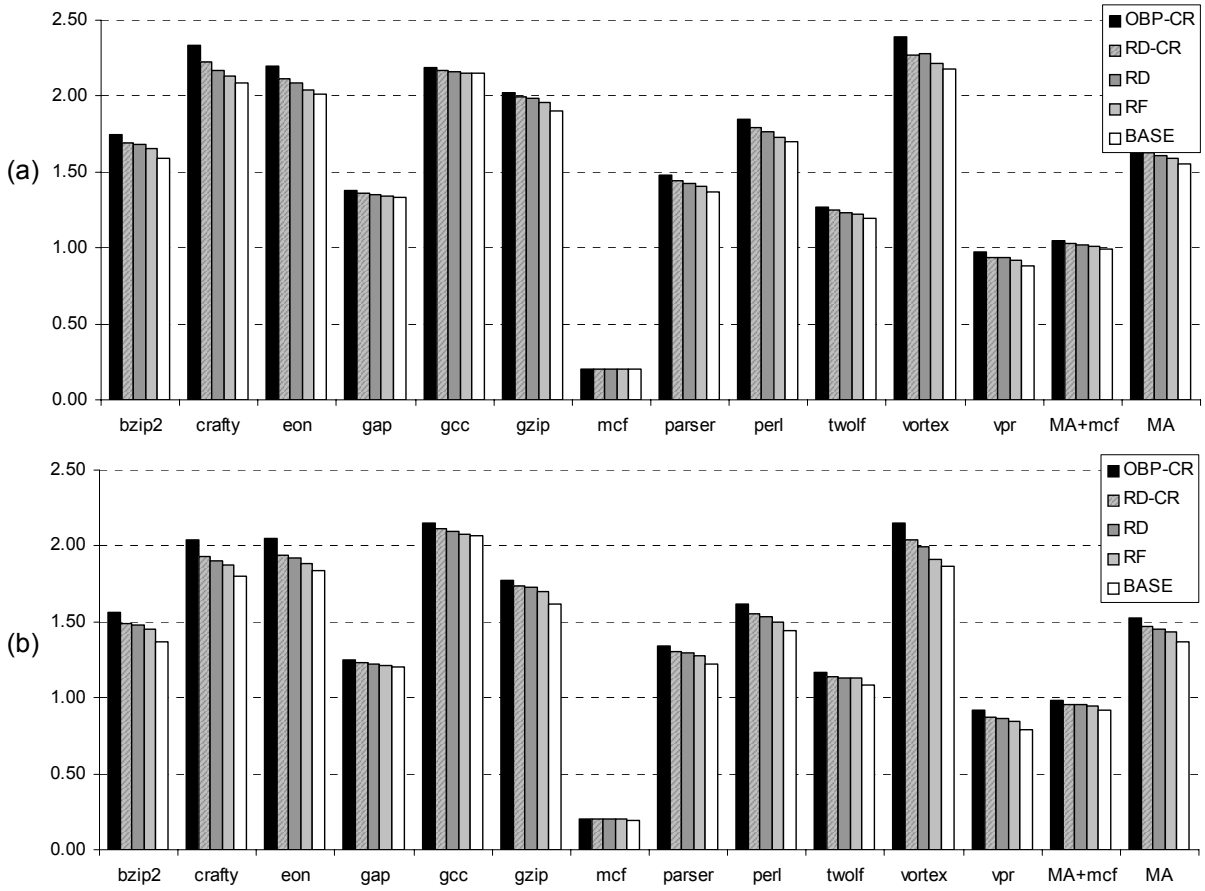

Figura 4.16 IPC para el procesador *4-inicio* y una etapa (a) o cuatro etapas (b) entre IQ y EX. OBP-CR dispone de un oráculo como predictor de banco. Los procesadores con predictores reales añadiendo mejoras son: Replicación usando slots de memoria libres (RF), Replicación en Emisión (RD) y este último añadiéndole la *recuperación en cadena* (RD-CR).

De nuevo se puede concluir que los programas individualmente siguen la tónica general de la media. En el único programa que RF mejora en rendimiento a RD es GCC y en el *8-inicio*. En todos los demás casos, tomar acciones basadas en la confianza de la predicción (RD) mejora las prestaciones de RF.

Se puede observar que ciertos programas, (dependiendo del procesador y del número de etapas entre IQ:EX), no mejoran su rendimiento al añadir la *recuperación en cadena* cuando se replican los *loads* (RD vs RD-CR), por ejemplo `TWOLF` en *4-inicio* con 4 etapas entre IQ:EX.

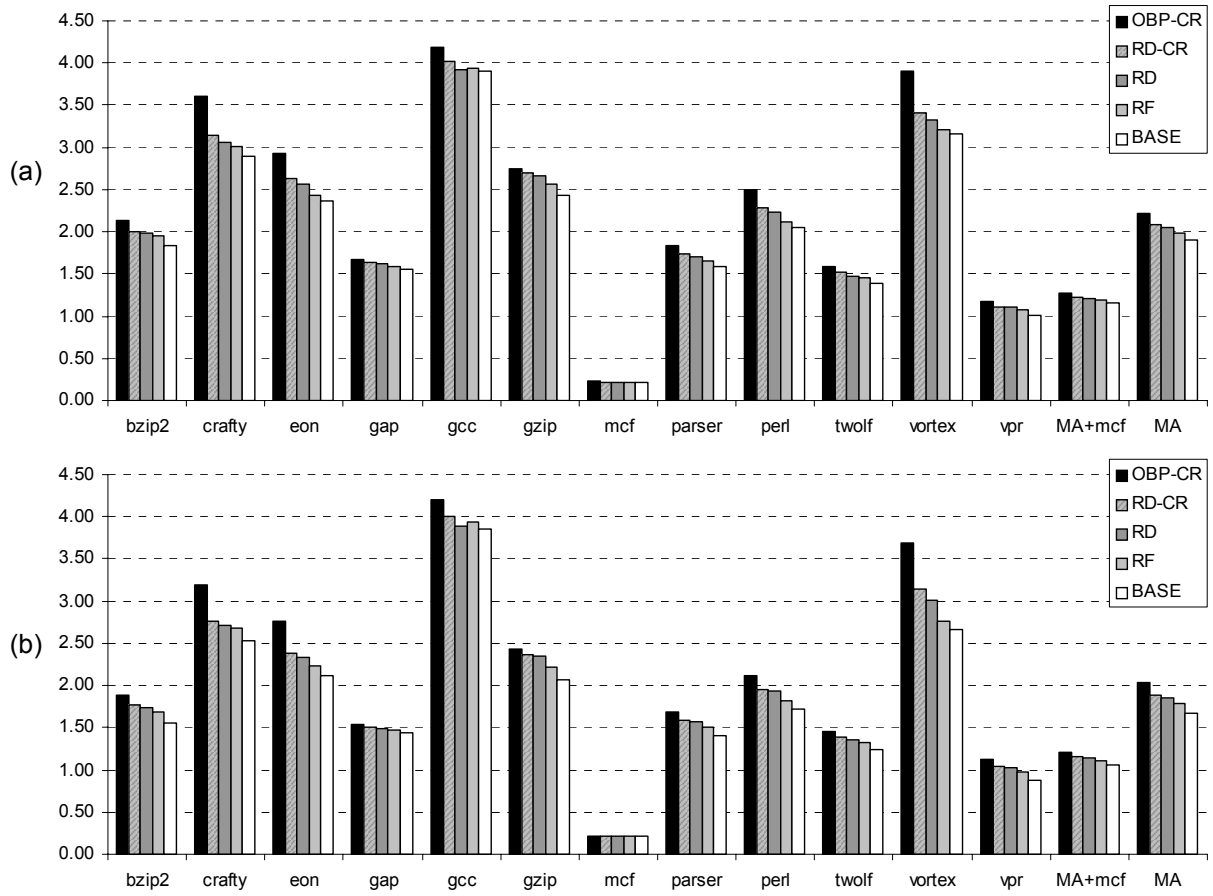

Figura 4.17 IPC para el procesador *8-inicio* y una etapa (a) o cuatro etapas (b) entre IQ y EX. OBP-CR dispone de un oráculo como predictor de banco. Los procesadores con predictores reales añadiendo mejoras son: Replicación usando slots de memoria libres (RF), Replicación en Emisión (RD) y este último añadiéndole la *recuperación en cadena* (RD-CR).

4.6 Conclusiones

Las tendencias actuales en el diseño de micro-arquitecturas, tales como incremento del ancho de inicio y caminos de datos con segmentados más profundos, tienen una influencia negativa en los procesadores con camino de datos a memoria seccionado. Esta influencia tiene relación directa con la tasa de errores del predictor de banco. En este capítulo hemos introducido dos

alternativas complementarias para reducir globalmente la penalización por error de predicción de banco.

La primera denominada *Recuperación en Cadena*, se centra en reducir la penalización debida a la acción necesaria de recuperación en un error de predicción de latencia. Hemos mostrado que CR es muy efectiva, alcanzando en el peor de los casos el 98,3% del IPC obtenido por un sistema sin penalización por error de especulación (confianza perfecta).

La segunda alternativa se centra en reducir la latencia *load-uso* en error de predicción de latencia. Esta técnica la hemos denominado Replicar en Dispatch y utiliza la confianza del predictor para no hacer uso de la predicción de banco. Como no existe predicción, se indica a la IQ que el *load* debe ser iniciado por todos los caminos de acceso a memoria. El incremento en rendimiento llega al 10,8% en la configuración *8-inicio* más profundamente segmentada respecto al BASE reduciéndose la pérdida respecto a un sistema con predicción ideal (OBP) al 7,5%.

Combinando la *recuperación en cadena* y *replicando en emisión*, eliminamos alrededor de dos tercios de la pérdida de IPC por error de predicción de banco en todas las configuraciones simuladas.

CAPÍTULO 5

Gestión de Contenidos en Caches L1 Multibanco

El objetivo de este capítulo es estudiar las diversas propuestas de distribución de contenidos existentes para caches multibanco. Presentaremos una taxonomía que agrupa a todas ellas y las evaluaremos en un entorno común. Prestaremos especial atención a los compromisos entre capacidad del primer nivel, errores de predicción de banco y contención en los puertos de inicio a memoria. Por último veremos el efecto al variar la latencia de L2 y la latencia de L1.

5.1 Introducción

Los compromisos en capacidad, latencia y número de puertos de la cache de primer nivel vistos en la introducción sobre caches multipuerto también se pueden aplicar a caches multibanco.

En una cache monolítica existe una función de mapeo que permite establecer el contenedor (o contenedores según el grado de asociatividad) donde debe alojarse un bloque de memoria. En una cache distribuida se dispone de varias caches independientes. La gestión de contenidos determina a que banco o bancos deben ir los datos una vez servido el fallo de cache. Se crea por tanto una nueva función de mapeo. La gestión de contenidos añade un nuevo grado de libertad.

Esta disyuntiva sobre donde colocar los datos no es nueva e inherente a las caches multibanco, el paradigma ya ha sido aplicado en otros niveles de la jerarquía. Los procesadores vectoriales entrelazan los bancos de memoria principal para maximizar el ancho de banda. Procesadores como el Intel Pentium IV, entre otros, reparte los contenidos de tal forma que las instrucciones de coma flotante y multimedia acceden directamente al segundo nivel de la jerarquía dejando libre el primer nivel para las instrucciones de enteros [HSU+01]. Nosotros nos centraremos en el primer nivel de la jerarquía multibanco con caminos a memoria seccionados.

En cualquier caso, son las instrucciones de acceso a memoria (*loads* y *stores*) las que deben acceder a los datos, y por tanto deben saber localizarlos. En el caso descrito en el parrafo anterior, se utiliza el código de operación para determinar el nivel de la jerarquía al que deben acceder las instrucciones. En una cache multibanco necesitaremos relacionar a las instrucciones con el banco apropiado. En función de la gestión de contenidos utilizada, este emparejamiento resultará más o menos predecible.

La ejecución de un programa genera flujos de instrucciones de accesos a memoria. Estos flujos no son uniformes y generan ráfagas. En concreto nos centraremos en ráfagas de *loads* listos para iniciar su ejecución que quieren acceder al mismo banco y por tanto comparten el mismo puerto de inicio a memoria. Estas ráfagas son producidas tanto por la dinámica del programa como por la gestión de contenidos y crean contención en los puertos de inicio a memoria de la IQ. El resultado es que un programa sufrirá más o menos ráfagas y de mayor o menor longitud según la gestión de contenidos empleada. La contención acarrea pérdida de prestaciones.

La contención proviene de diversos *loads* con destino al mismo banco. Más adelante veremos en que medida son debidos a accesos a la misma palabra, al mismo bloque o al mismo banco relacionados únicamente por la gestión de contenidos. Variando la gestión evitaremos unos conflictos al mismo banco pero crearemos otros. Conflictos al mismo bloque se pueden solventar dividiendo el bloque en unidades menores y distribuyéndolas entre diversos bancos (por ejemplo Entrelazado por Palabra). Sin embargo resulta difícil eliminar los conflictos debidos a *loads* que deben acceder a la misma palabra.

La Figura 5.1 muestra una medida de “conflictos” sobre el procesador *8-inicio* con cuatro bancos entrelazados por línea. Cada ciclo, en la etapa de inicio miramos si un *load* listo no puede ser iniciado por haber seleccionado ya a un *load* más viejo a ese puerto. En ese caso se mira si ambos quieren acceder a la

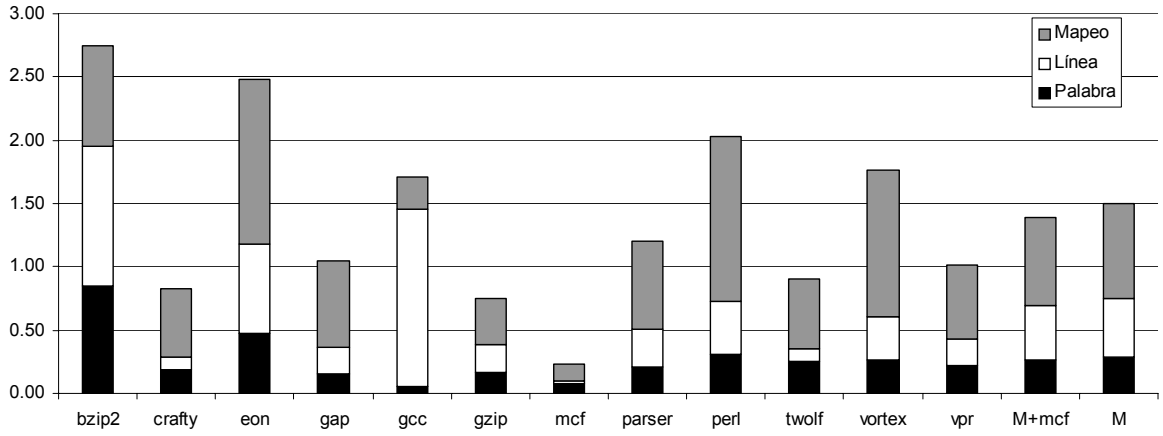

Figura 5.1 Contención por *loads* a la misma palabra, línea o banco (mapeo) en *8-inicio*.

misma palabra, al mismo bloque de L1 (línea) o simplemente han coincidido al mismo banco por el mapeo. El total ha sido normalizado con el número de *loads* consolidados.

Hay programas con muchos "conflictos" como BZIP2 o EON que ven como la latencia media de los *loads* se incrementa 2,5 ciclos debido a la contención. En media, los *loads* deben esperar 1,5 ciclos. De momento, no nos interesa tanto el valor absoluto como la proporción entre los tres tipos de conflicto. Con un entrelazado por línea con 4 bancos, en media, el 18,4% de los conflictos provienen de accesos a la misma palabra, el 27,2% a la misma línea y el 54,4% simplemente han coincidido por la función de entrelazado.

Una forma de eliminar todos estos conflictos es incrementar el número de puertos de acceso tal y como hacen las caches multipuerto. Nosotros nos centraremos en puntos de diseño con la misma capacidad y tiempo de acceso constante. Por ello, para aumentar el número de caminos de acceso replicaremos los datos a varios bancos (*mirror caching* [SoFr91]).

Por otra parte, la gestión de contenidos determina como se divide el flujo de accesos a memoria, y por tanto también los fallos de cache. Cada gestión saca mejor o peor partido de la capacidad disponible generando una tasa de fallos distinta. A ésto hay que añadir que al replicar perderemos capacidad efectiva en el primer nivel de la jerarquía.

En resumen, con cada gestión de contenidos variará la tasa de fallos de cache, la precisión del predictor de banco y la contención en los puertos de inicio a

memoria de la IQ. En este capítulo pondremos de manifiesto estos compromisos, propondremos una taxonomía de las distintas propuestas preexistentes y las evaluaremos en un entorno común.

En la sección 5.2 comentaremos la gestión de contenidos atendiendo al grado de replicación y a la política de distribución. En la sección 5.3 describiremos los puntos de diseño elegidos para evaluar las distintas gestiones de contenidos. En la sección 5.4 enunciamos el modelo de procesador y la jerarquía de memoria sobre la que se evaluarán y en la sección 5.5 mostraremos los resultados. Por último en la sección 5.6 presentaremos unas breves conclusiones.

5.2 Gestión de Contenidos

Como ya hemos dicho la gestión determina en que banco o conjunto de bancos deben ir los datos servidos en un fallo de cache. Como el camino a memoria está seccionado en bancos y cada banco de cache está enlazado con la IQ a través de un único puerto de inicio de ejecución a memoria, todos los *loads* deben ser emitidos a la IQ llevando consigo la información del puerto por el que deben iniciar la ejecución.

A la hora de clasificar los distintos esquemas de gestión de contenidos podemos centrarnos en dos aspectos; política de distribución y grado de replicación.

La **política de distribución** es la función de mapeo que decide donde colocar los datos. La política de distribución dispone de un mecanismo que predice o sugiere el banco destino de cada *load*. Podemos clasificar las políticas entre estáticas y dinámicas. Las políticas estáticas siempre colocan los datos en los mismos bancos. Las políticas dinámicas permiten que la asignación de los datos varíe durante la ejecución. Podemos distinguir cuatro políticas de distribución; distribución por tipo de datos, por dirección de memoria de los datos, por el conjunto de trabajo referenciado y en función de la contención.

El **grado de replicación** indica el número de bancos en los que podemos tener copias de los datos. De nuevo el grado de replicación puede ser estático o dinámico (bajo demanda según la ejecución).

Atendiendo a estos dos ejes, se pueden clasificar todas las propuestas de gestión de contenidos como muestra la Tabla 5.1.

Tabla 5.1 Taxonomía de las propuestas de gestión de contenidos.

Denominación	Grado de Replicación	Política de Distribución	Comentario
Tipo de Datos	Estático	Estática	AR - *Access Region* [ChYL01]
Dirección de los datos	Estático	Estática	Línea y Palabra [SoFr91]
Conjunto de datos	Estático	Dinámica	IWS - *Instructión Working Set* [RaPa03]
Consciente de la Contención	Dinámico	Dinámica	CA - *Conflict Aware* [LiRS01]

5.2.1 Grado de Replicación

Como ya hemos indicado, el grado de replicación nos indica el número de copias que tenemos repartidas entre los bancos.

La primera propuesta que existe atendiendo al grado de replicación es la *Mirror Cache*, propuesta por Sohi y Franklin [SoFr91] e implementada en procesadores como el Digital Alpha 21164. La propuesta original estaba pensada para dos bancos. En cada fallo de cache los datos se cargan sobre los dos bancos. La capacidad efectiva se reduce a la mitad, sin embargo los *loads* pueden acceder a cualquiera de los dos bancos y por tanto libres de contención. Otro inconveniente es que los *stores* deben actualizar ambas copias de la cache al consolidar.

Generalizando esta idea a cuatro bancos nos encontramos con tres posibles grados de replicación; No-Replicado, se coloca una única copia de los datos en un sólo banco, Parcialmente Replicado, se colocan dos copias de los datos, y Replicado Total donde los datos se colocan en todos los bancos.

Replicado Total, al igual que en la *mirror cache*, carga los datos en los cuatro bancos (Figura 5.2.a). La capacidad efectiva se ve reducida a la capacidad de un sólo banco, pero los accesos (*loads*) se ven libres de contención en los puertos de inicio, ya que al igual que en caches multipuerto, en cada ciclo se pueden servir cuatro peticiones cualesquiera, incluso a la misma palabra.

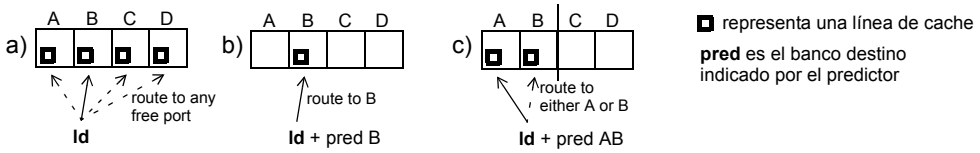

Figura 5.2 Grado de replicación; (a) Replicación Total, (b) No-Replicado y (c) Replicación Parcial.

No-Replicado permite una única copia de los datos en L1 (Figura 5.2.b). La capacidad efectiva es la mayor disponible (la suma de las capacidades de cada banco) pero los accesos sufrirán contención en los puertos de inicio de la IQ.

A mitad de camino se sitúa **Replicación Parcial**: se permiten múltiples copias de los datos (Figura 5.2.c). Replicación Parcial sacrifica capacidad efectiva para reducir contención.

La replicación de contenidos aporta un efecto secundario como es la mejora en precisión del predictor de banco, simplemente por el hecho de que es más fácil acertar el banco entre 2 bancos que entre 4, o incluso no necesitar predictor como en el caso de Replicación Total.

Los grados de replicación que hemos visto hasta ahora son estáticos. La decisión sobre el número de copias es fija. Sin embargo, se ha propuesto una política de distribución, que denominaremos *Conflict Aware,* que replica dinámicamente los contenidos para disminuir la contención [LiRS01].

5.2.2 Distribución de Contenidos

A continuación describiremos cada política de distribución y en la siguiente sección mezclaremos el grado de replicación y las políticas para definir los puntos de diseño evaluados.

Políticas de Distribución Estáticas:

Distribución por tipo de dato. Consiste en distribuir estructuras de datos entre bancos. En el contexto de caches de primer nivel, esta idea sólo ha sido aplicada por Cho y otros [ChYL01] para separar la región de memoria correspondiente a la pila del resto de regiones. Dependiendo de la arquitectura del lenguaje máquina, la mayor parte de las instrucciones pueden ser separadas

entre pila y no pila en la etapa de decodificación en función del código de operación o de los registros fuentes (registro SP). Existen referencias a pila que no hacen uso de registros especiales, por lo que los autores incluyen un predictor de Región de Acceso.

Distribución por dirección de memoria. Esta es la política más convencional y consiste en distribuir los datos según una función de *hash* aplicada a algunos bits de la dirección de memoria [SoFr91]. Las funciones de *hash* más habituales reparten por bloque de cache (entrelazado por Línea) o por palabra (entrelazado por Palabra).

Políticas de Distribución Dinámicas:

Distribución por working set (Instruction Working Set). Propuesta por Racunas y Patt [RaPa03], intenta colocar en un banco concreto todo el conjunto de datos de trabajo referenciados por cada *load*. Las líneas de cache referenciadas por diversos *loads* pueden migrar entre bancos en función de la confianza relativa entre dos predictores. El predictor iPAT sugiere al *load* donde buscar sus datos, el predictor dPAT indica en caso de fallo de cache donde colocar los datos. Los errores de predicción de banco requieren que los *loads* accedan a L2 como si de un fallo de cache se tratara, pero sin realizar la carga (*refill*) sobre el banco erróneo.

Distribución en función de los conflictos (Conflict Aware). Limaye y otros proponen en [LiRS01] un mecanismo de distribución dinámico, denominado *cachelet*, orientado a reducir la contención en los puertos de inicio. La idea básica es recordar por que puerto fue iniciado un *load* la última vez que se ejecutó. Si se produce contención, el *load* se inicia por otro puerto libre. La próxima vez que se ejecute se iniciará preferentemente por este último puerto. En caso de fallo de cache, los datos de L2 son colocados en el banco por el que se pidio y por tanto realizando réplicas dinámicamente bajo demanda pudiendo llegar a tener copias de los datos en todos los bancos. La decisión de cambiar de puerto para reducir la contención se puede hacer en emisión o en inicio cuando se sabe que existe realmente contención. Nosotros implementamos esta última por ser la más favorable.

5.3 Puntos de diseño estudiados

Las políticas descritas han sido evaluadas por sus respectivos autores en la mayoría de los casos en otros entornos, de forma analítica o con modelos de procesador y de jerarquía de memoria muy diversos. En la sección 5.5 evaluaremos el comportamiento de todas las políticas bajo el mismo entorno.

En la Tabla 5.2 presentamos los puntos de diseño ordenándolos en función del grado de replicación (en cierto modo en función de la capacidad efectiva de L1). La replicación de los datos introduce un compromiso, ya que si bien disminuye la contención, la reducción en capacidad efectiva acarrea un incremento en la tasa de fallos de cache L1.

Tabla 5.2 Puntos de diseño estudiados.

	Grado de Replicación		
	No-Replicado	**Parcialmente Replicado**	**Replicación Total**
Tipo de Datos		AR-2R	
Dirección de los datos	Word Line	Word-2R Line-2R	4R
Conjunto de datos	IWS	IWS-2R	
Consciente de la Contención			CA

No-Replicado:

Word: Distribución por la dirección de memoria de los datos entrelazando por palabra (bits 3 y 4).

Line: Distribución por la dirección de memoria pero entrelazando por línea (bits 5 y 6).

En ambos utilizamos el predictor *enhanced skewed binary predictor* comentado Capítulo 3 con 9 Kbytes por bit a predecir (8Kentradas de contadores de 3 bits y 9 bits de historia). Los *loads* clasificados por el predictor como de baja confianza son replicados selectivamente de forma conservadora (RD - replicación en emisión). Notar que replicar los *loads* implica iniciarlos por varios puertos lo cual puede incrementar la contención.

IWS: *Intruction Working Set*. Distribución dinámica por conjunto de datos. Los predictores iPAT y dPAT requieren un total de 16 Kbytes.

Replicado:

Parcialmente Replicado es evaluado agrupando los 4 bancos en dos conjuntos de dos bancos replicados cada uno. Las políticas empleadas son:

Word-2R: Entrelazado por palabra (bit 3).

Line-2R: Entrelazado por línea (bit 5).

En ambos casos necesitaremos predecir un único bit (predictor de 9 Kbytes), en caso de baja confianza se replica el *load* a ambos subconjuntos de bancos (dos réplicas).

IWS-2R: Distribución por conjunto de datos. La propuesta original [RaPa03] se evalúa sobre bancos con 2 puertos, nosotros los agruparemos en dos subconjuntos de dos bancos replicados cada uno. Los predictores requieren 8 Kbytes en total.

AR-2R: Distribución en función del tipo de datos. Al igual que los autores implementamos un predictor de Region de 4 Kbytes.

CA: *Conflict Aware*, distribución consciente de los conflictos, es otro punto de diseño con replicación, en este caso se pueden tener un número arbitrario de copias (desde una a cuatro). Se trabaja con los cuatro bancos de forma independiente y los datos se irán cargando en cada uno de ellos bajo demanda. Los autores, al igual que nosotros, evalúan la propuesta sobre cuatro bancos con un predictor de 8 Kbytes para recordar el puerto de inicio de la última ejecución.

4R: Por último evaluaremos Replicación Total, con cada fallo de cache cargaremos los datos en los cuatro bancos.

5.4 Procesador y memoria

Modelamos el procesador superescalar de grado 8 (*8-inicio*) descrito en el Capítulo 2, con un ciclo entre IQ y ejecución. En caso de error de especulación de latencia la *Recuperación* es selectiva *en Cadena*.

La cache de primer nivel está seccionada en 4 bancos direccionados independientemente. Cada banco dispone de un único puerto de acceso compartido por *loads*, *stores* en consolidación y *refill* desde L2. Modelamos un STB multipuerto a la misma latencia del primer nivel de cache.

Los *stores* al iniciar la ejecución son lanzados por un puerto de inicio a memoria libre y enviados al STB multipuerto. Por ello los *stores* no crean por si mismos contención, ni sufren errores de banco. Los *stores* consolidan su estado sobre L2. Si los datos residen en L1, los *stores* ya consolidados se colocan en un *buffer de escritura* local a cada banco y lo actualizarán en ciclos libres. Si existen réplicas de los datos, los *stores* serán colocados en los bancos que toque. El tamaño de los *buffers de escritura* está dimensiónado para no perjudicar a ningún punto de diseño (8 entradas con compactación por banco).

Para la desambiguación de memoria asumimos un predictor oráculo de independencia entre *store* y *load*, el predictor crea una dependencia entre *stores* y *loads* a la misma dirección, la cual es gestionada por la IQ.

5.5 Resultados

En los experimentos evaluaremos el rendimiento de las diversas propuestas variando la capacidad de los bancos del primer nivel entre 2 Kbytes y 16 Kbytes, siempre con 4 bancos. El rendimiento se expresa en media armónica de IPC sin tener en cuenta el programa `MCF`.

Como métrica fundamental haremos uso del IPC, estableciendo correlaciones con otras medidas tales como la tasa de fallos de cache L1 y la contención en los puertos de inicio a memoria. Para estas medidas sólo se presentan los resultados sobre bancos de 8 Kbytes.

La Figura 5.3 presenta la tasa de errores de predicción de latencia. Existen dos posibles causas de error de predicción de latencia, fallo de cache en L1 y error con confianza del predictor de banco. Hemos preferido agrupar las dos en una sola barra ya que a pesar de que cada una de ellas tiene penalizaciones distintas

(distinto ciclo de notificación y distinta latencia de resolución), la longitud total de la barra nos da una idea general para poder comparar las distintas propuestas. Los fallos de cache por programa y tamaño de banco se muestrán en la Figura b.6 del Apéndice B.

