Reducción de consumo en la caché de datos de nivel 1 utilizando un predictor de *forwarding*[1]

Pablo Carazo
Dpto. de Informática Aplicada
EU de Informática
Univ. Politécnica de Madrid
28031 Madrid
pcarazo@eui.upm.es

Rubén Apolloni
Univ. Nacional de San Luís
5700 San Luís
Argentina
rubenga@unsl.edu.ar

D. Chaver, F. Castro, L. Piñuel, F. Tirado
Dpto. de Arquitectura de Computadores y Automática. Facultad de Informática
Univ. Complutense de Madrid
28040 Madrid
{dani02, fcastror, lpinuel, ptirado}@pdi.ucm.es

Resumen

En la mayoría de los diseños de los procesadores actuales, el acceso a la caché de datos de nivel 1 (L1D) se ha convertido en uno de los componentes de mayor consumo debido a su incremento de tamaño y elevadas frecuencias de acceso. Para reducir este consumo, proponemos una sencilla técnica de filtrado. Nuestra idea se basa en un predictor de *forwarding* de alta precisión que determina si una instrucción de *load* tomará su dato vía forwarding a través de la LSQ –evitando en este caso el acceso a la L1D- o si debe ir a por él a la caché de datos. Nuestros resultados de simulación muestran que podemos ahorrar de media un 35% del consumo de la L1D, con una degradación despreciable de rendimiento.

1. Introducción

El consumo de energía en un microprocesador fuera de orden comprende múltiples estructuras incluyendo las cachés, bancos de registros, predictor de saltos, etc. Concretamente, las cachés internas (*on-chip*) aportan una parte significativa al consumo total. En este artículo se intenta reducir el consumo de la L1D de un procesador fuera de orden. Puede pensarse que esta investigación ya no es relevante dada la tendencia actual de la industria hacia las arquitecturas multinúcleo, donde –a veces- los *pipelines* utilizados son más simples. Sin embargo, estas arquitecturas homogéneas multinúcleo con pipelines en orden, sólo aportan beneficios apreciables para aplicaciones o cargas de trabajo escalables y algunos investigadores han destacado recientemente que los diseños futuros se beneficiarán de las arquitecturas asimétricas que combinan núcleos simples, eficientes energéticamente, con unos pocos núcleos complejos grandes consumidores de energía [1]. Las ineficiencias locales de un núcleo complejo pueden convertirse en mejoras globales del ratio rendimiento/vatios dada su capacidad de acelerar las partes secuenciales de una aplicación cuando estos núcleos complejos están ociosos. De esta forma, un único chip puede ser capaz de ofrecer una buena escalabilidad para las aplicaciones paralelas al tiempo que garantizan un rendimiento elevado en las partes secuenciales. En definitiva, tal y como se sugiere en [2], todavía deben investigarse métodos de mejorar el rendimiento secuencial a pesar de haber entrado en la era de los multinúcleos. Es más, si se utilizan varios núcleos fuera de orden –ya sea con diseño homogéneo o heterogéneo-, nuestra técnica se puede aplicar a cada caché L1D privada alcanzando un beneficio aún mayor.

El mecanismo que proponemos para reducir el consumo energético en la caché L1D se basa en un uso eficiente de la cola de loads y *stores* (LSQ), una estructura de los procesadores fuera de orden encargada de mantener todas la instrucciones de memoria en vuelo, detectando y forzando el cumplimiento de las dependencias de memoria. Una de las tareas fundamentales de la LSQ es suministrar el dato correcto a un load mediante forwarding –*store to load forwarding*- obviando el dato de la L1D y, por lo tanto, haciendo innecesario el acceso a la caché.

1 Este trabajo ha sido financiado por los proyectos CICYT-TIN 2008/508, Consolider Ingenio2010 2007/20111 y por la Red de Excelencia Europea Hipeac2

Aprovechándonos de la propuesta de Nicolaescu [3], que consigue incrementar significativamente el número de loads que reciben su dato de un store previo, y utilizando un predictor de forwarding preciso que sugiere si un load es probable que reciba su dato a través de forwarding, conseguimos reducir apreciablemente la cantidad de accesos a la caché L1D en una arquitectura x86-64. El bajo ratio de error de predicción que obtenemos hace que el IPC prácticamente no varíe.

