Eficacia vs. Eficiencia: Una Decisión de Diseño en RunAhead

Tanausú Ramírez, Adrián Cristal, Oliverio J. Santana, Alex Pajuelo y Mateo Valero
Departamento de Arquitectura de Computadores
UPC – Barcelona
{tramirez, adrian, osantana, mpajuelo, mateo}@ac.upc.edu

Resumen

Existe una gran diversidad de técnicas que incrementan el rendimiento de las aplicaciones al aliviar uno o varios de los problemas más importantes de los procesadores actuales. Por desgracia, muchas veces se descuida el estudio de los posibles efectos colaterales que pueden causar que los diseñadores de un procesador real decidan no implementar un determinado mecanismo.

En este artículo presentamos, como caso práctico, el análisis de los efectos colaterales causados por un mecanismo ampliamente conocido: *RunAhead.* Nuestra evaluación muestra que RunAhead es eficaz incrementando el rendimiento de aplicaciones en procesadores superescalares (12% para SpecINT). Sin embargo, lo hace de forma ineficiente, puesto que consume 4 veces más energía por instrucción retirada comparado con un procesador superescalar sin este mecanismo.

1. Introducción

Los procesadores superescalares actuales presentan dos problemas graves que limitan el rendimiento de las aplicaciones que ejecutan: la predicción de saltos y la larga latencia de las instrucciones que acceden a memoria para obtener datos.

El primero de estos problemas reduce la posibilidad de obtener suficientes instrucciones del programa para llenar grandes ventanas de instrucciones, especialmente en presencia de saltos difíciles de predecir, y por tanto, reduce el paralelismo a nivel de instrucción.

El segundo problema, conocido como *memory wall* [19], se debe a la disparidad de aumento de frecuencia entre el procesador y la memoria. Esto hace que las instrucciones de acceso a memoria cada vez presenten latencias más y más largas, reduciendo considerablemente la posible explotación del paralelismo a nivel de datos presente en las aplicaciones. Si a esto le unimos la naturaleza en orden de la etapa de commit de un procesador superescalar, la situación se agrava todavía más. Las instrucciones de larga latencia retardan el commit de las instrucciones posteriores, provocando que, antes o después, la ventana de instrucciones se llene completamente, lo que causará el bloqueo de la búsqueda, decodificación y ejecución de nuevas instrucciones, penalizando gravemente el rendimiento.

Se han propuesto muchas técnicas para aliviar estos problemas, y por tanto, incrementar el rendimiento global de un procesador superescalar. Pero muchas veces, en estas propuestas, no se tiene en cuenta que atacar estos problemas mediante mecanismos agresivos puede llevar a que, dependiendo del escenario, una idea sea inimplementable. Como claro ejemplo de esto, se puede citar algunas técnicas que utilizan ejecución especulativa. Estas técnicas se basan en ejecutar, de forma especulativa, muchas más instrucciones de las estrictamente necesarias, con el objetivo de adelantar la obtención de datos críticos. Estas instrucciones, por supuesto, consumen energía en su ejecución, lo cual incrementa los requisitos de energía de un procesador en el que se aplique un mecanismo especulativo.

En este artículo se plantea la pregunta de si es justificable o no este aumento de consumo para obtener lo que a veces es un incremento marginal del rendimiento. Para ello, hemos elegido como caso de estudio un mecanismo ampliamente conocido: *RunAhead*. Este mecanismo se basa en ejecutar especulativamente las instrucciones independientes de una instruccion de load con larga latencia. Una vez este load es retirado, las instrucciones ejecutadas de forma especulativa se descartan, permaneciendo únicamente los cambios

que se hayan producido en la jerarquía de memoria. De esta manera, RunAhead incrementa eficazmente el rendimiento de ciertas aplicaciones gracias a la prebúsqueda de datos. Por desgracia, como efecto colateral se presenta el hecho de que no es un mecanismo eficiente: para conseguir ese incremento de rendimiento necesita consumir 4 veces más energía que un procesador superescalar sin este mecanismo. Esto hace que *RunAhead* no sea una buena alternativa en escenarios donde la energía sea una limitación importante en el diseño del procesador.