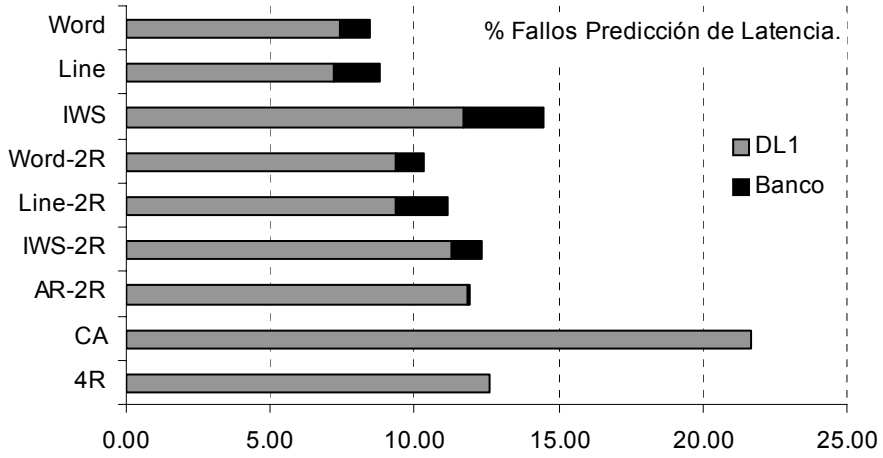

Figura 5.3 Tasa de errores de predicción de latencia (bancos de 8 Kbytes).

Podemos observar como los puntos de diseño con mayor grado de replicación sufren más fallos de cache. *Conflict Aware* (*CA*) destaca con más del 20% de los *loads* sufriendo fallos de cache debido a que se cambia la asignación del banco a los *loads* cuando hay contención. Con la misma capacidad efectiva, *IWS* tiene una tasa de fallos de cache mucho mayor que por ejemplo *Line* o *Word*, y lo mismo pasa en sus respectivos modelos replicados. La explicación tiene que ver con el desequilibrio de carga entre bancos. *IWS* tiende a concentrar la actividad en un banco (*working-set* del mismo *load* o de *loads* que lo comparten).

Si comparamos *Line*, *Line-2R* y *4R*, podemos ver como, al incrementar gradualmente el grado de replicación, la tasa de fallos de cache va creciendo debido a que disminuye la capacidad efectiva de L1. Por otro lado la tasa de errores con confianza del predictor de banco disminuye. Globalmente la longitud total de la barra crece.

En cuanto a los errores de predicción, destaca la precisión del predictor en *Access Region* (*AR-2R*) con una tasa de acierto que supera el 99%. Con el mismo número de conjuntos, *IWS* tiene más errores de predicción que *Line* y ambos más que *Word*. Recordemos que los errores de predicción de banco de los modelos *IWS* e *IWS-2R* son tratados como fallos de L1 aunque no se realice la carga de los datos al banco.

La Figura 5.4 presenta la segunda medida colateral, la contención en los puertos de inicio a memoria de la IQ. La medida muestra el número medio de ciclos que un *load* preparado debe esperar en la IQ al tener su puerto de inicio ocupado por un *load* más viejo. La idea general es: si en un ciclo nos centramos en los 4 *loads* más viejos, en *4R* se podrían ejecutar los 4 a la vez por lo que no hay contención. En *Line* podría ocurrir que los 4 quieran acceder al mismo banco, el primero no sufriría retardo, el segundo un ciclo, el tercero dos ciclos y el cuarto 3 ciclos.

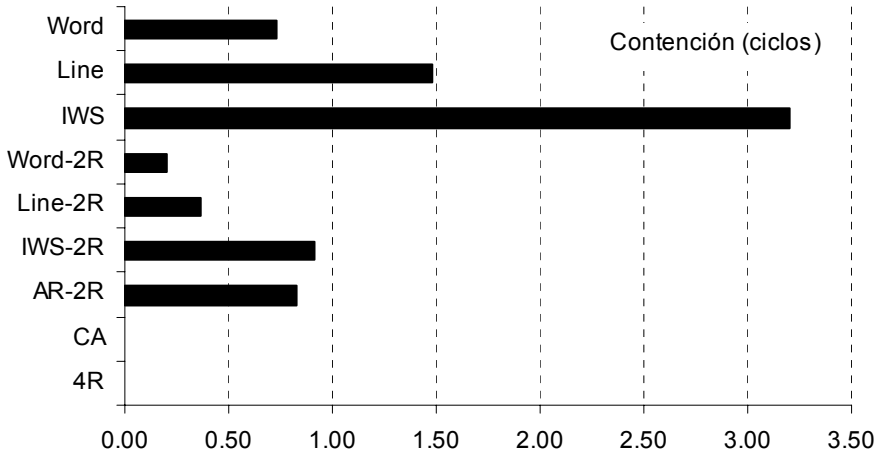

Figura 5.4 Media en ciclos que un *load* debe esperar en IQ por contención (8 Kbytes).

IWS llega a sobrepasar los tres ciclos de espera media por instrucción *load* consolidada (reejecuciones incluidas). *Word* llega a un mejor compromiso que *Line*. Los esquemas replicados reducen todos la contención por debajo de un ciclo. Los puntos *4R* y *CA* por definición no sufren retardo debido a la contención.

En la Figura 5.5 se muestra el rendimiento alcanzado por los puntos de diseño sin replicación, con bancos de cache desde 2 Kbytes a 16 Kbytes y dos ciclos de latencia. El rendimiento de *IWS* es claramente peor que el de las políticas de entrelazado por dirección (*Word* y *Line*), debido tanto a la contención como a su mayor tasa de errores de predicción de latencia (por fallos de cache L1). El IPC de *Word* siempre es mejor que el de *Line*. *Word* tiene menor contención, menor tasa de fallos de cache y presenta mejor precisión del predictor de banco.

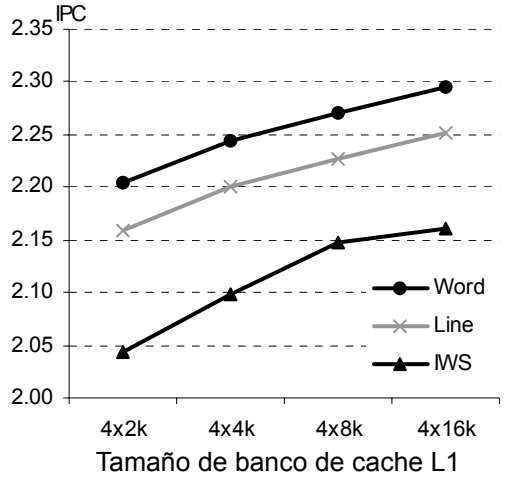

Figura 5.5 Media armónica de IPC para los puntos de diseño No-Replicados.

La Figura 5.6 muestra el rendimiento de los sistemas con replicación. Como ya hemos dicho, los esquemas con replicación sufren menos contención, tienen mejor precisión del predictor de banco pero la reducción en capacidad efectiva les hace sufrir más fallos de cache L1.

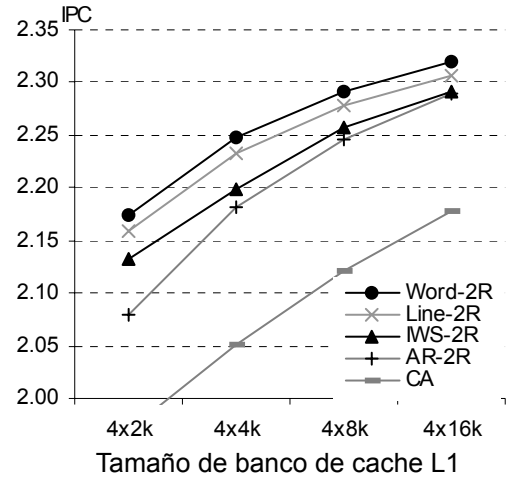

Figura 5.6 Media armónica de IPC para los puntos de diseño con replicación.

La política *Conflict Aware* tiene un IPC claramente peor que el resto. *CA* tiene demasiados fallos de cache (Figura 5.3) debido a los constantes cambios realizados en el mapeo para eliminar contención.

Access Region (*AR-2R*) e *IWS-2R* consiguen peor IPC que *Word-2R* y *Line-2R*. Ambos tienen peores funciones de reparto entre bancos desequilibrando la carga y padeciendo mayores tasas de fallos de cache y mayor contención.

Al igual que en la figura anterior, *Word-2R* siempre alcanza mejor IPC que *Line-2R*.

Finalmente analizamos la Figura 5.7 donde se muestra el mejor punto de diseño sin replicación (*Word*), el mejor de Parcialmente Replicado (*Word-2R*) junto al modelo con Replicación Total (*4R*).

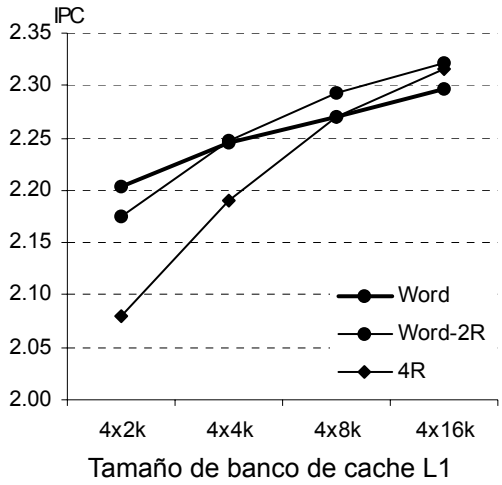

Figura 5.7 Media armónica de IPC en función del grado de replicación.

Word-2R provoca menos contención y menos errores de predicción de banco que *Word*, pero sufre más fallos de cache L1 ya que su capacidad efectiva es la mitad que en *Word*. Con cuatro bancos de 2 Kbytes, como la reducida capacidad es la principal fuente de pérdidas, *Word* es la mejor opción. Sin embargo, con bancos de más capacidad, la mejor opción pasa a ser *Word-2R*.

Incrementar la capacidad de los bancos hace que *4R* sea muy interesante. Con cuatro bancos de 16 Kbytes, el rendimiento de *4R* es similar al mejor punto de diseño, pero con un coste menor al no necesitar predictor y tener menor complejidad.

En resumen, el mejor punto de diseño son las políticas de distribución por dirección de memoria entrelazada por palabra (*Word* y *Word-2R*). El grado óptimo de replicación depende de la capacidad de los bancos.

5.5.1 Efecto de incrementar la latencia de L2

Si rehacemos de nuevo los experimentos incrementando la latencia del fallo de cache de L1 de 7 a 10 ciclos (para llegar a L2 y servir el dato), se ve (Figura 5.8) como el rendimiento en IPC baja en todos los puntos de diseño. El incremento de penalización de los fallos de L1 se sufre más en aquellas configuraciones con tasa de fallos de L1 más altas (bancos de L1 más pequeños y grado de replicación mayores).

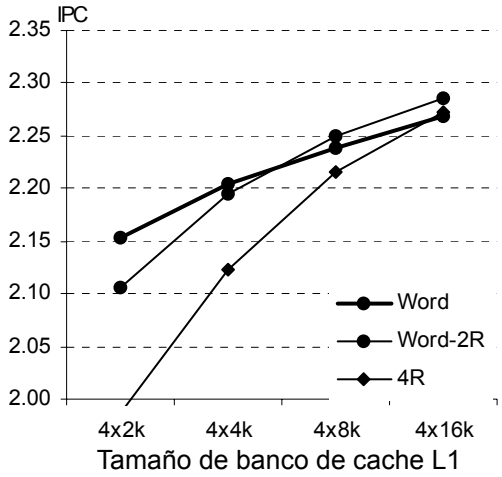

Figura 5.8 IPC de los mejores puntos de diseño al incrementar la latencia a L2.

Por ejemplo, con bancos de 2 Kbytes el IPC de la configuración Replicación Total (*4R*) disminuye un 4,3% mientras *Word* lo hace un 2,3%. El cruce entre *Word-2R* y *Word* se desplaza a la derecha y *Word* pasa a ser la mejor opción para bancos de hasta 4 Kbytes. Con bancos de 16 Kbytes todos los puntos de diseño mostrados obtienen rendimientos similares.

5.5.2 Efecto de incrementar la latencia de L1

Hasta este punto hemos supuesto, independientemente de la capacidad del banco, una latencia de 2 ciclos de acceso a L1. La latencia del banco, en ciclos, depende de la tecnología empleada y del diseño del resto del procesador (tiempo de ciclo, routing, etc.). Queda fuera del alcance de este trabajo el determinar esta latencia. Pero como es obvio bancos de distinta tamaños sufrirán distintas latencias.

La Figura 5.9 muestra el rendimiento de una cache L1 de 3 ciclos. Además, hemos añadido el rendimiento alcanzado por el mejor punto de diseño para cada

tamaño de banco con latencia de 2 ciclos extraído de la Figura 5.7 (*best L1=2*, marcados con una cruz, siendo: *Word* para bancos de 2 Kbytes, *Word-2R* para el resto).

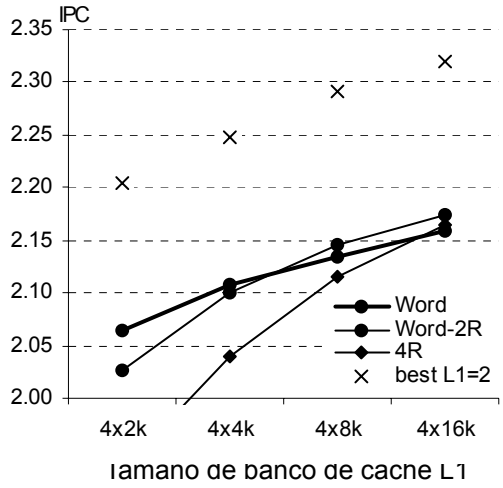

Figura 5.9 IPC al variar la latencia de L1 de 2 a 3 ciclos.

Como el 100% de los *loads* necesita un ciclo extra para servir el dato, el rendimiento baja globalmente. Un ciclo extra en el primer nivel hace perder más que añadir 3 a la latencia de L2 (como era de esperar).

Como se puede apreciar, bancos de 2 Kbytes y latencia 2 ciclos tienen un IPC mayor que cualquier punto de diseño de 3 ciclos. Incrementar la capacidad del banco no es una buena opción si para ello debemos incrementar su latencia.

En la Figura 5.10 se muestra de nuevo la Figura 1.2 del Capítulo 1 utilizada para motivar el trabajo. En la figura se muesta el rendimiento (IPC) del mismo procesador pero con el primer nivel de cache multipuerto. El *eje X* muestra la capacidad de la cache entre 8 Kbytes y 64 Kbytes. Las líneas muestran el rendimiento al ir variando la latencia (*lat*) o el número de puertos de la cache (*P*). Recordemos que la línea superior muestra el resultado de una cache L1 multipuerto de dos ciclos de latencia y cuatro puertos de acceso (*2lat-4P*).

Añadimos los resultados de los mejoreres puntos de diseño multibanco con cuatro bancos y latencia de cache L1 de dos ciclos (*best L1=2* con *Word* para bancos de 2 Kbytes y *Word-2R* para bancos entre 4 Kbytes y 16 Kbytes). Los resultados, mostrados con una cruz en la Figura 5.10, han sido alineados adecuadamente por capacidad efectiva (4 bancos de 2 Kbytes = 8 Kbytes, etc.). Nótese que aún teniendo la misma capacidad efectiva, en un caso hablamos de

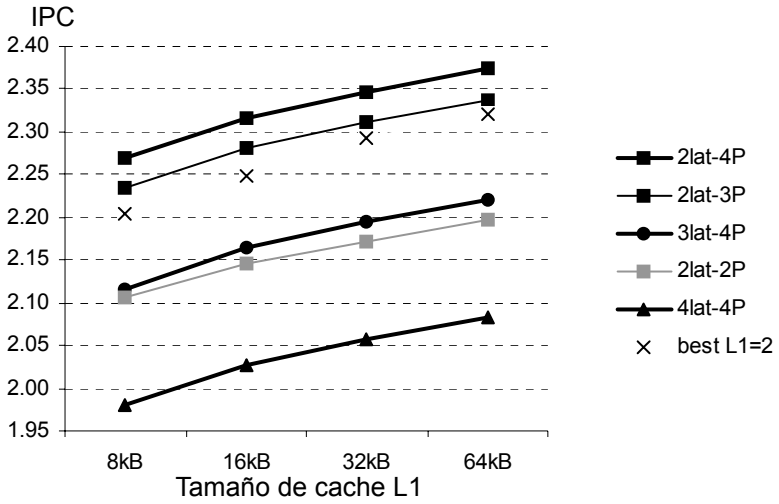

Figura 5.10 IPC al ir variando capacidad, latencia y número de puertos de una cache L1 multipuerto sobre el procesador *8-inicio* (Capítulo 1). Hemos añadido los resultados de los mejores puntos de diseño multibanco con cuatro bancos y dos ciclos de latencia (*best L1=2*).

cuatro bancos de 2 Kbytes y un sólo puerto y en el otro caso de una cache de 8 Kbytes y cuatro puertos. A ésto tenemos que añadir la ventaja de multibanco para colocar los bancos cerca de las unidades funcionales generadoras de la dirección y consumidoras del dato.

Word sobre 4 bancos de 2 Kbytes con 2 ciclos de latencia tiene un IPC mayor que la mayoría de los esquemas 4-multipuerto presentados en la introducción con latencias de 3 y 4 ciclos.

5.6 Conclusiones

Al enfrentarse al diseño de un primer nivel de cache multibanco, hay dos decisiones principales relacionadas con la gestión de contenidos: la política de distribución y el grado de replicación. Elegir un determinado punto de diseño establece el número de copias permitidas e identifica a los bancos que deben acojerlas. En general, las políticas de distribución de contenidos que fraccionan el flujo de datos en función de la dirección de memoria reparten más uniformemente los contenidos y obtienen las menores tasas de fallos de cache y las menores tasas de contención en los puertos de inicio a memoria. Nuestros resultados ponen de manifiesto que la mejor política de distribución es Entrelazado por Palabra.

El grado óptimo de replicación está sujeto a la capacidad del banco. En bancos pequeños de 2 Kbytes lo ideal es no replicar. Replicación Parcial sobre dos bancos conviene a tamaños de entre 4 Kbytes y 8 Kbytes. Con bancos mayores, el rendimiento de Replicación Total es equivalente a la del mejor gestor de contenidos pero con menor complejidad y coste al poder prescindir del predictor de banco.

En el espacio de diseño estudiado (bancos entre 2 y 16 Kbytes con latencias de 2 y 3 ciclos), incrementar la capacidad del banco nunca es una buena opción si involucra un aumento de latencia.

CAPÍTULO 6

Diseño del *Store Buffer* para Caches L1 Multibanco

El Store Buffer (STB) es un componente crítico en la ruta de datos de un procesador superscalar fuera de orden, ya que el suministro de datos desde él puede afectar al tiempo de ciclo.

En este capítulo nos centraremos en como diseñar el STB en sistemas con memoria cache de primer nivel multibanco. El objetivo es suministrar datos de stores en vuelo a loads dependientes a latencia del primer nivel mientras se mantiene un número elevado de stores en vuelo.

6.1 Introducción

En procesadores fuera de orden, para poder soportar excepciones precisas, las instrucciones *store* actualizan la memoria en orden de programa al consolidar. Para acelerar la ejecución del programa, se añaden dos funcionalidades denominadas: **adelantamiento** de *loads* independientes de *stores* no consolidados y **suministro de dato** de *store* no consolidado a un *load* que accede a la misma dirección (*data forwarding*).

El *adelantamiento* permite que una instrucción *load* que accede a una dirección de memoria distinta de las direcciones de los *stores* previos en orden de programa no consolidados obtenga el dato de la cache.

El *suministro de dato* permite que un *load* en vuelo obtenga el dato del *store* previo en orden de programa a la misma dirección de memoria que aún no ha consolidado, sin esperar a que el *store* del que depende haya consolidado.

Un *load* a una determinada dirección obtendrá el dato del *store* precedente en orden de programa a la misma dirección o de la memoria cache. En los dos casos, las instrucciones dependientes del load inician la ejecución especulativamente a tiempo para capturar el dato de la red de cortocircuitos.

Para acometer las distintas funcionalidades, se utiliza una estructura denominada *Store Buffer* (*STB*). El STB almacena la dirección y el dato de los *stores* hasta que consolidan. Por otro lado, con cierta lógica adicional y búsqueda por contenidos (*CAM*), se fuerza el orden a memoria *store-load* y se realiza el suministro de datos.

Para incrementar el paralelismo en el flujo de accesos a memoria se utiliza un predictor de independencia *store-load*. Cuando se predice que un *load* depende de uno o más *stores*, la IQ no inicia la ejecucción del *load* hasta que todos los *stores* de los que se ha predicho que depende han iniciado la ejecucción. Para determinar errores de predicción se utiliza una *cola de loads* denominada LDB. En el LDB se almacenan todos los *loads* no consolidados. Al ejecutar un *store* se accede al LDB y se comprueba si la dirección de memoria del *store* coincide con la de un *load* más joven en orden de programa. En caso de coincidir, se ha producido un error de especulación y hay que recuperarse. Esta recuperación se hace desde el RIB de forma no selectiva.

El STB es un componente crítico en los procesadores fuera de orden, ya que el suministro de datos puede afectar al tiempo de ciclo [AkRS03]. El circuito que identifica y suministra los datos es complejo y conforme aumenta el tamaño del STB se incrementa su retardo. Si las tendencias actuales de incremento de frecuencia de reloj, incremento de ancho de inicio de ejecución y/o aumento del número de instrucciones en vuelo se mantienen, el problema de latencia impuesto por el STB se agravará [AkRS03].

Una organización del primer nivel de datos multibanco requiere un STB distribuido, de forma que cada banco del STB esté acoplado a un banco de cache. Victor Zyuban y P.M. Kogge propusieron utilizar un STB multibanco de un sólo nivel en [ZyKo01]. En su trabajo, al emitir los *stores* a las colas de instrucciones (*dispatch*), se les asignaba una entrada en cada uno de los STB. Como el orden *store-load* y el suministro de datos se realiza en los bancos de STB, los *stores* no pueden ser eliminados del STB hasta la consolidación. La

emisión de instrucciones debe pararse si el STB está lleno. Tal y como mostraremos más adelante, asignar las entradas del STB al emitir y liberarlas al consolidar puede limitar significativamente el rendimiento del procesador.

En este capítulo nos centraremos en el diseño de un STB distribuidos con dos niveles adecuado para caches multibanco donde los bancos están integrados en caminos a memoria seccionados. Proponemos un primer nivel distribuido de pequeños STB (STB1) especializados en el suministro especulativo de datos, y un segundo nivel centralizado de STB (STB2) especializado en forzar el orden de acceso a memoria y mantener los *stores* hasta que consolidan. Adicionalmente, el STB2 es responsable de comprobar el suministro de datos realizado especulativamente desde STB1.

Para reducir el tamaño de STB1, retardamos la asignación de las entradas hasta la ejecución de los *stores*, posiblemente fuera de orden, asignando y rellenando los campos a la vez (dirección y dato). También permitimos el desalojo antes de que el *store* consolide: al ejecutar un nuevo *store*, si su banco de STB1 está lleno, se sobrescribe la nueva información en la entrada más antigua.

A la hora de emitir instrucciones, a los *stores* se les asignan entradas únicamente en STB2, de forma que la emisión sólo se para si agotamos las entradas de STB2, independientemente del tamaño de STB1. Cuando un *store* se ejecuta por el camino de memoria correcto, la dirección y el dato serán almacenados tanto en STB1 como en STB2.

La organización en dos niveles nos permite simplificar el diseño de STB1 y de STB2 sin pérdidas sustanciales de rendimiento: STB1 no utiliza la edad relativa para seleccionar el *store* a suministrar, y STB2 almacena todos los *stores* hasta que consolidan, pero no suministra datos.

Como STB1 no almacena todos los *stores* pendientes de consolidar, cualquier dato suministrado a un *load* por el primer nivel de la jerarquía es especulativo y debe ser verificado por STB2 (**comprobación de suministro**). Por ejemplo, como un *store* puede ser reemplazado del STB1 antes de consolidar, un *load* dependiente puede llegar al STB1 cuando el *store* ya ha salido, por lo que nadie le suministraría, o incluso le podría suministrar otro *store* distinto a la misma dirección. Los **errores de especulación de suministro**, son detectadas por STB2 cuando el *load* llega y accede a STB2.

En caso de error de especulación de suministro necesitaremos recuperarnos. La recuperación desde la cola de instrucciones (*IQ*), como es habitual en otros

errores de predicción de latencia, incrementa la presión en la IQ puesto que los *loads* y sus instrucciones dependientes deben ser retenidas en la IQ hasta que STB2 realice la comprobación de suministro.

Sin embargo, el escaso número de errores de especulación de suministro que hemos medido nos permite reducir la presión de la IQ al recuperarnos desde el *buffer de instrucciones renombradas* (**RIB** - *Renamed Instruction Buffer*). El RIB almacena todas las instrucciones en vuelo una vez decodificadas y renombradas. Esta política de recuperación de errores de especulación tiene una penalización mayor, pero permite que las instrucciones sean extraídas antes de la IQ. Concretamente las instrucciones serán extraídas tras el suministro del dato desde L1.

Finalmente, para reducir la contención en los puertos de inicio a memoria de la IQ, utilizaremos un predictor de no-suministro para los *stores*. Los *stores* con la marca de no-suministro son iniciados por cualquier puerto de inicio a memoria libre, sin entrar en STB1 y almacenándose únicamente en STB2. De esta forma se incrementa el ancho de banda efectivo de inicio de instrucciones. Este predictor es especialmente útil en sistemas con cache multibanco donde los bancos almacenan contenidos replicados.

Este capítulo se descompone en las siguientes secciones: en primer lugar veremos las limitaciones por tamaño y/o latencia de un STB de un sólo nivel y propondremos el sistema con STB en dos niveles, todo ello en la sección 6.2. A continuación en la sección 6.3 veremos los resultados de las diferentes propuestas. Comprobaremos el funcionamiento del STB en dos niveles sobre caches parciarmente replicadas (sección 6.4). Finalizaremos con trabajos relacionados (sección 6.5) y las conclusiones (sección 6.6).

6.2 STB en dos niveles en caches de primer nivel multibanco

En primer lugar resumiremos el modelo de procesador y del primer nivel de la jerarquía de memoria. A continuación pondremos de manifiesto el comportamiento y limitaciones de un STB distribuido de un sólo nivel. Por último, introduciremos las alternativas de diseño para superar estas limitaciones.

6.2.1 Modelo de procesador y L1 multibanco

A modo de resumen; modelamos el procesador fuera de orden superscalar de grado 8 (*8-inicio*), con un ROB de 256 entradas, con hasta 128 *loads* y 128 *stores* en vuelo, cuatro bancos de cache de 8 Kbytes y dos niveles adicionales de cache de 256 Kbytes y 2 Mbytes respectivamente. El resto de parámetros y modelo del procesador se pueden consultar en el Capítulo 2.

El primer nivel de cache está compuesto por 4 bancos entrelazados por dirección a nivel de línea de cache (32B). En el capítulo anterior constatamos que el *entrelazado por palabra* obtiene mejores resultados que línea. Entrelazar por palabra requiere replicar el campo de etiquetas o hacerlo multipuerto. *Entrelazado por línea* es el más usual en la bibliografía y las diferencias en IPC no son muy grandes. Utilizamos el predictor *enhanced skewed binary predictor* ya comentado con 9 Kbytes por bit a predecir (8Kentradas de contadores de 3 bits y 9 bits de historia). Los *loads* clasificados por el predictor como de baja confianza son replicados selectivamente de forma conservadora (RD - *replicación selectiva en emisión*). En error de especulación de latencia utilizaremos la *Recuperación en Cadena*.

En nuestro modelo, no se inicia la ejecución de *stores* mientras no estén disponibles los registros necesarios tanto para calcular la dirección como el dato. Los *stores* comparten puertos de inicio de ejecución con el resto de las instrucciones *loads* y enteros. Al igual que los *loads*, los *stores* entran en la IQ con la predicción del puerto por el que deben ser iniciados y permanecerán en la IQ hasta la validación del banco correcto en la etapa correspondiente.

Utilizamos un predictor de dependencia de memoria entre *stores* y *loads*. De esta forma los *loads* y sus instrucciones dependientes pueden comenzar la ejecución sin esperar la resolución de las direcciones de todos los *stores* previos. Utilizamos el predictor de desambiguación *Store-Sets* tal y como se describe en [ChEm98], (*Tabla SSI* 4K-entradas y *tabla LFS* de 128-entradas).

El orden predicho entre *stores* y *loads* es gestionado por la IQ. Si se ha predicho una dependencia entre un *store* y un *load*, la IQ retarda el inicio del *load* hasta el inicio del *store*. Los errores de especulación de orden son descubiertos al ejecutar el *store* accediendo al LDB. La ejecución del *store* se puede haber realizado posiblemente muchos ciclos después de que el *load* y sus instrucciones dependientes hayan abandonado la IQ. En caso de desorden nos recuperaremos desde el RIB a partir del *load* mal predicho y de forma no selectiva. La Figura 6.1 muestra el camino de datos simplificado.

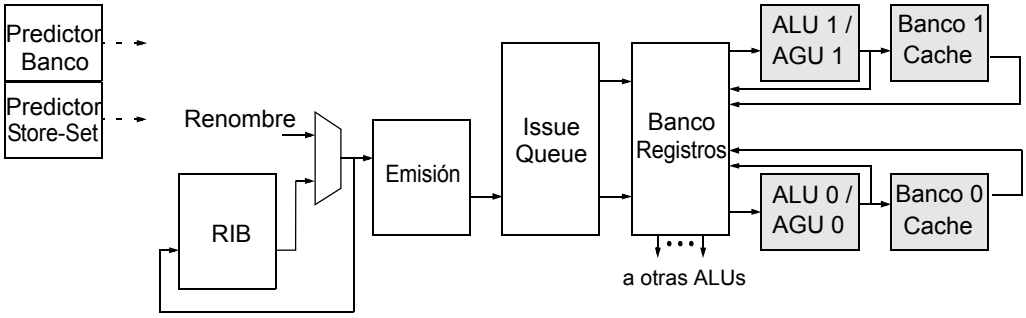

Figura 6.1 Camino de datos simplificado mostrando 2 bancos

6.2.2 Store Buffer

El *Store Buffer* (STB) es una estructura de datos que cumple una doble función. En primer lugar y para que un procesador fuera de orden pueda soportar excepciones precisas, las instrucciones *store* deben actualizar la memoria en orden de programa al consolidar. Para ello el *Store Buffer* almacena la dirección de acceso a memoria y el dato a guardar por el *store* una vez ejecutado, hasta que se sabe a ciencia cierta que se puede actualizar la memoria (cache) y el *store* pasa al estado de consolidado. La Figura 6.2 muestra la temporización del STB.

