UNIVERSIDAD DE ZARAGOZA

Tesis Doctoral

Zaragoza, Octubre de 2009

Alternativas de Diseño en Sistemas de Prebúsqueda *Hardware* de Datos

D. Luis Manuel Ramos Martínez

Directores:

Dr. José Luis Briz Velasco

Dr. Pablo Ibáñez Marín

DEPARTAMENTO DE INFORMÁTICA E INGENIERÍA DE SISTEMAS

Alternativas de Diseño en Sistemas de Prebúsqueda *Hardware* de Datos

Memoria presentada por

D. Luis Manuel Ramos Martínez

para obtener el título de Doctor en Informática

Departamento de Informática e Ingeniería de Sistemas

Zaragoza, Octubre de 2009

Dirigida por:

Dr. José Luis Briz Velasco

Dr. Pablo Ibáñez Marín

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

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

Resumen de la Tesis

Una solución interesante al problema de la creciente latencia de acceso a memoria es la prebúsqueda de datos. Gracias a las técnicas de prebúsqueda de datos, la latencia de los accesos puede ser ocultada solapándola con la ejecución de las instrucciones anteriores. A diferencia de la prebúsqueda software, la prebúsqueda hardware obtiene información dinámica de la ejecución de la aplicación, lo que en general permite predecir un mayor número de direcciones de acceso a memoria.

Aunque las bases de esta prebúsqueda fueron propuestas hace décadas, la investigación en estos métodos continúa abierta por una serie de razones. En primer lugar su rendimiento depende de los programas y del hardware (procesador y jerarquía de memoria), por lo que no existe un método universal perfecto. Además ambos elementos cambian continua y significativamente, lo que conlleva la continua revisión de métodos de prebúsqueda que, buenos o malos en el pasado, pueden cambiar su carácter con las nuevas aplicaciones o el nuevo hardware. Finalmente, la industria sólo ha incorporado las técnicas de prebúsqueda más sencillas, no necesariamente las más eficaces, sin duda debido a problemas de coste y diseño.

El objetivo de este trabajo ha sido investigar nuevos mecanismos de prebúsqueda hardware de datos que consigan aumentar las prestaciones de los ya conocidos, pero manteniendo siempre un bajo coste de implementación y extremando la simplicidad en el diseño, a fin de hacerlos atractivos para la industria. Para ello, se ha analizado el comportamiento de las instrucciones de acceso a memoria, estableciendo nuevos modelos de regularidad. También se han definido nuevos mecanismos de prebúsqueda y se han evaluado sus rendimientos en un procesador superescalar agresivo y con una jerarquía de memoria de alto rendimiento, comparándolos con los actuales. Por otro lado, se han propuesto técnicas para adaptar la agresividad de prebuscadores muy

agresivos con el objeto de minimizar las pérdidas por aplicación. Por último, se ha presentado un esquema general de prebúsqueda que permite ser adaptado a distintos objetivos de diseño. La evaluación de esta propuesta se realizó sobre el entorno determinado por el First JILP Data Prefetching Championship (DPC-1), en el que el trabajo obtuvo el Best Paper Award y el tercer puesto en prestaciones.

Palabras Clave- microarquitectura, jerarquía de memoria, caracterización de aplicaciones, modelos de regularidad de direcciones, prebúsqueda hardware de datos, prebúsqueda Secuencial Marcado, prebuscadores correlacionados.

a Javi y Erika

Agradecimientos

A todos los que de una forma u otra me han apoyado durante la elaboración de esta Tesis, en especial a José Luis, GRACIAS.

Gracias también a las siguientes entidades y proyectos, cuyo apoyo en financiación y recursos computacionales ha hecho posible este trabajo:

- Proyectos de I + D financiados en convocatorias públicas nacionales del Ministerio de Ciencia y Tecnología, Plan Nacional de I + D + I, Programa PRONTIC. 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.
- *Jerarquía de memoria de alto rendimiento*. TIN2007-66423.
- *gaZ*: Reconocimiento Grupo Emergente, Diputación General de Aragón años 2003 - 2009.
- HiPEAC: European N.E. on High-Performance Embedded Architecture and Compilation.
- HiPEAC2 NoE: European Network of Excellence on High Performance and Embedded Architecture and Compilation.
- Recursos computacionales que se listan a continuación:
  - DIIS: Departamento de Informática e Ingeniería de Sistemas.
  - DIEC: Departamento de Ingeniería Electrónica y Comunicaciones.
  - CEPBA: Centro Europeo de Paralelismo de Barcelona.
  - gaZ: grupo de arquitectura de Zaragoza.
  - I3A: Instituto de Investigación en Ingeniería de Aragón.

Contenido

CAPÍTULO 1

Introducción

En este capítulo se realiza una introducción al tema principal de la Tesis, la prebúsqueda hardware de datos. En primer lugar se exponen la motivación y el contexto tecnológico en el que se enmarca el trabajo realizado. A continuación, se realiza una descripción de trabajos relacionados. Finalmente, se resumen las contribuciones desarrolladas y se detalla la organización de la memoria.

1.1 Introducción

El tiempo de ejecución que los procesadores dedican al acceso a datos en memoria es muy alto, y es más que probable que siga creciendo [WBM+03] [ABI+06]. La prebúsqueda de datos ha demostrado ser una forma efectiva de ocultar la latencia de memoria. Esto es debido principalmente a la regularidad que presenta la secuencia de direcciones de las instrucciones de acceso a memoria, que permite predecir direcciones futuras y prebuscarlas con antelación.

La prebúsqueda de datos puede realizarse usando instrucciones especiales en el código (prebúsqueda *software*) o colocando un mecanismo *hardware* en el procesador que actúa según el comportamiento del programa (prebúsqueda *hardware*). La prebúsqueda *software* no requiere la modificación del procesador. Además, la mayoría de arquitecturas de lenguaje máquina actuales incluyen instrucciones de prebúsqueda, pero en ocasiones los compiladores no conocen el

comportamiento dinámico de un programa operando sobre una jerarquía de *cache* específica. La prebúsqueda *hardware* requiere la modificación del procesador, pero permite aprovechar la información generada por el programa mientras se ejecuta.

Recientemente se han realizado varias propuestas de éxito de prebúsqueda *hardware*, como por ejemplo las prebúsquedas basadas en el *Global History Buffer* (GHB) o nuevos prebuscadores de secuencias enfocados a servidores, como el *Spatial Memory Streaming* (SMS).

Sin embargo, si se observan los microprocesadores que están actualmente en el mercado, sólo se implementan los sistemas de prebúsqueda *hardware* más sencillos: por ejemplo prebúsqueda secuencial en UltraSPARC-IIIcu [Sun04], *stream buffers* secuenciales en Power4, Power5 y Power6 [TDF+02] [KST04], prebúsqueda secuencial y *stride* en microarquitecturas Intel [Dow06] y prebúsqueda secuencial en SPARC64 VI [Kre03].

Por otro lado, las memorias *cache on-chip* han experimentado numerosos desarrollos relativos a su capacidad y ancho de banda. La polución debida a la prebúsqueda ha sido reducida, gracias a las nuevas memorias *cache* multinivel de varios MB de capacidad [ABI+06]. La inclusión de varios niveles de *cache on-chip* incrementa el ancho de banda entre ellos, y las nuevas tecnologías DRAM (DDR2, DDR3, RAMBUS y XDR) incrementan el ancho de banda entre microprocesador y memoria principal. Esto hace que la actividad de prebúsqueda no interfiera negativamente con el resto del sistema, o lo haga en menor medida. Una tendencia actual en algunos microprocesadores integra el controlador de memoria dentro del *chip*, permitiendo estrategias de planificación adaptadas para prebúsqueda. También puede integrarse un prebuscador en el controlador.

1.2 Motivación

Los sistemas de prebúsqueda demandan distintos requerimientos: área del *chip*, energía, ancho de banda, espacio de almacenamiento,... Considerando las nuevas oportunidades que ofrecen las actuales *caches on-chip* multinivel y el controlador de memoria, es conveniente realizar una revisión de los mecanismos de prebúsqueda conocidos centrándose en dichos recursos.

El objetivo principal de esta Tesis es proponer y evaluar nuevos mecanismos de prebúsqueda *hardware* que suministren los datos al procesador con la precisión

y puntualidad requeridas, de forma que puedan ser aprovechados por las siguientes generaciones de procesadores de alto rendimiento, y estén sujetos a compromisos realistas de coste y complejidad que los haga atractivos para la industria.

Para desarrollar dicho objetivo, en esta Tesis se han seguido las siguientes líneas de trabajo:

- Analizar los modelos de regularidad de direcciones actuales y medir su cobertura.

- Definir nuevos modelos de regularidad dirigidos a incrementar la cobertura en un tipo concreto de aplicaciones donde actualmente es baja; examinar su utilidad para la prebúsqueda *hardware* de datos.

- Comparar entre sí el comportamiento de técnicas de prebúsqueda conocidas en un simulador detallado de un procesador de alto rendimiento y con una jerarquía de memoria multinivel de alta capacidad y gran ancho de banda.

- Definir nuevas técnicas de prebúsqueda (basadas tanto en modelos de regularidad de contexto como computacionales) y comparar su rendimiento con las ya conocidas.

- Investigar mecanismos simples y atractivos para la industria que ajusten la agresividad *y* permitan reducir pérdidas en aplicaciones puntuales sin reducir las prestaciones medias; analizar sus rendimientos comparándolos con los de otras técnicas de prebúsqueda.

- Definir un esquema general de sistema de prebúsqueda que pueda ajustarse a distintos objetivos de diseño.

- Comparar nuestras propuestas de prebúsqueda con las de otros investigadores en un entorno de simulación independiente.

En el siguiente apartado (sección 1.3) se pone en contexto la Tesis, realizando una descripción general de un sistema de prebúsqueda *hardware* de datos y describiendo tres ejemplos de prebuscadores ya conocidos que sirven para ilustrar tres orientaciones típicas muy diferentes. La sección 1.4 presenta una selección de trabajos relacionados. Después, la sección 1.5 resume las contribuciones de esta Tesis. Finalmente, la sección 1.6 detalla la organización de la memoria.

1.3 Prebúsqueda *hardware* de datos

1.3.1 Funcionamiento general de un sistema de prebúsqueda

Con el objetivo de disminuir la latencia media de acceso a memoria, los procesadores actuales disponen de una jerarquía de memoria *cache* de dos o tres

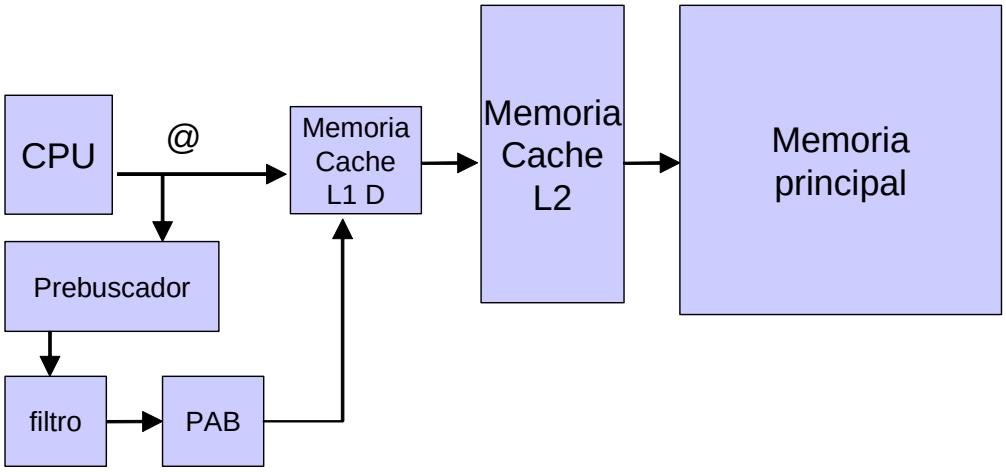

Figura 1.1 Ejemplo de jerarquía de memoria de dos niveles.

niveles (Figura 1.1). Los niveles cercanos al procesador son pequeños y rápidos. Cuanto más cerca están de la memoria principal disponen de más capacidad, pero también aumenta su latencia de acceso. Los programas que presentan localidad temporal y localidad espacial se benefician de una disminución en las latencias de acceso a los datos.

Cada vez que se ejecuta una instrucción de acceso a memoria (*load* o *store*) se comienza mirando en el nivel más cercano. Si ese dato, u otro perteneciente al mismo bloque, se ha referenciado recientemente y no ha sido expulsado, se producirá un acierto (o *hit*) y la latencia del acceso será pequeña; si no se producirá un fallo (o *miss*). En este caso, la política de expulsión establece cuál es el bloque a expulsar (habitualmente el menos recientemente utilizado) y se solicita el bloque al siguiente nivel. Cuando esta solicitud finaliza se produce el rellenado (o *refill*) en el contenedor seleccionado y se sirven los datos solicitados. A veces a un nivel de *cache* llega una solicitud de un bloque que ya ha sido solicitado previamente al siguiente nivel. Esto es lo que denominaremos*fallo secundario*.

La lógica del programa hace que las instrucciones de acceso a memoria muestren cierta regularidad en su secuencia de direcciones; es lo que se llama patrón o modelo de regularidad. Los sistemas de prebúsqueda *hardware* estudian de forma dinámica la secuencia de direcciones lanzada desde el procesador con el fin de reconocer patrones que permitan predecir las direcciones futuras. Si el sistema prebusca con cierta anticipación las direcciones predichas, la latencia de memoria del acceso se solapará con las instrucciones previas, con el consiguiente ahorro en ciclos y aumento de prestaciones en la ejecución del programa. Las prebúsquedas generadas por el mecanismo de prebúsqueda son enviadas a un *Prefetch Address Buffer*, donde esperan a ser lanzadas a memoria, ya que habitualmente tienen menos prioridad que las peticiones por demanda.

Cuando el sistema de prebúsqueda funciona correctamente el bloque prebuscado es usado posteriormente por una instrucción de acceso a memoria; es lo que se llama prebúsqueda *útil*. Sin embargo, en ocasiones ocurre que el sistema prebusca bloques que posteriormente no son utilizados; se llaman prebúsquedas *inútiles*. Éstas generan polución en *cache* y consumen ancho de banda, lo que puede perjudicar a las peticiones por demanda. Otras veces la prebúsqueda es correcta, pero se ha lanzado demasiado tarde y su respectiva demanda provoca un fallo secundario; en este caso la prebúsqueda es *útil* pero *tardía*. Si la antelación con la que se lanza es suficiente, la demanda acierta en *cache* y la prebúsqueda es *puntual*. También es posible que la antelación de la prebúsqueda sea demasiado grande y el bloque prebuscado sea expulsado de la *cache* antes de la demanda; en este caso se trata de un prebúsqueda *temprana*.

Nótese que cada nivel de *cache* puede tener su propio mecanismo de prebúsqueda, adaptado a sus características.

En ocasiones, algunos mecanismos de prebúsqueda son tan simples que generan muchas direcciones repetidas. Es estos casos se necesita algún mecanismo de filtrado de prebúsquedas redundantes.

La agresividad de un prebuscador puede ser parametrizada a través de su *grado* y *distancia*. El *grado* de un prebuscador indica el número de prebúsquedas consecutivas que se generan cada vez. La *distancia de prebúsqueda* indica la lejanía de la dirección desde la que comienzan a generarse las prebúsquedas. En general un prebuscador más agresivo conseguirá más ganancias (en los programas en los que funciona bien) y más pérdidas (en los que va mal) que uno menos agresivo. Esto hace que los prebuscadores muy agresivos consigan las

mayores ganancias medias, aunque obteniendo pérdidas considerables en algunos de los programas.

Para medir la eficiencia de un sistema de prebúsqueda en la ejecución de una aplicación se utilizan las siguientes métricas:

- **cobertura**: porcentaje de direcciones generadas por la aplicación que han sido prebuscadas por el sistema;
- **precisión**: porcentaje de prebúsquedas generadas que han sido usadas por la aplicación;
- **puntualidad**: porcentaje de prebúsquedas *puntuales* sobre el total;
- **rendimiento** (*speed-up*): $IPC_{\text{con prebúsqueda}} / IPC_{\text{sin prebúsqueda}}$.

Evidentemente, de todas ellas la más importante es la última, ya que mide el objetivo buscado en sí mismo.

Cuando se evalúa un sistema de prebúsqueda sobre una colección de aplicaciones, es habitual el uso de medias geométricas: media geométrica de *speed-ups* y media geométrica de tiempos normalizados de ejecución.

Describimos a continuación tres prebuscadores elegidos a modo de referencias concretas de lo acaba de exponerse. Corresponden a tres formas diferentes de orientar la prebúsqueda, históricamente muy separadas entre sí. En cierta medida pueden considerarse claves que ayudarán a situar el resto de trabajos relacionados que se exponen más tarde. De diferente manera aparecen y se utilizan a lo largo de prácticamente toda la Tesis.

1.3.2 Prebuscador *secuencial marcado*

El prebuscador *secuencial marcado* [Smi78] se considera un prebuscador clásico, muy útil cada vez que el programa recorre un vector de forma secuencial. Cada vez que se produce un fallo o que un bloque prebuscado es usado por primera vez, se solicita la prebúsqueda del siguiente bloque. Es muy efectivo en una amplia variedad de aplicaciones y tiene un coste muy bajo, ya que sólo se necesita 1 bit por bloque de cache. Sin embargo, también es habitual que genere muchas prebúsquedas que finalmente no son usadas por la CPU, pudiendo generar pérdidas significativas en aplicaciones concretas.

1.3.3 Prebuscador *stride*

El prebuscador *stride* es otro de los prebuscadores bien conocidos [ChB94], que muestra su utilidad cada vez que el programa accede a objetos que se encuentran ubicados en memoria de forma homogénea. Este prebuscador usa una tabla donde a cada instrucción de acceso a memoria, *load* o *store*, se le intenta asociar el *stride* (diferencia entre direcciones) seguido. Cada vez que se lanza una instrucción de acceso a memoria se accede a la tabla para consultar su *stride*. Si el *stride* guardado es igual al actual se lanza una prebúsqueda sumando dicho *stride* a la dirección actual. Si no se encuentra, o no coincide, no se lanza prebúsqueda y se actualiza la tabla.

Este prebuscador es más selectivo que el anterior, ya que guarda información asociada a la dirección de las instrucciones. Por ello en general consigue menor ganancia media, aunque es raro que presente pérdidas.

1.3.4 Un prebuscador de correlación basado en GHB: PC/DC

Los prebuscadores de correlación guardan información en tablas sobre los accesos a memoria e intentan predecir las futuras direcciones a acceder. Pueden ser considerados como una generalización de los prebuscadores *stride*. Generalmente estos métodos guardan las secuencias de direcciones que siguen a un fallo, o las secuencias de diferencias (*deltas*) de direcciones. Estas secuencias pueden ser asociadas a una dirección de fallo o al PC de la instrucción de acceso a memoria. Desde la aparición de los primeros predictores de Markov aplicados a prebúsqueda han surgido múltiples variaciones de estas ideas. Sin embargo, estos prebuscadores tienen una desventaja clara: el tamaño de las tablas suele ser enorme y la información guardada suele quedarse desactualizada. Los prebuscadores basados en GHB (*Global History Buffer*) fueron los primeros en solventar estos problemas y conseguían guardar más información sobre las instrucciones más frecuentes a la vez que mantenían el tamaño de las tablas bajo. Sin embargo, estos prebuscadores presentan otro problema, el gran número de accesos a las tablas necesarios para hacer una predicción y en consecuencia la latencia de dicha predicción.

El prebuscador de correlación basado en GHB que vamos a utilizar en esta Tesis es PC/DC. Este prebuscador consta de 2 tablas, una de ellas es el GHB, la otra es una tabla de índices que apuntan a entradas del GHB (Figura 1.2). El GHB actúa como una estructura FIFO en la que se encolan las direcciones de los

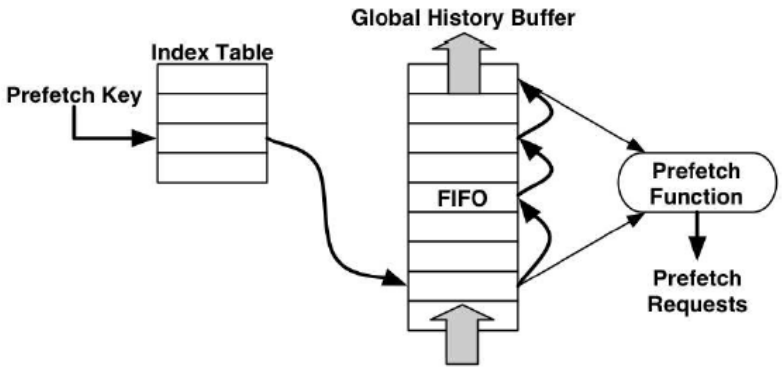

Figura 1.2 Estructura del prebuscador PC/DC.

últimos fallos que se han producido. Aquellas que han sido producidas por la misma instrucción están encadenadas a través de punteros, formando una lista. La primera tabla es indexada con el PC de la instrucción y contiene un puntero a la entrada de la GHB donde se encuentra el último fallo producido por dicha instrucción. Cada vez que se produce un nuevo fallo: 1) se inserta la dirección de fallo en el GHB y se encadena con el fallo anterior producido por dicha instrucción; 2) se actualiza el puntero de la tabla de índices; 3) se recorre la lista de fallos de dicha instrucción en el GHB calculando los *deltas* entre direcciones de fallos consecutivas y almacenándolos en una estructura llamada *Delta Buffer*; 3) los *deltas* calculados se van comparando con los dos primeros *deltas* de la lista; y 4) en caso de igualdad se ha encontrado un patrón. Finalmente, los *deltas* se extraen del *Delta Buffer,* uno por ciclo, para ser sumados a la última dirección lanzada por la instrucción, calculando así las direcciones a prebuscar.

En la Figura 1.3 se muestra un ejemplo de funcionamiento de un prebuscador PC/DC. Supongamos un código donde varias instrucciones de acceso a memoria (A, B, C, D, E, F) han ido produciendo algunos fallos cuyas direcciones han sido encoladas en el GHB. El último fallo de cada instrucción está apuntado desde la *Index Table*, con lo que al producirse un nuevo fallo puede encadenarse al anterior. Se puede observar cómo el último fallo producido ha sido efectuado por la instrucción C a la dirección 09. Partiendo de esta entrada de la tabla existe una cadena de punteros que se puede recorrer para conocer las últimas direcciones falladas por dicha instrucción en orden inverso al que se produjeron (09 08 06 05 04 02 01), ir calculando los *deltas* (1 2 1 1 2 1) e insertándolos en el *Delta Buffer.* Los dos primeros *deltas* obtenidos se guardan además en un registro (*Correlation Key Register* en la figura). Según se

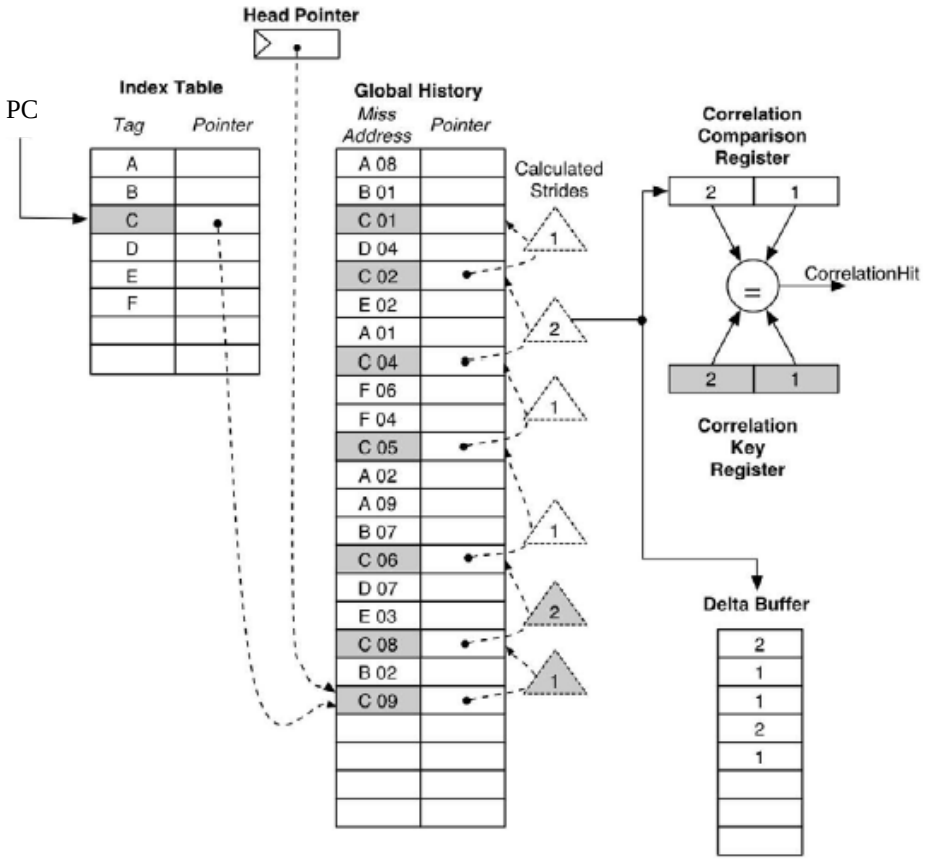

Figura 1.3 Ejemplo de funcionamiento de un prebuscador PC/DC.

van generando el resto de *deltas*, se van comparando con los dos primeros. En caso de coincidir se ha encontrado un patrón de *deltas* repetidos. Entonces se extraen dos *deltas* del *Delta Buffer* y los que quedan son la predicción de los próximos *deltas* que se producirán.

1.4 Trabajos relacionados

La investigación sobre prebúsqueda *hardware* de datos en los últimos años se ha centrado en varias líneas. Por un lado, la identificación de patrones de acceso a memoria complejos, que era un tema central hace unos años, sigue siendo un punto de interés [MKP05]. Aunque utiliza soporte del compilador, merece la pena destacar por su actualidad la contribución de [EMP09], que pone de manifiesto el interés y dificultades de esta línea. Por otro lado, hay trabajos

dedicados a mejorar la cobertura, precisión y puntualidad de prebuscadores agresivos previos basados en patrones específicos. Así, [HuL06] y [SSC03] mejoran los *stream buffers*. [HuL06] adapta el tamaño del *stream buffer* según el comportamiento del programa, para incrementar la precisión y puntualidad. [SSC03] añade otros predictores además del *stride* y el secuencial para incrementar la cobertura del prebuscador. [ZhL03] propone una forma de filtrar las prebúsquedas inútiles según sus efectos previsto en el reemplazo. [SMK+07] persigue un objetivo similar que tiene en cuenta la polución causada por una agresividad excesiva, con un coste de implementación notablemente más bajo. [WBM+03] usa el compilador para proporcionar pistas a un prebuscador *hardware* activado por los fallos de L2 que presentan localidad espacial y se ocupa de patrones de acceso indirecto.

Otra línea está dedicada a los prebuscadores de correlación. [HMK03] presenta el *Tag Correlating Prefetching* (TCP) que analiza y predice el patrón de *tags* asociados a un conjunto de cache dado. [NeS05] reorganiza las tablas de correlación en una tabla de índices y un *Global History Buffer* (GHB), y adapta el GHB para soportar la mayoría de algoritmos de prebúsqueda, reducir el coste del hardware y mejorar la precisión. [GaB06] compara implementaciones *software* y *hardware* en una arquitectura SMT. [SBC05] persigue mejorar la cobertura y la precisión de prebuscadores de correlación filtrando de la secuencia de fallos las ocurrencias que rompen los patrones reales.

Spatial Memory Streaming (SMS) en [SWA+06] identifica y predice patrones espaciales correlacionados por el código en los accesos a bloques dentro de una región de memoria. Constituye un prebuscador agresivo que explota formas irregulares de localidad espacial, pero usando tablas bastante grandes.

Los modelos detallados de la memoria principal se están haciendo cada vez más comunes, con propuestas que persiguen mejorar o dirigir la actividad de prebúsqueda [HuL06, WBM+03]. [WBM+03] coloca un árbitro de acceso en un controlador de memoria Rambus. Las peticiones de prebúsqueda se lanzan solo cuando no hay peticiones por demanda pendientes. [HuL06] integra un prebuscador de *streams* adaptativo en el controlador de memoria de una DRAM DDR2.

Algunos trabajos intentan colocar contenidos en la *cache* de forma inteligente, evitando prebuscar palabras o bloques inútiles. [PuA06] busca sólo las palabras de un bloque que es probable sean demandadas, según la historia de accesos previos al bloque. [CYF+04] puede predecir qué palabras de una región

contigua de memoria van a ser accedidas, de acuerdo con el comportamiento pasado. Se aplica a prebúsqueda de sub-bloques y de grupos de bloques.

Los modelos de consumo basados en simulación a nivel de circuitos no son frecuentes en trabajos de prebúsqueda. Los modelos CACTI dan sólo aproximaciones muy bastas [GMT04]. Guo et. al realizaron un trabajo de caracterización destacable, usando HSpice, que finalmente condujo a la propuesta de PARE [GNM05].

1.5 Contribuciones de la Tesis

Durante la elaboración de esta Tesis se han elaborado las publicaciones científicas que se describen a continuación:

- El estudio de patrones de acceso a memoria y su aplicación a prebúsqueda fue presentada en *IEEE Int. Symp. on Performance Analysis of Systems and Software* - ***ISPASS 2000*** celebrado en Austin en Abril de 2000 [RIV+00].

- La definición de un nuevo prebuscador junto con una comparativa de prestaciones conseguidas por distintos prebuscadores fue presentada en *MEmory performance DEaling with Applications, systems and architecture* - ***ME-DEA*** celebrado en Seattle en Septiembre de 2006 y recogido en *SIGARCH* ***Computer Architecture News*** 35, 4 (Sep. 2007) [RBI+07].

- La siguiente publicación es presentada en *14th International Conference on Parallel and Distributed Computing* - ***Euro-Par*** celebrado en Las Palmas de Gran Canaria en Agosto de 2008 y recogida en ***Lecture Notes in Computer Science*** 5168 [RBI+08]. Incluye la evaluación de varias propuestas para controlar dinámicamente la agresividad del prebuscador *secuencial marcado* y evitar así la aparición de pérdidas.

- Finalmente, se envió un esquema general de prebúsqueda adaptable con varios de nuestros prebuscadores a la competición *1st JILP* ***Data Prefetching Championship*** *(DPC-1)*, celebrado en Raleigh en Febrero de 2009 y que será publicado próximamente en *the Journal of Instruction-Level Parallelism (JILP)*. En la competición se obtuvo el tercer puesto en prestaciones y el *Best Paper Award*.

1.6 Organización de la memoria

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

El Capítulo 2 se dedica al análisis del comportamiento de las instrucciones de acceso a memoria, con vistas a posibles mejoras de la prebúsqueda. Se presentan los patrones ya conocidos y se define uno nuevo: el *linear link*.

El Capítulo 3 presenta la metodología y el entorno de evaluación que se utilizará en los próximos capítulos. Se detallan los modelos de procesador y de jerarquía de memoria utilizados. Se finaliza el capítulo describiendo la carga de trabajo y la plataforma de simulación.

En el Capítulo 4 se comparan las prestaciones de cinco prebuscadores diferentes. Dos de ellos son nuevas propuestas, uno basado en el patrón *linear link*; el otro un nuevo prebuscador de correlación basado en *deltas*.

El Capítulo 5 presenta diversos mecanismos sencillos para adaptar la agresividad del prebuscador *secuencial marcado*, evitando así la aparición de pérdidas en aplicaciones concretas.

En el Capítulo 6 se presenta un esquema general de prebúsqueda que puede ser adaptado a diferentes objetivos de diseño y se evalúa su eficacia en un entorno de simulación externo que permitió su comparación con prebuscadores contribuidos por otros autores.

Por último, en el Capítulo 7 se exponen las conclusiones finales de esta Tesis y las líneas que permanecen abiertas.

En el Apéndice A se muestran algunas pantallas de la aplicación VisualPtrace utilizada como herramienta de depuración y validación del simulador.

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

Análisis de patrones de las instrucciones de acceso a memoria

Este capítulo analiza el comportamiento de las instrucciones de acceso a memoria a fin de buscar oportunidades de mejora en los mecanismos de prebúsqueda. Se presentan los patrones de acceso a memoria y se clasifica cada acceso según el patrón seguido. Además de buscar patrones ya conocidos como last-address *y* stride*, se define también un nuevo patrón* (linear link)*, y se cuantifica su utilidad en prebúsqueda. Utilizando todos estos patrones se puede modelar y, por tanto predecir, entre el 80%-90% (según la aplicación) de las direcciones accedidas por las instrucciones de acceso a memoria.*

2.1 Introducción

El comportamiento regular de los programas ha sido utilizado en el diseño de procesadores para diseñar mecanismos *hardware* críticos que consigan buenas prestaciones, como los predictores de saltos o las memorias *cache*. Si nos centramos en la secuencia de direcciones de las instrucciones de acceso a memoria (*loads* y *stores*) esta regularidad puede ser explotada directamente por prebuscadores de datos o por la ejecución con dirección especulativa de las instrucciones *load*. Sin embargo, la cobertura de los modelos actuales basados en la regularidad de direcciones es baja en algunas áreas de aplicación.

Las aplicaciones que utilizan matrices dispersas y los programas con cálculo simbólico sobre estructuras de datos dinámicas son dos ejemplos en los que se consiguen coberturas bajas, en ocasiones inferiores a un 60%.

En este capítulo se pretende incrementar los niveles de cobertura, especialmente en este tipo de aplicaciones, con vistas a una posible aplicación a la prebúsqueda de datos. Por ello, vamos a definir un nuevo tipo de modelo de regularidad para la secuencia de direcciones: el *linear link*.

En la sección 2.2 se presentan los modelos de regularidad de direcciones ya conocidos. En la sección 2.3 definimos el nuevo modelo de regularidad de direcciones, el *linear link*. Después, la sección 2.4 presenta la amplia carga de trabajo y la metodología que serán utilizadas durante este capítulo. La sección 2.5 presenta los resultados de caracterización de direcciones encontrados. Finalmente, la sección 2.6 expone las conclusiones.

2.2 Modelos de regularidad de la secuencia de direcciones

Según Sazeides y Smith en [SaS96] los modelos de regularidad de direcciones de datos pueden ser clasificados en dos amplios grupos: los modelos de contexto y los modelos computacionales. En general, los modelos de contexto necesitan guardar mucha información y no suelen ser capaces de predecir direcciones que aún no han sido visitadas. El trabajo de [JoG99] es un ejemplo de estos modelos.

Por otro lado, los modelos computacionales tratan de extraer un conjunto de parámetros de la secuencia de direcciones y usarlo para definir recurrencias aritméticas simples. Si en la recurrencia sólo aparecen direcciones, la llamamos recurrencia de dirección. Si, en cambio, aparecen direcciones y valores, es una recurrencia de valor-dirección. Un buen ejemplo del uso de recurrencias de dirección para prebúsqueda es [BaC91]. La recurrencias de valor y dirección también han sido utilizadas para prebúsqueda en varios trabajos, como por ejemplo [MeH96] y [RMS98]. Nuestro modelo de regularidad, el *linear link*, es una generalización de estos trabajos.