El resto del artículo se organiza como sigue. La sección 2 resume los trabajos relacionados. En la sección 3 repasamos la implementación convencional y mostramos los nuevos mecanismos propuestos. En la sección 4 detallamos el entorno experimental mientras que en la sección 5 resaltamos y analizamos los resultados obtenidos. Finalizamos en la sección 6 relatando nuestras conclusiones.

2. Trabajo relacionado

Recientemente se han explorado muchas técnicas para reducir el consumo de energía de la caché. A continuación vamos a recapitular algunas de las más relevantes.

Una alternativa consiste en particionar las cachés en varias cachés más pequeñas [4] [5] [6] con la correspondiente reducción tanto en tiempo de acceso como en coste energético por acceso. Otro diseño, conocido como *filter cache* [7], consigue mejoras en el consumo filtrando referencias a la caché mediante una caché L1 excepcionalmente pequeña. Para minimizar la pérdida de eficiencia, tras la *filter cache* se coloca una caché L2 similar en tamaño y estructura a una típica caché L1. Una alternativa diferente, denominada caché de vías selectivas [8], permite inhibir un subconjunto de las vías de una caché de correspondencia directa en los periodos con una actividad moderada de la caché, al mismo tiempo que puede funcionar de forma completa en los periodos más intensivos de caché. Otra forma de ahorro de energía son las *loop caches* [9] que evitan algunos accesos a la caché de instrucciones. Consta de un pequeño *buffer* de instrucciones, denominado *loop cache*, que reduce la energía de extracción de instrucciones cuando se ejecuta un bucle corto. Este ahorro energético se consigue sin degradar el rendimiento. Otro enfoque diferente saca ventaja del comportamiento especial de las referencias a memoria: podemos sustituir la caché unificada de datos convencional, por múltiples cachés especializadas. Cada una maneja diferentes tipos de referencias a memoria de acuerdo a sus particulares características de localidad –ejemplos de esta idea son [10] [11], que explotan en ambos casos la localidad propia de los accesos a la pila-. Estas alternativas permiten mejorar en términos de rendimiento o de consumo energético. Jin y otros [12] consiguen ahorro de consumo en la caché L1D explotando la localidad espacial de los loads. En su técnica, los loads se traen siempre un macro dato desde la caché, incrementando las posibilidades de tener forwarding de load a load.

Nicolaescu y otros [3] proponen evitar el acceso a la caché de datos para aquellos loads que reciben su dato mediante forwarding. Para aumentar esta tasa, modifican el diseño de la LSQ para retener las instrucciones de load y store después de finalizadas (*commited*). De esta forma, un load posterior aumenta sus posibilidades de recibir el dato de una instrucción previa, ya sea de un store en vuelo, o de un store o load ya finalizado. El mecanismo –denominado cola de load store cacheada- se basa en la observación de la baja tasa de ocupación de la LSQ para algunas fases de algunos programas, lo que permite utilizar las entradas no ocupadas para albergar instrucciones de load y store finalizadas. Nuestro trabajo extiende y mejora este mecanismo.

Finalmente, dado que utilizamos un predictor de forwarding en nuestro diseño, mencionar que hay muchas propuestas basadas en predecir dependencias de memoria que proponen técnicas para saber por adelantado qué parejas de store-load tendrán dependencia y tomar las acciones oportunas [13] [14] [15] [16]. Sin embargo, todas ellas están sobredimensionadas para el objetivo de nuestro trabajo.

3. Filtrado de accesos a la L1D mediante un predictor de forwarding

3.1. Fundamentos

En la mayoría de los microprocesadores convencionales, cada load consulta la L1D para mover el dato requerido a un registro. En paralelo, se busca en la cola de stores (SQ) para ver si existe un store previo en vuelo que direcciona el mismo dato. Si se encuentra, el store suministra el dato correspondiente. En caso contrario, es la caché la que suministra dicho dato. La técnica que

proponemos se basa en la observación de que si el load toma su dato directamente de un store previo, el acceso a la L1D es totalmente innecesario, pudiendo evitarse y ahorrar algo de energía. Obviamente, esto sólo será útil si el porcentaje de loads que toman su dato de la SQ es suficientemente alto.