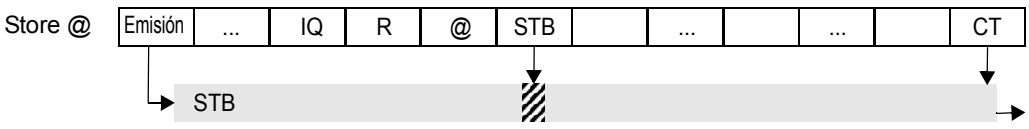

Figura 6.2 Camino de Datos de las instruciones *Store*. Asignación de entradas en el STB en *emisión*, actualización en STB en *ejecucción*, y liberación al *consolidar*.

El número de entradas del STB determina el número total de *stores* en vuelo ya que comunmente las entradas se reservan al *emitir* y se liberan al *consolidar* el *store*. Si todas las entradas del STB están asignadas se para la *emisión* de instrucciones.

La segunda función del *Store Buffer* permite accelerar la ejecución garantizando que se respetan las dependencias de acceso a memoria entre *stores* y *loads* a la misma dirección. En principio una instrucción *load* debería esperar a que todos los *stores* previos en orden de programa hayan actualizado la cache antes de poder acceder a ella.

La mayoria de los *loads* (aprox. 90% en nuestro modelo) son independientes de *stores* previos, esto es, acceden a direcciones distintas. Para mejorar el rendimiento se permite que estos *loads* ***adelanten*** el acceso a la cache sin tener que esperar a que los *stores* previos independientes consoliden. Al STB se le añade una búsqueda asociativa por contenidos (CAM). Al ejecutar el *load* se compara su dirección con la de todos los *stores* previos en el STB. Si no se encuentran coincidencias, el *load* accede a la cache.

Aproximadamente el 10% de instrucciones *load* encuentran uno o más *stores* a la misma dirección, previos en orden de programa, y sin consolidar. Para mejorar las prestaciones, se le añade al STB una funcionalidad denominada ***suministro de datos*** (*data forwarding*). El STB es consultado cuando se accede a la cache. El dato suministrado por el *load* a las instrucciones dependientes provendrá del STB o de la cache. Cuando existe coincidencia entre la dirección de memoria del *load* y de algún *store* previo en orden de programa, se suministra el dato desde el STB, en caso contrario desde la cache.

En la Figura 6.3 se muestra un esquema del STB compuesto del vector de direcciones (@) y de datos junto con los comparadores de dirección y la lógica de selección por edad.

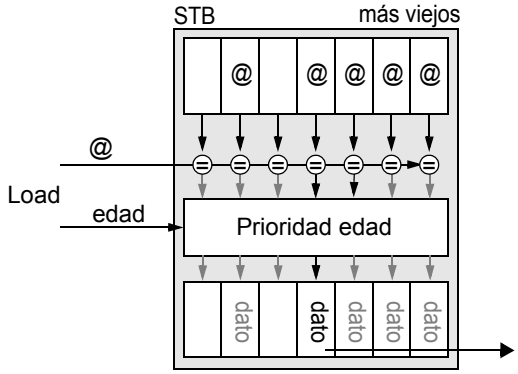

Figura 6.3 Lógica de suministro. Comparar dirección del *Load* contra la de todos los stores del STB. De los que coinciden, seleccionar el dato del *store* más joven de entre los más viejos que el *load*.

6.2.3 Motivación

Victor Zyuban y P.M Kogge en [ZyKo01] proponen por primer vez distribuir el STB colocando bancos de STB independientes y de un sólo puerto de acceso cerca de los bancos de la cache. Cada banco de STB es responsable de comprobar el rango de direcciones mapeado al banco de cache asociado. Por lo tanto el *suministro de datos* y el *control de dependencias store-load* se realiza localmente en cada banco de STB. Nosotros lo denominaremos STB distribuido de un sólo nivel (1L).

Con este enfoque, al emitir un *store*, debido a que no se conoce la dirección de memoria y por tanto el banco destino, se le asigna una entrada al *store* en cada uno de los bancos de STB. Cuando el *store* se ejecute sobre el banco correcto, la entrada correspondiente en ese banco y sólo en ese banco será actualizada. Conforme los *stores* van consolidando se irán liberando las entradas en todos los bancos de STB.

El rendimiento del STB distribuido de un solo nivel depende directamente de su tamaño y de su latencia. A continuación vamos a cuantificar el impacto de estos dos parámetros en el rendimiento.

La Figura 6.4 muestra el IPC de SPECint-2K conforme el número de entradas de cada uno de los cuatro bancos de STB varía entre 4 y 128. El IPC ha sido calculado suponiendo que la latencia de acceso al STB es la misma que al banco de cache, independientemente de su tamaño.

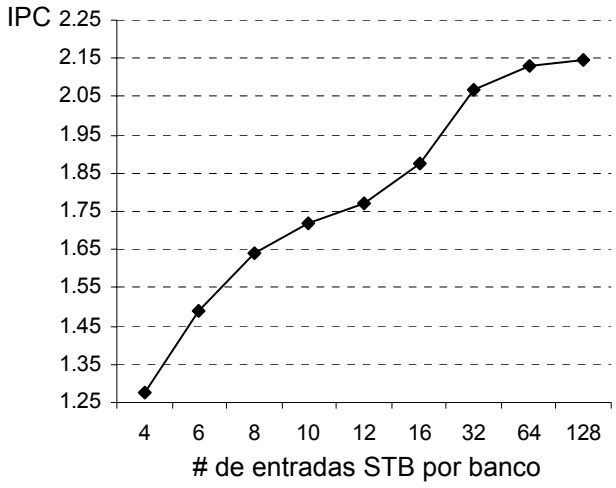

Figura 6.4 STB distribuido de un sólo nivel. Media harmónica de IPC con respecto al número de entradas por banco de STB. Procesador *8-inicio* con cuatro bancos seccionados.

Un sistema con esta distribución de STB ve limitado seriamente su rendimiento si el tamaño es muy reducido: con bancos de STB de 4-entradas el IPC disminuye un 40,5% respecto a bancos ideales de 128 entradas. A partir de 64 entradas, los bancos de STB no limitan el rendimiento, mientras en el rango 10-64 el IPC disminuye gradualmente. Por debajo de 10 entradas la pendiente de IPC es muy pronunciada.

La Figura 6.5 muestra el porcentaje de ciclos que STB está lleno y por tanto la *emisión* parada. En la Figura 6.5 observamos que con STBs de 4 entradas, cerca del 70% de los ciclos los STBs están llenos y por tanto la emisión parada. Con 128 entradas por STB, el porcentaje de ciclos que los STBs estan llenos es prácticamente cero. Observando las Figura 6.4 y Figura 6.5 se puede constatar la simetría entre ambas, poniendo de manifiesto que las diferencias en rendimiento son debidas a tener que parar la *emisión* de instrucciones cuando nos quedamos sin entradas en el STB.

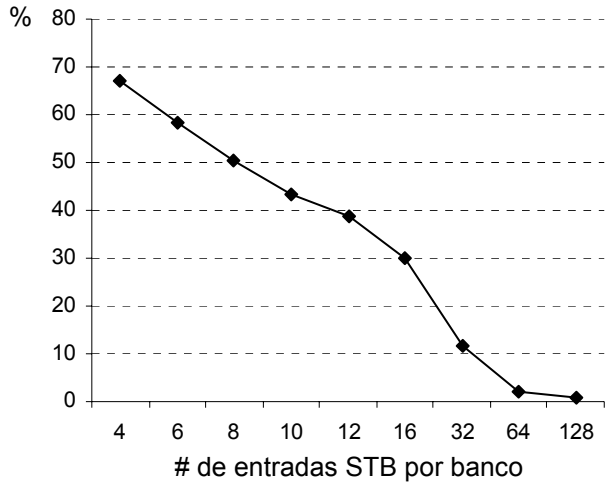

Figura 6.5 % de ciclos que el STB está lleno y por tanto la emisión parada respecto al total de ciclos de cada una de las simulaciones (en función del tamaño del STB)

El circuito que identifica y suministra los datos es complejo y su latencia aumenta en relación logarítmica al tamaño [AkRS03][CaLi04]. Ello puede dar lugar a que el tiempo de acceso al STB sea mayor que el tiempo de acierto en cache. En ese caso, la planificación de las instrucciones *loads* se complica al tener dos latencias de suministro distintas en función de la fuente. Cada *load* debe ir marcado con una latencia predicha. La asignación de los recursos y el despertar especulativo de las instrucciones dependientes se realiza en función de esa latencia predicha complicando la gestión en la IQ. Ejemplos de recursos son la red de cortocircuitos (*bypass network*) o los puertos de escritura al banco de registros.

La Figura 6.6 muestra la ejecución de un *load* de latencia corta (suministro desde cache) y uno de latencia larga (suministro desde el STB). Un *load* de latencia corta, Figura 6.6.a, accederá a la cache y al STB, planificando los recursos y el despertar de las instrucciones dependientes a la latencia de suministro desde la cache. Sin embargo, para poder recuperarnos en caso de error de especulación de suministro, el *load* y las dependientes deberán permanecer en la IQ hasta comprobar que no existen *stores* previos en el STB a la misma dirección. Si pasados "n" ciclos se descubre un acierto en el STB, como el dato suministrado es erróneo, deberemos anular y volver a ejecutar al *load* y a sus dependientes, esta vez con latencia larga para poder obtener el dato desde el STB.

Las instrucciones *load* planificadas como de latencia larga, Figura 6.6.b, deberán acceder a la cache y arrastrar el dato leído por si finalmente no acertamos en el STB (por haberse consolidado al *store* suministrador). El despertar y los recursos se planifican a latencia de suministro de STB. Como las instrucciones *load* de latencia corta y larga comparten recursos (p.e. puerto de escritura en el banco de registros), la ejecución de una instrucción de latencia larga inhibe la ejecución de una de corta que hubiera coincidido con la primera en el uso de estos recursos.

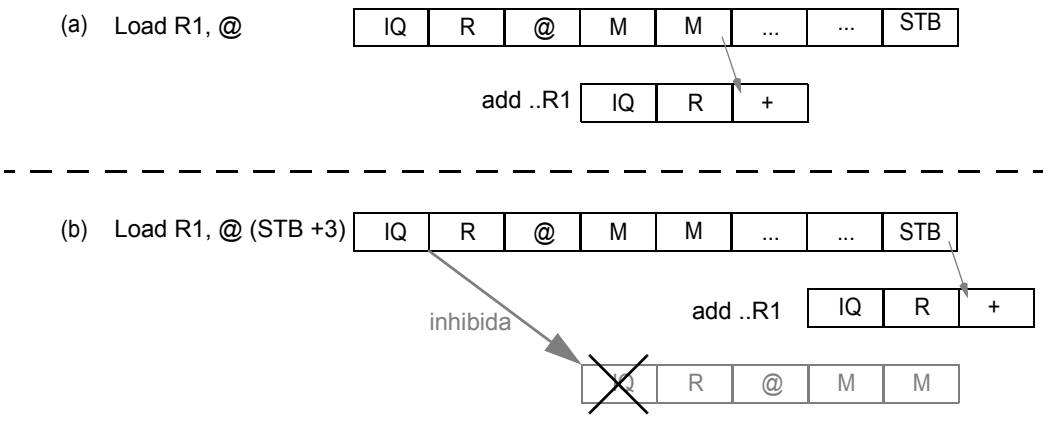

Figura 6.6 Ejecución de un *load* de latencia corta (a) y de un *load* de latencia larga (b).

Para clarificar la importancia de tener dos latencias distintas, vamos a ir incrementando la latencia del banco de STB en relación a la latencia del banco de cache. La Figura 6.7 muestra el resultado de esta simulación para bancos de STB de 32 entradas al incrementar la latencia desde 0 a 5 ciclos extra.

Simulamos dos modelos. El primero predice a ciegas siempre latencia corta de acierto en cache (barras negras). El otro modelo hace uso de un predictor para marcar las diferentes latencias de los *loads*: a) acierto en cache, b) suministro desde el STB. Como predictor implementamos un bimodal con 4K-entradas (barras grises).

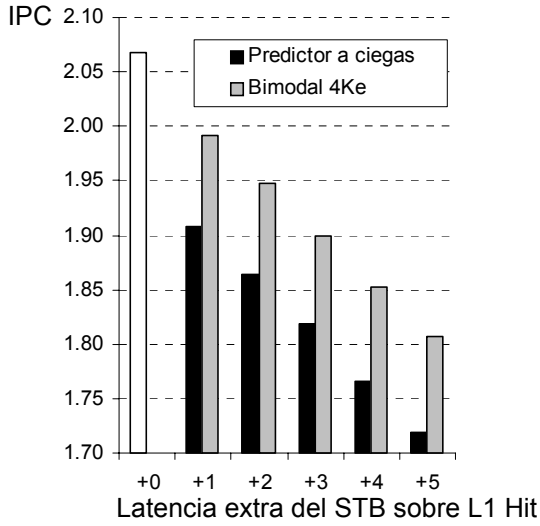

Figura 6.7 Rendimiento del STB distribuido en un sólo nivel con bancos de STB de 32 entradas. La latencia del acceso al STB se incrementa sobre la de los bancos de cache (+0 barra hueca) hasta 5 ciclos más (+5). Las barras negras muestran IPC con un predictor a ciegas (suministro desde cache), mientras las barras grises hacen uso de un predictor bimodal de 4K entradas para predecir la fuente del dato.

Como se puede observar, si los *loads* necesitan tan solo un ciclo más para llegar, acceder y suministrar el dato desde el banco de STB, la pérdida de IPC resultante es cercana al 8%. Si usamos el predictor bimodal, la degradación del IPC gira alrededor del 3% por cada ciclo adicional. El suministro *store-load* es muy predecible ya que en todas las configuraciones probadas el rendimiento de un sistema con un predictor oráculo de fuente del suministro (latencia) supera a uno con el predictor bimodal en menos de 1%.

La utilización del STB es clave para saber como incrementar el rendimiento en el compromiso tamaño-latencia en el entorno descrito hasta ahora. La Figura 6.8 muestra de forma indirecta la utilización del STB mostrando la vida media de un *store* en el STB.

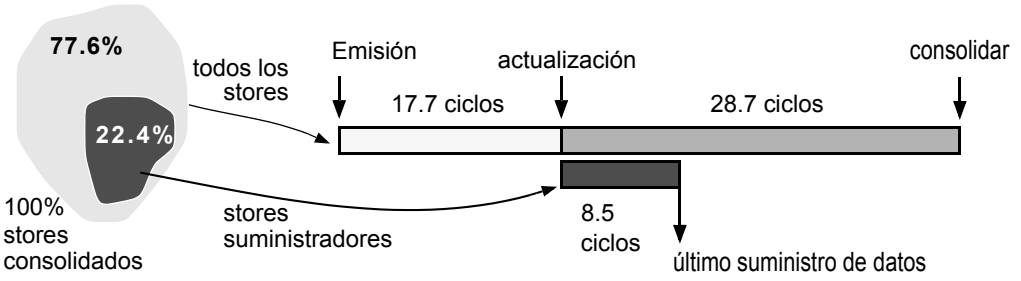

Figura 6.8 Vida media de un *store* en el STB (STB distribuido en un sólo nivel con bancos de STB de 128 entradas.

En media, cada *store* permanece 46,4 ciclos en el STB: 17,7 ciclos desde que fue emitido hasta que se ejecuta y 28.7 ciclos desde que se ejecuta hasta que consolida. Sólo el 22,4% de los *stores* suministra ese dato a *loads* más jóvenes.

Desde el punto de vista del suministro, durante los ciclos desde *emisión* a la *actualización*, las entradas del STB permanecen vacías y no son útiles para suministrar.

Los *stores* tienden a suministrar el dato pronto tras ejecutarse; en media, el último suministro de un *store* desde el STB ("*último uso*") ocurre 8.5 ciclos tras la ejecución.

La Figura 6.9 muestra el porcentaje acumulado de stores suministradores cuyo último suministro se produce por debajo de una cierta distancia en ciclos desde su ejecución (eje x). Podemos observar que el 24,09% de los *stores* que suministran lo hacen, por primera y última vez, al ciclo siguiente de su ejecución. Existe también un pequeño porcentaje de *stores* suministradores (3,5%) que lo realizan a más de 32 ciclos. El 75% del suministro se realiza en los primeros 7 ciclos, y el 90% entre los primeros 14 ciclos.

Así pues, desde el punto de vista del suministro desde el STB, hemos constatado que se utilizan muy pocas entradas, se asignan demasiado pronto (al *emitir* el *store*) y se liberan demasiado tarde (al *consolidar*).

Resumiendo, podemos concluir que se necesitan bancos de STB grandes para no tener que parar la emisión de instrucciones, pero que latencias de STB mayores que la del banco de cache perjudican al rendimiento y complican la gestión. Y sobre todo desde la perspectiva del suministro, los bancos de STB están altamente infrautilizados.

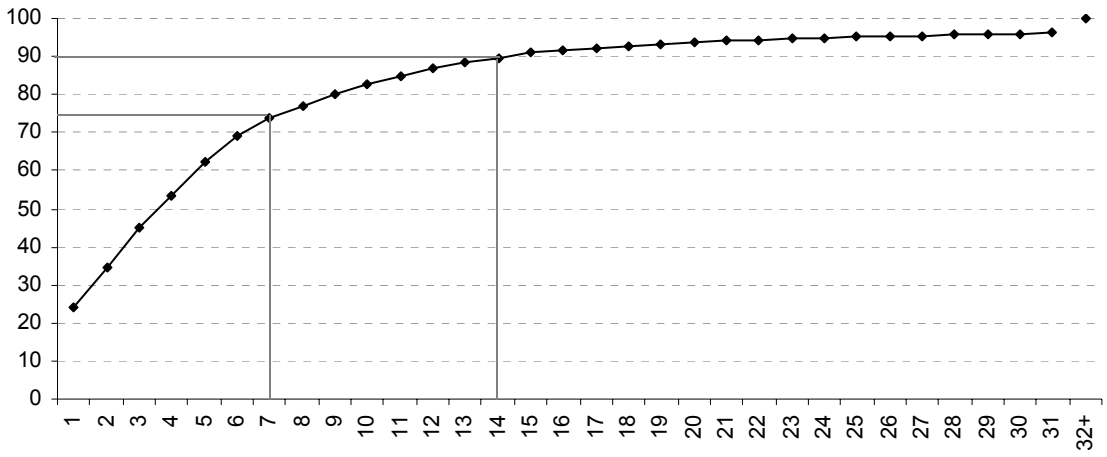

Figura 6.9 Porcentaje acumulado de *stores* suministradores cuyo último suministro se produce por debajo de una cierta distancia en ciclos desde su ejecución (*eje x*).

En el apéndice B, en la sección b.2 se detalla por programa el origen de los datos suministrados y el comportamiento de los *stores* suministradores.

6.2.4 Guías para el diseño del STB en dos niveles

Nuestro objetivo es mantener la latencia de acceso al banco de STB igual a la latencia de acceso al banco de cache. Sin embargo, asignar y liberar las entradas a los *stores* al emitir y al consolidar, respectivamente, requiere bancos de STB grandes (y lentos) para no parar constantemente la emisión de instrucciones.

Para superar estas limitaciones, proponemos un diseño de STB en dos niveles con políticas de asignación y actualización específicas para cada nivel (Figura 6.10). El primer nivel de STB (STB1) está distribuido y cada banco de STB tiene un único puerto de acceso. El segundo nivel de STB (STB2) está centralizado y es multipuerto, pero está situado fuera del camino crítico.

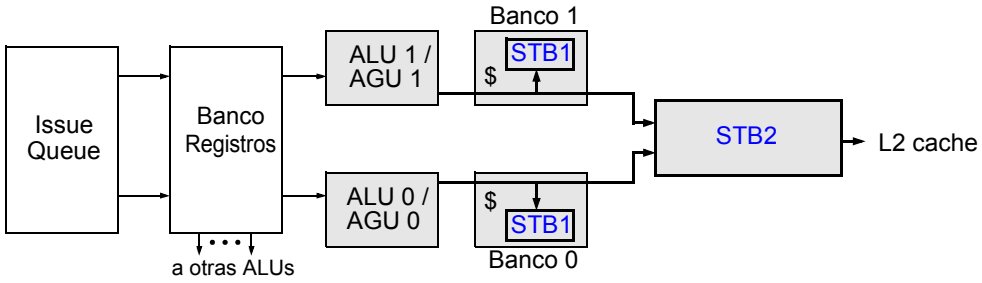

Figura 6.10 Camino de datos simplificado de ejemplo del STB distribuido en dos niveles

Los bancos de STB1 y de cache tienen el mismo tiempo de acceso. Inicialmente suponemos que la latencia del STB2 es de cinco ciclos extra (sobre suministro de L1), en la sección de resultados evaluaremos la sensibilidad a este parámetro.

Un STB en dos niveles permite desacoplar las diferentes tareas realizadas por el STB de un solo nivel. El STB1 sólo se preocupa del suministro especulativo de datos. El STB2 mantiene el orden entre *stores* y *loads*, suministra a velocidad de STB2 (en los casos de error por parte de STB1), y actualiza la memoria en orden de programa.

Al emitir instrucciones, se asignan entradas a los *stores* únicamente en STB2, y permanecerán asignadas hasta que el *store* consolide. Parar la emisión por falta de entradas en el STB dependerá únicamente del tamaño del STB2, no del tamaño del STB1.

A continuación describiremos diversas líneas maestras en el diseño del sistema de STB distribuido multinivel. Primero, explicaremos como sacar provecho del comportamiento observado en el suministro de datos por los *stores*. En segundo lugar, analizaremos la desventaja de la recuperación en caso de error de especulación de suministro desde la IQ, al igual que otros errores de especulación de latencia, y propondremos una solución. En tercer lugar, sugeriremos dos simplificaciones que permiten reducir la complejidad del STB1 y del STB2 sin pérdidas importantes de rendimiento. Y por último, para reducir la contención en los puertos de inicio a memoria de la IQ, propondremos identificar a los *stores* no suministradores y enviarlos directamente al STB2.

6.2.5 Políticas de asignación en STB1

Para reducir el número de ciclos que una entrada está asignada a un determinado *store*, seguimos las siguientes pautas:

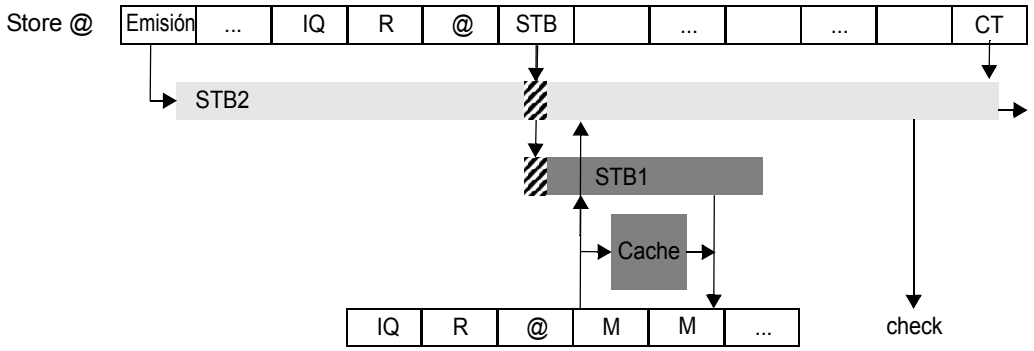

Figura 6.11 Camino de datos y gestión del STB en dos niveles.

Retardar la asignación hasta que el *store* alcance la etapa de ejecución (Figura 6.11). Ningún *store* desperdicia entrada en el STB1 hasta que no se dispone del dato. Tal y como mostramos en la Figura 6.8 esta política permite acortar el tiempo de vida de un *store* en el STB1 en aproximadamente un tercio. Como la asignación se realiza después de la comprobación de banco, cada *store* ocupa entrada en un único banco de STB1.

Liberación Temprana: Desalojar las entradas de STB1 sin esperar a la consolidación. Como la mayor parte del suministro se realiza pasados unos pocos ciclos desde la ejecución del *store* (ver de nuevo Figura 6.8), podemos desalojar las entradas anticipadamente. Este hecho inspira la siguiente política de reemplazo: a un *store* enviado a un banco de STB1 lleno, se le asigna la entrada del *store* que lleva más tiempo en ese banco. Así pues, los bancos de STB1 almacenan sólo los "n" *stores* más recientemente ejecutados.

En el diseño propuesto en dos niveles, un *load* obtendrá el dato proveniente desde el banco de cache, el banco de STB1 o bien desde STB2. En cualquier caso, predeciremos a ciegas el suministro a latencia de acierto en cache para todos los *loads*.

6.2.6 Recuperación en error de predicción de latencia

Los bancos de STB1 no disponen de todos los *stores* en vuelo, cualquier dato suministrado por L1 (banco de cache y STB1) es especulativo y debe ser

verificado por STB2. Para ello, la dirección de cada *load* es comparada con todas las direcciones de los *stores* previos presentes en STB2. Si se encuentra alguna coincidencia con *stores* previos, el más cercano en orden de programa es seleccionado. Acto seguido se comprueba si el dato del *load* fue o no suministrado por STB1 o si lo fue por un *store* distinto del correcto (***comprobación del suministro***). Si la comprobación acaba encontrando un ***error de especulación de suministro*** deberemos tomar acciones correctoras de recuperación.

Las instrucciones dependientes del *load* fueron despertadas especulativamente sin esperar a la comprobación de suministro. Si la IQ es la etapa de recuperación, el *load* y todas las instrucciones dependientes iniciadas deben permanecer en la IQ hasta que STB2 complete la comprobación (pasados varios ciclos sobre la latencia predicha de acierto en cache). Por consiguiente, la presión en la IQ se incrementa y posiblemente el número de ciclos en los que la IQ está llena también, lo cual nos puede hacer perder rendimiento. Teniendo esto en cuenta y viendo el bajo número de errores de especulación de suministro, nos sugiere que la recuperación desde el RIB (al igual que en errores de especulación de orden), puede ser altamente beneficiosa si desalojamos las instrucciones de la IQ después de comprobar el suministro desde L1 (cache hit o STB1 hit). Los resultados mostrados más adelante confirmaran esta intuición.

6.2.7 Simplificaciones del diseño

Eliminar el suministro de datos de STB2. Para poder llevar a cabo el suministro de datos desde STB2 se necesitan tantos puertos de lectura al campo de datos como bancos de cache, se necesitan caminos de escritura a la red de cortocircuitos, y el planificador de la IQ debe gestionar *loads* con dos latencias distintas. En función de la latencia seleccionada, la IQ despierta especulativamente a las instrucciones dependientes del *load*. En principio el *load* es iniciado por la IQ esperando que el servicio sea realizado por STB1/L1 cache. Si STB2 descubre un error de especulación de suministro, se llevan a cabo acciones de recuperación. Se marca el *load* de forma que sea de nuevo iniciado por la IQ pero asumiendo suministro desde STB2.

Tal y como mostraremos más adelante, el suministro desde STB2 no es estrictamente necesario, y puede ser eliminado sin pérdidas importantes de rendimiento. El *load* que no puede ser servido por el primer nivel deberá esperar a la consolidación del *store* para obtener el dato de la cache. Esta simplificación del STB2 elimina complejidad al planificador de la IQ (todos los *loads* tendrán

una única latencia) y permite eliminar puertos de lectura al campo de datos de STB2 y sus correspondientes puertos de entrada a la red de cortocircuitos.

Simplificar el suministro desde STB1. Las entradas del STB1 son asignadas fuera de orden, por lo que las entradas deben ir etiquetadas con la edad de los *stores*. Entre otras cosas, la edad es necesaria para seleccionar el dato a suministrar si hay varios *stores* a la misma dirección en STB1. En cualquier caso, el suministro especulativo debe ser comprobado en STB2, ya que el *store* adecuado en orden de programa podría haber sido desalojado del STB1.

El circuito de selección en base a la edad añade complejidad y latencia al suministro de datos desde STB1. Para simplificar este circuito proponemos seleccionar la entrada a suministrar de forma circular, **sin** comprobar las edades relativas. En esta nueva propuesta, el suministro se realizará desde la entrada más recientemente asignada después de comparar las direcciones tal y como muestra la Figura 6.12. Como siempre, STB2 comprobará el suministro realizado a todos los *loads*.

Para asegurar que la ejecución progresa y un *load* no es servido constantemente por un *store* más joven, después de un error de especulación de suministro y al iniciar la recuperación desde el RIB, se purgarán todas las entradas de *stores* más jóvenes que el *load* y se marcará al *load* para que al reejecutarse no le suministre el STB1.

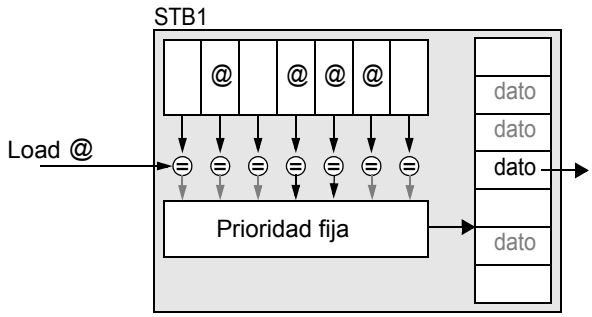

Figura 6.12 Lógica de suministro. Comparar dirección del *Load* contra la de todos los *stores* del STB1. De los que coinciden, seleccionar el dato del *store* que entró más tarde. El STB2 comprobará si el suministro fué o no acertado.

La edad relativa de las instrucciones se seguirá usando, pero siempre fuera del camino crítico de suministro. En concreto la edad la utilizamos para: purgar las entradas de STB1 en errores de especulación en saltos o suministro y sobre todo para etiquetar el suministro desde STB1 para que STB2 pueda constatar si fue bien realizado.

Como las entradas son asignadas fuera de orden, la purga puede provocar agujeros en STB1 (por ejemplo tras una recuperación de saltos). Compactar las entradas requiere circuiteria adicional. Nosotros hemos decidido no compactar por lo que la capacidad efectiva podría verse reducida.

Pondremos de manifiesto que, a pesar de no maximizar la capacidad del STB1 y de tener más riegos de errores de especulación de suministro, el suministro sin considerar la edad no afecta al rendimiento de forma significativa (sección 6.3.4).