En primer lugar presentamos la recurrencia *stride*, en la que se basa el prebuscador del mismo nombre que se introdujo en la sección 1.3.3. Usando únicamente esta recurrencia se puede modelar un gran porcentaje de direcciones en un amplio conjunto de aplicaciones, por ejemplo las relativas a recorridos

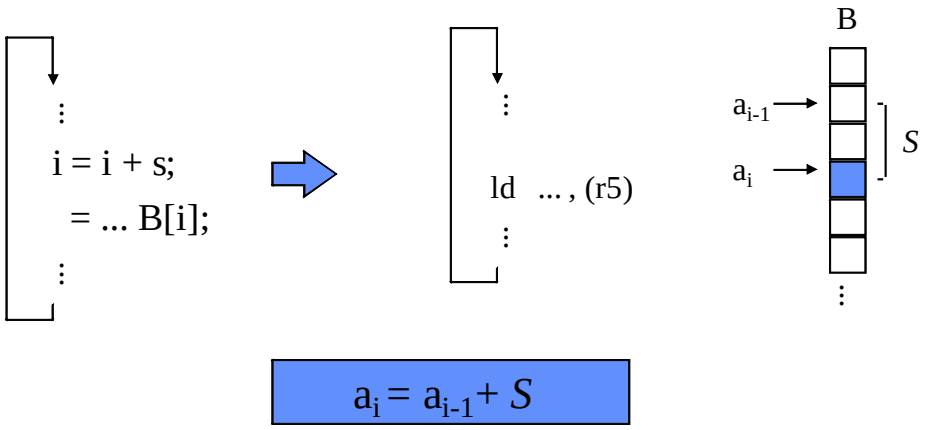

Figura 2.1 Ejemplo de recurrencia *stride*.

lineales a través de vectores. En la Figura 2.1 se muestra un bucle que accede a los elementos del vector con un *stride* de *s* elementos (o *S* bytes). La instrucción *load* que realiza la lectura calcula la dirección a_i en la iteración *i*, que puede ser calculada sumando *S* a a_{i-1}. Se puede apreciar cómo para calcular la siguiente dirección sólo se necesita la última dirección. Cuando S es 0 ocurre un caso particular, que también es llamado recurrencia *last-address*.

2.3 Un nuevo modelo de regularidad: el *linear link*

Cuando los datos de la aplicación están dispuestos en listas, árboles, grafos o matrices dispersas puede ser necesario disponer de los datos leídos por otros *loads* para poder calcular las direcciones. La Figura 2.2 muestra un ejemplo de patrón de acceso *linear link*. En el código de alto nivel (Figura 2.2 izquierda) podemos ver cómo un vector D es accedido a través de los índices guardados en un vector auxiliar I. Observando el código objeto (Figura 2.2 centro), se ve un primer *load* que lee un valor v_i del vector I. Posteriormente un segundo *load* realiza el acceso a D. La dirección de este segundo *load* es calculada por las dos instrucciones aritméticas a partir del valor leído por el primero. La primera instrucción aritmética multiplica v_i por el tamaño del elemento (α) y la segunda suma la dirección base de D (β). Este comportamiento puede ser modelado utilizando la recurrencia expresada en la Figura 2.2, relacionando de forma lineal el valor leído por el primer *load* con la dirección del segundo *load*.

Por lo tanto, el *linear link* relaciona el valor de un *load* (productor), con la dirección de un segundo *load* (consumidor) a través de la recurrencia expresada

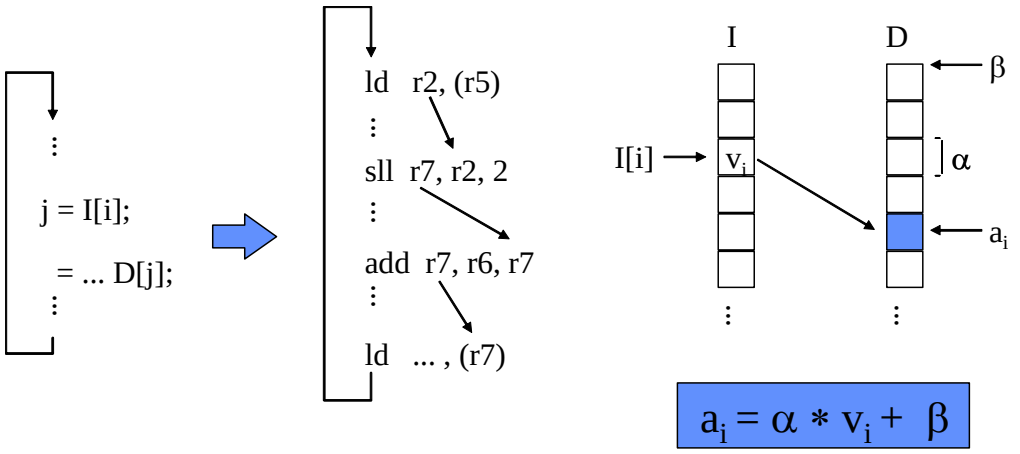

Figura 2.2 Ejemplo de recurrencia *linear link*.

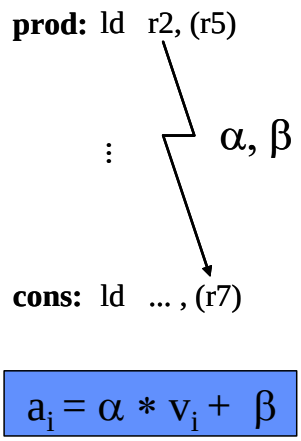

Figura 2.3 Representación de un *linear link*.

y de los parámetros α y β. El valor de estos parámetros varía en función del caso concreto. Por ejemplo, cuando se accede a una lista encadenada de elementos, α es 1 y β es el desplazamiento del puntero dentro del elemento. Otro caso corriente es el del ejemplo que hemos visto en Figura 2.2, el acceso a un vector a través de otro vector de índices. En este caso α es el tamaño del elemento del vector y β la dirección base del vector.

Existen dos trabajos previos que estudian casos particulares de este modelo y los aplican a prebúsqueda de datos. Por un lado [MeH96] define recurrencias valor-dirección, pero las aplica sólo cuando productor y consumidor son el mismo *load*. Por otro lado [RMS98] definen también este tipo de recurrencias, pero sólo cuando α=1 y con algunas restricciones en la forma que la que el

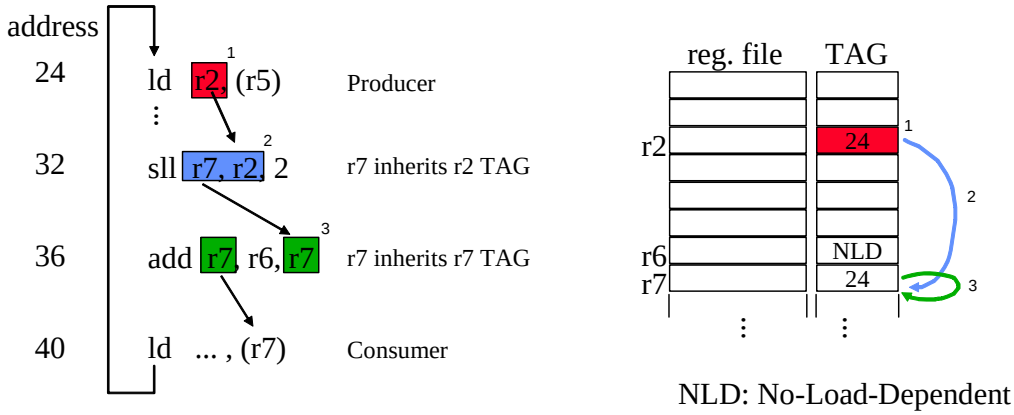

Figura 2.4 Ejemplo de funcionamiento del algoritmo de detección de recurrencias *linear link*.

cálculo se realiza entre productor y consumidor. Por lo tanto, nuestro *linear link* puede ser considerado como una extensión de las recurrencias valor-dirección modeladas en estos dos trabajos previos.

La idea principal del algoritmo utilizado para detectar las recurrencias *linear link* es el seguimiento de las dependencias verdaderas etiquetando los registros de arquitectura. Para ello, cada vez que se ejecuta un *load*, su registro destino es etiquetado con el PC del *load*. Además, siempre que se ejecuta una instrucción aritmética, su registro destino hereda la etiqueta de uno de los registros fuente. De esta forma, cuando un *load* consumidor se ejecuta, su registro fuente estará etiquetado con el PC del *load* productor, pudiendo establecerse la relación entre ambos.

La Figura 2.4 presenta un ejemplo de funcionamiento del algoritmo. Al ejecutarse el *load* productor, su registro destino (r2) es etiquetado con el PC (1). Después, la ejecución de la instrucción aritmética *sll* hace que r7 herede la etiqueta de su único registro fuente r2 (2). La siguiente instrucción aritmética (*add*) debe asignar a su registro destino (r7) la etiqueta de uno de sus registros fuente (r6 o r7). En este caso se elige la etiqueta asignada más recientemente, es decir, la de r7 (3). Finalmente, al ejecutarse el *load* consumidor puede disponer del PC del *load* productor consultando la etiqueta de su registro fuente.

Una vez que se ha detectado un posible *linear link* se deben calcular sus parámetros α y β. En este trabajo sólo hemos considerado α= 1, 2, 4 y 8. Para cada valor de α, se calcula su correspondiente β utilizando la recurrencia. y se almacena en una tabla (*Link Table*) (Figura 2.5). Es importante subrayar que un

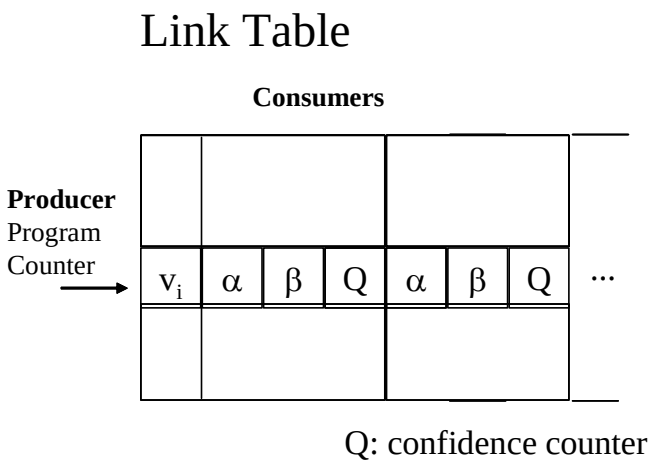

Figura 2.5 Almacenamiento de *linear links* en la *Link Table*.

mismo productor puede tener varios consumidores y un consumidor varios productores. Cada vez que se ejecuta el productor se actualiza v_i. Cada vez que se ejecuta el consumidor se comprueban los parámetros α y β y se actualiza el contador de confianza de la relación.

2.4 Carga de trabajo y metodología

Al ser el objetivo de este capítulo la caracterización de aplicaciones, se ha utilizado una amplia carga de trabajo. Hemos utilizado cinco colecciones de programas de prueba diferentes: Spec95, Nas [BBB+91], Perfect [CKP+90], Olden [RCR+95] e IAbench [RIV+00]. La colección IAbench (*Indirect Access benchmarks*) ha sido recopilada por nosotros y consta de 10 programas (3 de enteros y 7 de coma flotante). Se trata de programas de cálculo que utilizan vectores índice para acceder a matrices dispersas o para realizar cálculos simbólicos.

En este capítulo se ha utilizado SPARC V7 para SunOS, compilando con el mayor nivel de optimización. Las direcciones y valores de los *load* y los identificadores de los registros han sido recogidos con la herramienta *Shade* [CmK94]. La mayoría de los programas de prueba fueron ejecutados completos, excepto los más largos, de los que se ejecutaron 4.000 millones de instrucciones

después de un periodo de inicialización. Se pueden ver más detalles sobre la carga de trabajo de este capítulo en la Tabla 2.1.

Tabla 2.1 Benchmarks usados: (cs) = número de instrucciones saltadas (millones), (#inst) = números de instrucciones simuladas (millones), (%si) = porcentaje de cs + #inst sobre número total de instrucciones del programa, (%ld) = porcentajes de instrucciones *load*.

IAbench (C and Fortran)			
prog	**cs/#inst**	**%si**	**%ld**
bil	0/526	100	24
hop	0/842	100	33
lps	0/1362	100	26
nis	0/684	100	38
onm	0/2090	100	25
smv	0/2733	100	19
che	0/1535	100	16
s13	0/589	100	35
spi	4500/4000	40	23
sqr	0/1031	100	22
Aver		94	26

Olden (C)			
prog	**cs/#inst**	**%si**	**%ld**
bh	0/1432	100	27
em3	0/83	93	25
per	0/1199	36	11
pow	0/896	100	19
tre	0/166	100	15
bis	0/330	14	15
hea	0/165	86	31
mst	0/180	100	15
tsp	0/353	99	19
vrn	0/241	3	26
Aver		73	20

Spec95int (C)			
prog	**cs/#inst**	**%si**	**%ld**
com	0/4000	8	13
ijp	5000/4000	22	13
m88	10000/4000	18	18
cc2	0/545	100	17
go	10000/4000	46	19
li2	0/4000	7	21
prl	500/4000	16	20
vor	4000/4000	11	19
Aver		29	17

Spec95fp (Fortran)			
prog	**cs/#inst**	**%si**	**%ld**
app	4000/4000	13	24
aps	3000/4000	15	24
fp2	3000/4000	2	23
hyd	1000/4000	9	24
mgr	3000/4000	130	35
su2	6000/4000	20	27
swi	500/4000	11	32
tom	5000/4000	18	27
tur	1000/4000	2	9
wav	1000/4000	12	26
Aver		23	25

Tabla 2.1 Benchmarks usados: (cs) = número de instrucciones saltadas (millones), (#inst) = números de instrucciones simuladas (millones), (%si) = porcentaje de cs + #inst sobre número total de instrucciones del programa, (%ld) = porcentajes de instrucciones *load*.

Nas (Fortran)			
prog	**cs/#inst**	**%si**	**%ld**
abt	0/1052	100	44
asp	0/843	100	33
buk	67/13	100	24
cgm	0/512	100	37
emb	0/4000	44	28
fft	0/1587	100	24
Aver		91	31

Perfect (Fortran)			
prog	**cs/#inst**	**%si**	**%ld**
lg	0/1165	100	17
lw	0/4000	22	31
mt	0/645	100	26
na	0/3340	100	32
oc	0/4000	48	24
sd	0/1379	100	23
sm	1800/4000	14	19
sr	0/4000	50	40
tf	0/1746	100	31
ti	0/2458	100	38
ws	1500/4000	84	23
Aver		74	28

2.5 Resultados

En esta sección vamos a mostrar los patrones de regularidad encontrados en la secuencia de direcciones de las instrucciones *load*. Todos los resultados mostrados aquí son medios por colección de programas. En el apéndice B se incluyen todos los resultados por aplicación.

Comenzamos en primer lugar con el patrón *stride* (y *last address* como caso particular). La Figura 2.6 muestra, para cada colección de programas de prueba, el porcentaje medio de *loads* dinámicos (respecto al total de instrucciones ejecutadas) que siguen un patrón *stride* o *last address*. Las tres colecciones de programas de prueba de la derecha (Spec-fp, Nas and Perfect) realizan mayoritariamente cálculos numéricos sobre matrices densas. Las otras tres de la izquierda (IAbench, Olden and Spec-int) trabajan con estructuras indexadas, punteros y, en general, estructuras de datos más complejas. Para cada colección se indican la regularidades *stride* (str) y *last address* o *stride* cero (sc) encontradas. En los programas de cálculo numérico, la regularidad *stride* es dominante: 80% ó más. En las tres colecciones de la izquierda, la regularidad *stride* es menor, tal y como se esperaba, pero en ningún caso despreciable.

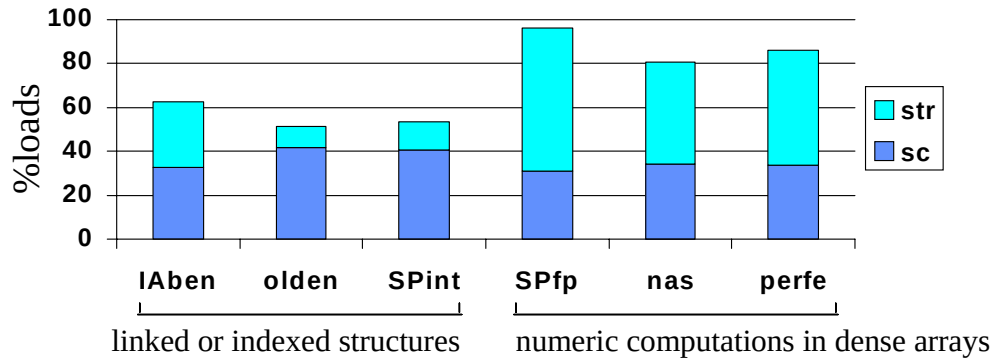

Figura 2.6 Regularidad *stride* (str) y *last-address* (sc) encontrada en los programas de prueba.

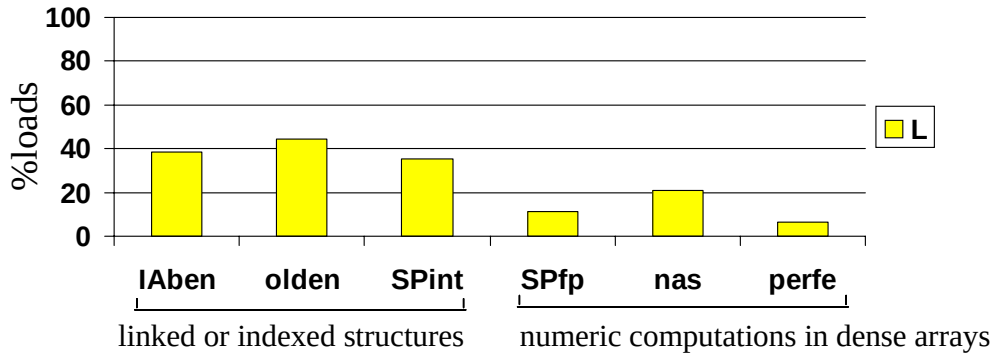

Figura 2.7 Regularidad *linear link* encontrada en los programas de prueba.

Además, es importante destacar que más de un tercio de las referencias siguen un patrón *last address* en todas las colecciones.

Continuamos ahora mostrando la regularidad *linear link* encontrada en los programas de prueba. Según se aprecia en la Figura 2.7, las tres colecciones de la izquierda muestran mayor patrón *linear link* (entre un 33% de Spec-int hasta 43% de Olden) que las tres de la derecha. A pesar de todo, el patrón encontrado en las tres colecciones de cálculo numérico tampoco parece que sea despreciable.

Si se buscan ambas regularidades simultaneamente (Figura 2.8), podemos ver que la regularidad media reconocida es mayor de 80% en todas las colecciones.

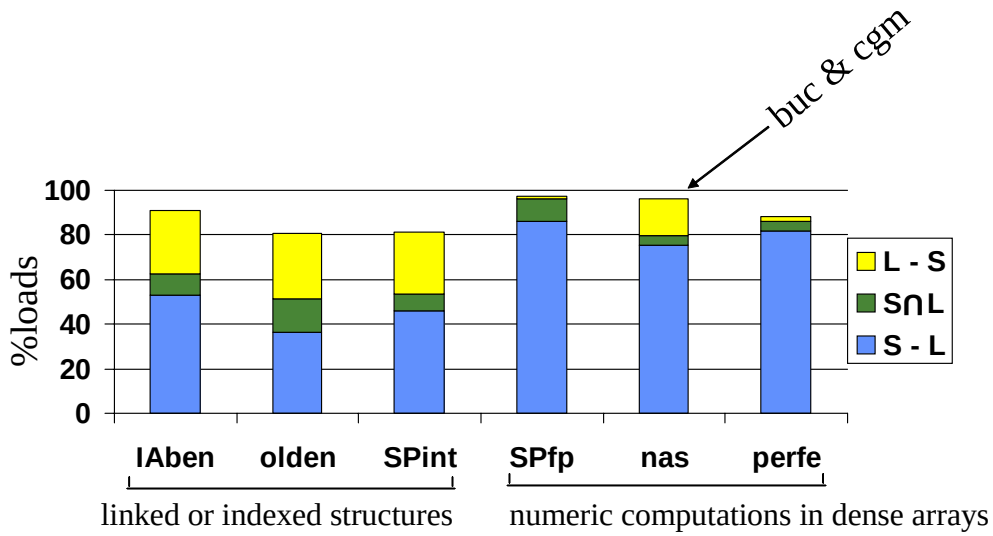

Figura 2.8 Regularidades *stride* y *linear link* encontrada en los programas de prueba.

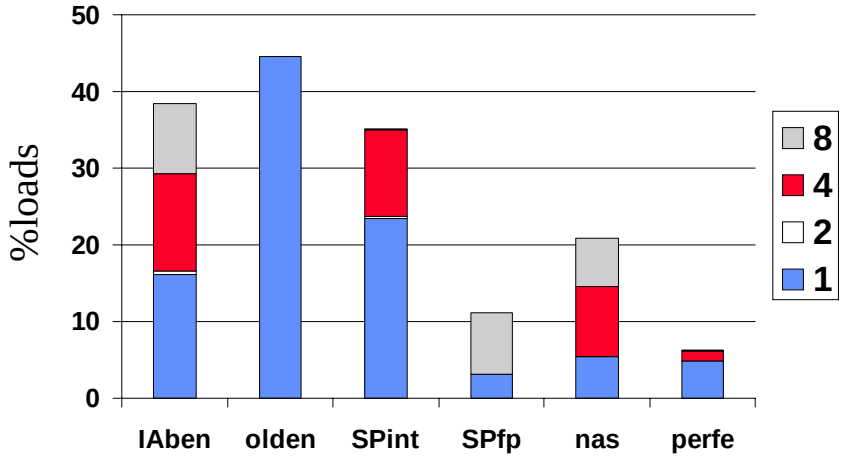

Figura 2.9 Descomposición del patrón *linear link* según el valor del parámetro α.

Además, se produce un solapamiento entre ambas regularidades, debido a que hay *loads* que siguen ambos patrones simultaneamente. Este solapamiento es casi completo en las tres colecciones de la derecha (excepto dos programas de Nas: buk y cgm). En las tres colecciones de la izquierda la mayoría de la contribución del patrón *linear link* no está solapada.

La Figura 2.9 descompone el patrón *linear link* según el valor de su parámetro α. Mientras que la colección Olden usa siempre α=1 (punteros), en el resto de

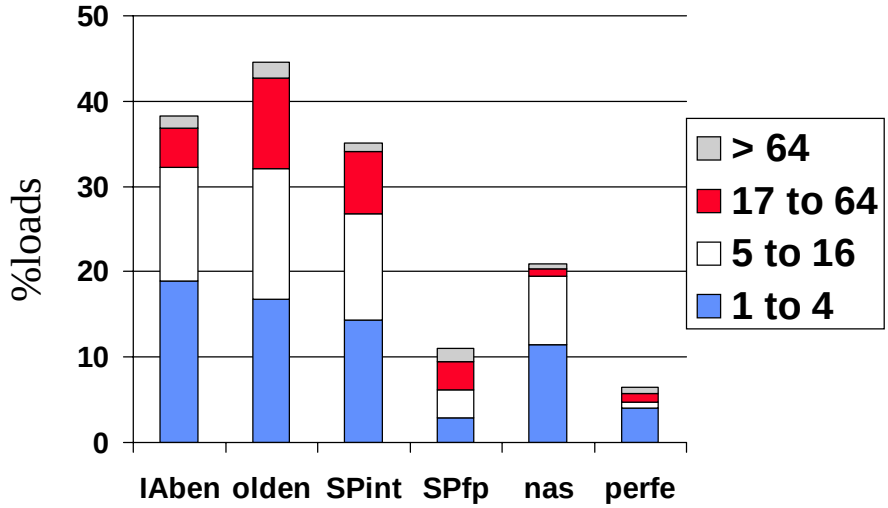

Figura 2.10 Descomposición del patrón *linear link* según la distancia entre *loads*.

colecciones los valores de α 4 y 8 son los más comunes, representando un 58% del patrón *linear link* en IAbench y un 33% en Spec-int.

Estimación de la utilidad de las regularidades para prebúsqueda

Los prebuscadores *hardware* de datos intentan ocultar la latencia de fallo de una instrucción de acceso a memoria solapándola con la ejecución de las instrucciones previas. Para tener una estimación de la utilidad de la regularidad *linear link* para prebúsqueda se pueden presentar varias métricas: el porcentaje de fallo de los *loads* consumidores, y el número de instrucciones que pueden ser solapadas (distancia entre *load* productor y consumidor). Un mecanismo de prebúsqueda basado en el patrón *linear link* podría conseguir prebuscar con éxito la mayoría de los fallos producidos por los loads consumidores: 57% de todos los fallos en IAbench, 88% en Olden y 40% en Spec-int.

La Figura 2.10 muestra una descomposición de la regularidad *linear link* según la otra métrica indicativa, la distancia entre loads (*link distance*). Se puede apreciar cómo entre un 37% a 49% de los *linear link* tienen una distancia menor de 5 instrucciones. Y sólo un 3% de los *linear link* tienen distancias mayores de 64 instrucciones. A pesar de las pequeñas distancias observadas Roth *et al*. en [RMS98] obtuvo mejoras de prestaciones considerables en la colección Olden utilizando un prebuscador basado en una forma restringida de *linear link* en un procesador fuera de orden 4-superescalar.

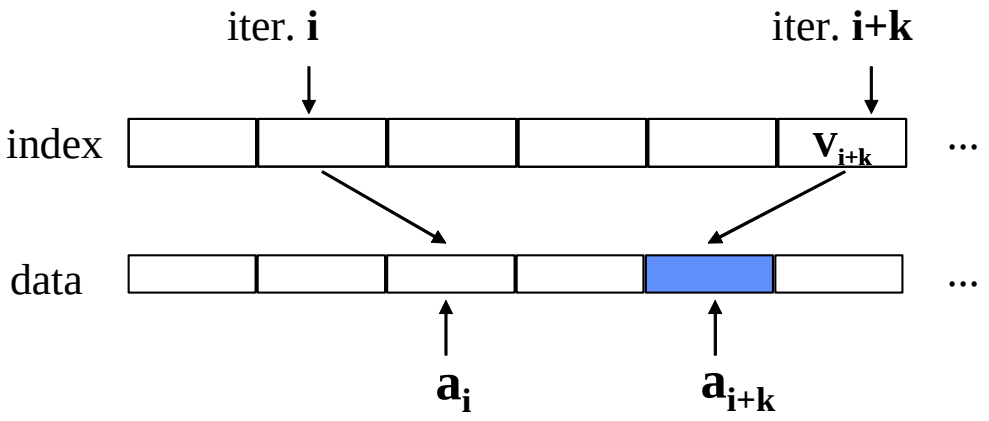

Figura 2.11 Ejemplo de predicción de la dirección del consumidor K instancias adelante.

2.5.1 Comportamiento de los loads productores

El patrón *linear link* puede ser utilizado para prebúsqueda de forma que la ejecución del productor prebusca la dirección que probablemente lanzará el consumidor. Sin embargo, acabamos de ver que la distancia entre productor y consumidor es habitualmente pequeña. Esto puede provocar que la prebúsqueda actúe tarde y que se oculten pocos ciclos de la latencia del *load* consumidor.

Por este motivo, pretendemos ahora prebuscar un *load* consumidor con una antelación de *K* instancias. Para ello vamos a prestar atención a la regularidad del *load* productor. La Figura 2.11 muestra un ejemplo. Supongamos que el *load* productor está accediendo a los elementos de un vector de índices de forma *stride*, y que el consumidor accede a otro vector usando el índice leído. Estando en la iteración *i* podemos predecir la dirección del load consumidor de la instancia *i*+*K* (a_i+*K*) siguiendo los siguientes pasos: 1) predecimos la dirección del productor *K* instancias adelante; 2) realizamos el acceso a memoria a dicha dirección; y 3) a partir del valor leído calculamos la dirección del consumidor usando la recurrencia.

Por lo tanto, vamos a intentar analizar los casos en los que la regularidad del productor nos permitiría aumentar la distancia de predicción.

- Si el valor leído por el productor es regular (*last value* o *stride value*), entonces el consumidor tendrá la misma regularidad en su dirección (*last*

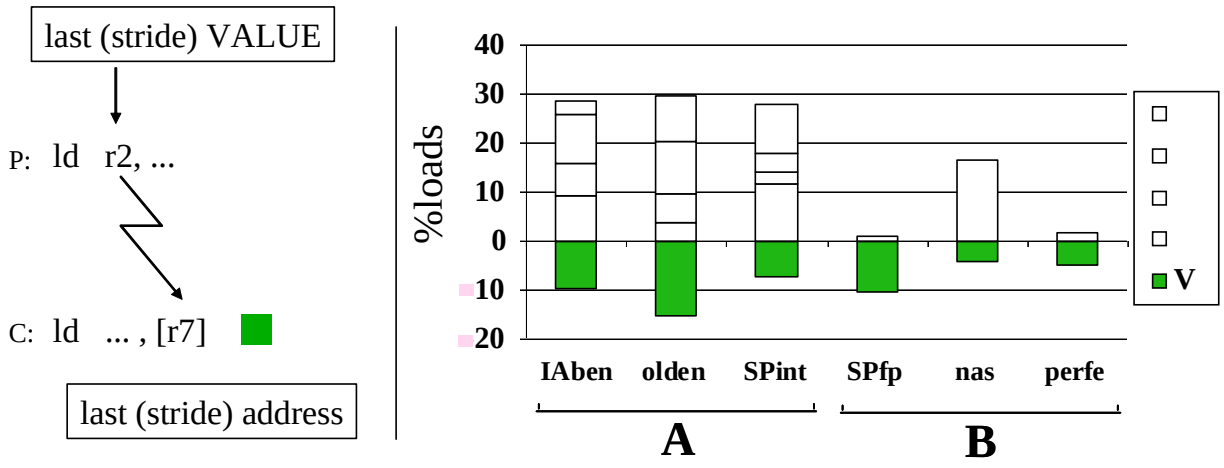

Figura 2.12 Porcentajes de loads consumidores cuyos productores presentan regularidad en su valor (V).

address o *stride address*), ya que el *linear link* es una función lineal, y su dirección de la iteración *i+K* podrá ser predicha directamente usando una recurrencia *stride*. Este es un caso que se produce habitualmente al acceder a variables globales a través de punteros. En las tres colecciones de programas de la izquierda es un caso frecuente, que se produce entre el 21% y 34% de la regularidad *linear link* (Figura 2.12).

– Cuando el productor no tiene regularidad en su valor, pero su dirección tiene regularidad *stride* o *last address*, se puede predecir la dirección del productor de la instancia *i+K* usando una recurrencia *stride*. Tal como hemos visto en la Figura 2.11, sólo se necesita un acceso a memoria para, aplicando después la recurrencia, conocer la dirección del consumidor de la instancia *i+K*. Este caso se produce cuando se accede a vectores que contienen índices a otros. En las tres colecciones de la izquierda de esta gráfica se produce este caso, en un 9% a 33% de la regularidad *linear link*, siendo especialmente importante en IAbench y Spec-int (Figura 2.13). Nótese también que en Spec-fp Nas y Perfect todos los *linear link* tienen algún tipo de regularidad en su productor.

– Si ni el valor del productor ni su dirección tienen regularidad, es posible que el productor tenga también una relación *linear link* consigo mismo (le llamamos *self-link load*). En este caso, para calcular la dirección de la iteración *i+K* se necesita un secuencia de accesos a memoria. Este es el caso habitual que aparece al acceder a una lista encadenada por punteros.

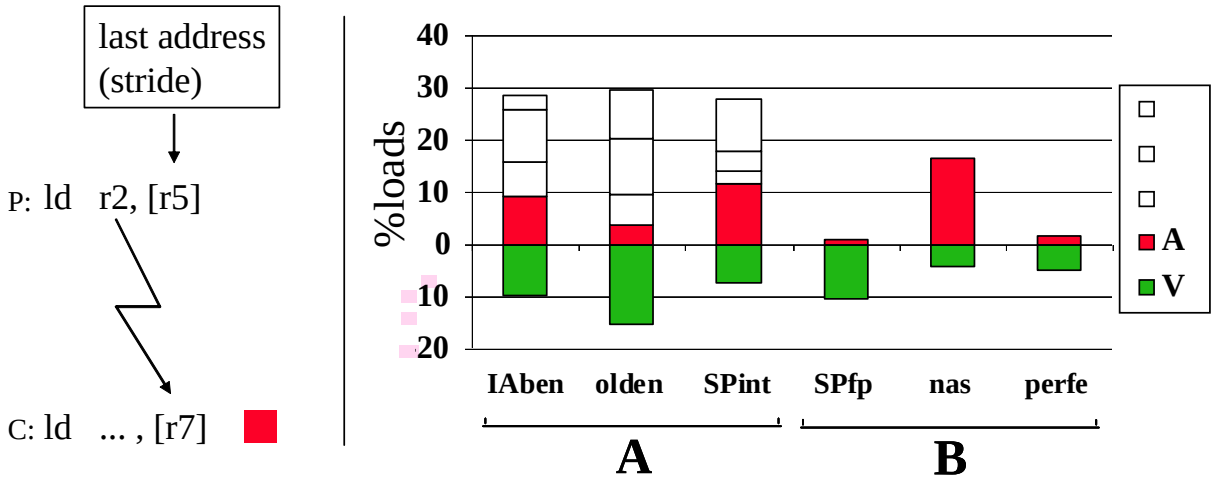

Figura 2.13 Porcentajes de loads consumidores cuyos productores presentan regularidad en su dirección (A).

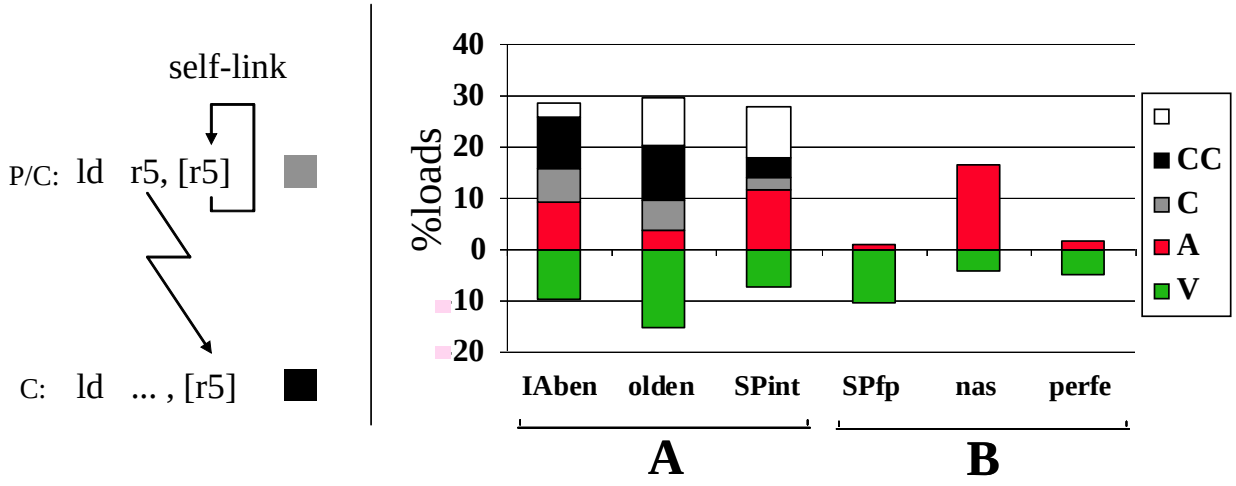

Figura 2.14 Porcentajes de loads con regularidad *self-link* (C) y sus consumidores (CC).