En un procesador RISC, la cantidad de registros arquitectónicos suele ser de 32, implementándose -por lo general- una arquitectura tipo registro a registro. Con esta configuración, el número de loads que obtienen su dato de stores previos es relativamente bajo (por ejemplo, en [17], menos del 15% de media), pudiendo hacer que los beneficios potenciales de intentar evitar el acceso a la L1D en estas poco frecuentes ocasiones, acabe dejando de tener sentido. Sin embargo, en una arquitectura de registro a memoria con tan sólo 16 registros arquitectónicos –como es el caso en x86-64, la arquitectura utilizada en el presente trabajo- el número de loads que toman su dato de un store previo es significativamente más alto como resultado de las operaciones extra provocadas por el desbordamiento del banco de registros (*register spilling*).

De forma complementaria, utilizaremos la propuesta de Nicolaescu [3], que incrementa significativamente el número de loads que reciben su dato a través de forwarding, debido tanto al forwarding desde un store de la SQ-Cacheada como al forwarding desde un load de la LQ-Cacheada.

En resumen, en una arquitectura x86-64 utilizando una LSQ-Cacheada tipo Nicolaescu, el número de forwardings puede ser relativamente alto –hasta un 40% de los loads-, lo que hace atractiva nuestra intuición inicial. Sin embargo, para ser capaces de filtrar estos accesos, necesitamos serializar las búsquedas en la LSQ y en la L1D, o saber por adelantado –por ejemplo haciendo una predicción- cuándo el load podrá o no recibir el dato mediante forwarding. Este es un aspecto clave todavía no abordado.

3.2. Estructura global

Tal y como acabamos de mencionar, una implementación obvia consistiría en serializar los accesos: el load busca primero en la SQ y después –sólo si es necesario- se accede a la caché. Sin embargo, este diseño no es eficiente: cuando no se encuentra un store previo del que hacer forwarding, el retardo en el que incurrimos al acceder a la L1D provocará una pérdida significativa de rendimiento. En este artículo utilizaremos un enfoque mucho más adecuado.

El diseño que proponemos (Figura 1) se basa en un predictor de forwarding: para cada load predecimos si podrá o no recibir su dato a través de forwarding. Por claridad expositiva nos referiremos a estos loads como *predichos-dependientes* y al resto como loads *predichos-independientes*. Para los loads predichos-dependientes sólo se accede a la SQ y a la LQ-cacheada, omitiendo el acceso a la L1D (por supuesto, con el riesgo de equivocarnos, en cuyo caso el acceso a la caché se lanza con un ciclo de retardo). Para los restantes loads, accedemos en paralelo tanto a la SQ, como a la LQ-cacheada y a la L1D (observar que en este caso, si el predictor se equivoca, el acceso a la L1D se torna innecesario). Un predictor de alta precisión aportará ahorros significativos de consumo con un coste despreciable en cuanto a rendimiento.

Se ha publicado mucho en el campo de la predicción de dependencias de memoria (sección 2). Sin embargo, se suelen emplear estructuras de predicción sofisticadas que exceden nuestro objetivo de predecir si un load recibirá su dato a través de forwarding. Por esta razón, no los hemos utilizado en nuestro trabajo. En su lugar, hemos evaluado dos tipos de predictores sencillos: basado en filtros Bloom [18] y basado en predictor de saltos [19].

Predictor basado en filtro Bloom En este primer tipo de predictores implementamos una tabla *hash* de contadores de coste bajo: en *issue*, cada load y store indexa, mediante una función hash aplicada a su dirección de memoria, una entrada de la tabla e incrementa el contador correspondiente. Posteriormente, en el commit, esta entrada se decrementa. Además, en el issue, los loads leen el contador para hacer la predicción antes de incrementarlo. Si el contador es mayor que cero, es probable -aunque no seguro- que haya otra instrucción de memoria coincidente en dirección y, por lo tanto, se predice que el load recibirá su dato a través de forwarding. Por otra parte, si el contador es cero, se predice el load como independiente.