6.2.8 Reducción de la contención en puertos de inicio a memoria

Como se puso de manifiesto en el capítulo anterior, la contención es un efecto a tener en cuenta en una cache multibanco. La contención aparece cuando una ráfaga de instrucciones listas para ejecutarse quieren salir todas al mismo banco. Todas las instrucciones compiten por el puerto de inicio y el rendimiento se resiente.

Sabemos que la mayor parte de los *stores* no suministran datos. Y además hemos comprobado que este comportamiento es altamente predecible. Se podría detectar a estos *stores* que no suministran y reconducirlos a través de cualquier otro puerto de inicio a memoria, saltándose STB1 y llegando directamente a STB2. Para ello proponemos utilizar un predictor bimodal que denominaremos predictor de *stores* no suministradores (NFS - *Non-Forwarding Store predictor*). Los *stores* marcados como no suministradores por el predictor NFS se inician por cualquier puerto de inicio a memoria libre, ya que todos están conectados a STB2, incrementando el ancho de banda efectivo y ayudando a reducir ráfagas. Esta técnica mejora el rendimiento del sistema base (sección 6.3.3), pero es especialmente útil en sistemas con bancos de cache replicados (sección 6.4).

6.3 Resultados

La evaluación experimental tiene cinco sub-secciones. La primera muestra las ventajas en rendimiento de las políticas propuestas de asignación / liberación junto con la cobertura del suministro *store-load* por el STB1. La recuperación desde el RIB y la supresión del suministro desde el STB2 se analiza en la segunda parte. La tercera evalúa la utilidad del predictor NFS reduciendo la contención en los puertos de inicio a memoria de la IQ. A continuación eliminaremos la comprobación de edad en el STB1. Por último mostraremos los resultados de cada una de las propuestas de forma individualizada por

programa. Todas las figuras (salvo mención explícita) muestran la media harmónica de IPC (eje Y) en función del tamaño del STB1 (eje X). El IPC mostrado ha sido calculado excluyendo al `MCF` que, al estar altamente limitado por memoria, independientemente de la configuración analizada, consigue un IPC muy bajo y constante. En cualquier caso, en la última sub-sección se muestra su comportamiento.

6.3.1 Políticas de asignación / liberación del STB1

La Figura 6.13 muestra el rendimiento de un sistema de STB distribuido en un sólo nivel (1L) y un sistema con dos niveles (2L_IQ). En el sistema uni-nivel, a los *stores* se les asigna una entrada en todos los STB al emitir las instrucciones. Esas entradas serán liberadas al consolidar. La comprobación de direcciones y el suministro a *loads* se realiza a latencia de L1. En el sistema con dos niveles (2L_IQ), las entradas del banco de STB1 se asignan en ejecución y se liberan temprano. Las instrucciones iniciadas especulativamente permanecen en IQ hasta que se realiza la comprobación en STB2, 5 ciclos después del suministro desde L1. El STB2 puede suministrar datos a latencia larga.

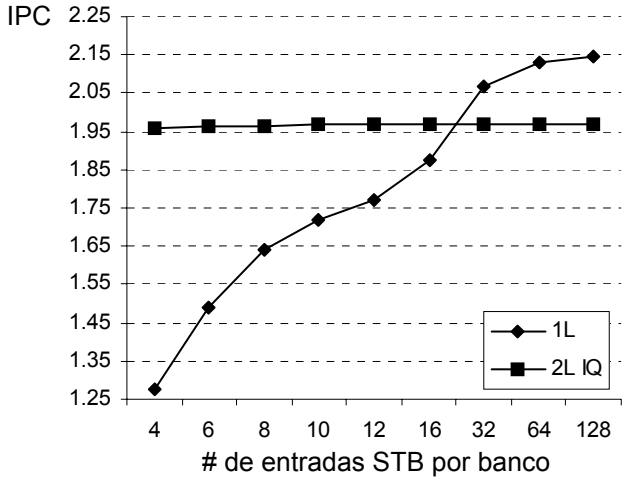

Figura 6.13 IPC para el sistema de STB distribuido en un sólo nivel (1L) con asignación desde emisión a consolidación y del sistema STB en dos niveles con asignación en ejecución y liberación anticipada (2L_IQ).

Para bancos con 16 entradas o menos, el sistema en dos niveles supera en rendimiento al sistema uni-nivel. La razón es el mejor uso de las entradas del STB1 al asignarlas cuando el dato está ya disponible y liberarlas conforme se necesita espacio. El sistema uni-nivel debe parar la emisión de instrucciones cuando el STB se llena, por contra, el sistema 2L_IQ no lo hace, ya que el

número de *stores* en vuelo depende del tamaño de STB2. Por todo ello la diferencia aumenta conforme disminuye el número de entradas.

Para entender estos resultados vamos a analizar la cobertura de los *loads* (Figura 6.14). En el sistema con dos niveles, un 100% de cobertura de los *loads* significa que todos los *loads* que deben leer de un *store* previo en vuelo lo hacen del STB1. La cobertura de los *loads* del sistema 1L se realiza simulando bancos de STB de 128 entradas y comprobando si el suministro se realiza desde alguno de los "n" últimos *stores* insertados en emisión. Con este experimento queremos poner de manifiesto la mejor utilización de las entradas del STB1 debidas a los dos factores; retardar la asignación de las entradas y la liberalización temprana.

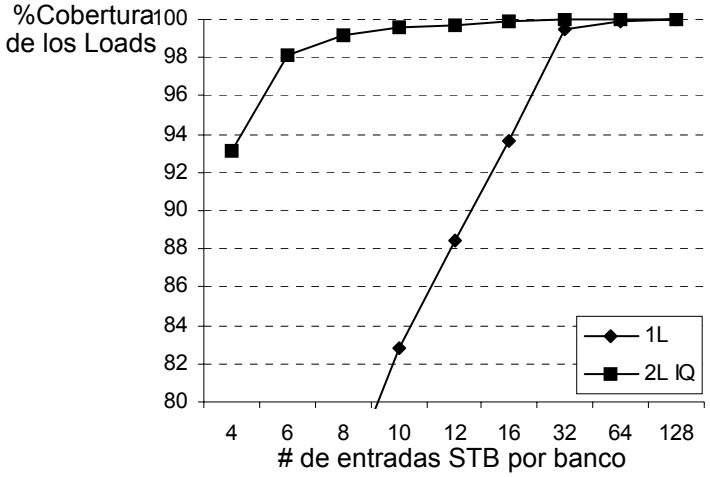

Figura 6.14 Cobertura de los *Loads* para un sistema de STB distribuido en un sólo nivel (1L) mirando los “n” últimos *stores* depachados y del sistema STB en dos niveles con asignación en ejecución y liberación anticipada (2L_IQ).

La cobertura de los *loads* desde el STB1 en el sistema 2L_IQ es muy alta, incluso para tamaños de banco muy pequeños. Por ejemplo, con bancos de tan sólo 8 entradas, menos del 1% de los *loads* que requieren suministro (0,13% del total de *loads*) no lo consiguen en STB1. Para conseguir la misma cobertura, si las entradas se asignasen en emisión se requerirían al menos 32 entradas por banco.

Por otra parte, 2L_IQ mantiene en la IQ las instrucciones iniciadas especulativamente hasta que se realiza la comprobación en STB2, y por ello la presión en la IQ aumenta. Cuando la IQ se llena, se debe parar la emisión de instrucciones perdiendo rendimiento. Esta degradación se puede cuantificar observando el IPC de los STBs de 128 entradas de nuevo en la Figura 6.13. Todos los *stores* en vuelo pueden ser almacenados en ambos sistemas en los bancos de STB y por tanto el suministro a los *loads* está asegurado. La

diferencia de IPC entre ellos (-9,2%) viene dada por la ocupación de la IQ extra sufrida por 2L_IQ. Con STB de 128 entradas, la IQ está llena un 25% de ciclos más en 2L_IQ que en 1L.

6.3.2 Aprovechando la alta cobertura de STB1

La alta cobertura de *loads* conseguida por el STB1 sugiere dos posibles optimizaciones en el sistema 2L_IQ: i) reducir la ocupación añadida en la IQ cambiando el mecanismo de recuperación, desalojando de la IQ a las instrucciones iniciadas especulativamente sin esperar la comprobación del STB2, y ii) eliminar la capacidad de suministro de datos desde STB2 para reducir la complejidad (gestión de dos latencias distintas y camino de datos para el suministro).

Recuperación desde el RIB

Como el número de *errores de especulación de suministro* es muy reducido, podemos recuperarnos en caso de error desde el RIB. Consecuentemente, el *load* y las instrucciones iniciadas especulativamente pueden abandonar la IQ tan pronto como se suministre el dato desde STB1/L1 cache. En este caso, cuando un error de especulación de suministro se descubra al comprobar en STB2, el *load* y **todas** las instrucciones en vuelo más jóvenes (dependientes o no) serán eliminadas y re-emitidas desde el RIB.

La recuperación desde el RIB tiene una penalización mucho mayor que desde la IQ, al ser no selectivo y encontrarse la etapa de recuperación a mayor distancia en ciclos de la etapa de resolución (comprobación en STB2) [BTME02]. Sin embargo, la penalización total, calculada como el número de errores por la penalización por error, es pequeña. En consecuencia, la pérdida de rendimiento al recuperarse desde el RIB puede ser menor que la pérdida de rendimiento inducida al tener que parar la emisión de instrucciones por tener la IQ llena.

La Figura 6.15 muestra el IPC para dos sistemas; uno con recuperación desde el RIB (2L_RIB) y otro con recuperación desde IQ (2L_IQ). Ambos son sistemas de STB en dos niveles con asignación en ejecución y liberación temprana de entradas en STB1 y posibilidad de suministro desde STB2.

También hemos llevado a cabo un conjunto nuevo de simulaciones para analizar la sensibilidad a la latencia de comprobación en STB2. La nueva latencia de

comprobación es de un ciclo y la Figura 6.15 muestra el rendimiento cuando la recuperación es desde el RIB o desde la IQ (2L_RIB_1 y 2L_IQ_1).

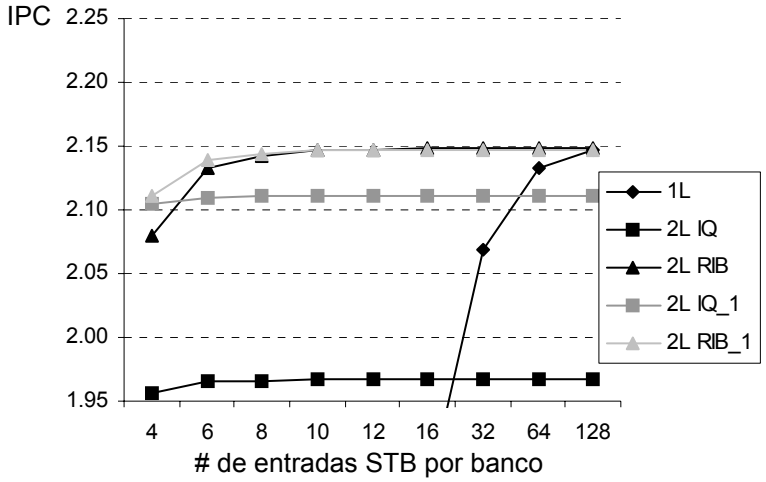

Figura 6.15 IPC compromiso recuperación desde IQ (2L_IQ) o desde el RIB (2L_RIB). Tambíen se muestra el IPC del sistema de STB distribuido en un sólo nivel (1L) y la variación en el rendimiento si la comprobación en STB2 se realiza en vez de a cinco ciclos a un sólo ciclo sobre el suministro de L1 (2L_IQ1 y 2L_RIB1).

La política de recuperación tiene mucha repercusión en el rendimiento: la recuperación desde el RIB alcanza una ventaja consistente con respecto a la recuperación desde la IQ para todos los tamaños de banco de STB1. Centrándonos en 5 ciclos de latencia de STB2 (2L_RIB, 2L_IQ), la ganancia aumenta desde un 6,3% a un 9,2% si nos movemos de 4 a 128 entradas. Por ejemplo, con bancos de STB1 de 8 entradas, al cambiar la etapa de recuperación de IQ al RIB se reduce el número de ciclos en los que la IQ está llena cerca de un 10%, mejorando un 9% el IPC. La recuperación desde el RIB es mejor que desde IQ debido a que elimina la ocupación extra de la IQ y a que el número de errores es muy pequeño.

Notese que con bancos de STB1 de 128 entradas el IPC del sistema 2L_RIB alcanza el IPC del sistema uni-nivel 1L, al no producirse recuperaciones y extraer las instrucciones de la IQ a latencia del primer nivel.

Como el número de errores de especulación de suministro es muy bajo, el rendimiento al recuperarnos desde el RIB es prácticamente independiente de la latencia del STB2 (líneas 2L_RIB y 2L_RIB1). No ocurre lo mismo si nos recuperamos desde IQ (líneas 2L_IQ y 2L_IQ1). Un incremento de la latencia de comprobación de STB2 reduce la capacidad efectiva de la IQ, ya que un número significativo de entradas de la IQ estan ocupadas por instrucciones especulativas, pendientes de la comprobación de suministro en STB2.

Eliminar la capacidad de suministro de datos desde STB2

Como el servicio desde STB2 es muy infrecuente (ver Figura 6.14) podemos eliminar la capacidad de suministro de datos desde STB2. Como ya comentamos, esto nos permitirá simplificar tanto el camino de datos como el de control. De todas formas, para cuando el *load* sea reejecutado tras un error de especulación de suministro, la mayoría de las veces el *store* ya habrá consolidado su estado en la cache de L1.

La Figura 6.16 muestra un sistema con capacidad de suministro desde STB2 (2L_RIB) y un sistema sin él (2L_RIB_NoFWD2). Ambos son sistemas con dos niveles de STB, con asignación en ejecución, liberación temprana y recuperación desde el RIB.

Las pérdidas de rendimiento varían entre un 2,2% y un 0,65% en función del tamaño de banco de STB1. En concreto, con bancos de STB1 de 8 entradas, la pérdida de IPC está en torno al 0,8%. Así pues, la capacidad de suministro desde STB2 puede ser eliminada sin pérdidas importantes de rendimiento. Las pérdidas son debidas al pequeño número de *loads* que deben esperar a que un *store* a la misma dirección consolide al no poder suministrar el dato ni desde STB1 ni desde STB2.

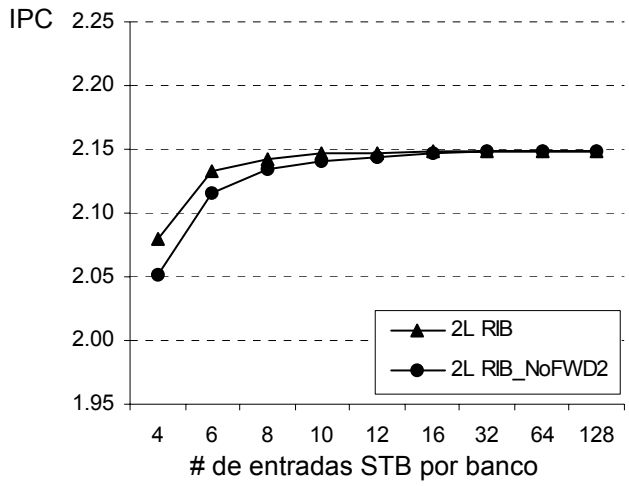

Figura 6.16 Variación en IPC entre sistemas con suministro desde STB2 (2L_RIB) y sin él (2L_RIB_NoFWD2)

6.3.3 70% de los *stores* no suministran

En esta sub-sección clasificaremos a los *stores* entre suministradores o no según el predictor NFS. Los *Stores* marcados como no-suministradores se enviarán directamente a STB2 por cualquier puerto de inicio a memoria disponible.

Existe un compromiso a la hora de diseñar el predictor. Si se reduce el número de *stores* clasificados como suministradores, reduciremos la contención en los puertos. Sin embargo el número de errores de especulación de suministro también podría aumentar debido a la mala clasificación de *stores* suministradores.

Hemos diseñado el predictor sesgado para reducir el número de *stores* mal clasificados como no-suministradores. Como predictor NFS implementamos un simple predictor bimodal con 4K contadores de 3 bits indexados por la contador de programa (PC). La Tabla 6.1 muestra algunas estadísticas del predictor. El 64% de los *stores* son clasificados como no-suministradores, reduciendo la contención en los puertos de inicio a memoria de la IQ y reduciendo la presión sobre el STB1, ya que no ocupan entradas. Sin embargo, el 0,47% de los *stores* que deberían suministrar el dato antes de consolidar, son erróneamente clasificados como no-suministradores y fuerzan un error de especulación de suministro.

Tabla 6.1 Estadisticas del predictor de *stores* no-suministradores (NFS predictor)

predice suministro		predice no-suministro	
acierta	falla	acierta	falla
25,45%	10,03%	64,05%	0,47%

La Figura 6.17 muestra la cobertura de los *loads* para sistemas con predictor NFS (2L_RIB_NoFWD2_NFSP) y sin él ((2L_RIB_NoFWD2). Podemos observar que el sistema con predictor NFS se comporta bien con bancos de STB1 pequeños con tan sólo 4 y 6 entradas. Por encima de 6 entradas, es mejor la cobertura del sistema sin predictor NFSP, debido a los mencionados 0,47% de *stores* que deberían suministrar pero son mal clasificados.

La Figura 6.18 muestra el IPC para los sistemas con y sin predictor. En todos los modelos se utilizan dos niveles de STB con asignación en ejecución y liberación temprana de entradas de STB1, recuperación desde el RIB y sin capacidad de suministro desde STB2. El sistema con predictor NFS (2L_RIB_NoFWD2_NFSP) alcanza siempre mejores rendimientos que el sistema sin él (2L_RIB_NoFWD2).

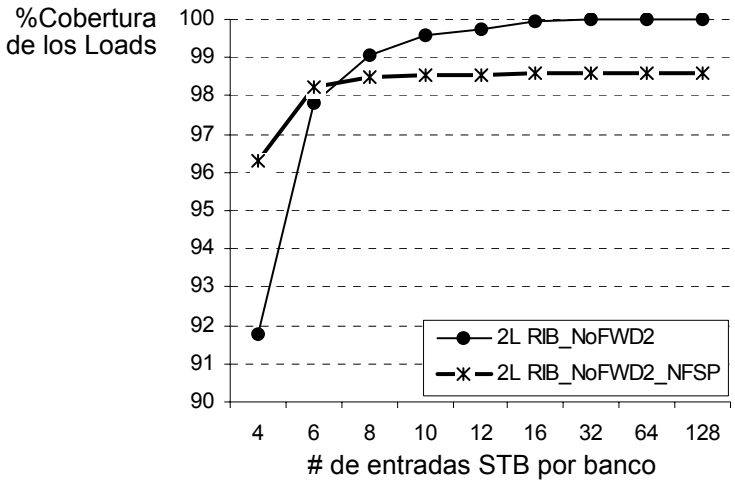

Figura 6.17 Cobertura de los *Loads* en sistemas 2L sin predictor NFS (2L_RIB_NoFWD2) y con él (2L_RIB_NoFWD2_NFSP)

El predictor NFSP reduce la contención en los puertos de inicio a memoria de la IQ, y a su vez, reduce el número de *stores* que se insertan en STB1. Para analizar los dos efectos, realizamos un experimento consistente en simular un sistema con predictor NFS (2L_RIB_NoFWD2_nfsp), pero esta vez la marca del predictor se usa únicamente para filtrar la inserción de los *stores* en el banco de STB1, y no para reducir la contención en los puertos de inicio a memoria de la IQ. Con este modelo todos los *stores* deben ser iniciados por el puerto marcado por el predictor de banco.

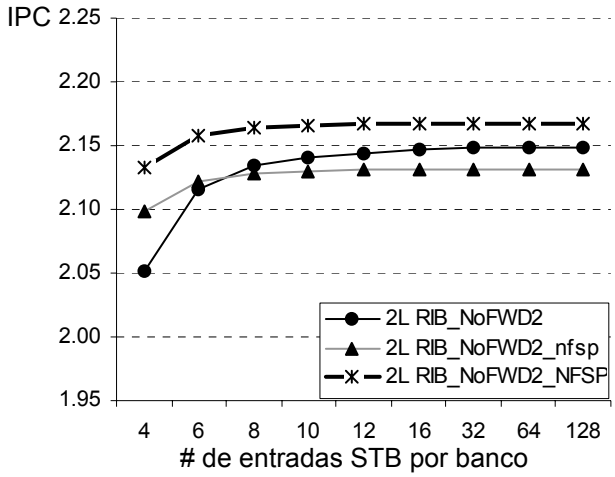

Figura 6.18 IPC sistemas de STB en dos niveles con predictor NFS (2L_RIB_NoFWD2_NFSP), sin él (2L_RIB_NoFWD2), y haciendo uso del predictor sólo para filtar la inserción en STB1 (2L_RIB_NoFWD2_nfsp).

Comparando los sistemas 2L_RIB_NoFWD2 y 2L_RIB_NoFWD2_nfsp, se observa que con bancos pequeños (4 y 6 entradas), el filtrado de la inserción en STB1 de *stores* mejora el rendimiento. A partir de ese tamaño, las instrucciones *store* mal clasificadas como no-suministradoras reducen la cobertura de *loads* y el rendimiento.

En consecuencia, la ganancia neta es debida a la reducción en la contención por los puertos de inicio; se puede ver comparando el rendimiento de los sistemas 2L_RIB_NoFWD2_NFSP y 2L_RIB_NoFWD2_nfsp. La ganancia de rendimiento debida a la reducción de contención es mayor que la pérdida causada por la clasificación errónea de *stores* suministradores.

6.3.4 Suministro de datos desde STB1 sin comprobar la edad

Hasta aquí, el STB1 comprueba explícitamente las edades relativas para suministrar datos. Para simplificar el camino de datos del suministro en STB1 se comprueba sólo la dirección de memoria y no se comprueba la edad. En el caso de detectar varias coincidencias de dirección, se suministra el dato correspondiente al *store* coincidente insertado en último lugar.

La Figura 6.19 muestra el IPC del sistema que comprueba edades (2L_RIB_NoFWD2_NFSP) y de otro sistema donde STB1 suministra sin comprobarlas (2L_RIB_NoFWD2_NFSP_NoAGE). Ambos son dos niveles de STB con asignación en ejecución y liberación temprana de entradas de STB1, recuperación desde el RIB, sin capacidad de suministro desde STB2 y con predictor NFS.

La degradación del rendimiento se sitúa en alrededor del 0,6% de IPC para todos los tamaños de banco de STB1. Por tanto, desde la perspectiva del suministro especulativo, el diseño del STB1 puede ser simplificado con pérdidas insignificantes de rendimiento. La degradación viene mayoritariamente de errores de especulación de suministro realizados por *stores* más jóvenes a la misma dirección.

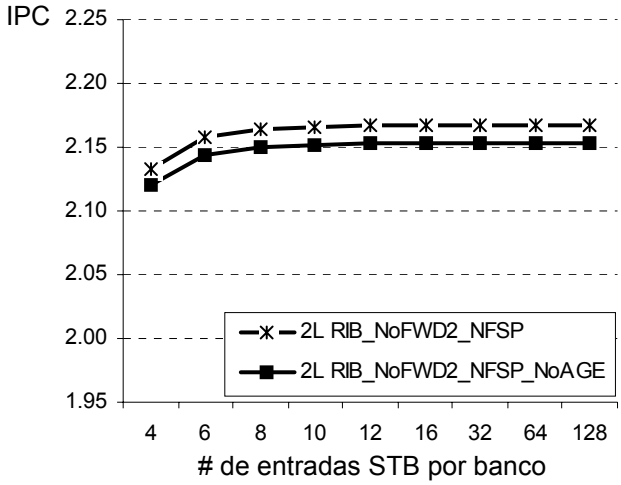

Figura 6.19 Variación de IPC en sistemas en dos niveles con comprobación explícita de edad en STB1 (2L_RIB_NoFWD2_NFSP) y sin comprobar la edad en STB1 (2L_RIB_NoFWD2_NFSP_NoAGE)

Al recuperarnos desde el RIB, las instrucciones vuelven a retomar las predicciones originales, en concreto la predicción de banco original previa a ser validada. Se da la circustancia de que si falla la predicción de banco del *load*, nos podríamos encontrar que el acceso al STB1 se realiza más tarde que un *store* más joven en orden de programa a la misma dirección (sin error de predicción de banco), causando un nuevo error de especulación de suministro al no compararse las edades relativas (Figura 6.20). Esta situación seguiría produciéndose con cada nueva recuperación. Para garantizar que la ejecución progresa es suficente con marcar al *load* que causó el error de especulación de suministro (primera instrucción a recuperar desde el RIB) para que no acceda al STB1.

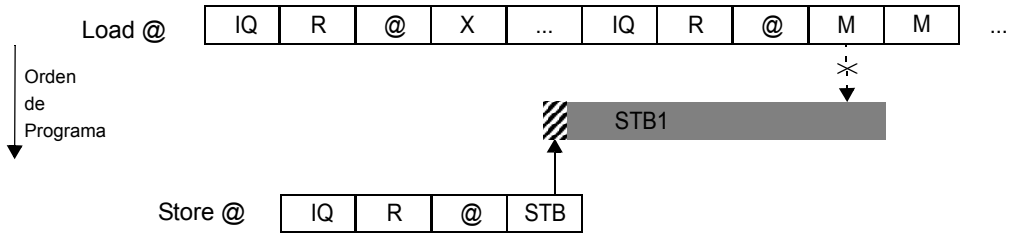

Figura 6.20 Re-ejecución de un *load* desde el RIB con error de predicción de banco, seguido de un *store* a la misma dirección sin error de banco.

6.3.5 Resultados por programa

En esta sub-sección mostramos el efecto de las distintas decisiones fijándonos en el comportamiento de cada programa. Para ello definimos dos puntos de referencia: un STB uni-nivel con asignación de *stores* en emisión en todos los bancos, liberación al consolidar, y bancos de 128 entradas a latencia de L1 (1L). El segundo punto de referencia es el mismo sistema pero usando un predictor bimodal de latencia ya que ahora el acceso al STB necesita 3 ciclos extra (1L_+3c).

Los modelos de referencia son comparados con los sistemas de dos niveles presentados hasta ahora con bancos de STB1 de 8 entradas. La primera barra corresponde a un sistema con recuperación desde IQ (2L_IQ). La segunda barra a un sistema con recuperación desde el RIB (2L_RIB). Sobre este sistema eliminamos la capacidad de suministro desde STB2 (2L_RIB_NoFWD2), añadimos el predictor NFS (2L_RIB_NoFWD2_NFSP) y finalmente STB1 suministra sin comparar edades (2L_RIB_NoFWD2_NFSP_NoAGE). Nótese que los modelos son presentados en el mismo orden en que fueron analizados en las sub-secciones previas.

Tal y como muestra la Figura 6.21 todos los programas siguen las mismas tendencias. Las políticas de asignación / liberación y recuperación son las principales fuentes de ganancia en rendimiento en todos los programas simulados. La eliminación de la capacidad de suministro desde STB2 y el suministro desde STB1 sin comparar direcciones producen pérdidas mínimas de rendimiento. En resumen, un sistema de STB en dos niveles con suministro a una latencia fija, compuesto de bancos de STB1 simples y de tan sólo 8 entradas, obtiene aproximadamente el mismo rendimiento que un sistema ideal con bancos de 128 entradas por STB.

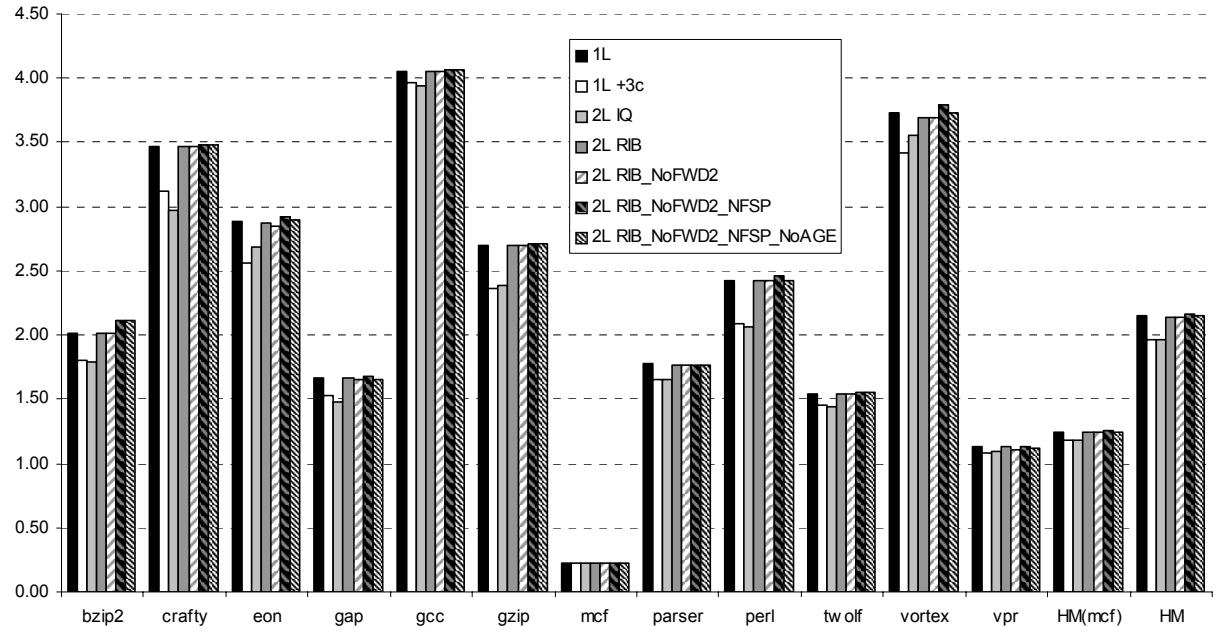

Figura 6.21 IPC por programas individuales, HM(mcf) representa la media harmónica considerando el MCF.