En las colecciones marcadas como **A** en la Figura 2.14 los productores *self-link* aparecen en el 18% a 42% de los casos de *linear link*.

– Finalmente, del resto de *loads* con regularidad *linear link* no hemos podido identificar ningún tipo de regularidad en su productor, por lo que no es posible predecir la dirección del *load* consumidor de la iteración *i+K*. En las colecciones **A** los *linear link* con regularidad del productor desconocida varía entre 8% y 28% (ver Figura 2.15).

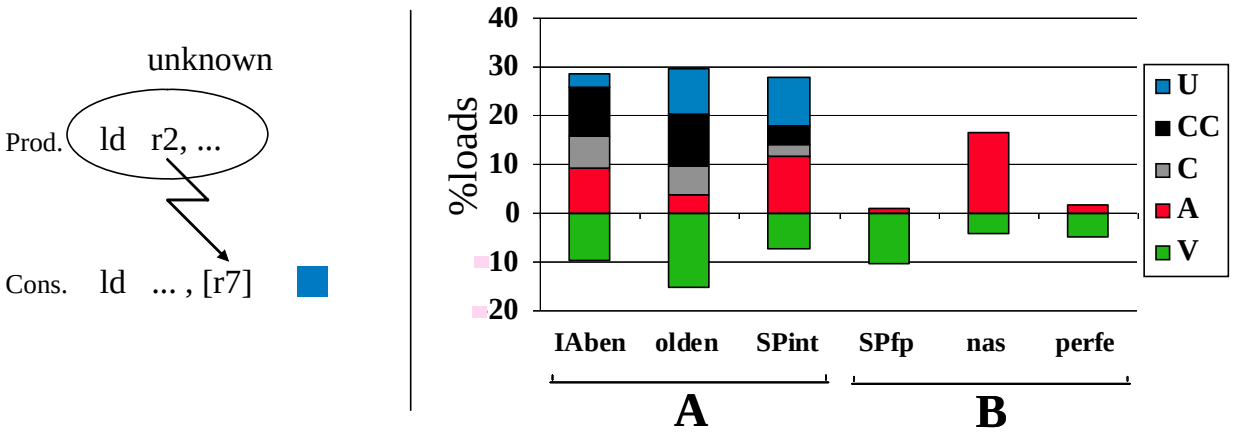

Figura 2.15 Porcentajes de loads consumidores cuyos productores presentan una regularidad desconocida (U).

2.6 Conclusiones

En este capítulo se ha definido un nuevo modelo de regularidad (el *linear link*) que relaciona la dirección de un *load* con los valores leídos por otros *loads* a través de funciones lineales.

Este nuevo modelo representa aproximadamente un tercio de todas las referencias de memoria. La regularidad media observada utilizando todos los modelos simultaneamente es siempre superior al 80%.

Para analizar la presencia de este nuevo modelo de regularidad hemos usado una amplia carga de trabajo. Además hemos buscado programas de prueba que utilicen vectores índice para acceder a matrices dispersas o para realizar cálculos simbólicos y los hemos reunido en una nueva colección (IAbench).

Como resultado del análisis se han obtenido métricas que muestran que el nuevo modelo de regularidad podría ser utilizado para prebúsqueda *hardware* de datos. En los próximos capítulos se define un modelo detallado de procesador y memoria y se experimenta con sistemas de prebúsquedas basados en diversos modelos de regularidad.

CAPÍTULO 3

Metodología y Entorno de Evaluación

En este capítulo se presentan los elementos comunes de la metodología y entorno de evaluación utilizados en los próximos capítulos: un procesador fuera de orden superescalar agresivo conuna jerarquía de memoria de altas prestaciones, modelado a nivel de ciclo utilizando una adaptación de SimpleScalar 3.0 y programas de prueba seleccionados Spec CPU 2K, Olden e IABench.

3.1 Introducción

Todas las propuestas de prebúsqueda que se presentan en este documento han sido analizadas mediante la simulación detallada de la ejecución de un conjunto de programas de prueba sobre un determinado modelo de procesador.

En el área de Arquitectura y Tecnología de Computadores la simulación es método habitual para la investigación. Un simulador de un procesador permite modelar su actividad interna a nivel de ciclo. Además, es posible centrarse en los aspectos realmente importantes aumentando el nivel de detalle de su simulación, simplificando otros aspectos no tan importantes. El simulador imita el comportamiento de las instrucciones en el procesador modelado (etapas del segmentado, ocupación de recursos, etc.). A partir de la simulación se puede evaluar el rendimiento de un procesador actual ante algunas modificaciones, por ejemplo la inclusión de un sistema de prebúsqueda, o incluso inferir el comportamiento de procesadores futuros. Sin embargo, la utilización de

simuladores tiene también algunas desventajas, como la influencia de los errores de programación y la dificultad de validar los resultados.

En este capítulo se describe en detalle el modelo de procesador utilizado. En la sección 3.2 se hace una descripción general del mismo y se describen los modelos de desambiguación de memoria y de predicción de latencia. En la sección 3.3 se presenta detalladamente la jerarquía de memoria utilizada y el camino de datos de las instrucciones de acceso a memoria.

En la sección 3.4 se presenta el simulador junto con una breve descripción de las modificaciones efectuadas. También se describe el método de validación de los resultados. La sección 3.5 detalla los programas utilizados como *benchmarks* (o programas de prueba). Por último, la sección 3.6 presenta los recursos *hardware* y *software* empleados durante el desarrollo de esta Tesis.

3.2 Procesador base

En esta sección se describen las características principales del procesador modelado.

3.2.1 Descripción general del procesador

Modelamos un procesador *fuera de orden* superescalar y supersegmentado. La organización en bloques es la que se muestra en la Figura 3.1. El procesador se

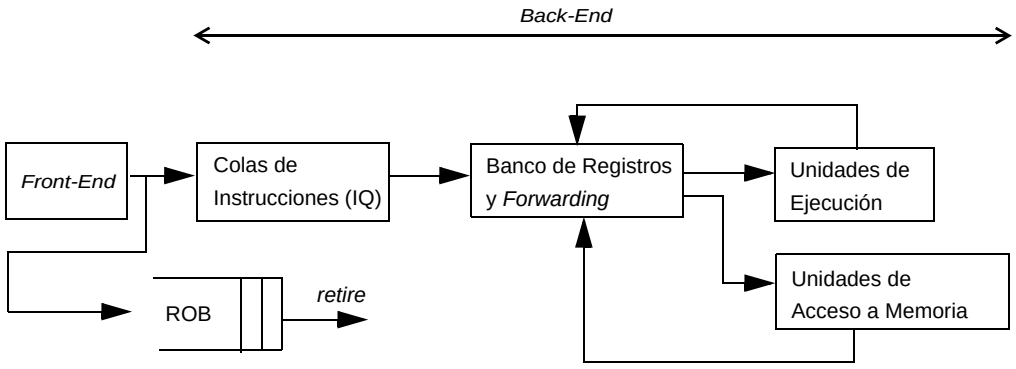

Figura 3.1 Organización general del procesador.

compone principalmente de un frontal (*Front-End*) en orden y un cuerpo (*Back-End*) fuera de orden. Se modela un frontal de ocho etapas: *búsqueda (fetch), acceso a cache de instrucciones, selección, alineación, decodificación (decode), renombre (rename)* y *emisión (issue)* a las colas de instrucciones (IQ),

donde esperarán a ser ejecutadas por el *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 por el *Front-End*, esperan en la IQ al inicio de la ejecución (etapa *dispatch*). La IQ contiene una matriz de dependencias y una lógica de selección. La matriz de dependencias permite determinar qué instrucciones están listas, por tener disponibles todos sus registros fuente. La lógica de selección inicia la ejecución (*dispatch*) de las instrucciones más viejas entre las ya listas.

Tras el inicio de ejecución (*dispatch*) las instrucciones acceden al banco de registros o a la red de cortocircuitos (etapa *reg.read*) para leer sus operandos fuentes y comienzan su ejecución en las unidades funcionales apropiadas (etapa *execute*). Una vez ejecutadas, las instrucciones escriben los resultados en el banco de registros de renombre (etapa *write-back*). Finalmente, las instrucciones que han completado su ejecución consolidan su estado en orden de programa (etapa *commit*) y son retiradas del ROB (etapa *retire*).

La Figura 3.2 resume la segmentación en etapas por las que pasan todas las instrucciones.

3.2.2 Parámetros del procesador

El procesador modelado es bastante agresivo, ya que tiene un grado de escalaridad ocho (*8-inicio*). Es capaz de buscar ocho instrucciones por ciclo

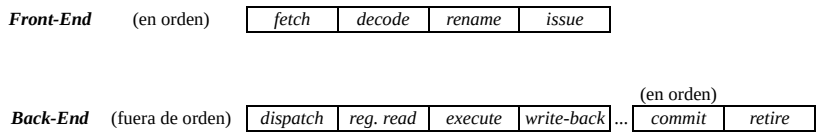

Figura 3.2 Segmentación en etapas de las instrucciones. Se asume 1 ciclo de latencia de ejecución.

utilizando dos direcciones (PC) distintas; decodificar, renombrar y emitir ocho instrucciones por ciclo. Permite iniciar la ejecución (*dispatch*) fuera de orden de hasta 4 instrucciones de coma flotante más 8 instrucciones de enteros por ciclo, incluidos un máximo de 4 accesos a memoria (compartiendo con enteros los puertos de inicio desde la IQ). El resto de parámetros y latencia de las unidades funcionales se muestra en la Tabla 3.1.

Tabla 3.1 Parámetros principales del procesador simulado.

Parámetros	
Ancho de búsqueda, decodificación y emisión	8 instrucciones / ciclo
Ancho de inicio de ejecución desde IQ	8 enteros + 4 c. f.
Ancho de *retire*	16 instrucciones / ciclo
Predictor de saltos: híbrido	(*bimodal+gshare*) 16 bits
Entradas en el Reorder Buffer	256
Nº de *Loads* y *Stores* en vuelo	128
Entradas en IQ enteros / IQ coma flotante	64 / 32
Unidades funcionales enteros (ALU/MUL)	8 / 2
Unidades funcionales c. f.	4 / 4

en ciclos	**Latencia**	**Latencia 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

Puede haber un máximo de 256 instrucciones en vuelo entre las etapas de emisión y consolidación. El número máximo de instrucciones de acceso a memoria en vuelo, *loads* y *stores*, es de 128. 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.

3.2.3 Desambiguación

Se va a utilizar un modelo de desambiguación de memoria (dependencias de datos a través de memoria) basado en el adoptado por el procesador Digital Alpha 21264 [KMW98] [Kes99]. 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*). Las entradas en las colas se asignan en orden al emitir, se rellenan en ejecución, y finalmente se liberan al consolidar. El modelo de desambiguación utilizado es *perfect store sets*, es decir, se utilizará una predicción de dependencias *Store-Load* perfecta [ChE98]. Así, un *load* quedará bloqueado únicamente cuando haya un *store* previo a su misma dirección, haya sido ya calculada o no. Si no utilizáramos esta técnica un *load* con una dependencia falsa sería bloqueado haciendo parecer mejor el prebuscador.

Los *stores* se inician en orden en la etapa *commit*, cuando los registros necesarios para calcular la dirección y el dato están disponibles.

3.2.4 Predicción de latencia

Las instrucciones *load* tienen latencia variable dependiendo del nivel de la jerarquía de cache en la que se encuentre el dato accedido. Como un gran porcentaje de estas instrucciones acierta en cache, se puede despertar especulativamente a las instrucciones dependientes del *load*, suponiendo latencia de acierto en *cache*, y consiguiendo así una ejecución más rápida de las instrucciones dependientes.

Por tanto, 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 cuando las instrucciones dependientes directamente del *load* pueden ser iniciadas, y termina cuando se conoce el resultado del *look-up* en *cache*.

La Figura 3.3 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. La IQ asume latencia de acierto en *cache* para el *load* y se despierta especulativamente a las instrucciones dependientes. Durante el cuarto y quinto ciclo se pueden iniciar de forma especulativa instrucciones dependientes del *load*.

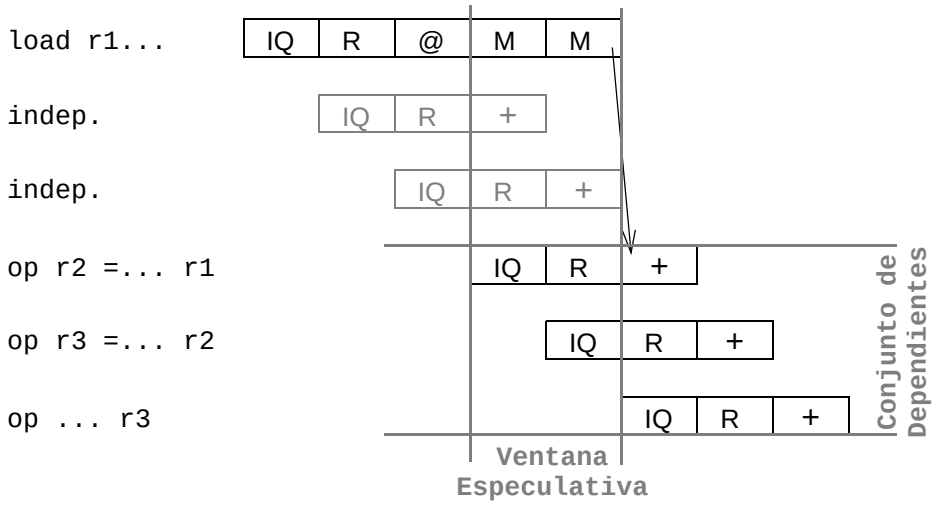

Figura 3.3 Predicción de latencia, diagrama de tiempos asumiendo 1 ciclo entre IQ y EX. Por claridad mostramos el inicio de una única instrucción por ciclo.

En el procesador que hemos modelado se ha realizado una simplificación del mecanismo, de forma que en caso de fallo de predicción se lanzan sólo instrucciones del *conjunto de dependientes* durante el primer ciclo de la *ventana especulativa*. Se ha comprobado que el error cometido es despreciable con respecto a la influencia de los sistemas de prebúsqueda en las prestaciones del procesador.

3.3 Jerarquía de Memoria

En esta sección se procede a hacer una descripción detallada de la jerarquía de de memoria de datos utilizada. Se trata de una jerarquía *on-chip*, de alta capacidad y de gran ancho de banda, similar a la utilizada por Itanium 2 de Intel [NaH02]. Dispone de tres niveles de *cache*, todos ellos dentro del *chip*, y cada uno con un objetivo diferente. El primer nivel tiene por objetivo obtener la mínima latencia en el acceso a un conjunto de datos muy reducido. El segundo nivel busca maximizar el número accesos a memoria por ciclo, a una latencia razonable. El tercer nivel, por su parte, pretende almacenar la máxima cantidad de datos para evitar tener que salir fuera del *chip*.

La Figura 3.4 muestra los principales componentes de la jerarquía de memoria

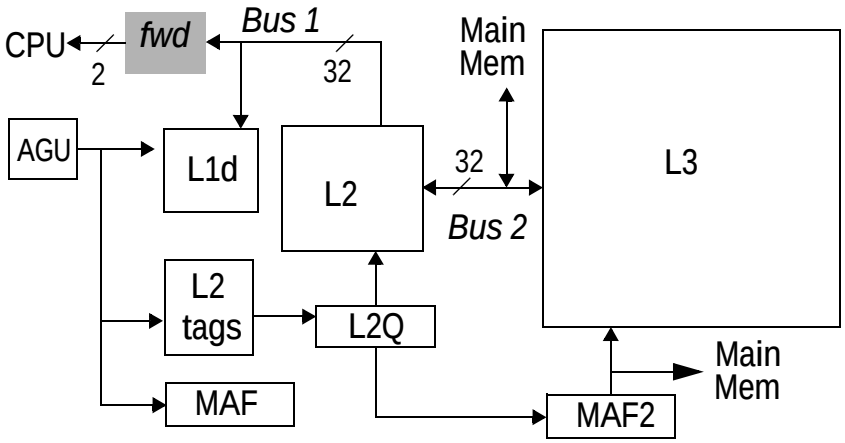

Figura 3.4 Componentes principales de la jerarquía de memoria de datos. AGU: *Address Generation Unit*. MAF: *Miss Address File*. fwd: *forwarding crossbar*.

de datos. El sistema de memoria soporta el lanzamiento de hasta cuatro *loads* por ciclo, o un *store* y hasta dos *loads*, o dos *stores*. En cada ciclo, las instrucciones lanzadas calculan su dirección (**AGU**) y acceden en paralelo al primer nivel de cache de datos (**L1d**), a los *tags* de L2 (**L2 tags**) y al *Miss Address File* (**MAF**), donde se almacenan las direcciones de los bloques que están pendientes en **L1d**. Para soportar los 4 accesos simultáneos, el contenido de la *cache* de datos de primer nivel (**L1d**) está replicado en 4 bancos e incluye un *store buffer* replicado (x2).

Los *loads* que aciertan en **L1d** son servidos a través de **Bus 1** y la red de cortocircuitos (***fwd***). Los que aciertan en **MAF** simplemente esperan al rellenado del bloque. El resto de loads son enviados a la **L2Q** (*Level 2 Queue)* después de acceder a los *tags* de L2 (**L2 tags**). La **L2Q** puede enviar en 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 accesos que han fallado en **L2 tags** son enviados a **MAF2**. La **MAF2** es la encargada del acceso a L3 y a memoria principal. La recarga (*refill*) desde L3 a L2 se hace a través de **Bus 2** y se ocupan 8 bancos de L2. La recarga desde memoria a L3 y L2 también se hace a través de **Bus 2**. Siempre que se hace un rellenado en L2 el bloque crítico se escribe en paralelo en L1d y se despiertan las instrucciones dependientes de hasta 2 instrucciones *load* desde la IQ para que puedan capturar el dato a tiempo.

La Figura 3.5 muestra la segmentación en etapas del acceso a la jerarquía de

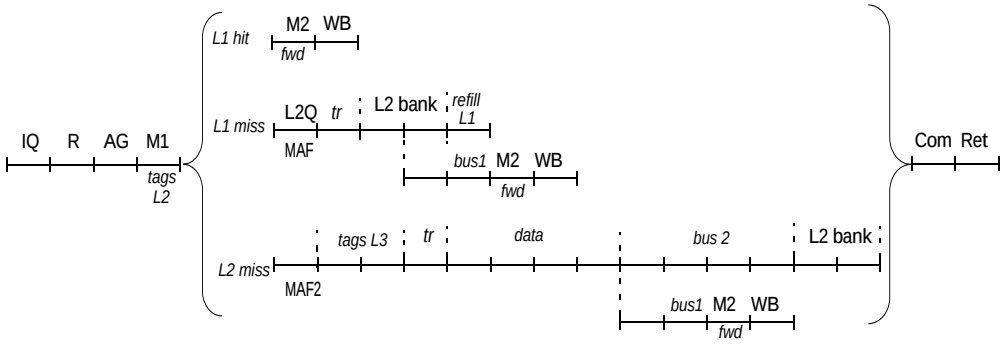

Figura 3.5 Segmentación del acceso a la jerarquía de memoria de datos para una instrucción *load*.

memoria de una instrucción *load*. Se contemplan tres casos diferentes: 1) acierto en L1d (***L1 hit***); 2) fallo en L1d y acierto en L2 (***L1 miss***); y 3) fallo en L1d y L2 y acierto en L3 (***L2 miss***). En la etapa M1 se accede a L1d, L2 tags y MAF. En la etapa M2 se envía el dato a través de la red de cortocircuitos (*fwd*). Las instrucciones pendientes de fallos en el primer nivel son reiniciadas desde la IQ a tiempo de coger el dato coincidiendo con la recarga desde el segundo nivel.

Los *stores* acceden a memoria de la misma forma que los *loads*, pero hay que tener en cuenta que en **L1d** la política de actualización de escrituras es "inmediata" y la de fallo en escritura "no-asignación no-carga" (*Write Through y Write Around*). De esta forma, todas las instrucciones *store* son consolidadas sobre L2 (*Write Back*). **L1d** dispone también de cuatro *buffers* de escritura con compactación (*Coalescing Write Buffers)* de seis entradas, que permiten realizar las escrituras en los bancos en los ciclos libres.

Por otro lado, vamos a utilizar una memoria ideal de instrucciones. No se modela ningún tipo de jerarquía, sino que cada acceso a la memoria de instrucciones sufre una latencia fija de 1 ciclo. Esta decisión no influye de manera importante en las operaciones sobre los datos. Con un sistema de memoria de instrucciones real los porcentajes de mejora de prestaciones de un mecanismo de prebúsqueda podrían ser algo menores.

Para finalizar, se muestra en la Tabla 3.2 un resumen de los parámetros principales de la jerarquía de memoria.

Tabla 3.2 Parámetros principales de la jerarquía de memoria.

Cache L1 d	16 KB, block 32 B, 2-way, lat. 2 cycles, write-through non-allocate, 4 Coalescing Write Buffers (CWB), Mem. Address File (MAF): 16 entries
Cache L1 i	ideal
Cache L2	256 KB, block 128 B, 8-way, 16 banks 16-interleaved, bank access lat: 2 cycles, ld/use lat: 8 cycles, write-back allocate, L2Q 32 entries, WB 6 entries, MAF2: 8-16 entries[a]
Cache L3	4 MB, block 128 B, 16-way, tag access: 2 cycles pipelined, data access: 4 cycles, ld/use lat: 13 cycles, write-back allocate; WB 2 entries
Memory	Latency 200 cycles; bandwith 1/20 cycles

a. 8 en el Capítulo 4, 16 en el capítulo 5.

3.4 Simulador dirigido por ejecución

Los experimentos realizados en esta Tesis se han realizado utilizando un simulador basado en SimpleScalar 3.0c [BuA97]. 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. Nosotrossimulamos código Alpha [Alp99].

De los diferentes simuladores que vienen con SimpleScalar hemos utilizado únicamente *sim-outorder,* ya que es el único de los que componen SimpleScalar que realiza la simulador temporal y por tanto el único que es capaz de dar medidas de prestaciones. Para ello se modelan ciclo a ciclo todas las etapas y actividades por las que deben pasar las instrucciones para ser ejecutadas. El simulador *sim-outorder* ha sido modificado ampliamente para modelar el procesador superescalar descrito anteriormente. Las modificaciones más complejas y significativas se refieren a los siguientes aspectos:

- 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. Las instrucciones dependientes de *loads* se inician especulativamente prediciendo que el *load* acertará en el primer nivel de *cache*, aunque en caso de fallo sólo se hace durante el primer ciclo de la *ventana especulativa*. El camino de datos de las instrucciones *load* y *store* se ha modelado con mucho detalle.

- Predictor de orden *store-load*. Se ha implementado una gestión de predicción perfecta de dependencias *store-load* en la IQ.

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

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 complejo 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*).

- Tres propuestas de mecanismos de prebúsqueda han sido enviadas a una competición de prebuscadores (DPC-1), donde los organizadores establecían el procesador modelado, el entorno de simulación y un conjunto de restricciones a seguir por los sistemas presentados. El capítulo 6 está dedicado íntegramente a nuestra participación en esta competición. Consideramosque los resultados obtenidos aportan un buen criterio devalidación de esta Tesis.

Con todo ello y aunque la simulación paso a paso sea correcta, resulta difícil validar la corrección temporal de la ejecución de las instrucciones.

Durante el desarrollo de la jerarquía de memoria se ha utilizado una aplicación visual [Tor05], que permite seguir ciclo a ciclo el estado del segmentado de una forma cómoda. Esta aplicación es muy útil a la hora de validar la corrección temporal de lo que se quiere modelar. En el Apéndice A se muestra un extracto del manual de usuario de la herramienta.

3.5 Carga de Trabajo (Benchmarks)

Como carga de trabajo hemos seleccionado un amplio conjunto de programas de prueba SPEC 2K, Olden [RCR+95] e IAbench [RIV+00] (ver Tabla 3.3).

Tabla 3.3 Tasas de fallos en L1, L2, L3, IPC y parámetros de los programas de prueba utilizados.

Progr.	L1 mr	L2 mr	L3 mr	IPC	Parameters and group	
vpr	7,2%	2,5%	0,3%	1.29	SPEC CPU 2000 int (CINT)	
gcc	2,4%	0,5%	0,1%	5.19		
mcf	34,1%	19,6%	13,2%	0.24		
parser	7,6%	0,8%	0,0%	2.27		
gap	1,4%	0,1%	0,1%	1.74		
vortex	2,5%	0,3%	0,1%	4.72		
bzip2	3,1%	1,2%	0,0%	2.44		
twolf	12,6%	4,3%	0,0%	1.96		
wupwise	3,3%	0,8%	0,7%	2.88	SPEC CPU 2000 fp (CFP)	
swim	23,8%	5,0%	5,0%	0.81		
mgrid	7,4%	1,8%	0,9%	1.94		
applu	13,8%	3,0%	2,9%	1.33		
galgel	15,7%	3,3%	0,2%	3.31		
art	73,7%	41,5%	0,0%	2.22		
equake	19,3%	3,4%	3,2%	0.50		
facerec	4,5%	2,2%	0,2%	2.07		
ammp	12,1%	4,6%	0,1%	2.74		
fma3d	3,0%	0,5%	0,4%	2.45		
apsi	1,2%	0,1%	0,1%	4.57		
em3d	27,7%	3,8%	0,0%	2.92	em3d 2000 10 100 100	Olden
mst	14,1%	10,9%	9,4%	0.76	mst 1024 1	
perimeter	4,3%	0,5%	0,6%	1.60	perimeter 10 1	
treeadd	4,7%	0,6%	0,6%	1.63	treeadd 20 1	
tsp	1,9%	0,2%	0,1%	2.60	tsp 100000 0	
csrlite	12,4%	3,1%	2,8%	0.72	csrlite 30 20 gre__115.rua impcol_c.rua mcca.rua west0156.rua	IA bench
sparse	37,1%	12,3%	0,4%	0.64	(500x500 sparse matrix with 5000 non-zero elements)	

Con respecto a SPEC 2K, 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`

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 3.7). 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 3.7 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 [SPH+02] en 2002 (*Simpoints*), tras un calentamiento de *caches* y predictores de 200 millones de instrucciones. Estos puntos de simulación fueron definidos siguiendo técnicas de análisis de fases de ejecución en programas con miles de millones de instrucciones. 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 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 interacción 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 pudiéndolas 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 SPEC 2K.

La colecciones de programas Olden e IAbench, que fueron presentadas en la sección 2.4, han sido ejecutadas completas. Estas dos colecciones sólo se utilizan en el Capítulo 4, ya que son programas que contienen especialmente patrones *linear link*.

3.6 Plataforma de Simulación

Para realizar las simulaciones de esta Tesis se ha necesitado una gran potencia de cálculo. Los primeros pasos se realizaron en máquinas Sun SPARC de los departamentos *Informática e Ingeniería de Sistemas* y *Ingeniería Electrónica y Comunicaciones* (`prometeo` y `tsc1`). Posteriormente, se utilizaron recursos de computación del Centro Europeo de Paralelismo de Barcelona CEPBA sobre dos máquinas Digital Alpha (`Kemet` y `Kerma`).

Más tarde, las simulaciones se realizaron en el cluster `atpc`, perteneciente al gaZ. Se trata de un cluster de PCs Intel Pentium 4 bajo Linux que utilizaba Condor como herramienta de distribución de trabajos. Este cluster llegó a tener hasta 36 nodos con velocidades de reloj entre 1,4 a 3,2 Ghz. Condor (`http://www.cs.wisc.edu/condor/`) es un sistema de gestión de colas de trabajos y de recursos, corre bajo Microsoft Windows NT y varios Unix. En el disco duro local de cada nodo del cluster se realiza una copia de las *trazas eio* de los programas de prueba para minimizar el tráfico de red.

Las últimas simulaciones fueron realizadas en `hermes`, otro cluster con Condor de más de 140 procesadores perteneciente al I3A.

Para la depuración de código se ha utilizado Microsoft Visual Studio 2005 - Professional Edition.

CAPÍTULO 4

Prebúsqueda *hardware* de datos en una jerarquía *on-chip* de altas prestaciones

En un procesador superescalar agresivo y con una jerarquía de memoria de alto rendimiento se ejecutan cinco prebuscadores de naturaleza muy diferente. Secuencial marcado, stride *y PC/DC se toman como prebuscadores de referencia.* Linear link *explora la aplicación a prebúsqueda del método de detección de patrones propuesto en el capítulo 2, y PDFCM es un nuevo prebuscador de correlación con un buen equilibrio rendimiento / coste. Todos ellos se evalúan con distintas agresividades.* Secuencial marcado *y PDFCM son los dos prebuscadores que más oportunidades de ganancia de prestaciones nos ofrecen.*

4.1 Introducción

Los sistemas de prebúsqueda *hardware* pueden conseguir excelentes resultados al ocultar la latencia de acceso a memoria. Además, las recientes jerarquías de memoria *on-chip* presentan una gran capacidad y ancho de banda. En este capítulo pretendemos analizar cómo se comportan las diferentes técnicas de prebúsqueda en este nuevo entorno. Por ello, vamos a utilizar un procesador superescalar fuera de orden agresivo junto con una jerarquía de memoria de gran capacidad y ancho de banda, similar a la utilizada por el procesador Itanium2 [NaH02]. Ambos han sido descritos ampliamente en el capítulo anterior.

En este entorno vamos a analizar el comportamiento de varios prebuscadores. Dos de ellos son clásicos y bien conocidos: *secuencial marcado* y *stride*; los

otros dos son prebuscadores de correlación: PC/DC y PDFCM. PC/DC responde a una familia de técnicas basadas en almacenar la secuencia de fallos en una estructura llamada GHB *(Global History Buffer)* [GMT04] [NeS04] [NeS05] [NDS04]. El recorrido a través de esta estructura se efectúa siguiendo los punteros que unen las entradas cuyos fallos fueron generados por la misma instrucción de acceso a memoria. El prebuscador PDFCM es una nueva propuesta. Es también un prebuscador de correlación, pero no correlaciona direcciones sino diferencias entre direcciones (*deltas*), siguiendo la idea de un predictor de valor, DFCM (*Differential Finite Context Machine*) que obtiene muy buenos resultados [GVB01].

En las próxima sección (4.2) se detallan los prebuscadores referidos, haciendo especial énfasis en nuestra nueva propuesta, PDFCM. La sección 4.3 expone los resultados obtenidos al simular todos ellos en el procesador definido en el capítulo anterior. Para finalizar el capítulo, la sección 4.4 presenta las conclusiones extraídas de este estudio.

4.2 Descripción de los prebuscadores analizados

4.2.1 Secuencial marcado

El prebuscador *secuencial marcado* fue introducido en la sección 1.3.2. La implementación que utilizamos en este capítulo trabaja sobre el segundo nivel de cache, es decir: 1) genera prebúsquedas cada vez que se produce un fallo o un primer uso en L2; y 2) las prebúsquedas generadas (entre 1 y 4 bloques, según el grado) son enviadas a L2. Por tanto, el coste de este prebuscador es de 1 bit por cada bloque de L2 (2 Kbits en total). Se consideran los accesos producidos tanto por instrucciones de carga *(loads)* como de almacenamiento *(stores)*.

4.2.2 Stride

El prebuscador *stride* es otro de los prebuscadores bien conocidos (sección 1.3.3). Este prebuscador usa una tabla donde a cada instrucción de acceso a memoria, *load* o *store*, se le intenta asociar el *stride* (diferencia entre direcciones) seguido. El prebuscador *stride* que vamos a utilizar sólo inserta un *load* en la tabla cuando falla en el segundo nivel de *cache*. Usando esta técnica descrita por Ibañez *et al* en [IVB+98] el tamaño de la tabla puede ser reducido considerablemente sin perder prestaciones del prebuscador. Además, la

actualización de la tabla se realiza en orden en la etapa *commit* de la instrucción de acceso a memoria.

4.2.3 Linear link

El prebuscador *linear link* utilizado se basa en el patrón del mismo nombre descrito en la sección 2.3. Cada vez que se encuentra una posible relación entre *loads* a través del mecanismo descrito en dicha sección, esta se almacena en la *Link Table*. Cuando los parámetros de la relación (α, β) sean ajustados y la relación tenga la confianza suficiente, el *load* productor, tan pronto como obtenga su dato, utilizará la recurrencia *linear link* para prebuscar las direcciones de todos sus consumidores.

4.2.4 Un prebuscador de correlación basado en GHB: PC/DC

El prebuscador de correlación basado en GHB que vamos a utilizar para este estudio es PC/DC y ha sido descrito ya ampliamente en la sección 1.3.4.

4.2.5 Un nuevo prebuscador de correlación

Vamos a definir ahora una nueva propuesta de prebuscador de correlación. Para ello nos basamos en un mecanismo de predicción de valor, que ha demostrado su efectividad, conocido como DFCM [GVB01]. Este mecanismo almacena las correlaciones que se producen en los *deltas* de los valores (diferencias entre valores) resultado de ejecutar las instrucciones, y utiliza dicha información para predecir con precisión los valores futuros. En nuestro caso, lo aplicamos a las direcciones de fallo de las instrucciones de acceso a memoria. Utilizamos la información guardada de una instrucción dada para predecir la siguiente dirección de dicha instrucción que producirá un fallo. En relación al método original aplicado a predicción de valor, la aplicación a la predicción de direcciones de fallo tiene al menos dos ventajas: 1) el número de instrucciones de acceso a memoria es menor que el número de instrucciones que producen resultado; 2) un gran número de instrucciones de acceso a memoria acierta en *cache*, y sólo aplicamos el método a las direcciones de aquellas que fallen. Estas dos ventajas permiten que el tamaño de las tablas utilizadas para aplicar el método a prebúsqueda puedan ser mucho menores que cuando se orienta a la predicción de valor.