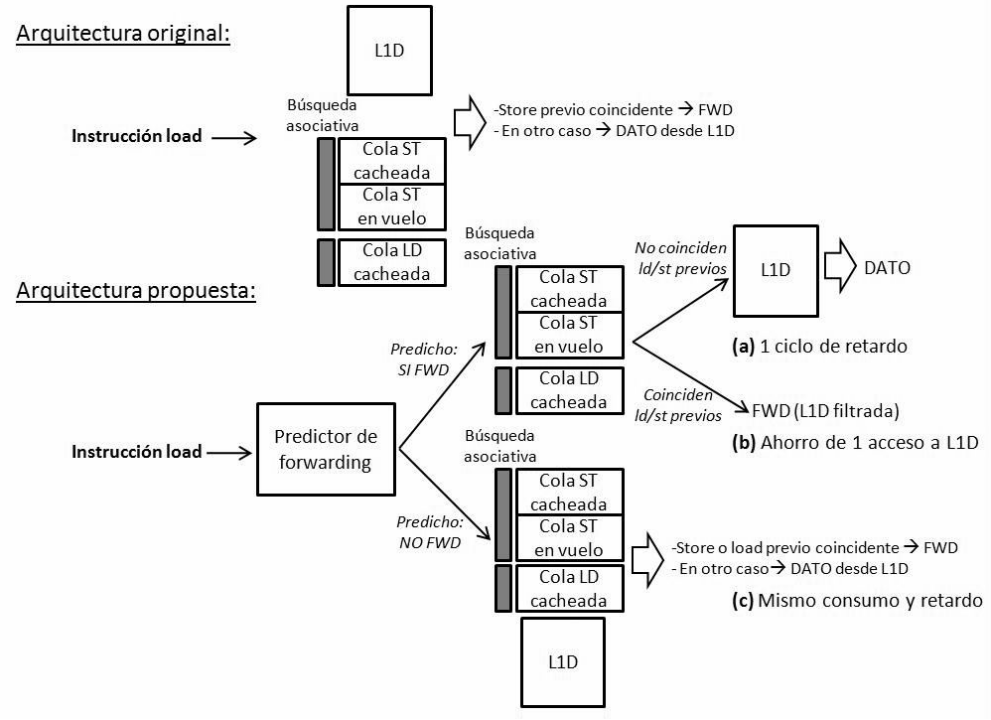

Figura 1. Arquitectura original y modificada. En la arquitectura original, tanto la LSQ como la L1D se acceden en paralelo. En nuestra propuesta, si el load es predicho-dependiente, el acceso se serializa (primero se accede a LSQ y, solo si es necesario, se accede después a la L1D). Si el load es predicho-independiente, el acceso se realiza como en la arquitectura original.

Como se explica en [20], en este caso podrían evitarse los accesos a la LQ y SQ. Sin embargo, dado que un acceso a la cache L1D consume mucha menos energía que un acceso a la LQ-SQ, en este artículo no consideramos esta capacidad de filtrado en LQ y SQ, ya que requeriría de un estudio en mayor profundidad.

Basado en predictor de saltos Este segundo tipo de predictores se basa en el bien conocido predictor de saltos bimodal. Similar a lo que sucede con las instrucciones de salto, los loads suelen tener un comportamiento fuertemente sesgado, de forma que dicho predictor debería funcionar bien. Una ventaja de este predictor bimodal respecto del basado en filtro Bloom es que la predicción se puede realizar tan pronto como se decodifica la instrucción de load, utilizando para ello el contador de programa. Por el contrario, un filtro Bloom debe consultarse con la dirección de memoria del load, que necesita calcularse previamente, de forma que la predicción, en este caso, debe retrasarse a la fase de issue.

Predictor combinado Finalmente, mencionar que también hemos considerado en nuestro trabajo un predictor combinado, mezclando un filtro Bloom con el predictor bimodal. Nuestra predicción final sólo dictará que un load recibirá su dato a través de forwarding si ambas estructuras predicen que el load será dependiente. Esta estructura se beneficia tanto de la información de forwarding del pasado de los loads, como de la información aportada por su dirección, dando los mejores resultados tal y como puede apreciarse en la sección de evaluación.