6.4 Dos niveles de STB en caches con replicación de datos

Como vimos en el Capítulo 5, la contención en los puertos de inicio a memoria es una de las debilidades de las caches de primer nivel multibanco. Si múltiples instrucciones de memoria listas deben ir a un determinado banco con un sólo puerto, tendrán que ser serializadas. Las caches replicadas (Capítulo 5) pueden ayudar a reducir la contención, aún disponiendo de un solo puerto a cada banco, los *loads* pueden ser servidos desde diversos bancos replicados. Es más, el replicado consigue aumentar la precisión del predictor de banco al disminuir las alternativas entre las que elegir. Sin embargo, la replicación disminuye la capacidad efectiva de la cache ya que en cada fallo de cache la línea es cargada en varios bancos simultáneamente.

En esta sección evaluaremos nuestra propuesta de STB distribuido de dos niveles sobre una configuración de cache que replica datos para aumentar el ancho de banda. En concreto lo evaluaremos sobre un esquema dos-replicado con dos conjuntos de dos bancos cada uno (los dos bancos de cada conjunto almacenan los mismos valores). La distribución del STB1 tiene un gran

inconveniente en caches replicadas: como los *loads* pueden ser servidos por los distintos bancos de un conjunto, los *stores* deben ser iniciados por varios puertos, incrementando la ocupación de los puertos y probablemente la contención. Además, la replicación aumenta el número de *stores* a guardar en cada banco (al dividir en menos caminos el flujo de *stores*)

Al igual que en las secciones previas mantenemos la configuración de cuatro puertos de inicio a memoria, cada uno de ellos conectados a un banco de L1 (un solo puerto de acceso al banco de cache y al banco de STB1). El predictor NFS presentado en la sección 6.2.8 es un buen candidato para superar el mencionado inconveniente del STB1 distribuido, permitiendo reducir tanto la contención como el número de *stores* a insertar en STB1. La reducción de la contención es incluso mayor que en el esquema no replicado ya que al usar el predictor NFS en el esquema dos-replicado, sólo los *stores* marcados como suministradores serán iniciados por dos puertos. Por el contrario, los *stores* marcados como no suministradores son iniciados únicamente por uno de los cuatro puertos.

La Figura 6.22 muestra la media harmónica de IPC para diferentes tamaños de STB1 en el esquema dos-replicado, con y sin predictor NFS. Incluimos también el IPC del esquema no replicado con el predictor NFS (cogido de la Figura 6.19). Los tres sistemas se recuperan desde el RIB, no suministran desde STB2 y STB1 no compara edades.

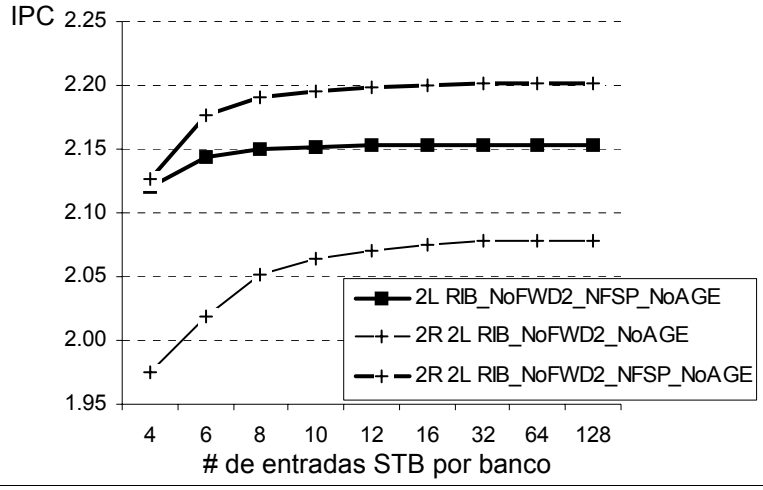

Figura 6.22 IPC para un sistema de STB en dos niveles 4-multibanco con y sin replicación de datos. Variando el tamaño del banco de STB1 entre 4 y 128 entradas.

Como se observa, el efecto del predictor NFS es importante en el esquema dos-replicado. El esquema dos-replicado es la mejor opción en todos los tamaños de

STB1 evaluados en este punto de diseño (4 bancos de L1 con 8 Kbytes cada uno, predictor de banco, etc.).

En la Figura 6.23 comparamos los dos esquemas con y sin replicación, variando el tamaño del banco de cache desde 2 Kbytes a 32 Kbytes y manteniendo el de STB1 en 8 entradas por banco. Ambos sistemas con predictor NFS, recuperación desde el RIB, sin suministro desde STB2 y sin comparación de edad en STB1.

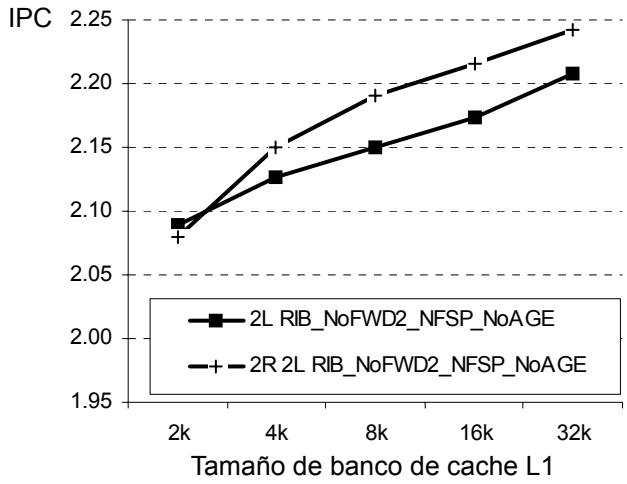

Figura 6.23 IPC para un sistema de STB en dos niveles 4-multibanco con y sin replicación de datos. Variando el tamaño del banco de cache entre 2 Kbytes y 32 Kbytes, y tamaño fijo de bancos de STB1 de 8 entradas.

Para todos los tamaños de banco de cache en L1, el esquema dos-replicado alcanza un buen compromiso entre capacidad efectiva de cache L1, precisión del predictor de banco y ancho de banda efectivo con L1.

6.5 Trabajos relacionados

En [YMRJ99], Yoaz y otros presentan el camino a memoria seccionado para una cache de primer nivel de dos bancos. Proponen iniciar las instrucciones *store* a ambos bancos. En este capítulo hemos mostrado que la contención en los puertos de inicio a memoria es un problema, y consecuentemente, mandar los *stores* a todos los bancos perjudica el rendimiento.

Zyuban y otros distribuyen la *cola de loads* y *stores* (LSQ) en bancos [ZyKo01]. Al emitir un *store*, se asigna entradas en todos los bancos y permenacerán asignadas hasta que consolide. Se necesitan muchas entradas para no parar la emisión de instrucciones, tal y como mostramos en la sección 6.2.3. Además,

los bancos de LSQ hacen uso de la edad para suministrar datos si hay varios *stores* a la misma dirección.

Racunas y Patt propusieron una nueva política de distribución de contenidos en caches de primer nivel multibanco [RaPa03]. Al igual que el trabajo anterior, distribuyen los STB locales asignando las entradas al emitir y las liberan al consolidar. Así que los bancos locales de STB necesitan muchas entradas para no parar la emisión. La asignación en los STB locales se hace en función del predictor de banco. La información de los *stores* debe ser movida de banco en dos situaciones; i) en caso de error de predicción de banco, y ii) cuando una línea de cache es dinámicamente re-asignada de un banco a otro, todas las entradas de los STB locales relacionadas con esa línea deben ser movidas desde el banco original al nuevo emplazamiento. Al mover *stores* de un banco a otro, se debe asegurar que se dispone de espacio en el STB local destino. Un STB global con capacidad de suministro se encarga de proporcionar los datos a los *loads* que sufren un error de predicción de banco. Los STB locales deciden entre los diversos *stores* a suministrar en función de la dirección y la edad.

Contrariamente a las dos propuestas anteriores, nosotros podemos diseñar bancos de STB1 con muy pocas entradas al asignarlas más tarde y liberarlas antes. Como las entradas del STB1 se asignan cuando ya se ha ejecutado el *store* por el camino correcto, nuestro diseño no necesita comunicación entre bancos. Además de todo esto, nuestro diseño de STB1 no compara edades para suministrar especulativamente datos y el STB2 no requiera la capacidad de suministro.

Akkary y otros propusieron una organización jerárquica de la *cola de stores* sobre un sistema de cache centralizada tipo Pentium IV (un *load* por ciclo) [AkRs03]. Consistía en un STB1 rápido más un STB2 más grande y lento (ambos centralizados). Las entradas del STB1 se asignan a los *stores* al emitir en orden FIFO. Cuando el STB1 se llena, el *store* más viejo se desplaza al STB2. Ambos STBs suministran datos, pero a diferentes latencias. Para conocer si la dirección no esta en STB2 y poder eliminar accesos, usan un filtro (MTB) tipo *Bloom Filter* [Bloo70]. Nuestra propuesta también se basa en STBs en dos niveles pero centrándonos en configuraciones de caches de L1 multibanco, las cúales requieren analizar más desafíos, tales como la gestión de múltiples bancos de STB1 y la congestión en los puertos de inicio a memoria. Nosotros reducimos el tamaño del STB1 al retardar la asignación de entradas a la ejecución y sugerimos diseños simples tanto del STB1 (no comparar edad) como del STB2 (no suministro).

Para eliminar búsquedas en la *cola de stores*, Setthumadhaven y otros [SDB+03] proponen utilizar un *Bloom filter*. Así mismo usan otro filtro para decidir que *loads* almacenar en la *cola de loads*, de esta forma reducen el ancho de búsqueda de la *cola de stores* y el tamaño de la *cola de loads*. De forma similar, Park y otros reducen el ancho de búsqueda en STB al usar un predictor de pareja *store-load* basado en el predictor *Store-Set* [PaOV03]. Además parten el STB en múltiples trozos más pequeños y acceden a ellos en cadena y por tanto con latencias variables de suministro. Las ideas en ambos trabajos se podrían aplicar junto con otras ideas en nuestro STB2 para filtrar tanto el número de búsquedas como el número de entradas en las que buscar y con ello reducir el número de puertos de acceso o el consumo de esta estructura.

6.6 Conclusiones

En este capítulo estudiamos como diseñar un *Store Buffer* distribuido y en dos niveles adecuado para caches multibanco de primer nivel. Nuestro objetivo es el suministro especulativo de datos procedentes de *stores* no consolidados a *loads* y todo ello dentro de la latencia de acceso al banco de cache. El suministro es realizado especulativamente desde un STB distribuido de primer nivel (STB1) compuesto por bancos pequeños. Algunos ciclos más tarde, un STB centralizado de segundo nivel (STB2) comprueba el suministro especulativo y fuerza el orden correcto de acceso a memoria entre *stores* y *loads*.

Las entradas del STB2 se asignan al emitir los *stores* y se liberan al consolidarlos. Sin embargo, retardamos la asignación de las entradas del STB1 hasta la ejecución del *store* por el banco correcto, y permitimos la liberación temprana de las entradas sin esperar a la consolidación. Si un banco de STB se llena, las entradas son re-asignadas en orden FIFO. Esta política de asignación / liberación nos permite reducir el tamaño de STB1, asignar las entradas sólo en el banco destino sin usar la predicción de banco y no parar la emisión de instrucciones al llenar STB1.

Además, la distribución de papeles entre los niveles de STB permite aplicar dos simplificaciones al diseño sin pérdidas de rendimiento, dígase la no comparación de la edad en el suministro desde STB1 y la eliminación de la capacidad de suministro desde STB2.

Como el suministro desde STB1 es especulativo, nuestro sistema necesita un mecanismo que permita recuperarnos en caso de errores de especulación de suministro. Hemos comprobado que la recuperación desde la IQ, en este caso,

perjudica al rendimiento al tener que mantener al *load* y a sus instrucciones dependientes en la IQ durante mucho tiempo. Como alternativa, proponemos recuperarnos del *buffer de instrucciones renombradas* (RIB), lo cuál, a pesar de tener una penalización mayor, nos da un mejor rendimiento.

Por último, el uso de un predictor de *stores* no suministradores, nos permite reducir la contención en los puertos de inicio a memoria de la IQ. Los *stores* marcados como no-suministradores se iniciarán por cualquier puerto de inicio a memoria libre, incrementando el ancho efectivo con memoria. Esta mejora es especialmente útil si la contención por *stores* es alta, como por ejemplo en esquemas con caches multibanco replicadas.

Siguiendo nuestras indicaciones, un sistema distribuido de STB con dos niveles y bancos de STB1 de 8 entradas (sin comparación de edades en STB1 y sin suministro desde STB2), obtiene el mismo rendimiento que un STB distribuido ideal de 128 entradas, capaz de suministrar en un ciclo.

CAPÍTULO 7

Conclusiones y Líneas Abiertas

7.1 Conclusiones

La latencia del primer nivel de cache es uno de los factores clave que limitan el rendimiento obtenible en un procesador. Esta Tesis se centra en el diseño distribuido del primer nivel de cache en procesadores superescalares con ejecución fuera de orden. Asumimos que el acceso al primer nivel de memoria se realiza mediante caminos independientes, completamente separados en secciones. El objetivo global es disminuir la latencia *load-uso*, condición necesaria para obtener un buen rendimiento en la mayoría de los programas. En concreto, nos centramos en la distribución en bancos de la memoria cache y del *Store Buffer*, en los predictores de acceso a los bancos, en la distribución de contenidos de los bancos y en las políticas adecudas de recuperación.

7.1.1 Predictor de banco

Cuando se utilizan caminos a memoria seccionados para acceder al primer nivel de cache se necesita un predictor de banco para encaminar las peticiones a memoria. El predictor seleccionado permite realizar varias predicciones por ciclo y es una variante del predictor de saltos denominado enhanced-gskew [MiSU97] al que se ha añadido una medida de confianza. Tras haber analizado su comportamiento, se han dimensionado sus tres tablas con 2^{13} entradas (capacidad total de 9 Kbytes por bit a predecir), consiguiéndose que los errores con confianza se sitúen por debajo del 2%.

7.1.2 Contrarrestando la penalización por error de banco

En un procesador con caminos a memoria seccionados para acceder al primer nivel de cache, la pérdida de rendimiento tiene relación directa con la tasa de errores del predictor de banco. Se han propuesto dos mecanismos complementarios para reducir globalmente la penalización por error de predicción de banco.

- La primera es la *Recuperación en Cadena*, un mecanismo nuevo de recuperación selectiva indicado para reducir la penalización de recuperación tras un error de predicción de banco. Este mecanismo se puede utilizar también para la recuperación de otras especulaciones de latencia, como la suposición de acierto en cache.

- El segundo mecanismo planifica la ejecución de un *load* por todos los caminos de acceso a memoria si la confianza de la predicción de banco es baja. De esta forma se descarta la predicción de encaminamiento por un banco único cuando es probable que vaya a producirse un error de predicción de banco, reduciéndose la latencia *load-uso* y eliminándose la penalización por recuperación. Hemos denominado a este mecanismo *Replicación selectiva en Emisión*.

Combinando *Recuperación en Cadena* y *Replicación selectiva en Emisión*, eliminamos alrededor de dos tercios de la pérdida de IPC por error de predicción de banco en todas las configuraciones simuladas.

7.1.3 Gestión de contenidos

Al enfrentarse al diseño de un primer nivel de cache multibanco, hay dos decisiones principales relacionadas con la gestión de contenidos: la *política de distribución* y el *grado de replicación*. Elegir un determinado punto de diseño establece el número de copias permitidas (grado de replicación) e identifica a los bancos que deben acogerlas.

- En general, las *políticas de distribución* de contenidos que fraccionan el flujo de datos en función de la dirección de memoria consiguen un reparto más uniforme y por ello obtienen las menores tasas de fallos de cache y las menores tasas de contención en los puertos de inicio a memoria. Nuestros resultados ponen de manifiesto que la mejor política de distribución es *Entrelazado por Palabra*.

- El grado óptimo de replicación está sujeto a la capacidad del banco. En bancos pequeños lo ideal es no replicar. *Replicación Parcial* sobre dos bancos conviene a tamaños medios. Con bancos mayores, el rendimiento de *Replicación Total* es equivalente a la del mejor gestor de contenidos pero con un coste menor.

En el espacio de diseño estudiado (bancos entre 2 y 16 Kbytes con latencias de 2 y 3 ciclos), una cache con cuatro bancos de 2 Kbytes a 2 ciclos de latencia supera en rendimiento a cualquier otra de 3 ciclos. Por ello, incrementar la capacidad del banco nunca es una buena opción si involucra un aumento de latencia.

7.1.4 Diseño del *Store Buffer*

Nuestro objetivo es suministrar datos procedentes de *stores* no consolidados a *loads*, con la misma latencia de acceso al banco de cache. Tras estudiar el comportamiento de los *stores* y del suministro *store-load* se propone un *Store Buffer* distribuido y en dos niveles.

En concreto, proponemos un primer nivel distribuido (STB1), compuesto por bancos pequeños especializados en el suministro especulativo de datos, y un segundo nivel centralizado (STB2), que almacena todos los *stores* en vuelo, especializado en garantizar el orden de acceso a memoria. En particular, STB2 es responsable de comprobar el suministro de datos realizado especulativamente desde STB1.

Las entradas de STB2 se asignan al emitir los *stores* y se liberan al consolidarlos. Sin embargo retardamos la asignación de entradas en los bancos de STB1 hasta la ejecución del *store* por el banco correcto. Si un banco de STB1 se llena, las entradas son reasignadas en orden FIFO. Esta política de asignación-liberación permite un rendimiento elevado con tamaños muy reducidos de los bancos de STB1.

Además, la distribución de papeles entre los niveles de STB permite aplicar dos simplificaciones al diseño sin pérdidas significativas de rendimiento. Estas simplificaciones son la supresión de la comparación de edad en el suministro desde STB1 y la eliminación de la capacidad de suministro desde STB2.

Como el suministro desde STB1 es especulativo, es necesario un mecanismo de recuperación cuando se produce un error de suministro *store-load*. Hemos detectado que la recuperación desde IQ perjudica excesivamente al

rendimiento. Como alternativa, y debido al bajo número de errores de suministro proponemos la recuperación desde el *buffer de instrucciones renombradas* (RIB).

Puesto que la mayoría de los *stores* no suministran a *loads* más jóvenes y sucede que este comportamiento es altamente predecible, proponemos el uso de un *predictor de stores no suministradores*. Este predictor permite reducir la contención en los puertos de inicio a memoria de IQ. Los *stores* marcados como no-suministradores se iniciarán por cualquier puerto de inicio a memoria libre, incrementando el ancho de banda efectivo con memoria. Esta mejora es especialmente útil si la contención provocada por los *stores* es elevada, como por ejemplo en los esquemas de cache multibanco con contenidos replicados.

Siguiendo nuestras indicaciones, un sistema distribuido de STB con dos niveles y bancos de STB1 de 8 entradas (sin comparación de edades en STB1 y sin suministro desde STB2), obtiene el mismo rendimiento que un STB distribuido ideal de 128 entradas.

7.2 Líneas Abiertas

A continuación citamos algunas líneas abiertas derivadas del trabajo realizado.

- Los predictores de banco utilizados consiguen un rendimiento notable. Sin embargo, la reducción de su complejidad o de su tasa de errores son objetivos interesantes. Creemos interesante añadir al predictor un mecanismo para el reconocimiento explícito de secuencias de referencias que siguen un patron stride. El objetivo es reducir los conflictos en las tablas de predicción, lo cual reducirá los errores de predicción. Así mismo, es lógico evaluar la tasa de error utilizando otros predictores de bits que se basan en el *Perceptron* [JiLi02][Sezn04][TaSk04].

- Para reducir la penalización en errores de predicción de banco se ha añadido confianza al predictor. Cuando no existe confianza en la predicción la instrucción *load* se inicia por todos los caminos de acceso a memoria con el objetivo de reducir la latencia *load-uso*. Sin embargo, este mecanismo utiliza ancho de banda de inicio. Nos parece interesante evaluar otras alternativas como la replicación dinámica y selectiva de bloques de cache utilizando la confianza.

- El número de entradas del segundo nivel de STB y del LDB es proporcional al número de instrucciones en vuelo y requiere tantos caminos de acceso a

etiquetas como el ancho de banda de inicio a los puertos de memoria. Otros trabajos se han centrado en el escalado de procesadores con STB y LDB centralizados, lo cual es parejo al segundo nivel propuesto en este trabajo. Creemos que la caracterización del comportamiento de los *stores* y del suminsitro *store-load* puede permitir un mejor escalado del segundo nivel de STB y del LDB.

- La diferencia de velocidad entre el procesador y la memoria principal sigue aumentando. Para tolerar esta latencia diversos trabajos estudian técnicas para explotar el paralelismo distante en presencia de fallos de cache. El diseño distribuido y multinivel del *Store Buffer* parece estar especialmente indicado en entornos con muchas instrucciones en vuelo. En concreto creemos interesante analizar su comportamiento en las propuestas actuales de procesadores con ventanas de instrucciones muy grandes [COLV04] [CSVM04][AkRS03][SRA+04].

- Todo el trabajo realizado en esta Tesis se ha desarrollado para procesadores con un solo flujo de ejecución (*single-threaded processors*). Pensamos que la aplicación de las ideas propuestas en procesadores *multithread* puede requerir otros compromisos de diseño. Decidir, por ejemplo, qué recursos de predicción pueden ser compartidos y cuales privados, o bien, determinar cómo deben escalarse los bancos de STB1 en función del número de *threads*, pueden ser aspectos interesantes a investigar.

APÉNDICE A

VisualPtrace

En este apéndice se introduce la aplicación VisualPtrace realizada durante la Tesis para ayudar a validar y comprender mejor el funcionamiento del procesador simulado.

a.1 Introducción

La validación de un simulador detallado de un procesador superscalar fuera de orden es una tarea compleja y no libre de errores. Como ya se comentó en la sección 2.4, hemos aplicado distintas técnicas para depurar el código y dar credibilidad a los resultados obtenidos.

Aún siendo correcta la ejecución del código existen multitud de interacciones entre instrucciones y entre eventos en distintas etapas del segmentado. Para clarificar el discurrir de las instrucciones se creó una aplicación visual que permite observar y seguir el estado del segmentado. Con el programa se puede validar la correción temporal de la ejecución, profundizar en el conocimiento de cómo fluyen las instrucciones en un fuera de orden, y analizar el comportamiento ante distintas situaciones.

En este apéndice queremos mostrar algunas capturas de pantalla que pongan de manifiesto la utilidad de la aplicación.

.

a.2 Descripción de la Herramienta

SimpleScalar dispone de herramientas para generar la traza de ejecución de las instrucciones y su paso por las etapas del segmentado. Su uso se invoca a través de un parámetro en la línea de comandos:

```
sim-outorder -config 81mp4.conf -ptrace gcc_mp4.trc 4101:+2000 gcc.eio
```

El resultado es un fichero de texto donde queda codificada la ejecución. La Figura a.1 muestra un trozo del fichero de traza. Por ejemplo: @ indica el inicio del ciclo y el ciclo actual, el signo + marca la entrada de una nueva instrucción en el segmentado, o el simbolo * señala el paso por una etapa.

```
@ 1968700
+ 6785221 0x1200d24e0 0x00000000 stq       r11,144(r30)
* 6785221 IF 0x00000000 0x00000000
+ 6785222 0x1200d24e4 0x00000000 stq       r9,160(r30)
* 6785222 IF 0x00000000 0x00000000
+ 6785223 0x1200d24e8 0x00000000 br        r31,0x64
* 6785223 IF 0x00000000 0x00000000
+ 6785224 0x1200d254c 0x00000000 ldq       r4,144(r30)
* 6785224 IF 0x00000000 0x00000000
+ 6785225 0x1200d2550 0x00000000 lda       r17,6656(r31)
* 6785225 IF 0x00000000 0x00000000
+ 6785226 0x1200d2554 0x00000000 ldq       r16,160(r30)
* 6785226 IF 0x00000000 0x00000000
+ 6785227 0x1200d2558 0x00000000 ldah      r23,1(r29)
* 6785227 IF 0x00000000 0x00000000
+ 6785228 0x1200d255c 0x00000000 beq       r4,0xd8
* 6785228 IF 0x00000000 0x00000000
@ 1968701
* 6785221 DA 0x00000000 0x00000000
* 6785222 DA 0x00000000 0x00000000
* 6785223 DA 0x00000000 0x00000000
* 6785224 DA 0x00000000 0x00000000
* 6785225 DA 0x00000000 0x00000000
* 6785226 DA 0x00000000 0x00000000
* 6785227 DA 0x00000000 0x00000000
* 6785228 DA 0x00000000 0x00000000
+ 6785229 0x1200d2560 0x00000000 bne       r16,0xd4
* 6785229 IF 0x00000000 0x00000000
```

Figura a.1 Traza generada por sim-outorder.

Seguir la secuencia de eventos con esta traza resulta prácticamente imposible. SimpleScalar no dispone de una herramienta adecuada de visualización de la traza. VisualPtrace viene a llenar ese vacío mostrando de forma gráfica la traza de ejecución.

La Figura a.2 muestra (girada 90 grados) la captura de la ventana principal de VisualPtrace. La pantalla se divide en cuatro zonas. En la zona superior izquierda se sitúan los controles generales. Debajo, a la izquierda se muestra el listado de instrucciones en vuelo. A su derecha se puede observar el paso por las etapas principales del segmentado de cada una de las instrucciones. Como estamos interesados en el primer nivel de memoria multibanco, en la parte superior al segmentado se muestra la ocupación de los puertos de acceso a memoria.

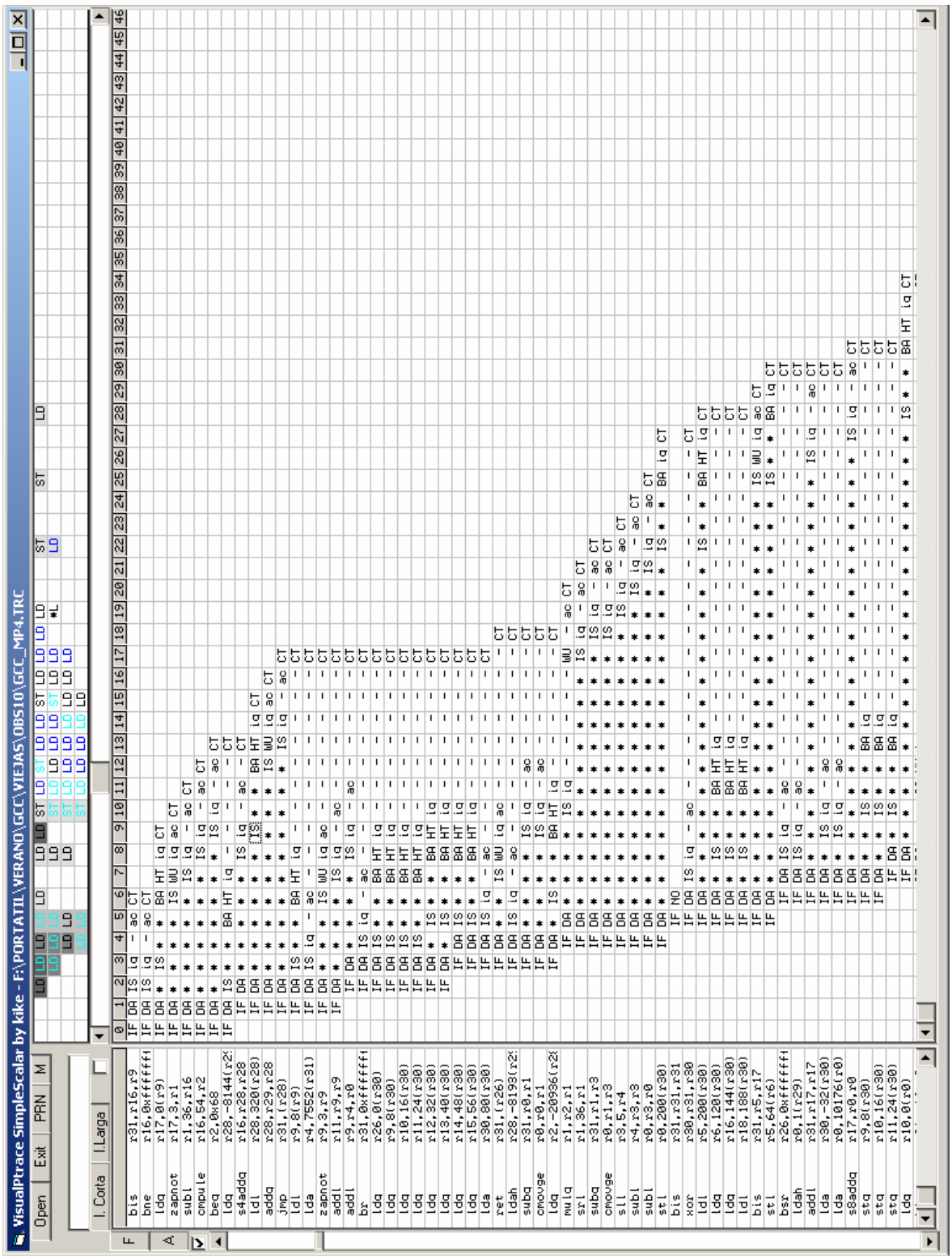

Figura a.2 Captura pantalla principal de VisualPtrace.

La aplicación permite alejar el punto de vista para mostrar un mapa donde se puede ver globalmente un mayor número de instrucciones y ciclos. La Figura a.3 muestra una captura de pantalla del mapa. De arriba a abajo se encuentran; menús de la aplicación y botones de control, uso de los puertos a los bancos de memoria y el resto muestra el paso de las instrucciones por el segmentado.

Cada instrucción (eje y) puede ser representada por uno, dos o cuatro pixels. El eje x representa el tiempo (ciclos). Los grises de las instrucciones codifican información. Las instrucciones en negro corresponden a camino erróneo de saltos. La zona gris oscuro marca la estancia en la IQ. El ciclo blanco corresponde al inicio de ejecución de la instrucción y por último la zona gris clara la espera a la consolidación. La longitud total de la barra delimita la estancia de la instrucción en el ROB.

Las cajas de texto asociadas a cada bloque de istrucciones (comprendidas entre dos saltos mal predichos) indican el IPC de ese bloque.

Existen otra serie de pantallas, que no mostramos, que dan acceso a más información sobre las instrucciones o sobre las estructuras del procesador.

La aplicación está orientada a Windows, al posar el ratón sobre las instrucciones se muestra en pantalla información adicional. Por ejemplo, de una instrucción *load* podremos saber la dirección efectiva de acceso, el banco determinado por el predictor de banco, el banco real al que debe acceder, y otra serie de flags definidos por el usuario.