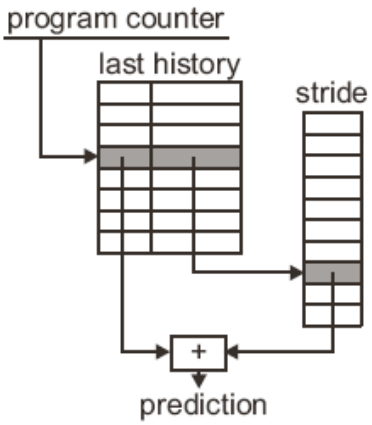

Figura 4.1 Predicción de valor basada en DFCM.

Predicción de valor basada en DFCM

Al igual que el PC/DC, el mecanismo de predicción de valor basado en DFCM usa también 2 tablas (Figura 4.1). En este caso, la primera tabla almacena el último valor producido por la instrucción y la historia de los últimos *deltas*, que son las diferencias entre valores sucesivos generados por una instrucción dada. Para ser más precisos, esta historia se almacena en la tabla tras aplicarle una función *hash*. El *hash* de la historia se usa como índice para acceder a la segunda tabla, donde se obtiene el próximo *delta* predicho.

El mecanismo tiene 2 fases: *predicción* y *actualización*. La fase de *predicción* comienza en cuanto se conoce la dirección (PC) de la instrucción. Con ella se accede a la primera tabla y se obtienen el último valor de la instrucción y el *hash* de la historia de deltas. Con este último se indexa la segunda tabla para leer el *delta* predicho. Este *delta* se suma al último valor, obteniendo así el valor predicho. Cuando la instrucción acaba su ejecución, y conocido ya su resultado, comienza la fase de *actualización*. Se resta el resultado del último valor y se comprueba que esta diferencia coincide con el delta predicho. En caso contrario se actuliza la entrada de la segunda tabla con la nueva diferencia. Finalmente, se actualiza la entrada de la primera tabla con el valor actual y con la función *hash* de la nueva historia de *deltas* (considerando también el *delta* actual).

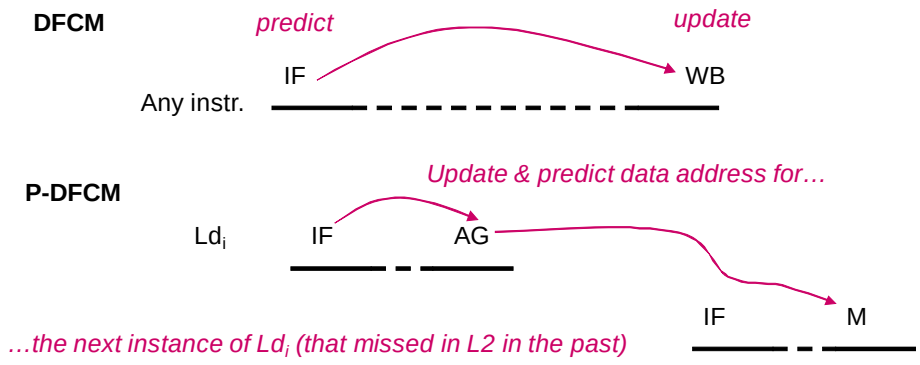

Figura 4.2 Diferencia entre el funcionamiento de DFCM y PDFCM.

PDFCM: Prebúsqueda basada en DFCM

En el prebuscador PDFCM que proponemos se busca correlacionar las diferencias (*deltas*) entre direcciones de fallo en L2 de las instrucciones de acceso a memoria. Se utilizan contadores de confianza en la primera tabla para deshabilitar la prebúsqueda de aquellas entradas en las que la predicción se efectúa incorrectamente.

Existe una notable diferencia entre ambos métodos (DFCM vs. PDFCM). En la Figura 4.2 se ve cómo en la etapa *fetch* (IF) de cualquier instrucción DFCM predice cuál va a ser su resultado. Después, en la etapa *writeback* (WB) realiza la actualización de las tablas teniendo en cuenta el resultado real de la instrucción. Sin embargo PDFCM funciona de una manera diferente. En la etapa AG (*Address Generation*) de las instrucciones de acceso a memoria que fallan en *cache*: 1) se *actualiza* la información de las tablas con la dirección que acaba de fallar; y 2) se *predice* cual será la próxima dirección de memoria lanzada por dicha instrucción que fallará. La inserción en la tabla se realiza en orden, en la etapa *commit*.

La Figura 4.3 describe con más detalle el funcionamiento de PDFCM. Tan pronto como se conoce la dirección de la instrucción de acceso a memoria, *load* o *store*, y se sabe que produce un fallo en *cache*, actualizamos su entrada correspondiente en las tablas. Para ello (ver Figura 4.3.a): 1) se calcula un nuevo *delta* restando la dirección actual y la anterior (que estaba almacenada en la primera tabla); 2) se indexa la *Delta Table* (DT) con la historia de los últimos *deltas* y se guarda el nuevo *delta*; 3) a partir de la historia de *delta* y el nuevo

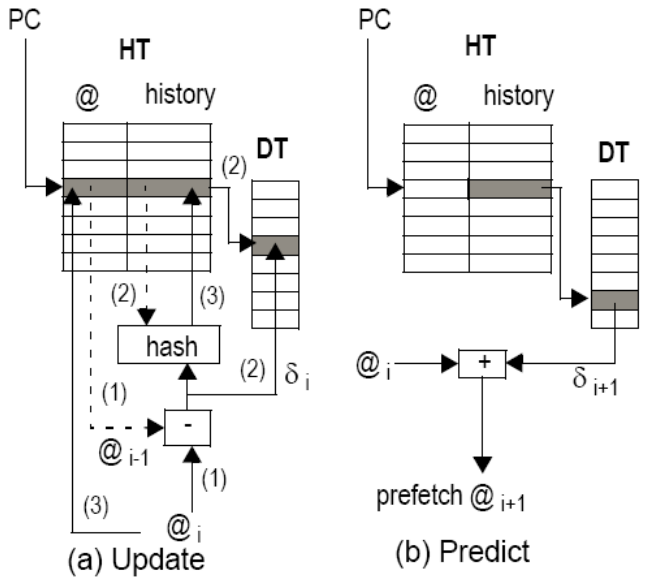

Figura 4.3 Funcionamiento de PDFCM.

delta se crea una nueva historia que es almacenada en la entrada de la *History Table* (HT) junto con la dirección actual. Justo después de la actualización se efectúa la fase de predicción (Figura 4.3.b), donde se indexa la DT con la nueva historia para conseguir un *delta*, que sumado a la dirección actual genera la predicción de la próxima dirección de fallo de dicha instrucción de acceso a memoria.

Finalmente, se incluye el algoritmo de funcionamiento de PDFCM en la Figura 4.4.

4.3 Resultados

Las simulaciones se efectúan usando la plataforma de simulación descrita en la sección 3.6 y modelando el procesador que se ha detallado ampliamente en la sección 3.2. junto con la jerarquía de memoria explicada en la sección 3.3.

Las aplicaciones de prueba o *benchmarks* ejecutados son SPEC CPU 2k, Olden e IABench (ver descripción en sección 3.5). Para las aplicaciones SPEC CPU 2k ejecutamos únicamente los puntos de simulación simples (*single simulation points* [SPH+02]) con un precalentamiento previo de *caches* y predictor de saltos de 200 millones de instrucciones. Olden e IABench se ejecutan completos. De

```
Global types and variables:
    HT_entry fields:
      {tag, last_address; history; confidence;}
    DT_entry DT[DT_SIZE];
    HT_entry HT[HT_SIZE];
    HT_entry hte; // temporal register to hold an HT entry
    int last_δ, act_δ;
    pred_address; // store the address we predict each cycle
A. Update and Predict
Parameters:
    PC; // address of the ld or st instruction
    address; // primary miss or 1st use of prefetched block
Locals:
    Register new_history;
    int pred_δ; // predicted delta
// Update and Predict begin{
01  Look up HT with PC;
02  if ( hit ) // read HT entry and save it to a register:
03    hte = HT[PC];
04  else
05    replace HT entry; exit;
06  last_δ = DT[hte.history];
07  act_δ = address - hte.last_address;
08  DT[hte.history] = act_δ;
09  hte.history = hash(hte.history, act_δ );
10  hte.last_address = address;
11  HT[PC] = hte;
12  (act_δ == last_δ ? hte.confidence++; hte.confidence--;)
13  if (hte.confidence >= 2 ) {
14    pred_δ = DT[hte.history];
15    pred_address = address + pred_δ;
    }
} // Update and Predict end
B. Predict Next // called when degree > 1
Locals:
    int next_δ; // speculative stride used to apply degree
      // we use hte.history for the speculative history
// Predict next begin{
01  next_δ = pred_address - address;
02  hte.history = hash(hte.history, next_δ );
03  next_δ = DT[hte.history];
04  pred_address += next_δ;
} // Predict next end
```

Figura 4.4 Algoritmo de funcionamiento de PDFCM.

todas las aplicaciones, se han seleccionado únicamente aquéllas que con una *cache* de datos de segundo nivel ideal consiguen un *speed-up* mínimo de 2%.

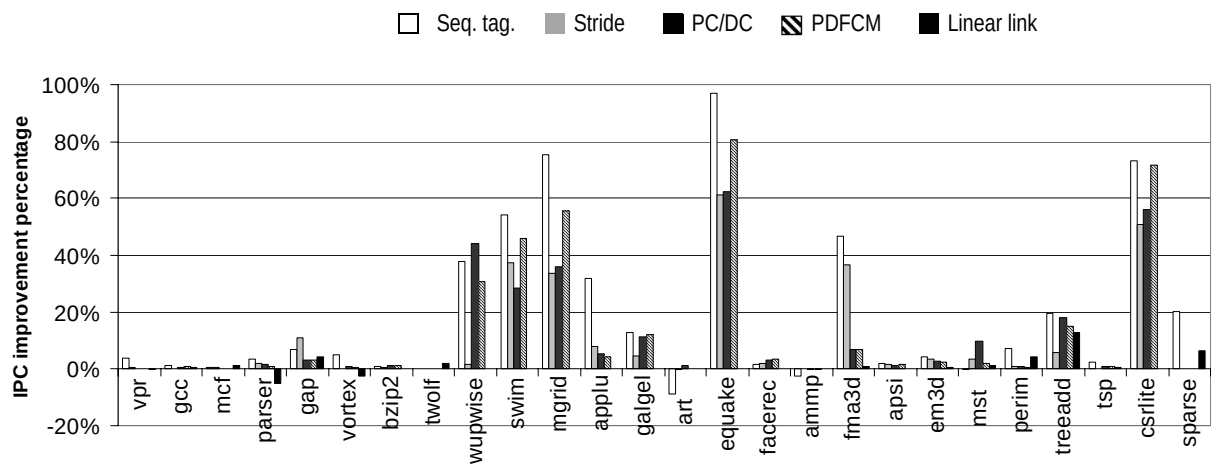

Figura 4.5 Porcentaje de mejora en IPC según prebuscador (grado 1, distancia 1).

En la Figura 4.5 se muestra el *speed-up* obtenido en cada aplicación por cada prebuscador utilizando grado de prebúsqueda 1 y distancia de prebúsqueda 1. Por un lado, la ganancia obtenida por el prebuscador *linear link* no es muy grande y afecta a un número reducido de aplicaciones (treeadd: 13%, sparse: 7%, perim: 4%, twolf: 2%). En un procesador como el que modelamos, que puede llegar a lanzar hasta 12 instrucciones por ciclo, la distancia entre productor y consumidor es habitualmente muy pequeña (de 1 a 4 ciclos), por lo que el ahorro generado por la prebúsqueda no resulta significativo.

Analizando los resultados del resto de prebuscadores observamos que una jerarquía de cache *on-chip* de altas prestaciones ofrece una gran oportunidad a antiguos prebuscadores como *secuencial marcado* o *stride*. De las trece aplicaciones en las que algún sistema de prebúsqueda obtiene más de un 5% de mejora, el *secuencial marcado* es el mejor en diez de ellas. Sin embargo, hay dos aplicaciones (art y ammp) en las que ya con grado 1 muestran pérdidas.

Las dos figuras siguientes (Figura 4.6 y Figura 4.7) dan una idea de la precisión y de la agresividad de cada prebuscador, respectivamente. Podemos observar que el número de prebúsquedas inútiles se dispara con el prebuscador *secuencial marcado*, hasta el punto de provocar pérdidas en dos de las aplicaciones. Sin embargo, hay otras aplicaciones (como vpr y sparse) en las que el *secuencial marcado* es el único método de prebúsqueda capaz de obtener mejoras.

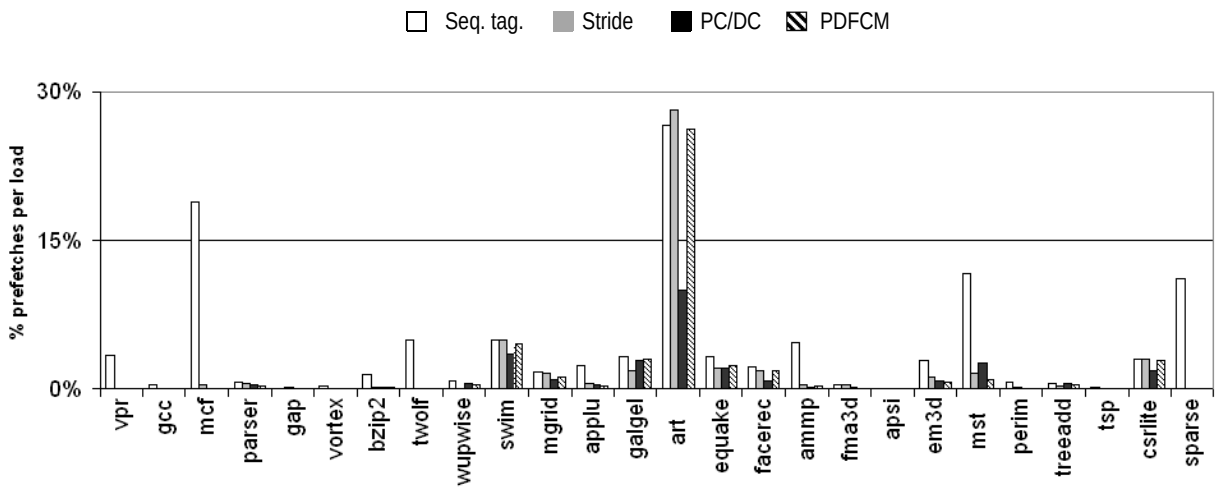

Figura 4.6 Porcentajes de prebúsquedas generadas por load consolidado.

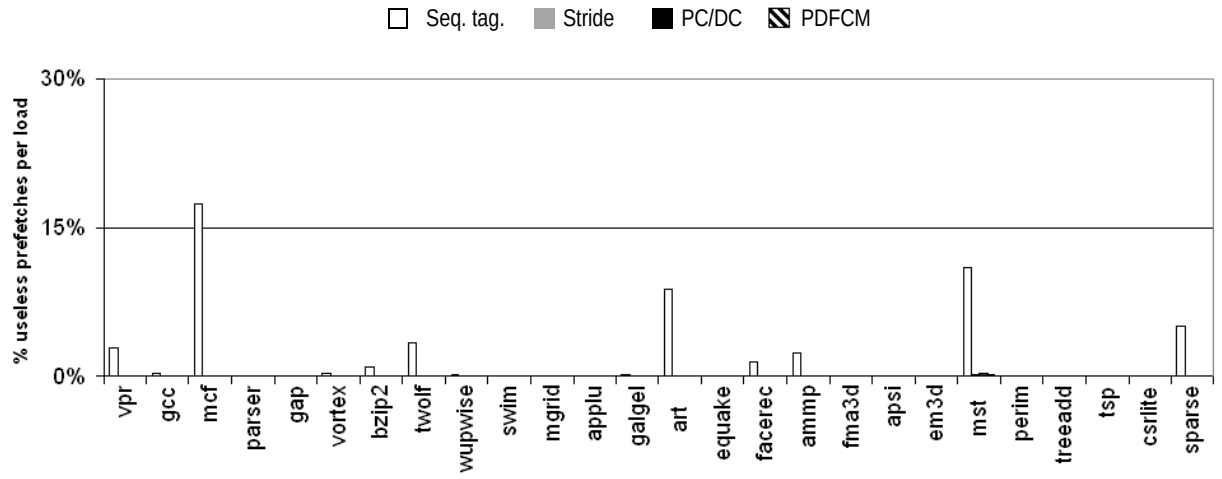

Figura 4.7 Porcentajes de prebúsquedas inútiles por load consolidado.

Por otro lado, y tal como se esperaba, *stride*, PC/DC y PDFCM tienen un comportamiento realmente selectivo. Si los comparamos, podemos ver que PDFCM es el mejor en cinco programas, *stride* el mejor en tres programas y PC/DC es el mejor en otros tres.

Vamos a fijarnos ahora en qué es lo que sucede cuando variamos el grado y la distancia de prebúsqueda. En general, incrementar el grado o la distancia beneficioso para todos los prebuscadores. Pero observemos el comportamiento del *secuencial marcado* por aplicación (Figura 4.8). En los casos en que con

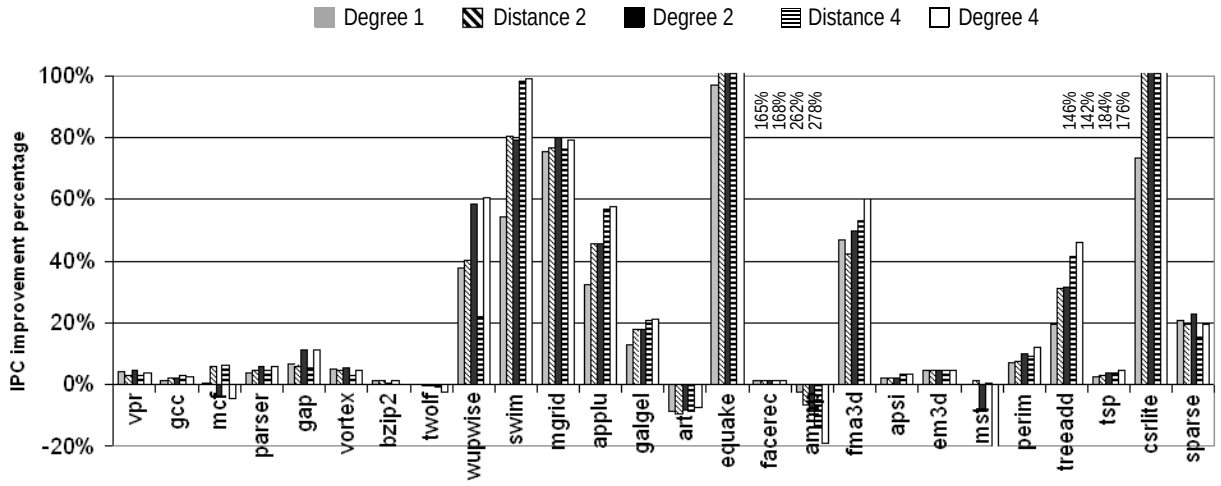

Figura 4.8 Porcentaje de mejora de IPC por aplicación para *Secuencial Marcado* con distintos grados y distancias.

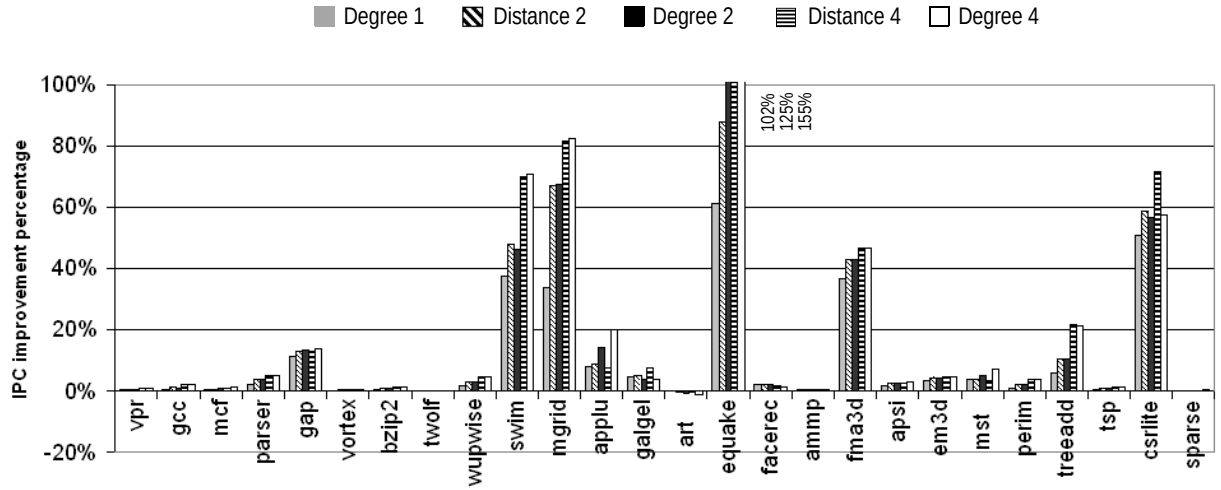

Figura 4.9 Porcentaje de mejora de IPC por aplicación para *Stride* con distintos grados y distancias.

grado 1 se obtenían beneficios, éstos se incrementan al subir el grado o la distancia. Sin embargo, en otras aplicaciones aparecen nuevas pérdidas o se incrementan las que ya había con grado 1.

Si miramos el prebuscador *stride* (Figura 4.9), vemos que obtiene prestaciones bastante similares al incrementar el grado o la distancia de prebúsqueda. Sin embargo debemos tener en cuenta que incrementar la distancia genera mucha menos prebúsquedas que incrementar el grado.

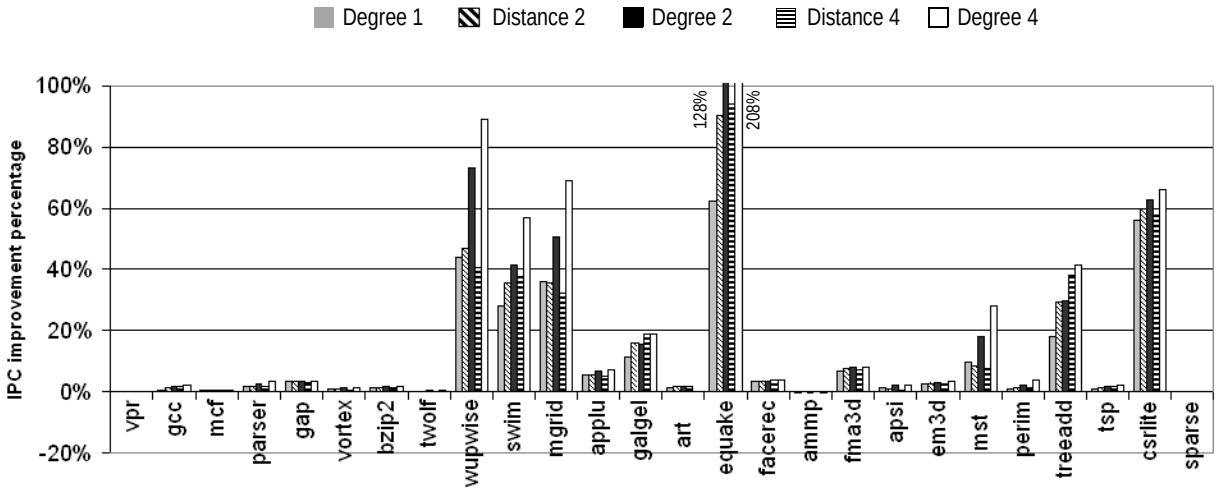

Figura 4.10 Porcentaje de mejora de IPC por aplicación para PC/DC con distintos grados y distancias.

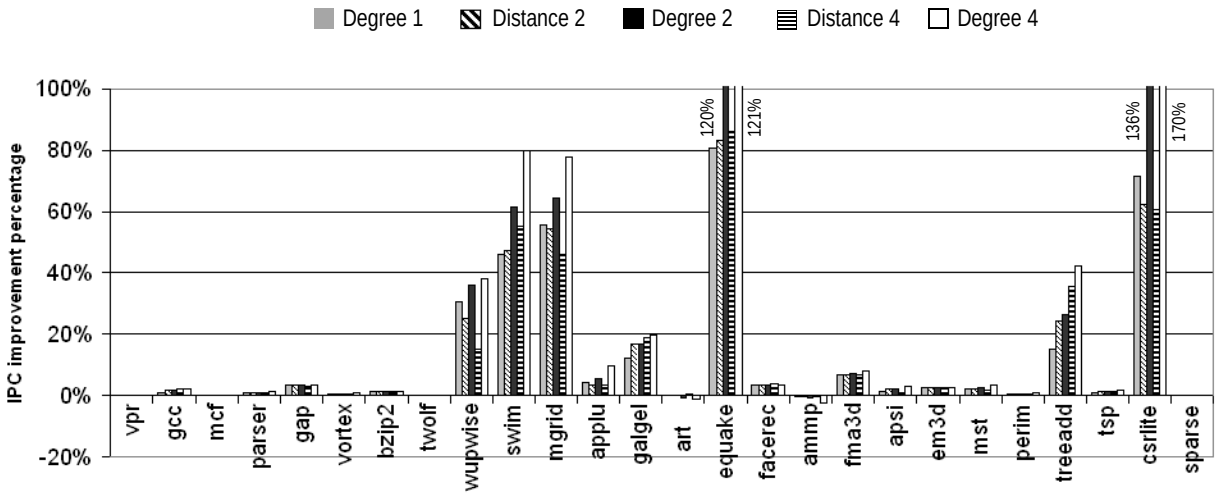

Figura 4.11 Porcentaje de mejora de IPC por aplicación para PDFCM con distintos grados y distancias.

El prebuscador PC/DC, (Figura 4.10) por su parte, obtiene poco beneficio al incrementarse la distancia o incluso la mejora de IPC decrece (wupwise, mgrid y mst). Sin embargo, incrementar el grado de prebúsqueda siempre es positivo sin ningún coste adicional.

Las conclusiones con respecto a PDFCM son similares (Figura 4.11) aunque en este caso el incrementar el grado o la distancia conlleva cierto coste *hardware*. Puede observarse que los resultados al aumentar el grado son un poco mejores que cuando se aumenta la distancia.

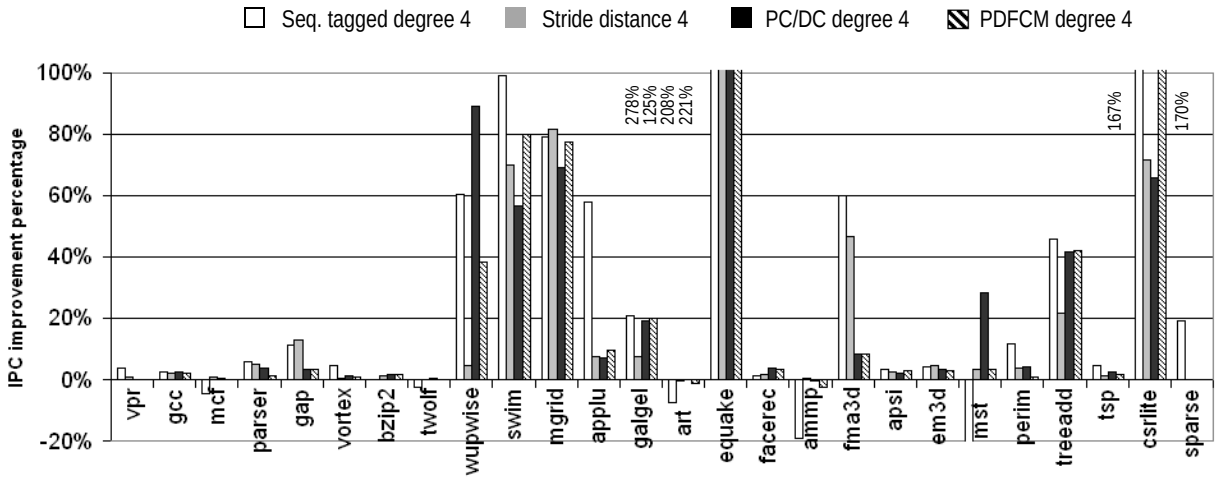

Figura 4.12 Porcentaje de mejora de IPC por aplicación para todos los prebuscadores con el grado o distancia indicado.

Para clarificar las comparaciones, hemos reunido los resultados de todos los prebuscadores (Figura 4.12) eligiendo la opción más razonable para cada uno de ellos (distancia 4 para *stride* y grado 4 para el resto), sobre las catorce aplicaciones en las que algún prebuscador obtiene un *speed-up* mayor que 5%. Queremos destacar las siguientes observaciones:

- En diez de las catorce aplicaciones *secuencial marcado* es el mejor, pero presenta perdidas en otras cuatro.
- *Stride* es el mejor en parser, gap, mgrid y fma3d.
- PC/DC y PDFCM se comportan de forma similar. PDFCM obtiene más prestaciones en cuatro aplicaciones y PC/DC en dos.

Sin embargo, si miramos el número de accesos a las tablas por *load* consolidado de ambos prebuscadores (Figura 4.13), PC/DC accede 2,6 veces más que PDFCM cuando consideramos grado 4, relación que aumenta a cuatro veces más en grado 1, debido a que el PC/DC necesita el mismo número de accesos para cualquier grado.

4.4 Conclusiones

Hemos analizado el comportamiento de diversos prebuscadores en términos de prestaciones, número de prebúsquedas generadas y número de prebúsquedas

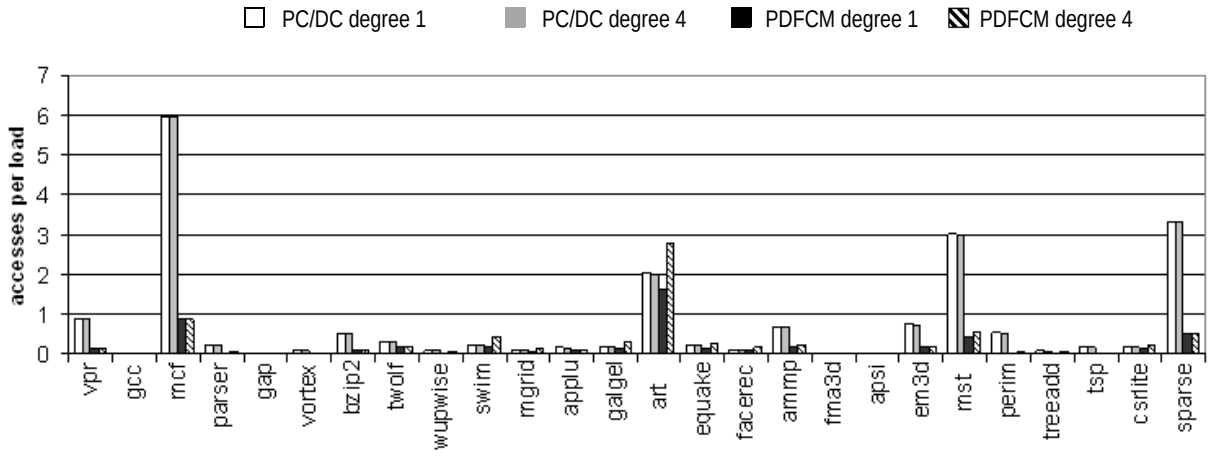

Figura 4.13 Número de accesos a tablas por load consolidado por aplicación según prebuscador y grado indicado.

usadas en un procesador superescalar agresivo junto a una jerarquía de memoria *on-chip* de altas prestaciones.

Por un lado, el prebuscador *linear link* no parece tener éxito en un procesador fuera de orden tan agresivo. Un análisis más en profundidad muestra que el *load* consumidor se lanza pocos ciclos después con lo que la ganancia queda oculta por el fuera de orden. Por ello nos centramos en lo que sigue en los prebuscadores de correlación y en el *secuencial marcado*.

Por otro lado, hemos visto cómo el aumento de la agresividad, tanto en forma de grado como de distancia, es beneficioso en general para todos los prebuscadores. En media, parece que el *secuencial marcado* es un claro ganador, pero también genera grandes pérdidas en algunas aplicaciones, especialmente al aumentar el grado o la distancia, debido al número de prebúsquedas inútiles generadas. A pesar de esto, consigue ganancias en otras aplicaciones en las que los prebuscadores basados en reconocimiento de patrones no consiguen resultados. Es decir, la agresividad del *secuencial marcado* resulta beneficiosa en aplicaciones con recorridos erráticos de memoria, cuando las estructuras no están reservadas de forma muy dispersa. También parece que merece la pena prestar atención a cómo reducir las pérdidas en aquellas aplicaciones en las que este prebuscador fracasa, manteniendo los activos de sencillez y bajo coste que le hacen atractivo frente a otros prebuscadores más complejos.

También se aprecia que los beneficios de los prebuscadores basados en patrones en las aplicaciones de enteros son muy bajos, y siguen siendo un reto a conseguir.

Finalmente, y con respecto a PC/DC, hemos demostrado que es posible un enfoque alternativo en el que manteniendo similar el tamaño de las tablas, se eviten los recorridos de listas encadenadas y se consigan iguales o mejores prestaciones.

CAPÍTULO 5

Prebúsqueda de datos adaptativa de bajo coste

El prebuscador secuencial marcado *es uno de los prebuscadores más simples y agresivos que existe. En media consigue muy buenas prestaciones, pero puede provocar pérdidas importantes en algunas aplicaciones. Aquí se proponen diversas formas de adaptar la agresividad de este prebuscador para minimizar las pérdidas por aplicación, manteniendo las prestaciones medias y el bajo coste hardware.*

5.1 Introducción

Los sistemas de prebúsqueda que usan *secuencial marcado* (ver sección 1.3.2) son atractivos porque con un coste bajo pueden obtener los mejores *speed-ups*. Sin embargo también pueden provocar una gran presión sobre el sistema de memoria, e incluso la aparición de pérdidas en algunas aplicaciones hostiles. Para intentar reducir estas pérdidas se pueden aplicar técnicas de filtrado y técnicas de ajuste de la agresividad del prebuscador (grado y distancia de prebúsqueda). Algunas de estas técnicas necesitan una cantidad de *hardware* considerable [ZhL07], por lo que nos hemos planteado el objetivo de investigar mecanismos simples y efectivos que pueda ser atractivos para la industria.

En este capítulo vamos a usar el prebuscador más simple conocido (*secuencial marcado*) junto a distintas políticas de grado, que serán definidas y evaluadas. Todas ellas serán comparadas con cuatro prebuscadores de referencia. El primero de ellos es el prebuscador *stride* optimizado (sección 4.2.2). Los otros

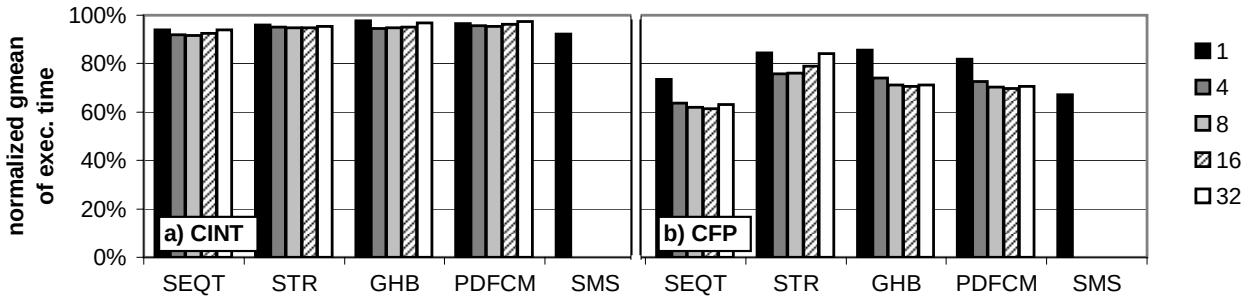

Figura 5.1 Prestaciones obtenidas por todos los prebuscadores.

tres pueden considerarse como las tres mejores propuestas de los últimos 5 años: un prebuscador basado en GHB (sección 1.3.4), un prebuscador de correlación (sección 4.2.5) y un prebuscador SMS (*Spatial Memory Streaming*) [WA+06].