3.3. Soporte para coherencia y consistencia

La LSQ de la arquitectura base recibe peticiones de invalidación de procesadores remotos, de forma que nuestra técnica puede dar soporte relativamente fácil a la funcionalidad de coherencia y consistencia. Sin embargo, debemos resaltar una situación conflictiva que aparece en nuestro diseño si contamos con un protocolo de coherencia MESI: si reemplazamos un dato de la

L1D pero permanece en la LSQ-Cacheada, la línea *Shared* no se activará como respuesta a una petición de lectura remota, pudiendo poner erróneamente el dato remoto en estado Exclusivo (en lugar de Shared). Una posible solución sería forzar que la LSQ activase la línea Shared para cada lectura remota a bloques implicados en forwarding. Como trabajo futuro pretendemos mejorar este tratamiento ya que, aunque sencillo, es relativamente ineficiente.

4. Entorno experimental

Nuestra evaluación la hemos realizado con PTLsim [21], una herramienta de simulación eficiente con precisión a nivel de ciclo. La microarquitectura que modela por defecto PTLsim es una mezcla de distintas características de un Intel Pentium 4 [22] [23], un AMD K8 y un Intel Core 2 [24]. Algunos de los principales parámetros de simulación se listan en la Tabla 1.

Predictor de saltos	Combinado (Bimod-2bits + Gshare), 2K BTAC
Tamaño cola fetch inst.	32
Tamaño ROB	128
Tamaño LSQ	80 (LQ: 48, SQ: 32)
Tamaño LSAP	16
Registros físicos	256
Unidades funcionales	8: 4 ALU (2 INT, 2 FP), 2 Load, 2 Store
Ancho Fetch/Decode/ Issue/Commit	4/4/4/4
Caché instrucciones L1	32KB (4 vías, línea 64B)
Caché de datos L1	16KB (4 vías, línea 64B, 2 ciclos de latencia)
Caché de datos L2	256KB (16 vías, línea 64B, 6 ciclos de latencia)
Caché de datos L3	4MB (32 vías, línea 64B, 16 ciclos de latencia)
Latencia de memoria principal	140 ciclos

Tabla 1. Parámetros por defecto de PTLsim

Hemos realizado pruebas utilizando 24 aplicaciones de la plataforma SPEC CPU2006 compilada para la arquitectura x86-64. Los parámetros de la tecnología se corresponden con 45 nm y 1.0V V_{dd}. Simulamos regiones de 100M de instrucciones después de alcanzar un punto de paso a modo simulado (*triggering point*) [25], que marca el inicio del área de código en el que el comportamiento de la aplicación es representativo de la ejecución completa. No obstante, estamos en un proceso de afinar todavía más el punto de paso a modo simulado y evaluar también con una traza de más de 100M de instrucciones.

Para medir el impacto de nuestra técnica de filtrado de acceso a la caché de datos sobre el consumo de dicha caché, utilizamos CACTI 5.3 [8] para modelizar la caché de la Tabla 1. En concreto, para estimar el consumo de energía de la caché, multiplicamos el número de lecturas y escrituras a la L1D por el consumo de cada tipo de accesos a esta caché. Además, hemos modificado el simulador para incorporar nuestros predictores en la microarquitectura simulada, pero el consumo energético que esto conlleva puede despreciarse en comparación con los ahorros de energía obtenidos en la caché.

A continuación hacemos algunos análisis cuantitativos para comprender mejor la efectividad del diseño propuesto.

5. Evaluación

5.1. Principales resultados

En esta sección comparamos el consumo de la caché de datos y el rendimiento del sistema completo utilizando la microarquitectura base y la modificada. La Figura 2 muestra el ahorro de energía que se consigue con nuestra técnica en el acceso a la caché de datos, mientras que la Figura 3 ilustra el impacto en el rendimiento. En ambas figuras utilizamos el predictor combinado ya que ofrece los mayores valores de precisión tal y como se explica en la sección 5.2. Podemos extraer las conclusiones siguientes.

Primero, gracias a la inclusión de nuestros cambios, una parte significativa de los loads se predicen correctamente como predichos-dependientes, evitando de esta forma el acceso a la caché. De esta manera eliminamos una parte significativa del consumo dinámico de energía de la L1D tal y como se muestra en la Figura 2. De media, para los cuatro predictores combinados estudiados –variando el tamaño del predictor bimodal-, el ahorro de consumo en la L1D ronda el 35%. Puede observarse que el ahorro de consumo presenta algunos casos extremos como en *libquantum* y *GemFDTD*.