La zona de pantalla reservada para marcar el uso de los puertos de acceso a memoria también codifica información adicional. Por un lado tendremos la distinción entre *LoaDs* o *STores*, camino correcto de saltos o erróneo, temperatura relativa, acceso a la región de pila o no pila, indicación de si el acceso es a la misma palabra o línea del último acceso realizado en ese banco, etc. La temperatura relativa es una característica que hace referencia a la contención en los puertos de inicio de la IQ. Cada ciclo, las instrucciones listas para iniciar la ejecución más viejas de la IQ tendrán la temperatura más alta, mientras las más jovenes serán las más frías. Observando la temperatura relativa de las instrucciones que hacen uso de los puertos podemos inferir la contención en los puertos de inicio de la IQ.

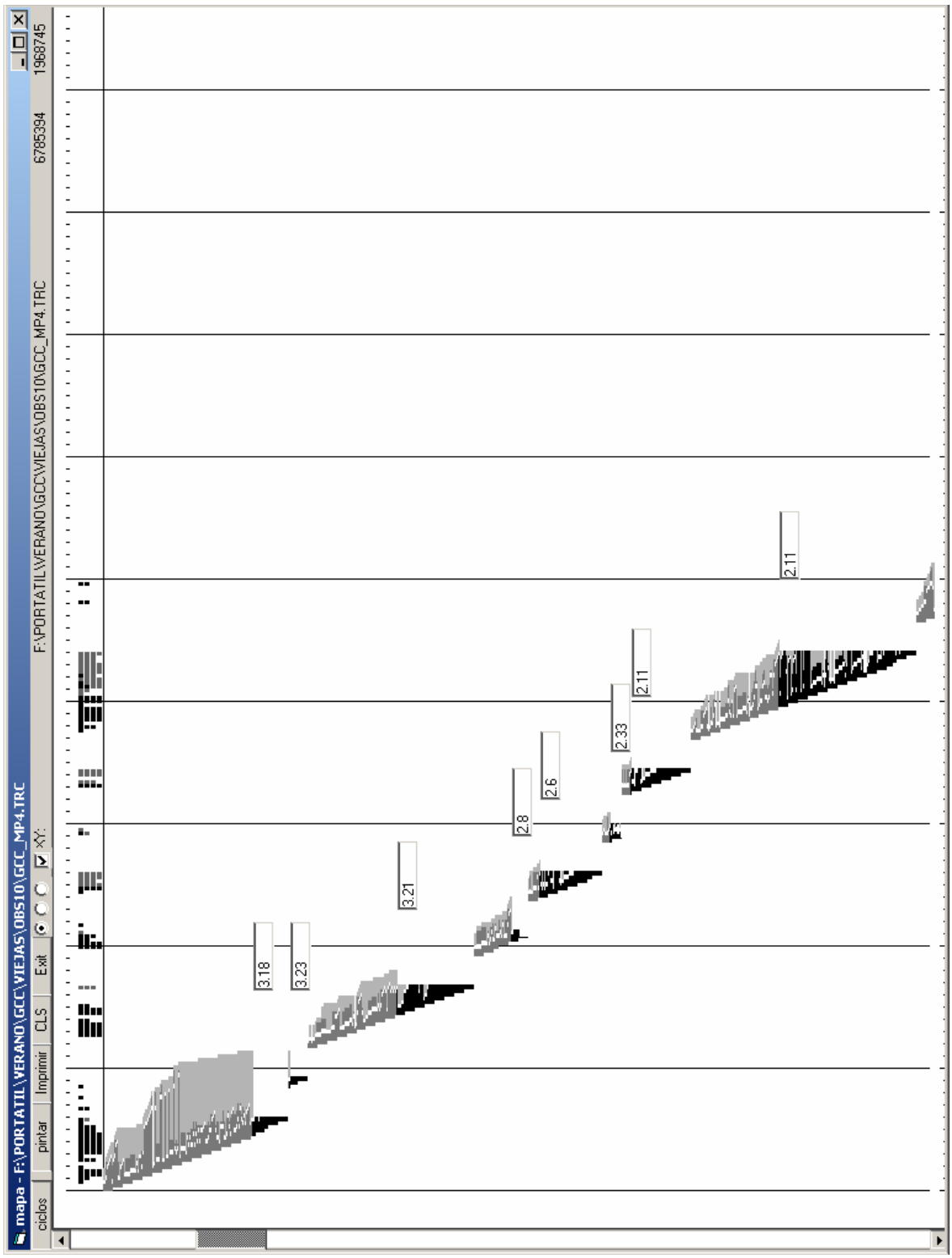

Figura a.3 Captura pantalla Mapa de VisualPtrace

En la cuadrícula donde se representa el paso de las instrucciones por etapas sólo se muestra el primer evento que le ocurre a esa instrucción en ese ciclo. En el seguimiento de la instrucción con los cursores podremos observar la totalidad de los eventos ciclo a ciclo.

Los símbolos mostrados y su significado es:

- IF: última etapa de búsqueda y renombre
- DA: decodificación y emisión (inserción en IQ y ROB)
- NO: instrucción NOP, eliminada al decodificar (Digital Alpha 21264)
- IS: inicio de ejecución
- iq: sacar de IQ (sin especulación de latencia o ya validada)
- ac: completada su ejecución, puede ser consolidada
- CT: etapa de consolidación
- *: estancia en IQ
- -: estancia en ROB

Centrándonos en las instrucciones de acceso a memoria:

- BA: acceso al banco de cache DL1
- WU: despertar de las dependientes
- NB: notificación de error de predicción de banco
- XX: anulación por error de predicción de latencia
- HT: acierto en DL1
- MI: fallo en DL1
- .... otras

El interface de SimpleScalar para generar la traza es muy simple. Instrumentando el código fuente del simulador resulta muy sencillo añadir nuevos eventos asociados a las instrucciones. Cada evento está determinado por dos letras y va asociado a una instrucción en un determinado ciclo. VisualPtrace lee el fichero de texto y lo representa. Esto permite que la aplicación sea bastante general y se pueda adaptar a diversos entornos (usuarios de SimpleScalar).

Al mostrar el discurrir de las instrucciones por el segmentado y las relaciones de unas con otras, resulta fácil comprobar el correcto funcionamiento. La

aplicación permite situarse en un punto determinado de la ejecución o buscar las ocurrencias de un determinado evento.

Junto a VisualPtrace se desarrollaron una serie de filtos que permiten comparar las ejecuciones sobre dos modelos de procesador distintos. La idea general es, que si ambos procesadores comparten el mismo predictor de saltos, la secuencia de saltos mal predichos será similar (no exactamente iguales por la temporalidad de la actualización y recuperación). Además, debido a la alta penalización de recuperación, al llegar la primera instrucción del camino correcto el ROB suele estar ya vacío. Dividiendo la ejecución en bloques de instrucciones delimitadas por errores de predicción de saltos, podemos comparar el número de ciclos necesarios para ejecutarlos en cada modelo de procesador.

Sabiendo en qué bloques se encuentran las mayores diferencias, (ponderado por el número de veces que se ejecutan), podremos centrar nuestra atención en esos puntos interesantes de la ejecución. Por ejemplo si queremos ver los efectos de la contención podríamos comparar un procesador con un modelo de cache de primer nivel multipuerto con otro procesador con el primer nivel multibanco (camino a memoria seccionado), o si queremos ver el efecto de los errores de predicción de banco comparar un modelo con predicción perfecta de banco con un modelo con predictor real.

Visualizando ambas ejecuciones simultáneamente se pueden analizar las diferencias. Por ejemplo, las Figura a.4 y Figura a.5 muestran la ejecución del mismo fragmento de código en un procesador con dos modelos de primer nivel de memoria. La primera muestra la ejecucción en un multipuerto, mientras la segunda lo hace sobre un multibanco con predictor de banco real.

Analizando el segmentado podemos ver como se entrelaza el inicio de ejecución de los ocho primeros *loads* o por ejemplo como no afecta el error de banco del primer *load* al haberse predicho bien la dirección de salto (retorno) de la instrucción `ret` (10ª instrucción).

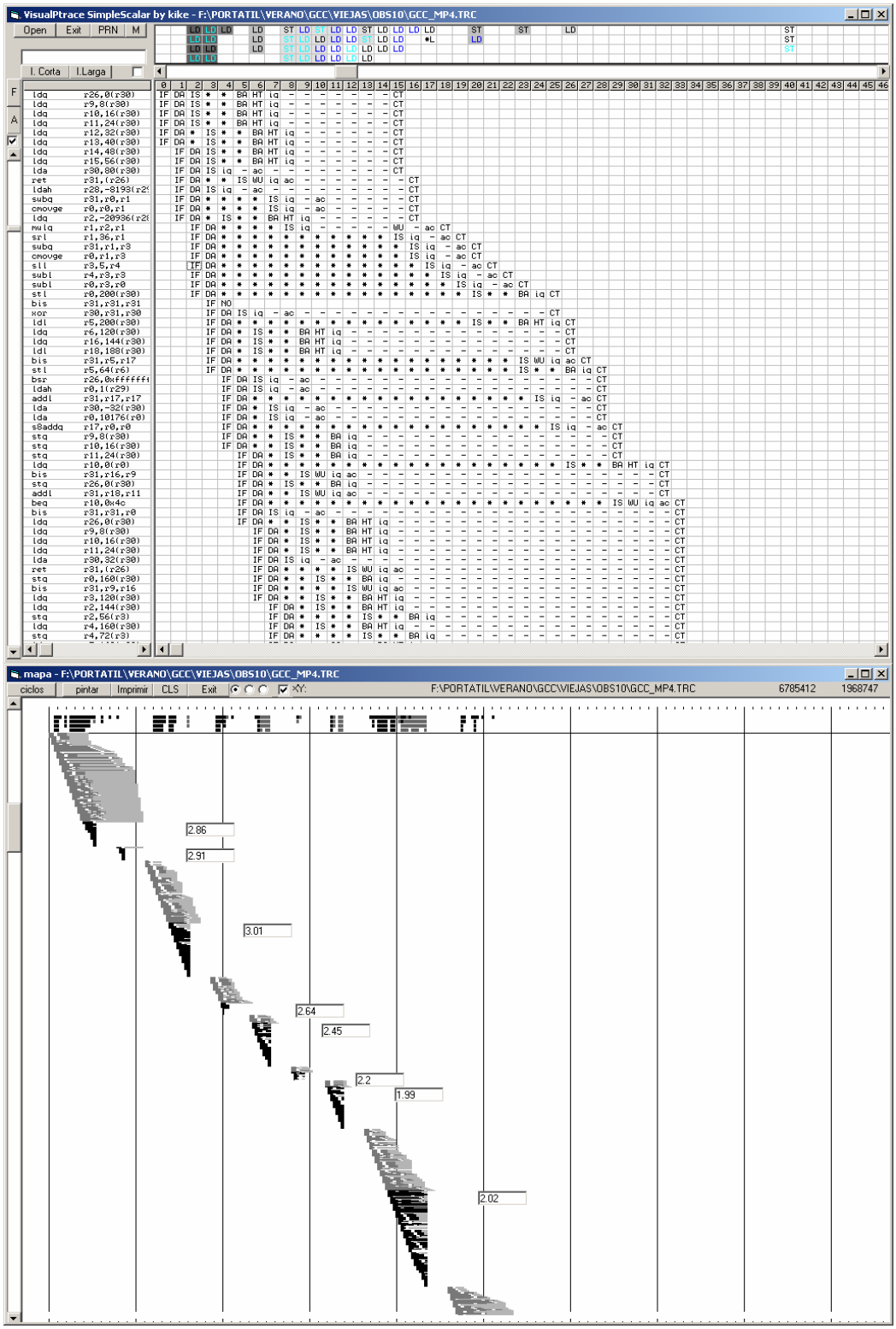

Figura a.4 Visualización de la ejecución de un código sobre un procesador 4-multipuerto.

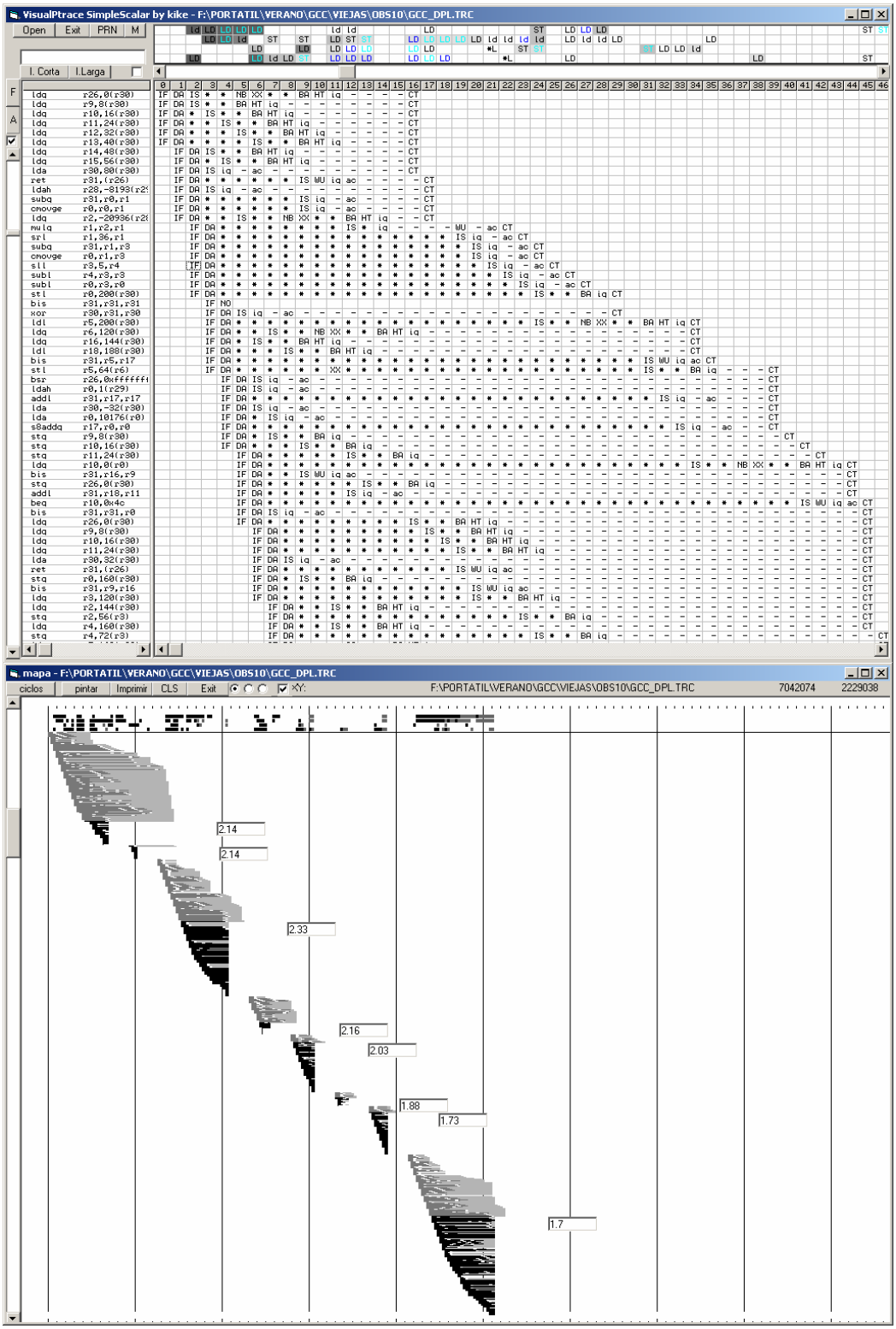

Figura a.5 Visualización de la ejecución de un código sobre un procesador 4-multibanco.

Como curiosidad comentar dos conclusiones importantes que pudimos observar gracias a este método y a la aplicación VisualPtrace:

- En el modelo base de recuperación por errores de especulación de latencia, los *loads* con error de banco del camino erróneo de saltos, al bloquear el inicio de ejecución durante el ciclo de recuperación, retardaban la validación del salto mal predicho.

- Un procesador bien dimensionado, era capaz de tolerar bastante bien la contención y los errores de predicción de banco, siempre y cuando ambas situaciones no se den en una cadena de dependencia de un salto mal predicho, (muchos trabajos en multibanco modelan predicción de saltos perfecta :-).

a.3 Estado actual de VisualPtrace

Personalmente creo que existen dos ámbitos en los que resulta interesante su aplicación; en el entorno docente donde se quiera mostrar la ejecución especulativa fuera de orden, y en la investigación para analizar el comportamiento.

APÉNDICE B

Otros Resultados

En este apéndice se plasman otros resultados referenciados a lo largo de la memoria o que consideramos importante dejar reflejados en la memoria.

b.1 Instrucciones ejecutadas

Como ya mencionamos, se simulan un total de 100 millones de instrucciones contiguas (SimPoints [SPHC02]). Del total de instrucciones, el 38% corresponden a referencias a memoria de las que el 25% son *loads*. Por otro lado, el 14% son saltos. En la Tabla b.1 se pueden consultar las proporciones en cada programa de la carga de trabajo referidas a millones de instrucciones (y como se ejecutan 100 millones, en tanto por ciento).

Tabla b.1 En millones, número de instrucciones de cada tipo ejecutadas por cada benckmark.

mill.	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	media
loads	22	29	25	24	14	21	34	22	27	23	25	30	25
stores	12	6	21	11	27	8	8	10	17	7	14	12	13
saltos	15	11	12	14	11	10	26	16	15	13	18	11	14

En la Figura b.1 se muestran de forma gráfica estos mismos valores.

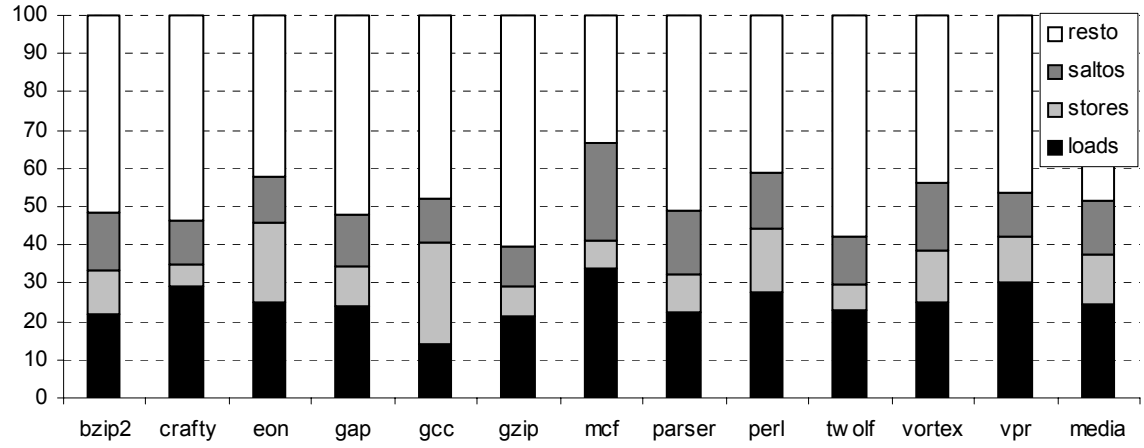

Figura b.1 Proporción de instrucciones en millones (100 millones simulados)

Como se puede observar, todos los programas siguen aproximadamente las mismas tendencias. Destaca MCF por su alto porcentaje de *loads* (34%) y saltos (26%) respecto a los demás. Así mismo destaca GCC con un 27% de *stores*.

Centrándonos en las referencias a memoria, en la Figura b.2 se muestra el tanto por ciento de *loads* y *stores* sobre el total de referencias. La media se sitúa en torno al 66% de *loads* contra un 34% de *stores*. Como ya se comentó en el párrafo anterior, destaca de la media GCC con cerca de un 66% de *stores* o CRAFTY (MCF) con cerca del 84% de *loads*.

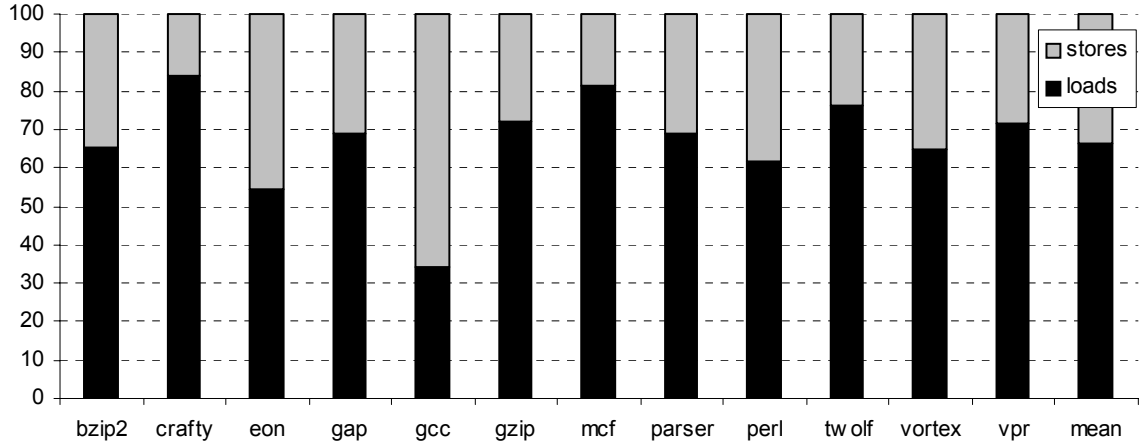

Figura b.2 Proporción de *loads*/*stores* respecto al total de referencias.

b.2 Origen de los datos

Otra cuestión también interesante es ver el origen dinámico de los datos leídos por los *loads*. En este caso nos centraremos en la ejecución sobre el procesador *8-inicio* ya presentado con la configuración de STB distribuido en un sólo nivel del Capítulo 6.

En primer lugar analizaremos la Figura b.3 que muestra el origen de los datos leidos por los *loads* descomponiéndolo en: a) banco de cache (DL1), b) lectura desde el banco de cache coincidiendo con el *refill* de L2 a L1 (*Refill* L2), c) desde el STB distribuido en un sólo nivel (STB) o d) desde el *Store Buffer* consolidados pendientes de actualizar el banco de cache en ciclos libres (STB_CT).

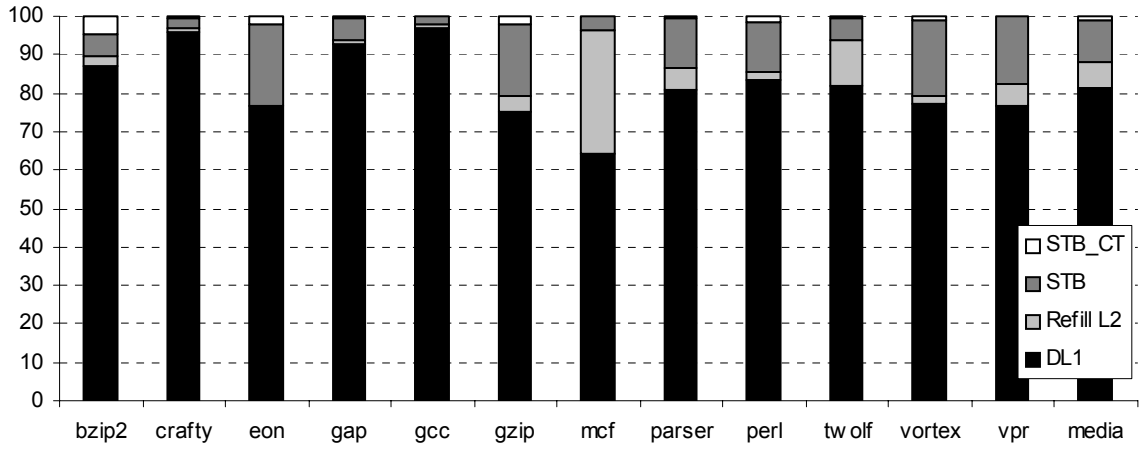

Figura b.3 Origen de los datos leidos por los *loads*.

En media los *loads* recogen el dato, (en su última ejecución), el 81% desde el banco de cache, el 7% coinciciendo con el *refill* desde L2 a L1, el 11% desde el STB de *stores* no consolidados y el 1% desde el *Store Buffer* consolidados.

Si nos centramos en los *stores*, la Figura b.4 muestra sobre el total de *stores* el porcentaje de *stores* suministradores, descompuesto entre los que lo hacen una única vez (FWD1) y los que lo hacen dos o más veces (FWD+). El resto (no_FWD) no suministran el dato a *loads* más jóvenes.

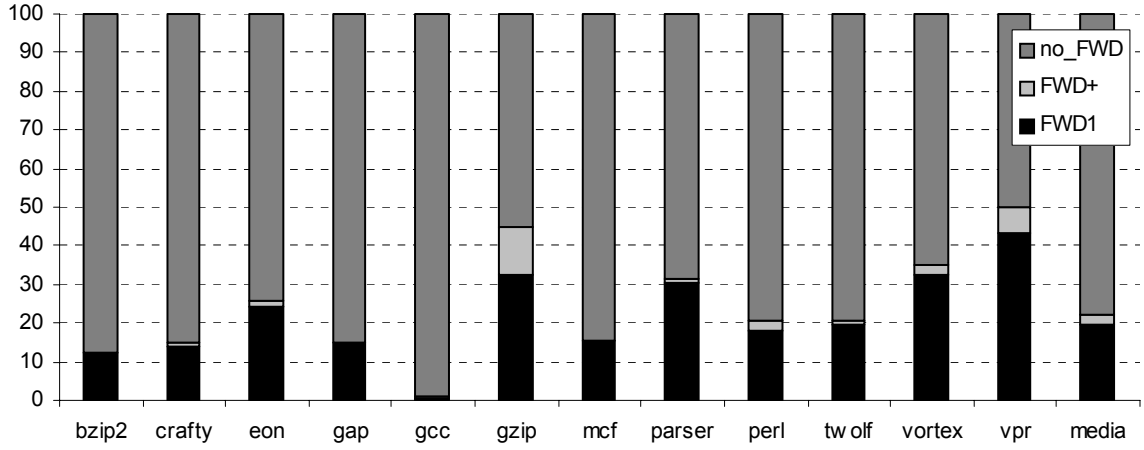

Figura b.4 Proporción de *stores* que suministran una sóla vez (FWD1), dos o más veces (FWD+) o ninguna (no_FWD) sobre el total de *stores*..

La Figura b.5 muestra el porcentaje de veces que el suministro *store-load* se realiza en el rango de memoria correspondiente a la pila del programa (*stack*) o a otras zonas de memoria (*heap* o *data*). El 70% del suministro en media, se realiza por accesos a la pila (FWD_pila).

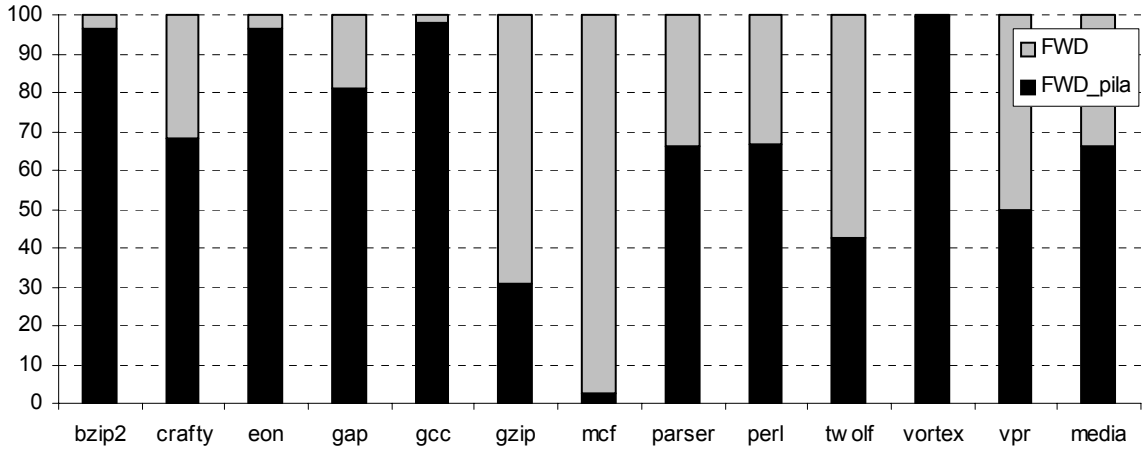

Figura b.5 Proporción de *stores* a pila suministradores sobre el total de *stores* suministradores.

b.3 Tasa de fallos de cache

Un punto interesante para acabar de entender el IPC es la tasa de fallos en el primer nivel de la jerarquía. En la Figura b.6 se muestra la tasa de fallos de L1 para una cache 4-multibanco. El punto 2k-8k corresponde a cuatro bancos de 2 Kbytes por lo que la capacidad total de la cache es 8 Kbytes. El resto corresponden a bancos de 4 Kbytes (4k-16k), 8 Kbytes (8k-32k) y 16 Kbytes (16k-64k).

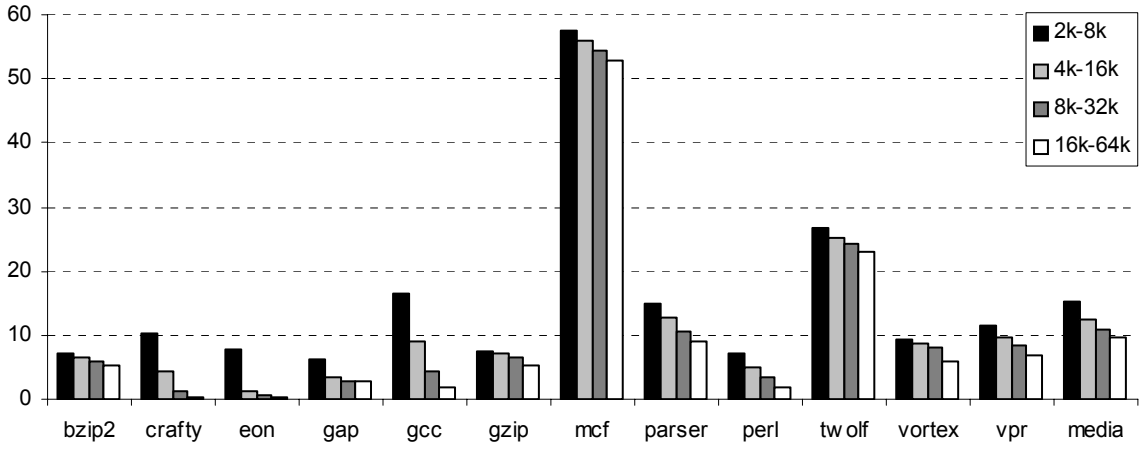

Figura b.6 Tasa de fallos del primer nivel 4-multibanco.

b.4 Predictor de banco

El predictor *egskew* fue presentado en el Capítulo 3. En este apéndice se muestran los resultados por programa.