La siguiente sección de este capítulo (5.2) presenta los resultados preliminares al comparar los distintos prebuscadores según sus prestaciones y la presión ejercida por las prebúsquedas en la jerarquía de memoria. En la sección 5.3 se definen todas las políticas de grado-distancia que serán usadas para ajustar la agresividad del prebuscador *secuencial marcado*. Después, en la sección 5.4 se evalúan todas las políticas en términos de prestaciones y de presión sobre el sistema de memoria. La siguiente sección (5.5) describe la utilización del *Prefetch Address Buffer* (PAB) como un elemento de filtrado de prebúsquedas redundantes. Finalmente, la sección 5.6 presenta las conclusiones.

5.2 Comparativa preliminar de los prebuscadores

La Figura 5.1 muestra una comparativa de las prestaciones obtenidas en las mejores opciones grado / distancia de todos los prebuscadores. Para cada prebuscador se representa la media geométrica de los tiempos de ejecución normalizados. Cada prebuscador es ejecutado con diferentes grados 1, 4, 8, 16 y 32, excepto SMS y *stride* (STR). En el caso de SMS, el grado de prebúsqueda es, por la naturaleza del método, variable entre 1 y 16, dependiendo del patrón reconocido. Los resultados en STR han resultado mejores al aplicar políticas de distancia. Las aplicaciones se han separado según su tipo: entero o coma flotante. En la figura se aprecia cómo el *secuencial marcado* (SEQT) con grado 8 consigue los mejores resultados, aunque con una diferencia despreciable en el caso de las aplicaciones de enteros. En las aplicaciones de coma flotante el tiempo de ejecución medio se reduce un 8% más que en el caso de SMS.

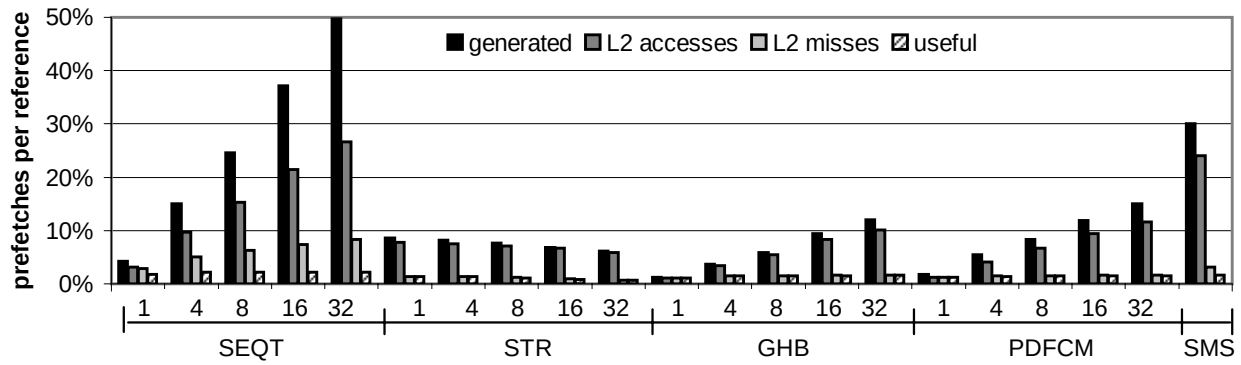

Figura 5.2 Presión sobre la jerarquía de memoria de los prebuscadores SEQT, STR, GHB, PDFCM, y SMS. Las métricas indicadas son sobre el número de referencias consolidadas.

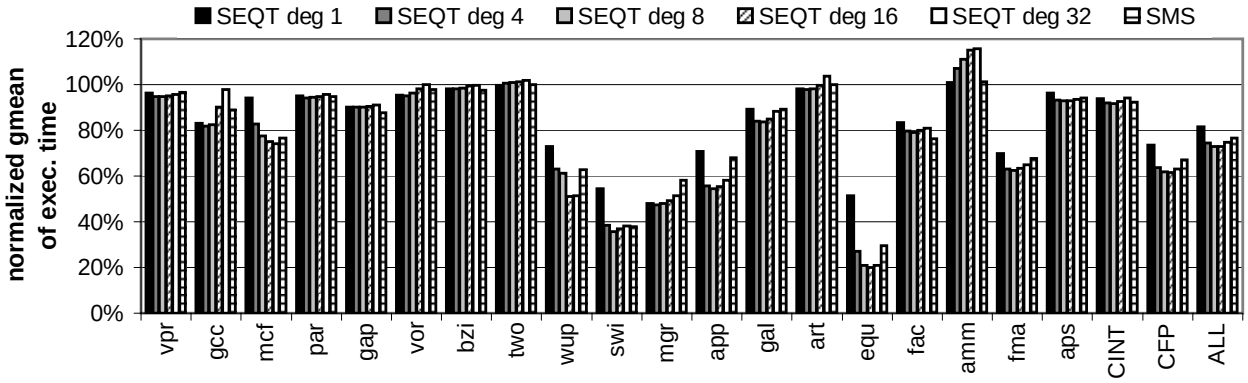

Figura 5.3 Descomposición por aplicación de las prestaciones obtenidas por los prebuscadores SEQT y SMS.

Si miramos la presión de las prebúsquedas sobre la jerarquía de memoria (Figura 5.2) vemos que *secuencial marcado* y SMS ejercen una presión mucho mayor que el resto de prebuscadores, que resultan ser mucho más selectivos.

Por otro lado, si miramos la descomposición por aplicación de los 2 mejores prebuscadores (*secuencial marcado* grado 8 - *SEQT deg 8* y SMS) en la Figura 5.3, podemos apreciar que el prebuscador *SEQT deg 8* provoca pérdidas insignificantes en twolf y pérdidas algo más importantes en ammp.

Estos resultados confirman la intuición que ya se adelantó en el Capítulo 4: merece la pena buscar mecanismos que permitan reducir las pérdidas en las aplicaciones hostiles por la excesiva agresividad de estos prebuscadores, y que mantengan su atractiva simplicidad y bajo coste.

5.3 Políticas de grado-distancia

En esta sección definiremos diferentes políticas de grado-distancia que puedan ser aplicadas a los prebuscadores agresivos como *secuencial marcado*, para ajustar su agresividad dinámicamente. En todo caso, buscamos políticas sencillas que tengan un *coste hardware* muy bajo. El funcionamiento de todas ellas será evaluado en la sección 5.4.

Cada política será definida con la ayuda de un gráfico en el que se representan todos los tipos de acceso a memoria: prebúsquedas, fallos de demanda, aciertos de demanda y primeros usos de prebúsqueda. Todos ellos se distribuyen en un espacio en 2 dimensiones en el que el eje *x* indica la dirección del bloque accedido y el eje *y* representa el tiempo en el que se ha producido el acceso. Algunas políticas son ya conocidas, otras son nuevas. Se recogen aquí todas ellas.

Prebúsqueda con grado fijo, Deg(*x*)

Cuando el prebuscador *secuencial marcado* usa la política Deg(*x*) (también conocida como *grado fijo x*) prebusca los siguientes *x* bloques cada vez que se produce un fallo o un primer uso de un bloque prebuscado.

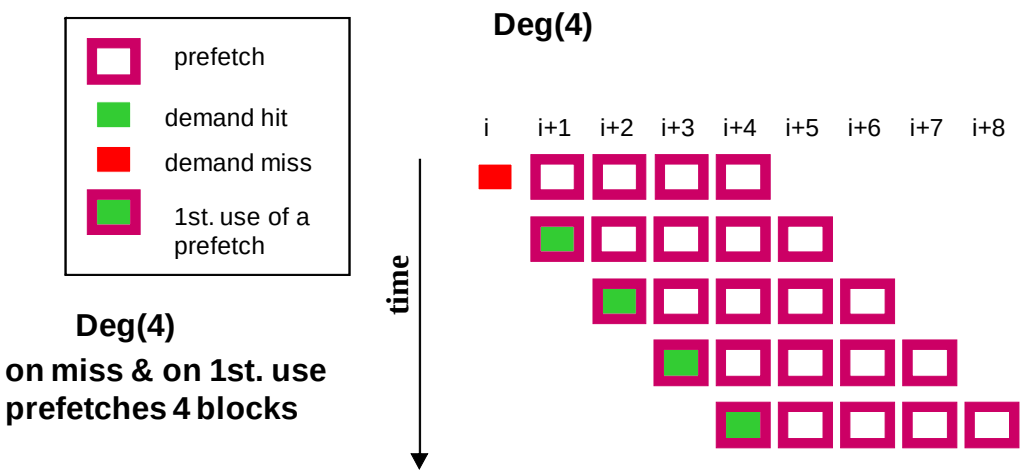

Figura 5.4 Comportamiento de la política de grado Deg(4) cuando se produce en memoria un recorrido secuencial (*i*, *i*+1, *i*+2, *i*+3, *i*+4).

En la Figura 5.4 se puede ver un ejemplo del funcionamiento de la política. Deg(4) cuando se produce en memoria un recorrido de la secuencia *i*, *i*+1, *i*+2, *i*+3, *i*+4. Se observa cómo al producirse un fallo de demanda en el bloque *i* se prebuscan los bloques *i*+1, *i*+2, *i*+3 e *i*+4. Después, el uso del bloque prebuscado *i*+1 provoca la prebúsqueda de *i*+2, *i*+3, *i*+4 e *i*+5. Lo mismo ocurre con el primer uso de los siguientes bloques. Se puede observar cómo cada bloque es prebuscado cuatro veces, lo cual provoca en memoria una sobrecarga de accesos innecesaria.

Ésta suele ser la política de grado por defecto del *secuencial marcado*. Su agresividad es muy alta, y produce pérdidas de rendimiento en algunas aplicaciones.

Prebúsqueda a distancia fija, Dist(*x*)

Cuando la política usada por el prebuscador *secuencial marcado* es Dist(*x*) (*distancia fija x*) se prebusca únicamente el bloque *x*-ésimo.

En la Figura 5.5 se representa el comportamiento del prebuscador con política. Dist(4) para el mismo recorrido secuencial. En esta ocasión el fallo en el bloque *i* provoca la prebúsqueda de *i*+4. Los consiguientes fallos en *i*+1, *i*+2 e *i*+3 prebuscan los bloques *i*+5, *i*+6 e *i*+7 respectivamente. Finalmente el primer uso del bloque prebuscado *i*+4 provoca que se prebusque *i*+8.

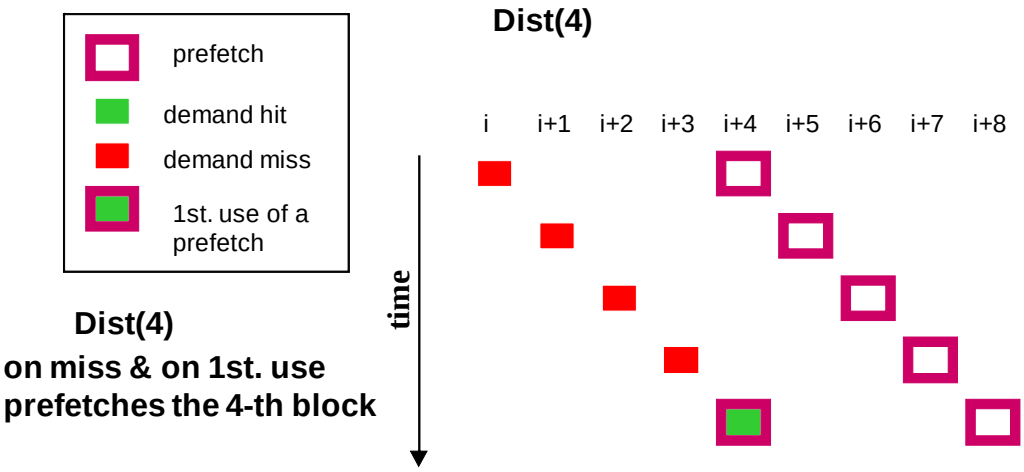

Figura 5.5 Comportamiento de la política de grado Dist(4) cuando se produce en memoria un recorrido secuencial (*i*, *i*+1, *i*+2, *i*+3, *i*+4).

Puede observarse que en esta ocasión no se generan prebúsquedas repetidas, pero se producen *x* fallos hasta que se comienzan a referenciar los bloques prebuscados. Esta política de grado es bastante menos agresiva que la anterior.

Prebúsqueda con grado-distancia, Deg-dist(*x*)

La política Deg-dist(*x*) es una mezcla de las dos anteriores. Cuando se produce un fallo se comporta como Deg(*x*), prebusca *x* bloques; cuando se produce un primer uso de prebúsqueda se comporta como Dist(*x*), prebuscando sólo el bloque *x*-ésimo.

En la Figura 5.6 se muestra un ejemplo del comportamiento del prebuscador con política Deg-dist(4). Se observa cómo al producirse el fallo en el bloque *i* se prebuscan los bloques *i*+1, *i*+2 e *i*+3. Los accesos sucesivos por demanda a *i*+1, *i*+2, *i*+3 e *i*+4 provocan sendos primeros usos de dichos bloques prebuscados y la generación de las prebúsquedas *i*+5, *i*+6, *i*+7 e *i*+8 respectivamente. La agresividad de esta política de grado es similar a la de Deg(*x*), aunque esta no genera prebúsquedas redundantes. En el caso de confirmarse el recorrido secuencial sólo se produce un fallo de demanda. Sin embargo, si no se produce dicho recorrido se generan *x* prebúsquedas que resultarán inútiles.

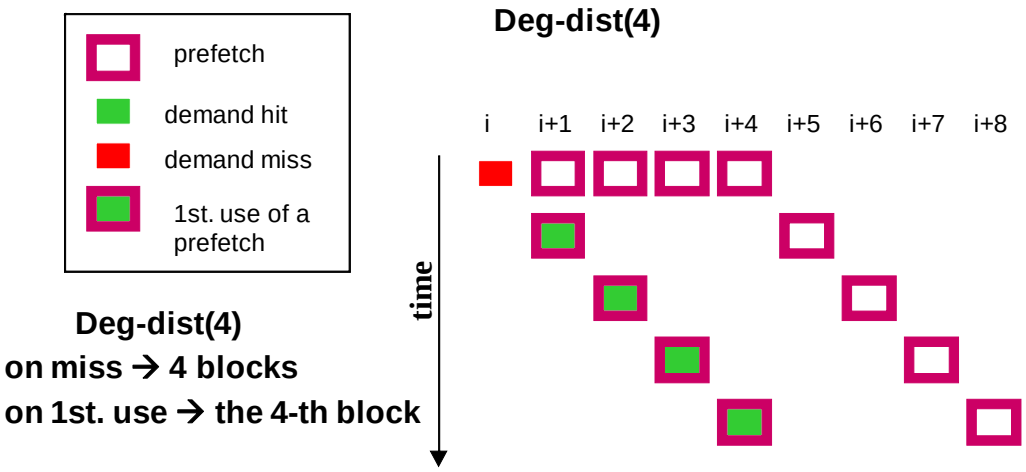

Figura 5.6 Comportamiento de la política de grado Deg-dist(4) cuando se produce en memoria un recorrido secuencial (*i*, *i*+1, *i*+2, *i*+3, *i*+4).

Grado variable simple, Deg(1-*x*)

La política Deg(1-*x*) también tiene comportamiento diferente según el tipo de evento. En caso de fallo prebusca un único bloque. En caso de primer uso de prebúsqueda genera *x* prebúsquedas.

La Figura 5.7 muestra un ejemplo de un prebuscador *secuencial marcado* con política Deg(1-4) ante un recorrido secuencial de demanda. El fallo en el bloque *i* genera la prebúsqueda del bloque *i*+1. Los primeros usos de los bloque *i*+1, *i*+2 *i*+3 e *i*+4 provocan cada uno de ellos la prebúsqueda de los 4 siguientes bloques en cada caso.

De nuevo, en esta política, aparecen las prebúsquedas redundantes. Cada bloque es prebuscado *x* veces. Sin embargo, su agresividad es un poco más moderada, ya que en el caso de no encontrarnos ante un recorrido secuencial, sólo se habría prebuscado un bloque inútil.

Grado adaptativo simple en un sentido: Ad1(*x*)

Ad1(*x*) es una política de grado adaptativa. Su agresividad se adapta dinámicamente según la utilidad de las prebúsquedas previas. En el caso de un fallo de demanda se prebusca siempre un bloque. Cuando se produce un primer uso, se prebuscan un número variable de bloques, entre 0 y *x*, dependiendo de dicha medida de utilidad. Para decidir cuál es el número de bloques a prebuscar

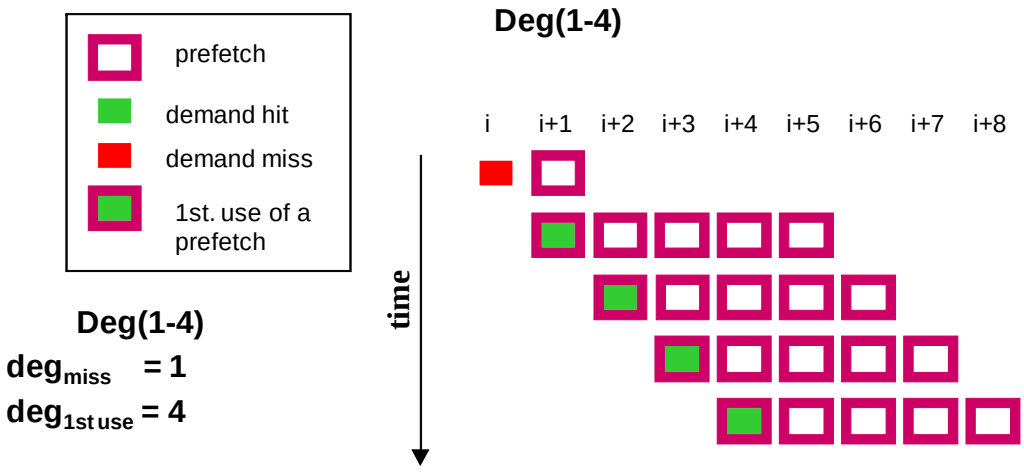

Figura 5.7 Comportamiento de la política de grado Deg(1-4) cuando se produce en memoria un recorrido secuencial (*i*, *i*+1, *i*+2, *i*+3, *i*+4).

en cada momento se utilizan dos contadores, uno para prebúsquedas usadas y otro para prebúsquedas no usadas y reemplazadas. El funcionamiento del sistema es sencillo. Cada vez que se cuentan cien primeros usos de prebúsqueda se incrementa el grado (saturando en *x*); cada vez que se cuentan cincuenta reemplazos de prebúsquedas no usadas se decrementa el grado (saturando en 0). De esta forma, la agresividad del prebuscador se adapta dinámicamente. Cada prebúsqueda no usada contribuye a que la agresividad se decremente; y cada prebúsqueda usada contribuye a que la agresividad se incremente.

En la Figura 5.8 se puede observar un ejemplo de una política. Ad1(4). Se supone un grado inicial 0 (representado en la figura por la variable *deg*). El fallo por demanda en el bloque *i* provoca la prebúsqueda de un único bloque (*i*+1). El primer uso del bloque prebuscado *i*+1 no genera ninguna prebúsqueda, ya que *deg* = 0. El fallo del acceso por demanda a *i*+2 genera la prebúsqueda de *i*+3. El primer uso de *i*+3 no genera prebúsqueda, pero supongamos que hace que el contador de prebúsquedas usadas llegue a 100. Esto hará que *deg* se incremente a 1. El fallo en *i*+4 genera la prebúsqueda de *i*+5. Ahora sí, el primer uso de *i*+5 genera la prebúsqueda de un bloque (*i*+6), ya que *deg*=1.

Ad1(*x*) podría ser definida como Deg(1-*deg*), siendo la variable *deg* ajustada según el mecanismo descrito previamente dependiendo de la utilidad de las prebúsquedas. Por lo tanto, su agresividad mínima es igual a la de una política Deg(1-0) y la máxima la de Deg(1-*x*). Cuando *deg*>1 y se producen primeros usos, esta política también genera prebúsquedas redundantes. En el caso de

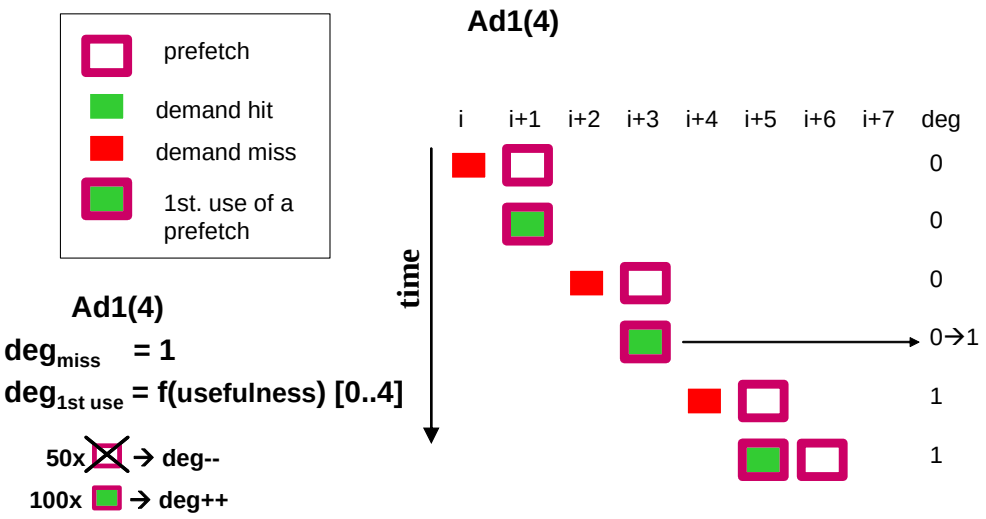

Figura 5.8 Comportamiento de la política de grado Ad1(4) cuando se produce en memoria un recorrido secuencial (*i*, *i*+1, *i*+2, *i*+3, *i*+4, *i*+5).

encontrarnos ante un recorrido no secuencial, sólo se generaría una prebúsqueda inútil.

Grado adaptativo simple en dos sentidos: Ad2(*x*)

Ad2(*x*) es nuestra siguiente propuesta de política de grado adaptativa. Funciona de forma similar a la política de grado anterior, pero además es capaz de detectar y adaptarse a recorridos secuenciales en memoria en ambos sentidos. Para ello, cuando se produce un fallo por demanda, se prebuscan dos bloques, el anterior y el siguiente al que ha producido el fallo, siendo marcado cada uno de ellos con un bit que indica el sentido (ascendente o descendente). Posteriormente, cuando se produce un primer uso de prebúsqueda, se consulta el bit que indica el sentido del recorrido y se prebuscan los siguientes *deg* bloques en dicho sentido. En este caso, el número de bloques a prebuscar viene también determinado por la utilidad de las prebúsquedas previas, usando el mismo mecanismo que la política de grado anterior.

En la figura se observa un ejemplo de una política Ad2(4). Se supone que *deg* vale inicialmente 2. El fallo por demanda en el bloque *i* causa que se prebusquen los bloques *i*+1 e *i*-1, marcándose cada bloque prebuscado con el sentido (ascendente o descendente) con respecto al bloque que provocó el fallo. Cuando se referencia por primera vez el bloque *i*+1, se observa que está marcado con sentido ascendente, por lo que se prebuscan 2 bloques (deg=2) en sentido ascendente *i*+2 e *i*+3. De la misma forma el acceso al bloque prebuscado

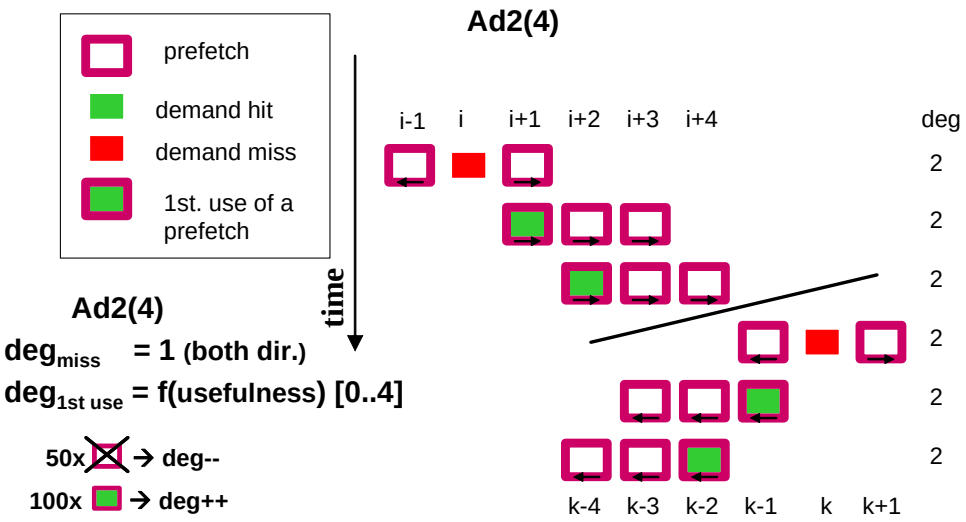

Figura 5.9 Comportamiento de la política de grado Ad2(4) cuando se produce en memoria los recorridos secuenciales (*i*, *i*+1, *i*+2, y *k*, *k*-1, *k*-2).

i+2 genera las prebúsquedas de *i*+3 e *i*+4. Posteriormente otro acceso al bloque *k* provoca un nueva fallo y se prebuscan las bloque *k*+1 y *k*-1. Esta vez, el recorrido es descendente, por lo que al usar el bloque *k*-1 se prebusca *k*-2 y *k*-3 y al usar *k*-2 se prebusca *k*-3 y *k*-4.

La política de grado Ad2(*x*) también genera prebúsquedas redundantes cuando *deg*>1 y se producen primeros usos de prebúsqueda. En el caso de encontrarnos en un recorrido no secuencial, se generan 2 prebúsquedas inútiles por cada fallo de demanda producido.

Grado adaptativo basado en puntualidad y polución, Ad3(*x*)

La política de grado adaptativa Ad3(*x*) es también similar a Ad1(*x*), pero en este caso la agresividad de prebuscador se ajusta no sólo utilizando la utilidad de las prebúsquedas, sino también su *puntualidad* y la *polución* que generan (ver sección 1.3.1). La utilidad de las prebúsquedas se mide con dos contadores, igual que en Ad1(*x*). Para cuantificar la puntualidad se utiliza un contador, que es incrementado cada vez que se detecta una prebúsqueda tardía, esto es, un primer uso de una prebúsqueda que aún no ha finalizado. Cada cincuenta prebúsquedas tardías, el grado del prebuscador es incrementado, aumentando así su agresividad. Se necesita también otro contador para los eventos de polución. Éstos se detectan a través de un filtro de Bloom, direccionando una tabla de bits con una función *hash* (*xor* entre los primeros y los últimos doce bits) de los bloques expulsados por prebúsquedas. Este mecanismo está basado

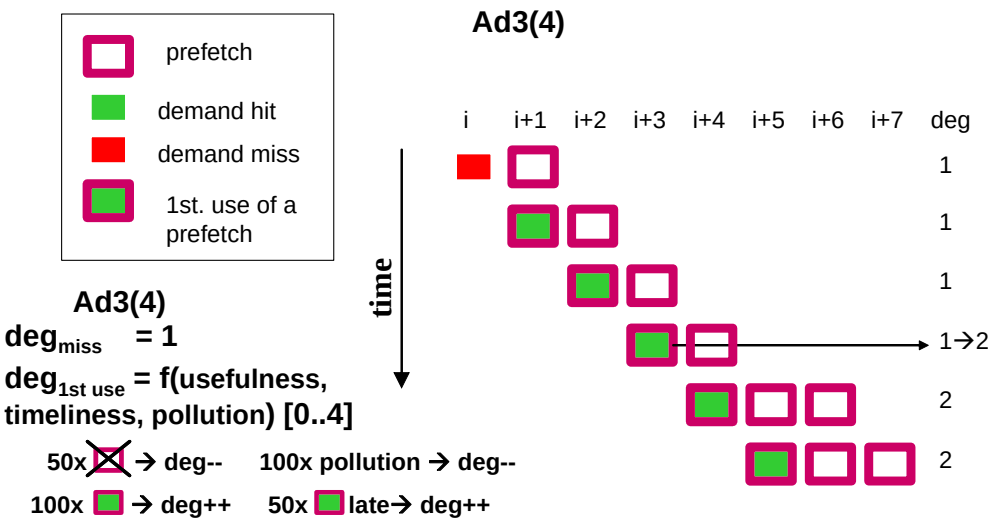

Figura 5.10 Comportamiento de la política de grado Ad3(4) cuando se produce en memoria el recorrido secuencial (*i*, *i*+1, *i*+2, *i*+3, *i*+4, *i*+5).

en [SMH+07]. De esta forma, cada vez que se acumulan cien eventos de polución se decrementa el grado.

La Figura 5.10 muestra un ejemplo de una política Ad3(4). El fallo en el bloque *i* genera la prebúsqueda de *i*+1. Como *deg*=1 los primeros usos de *i*+1, *i*+2 e *i*+3 prebuscan un bloque cada uno: *i*+2, *i*+3 e *i*+4 respectivamente. El primer uso del bloque *i*+3 ha sido un fallo secundario (la prebúsqueda no ha concluido aún cuando el bloque es accedido) que hace que su contador llegue a 50 y provoca que *deg* se incremente a 2. A partir de este momento cada primer uso genera dos prebúsquedas (*i*+4 genera *i*+5 e *i*+6; *i*+5 genera *i*+6 e *i*+7).

Como se observa en el ejemplo, esta política de grado también genera prebúsquedas redundantes. Por otro lado, cuando el recorrido seguido por el programa no es secuencial se genera una prebúsqueda inútil.

Prebúsqueda adaptativa por regiones, Ad4(*x*,*y*)

La siguiente política adaptativa que hemos definido, Ad4(*x*,*y*), aplica Ad2(*x*) a *y* regiones de memoria. En caso de fallo se prebuscan los bloques anterior y siguiente al accedido. En un primer uso de prebúsqueda se generan el número de bloques indicado por el registro *deg* de la región en la dirección indicada por el bit de dirección del bloque accedido. En esta política de grado adaptativa el espacio de direcciones de memoria es dividido en *y* regiones de igual tamaño. En cada una de ellas se dispone de los dos contadores, *usos* y *reemplazos*, y de

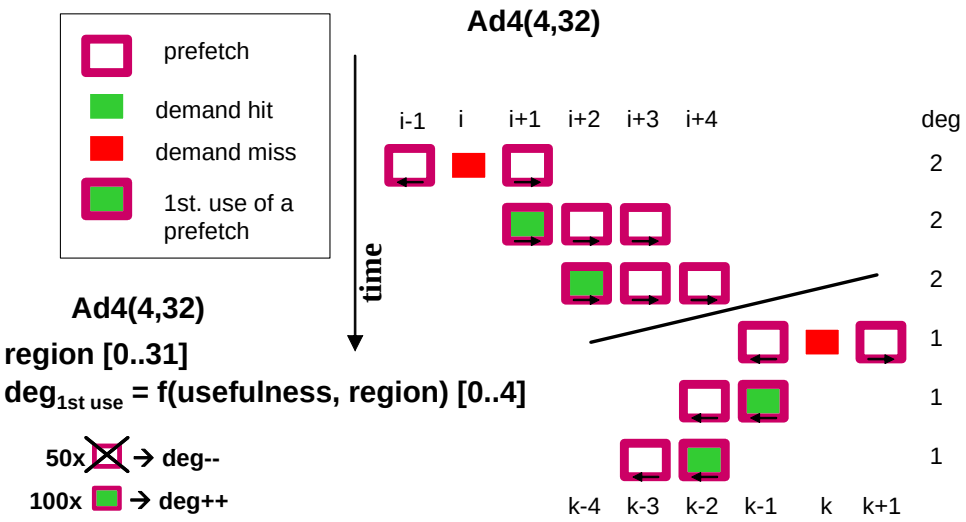

Figura 5.11 Comportamiento de la política de grado Ad4(4,32) cuando se produce en memoria los recorridos secuenciales (*i*, *i*+1, *i*+2, y *k*, *k*-1, *k*-2).

un registro *deg* que establece el número de bloques a prebuscar cuando se produce un primer uso de prebúsqueda en dicha región. Cuando los contadores de usos y reemplazos llegan a su nivel de saturación, se reinician e incrementan o decrementan el registro *deg* de la región correspondiente, tal como se ha explicado en la política Ad1(*x*).

En la figura se observa un ejemplo de una política Ad4(4,32). El acceso al bloque *i* produce un fallo y se prebuscan los bloques *i*+1 e *i*+2. El registro *deg* de la región vale 2, así que el acceso a *i*+1 prebusca a *i*+2 e *i*+3 y el acceso a *i*+2 prebusca los bloque *i*+3 e *i*+4. Posteriormente se produce un fallo en un bloque *k* perteneciente a otra región de memoria y que prebusca *k*+1 y *k*-1. El posterior acceso a *k*-1 provoca la prebúsqueda de un solo bloque (*k*-2), debido a que el registro *deg* de la nueva región vale 1. Igualmente el primer uso de *k*-2 prebusca el bloque de dirección *k*-3.

Cuando el registro *deg* de la región correspondiente es mayor que 1, esta política de grado también genera prebúsquedas redundantes. Al prebuscarse en ambos sentidos, un fallo provocado por un recorrido no secuencial puede generar dos prebúsquedas inútiles.