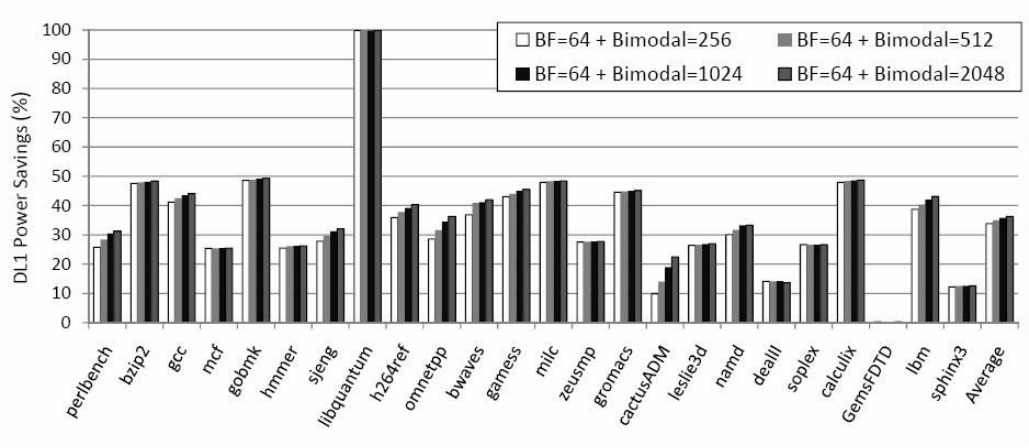

Figura 2. Ahorro de energía en la caché L1D

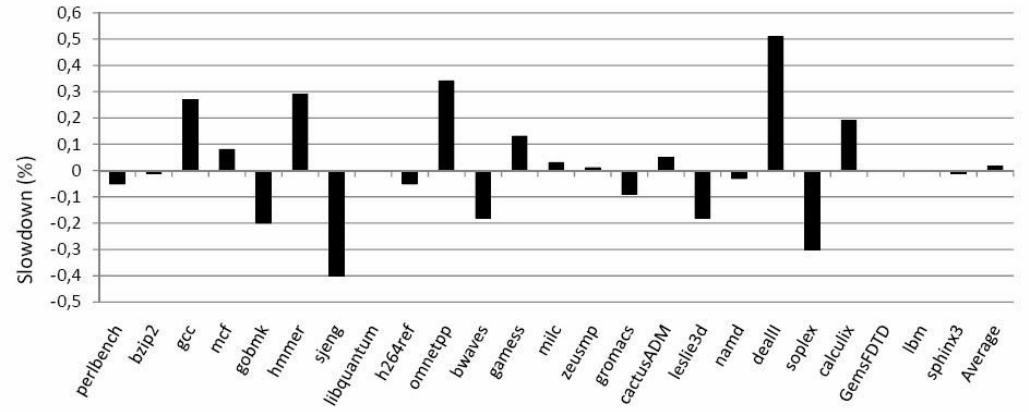

Figura 3. Impacto en el rendimiento utilizando un predictor BF de 64 entradas más otro bimodal de 256 entradas

Obviamente, con un predictor de forwarding de alta precisión y una pérdida de rendimiento muy reducida –Figura 3- el ahorro de consumo obtenido en cada aplicación por los accesos a la L1D está muy correlacionado con el porcentaje total de forwarding presente en la propia microarquitectura base.

Como segunda conclusión, resaltar que el rendimiento medio obtenido con nuestras modificaciones permanece casi invariable –ver Figura 3-. Sin embargo, si nos fijamos en las aplicaciones individualmente, observamos un comportamiento irregular. Algunas aplicaciones experimentan una pérdida de rendimiento moderada. Esto se explica por el ratio de error de predicción observado en cada aplicación, ya que incrementa ligeramente la latencia real de los loads. Por lo tanto, a mayor error de predicción, mayor pérdida de rendimiento. Además, evitar el acceso a la L1D para algunos loads, puede afectar a la política de reemplazo en relación con la microarquitectura base, resultando en fluctuaciones aleatorias del IPC.