Se representa el barrido de historias para tablas desde 2^{12} a 2^{15} entradas (4K a 32K entradas para cada una de las tres tablas). Las gráficas muestran por separado los resultados en tanto por uno de los bits 3, 4, 5 y 6.

Presentamos tres gráficas: el total de referencias clasificadas como de baja confianza (*T_sin_conf*), el total de errores de banco (*T_error*), y los errores con confianza (*Error_conf*). Las escalas de las gráficas varían entre programas para representar mejor los resultados. Sin embargo los pasos en el eje *Y* se mantienen fijos a: 0'05, 0'02 y 0'01.

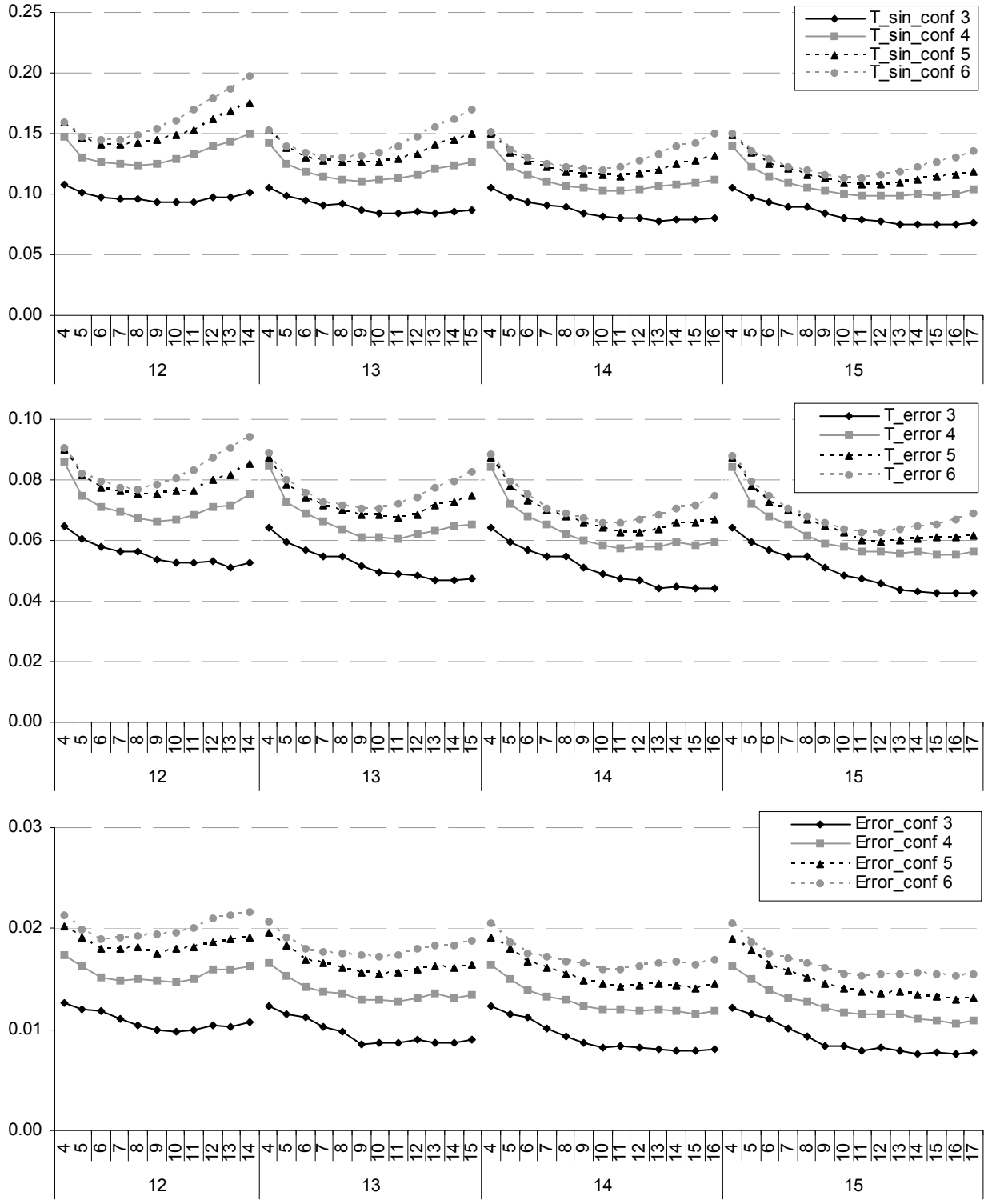

Figura b.7 Resultados predictor de banco `Mint`.

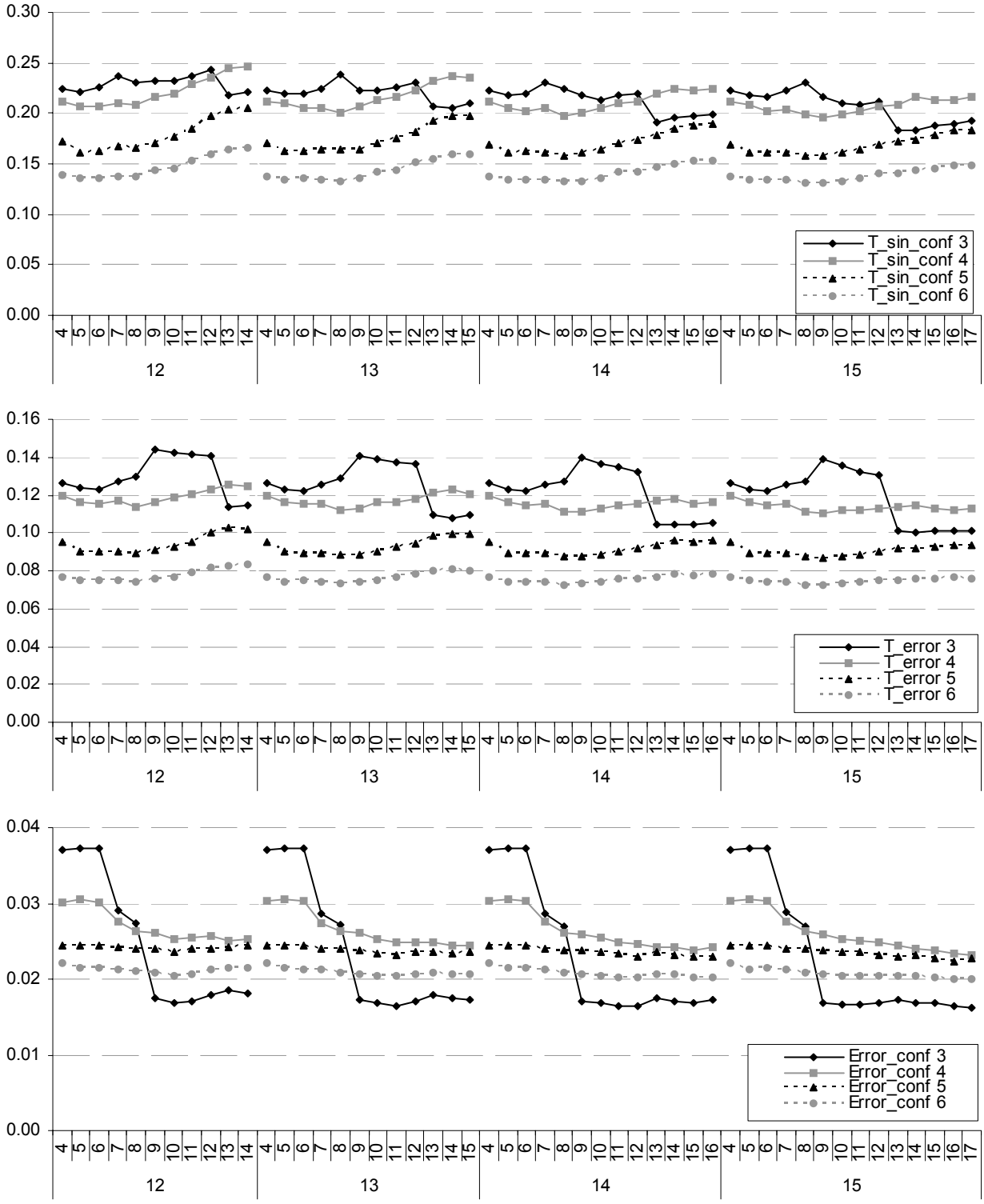

Figura b.8 Resultados predictor de banco `BZIP2`.

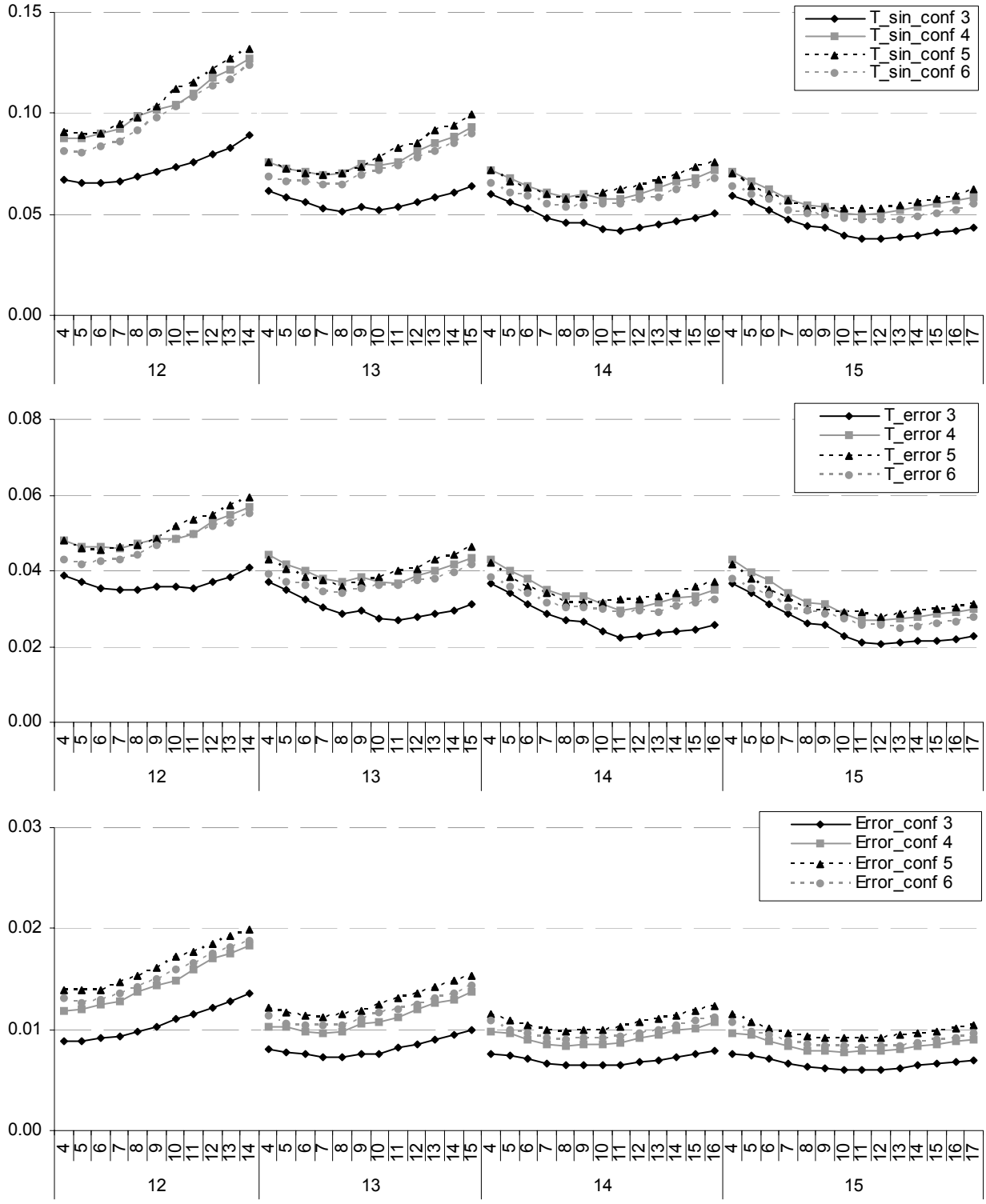

Figura b.9 Resultados predictor de banco CRAFTY.

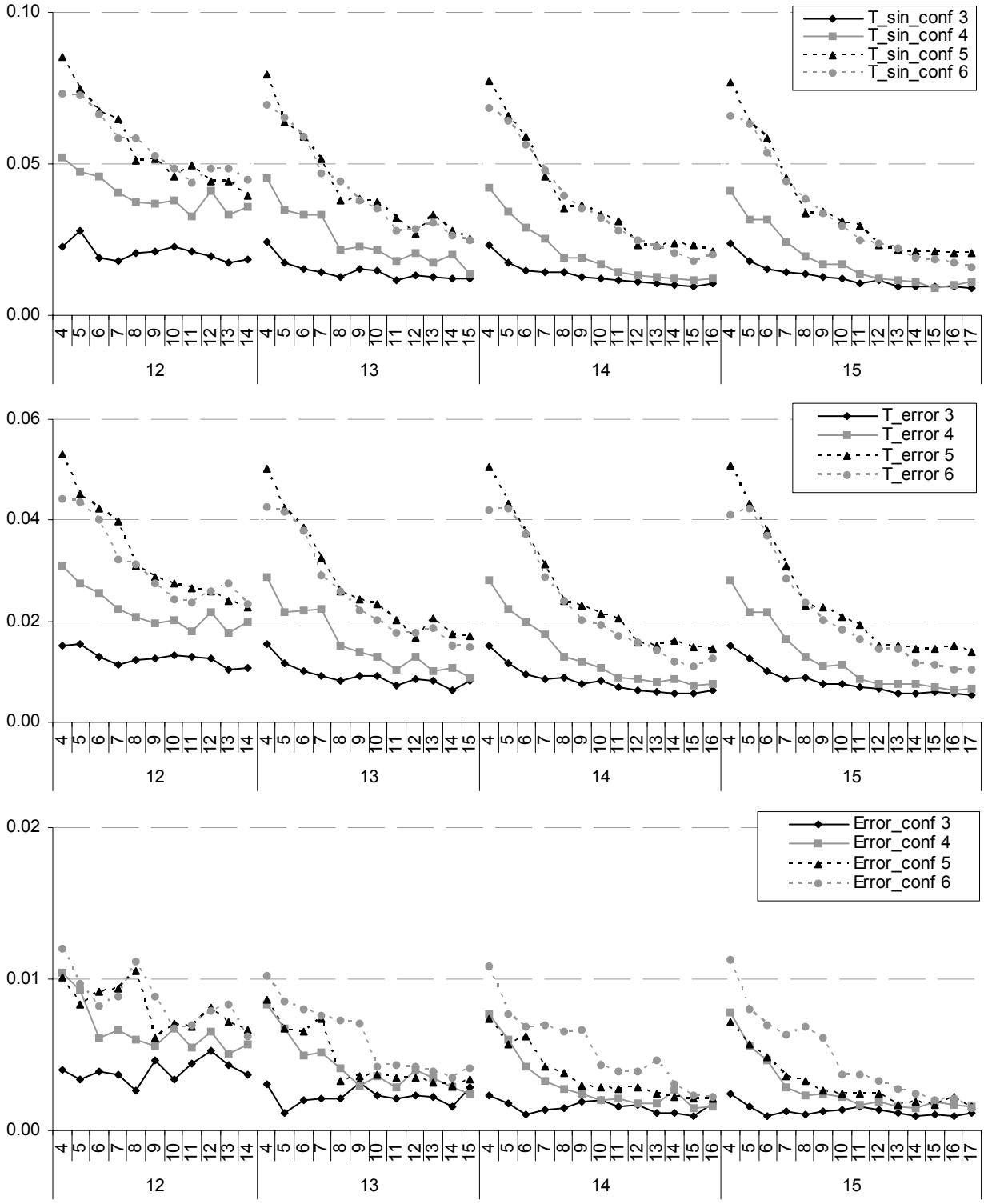

Figura b.10 Resultados predictor de banco EON.

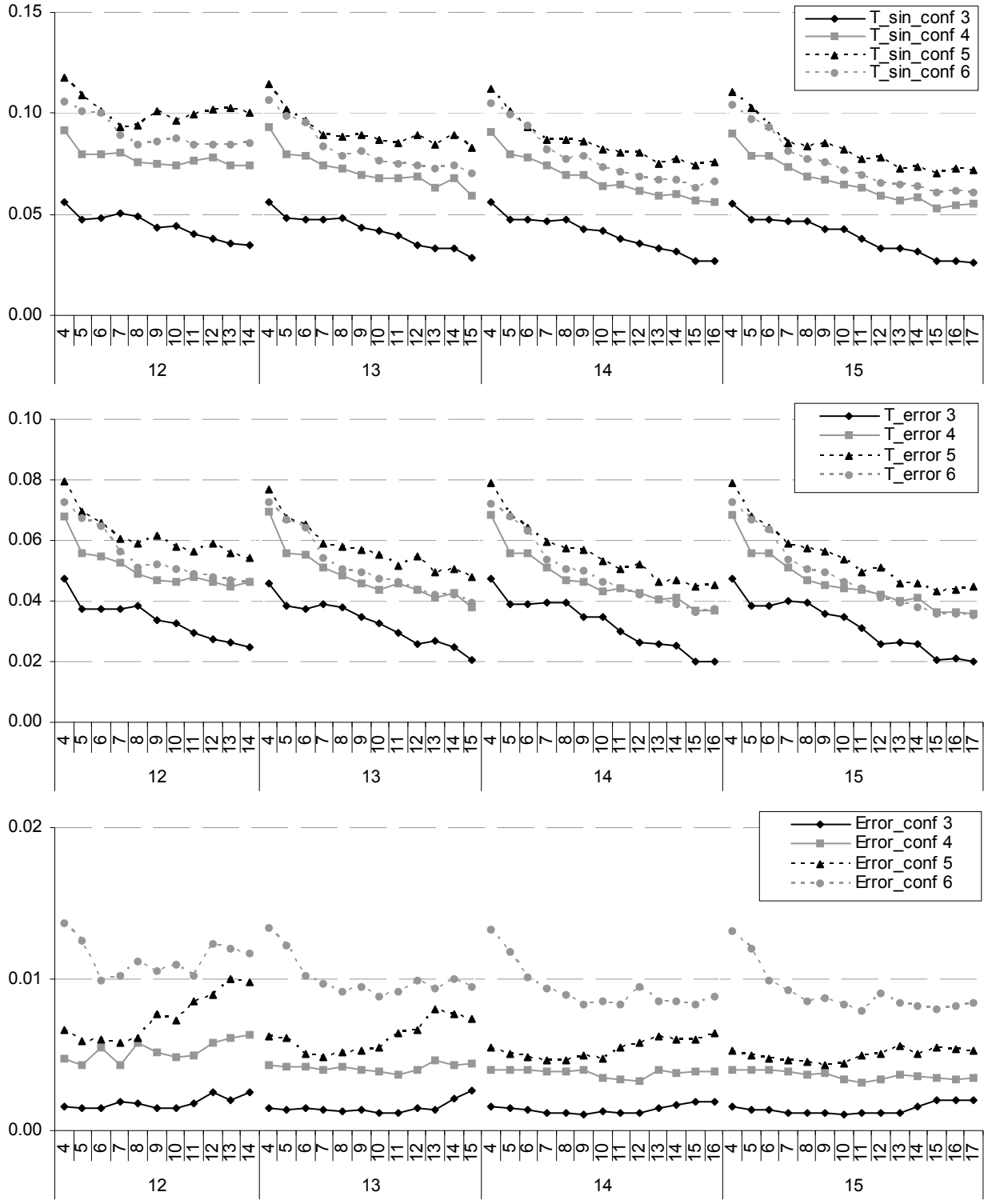

Figura b.11 Resultados predictor de banco GAP.

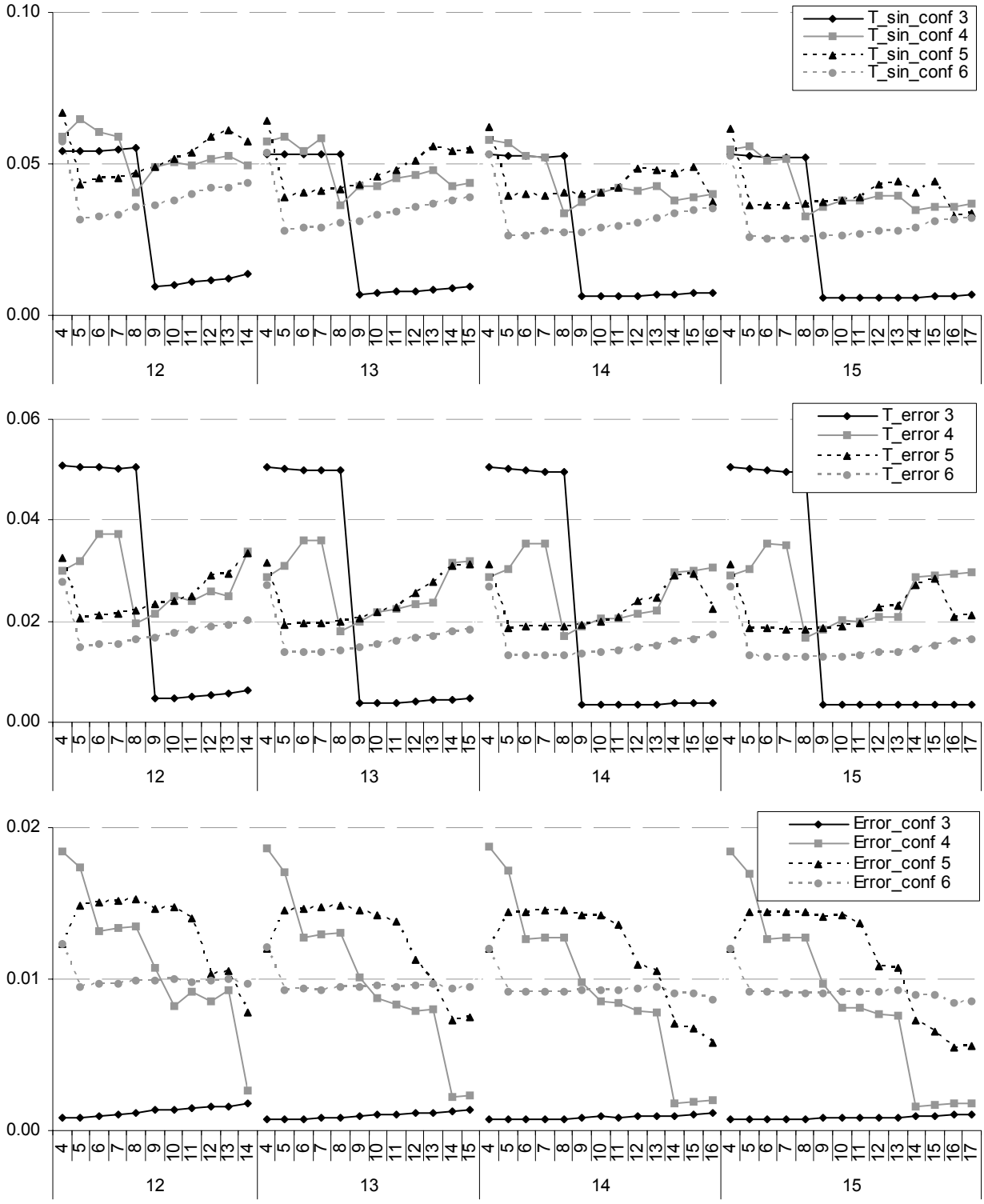

Figura b.12 Resultados predictor de banco GCC.

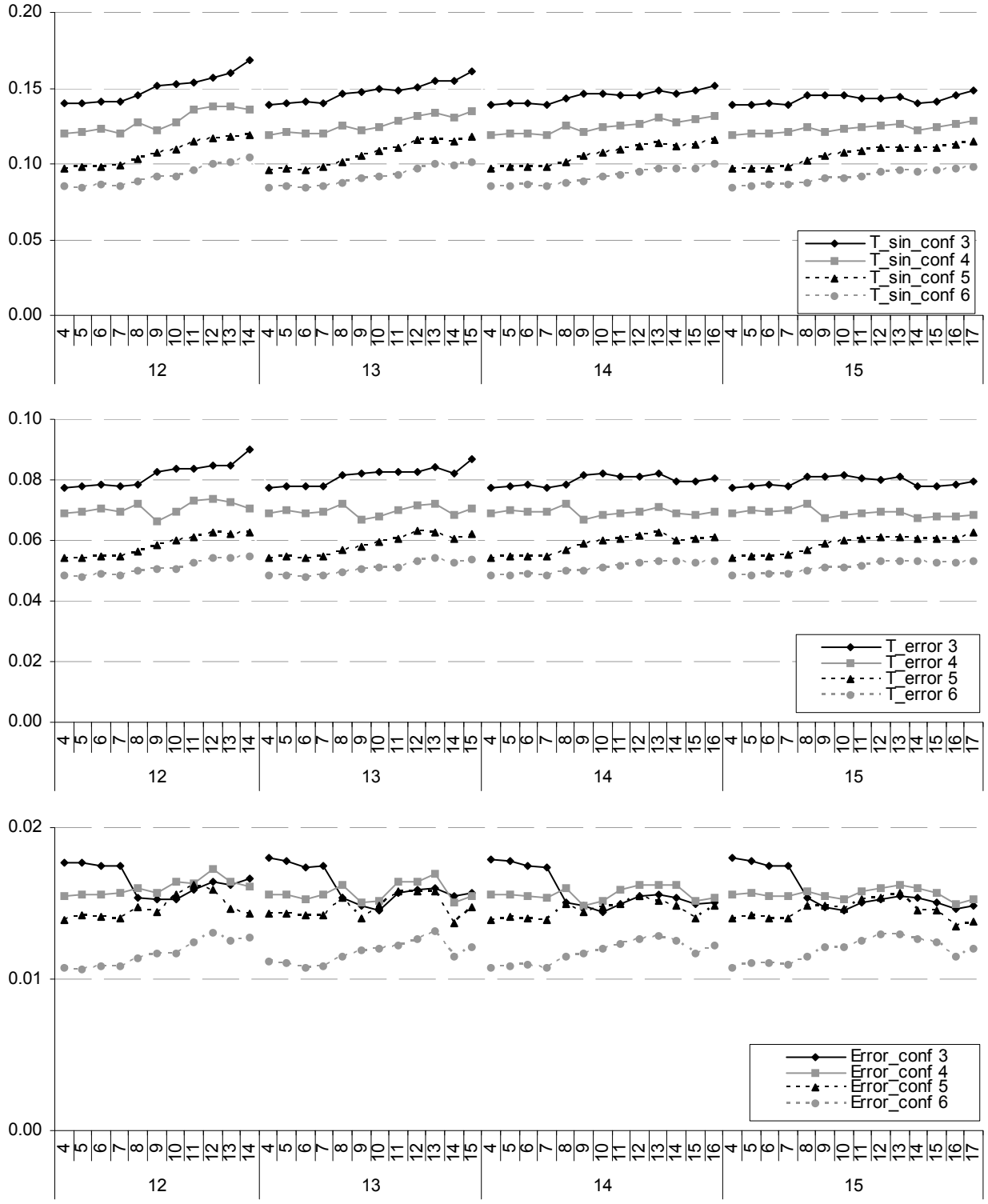

Figura b.13 Resultados predictor de banco GZIP.

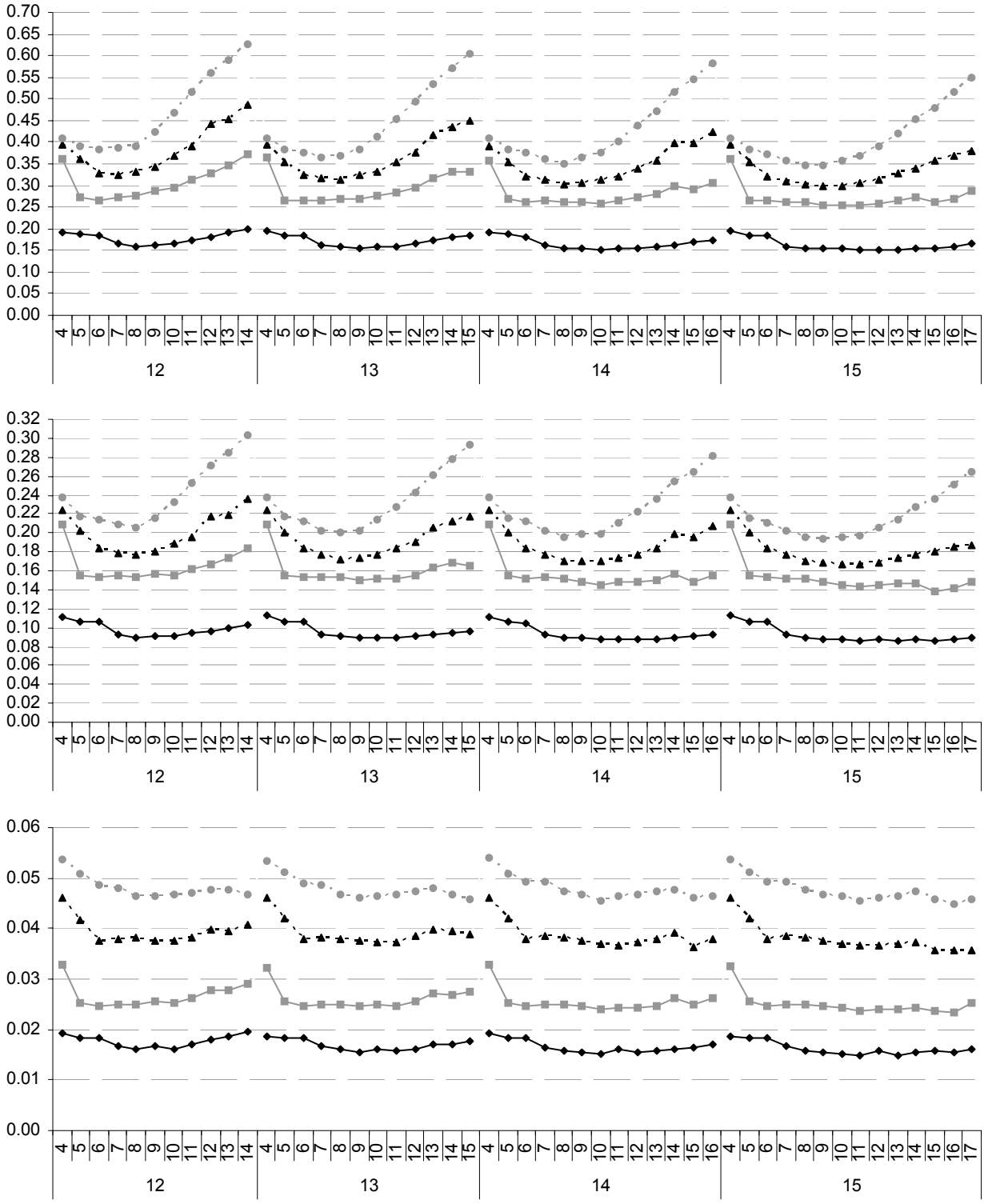

Figura b.14 Resultados predictor de banco `MCF`.