Prebúsqueda secuencial adaptativa de Dahlgren, Ad5(*x*)

La política de grado que denominaremos Ad5(*x*) no está definida por nosotros. La hemos implementado según [DDS93]. En esta política, el grado de

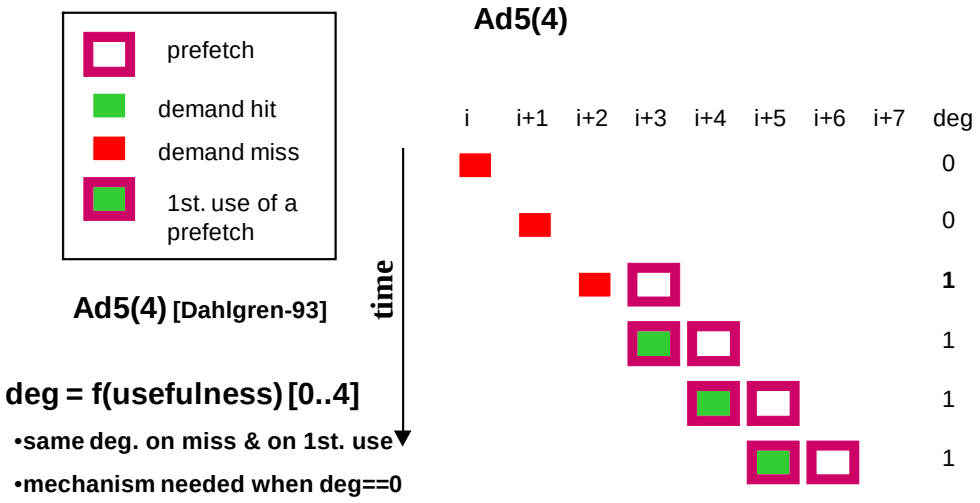

Figura 5.12 Comportamiento de la política de grado Ad5(4) cuando se produce en memoria el recorrido secuencial (*i*, *i*+1, *i*+2, *i*+3, *i*+4, *i*+5).

prebúsqueda se establece a través de la variable *deg*, tanto si se produce un fallo como en el caso de un primer uso de prebúsqueda. El grado de prebúsqueda puede variar en el rango 0-*x*, según la *utilidad* de las prebúsquedas previas. Para cuantificar la utilidad de la prebúsqueda, se utilizan dos contadores, uno de *prebúsquedas* y otro de *primeros usos*. Cada vez que el primero cuenta dieciséis prebúsquedas se mira el segundo contador (número de primeros usos), y en función de su valor se incrementa o decrementa *deg*. Esta política es la única de las que se presentan aquí en la que la prebúsqueda puede llegar a desactivarse totalmente (cuando *deg*=0). Por ello, necesita un mecanismo especial que detecte cuándo la prebúsqueda puede ser útil de nuevo para incrementar *deg* a 1.

En la Figura 5.12 se representa un ejemplo de una política Ad5(4) cuando el programa realiza un recorrido secuencial en memoria *i*, *i*+1, *i*+2,... *i*+5. Se observa cómo inicialmente la prebúsqueda está desactivada y se producen fallos en los accesos a los bloques *i* e *i*+1, sin generarse ninguna prebúsqueda. En este momento el mecanismo de activación de la prebúsqueda establece que ésta podría ser útil e incrementa *deg* a 1. A partir de este momento tanto fallos como primeros usos generarán una prebúsqueda. El fallo en *i*+2 genera la prebúsqueda de *i*+3, y los primeros usos de *i*+3, *i*+4 e *i*+5 generan, respectivamente, las prebúsquedas de los bloques *i*+4, *i*+5 e *i*+6.

Cuando se recorre un recorrido secuencial, esta política adaptativa genera *deg*-1 prebúsquedas redundantes. Cuando el recorrido no es secuencial, un acceso que falle en cache generan *deg* prebúsquedas inútiles.

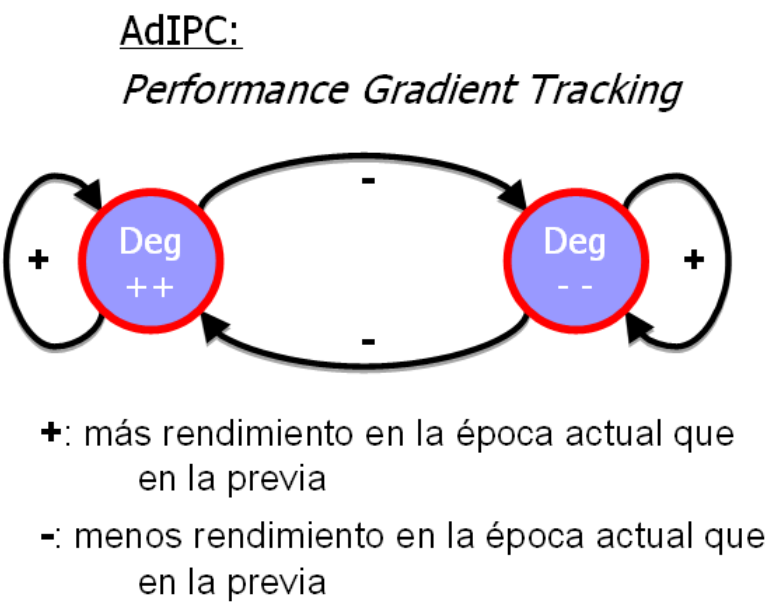

Figura 5.13 Autómata de estados de la política de grado AdIPC.

Prebúsqueda adaptativa basada en rendimiento, AdIPC

Finalmente, la última política de grado de agresividad adaptativa que se ha definido ha sido AdIPC. Todas las políticas adaptativas anteriores están basadas en realizar diversas métricas obre las prebúsquedas realizadas, como *utilidad*, *puntualidad* y *polución*. Cada vez que se produce un cierto número de eventos de cada una de ellas se incrementa o decrementa el grado de prebúsqueda, con la esperanza de que influya positivamente en la eficacia del sistema de prebúsqueda. La política de grado AdIPC sigue una estrategia diferente, midiendo directamente las instrucciones que se ejecutan cada cierto número de ciclos (época) y tomando una decisión sobre el grado de prebúsqueda que maximice el número de instrucciones ejecutadas en cada época. Para ello, un autómata de dos estados (Figura 5.13) guarda información sobre lo que se hizo al finalizar la época anterior (*aumentar grado* o *disminuir grado*). Si la época actual ha conseguido ejecutar más instrucciones que la anterior, la decisión sobre la modificación del grado fue acertada y se mantiene el estado. Si por el contrario se han ejecutado menos instrucciones que en la anterior, se asume que la modificación del grado fue equivocada y se toma la decisión contraria, cambiando el estado del autómata en consecuencia. Este mecanismo va ajustando dinámicamente el valor de la variable *deg*, que es la que determina el

grado de prebúsqueda en caso de fallo o primer uso, pudiendo tomar los siguiente valores [0, 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 64].

Se han evaluado distintos tamaños de época y 512 Kinstrucciones parece un tamaño razonable. Se ha valorado la posibilidad de añadir un mecanismo que ajuste el tamaño de la época según la aplicación ejecutada. Finalmente esta idea ha sido desestimada debido a su coste y al poco beneficio medio que obtendría.

Igual que las otras políticas adaptativas, esta también genera prebúsquedas redundantes (*deg*-1) cuando se va siguiendo un recorrido secuencial. En el caso de un recorrido no secuencial, cada acceso que falle en *cache* podría generan hasta *deg* prebúsquedas inútiles.

Para concluir esta sección se presenta aquí un resumen de las características principales de todas las políticas de grado definidas (ver Tabla 5.1). Para cada política se indican los bloques que se prebuscan en caso de fallo, los bloques que se prebuscan en caso de primer uso, el rango y la forma de calcular la variable *deg*, y finalmente el número de fallos obligatorios (*n. cold misses*), el número de prebúsquedas redundantes y el número de prebúsquedas inútiles que se producen en caso de un recorrido no secuencial.

Tabla 5.1 Características principales de las políticas de grado definidas.

Política	bloques prebusq. en fallo	bloques prebusq. en primer uso	cálculo deg	n. cold misses	n. prebusq. redundantes	n. prebusq. inútiles
Deg(x)	1, 2, ... x	1, 2, ... x		0	x-1	x
Dist(x)	x	x		x-1	0	1
Deg-dist(x)	1, 2, ... x	x		0	0	x
Deg(1-x)	1	1, 2, ... x		0	x-1	1
Ad1(x)	1	1, 2, ... deg	[0-x] f(utilidad)	0	deg-1	1
Ad2(x)	+1, -1	1, 2, ... deg ó -1, -2, ... -deg	[0-x] f(utilidad)	0	deg-1	2
Ad3(x)	1	1, 2, ... deg	[0-x] f(utilidad, polución, puntualidad)	0	deg-1	1
Ad4(x,y)	+1, -1	1, 2, ... deg ó -1, -2, ... -deg	[0-x] f(utilidad, región)	0	deg-1	2
Ad5(x)	1, 2, ... deg	1, 2, ... deg	[0-x] f(utilidad)	0	deg-1	deg
AdIPC	1, 2, ... deg	1, 2, ... deg	[0-x] f(ΔIPC)	0	deg-1	deg

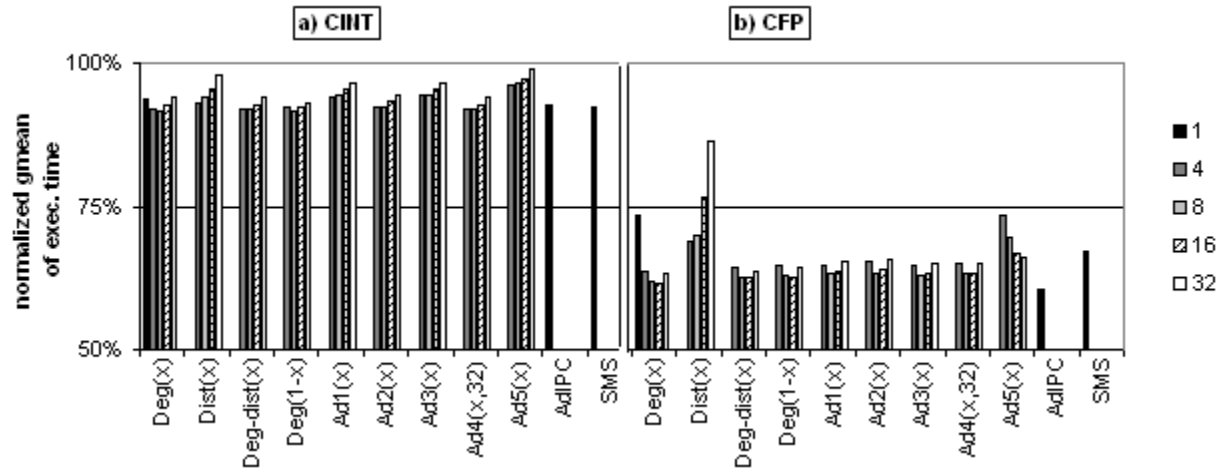

Figura 5.14 Media geométrica del tiempo de ejecución normalizado de un prebuscador *secuencial marcado* con las diferentes políticas de grado y de SMS.

5.4 Resultados

Para analizar los resultados obtenidos por las diferentes políticas de grado vamos a comparar todas ellas con el SMS. Este prebuscador selectivo obtuvo los mejores resultados en las pruebas preliminares, de modo que lo usaremos como referencia para evaluar nuestras propuestas. De todas las aplicaciones ejecutadas únicamente ammp y art presentan algunas pérdidas. Con los mecanismos adaptativos (todos excepto Ad5) estos programas siguen presentando alguna pequeña pérdida, aunque de cuantía similar a las obtenidas por el selectivo SMS. La descomposición por aplicaciones se muestra en el apéndice B.

En la Figura 5.14 se representan las medias geométricas de los tiempos de ejecución normalizados de todas las políticas de grado con respecto a un sistema sin prebúsqueda. La última barra corresponde al prebuscador selectivo SMS. Se observa que las mejores opciones para aplicaciones de enteros corresponden a los grados 4 y 8, mientras que para aplicaciones de coma flotante los grados 8 y 16 consiguen los mejores resultados. Por otro lado Ad5(*x*) y Dist(*x*) muestran el peor rendimiento, tanto en INT como en FP. El resto de políticas consiguen, en media, rendimiento similar a Deg(x) en los grados mencionados. Entre las políticas adaptativas, para los programas de enteros la mejor es Ad4(8,32), con pequeñas diferencias (menores que un 1%) respecto a otras adaptativas y a SMS. En los *benchmarks* de coma flotante SMS es vencido por todos los

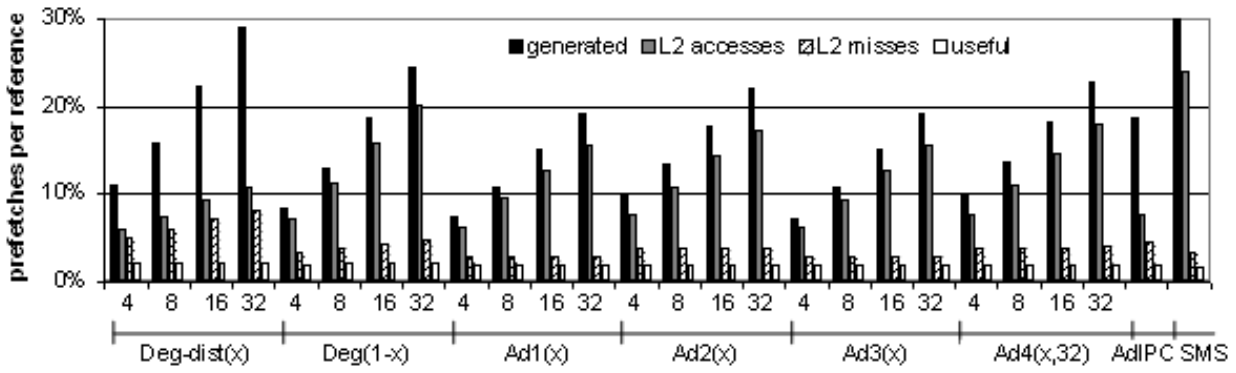

Figura 5.15 Porcentaje de prebúsquedas por referencia que, para cada política de grado, a) son generadas; b) acceden a L2; c) fallan en l2; d) son útiles.

adaptativos (excepto Ad5(*x*)). En estas aplicaciones la mejor política de grado es AdIPC, con diferencias de un 3% con otros adaptativos y de un 7% con respecto a SMS. En conjunto, las políticas de grado AdIPC, Ad4(8,32) y Ad2(8) parecen ser las mejores en media.

La Figura 5.15 muestra la presión ejercida por cada prebuscador y grado sobre la jerarquía de memoria. En cada grupo, la primera barra indica el número de prebúsquedas generadas por referencia a memoria, la segunda las prebúsquedas que acceden a memoria (L2), la siguiente las que fallan en el acceso y la última las que finalmente son útiles. Se puede observar cómo todas las políticas son menos agresivas que el SMS. Si observamos el porcentaje de prebúsquedas inútiles (la diferencia entre las dos últimas barras) dos de nuestras propuestas, Ad1(8) y Ad3(8) generan menos tráfico inútil que SMS. Sin embargo, otras tres, Ad2(8), Ad4(8,32) y AdIPC, generan más prebúsquedas inútiles que SMS.

5.5 El *Prefetch Address Buffer* como elemento de filtrado

Como se ha comprobado en el apartado anterior, a pesar de haber reducido su agresividad, la presión ejercida por los prebuscadores sobre la jerarquía de memoria es considerable. Además, la mayoría de políticas de grado usadas generan un gran número de prebúsquedas redundantes. En este apartado se pretende incrementar la capacidad de filtrado del *Prefetch Address Buffer* (PAB) sin que las prestaciones de los prebuscadores se vean afectadas.

En primer lugar analizamos el funcionamiento de un PAB normal y su capacidad natural de filtrado debida al hecho de que se hace *lookup* en el mismo antes de insertar cada nueva prebúsqueda. Supongamos un mecanismo de

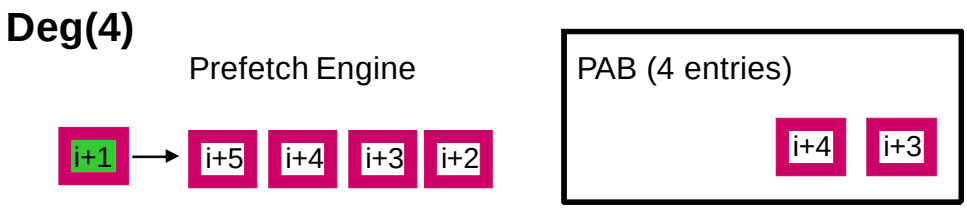

Figura 5.16 Capacidad de filtrado del *Prefetch Address Buffer* normal de las prebúsquedas redundantes de un prebuscador *secuencial marcado.*

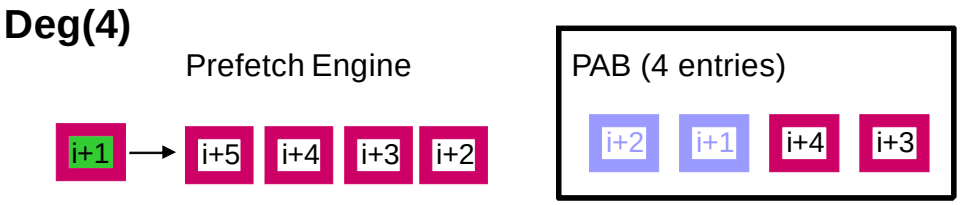

Figura 5.17 Capacidad de filtrado del *Prefetch Address Buffer* mejorado de las prebúsquedas redundantes de un prebuscador *secuencial marcado.*

prebúsqueda *secuencial marcado* de grado fijo 4, Deg(4), en el que ante un fallo en el acceso a un bloque *i* se generan prebúsquedas para los bloques *i*+1, *i*+2, *i*+3 e *i*+4. Digamos que las prebúsquedas a los bloques *i*+1 e *i*+2 son lanzadas desde el PAB y acaban haciendo el *refill* en la memoria *cache*. Cuando posteriormente se produce el primer uso del bloque *i*+1, se generan las prebúsquedas de los bloques *i*+2, *i*+3, *i*+4 e *i*+5. De éstas, *i*+3 e *i*+4 se filtran, ya que aún permanecen en el PAB (ver Figura 5.16). Vemos que sin embargo *i*+2 ha vuelto a ser generada y volverá a acceder a L2, donde producirá un acierto.

La idea que proponemos es mantener las prebúsquedas ya lanzadas en las entradas libres del PAB, de forma que puedan filtrarse. De esta forma se consigue un PAB con mayor capacidad para filtrar prebúsquedas redundantes. Veamos el mismo ejemplo que antes, pero utilizando ahora el PAB propuesto. De nuevo, ante el acceso que falla en el bloque *i* se generan las prebúsquedas *i*+1, *i*+2, *i*+3 e *i*+4. Supongamos otra vez que *i*+1 e *i*+2 son lanzadas y finalizan su acceso haciendo el *refill* de los bloques en L2. En el PAB permanecen *i*+3 e *i*+4, pero en esta ocasión mantenemos *i*+1 e *i*+2 en las dos entradas “libres”. Cuando se produce el primer uso de *i*+1, se vuelven a generar las prebúsquedas a *i*+2, *i*+3, *i*+4 e *i*+5. Pero ahora (ver Figura 5.17) el *lookup* que se realiza en el PAB al insertarlas hace que también se filtre *i*+2. Nótese que al insertar *i*+5 se reclamará la entrada “libre” ocupada por *i*+1.

Es importante recalcar que esta estrategia reduce el número de accesos por prebúsqueda a L2 sin merma de rendimiento en ninguno de los prebuscadores. El porcentaje de reducción de accesos es diferente según el prebuscador: 2% para Deg-dist(*x*), y entre 25%-40% para el resto de nuestras propuestas. Aplicando esta modificación el SMS reduce en un 49% los accesos por prebúsqueda a L2, pero continúa siendo el prebuscador que más presión genera.

5.6 Conclusiones

En este capítulo se han presentado distintas formas de ajustar la agresividad del prebuscador *secuencial marcado* de forma que consigue funcionar de forma similar o mejor que uno de los mejores prebuscadores conocidos hasta el momento, el SMS.

De todas las técnicas que proponemos, las que se comportan mejor son las que adaptan la agresividad del prebuscador basándose en la utilidad de las prebúsquedas realizadas previamente (Ad2 y Ad4) o en la variación del IPC (AdIPC). Todos estos métodos igualan las prestaciones obtenidas por el prebuscador SMS en aplicaciones enteras, mientras que obtienen más prestaciones en coma flotante con un 60% menos de accesos en L2.

Además los *costes hardware* de estas propuestas son mínimos. Ad2 necesita únicamente un bit extra por bloque de *cache*. El prebuscador adaptativo Ad4 propuesto (Ad4(8,32)) necesita adicionalmente una tabla de 64 Bytes, muy inferior a la tabla de 32 KB necesitada por el prebuscador SMS. Por su parte, AdIPC sólo requiere de dos contadores.

También se ha comprobado que el propio *Prefetch Address Buffer* (PAB) puede servir como elemento de filtrado y ayudar a reducir la presión generada por el prebuscador en L2 en un 30%.

Finalmente, se ha comprobado que todas las propuestas presentadas en este capítulo son opciones razonables, que no presentan pérdidas y tienen un *coste hardware* muy pequeño.

CAPÍTULO 6

Prebúsqueda multinivel adaptativa

El éxito de un sistema de prebúsqueda depende tanto del diseño de la cache como de las características concretas de la aplicación, por lo que el mejor prebuscador en unos casos puede no serlo en otros. En este capítulo se presenta un esquema general de prebúsqueda en dos niveles que puede adaptarse a diferentes objetivos de diseño. Para demostrar su utilidad, se evalúan tres configuraciones dirigidas a tres objetivos distintos (mínimo coste, mínimas pérdidas y máximas prestaciones). Además, se incorporan nuevos métodos de filtrado y adaptación. La evaluación se ha realizado sobre el entorno determinado por el First JILP Data Prefetching Championship (DPC-1), en el que el trabajo obtuvo el Best Paper Award.

6.1 Introducción

El rendimiento de un sistema de prebúsqueda concreto depende tanto de las aplicaciones como de la jerarquía de memoria del procesador. Hasta el momento no ha habido ningún sistema de prebúsqueda que funcione óptimamente para todas las aplicaciones. Por un lado, los prebuscadores agresivos, como *secuencial marcado* o *stream buffers*, consiguen un rendimiento medio muy bueno pero pueden provocar una gran presión sobre el sistema de memoria e incluso pérdidas de prestaciones en algunas aplicaciones concretas. Para reducir estas pérdidas es habitual que estos prebuscadores usen métodos de filtrado,

pero estos provocan una bajada del rendimiento medio y además su coste *hardware* es importante. También es habitual el uso de mecanismos adaptativos para ajustar dinámicamente la agresividad de estos prebuscadores.

Por otro lado están los prebuscadores de correlación, por ejemplo PC/DC (sección 1.3.4) o PDFCM (sección 4.2.5). Son mucho más selectivos, ya que usan tablas donde guardan información sobre el comportamiento del programa en memoria con el objetivo de predecir futuros accesos. Sin embargo algunos de ellos necesitan tablas de gran tamaño y otros necesitan un gran número de accesos a las tablas.

A la hora de diseñar un mecanismo de prebúsqueda *hardware* de datos hay, al menos, tres factores importantes: 1) la capacidad de almacenamiento necesaria para implementar dicho mecanismo de prebúsqueda (o *coste hardware*); 2) la presencia de pérdidas en alguna aplicación; 3) el rendimiento medio obtenido en el conjunto de aplicaciones. Entre ellos existe una fuerte relación. Por ejemplo, si disminuimos mucho el *coste hardware* bajará el rendimiento medio. Si moderamos la agresividad del prebuscador para conseguir reducir pérdidas en algunas aplicaciones, también bajará el rendimiento medio. Si conseguimos aumentar el rendimiento medio volverán a aparecer pérdidas en algunas aplicaciones.

Por lo tanto, parece claro que según el objetivo buscado las decisiones de diseño han de ser diferentes.

La siguiente sección de este capítulo (6.2) presenta la propuesta de esquema general de sistema de prebúsqueda adaptable a distintos objetivos. En la sección 6.3 se detalla la infraestructura de simulación en la que se evalúan las propuestas (provista por el DPC-1), así como las propuestas del resto de participantes y los resultados obtenidos por nuestra contribución. Finalmente, la sección 6.4 presentan las conclusiones.

6.2 Esquema general de prebúsqueda adaptable a objetivos

La Figura 6.1 muestra la estructura de nuestra propuesta de esquema general de prebúsqueda adaptable a distintos objetivos. En cada nivel de *cache* (L1 y L2) se dispone de los siguientes componentes:

- un *motor de prebúsquedas*, diferente dependiendo del nivel de *cache* y del objetivo perseguido;

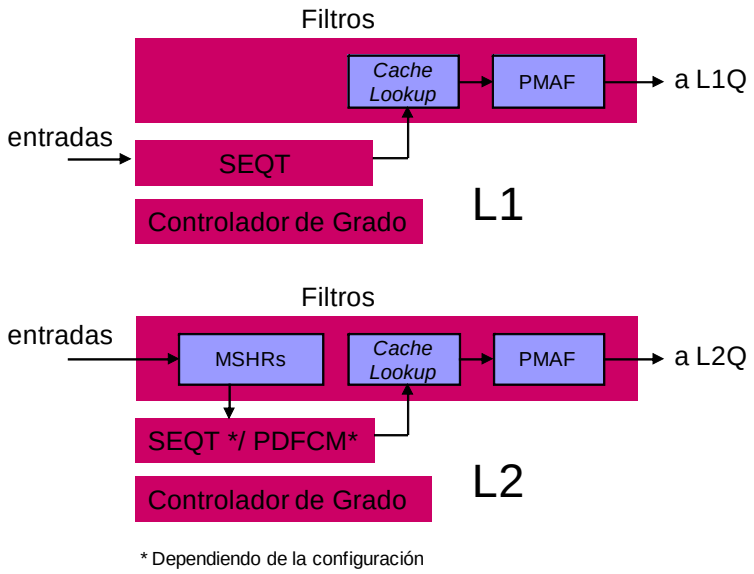

Figura 6.1 Propuesta de esquema general de prebúsqueda adaptable a objetivos.

- cada *motor de prebúsquedas* incluye un *autómata de grado*, capaz de generar una prebúsqueda por ciclo, a partir de la información almacenada;
- un *controlador de grado* que adapta la agresividad del *motor de prebúsquedas*;
- y varios filtros (*Cache Lookup* y *PMAF* en L1; *MSHR*, *Cache Lookup* y *PMAF* en L2).

A continuación se describen detalladamente cada uno de los componentes.

Motor de prebúsqueda *secuencial marcado* (SEQT)

El *motor de prebúsqueda secuencial marcado* prebusca el bloque siguiente al referenciado cuando se produce un fallo, o cuando un bloque prebuscado anteriormente se referencia por primera vez (ver sección 1.3.2). Dentro del esquema general que hemos definido, este *motor de prebúsqueda* se puede colocar tanto en L1 como en L2. En este último caso se tienen en cuenta las referencias de L1 generadas tanto por demanda como por prebúsqueda (siempre que fallen en L2 o usen un bloque prebuscado por primera vez). Se tienen en cuenta todas las instrucciones de acceso a memoria, *load* y *store*.

Cada *motor de prebúsqueda* SEQT incluye un *autómata de grado* (Figura 6.2) que va incrementando la dirección del bloque a prebuscar hasta el grado

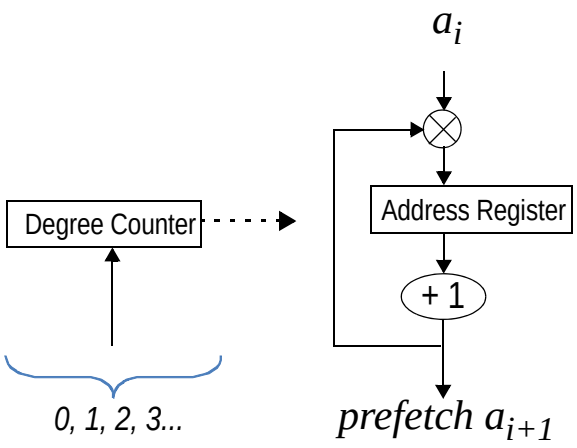

Figura 6.2 *Autómata de grado* del *motor de prebúsqueda secuencial marcado* (SEQT).

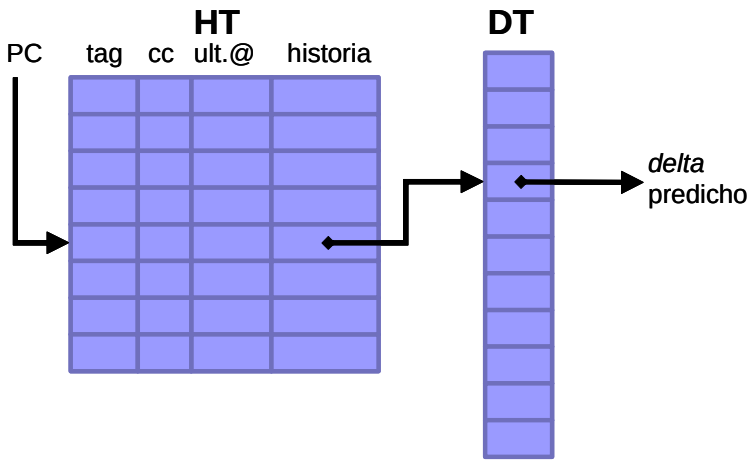

Figura 6.3 Tablas del *motor de prebúsqueda* PDFCM.

indicado por el *controlador de grado*, generando de esta forma una prebúsqueda por ciclo.

Motor de prebúsqueda PDFCM

El *motor de prebúsqueda* PDFCM implementa el mecanismo de prebúsqueda de correlación de *deltas* PDFCM, que ya ha sido descrito en la sección 4.2.5. En este caso, y para cumplir las condiciones de participación en el *DPC-1* (máximo 32 Kbits de información) las tablas han sido redimensionadas: la *History Table* consta de 256 entradas y la *Delta Table* de 512 entradas (ver Figura 6.3). Este *motor de prebúsqueda* se alimenta con los fallos de L2 y los primeros usos de

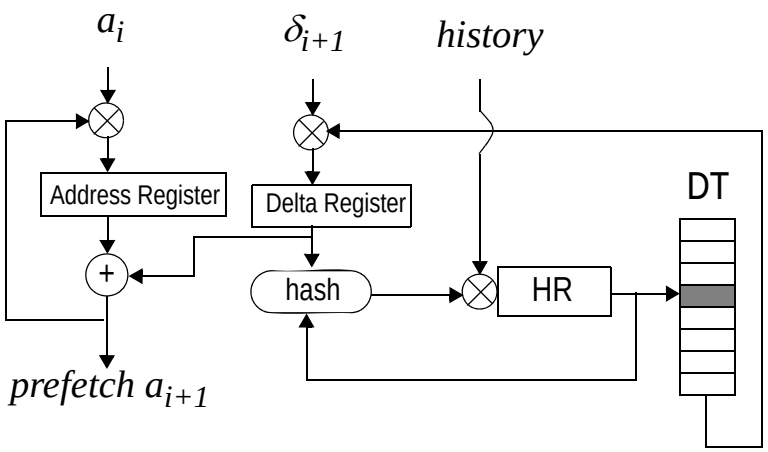

Figura 6.4 *Autómata de grado* del *motor de prebúsqueda* PDFCM.

los bloques prebuscados en L2. Su operación consta de 3 fases: *actualización*, *predicción* y *autómata de grado.* Las dos primeras fueron descritas al definir este prebuscador en el Capítulo 4. La gestión del grado varía respecto al prebuscador original, porque en el esquema que proponemos aquí el grado depende de los dos controladores que se describirán en breve. En conjunto, el funcionamiento ahora queda como sigue:

- En la fase de *actualización* se accede a la HT usando el PC de la instrucción para conseguir su historia de *deltas*. Con esta historia se indexa la DT para actualizar la tabla con el *delta* actual.

- En la segunda fase (*predicción*) se incorpora el *delta* actual a la historia, obteniendo así una nueva historia. Esta se usa para indexar la DT y predecir un nuevo *delta* que permitirá determinar la dirección a prebuscar.

- En la última fase, el *autómata de grado* (Figura 6.4) incorpora el *delta* predicho a la historia obteniendo en este caso una historia especulativa, con la que se accede a la DT para predecir un nuevo *delta*. Este *delta* se suma a la última dirección para conseguir la nueva dirección a prebuscar. El autómata realiza esta actividad cada ciclo, hasta el grado indicado por el *controlador de grado* correspondiente.

Controlador de grado en L1

El componente *controlador de grado* decide en cada momento cuál es el grado de prebúsqueda máximo del *motor de prebúsquedas* correspondiente. En el caso

del primer nivel de cache su *controlador de grado* es muy simple, ya que implementa una política de grado estática llamada *Grado(1-4)*. Esta política consiste en prebuscar con grado 1 cuando se ha producido un fallo y prebuscar con grado 4 en caso de un primer uso de prebúsqueda. Para más detalle sobre esta política de grado consultar la sección 5.3.

Controlador de grado en L2

El *controlador de grado* del segundo nivel de cache es un poco más complejo. En este caso el grado máximo del *motor de prebúsqueda* se ajusta dinámicamente realizando un seguimiento del gradiente de las prestaciones (*Performance Gradient Tracking*).

Este controlador puede estar en 2 estados (*incrementando grado* y *decrementando grado*). La idea consiste en mantener el estado mientras ello ayude a las prestaciones. Así, al final de cada época (64 Kciclos): 1) se comparan la prestaciones de la época actual con la de la época anterior; 2) si la actual tiene más prestaciones se mantiene el estado, cambiándolo en caso contrario; y 3) se incrementa o decrementa el grado según el nuevo estado. Esta *política de grado* se describe en detalle en la sección “Prebúsqueda adaptativa basada en rendimiento, AdIPC” en la página 70.

Filtros

En el entorno de simulación facilitado por el DPC-1 no se incluye ningún mecanismo para filtrar los fallos secundarios. Éstos pueden afectar en gran medida al aprendizaje de nuestro motor de prebúsqueda PDFCM. Por eso incluimos 16 MSHRs en la *cache* de segundo nivel para filtrar los fallos secundarios de la secuencia de referencias. En el primer nivel no incluimos MSHRs por las restricciones de espacio impuestas por el DPC-1, y por que afecta en menor medida, al no incluirse un prebuscador PDFCM en el primer nivel.

El segundo componente de filtrado que incluimos en el esquema general es el *Cache Lookup*. Este componente elimina las prebúsquedas generadas que ya están disponibles en la *cache*.

Finalmente, el componente *PMAF* es una estructura *FIFO* (similar a los MSHRs) que mantiene hasta 32 direcciones de los bloques prebuscados que ya han sido solicitados pero aún no han sido servidos.

6.2.1 Tres objetivos, tres configuraciones

Una vez que han sido definidos todos los componentes que se van a utilizar, definimos las tres configuraciones diferentes en función de los tres objetivos buscados: minimizar el coste *hardware* (*Mincost*), minimizar la presencia de pérdidas en todas las aplicaciones (*Minloss*) y maximizar las prestaciones (*Maxperf*).

Todas las configuraciones comparten:

- los filtros que acaban de describirse;
- la política de grado adaptativa por seguimiento del gradiente de las prestaciones en L2;
- un *motor de prebúsqueda* SEQT con política de grado estática Grado(1-4).

La tres configuraciones se diferencian únicamente en los *motores de prebúsqueda* utilizados en L2:

- en *Mincost* se utiliza un *motor de prebúsqueda* SEQT;
- en *Minloss* un *motor de prebúsqueda* PDFCM;
- en *Maxperf* se utilizan ambos *motores de prebúsqueda* simultaneamente SEQT y PDFCM, compartiendo ambos el mismo *controlador de grado*.