5.2. Predictores de forwarding

Para comparar la precisión de los predictores de forwarding evaluados –filtro Bloom, bimodal (con 1 y 2 bits por entrada), y bimodal (2 bits) más filtro Bloom- siguiendo a Grunwald y otros, utilizamos las métricas siguientes ya manejadas en estimadores de confianza para especulación de control [27]:

- *Predictive Value of a Positive Test* (PVP). Identifica la probabilidad de que la predicción de un load como dependiente sea correcta. Se calcula como el ratio entre el número de loads correctamente predichos-dependientes y el número total de loads predichos como dependientes.
- *Predictive Value of a Negative test* (PVN). Identifica la probabilidad de que la predicción de un load como independiente sea incorrecta. Se calcula como el ratio entre el número de loads predichos erróneamente como independientes y el número total de loads predichos como independientes.

En nuestro caso, utilizando predictores con un alto PVP evitamos degradar el rendimiento. Por otra parte, si se predijesen incorrectamente como independientes muchos loads (elevado PVN), muchos accesos a la caché devendrían innecesarios, provocando una pérdida de oportunidades para reducir el consumo de energía de la L1D. Por lo tanto, en nuestro diseño, sólo son aceptables valores muy altos de PVP y valores muy bajos de PVN.

En la Figura 4 mostramos las mediciones de PVP y PVN para los diferentes tamaños de todos los predictores analizados. Intuitivamente, según se incrementa el tamaño de cualquier predictor, aumentará el PVP y decrementará el PVN, llevándonos a un mejor comportamiento del predictor.

Observar que el PVN para un filtro Bloom siempre es cero, ya que no existen falsos negativos –cuando se predice un load como independiente, el predictor nunca yerra-. De esta figura puede concluirse –tal y como la intuición dicta- que combinar la información del comportamiento pasado respecto de forwarding (predictor bimodal) y las direcciones de memoria (filtro Bloom) conlleva un predictor de mayor precisión.

6. Conclusiones

Las principales aportaciones de este artículo son:

- Hemos implementado y evaluado la propuesta de Nicolaescu [3] en un modelo de microarquitectura diferente y más común –la extendida arquitectura x86-64-.
- En vez de serializar los accesos a la LSQ y a la L1D tal y como hace Nicolaescu [3], nosotros hacemos ambos accesos en paralelo utilizando un sencillo predictor de forwarding.
- Hemos estudiado exhaustivamente la efectividad de distintos predictores, eligiendo el óptimo sopesando entre precisión y ahorro de consumo. Utilizando este tipo de predictor en una arquitectura x86-64 en combinación con la propuesta de Nicolaescu [3], somos capaces de filtrar una cantidad significativa de accesos.

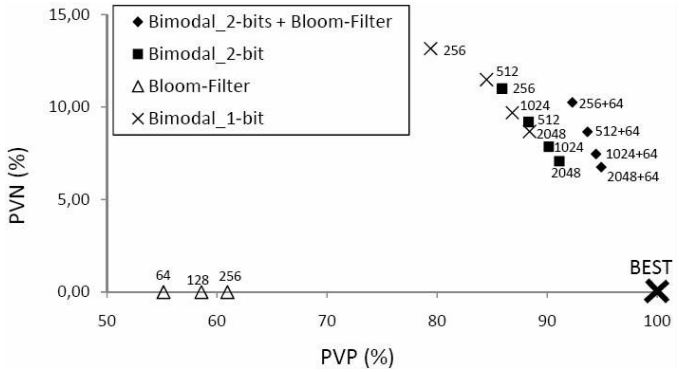

Figura 4. PVP y PVN para los predictores estudiados. Los resultados mostrados son valores medios para todas las aplicaciones. Para los predictores bimodales (de 1 y 2 bits) los puntos mostrados se corresponden con tamaños de 256, 512, 1K y 2K. Para el filtro Bloom se muestran resultados con 64, 128 y 256 entradas. Finalmente, el predictor combinado utiliza un filtro Bloom de 64 entradas y un predictor bimodal de 2 bits con 256, 512, 1K y 2K entradas. El predictor combinado alcanza el valor más alto de PVP y el más bajo de PVN, siendo el más cercano al mejor (BEST) de los comportamientos.

En definitiva, el mecanismo de filtrado propuesto consigue un ahorro de energía en la L1D del 35% de media para la configuración estudiada. La inclusión de este esquema apenas varía el rendimiento global –menos de un 0,1% de ralentización por término medio-, agregando un hardware simple y de bajo coste, menos de 100B.