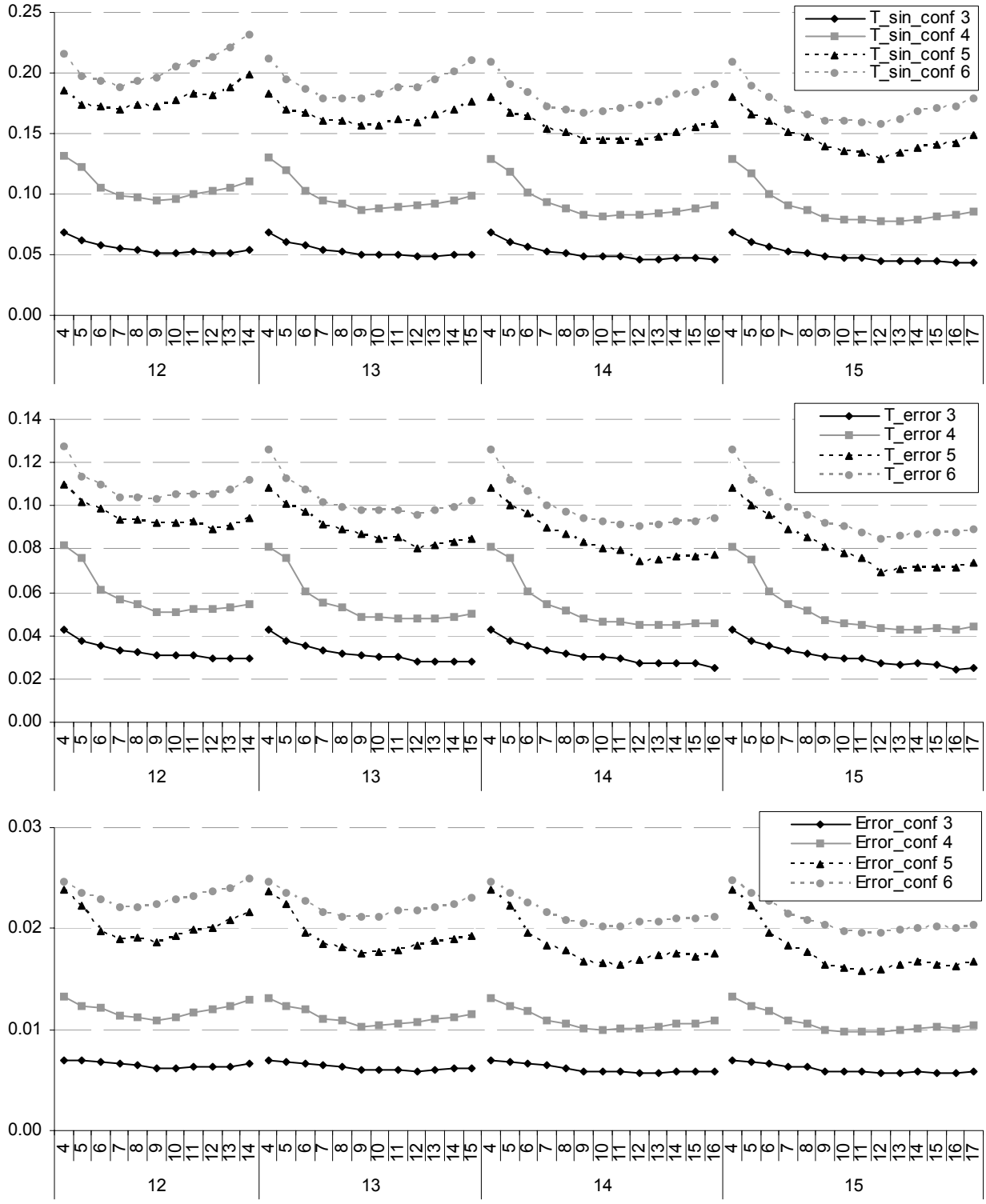

Figura b.15 Resultados predictor de banco PARSER.

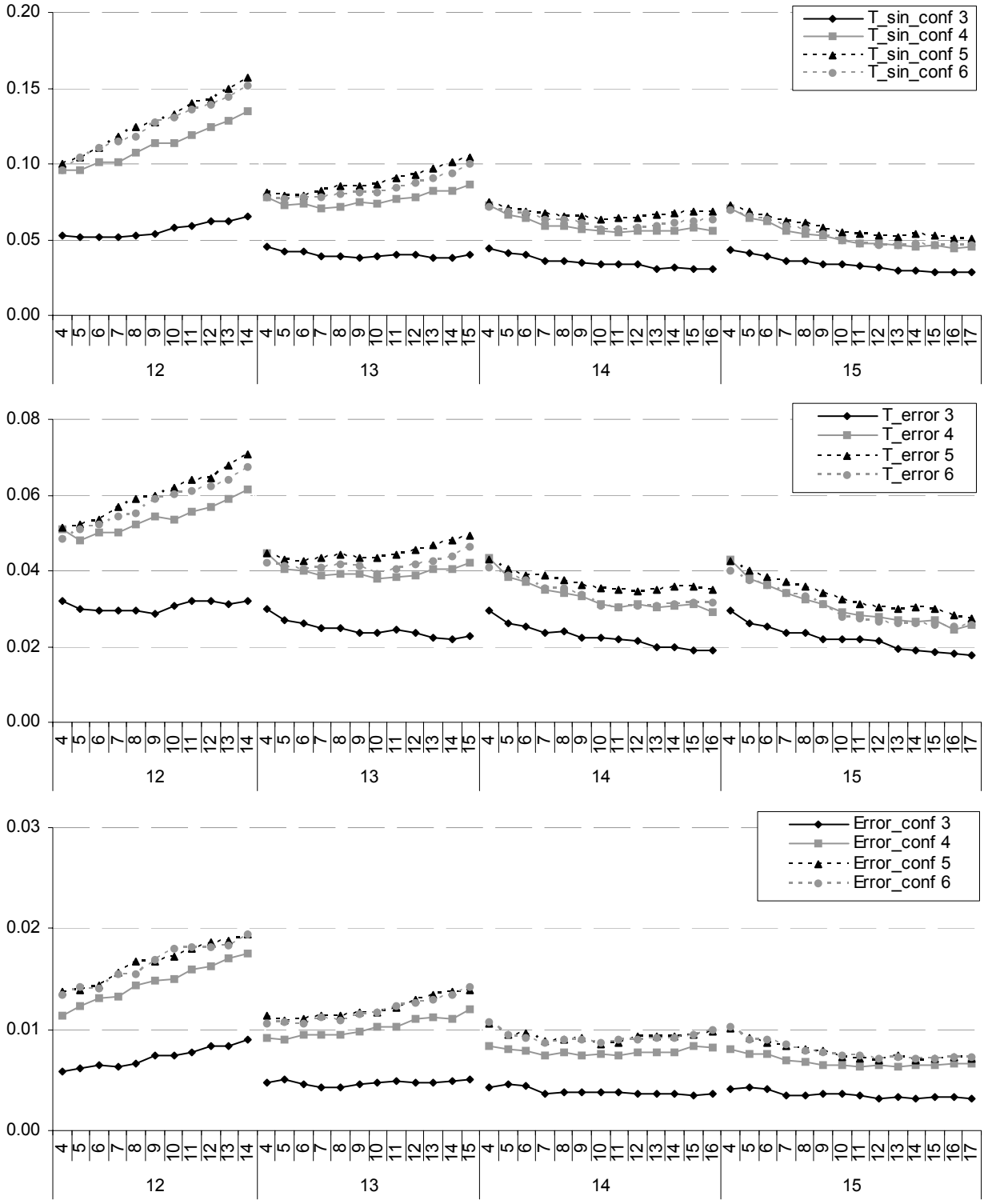

Figura b.16 Resultados predictor de banco `PERL`.

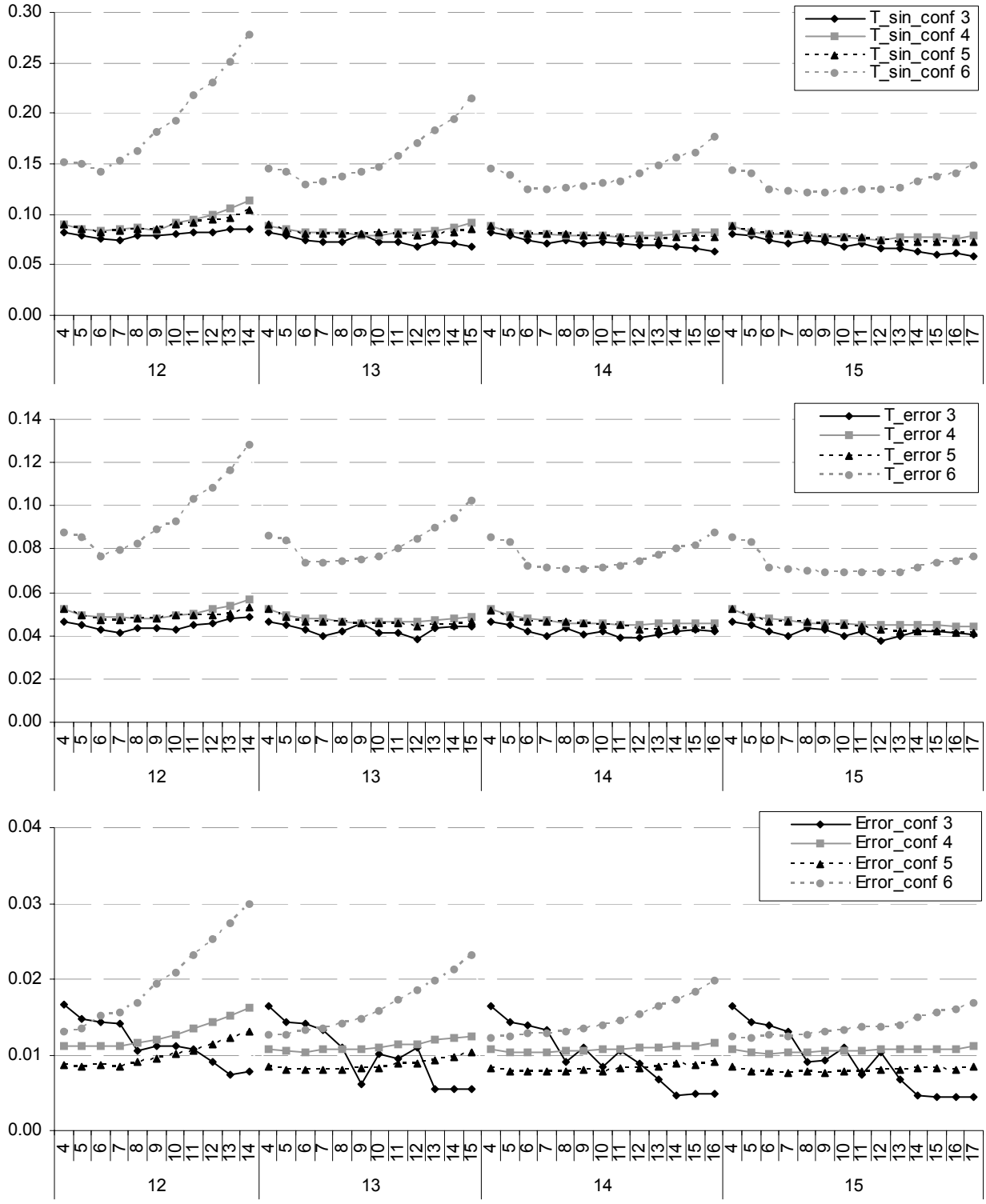

Figura b.17 Resultados predictor de banco `TWOLF`.

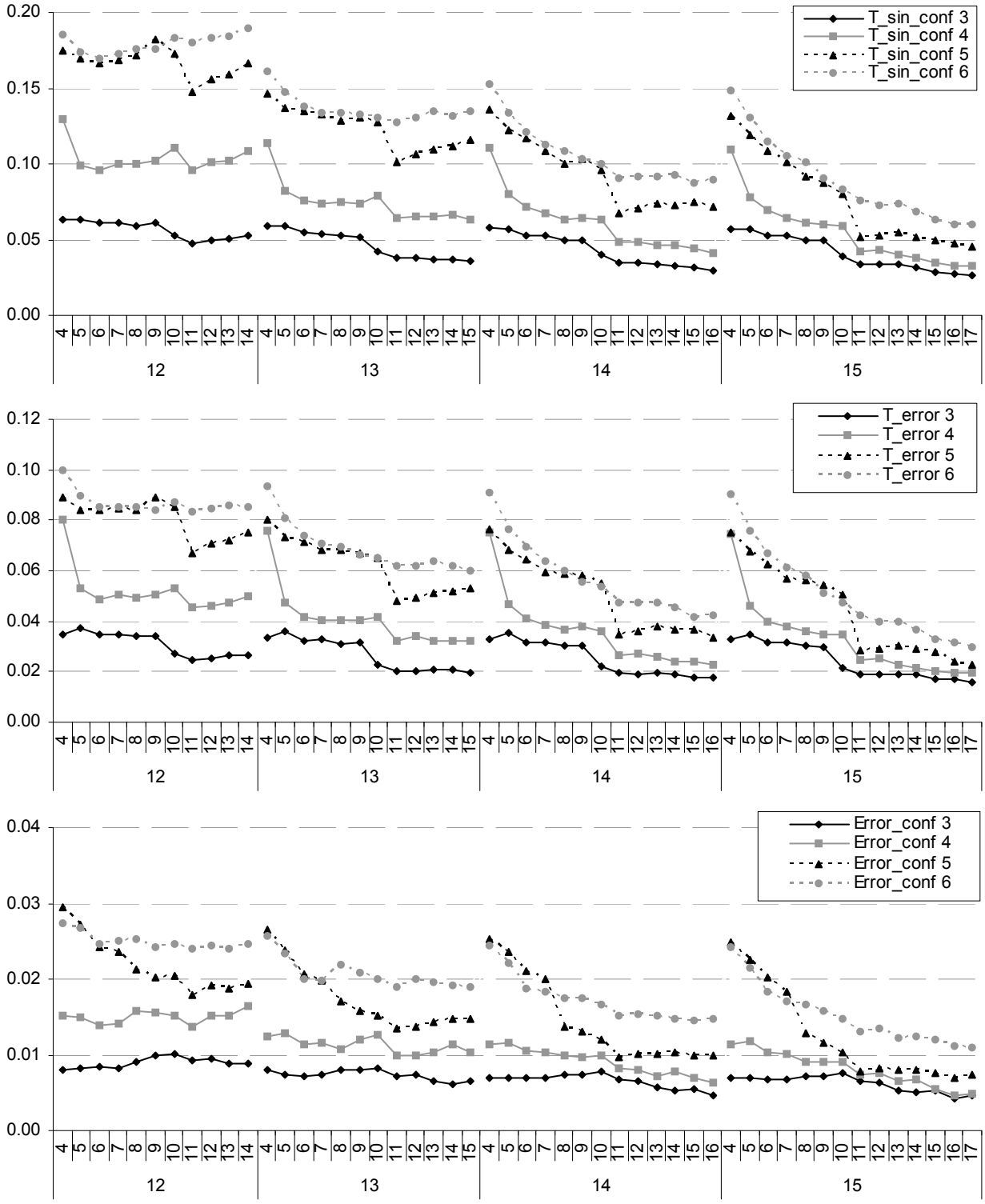

Figura b.18 Resultados predictor de banco VORTEX.

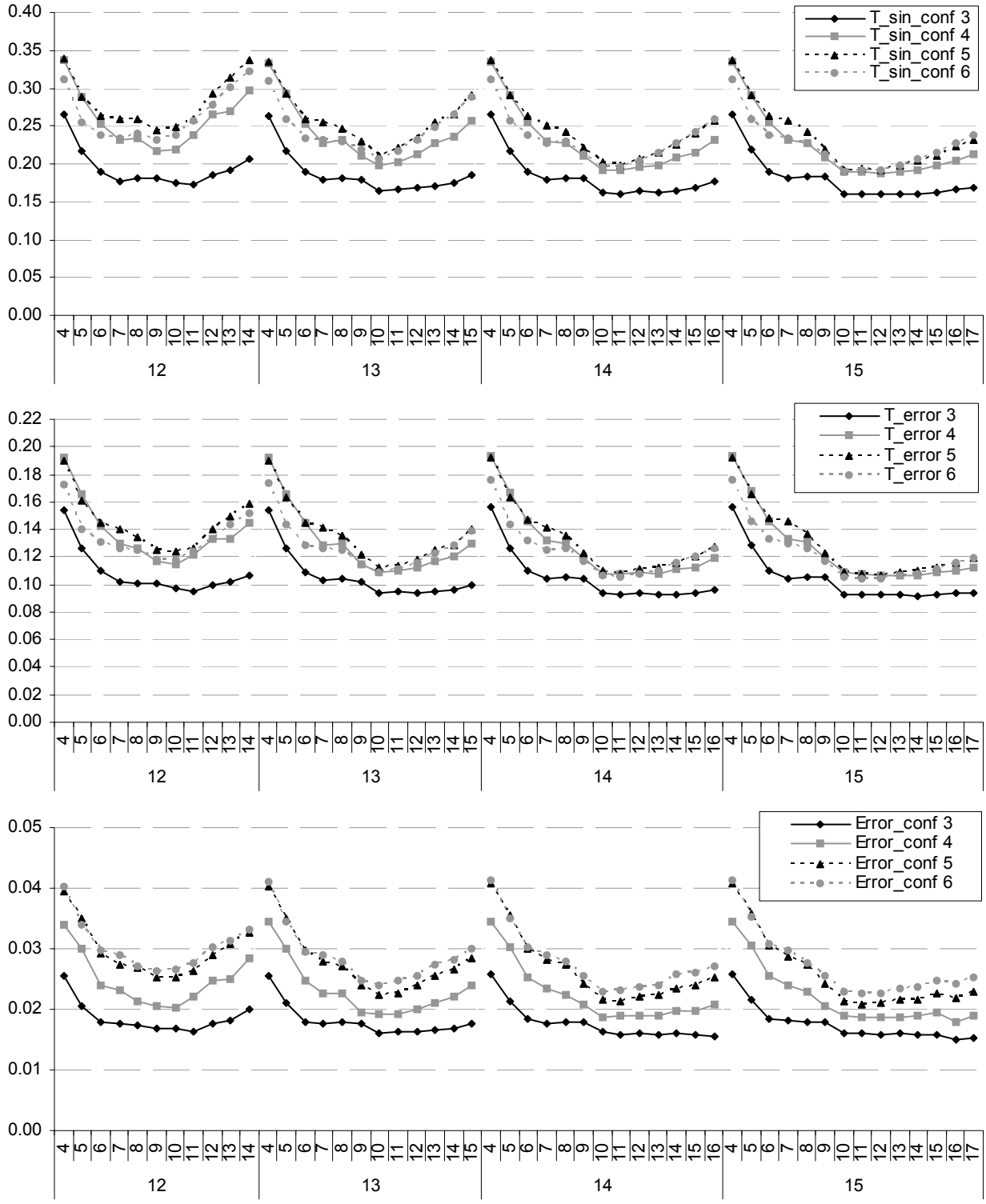

Figura b.19 Resultados predictor de banco VPR.

Listado de Referencias

[AHKB00] V. Agarwal, M.S. Hrishikesh, S.W. Keckler and D. Burger, “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures,” in Proc. of the 27th ISCA, pp. 248-259, June 2000

[AkRS03] H. Akkary, R. Rajwar, and S.T. Srinivasan, “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,” in Proc. of 36th MICRO, pp. 423-434. December 2003.

[AlKo97] D.H. Albonesi and I. Koren. “Improving the Memory Bandwidth of Highly-Integrated, Wide-Issue, Microprocessor-Based Systems,” in Proc. 1997 Conf. on Parallel Architectures and Compilation Techniques (PACT '97).

[Alph99] “Alpha 21264 microprocessor hardware reference manual,” July 1999. Compaq Computer Corporation.

[BaDA03] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, “Dynamically managing the communication-parallelism trade-off in future clustered processors,” in Proc. of the 30th ISCA, pp. 275–287, June 2003.

[BJR+99] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, “Correlated Load-Address Predictors,“ in Proc. of 26th ISCA, pp. 54-63 , May 1999.

[Bloo70] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Communications of the ACM, 13(7):422.426, July 1970.

[Bog+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 90nm Technology,” in Intel Technology Journal, vol. 08, Iss. 01, February 2004

[BTME02] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose Loops Sink Chips," in Proc. of 8th HPCA, pp. 299-310, February 2002.

[BuAu97] D.C. Burger and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," UW Madison Computer Science Technical Report #1342, June 1997.

[Case93] B. Case, "Intel Reveals Pentium Implementation Details", Microprocessor Report vol 7, n 4, pp 1-9, March 29, 1993.

[CaLi04] H. W. Cain, M. H. Lipasti, "Memory Ordering: A Value-Based Approach," in Proc. of 31st ISCA, pp. 90-101, June 2004.

[ChYL01] S. Cho, P. Yew, and G. Lee, "A High-Bandwidth Memory Pipeline for Wide Issue Processors," IEEE Trans. on Computers, vol. 50, no. 7, pp. 709-723, July 2001.

[ChEm98] G.Z. Chrysos and J.S. Emer., "Memory Dependence Prediction Using Store Sets," in Proc. of the 25th ISCA, pages 142–153, June 1998.

[CoLl92] J. Cortadella and J.M. Llabería, "Evaluation of A+B=K Conditions without Carry Propagation," IEEE Trans. on Computers, vol. 41, no. 11, pp. 1484-1488, November 1992.

[COLV04] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors," in Proc. of 10th HPCA, pp. 48-59, February 2004.

[CSVM04] A. Cristal, O.J. Santana, M Valero, and J.F. Martínez, "Toward kilo-instruction processors," ACM Trans. Archit. Code Optim., vol. 1, no 4, pp. 1544-3566, 2004.

[EdRR95] J. Edmondson, P. Rubinfeld, and V. Rajagopalan, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," IEEE Micro, vol. 15, no. 2, pp. 33-43, April 1995.

[FaFi98] J.A. Farrell and T.C. Fisher, "Issue Logic for a 600 Mhz Out-of-Order Execution Microprocessor," IEEE J. of Solid-State Circuits, vol. 33, no. 5, pp. 707-712, May 1998.

[GoVB01] B. Goeman, H. Vandierendonck and K. de Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," in HPCA 2001, pp. 207

[Gwen93] L. Gwennap, "IBM Regains Performance Lead with Power2," Microprocessor Report, vol. 7, no 13, October 4, 1993.

[HSU+01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal Q1, 2001.

[Hsu94] P. Hsu, "Design of the R8000 Microprocessor," IEEE Micro, vol.14, pp. 23-33, April 1994.

[JiLi02] D.A. Jiménez and C. Lin, “Neural methods for dynamic branch prediction,” ACM Trans. Comput. Syst., vol. 20, no 4, pp.369-397, 2002.

[JuNT97] T. Juan, J. Navarro, and O. Temam, “Data Caches For Superscalar Processors",” in Proc. of ICS 97 Viena, Austria, pp. 60-67.

[KMAC94] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, “The AMD Opteron Processor for Multiprocessor Servers,” IEEE Micro, vol. 23, no. 2, pp. 66-76 March/April 2003.

[KeMW98] R.E. Kessler, E.J. MacLellan, and D.A. Webb, “The Alpha 21264 Microprocessor Architecture,” in Proc. of ICCD´98, pp. 90-95. October 1998.

[Kess99] R.E. Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro, vol. 19, no. 2, pp. 24- 36, March/April 1999.

[Kuma97] A. Kumar, “The HP PA-8000 RISC CPU,” IEEE Micro, vol. 17, no. 2, pp. 27-32, March-April 1997.

[LKL+02] A.R. Lebeck, J. Koppanali, T. LI, J. Patwardhan, and E. Rotenberg, “A Large, Fast Instruction Window for Tolerating Cache Misses,” in Proc. of 29th ISCA, pp. 59-70, May 2002.

[LiRS01] D. Limaye, R. Rakvic, and J.P. Shen, “Parallel Cachelets,” in Proc. 19th ICCD, pp. 284-292. September 2001.

[MBB+98] M. Matson, D. Bailey, S. Bell, L. Biro, S. Butler, J. Clouser, J. Farrell, and M. Gowan, “Circuit Implementation of a 600MHz Superscalar RISC Microprocessor,” Proc. of ICCD’98, pp. 104-110, October 1998.

[Matz97] D. Matzke, “Will Physical Scalability Sabotage Performance Gains,” IEEE Computer 30(9): 37-39, September 1997.

[MiSU97] P. Michaud, A. Seznec, and R. Uhlig, “Trading Conflict and Capacity Aliasing in Conditional Branch Predictors,” in Proc. of 24th ISCA, pp. 292-303, June 1997.

[MoLO01] E. Morancho, J.M. LLabería, and A. Olivé, “Recovery Mechanism for Latency Misprediction,” Proc. of PACT-2001, pp. 118-128, Sept. 2001.

[MoBV97] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, “Dynamic speculation and synchronization of data dependences,” in Proc. of the 24th ISCA, pp. 181-193, June 1997.

[MoSo97] A. Moshovos, and G. S. Sohi, “Streamlining Inter-operation Memory Communication via Data Dependence Prediction,” in Proc. of MICRO-30, pp.235-245, December 1997.

[NaHa02] S. Naffziger, and G. Hammond, “The Implementation of the Next Generation 64b Itanium™ Microprocessor,” IEEE J. Solid State Circuits, vol. 37, no. 11, pp. 1448-1460, November 2002.

[NeVB00] H. Neefs, H. Vandierendonck, and K. De Bosschere, “A Technique for High Bandwidth and Deterministic Low Latency Load/Store Accesses to Multiple Cache Banks,” in Proc. of 6th HPCA, pp. 313-324, January 2000.

[PaOV03] Il Park, Chong Liang Ooi, T. N. Vijaykumar, “Reducing Design Complexity of the Load/Store Queue,” in Proc. of 36th MICRO, pp. 411-422. December 2003.

[PoKG01] D. Ponomarev, G. Kucuk, and K. Ghose, “Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources,” in Proc. of MICRO-34th, December 2001

[RaPa03] C. Racunas and Y.N. Patt, “Partitioned First-Level Cache Design for Clustered Microarchitectures,” in Proc. of 17th ICS, pp. 22-31. June 2003.

[RTDA97] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin, “On High-Bandwidth Data Cache Design for Multi-Issue Processors,” in Proc. of MICRO-30th, pp. 46-56, December 1997.

[SDB+03] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, S. W. Keckler, “Scalable Hardware Memory Disambiguation for High ILP Processors,” in Proc. of 36th MICRO, pp. 399-410, December 2003.

[Sezn04] A. Seznec, "Revisiting the Perceptron Predictor," IRISA, publication interne n 1620, may. 2004, 21 pages.

[SFKS02] A. Seznec, S. Felix, V. Krishnan and Y. Sazeides, “Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor,” in Proc. of 29th ISCA, pp. 295-306, May 2002.

[SPHC02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically Characterizing Large Scale Program Behaviour,” in Proc. of ASPLOS, October 2002.

[SoFr91] G.S. Sohi and M. Franklin, “High-Bandwidth Memory Systems for Superscalar Processors,” in Proc. of 4th ASPLOS, pp. 53-62, April 1991.

[SRA+04] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, “Continual flow pipelines,” in Proc. of ASPLOS, pp. 107-119, Octuber 2004.

[TaSk04] D. Tarjan and K. Skadron, “Revisiting the Perceptron Predictor Again,” UNIV. OF VIRGINIA DEPT. OF COMPUTER SCIENCE TECH. REPORT CS-2004-28, SEPT. 2004

[TIVL03] E. Torres, P.E. Ibañez, V. Viñals, and J.M. Llabería, “Counteracting Bank Mispredictions in Sliced First-Level Caches,” 9th EuroPar, LNCS 2790, pp. 586-596, September 2003.

[TIVL04] E. Torres, P.E. Ibañez, V. Viñals, and J.M. Llabería, “Contents Management in First-Level Multibanked Data Caches,” 10th EuroPar, LNCS 3149, pp. 516-524, September 2004.

[TIVL05] E. Torres, P.E. Ibañez, V. Viñals, and J.M. Llabería, “Store Buffer Design in First-Level Multibanked Data Caches,” to appear in Proc. of 32th ISCA, pp xxx-xxx, June 2005.

[TDF+01] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. S. aroy, “POWER4 system microarchitecture,” in IBM Journal of Research and Development, 26(1):5-26, January 2001.

[WiOl01] K. Wilson and K. Olukotun, “High Bandwidth On-Chip Cache Design,” IEEE Transactions on Computers, Vol. 50, No. 4, pp. 292-307, April 2001

[YMRJ99] A. Yoaz, E. Mattan, R. Ronen, and S. Jourdan., “Speculation Techniques for Improving Load Related Instruction Scheduling,” in Proc. of 26th ISCA, pp. 42-53, May 1999.

[ZyKo01] V. Zyuban and P.M. Kogge, “Inherently Lower-Power High-Performance Superscalar Architectures,” IEEE Trans. on Computers, vol. 50, no. 3, pp. 268-285, March 2001.