El dimensionado de las tablas del *motor de prebúsqueda* PDFCM y de los componentes de filtrado ha sido establecido atendiendo a los límites de costes *hardware* definidos por la competición (capacidad de almacenamiento de información necesario). Los requerimientos de almacenamiento finales de las tres configuraciones son: *Mincost* 1255 bits, *Minloss* 20784 bits y *Maxperf* 20822 bits.

La evaluación de estas tres configuraciones ha sido realizada en el entorno del *First JILP Data Prefetching Championship DPC-1*, que se presenta a continuación.

6.3 First JILP Data Prefetching Championship

En esta sección se describe la infraestructura de simulación del *First JILP Data Prefetching Championship*, las propuestas enviadas por otros participantes y los resultados del campeonato.

El objetivo de esta competición es comparar distintos algoritmos de prebúsqueda en un marco común. Los participantes deben proponen prebuscadores para ambos niveles de *cache* L1 y L2, sin exceder de la capacidad de almacenamiento definida por las reglas de la competición. Las propuestas se evalúan según las prestaciones obtenidas sobre un conjunto de programas de prueba (*benchmarks*) utilizando el marco de simulación propuesto.

Para participar, cada participante puede enviar un máximo de 3 contribuciones. Cada contribución consta de un resumen (*abstract*), un artículo y un fichero *C++ header* con el código fuente comentado.

6.3.1 Reglas de la competición

Cada participante debe diseñar y evaluar sus algoritmos en el marco de simulación propuesto. El marco de simulación se facilita en binario y no puede ser modificado, excepto el fichero *C++ header* donde se introduce el algoritmo de prebúsqueda. Las contribuciones tienen que ser recompiladas y ejecutadas con la versión original del marco de simulación. Cada propuesta es ejecutada con 3 configuraciones por la organización del DPC-1. En cada configuración se calcula la media geométrica de los *speed-ups* obtenidos por un conjunto de programas de prueba (*benchmarks*) no distribuidos. La puntuación final es la suma de las puntuaciones de cada configuración.

El campeón en prestaciones recibirá un trofeo. Además un número seleccionado de artículos se publicarán en un número especial del *Journal of Instruction-Level Parallelism (JILP)*. Por otro lado, el comité de programa seleccionará el mejor artículo técnico (*Best Paper Award*). En la página *web* del *DPC-1* se publicarán los códigos fuente y los resultados de todas las contribuciones.

6.3.2 Descripción del marco de simulación

El marco de simulación facilitado se basa en el simulador CMP$im. Se modela un procesador fuera-de-orden (*out-of-order*) con los siguientes parámetros:

- Ventana de instrucciones de 128 entradas sin restricciones de planificación (es decir, se puede ejecutar cualquier instrucción preparada de la ventana).

- Procesador *superescalar* de grado 4 segmentado en 15 etapas. En cada ciclo pueden lanzarse un máximo de 2 *loads* y 1 *store*.

- Predicción perfecta de saltos.

- Todas las instrucciones tienen latencia 1 excepto los fallos de *cache*. Los accesos que fallan en L1 y aciertan en L2 tienen una latencia de 20 ciclos. La latencia de acceso a memoria es de 200 ciclos. Por lo tanto, la máxima latencia de un acceso es de 220 ciclos.

- El modelo de memoria de datos consiste en una jerarquía *cache* de 2 niveles (L1 + L2). La *cache* de datos de primer nivel (L1) es de 32 KB 8-asociativa con reemplazo LRU. La *cache* de datos de segundo nivel (L2) es 16-asociativa con reemplazo LRU, y su tamaño es diferente según la configuración (512 KB - 2 MB).

- Cada *cache* dispone de una cola donde se almacenan las peticiones pendientes de acceso al siguiente nivel (tanto demandas como prebúsquedas) en orden *FIFO* (*First Input First Output*).

- En dichas colas no se realiza gestión de prioridad entre prebúsquedas y demandas. Si en el mismo ciclo se generan peticiones por prebúsqueda y por demanda son estas últimas las que se insertan primero en la cola. El ancho de banda con el que estas colas acceden puede ser limitado o infinito (según la configuración ejecutada).

- Los bloques de *cache* se prebuscan completos.

- Cuando un bloque llega a la *cache* reemplaza al bloque LRU, tanto haya sido una petición por demanda o por prebúsqueda.

- No se usan direcciones físicas.

- No tiene MSHR (*Miss-Status-Holding-Register*), por lo que no se filtran peticiones a bloques que ya están en tránsito y vuelven a ser enviadas.

- Las tres configuraciones del procesador que se ejecutarán son:

  - **Configuración 1**: tamaño de *cache* L2 2MB. Colas de peticiones pendientes con ancho de banda "casi infinito" (hasta 1000 peticiones por ciclo).
  - **Configuración 2**: tamaño de *cache* L2 2MB. Colas de peticiones pendientes con ancho de banda limitado (1 petición por ciclo a L2; 1 petición cada 10 ciclos a memoria principal).
  - **Configuración 3**: tamaño de *cache* L2 256 KB. Colas de peticiones pendientes con ancho de banda limitado (igual que en la configuración 2).

- El marco de simulación se distribuirá como una librería junto con un fichero *C++ header* donde el participante deberá añadir el código de su prebuscador.

- En cada ciclo el código del prebuscador podrá acceder a una estructura de datos que contiene información sobre los últimos accesos a L1: ciclo, dirección virtual de la instrucción que accede (*PC*), identificador de petición, tipo de instrucción (*load* o *store*), dirección virtual y respuesta de *cache* L1 (*hit/miss*). De igual forma podrá acceder a una estructura con información relativa al último acceso a L2 (identificador de la petición, dirección virtual, respuesta de *cache* L2 (*hit/miss*).

- Además cada bloque de *cache* está marcado con un bit que indica si ha sido traído por prebúsqueda. Se dispone de varias funciones para acceder y modificar dicho bit y para comprobar si un bloque está o no disponible en cada *cache* (*look-up*). Estas funciones pueden ser usadas de forma ilimitada.

- Cada propuesta de prebúsqueda puede utilizar un máximo de 32 Kbits de información, incluyendo todo el estado necesario para los prebuscadores de L1 y de L2. Sin embargo, la complejidad lógica del algoritmo utilizado no está limitada.

En nuestra opinión el marco de simulación establecido tiene algunas desventajas:

- no ofrece información sobre otras instrucciones, por ejemplo, cuándo las instrucciones atraviesan la etapa commit;

- no ofrece información sobre ocupación de colas;

- no dispone de MSHRs para filtrar misses secundarios;

- se limita la capacidad de almacenamiento, pero no la complejidad del algoritmo de prebúsqueda, lo que ha permitido que algunas propuestas sean de una complejidad algorítmica extrema.

6.3.3 Propuestas del resto de participantes

En esta sección se analizan las propuestas enviadas a la competición por el resto de participantes.

Dimitrov y Zhou [DiZ09] presentan un diseño que analiza en ambos niveles de *cache* las secuencias de fallos locales y la secuencia de fallos global. El objetivo es identificar localidades conocidas, como localidad *stride* y localidad contextual, y otras nuevas como *Global Stride* y *Most Common Stride*. También centran su trabajo en eliminar prebúsquedas redundantes e incluyen un mecanismo que desconecta el prebuscador cuando ya no es beneficioso. La idea de este mecanismo es no prebuscar un subconjunto de bloques de la cache y comparar el *miss ratio* obtenido con y sin prebúsqueda. Quizás lo más original de esta propuesta en comparación con el resto es que envía los bloques prebuscados a L1.

Ferdman *et al.* [FSF09] describen su prebuscador ya conocido, *Spatial Memory Streaming* (SMS) y le añaden un nuevo mecanismo que denominan *Pattern Bit-Vector Rotation* que les permite reducir sus amplios requerimientos de almacenamiento.

Por su parte, Grannaes *et al.*[GJN09] proponen una mejora de PC/DC que lo hace más eficiente en cuanto a número de accesos y requerimiento de almacenamiento. Sin embargo, su mecanismo trabaja únicamente en L2, de ahí su menor rendimiento en relación a propuestas que también consideran L1.

La propuesta de Isii *et al.* [IIH09] subraya que la ejecución fuera de orden y la ejecución especulativa afectan negativamente a la prebúsqueda. Con este motivo, definen un algoritmo de reconocimiento de patrones de acceso a los bloques utilizados que es inmune a dichas optimizaciones. La viabilidad de este método es discutible. Este algoritmo consta de los siguientes pasos: 1) detecta *hot zones*; 2) guarda información sobre el acceso a los bloques de la *hot zone*; 3) busca patrones de acceso en dicha información para generar candidatos a ser prebuscados; y 4) selecciona las prebúsquedas a realizar de entre los candidatos. Además en L1 incluyen un predictor de longitudes de secuencias que decide cuando la secuencia deja de prebuscarse [HuL06]

Liu *et al.* [LHP+09] evalúan distintas técnicas de mejora para optimizar un prebuscador de secuencias (*stream prefetcher*): 1) detector de *stride*; 2) eliminación de ruido de reconocimiento de *streams* al acceder a una estructura de árbol o grafo cuyos nodos ocupan más de un bloque; 3) relanzamiento de *streams* repetitivos por el acceso a vectores; y 4) eliminación de *streams* inactivos (*Dead Streams*). Se centran únicamente en L2.

Sharif y Lee [ShL09] extienden la idea del *Global History Buffer* (GHB) añadiendo *Local History Buffers*, que permiten guardar la información del acceso de memoria realizado por PCs específicos.

Finalmente, Verma *et al.* [VKP09] proponen un prebuscador híbrido secuencial-*stride* que adapta dinámicamente su agresividad (grado y distancia de prebúsqueda) basándose en diversas métricas, como la precisión y la puntualidad de las prebúsquedas.

6.3.4 Resultados de la competición

Todas las propuestas de los participantes han sido recogidas por los organizadores del DPC-1, que las han ejecutado en el marco de simulación descrito en la sección 6.3.2. Se ha ejecutado SPEC CPU 2006, compilados con icc, con un periodo de calentamiento inicial de 40.000 millones de instrucciones. Posteriormente se han ejecutado 100 millones de instrucciones durante las cuales se ha medido las prestaciones del sistema de prebúsqueda. Todos los resultados han quedado recogidos en la página web del DPC-1[1].

La Figura 6.5 muestra los *speed-ups* con respecto a un sistema sin prebúsqueda obtenidos para cada aplicación por nuestras tres propuestas, ejecutadas en la configuración 3 del DPC-1. El último grupo de columnas representa la media geométrica. Tal y como se esperaba *Mincost* obtiene el peor rendimiento medio, mientras que *Maxperf* obtiene el mejor. También se aprecia que de las 12 aplicaciones cuyo *global miss ratio* es menor de 0,2%, la prebúsqueda no obtiene apenas ningún resultado, con la excepción de h264ref y tonto. Finalmente se observa que *Minloss* muestra sólo pérdidas en astar y povray, pero son realmente despreciables.

1. http://www.jilp.org/dpc/online/DPC-1%20Program.htm

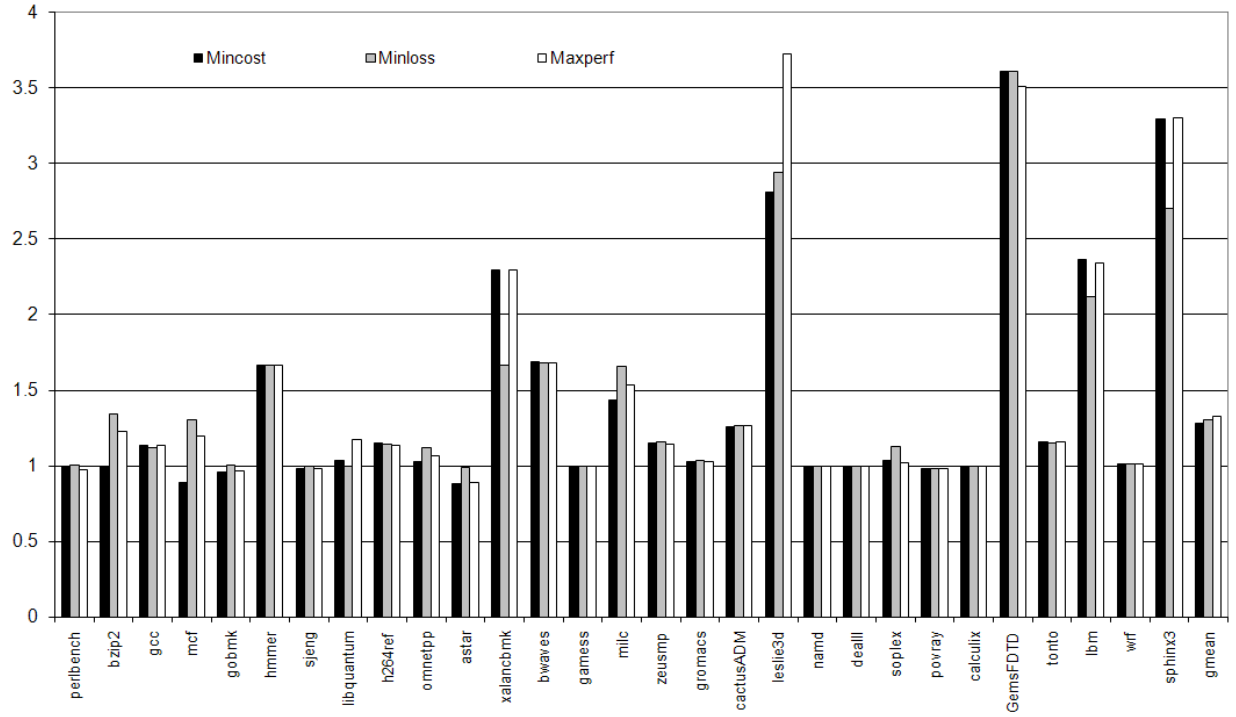

Figura 6.5 *Speed-ups* por aplicación obtenidos por las tres configuraciones.

Aparte de los resultados obtenidos por la organización del DCP-1, hemos evaluado por nuestra cuenta la eficiencia conseguida por la política de grado dinámica implementada por el *controlador de grado* de L2. Así, en la Figura 6.6 se muestran los *speed-ups* obtenidos por prebuscadores similares, pero usando un grado de prebúsqueda fijo de 1, 4, 16 y 64 en L2. El prebuscador que usa el *controlador de grado* con mecanismo adaptativo obtiene en media los rendimientos más altos. El grado fijo óptimo depende de la aplicación. Por ejemplo, 1 en astar, 4 en gemmsFDTD y 16 en leslie3d. En conjunto, la política de grado adaptativa obtiene prestaciones similares que el mejor grado fijo.

Todas las propuestas de los participantes han sido ejecutadas en las 3 configuraciones. En cada configuración se ha calculado la media geométrica de los *speed-ups* sobre un sistema sin prebúsqueda obtenidos por el conjunto de programas de prueba. La puntuación final se ha calculado sumando las puntuaciones de cada configuración (Figura 6.7).

En la Figura 6.8 se puede ver la descomposición de los resultados según las 3 configuraciones para los 3 primeros clasificados. Se puede observar cómo

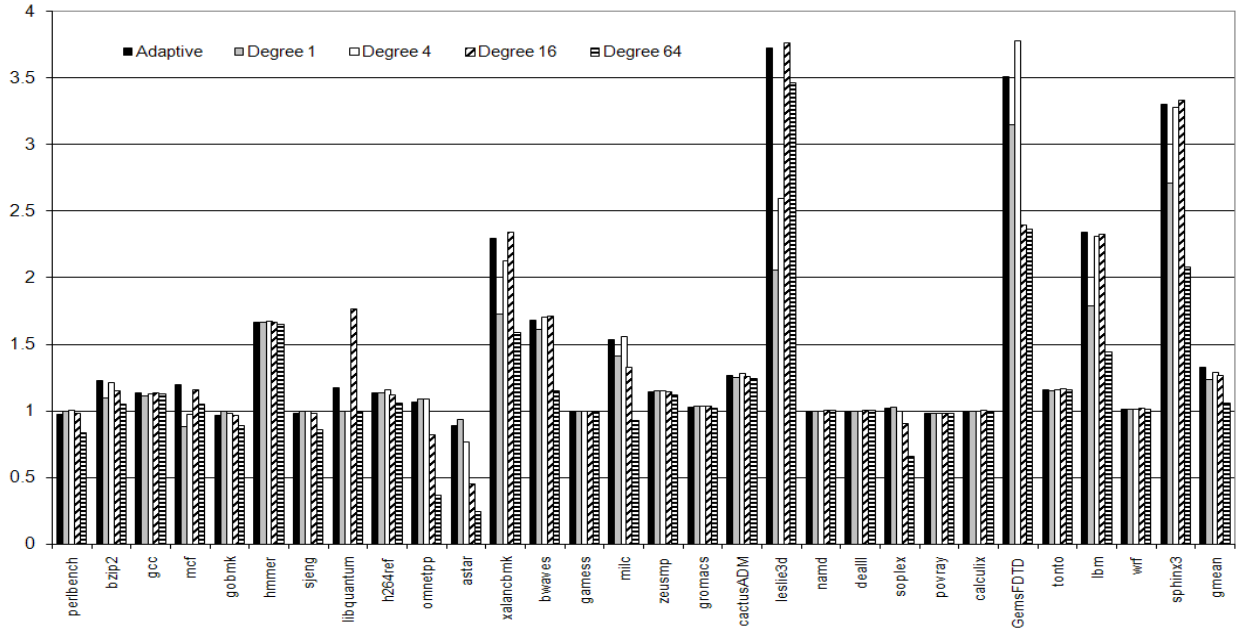

Figura 6.6 *Speed-ups* obtenidos por prebuscadores de grado adaptativo y grado fijo 1, 4, 16 y 64.

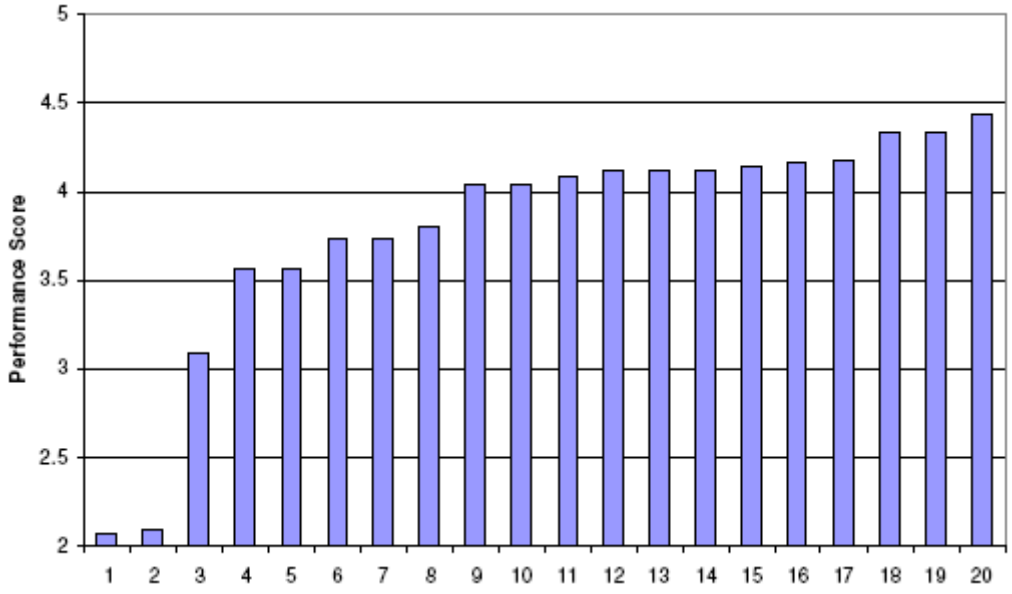

Figura 6.7 Puntuación final obtenida por todos los participantes.

nuestra propuesta *Maxperf* consigue ganar en la configuración 1, mientras que la de Isii *et al.* gana en las configuraciones 2 y 3.

Rank	Score	Config1 G. Mean	Config2 G. Mean	Config3 G. Mean
1	*4.437*	1.468	*1.471*	*1.498*
2	4.335	1.487	1.438	1.410
3	4.332	*1.509*	1.406	1.417

Figura 6.8 Resultados obtenidos por los tres primeros clasificados en las tres configuraciones.

6.4 Conclusiones

En este capítulo se ha propuesto un esquema general de prebúsqueda que permite ser adaptado a distintos objetivos. Se han definido tres configuraciones diferentes de dicho esquema para tres objetivos diferentes: minimizar el coste *hardware*, minimizar la presencia de pérdidas en todas las aplicaciones y maximizar las prestaciones.

La evaluación de nuestras configuraciones se ha efectuado en el marco del *Primer Campeonato de Prebuscadores de Datos (First JILP Data Prefetching Championship)*, Este campeonato ha sido la mejor oportunidad para comparar nuestro trabajo con el del resto de grupos de investigación, ya que todos los participantes han usado la misma infraestructura de simulación. Esto ha permitido comparar todas las propuestas en igualdad de condiciones.

Nuestra participación obtuvo el premio al mejor artículo (*Best Paper Award*) y el tercer puesto en prestaciones, en un empate técnico con el segundo clasificado. Además la viabilidad del método propuesto por el primer clasificado es muy discutible, frente a la simplicidad de nuestras configuraciones.

Finalmente, todos los códigos fuente de las propuestas están a disposición de la comunidad científica a través de la página web del DPC-1, lo que permite que cualquier grupo de investigación pueda analizar detalladamente todos los métodos.

CAPÍTULO 7

Conclusiones y líneas abiertas

7.1 Conclusiones

La latencia de las referencias a memoria es uno de los factores clave que limitan el rendimiento de un procesador. Este trabajo se ha centrado en el diseño de mecanismos de prebúsqueda *hardware* de datos que oculten la latencia de acceso a memoria de forma eficiente. Se ha comenzado con un estudio de patrones de acceso conocidos que ha llevado a la definición de uno nuevo (el *linear link*). A continuación se han presentado nuevos mecanismos de prebúsqueda (uno basado en *linear link* y otro de correlación), y se han estudiado nuevos mecanismos de control de prebuscadores agresivos. Finalmente se ha definido un esquema general para el diseño de sistemas de prebúsqueda orientados a objetivos concretos. Utilizando elementos de las propuestas de los capítulos anteriores, la competitividad de este esquema frente a otras aproximaciones ha podido demostrarse con éxito en una plataforma de comparación ajena y neutral.

7.1.1 Definición de un nuevo patrón de acceso a memoria

Se ha definido un nuevo modelo de regularidad (el *linear link*) que relaciona la dirección de un *load* con los valores leídos por otros *loads* a través de funciones lineales.

Este nuevo modelo representa aproximadamente un tercio de todas las referencias de memoria. La regularidad media observada utilizando todos los modelos simultaneamente es siempre superior al 80%.

Para analizar la presencia de este nuevo modelo de regularidad hemos usado una amplia carga de trabajo. Además hemos buscado programas de prueba que utilicen vectores índice para acceder a matrices dispersas o para realizar cálculos simbólicos y los hemos reunido en una nueva colección (IAbench).

Como resultado del análisis se han obtenido métricas que muestran que el nuevo modelo de regularidad puede ser utilizado para prebúsqueda *hardware* de datos.

7.1.2 Búsqueda de nuevos mecanismos de prebúsqueda

Hemos definido dos nuevos prebuscadores (uno basado en *linear link* y otro de correlación) y los hemos comparado con otros prebuscadores conocidos. La comparativa se ha realizado en términos de prestaciones, número de prebúsquedas generadas y número de prebúsquedas usadas, utilizando un simulador de un procesador superescalar agresivo junto a una jerarquía de memoria *on-chip* de altas prestaciones.

Por un lado, el prebuscador basado en *linear link* no parece tener éxito en un procesador fuera de orden tan agresivo. Un análisis más en profundidad muestra que el *load* consumidor se lanza pocos ciclos después con lo que la ganancia queda *oculta* por el fuera de orden. Por ello nos centramos en lo que sigue en los prebuscadores de correlación y en el *secuencial marcado*.

Por otro lado, el prebuscador de correlación propuesto (PDFCM) tiene un enfoque alternativo a PC/DC. Manteniendo similar el tamaño de las tablas se evitan los recorridos de listas encadenadas y se consigan iguales o mejores prestaciones.

Hemos visto también cómo el aumento de la agresividad, tanto en forma de grado como de distancia, es beneficioso en general para todos los prebuscadores. En media, parece que el *secuencial marcado* es un claro ganador, pero también genera grandes pérdidas en algunas aplicaciones, especialmente al aumentar el grado o la distancia, debido al número de prebúsquedas inútiles generadas. A pesar de esto, consigue ganancias en otras aplicaciones en las que los prebuscadores basados en reconocimiento de patrones no consiguen resultados. Es decir, la agresividad del *secuencial*

marcado resulta beneficiosa en aplicaciones con recorridos erráticos de memoria, cuando las estructuras no están reservadas de forma muy dispersa.

También se aprecia que los beneficios de los prebuscadores basados en patrones en las aplicaciones de enteros son muy bajos, y siguen siendo un reto a conseguir.

7.1.3 Control de agresividad de prebuscadores agresivos

Orientados por las conclusiones anteriores, se han presentado distintas formas de ajustar la agresividad del prebuscador *secuencial marcado* de forma que consigue funcionar de forma similar o mejor que uno de los mejores prebuscadores conocidos hasta el momento, el SMS.

De todas las técnicas que proponemos, las que se comportan mejor son las que adaptan la agresividad del prebuscador basándose en la utilidad de las prebúsquedas realizadas previamente (Ad2 y Ad4) o en la variación del IPC (AdIPC). Todos estos métodos igualan las prestaciones obtenidas por el prebuscador SMS en aplicaciones enteras, mientras que obtienen más prestaciones en coma flotante con un 60% menos de accesos en L2.

Además los *costes hardware* de estas propuestas son mínimos. Ad2 necesita únicamente un bit extra por bloque de *cache*. El prebuscador adaptativo Ad4 propuesto (Ad4(8,32)) necesita adicionalmente una tabla de 64 Bytes, muy inferior a la tabla de 32 KB necesitada por el prebuscador SMS. Por su parte, AdIPC sólo requiere de dos contadores.

También se ha comprobado que el propio *Prefetch Address Buffer* (PAB) puede servir como elemento de filtrado y ayudar a reducir la presión generada por el prebuscador en L2 en un 30%.

Finalmente, se ha comprobado que todas las propuestas presentadas en este capítulo son opciones razonables, que no presentan pérdidas y tienen un *coste hardware* muy pequeño.

7.1.4 Definición de un esquema de prebúsqueda adaptable a objetivos

Se ha propuesto un esquema general de prebúsqueda que permite ser adaptado a distintos objetivos. Se han definido tres configuraciones diferentes de dicho esquema para tres objetivos diferentes: minimizar el coste hardware, minimizar la presencia de pérdidas en todas las aplicaciones y maximizar las prestaciones.

La evaluación de nuestras configuraciones se ha efectuado en el marco del *Primer Campeonato de Prebuscadores de Datos (First JILP Data Prefetching Championship).* Este campeonato ha sido la mejor oportunidad para comparar nuestro trabajo con el del resto de grupos de investigación, ya que todos los participantes han usado la misma infraestructura de simulación. Esto ha permitido comparar todas las propuestas en igualdad de condiciones.

Nuestra participación obtuvo el premio al mejor artículo (*Best Paper Award*) y el tercer puesto en prestaciones, en un empate técnico con el segundo clasificado. Además la viabilidad del método propuesto por el primer clasificado es muy discutible, frente a la simplicidad de nuestras configuraciones.

Finalmente, todos los códigos fuente de las propuestas están a disposición de la comunidad científica a través de la página web del DPC-1, lo que permite que cualquier grupo de investigación pueda analizar detalladamente todos los métodos.

7.2 Líneas abiertas

A continuación citamos algunas líneas abiertas derivadas del trabajo realizado.

- Adaptar los mecanismos de prebúsqueda actuales a los diferentes niveles de *cache on-chip*.
- Estudiar técnicas de colocación inteligente de datos en los diferentes niveles de *cache*.
- Estudiar y desarrollar técnicas de prebúsqueda específicas para procesadores *multicore*.
- Desarrollar técnicas de prebúsqueda centradas en el controlador de memoria, incluyendo un simulador detallado.

APÉNDICE A

VisualPtrace

En este apéndice se muestran algunas pantallas de la aplicación VisualPtrace utilizada para ayudar a depurar el funcionamiento del procesador y la jerarquía de memoria diseñados. Esta herramienta fue diseñada en [Tor05]. Lo que aquí se presenta está extraído del manual de dicha 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 ciclo

actual, el signo + marca la entrada de una nueva instrucción en el segmentado, o el símbolo * 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 permite mostrar de forma gráfica la traza de ejecución.

La Figura a.2 muestra 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.

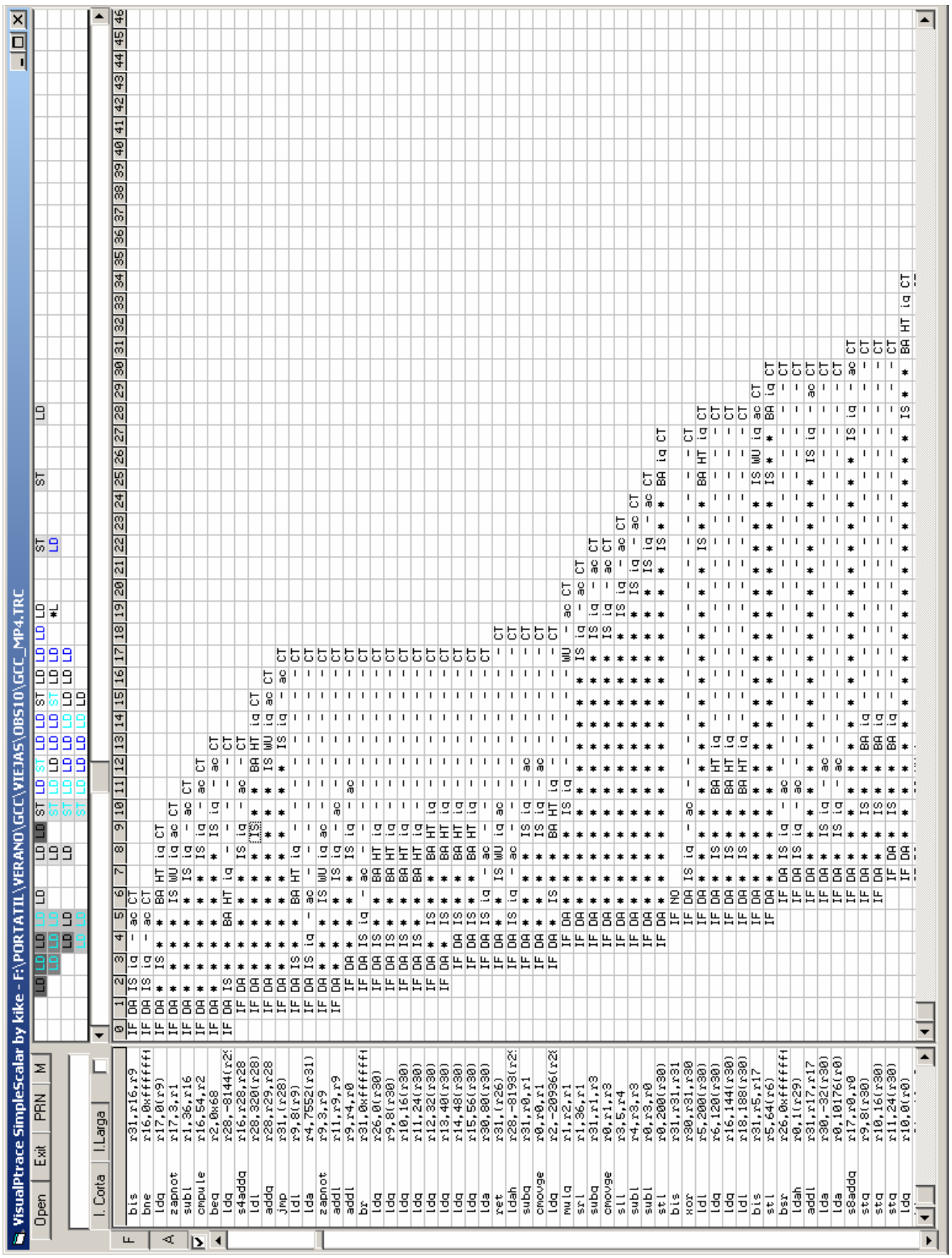

Figura a.2 Captura pantalla principal de VisualPtrace.

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.

APÉNDICE B

Otros resultados

En este apéndice se recogen otros resultados que han sido considerados importantes y se desea incluir en la memoria.

b.1 Patrones de regularidad: descomposición por aplicación

En la sección 2.5 se mostraron los resultados medios por colección de programas obtenidos al analizar los patrones de regularidad de dirección en una amplia carga de trabajo. Concretamente se mostraron la cobertura, la descomposición según el valor del parámetro α, la descomposición según la distancia entre *loads* productor y consumidor, y la descomposición según el comportamiento del *load* productor. En este apéndice se incluyen estos mismos resultados por aplicación (Figuras b.1, b.2, b.3 y b.4).