Referencias

[1] Bower, F., Sorin, D., Cox, L.: The impact of dynamically heterogeneous multicore processors on thread scheduling. Micro, IEEE 28(3) (Mayo-Junio 2008) 17-25

[2] Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. IEEE Computer 41(7) (2008) 33-38

[3] Nicolaescu, D., Veidenbaum, A., Nicolau, A.: Reducing Data Cache Energy Consumption via Cached Load/Stores Queue. ISLPED'03. 252-257

[4] Ghose, K., Kamble, M.: Reducing Power in Superescalar Processor Cache using Subbanking, Multiple Line Buffer and Bit-Line Segmentation. ISLPED'99. 70–75

[5] Su, C.L., Despain, A.M.: Cache Designs for Energy-Efficiency. HICSS'95. 306–314

[6] Racunas, P., Patt, Y.N.: Partitioned First-Level Cache Design for Clustered Microarchitectures. ICS'03. 22–31

[7] Kin, J., Gupta, M., Mangione-Smith, W.: The Filter Cache: An Energy Efficient Memory Structure. MICRO'97. 184–193

[8] Albonesi, D.: Selective Cache Ways: On-Demand Cache Resource Allocation. Journal of Instruction-Level Parallelism 2 (2000)

[9] Lee, L., Moyer, B., Arends, J.: Instruction Fetch Energy Reduction Using Loop Cache for Embedded Applications with Small Loops. ISLPED'99. 267-269

[10] Lee, H., Smelyanskiy, M., Newburn, C., Tyson, G.: Stack Value File: Custom Microarchitecture for the Stack. HPCA'01. 5-14

[11] Cho, S., Yew, P., Lee, G.: Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor. ISCA'99. 100-110

[12] Jin, L., Cho, S.: Reducing Cache Traffic and Energy with Macro Data Load. ISLPED'06. 147-150

[13] Moshovos, A., Breach, S., Vijaykumar, T., Sohi, G.: Dynamic Speculation and Synchronization of Data Dependences. ISCA'97. 181-193

[14] Chrysos, G., Emer, J.: Memory Dependence Prediction Using Store Sets. ISCA'98. 142-153

[15] Subramaniam, S., Loh, G.: Store Vectors for Scalable Memory Dependence Prediction and Scheduling. HPCA'06. 65-76

[16] Park, I., Ooi, C., Vijaykumar, T.: Reducing Design Complexity of the Load/Store Queue. MICRO'03. 411-422

[17] Castro, F., Chaver, D., Piñuel, L., Prieto, M., Huang, M., Tirado, F.: LSQ: A Power Efficient and Scalable Implementation. IEEE Proceedings: Computer and Digital Techniques 153(6) (Noviembre 2006) 389-398

[18] Bloom, B.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communic. of the ACM 13(7) 1970. 422-426

[19] McFarling, S.: Combining Branch Predictors. Technical report tn-36, Western Research Laboratory, Digital Equipment Corporation (Junio 1993)

[20] Sethumadhavan, S., Desikan, R., Burger, D., Moore, C., Keckler, S.: Scalable Hardware Memory Disambiguation for High ILP Procs. MICRO'03. 399-410

[21] Yourst, M.T.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. ISPASS'07. 23-34

[22] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., Roussel, P.: The Microarchitecture of the Pentium 4 Proc. Intel Technology Journal (Q1 2001)

[23] Boggs, D., Baktha, A., Hawkins, J., Marr, D., Miller, J., Roussel, P., Singhal, R., Toll, B., Venkatraman, K.: The Microarchitecture of the Intel Pentium Processor on 90nm Technology. Intel Technology Journal 8(1) (Febrero 2004) 1-17

[24] Univ. de Conpenhague. College of Eng.: The Microarch. of Intel and AMD CPU's: an Optimization Guide for Assembly Programmers and Compiler Makers. (2009)

[25] Apolloni, R., Chaver, D., Castro, F., Piñuel, L., Prieto, M., Tirado, F.: An Efficient Memory Disambiguation Hardware in an x86 Architecture. XX Jornadas de Paralelismo (Septiembre 2009)

[26] http://www.hpl.hp.com/research/cacti

[27] Grunwald, D., Klauser, A., Manne, S., Pleszkun, A.: Confidence Estimation for Speculation Control. ISCA'98. 122-131