La estructura del resto de este artículo es la siguiente. La Sección 2 resume el trabajo realizado con anterioridad para aliviar el problema de la latencia de memoria. La Sección 3 describe el mecanismo de RunAhead. La metodología de este estudio se encuentra en la Sección 4. La Sección 5 evalúa *RunAhead* en términos de rendimiento y eficiencia. Finalmente, la Sección 6 presenta nuestras conclusiones.

2. Trabajo relacionado

La técnica más utilizada para reducir los tiempos de acceso a memoria es la incorporación , en el procesador, de pequeñas memorias con un tiempo de acceso reducido, conocidas como caches [3,12], que se basan en explotar la localidad espacial y temporal de los datos. Su principal inconveniente es que, si su tamaño no es elegido cuidadosamente, podrían llegar a provocar penalizaciones en el tiempo de ciclo del procesador.

Desde otro punto de vista, los mecanismos de prebúsqueda predicen la siguiente dirección efectiva de una instrucción de acceso a memoria para anticipar el acceso a sus datos. Estas técnicas pueden clasificarse en diversas categorías. Las técnicas de prebúsqueda software [6,17] están basadas en introducir instrucciones de prebúsqueda en tiempo de compilación. Por otro lado, las técnicas de prebúsqueda hardware [7,13,18] calculan las futuras direcciones efectivas a las que puede acceder una instrucción estudiando la historia de las ejecuciones previas de esta instrucción.

Existen técnicas basadas en el uso de varios contextos de ejecución o threads [5,9,11], como los *assisted-threads* y *helper threads* que utilizan un thread auxiliar para ejecutar secuencias de código que realicen prebúsqueda, ayudando al thread principal.

Por desgracia, existe el riesgo de que un mal comportamiento de este mecanismo resulte perjudicial, puesto que puede saturar el bus de memoria accediendo a datos que no se van a utilizar y, además, puede polucionar las caches del procesador, causando más accesos a memoria al reemplazar datos muy accedidos.

Trabajos recientes han propuesto nuevas arquitecturas para aliviar el *memory wall* alargando, virtualmente, la ventana de instrucciones. Lebeck y otros [4] propusieron el *Waiting Instruction Buffer* (WIB). Cuando se encuentra un load de larga latencia, dicha instrucción y sus correspondientes dependientes son extraídas de la ventana de instrucciones y ubicadas en el WIB hasta que se complete el acceso a memoria. De esta forma se liberan recursos que pueden ser utilizados por instrucciones independientes de los loads de larga latencia. Sin embargo, esta técnica necesita incrementar el número de registros del procesador para conseguir un rendimiento competitivo.

Los *kilo-instruction processors* [1] ocultan las latencias de memoria largas manteniendo miles de instrucciones en vuelo sin tener que incrementar las estructuras del procesador más allá de un tamaño razonable. Para ello, utilizan múltiples *checkpoints* para permitir que las instrucciones puedan retirarse del procesador fuera de orden y, al mismo tiempo, manejar de forma inteligente el resto de estructuras del procesador.

Con filosofía similar, la arquitectura *Continual Flow Pipelines* [16] propone un diseño sobre CPR [10] con estructuras escalables y multi-checkpoint para ejecutar instrucciones independientes de un load de larga latencia, guardando las instrucciones dependientes en una estructura auxiliar.

3. Caso de estudio: RunAhead

El mecanismo de RunAhead fue propuesto por primera vez por Dundas y Mudge [8] para mejorar el rendimiento de la cache de datos en un procesador en orden. Posteriormente, Mutlu y otros [14] lo adaptaron a procesadores con ejecución fuera de orden para aliviar el problema causado por las instrucciones de acceso a memoria con latencias largas.