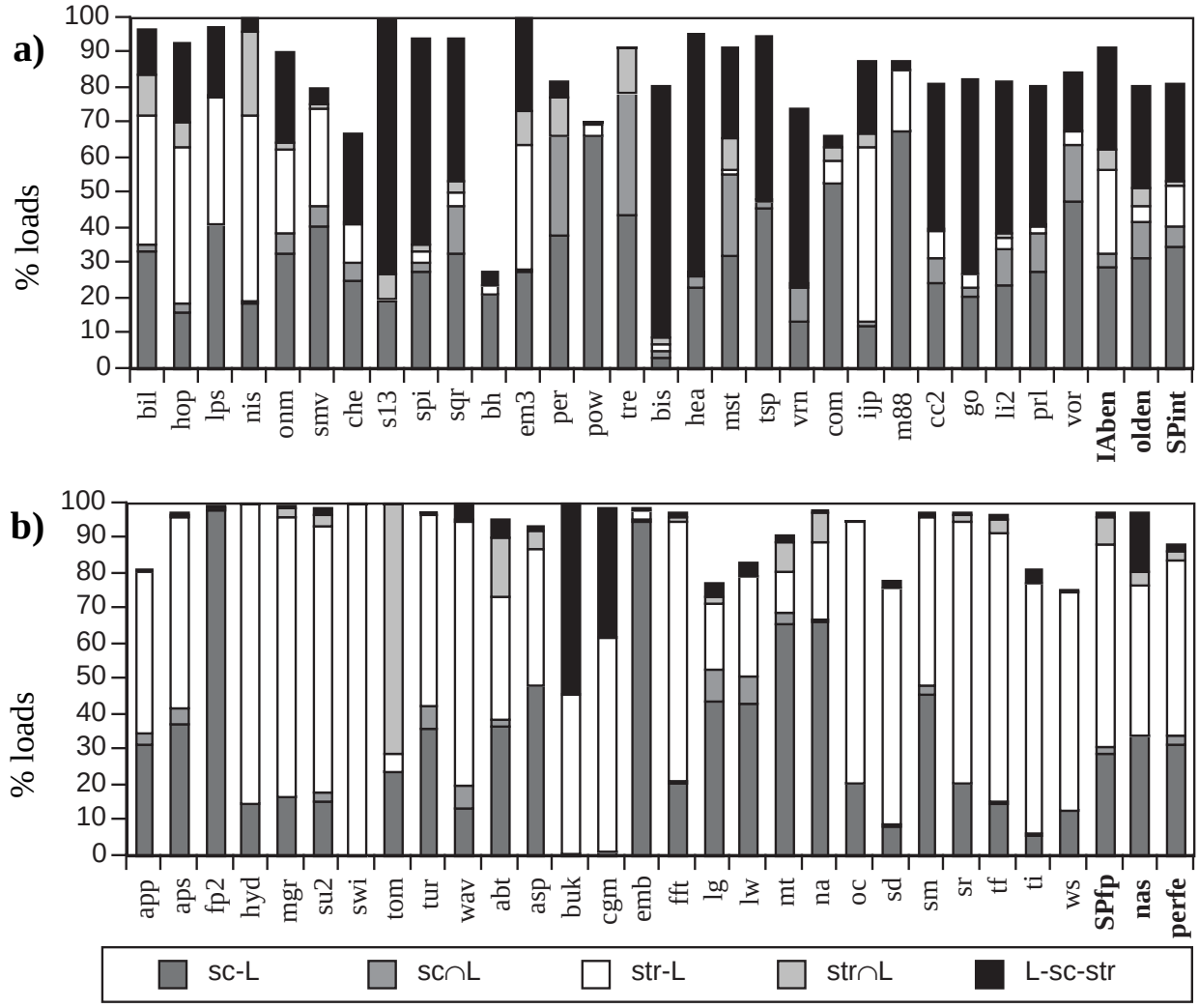

Figura b.1 Regularidades *stride cero* (sc), *stride* (str) y *linear link* encontrados en cada aplicación.

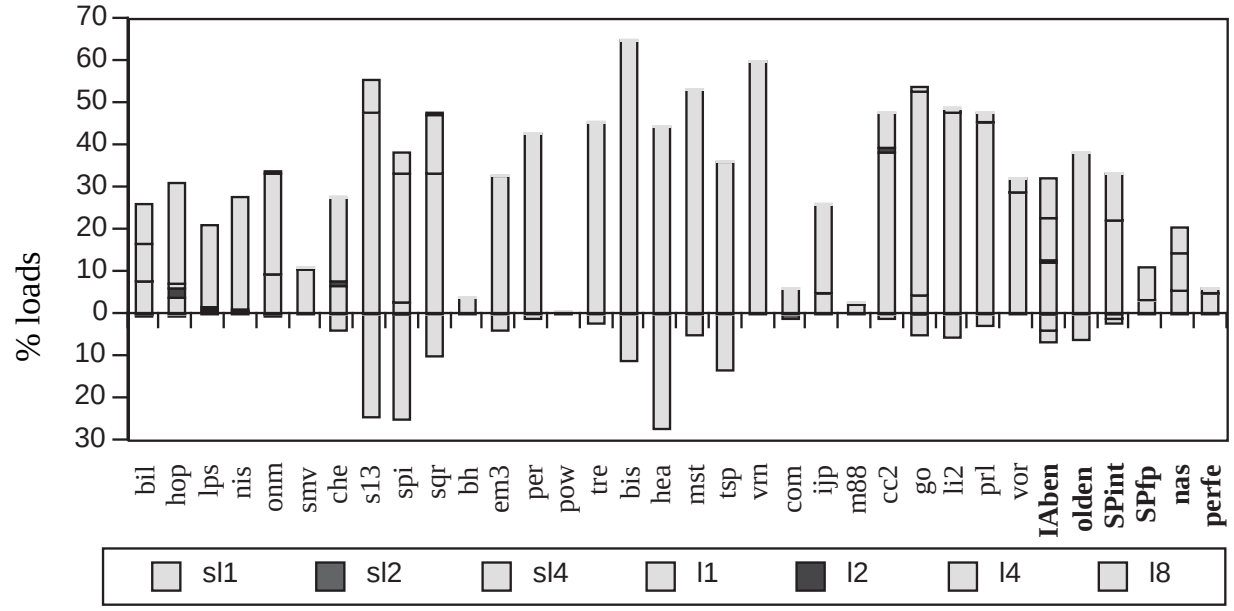

Figura b.2 Descomposición del patrón *linear link* según el valor del parámetro α.

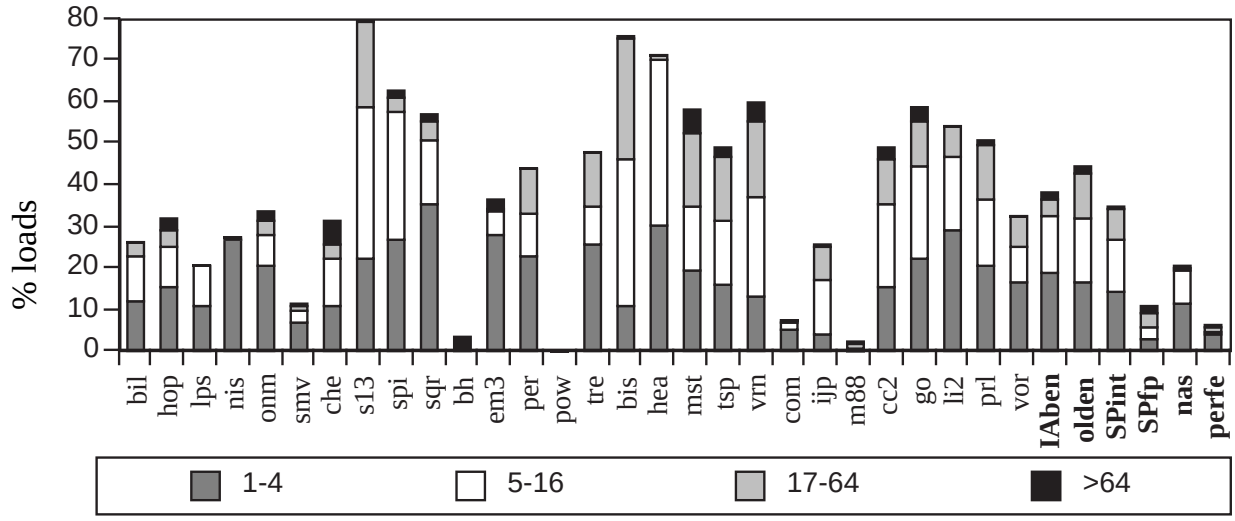

Figura b.3 Descomposición del patrón *linear link* según la distancia entre *loads*.

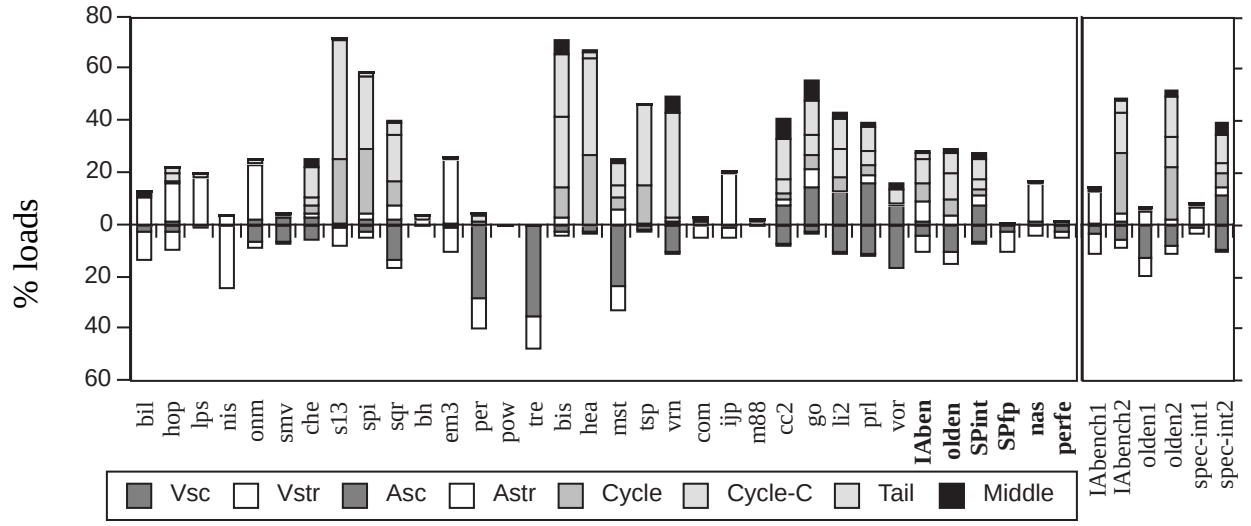

Figura b.4 Descomposición del patrón *linear link* según el comportamiento del *load* productor: valor *stride cero* (Vsc); valor *stride* (Vstr); dirección *stride cero* (Asc); dirección *stride* (Astr); *self-link* (Cycle); consumidor de *self-link* (Cycle-C); regularidad desconocida (Tail, Middle).

b.2 Políticas de grado: resultados por aplicación

En la sección 5.4 se mostraron las medias geométricas de los tiempos de ejecución normalizados de las políticas de grado propuestas con respecto a un sistema sin prebúsqueda. En este apéndice se descomponen los resultados por aplicación (Tablas b.1 - b.10).

Tabla b.1 Tiempos de ejecución de la política de grado Deg(x) respecto a un sistema sin prebúsqueda.

Deg(x)	**1**	**4**	**8**	**16**	**32**
vpr	96,3%	94,9%	94,8%	95,0%	95,7%
gcc	83,0%	81,9%	82,6%	90,1%	97,8%
mcf	94,1%	82,7%	77,7%	75,0%	74,0%
parser	95,2%	94,1%	94,3%	94,9%	95,7%
gap	90,0%	90,0%	90,1%	90,4%	91,0%
vortex	95,4%	95,2%	96,3%	98,2%	100,1%
bzip2	98,0%	98,3%	98,6%	99,2%	99,7%
twolf	99,6%	100,5%	100,8%	101,3%	101,8%
wupwise	72,8%	63,1%	61,3%	51,1%	51,3%
swim	54,5%	38,5%	35,7%	37,0%	38,3%
mgrid	48,0%	47,4%	48,0%	49,1%	51,4%
applu	70,6%	55,7%	54,6%	55,4%	58,0%
galgel	89,2%	83,9%	83,6%	85,0%	88,2%
art	98,1%	97,7%	98,0%	99,6%	103,7%
equake	51,4%	27,0%	21,0%	20,1%	21,1%
facerec	83,4%	79,6%	79,0%	80,0%	80,8%
ammp	100,9%	107,0%	111,2%	115,0%	115,6%
fma3d	69,7%	63,2%	62,5%	63,4%	65,1%
apsi	96,2%	93,2%	93,1%	93,0%	93,4%
gmean int	93,8%	92,0%	91,6%	92,6%	94,1%
gmean fp	73,6%	63,8%	61,8%	61,5%	63,2%
gmean	81,5%	74,4%	72,9%	73,1%	74,7%

Tabla b.2 Tiempos de ejecución de la política de grado Dist(x) respecto a un sistema sin prebúsqueda.

Dist(x)	4	8	16	32
vpr	97,3%	97,5%	98,2%	99,1%
gcc	81,4%	81,9%	85,8%	95,7%
mcf	87,1%	91,2%	92,3%	95,7%
parser	94,7%	95,1%	95,8%	97,3%
gap	90,5%	91,1%	92,4%	94,9%
vortex	97,2%	99,2%	101,3%	101,6%
bzip2	98,2%	98,2%	98,6%	99,2%
twolf	100,5%	100,4%	100,6%	100,5%
wupwise	82,9%	79,0%	87,8%	87,0%
swim	45,0%	45,8%	55,8%	73,9%
mgrid	51,6%	56,8%	67,7%	81,9%
applu	59,8%	63,2%	71,1%	83,6%
galgel	84,7%	85,2%	87,9%	93,7%
art	98,2%	99,5%	99,5%	103,9%
equake	31,8%	32,1%	42,3%	62,7%
facerec	78,7%	79,9%	80,9%	87,3%
ammp	106,4%	106,1%	105,5%	104,2%
fma3d	67,7%	69,0%	74,0%	83,9%
apsi	93,5%	93,5%	94,2%	95,7%
gmean int	93,1%	94,1%	95,5%	98,0%
gmean fp	68,7%	69,9%	76,4%	86,2%
gmean	78,1%	79,2%	83,9%	91,0%

Tabla b.3 Tiempos de ejecución de la política de grado Deg-dist(x) respecto a un sistema sin prebúsqueda.

Deg-dist(x)	**4**	**8**	**16**	**32**
vpr	95,1%	95,0%	95,2%	95,8%
gcc	81,8%	82,6%	87,5%	98,2%
mcf	83,3%	78,8%	76,0%	74,6%
parser	94,1%	94,4%	94,9%	95,7%
gap	90,0%	90,1%	90,4%	91,0%
vortex	95,3%	96,4%	98,2%	100,0%
bzip2	98,5%	98,8%	99,3%	99,6%
twolf	100,5%	100,8%	101,4%	102,0%
wupwise	63,2%	62,4%	56,8%	54,2%
swim	40,0%	36,2%	35,0%	37,1%
mgrid	47,6%	48,1%	50,2%	52,7%
applu	56,3%	56,0%	56,1%	57,1%
galgel	83,8%	83,5%	85,1%	87,8%
art	98,0%	98,4%	99,2%	102,3%
equake	27,7%	22,3%	21,8%	21,8%
facerec	79,5%	78,7%	80,3%	81,1%
ammp	107,0%	111,2%	114,6%	115,1%
fma3d	65,9%	63,2%	64,2%	65,7%
apsi	93,2%	93,1%	93,2%	93,6%
gmean int	92,1%	91,8%	92,5%	94,2%
gmean fp	64,5%	62,6%	62,5%	63,5%
gmean	74,9%	73,5%	73,7%	75,0%

Tabla b.4 Tiempos de ejecución de la política de grado Deg(1-x) respecto a un sistema sin prebúsqueda.

Deg(1-x)	**4**	**8**	**16**	**32**
vpr	95,5%	94,9%	94,6%	94,7%
gcc	81,4%	81,6%	89,1%	95,7%
mcf	87,6%	81,4%	76,6%	74,5%
parser	94,1%	94,2%	94,5%	95,1%
gap	90,0%	90,2%	90,5%	91,2%
vortex	95,0%	95,4%	96,4%	97,8%
bzip2	97,7%	97,6%	97,8%	98,1%
twolf	99,9%	100,1%	100,5%	100,7%
wupwise	65,6%	64,7%	55,4%	56,0%
swim	40,5%	38,3%	39,3%	41,0%
mgrid	47,3%	48,5%	49,6%	51,8%
applu	58,0%	56,1%	57,2%	59,6%
galgel	84,1%	83,8%	85,1%	88,2%
art	98,6%	98,9%	99,2%	103,0%
equake	28,2%	22,7%	21,6%	22,9%
facerec	77,1%	76,8%	78,0%	81,4%
ammp	103,4%	105,5%	107,7%	109,0%
fma3d	63,5%	63,1%	64,0%	65,6%
apsi	93,3%	93,2%	93,2%	93,6%
gmean int	92,5%	91,7%	92,2%	93,1%
gmean fp	64,5%	62,8%	62,5%	64,5%
gmean	75,1%	73,7%	73,6%	75,3%

Tabla b.5 Tiempos de ejecución de la política de grado Ad1(x) respecto a un sistema sin prebúsqueda.

Ad1(x)	**4**	**8**	**16**	**32**
vpr	96,8%	96,9%	96,9%	96,9%
gcc	81,4%	81,6%	89,0%	95,7%
mcf	97,2%	97,3%	97,2%	96,8%
parser	94,2%	94,3%	94,5%	95,1%
gap	90,0%	90,2%	90,5%	91,2%
vortex	97,1%	96,9%	97,0%	97,2%
bzip2	98,6%	98,6%	98,6%	98,7%
twolf	99,7%	99,7%	99,7%	99,7%
wupwise	68,7%	70,2%	70,3%	70,2%
swim	40,5%	38,3%	39,3%	41,0%
mgrid	47,3%	48,5%	49,6%	51,8%
applu	58,1%	56,0%	57,1%	59,1%
galgel	84,3%	84,1%	85,4%	88,5%
art	98,6%	98,9%	99,2%	103,0%
equake	28,3%	22,7%	21,6%	22,9%
facerec	78,3%	78,0%	79,8%	81,6%
ammp	100,3%	100,2%	100,3%	100,3%
fma3d	63,5%	63,1%	64,0%	65,6%
apsi	93,3%	93,2%	93,2%	93,6%
gmean int	94,2%	94,3%	95,4%	96,4%
gmean fp	64,7%	63,1%	63,6%	65,3%
gmean	75,8%	74,7%	75,4%	76,9%

Tabla b.6 Tiempos de ejecución de la política de grado Ad2(x) respecto a un sistema sin prebúsqueda.

Ad2(x)	**4**	**8**	**16**	**32**
vpr	95,4%	95,4%	95,5%	95,5%
gcc	81,7%	82,0%	89,2%	96,0%
mcf	89,9%	89,9%	89,6%	89,2%
parser	92,9%	93,0%	93,3%	93,9%
gap	85,0%	85,4%	85,9%	86,8%
vortex	96,4%	96,5%	96,5%	96,6%
bzip2	98,9%	98,9%	99,0%	99,0%
twolf	100,1%	100,1%	100,2%	100,2%
wupwise	68,7%	69,0%	69,3%	68,4%
swim	41,7%	39,3%	40,6%	42,7%
mgrid	47,6%	48,7%	49,7%	52,0%
applu	58,3%	56,3%	57,3%	59,4%
galgel	84,5%	84,2%	85,6%	88,5%
art	98,1%	98,4%	99,4%	103,1%
equake	28,4%	22,8%	21,7%	22,9%
facerec	80,8%	80,4%	81,7%	82,9%
ammp	101,3%	101,3%	101,4%	101,5%
fma3d	63,5%	62,9%	63,9%	65,5%
apsi	93,4%	93,2%	93,2%	93,6%
gmean int	92,3%	92,4%	93,5%	94,5%
gmean fp	65,2%	63,4%	63,9%	65,6%
gmean	75,5%	74,3%	75,0%	76,5%

Tabla b.7 Tiempos de ejecución de la política de grado Ad3(x) respecto a un sistema sin prebúsqueda.

Ad3(x)	**4**	**8**	**16**	**32**
vpr	96,9%	96,9%	96,9%	97,0%
gcc	81,4%	81,6%	89,0%	95,7%
mcf	97,3%	97,3%	97,4%	97,5%
parser	94,4%	94,4%	94,6%	95,0%
gap	90,0%	90,2%	90,5%	91,2%
vortex	97,0%	97,0%	97,0%	97,2%
bzip2	98,9%	98,9%	98,9%	98,9%
twolf	99,7%	99,7%	99,7%	99,7%
wupwise	68,0%	67,6%	67,7%	68,0%
swim	40,5%	38,3%	39,3%	41,0%
mgrid	47,3%	48,4%	49,5%	51,7%
applu	58,1%	56,0%	57,0%	59,1%
galgel	84,4%	84,1%	85,4%	88,5%
art	98,3%	98,9%	99,2%	103,0%
equake	28,3%	22,7%	21,6%	22,9%
facerec	78,3%	78,0%	79,9%	81,6%
ammp	100,1%	100,1%	100,1%	100,1%
fma3d	63,5%	63,1%	64,0%	65,6%
apsi	93,3%	93,2%	93,2%	93,6%
gmean int	94,3%	94,3%	95,5%	96,5%
gmean fp	64,6%	62,9%	63,3%	65,1%
gmean	75,8%	74,6%	75,3%	76,8%

Tabla b.8 Tiempos de ejecución de la política de grado Ad4(x,32) respecto a un sistema sin prebúsqueda.

Ad4(x,32)	**4**	**8**	**16**	**32**
vpr	95,3%	95,2%	95,2%	95,2%
gcc	82,0%	82,0%	89,1%	95,6%
mcf	89,2%	87,8%	86,5%	87,0%
parser	92,9%	92,9%	93,2%	93,8%
gap	85,0%	85,4%	85,9%	86,8%
vortex	95,3%	95,2%	95,4%	96,2%
bzip2	97,9%	97,7%	97,7%	97,8%
twolf	100,2%	100,1%	100,1%	100,1%
wupwise	68,1%	67,7%	63,8%	61,7%
swim	41,7%	39,2%	40,7%	42,7%
mgrid	47,6%	48,7%	49,7%	52,1%
applu	58,3%	56,3%	57,4%	59,6%
galgel	84,2%	84,0%	85,3%	88,2%
art	98,2%	98,5%	99,3%	102,9%
equake	28,4%	22,8%	21,7%	22,9%
facerec	80,7%	80,3%	81,2%	82,6%
ammp	100,9%	101,0%	101,3%	101,4%
fma3d	63,5%	62,9%	63,9%	65,5%
apsi	93,3%	93,2%	93,2%	93,6%
gmean int	92,0%	91,9%	92,8%	94,0%
gmean fp	65,1%	63,3%	63,4%	65,0%
gmean	75,3%	74,0%	74,4%	75,9%

Tabla b.9 Tiempos de ejecución de la política de grado Ad5(x) respecto a un sistema sin prebúsqueda.

Ad5(x)	**4**	**8**	**16**	**32**
vpr	100,0%	100,0%	100,0%	100,0%
gcc	83,5%	84,5%	90,8%	102,5%
mcf	100,3%	100,4%	100,5%	100,8%
parser	95,9%	95,8%	96,0%	96,3%
gap	92,2%	91,7%	91,8%	92,0%
vortex	99,8%	99,8%	99,9%	100,0%
bzip2	99,8%	99,8%	99,8%	99,7%
twolf	100,0%	100,0%	100,0%	100,0%
wupwise	69,1%	68,8%	68,8%	68,8%
swim	54,6%	44,3%	38,9%	36,7%
mgrid	60,4%	59,6%	57,6%	56,3%
applu	71,0%	66,1%	63,0%	62,0%
galgel	88,9%	89,0%	90,4%	92,1%
art	98,4%	99,0%	101,2%	106,1%
equake	38,8%	30,5%	24,8%	22,1%
facerec	85,4%	83,7%	83,4%	83,4%
ammp	100,8%	101,1%	101,2%	101,2%
fma3d	73,8%	70,5%	69,5%	69,3%
apsi	95,0%	93,9%	94,0%	94,3%
gmean int	96,3%	96,3%	97,3%	98,8%
gmean fp	73,4%	69,4%	67,0%	66,1%
gmean	82,3%	79,7%	78,4%	78,3%

Tabla b.10 Tiempos de ejecución de la política de grado AdIPC y del prebuscador SMS respecto a un sistema sin prebúsqueda.

	AdIPC	**SMS**
vpr	95,5%	96,5%
gcc	83,2%	88,9%
mcf	84,3%	76,6%
parser	95,7%	94,6%
gap	90,6%	87,6%
vortex	95,4%	98,0%
bzip2	98,7%	97,6%
twolf	100,0%	100,1%
wupwise	53,3%	62,9%
swim	33,9%	37,9%
mgrid	48,3%	58,3%
applu	54,3%	68,1%
galgel	84,4%	89,3%
art	98,0%	100,0%
equake	20,5%	29,6%
facerec	80,3%	76,4%
ammp	103,0%	101,3%
fma3d	62,7%	67,7%
apsi	93,6%	94,2%
gmean int	92,7%	92,2%
gmean fp	60,4%	67,1%
gmean	72,3%	76,7%

REFERENCIAS

Listado de Referencias

[ABI+06] J. Alastruey, J. L. Briz, P. Ibañez, V. Viñals, “Software Demand, Hardware Supply” IEEE Micro, vol. 26, no. 4, pp. 72-82, Jul/Aug. 2006.

[Alp99] “Alpha 21264 Microprocessor Hardware Reference Manual,” July 1999. Compaq Computer Corporation.

[BaC91] J.L. Baer and F.T. Chen, “An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty”, Supercomputing'91, pp. 176-186, 1991.

[BBB+91] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, H.D. Simon, V. Venkatakrishnan, S.K. Weeratunga, “The NAS Parallel Benchmarks”, Int. Journal of Supercomputer Aplications. Vol. 5 No. 3, pp. 66-73, 1991.

[BuA97] D.C. Burger and T.M. Austin, “The SimpleScalar Tool Set, Version 2.0,” UW Madison Computer Science Technical Report #1342, June 1997.

[ChB94] T.F. Chen and J.L. Baer, “A Performance Study of Software and Hardware Data Prefetching Schemes”, in proceedings of the 21st Annual International Symposium on Computer Architecture, pp. 223-232, IEEE Computer Society Press, Chicago, 1994.

[ChE98] G. Chrysos and J. Emer, “Memory dependence prediction using store sets”, in 25th Annual International Symposium on Computer Architecture, June 1998.

[CKP+90] G. Cybenko, L. Kipp, L. Pointer and D. Kuck, “Supercomputer Performance Evaluation and the Perfect Benchmarks”, 4th ICS, June 1990.

[CmK94] B. Cmelik and D. Keppel, “Shade: A Fast Instruction-Set Simulator for Execution Profiling”, 1994 SIGMETRICS, pp. 128-137.

[CYF+04] C. F. Chen, S. Yang, B. Falsafi and A. Moshovos, “Accurate and Complexity-Effective Spatial Pattern Prediction”, in proc. of the 10th Int. Symposium on High Performance Computer Architecture (HPCA 04), pp. 276- 287, 2004.

[DDS93] F. Dahlgren, M. Dubois and P. Stenstrom, “Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors”, ICPP, CRC Press, Boca Raton, Fla., pp. 156-163, 1993.

[DiZ09] M. Dimitrov and H. Zhou, “Combining Local and Global History for High Performance Data Prefetching”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[Dow06] J. Doweck, “Inside Intel Core Microarchitecture and Smart Memory Access”, White Paper, Intel Corporation, 2006.

[EMP09] E. Ebrahimi, O. Mutlu and Y.N. Patt, “Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems”, HPCA 2009, Raileigh, NC (USA): 7-17.

[FSF09] M. Ferdman, S. Somogyi and B. Falsafi, “Spatial Memory Streaming with Rotated Patterns”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[GaB06] I. Ganusov and M. Burtscher, “Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading”, in proc. of Parallel Architectures and Compilation Techniques (PACT 06) pp. 144 - 153, Sept. 2006.

[GJN09] M. Grannaes, M. Jahre and L.Natvig, “Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[GMT04] D. Gracia, G. Mouchard and O. Temam, “MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms”, proc. of the 37th Int. Symp. on Microarchitecture (MICRO-37), pp. : 43-54. December 2004.

[GNM05] Y. Guo, M.B. Naser and C.A. Moritz, “PARE: a Power-Aware Hardware Data Prefetching Engine”, in proc. of the 2005 Int. Symp. on Low Power and Design (ISPLED 05), pp. 339-344, 2005.

[GVB01] B. Goeman, H. Vandierendonck and K. De Bosschere. “Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency”, in procs. of the 7th Int. Symp. on High-Performance Computer Architecture (HPCA) pp. 207-218. Monterrey, Mexico 2001.

[HMK03] Z. Hu, M. Martonosi, S. Kaxiras, “TCP Tag Correlating Prefetchers”, in procs. of the 9th Int. Symposium on High Performance Computer Architecture (HPCA 03), pp. 317- 326. 2003.

[HuL06] I. Hur and C. Lin, “Memory Prefetching Using Adaptive Stream Detection”, in proc. of Int. Symp. On Microarchitecture, 2006.

[IIH09] Y. Ishii, M. Inaba, and K. Hiraki, “Access Map Pattern Matching Prefetch: Optimization Friendly Method”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[IVB+98] P. Ibáñez, V. Viñals, J.L. Briz, and M.J. Garzarán, “Characterization and Improvement of Load/Store Cache-based Prefetching”, in proc. of Int. Conf. on Supercomputing (ICS) Melbourne, Australia. pp.369-376 July 1998.

[JoG99] D. Joseph and D. Grunwald, “Prefetching Using Markov Predictors”, IEEE Trans. on Computers, vol. 48, No 2, pp. 121-133, 1999.

[Kes99] R.E. Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro, vol. 19, no. 2, pp. 24- 36, March/April 1999.

[KMW98] 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.

[Kre03] K. Krewell, “Fujitsu Makes SPARC See Double”, Microprocessor Report, Nov 2003.

[KST04] R. Kalla, B. Sinharoy and J.M. Tendler, “IBM Power5 Chip: A Dual-Core Multithreaded Processor”, IEEE Micro, vol. 24, no. 2, pp. 40-47, Mar./Apr. 2004, doi:10.1109/MM.2004.1289290

[LHP+09] G. Liu, Z. Huang, J.K. Peir, X. Shi and L. Peng, “Enhancement for Accurate Stream Prefetching”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[MeH96] S. Mehrotra and L. Harrison, “Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Program”, in proc. 10th Int. Conf. on Supercomputing, pp. 133-139, May 1996.

[MKP05] O. Mutlu, H. Kim, and Y. N. Patt, “Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns”, proc. of the 38th Int'l Symp. on Microarchitecture (MICRO 05), pp 233-244, Nov. 2005.

[NaH02] 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.

[NDS04] K.J. Nesbit, A. S. Dhodapkar and J.E. Smith. “AC/DC: An Adaptive Data Cache Prefetcher”, in proc. of the 13th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT) Sept. 2004.

[NeS04] K.J. Nesbit and J.E. Smith. “Data Cache Prefetching Using a Global History Buffer”, in proc. of the 10th Annual Int. Symp. on High Performance Computer Architecture (HPCA) pp: 96-105, Madrid, Spain 2004.

[NeS05] K.J. Nesbit and J.E. Smith, “Data Cache Prefetching Using a Global History Buffer”. IEEE Micro 25 (3), pp. 90-97. May/June 2005.

[PuA06] P. Pujara and A. Aggarwal, “Increasing the cache efficiency by eliminating noise”, in proc. of the 12th Int. Symposium on High Performance Computer Architecture (HPCA 06), pp. 145- 154, 2006.

[RBI+07] L.M. Ramos, J.L. Briz, P. Ibáñez and V. Viñals, “Data Prefetching in a Cache Hierarchy with High Bandwidth and Capacity”, SIGARCH Comput. Archit. News 35, 4 (Sep. 2007), 37-44, DOI= http://doi.acm.org/10.1145/1327312.1327319.

[RBI+08] L.M. Ramos, J.L. Briz, P. Ibáñez and V. Viñals, “Low-cost Adaptive Hardware Prefetching”, LNCS - procs. of 14th International Conference on Parallel and Distributed Computing, August 26-29, 2008, Las Palmas de Gran Canaria, Spain. DOI= http://dx.doi.org/10.1007/978-3-540-85451-7_36.

[RBI+09] L.M. Ramos, J.L. Briz, P.E. Ibáñez and V. Viñals, “Multi-level Adaptive Prefetching based on Performance Gradient Tracking”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[RCR+95] A. Rogers, M. Carlisle, J. Reppy and L. Hendren, “Supporting Dynamic Data Structures on Distributed Memory Machines”, ACM Trans. on Programming Languages and Systems, March 1995.

[RIV+00] L.M. Ramos, P. Ibáñez, V. Viñals, and J.M. Llabería, “Modelling Load Address Behaviour Through Recurrences”, in proc. of IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS 00), Austin, pp. 101-108, Apr. 2000.

[RMS98] A. Roth, A. Moshovos and G.S. Sohi, “Dependence Based Prefetching for Linked Data Structures”, 8th ASPLOS, pp. 115-126, Oct. 1998.

[SaS96] Y. Sazeides and J.E. Smith, “The Predictability of Data Values”, 29thMICRO, pp. 238-247, 1996.

[SBC05] S. Sharma, J. G. Beu and T. M. Conte, “Spectral Prefetcher: an effective mechanism for L2 cache prefetching”, ACM Trans. on Architecture and Code Optimization vol. 2, n, 4, pp.423-450, Dec 2005.

[ShL09] A. Sharif and H.H.S. Lee, “Data Prefetching Mechanism by Exploiting Global and Local Access Patterns”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[SMH+07] S. Srinath, O. Mutlu, K. Hyesoon and Y.N. Patt, “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, HPCA-13 pp. 63-74, 2007.

[Smi78] A.J. Smith, “Sequential Program Prefetching in Memory Hierarchies”, Computer, vol. 11, no. 12, pp. 7-21, Dec. 1978, doi:10.1109/C-M.1978.218016.

[SMK+07] S. Srinath, O. Mutlu, H. Kim and Y.N. Patt, “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers”, in HPCA-13, pp. 63–74, 2007.

[SPH+02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically Characterizing Large Scale Program Behaviour,” in proc. of ASPLOS, October 2002.

[SSC03] S. Sair, T. Sherwood, and B. Calder, “A Decoupled Predictor-Directed Stream Prefetching Architecture”, IEEE Trans. Comput. vol. 52, n.3, pp. 260-276, Mar. 2003.

[Sun04] UltraSPARC III Cu - User's Manual.Sun Microsystems, January 2004, http:// www.sun.com/processors/manuals/ USIIIv2.pdf

[SWA+06] S. Somogyi, T.F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Spatial Memory Streaming”, in proc. of 33rd. Intl. Symp. on Computer Architecture (ISCA 06). pp.252 - 263 , 2006.

[TDF+02] J. Tendler, J. Dodson, J. J.S. Fields, H. Le, and B. Sinharoy, “POWER4 system microarchitecture. IBM”, Journal of Research and Development, vol. 46 n.1, pp. 5-25, Jan. 2002.

[Tor05] E.F. Torres, “Alternativas de Diseño en Memorias Cache de Primer Nivel Multibanco”, Tesis, Universidad de Zaragoza, Abril 2005.

[VKP09] S. Verma, D.M. Koppelman and L. Peng , “A Hybrid Adaptive Feedback Based Prefetcher”, 1st JILP Data Prefetching Championship, Raleigh, Feb 2009.

[WBM+03] Z. Wang, D. Burger, K.S. McKinley, S.K. Reinhardt, and C.C. Weems, “Guided region prefetching: a cooperative hardware/software approach”, in proc. of the 30th Annual Int'l Symposium on Computer Architecture (ISCA 03), pp. 388-398, Jun. 03.

[ZhL03] X. Zhuang and H.S. Lee, “Reducing Cache Pollution via Dynamic Data Prefetch Filtering”, IEEE TOC 56(1), January 2007 pp 18- 31.

[ZhL07] X. Zhuang and H.H.S. Lee, “Reducing Cache Pollution via Dynamic Data Prefetch Filtering”, IEEE Transactions on Computers vol 56, no. 1 pp. 18-31, January 2007.