La idea básica de RunAhead es realizar una ejecución especulativa de las instrucciones independientes a un load de larga latencia que, potencialmente, pueda bloquear el procesador debido a la falta de entradas en el reorder buffer. Su funcionamiento es bastante simple. En el momento que un load llega a ser la instrucción más antigua de la ventana de instrucciones, se realiza un *checkpoint* del estado actual del procesador para, posteriormente, una vez que los datos del load estén disponibles, recuperar el estado correcto del procesador. En este *checkpoint* se guarda el mapeo entre registros físicos y lógicos, el registro de historia del predictor de saltos, la *Return Address Stack* y la dirección de la instrucción de load. A continuación, el load de larga latencia se extrae de la ventana de instrucciones y su registro destino se marca como inválido, En este punto, el procesador entra en modo *RunAhead*.

En este modo, las instrucciones que son dependientes del load de larga latencia se eliminan del procesador. Para conocer qué instrucciones son dependientes de ese load, se modifica la tabla de renombramiento de registros, extendiendo cada entrada con un bit llamado INV. Las instrucciones propagan el valor del bit INV a su operando destino, haciendo un OR lógico con el valor de este bit correspondiente a sus operandos fuente. Las instrucciones que tengan un operando fuente, o ambos, con el valor INV, son eliminadas. De esta forma, se consigue impedir la ejecución de toda la cadena de dependencias del load que causó la entrada en modo RunAhead.

Para permitir que la ventana de instrucciones avance todavía más, también se eliminan las instrucciones con latencias largas junto con sus instrucciones dependientes. El resto de instrucciones son ejecutadas de forma especulativa. Es necesario resaltar que la mayor diferencia entre la ejecución especulativa y la no especulativa radica en que, en la primera, las instrucciones de store no modifican el contenido de la memoria.

En el momento que los datos del load de larga latencia están disponibles, el procesador sale del modo RunAhead. Para esto se restaura el estado del procesador en la instrucción de load que desencadenó el modo RunAhead, utilizando el *checkpoint* creado a tal efecto, y se descartan todas las instrucciones que se han ejecutado en ese modo.

Hay que tener en cuenta que, en realidad, no se elimina todo el trabajo realizado durante el modo RunAhead: los accesos a memoria realizados por las instrucciones de load se mantienen. Gracias a esto, durante el modo RunAhead se realiza prebúsqueda de datos para las instrucciones de load de larga latencia, incrementando el rendimiento del procesador.

Una gran ventaja de RunAhead frente a mecanismos clásicos de prebúsqueda es que consigue, mediante esta ejecución especulativa, adelantar los datos para cadenas dependientes de loads. Esta diferencia es notable en códigos donde se recorren listas enlazadas o estructuras de árboles.

No obstante, la principal ventaja de este mecanismo es que puede ser implementado con una pequeña cantidad de hardware adicional. Lo primero y más importante es incorporar el sistema de *checkpointing* para guardar la información necesaria para restaurar el estado del procesador. El tipo concreto de *checkpoint* dependerá de la microarquitectura del procesador. Por otro lado, también hace falta un mecanismo que impida a las instrucciones de store actualizar la memoria en commit durante el modo RunAhead. Para tal fin se puede utilizar un pequeño buffer que mantenga el estado de los stores, y a su vez permita proporcionar los datos correspondientes a los loads que aparezcan posteriormente en modo RunAhead. Los stores que fallan en el segundo nivel de cache, en modo RunAhead, se convierten en loads, es decir, se accede a la memoria para prebuscar datos, pero no los modifican. Por último, hace falta añadir el bit de invalidez (INV) a la tabla de renombrado que permita identificar cuando un registro es válido o inválido.

4. Metodología

Para la obtención de resultados hemos usado el simulador SMTsim, [20] para modelar un procesador con ejecución fuera de orden. El simulador incluye un sistema de memoria detallado, modelando aspectos importantes como las latencias de cache, contención del bus, conflictos entre bancos y retardos en la emisión de las peticiones de memoria.

El mecanismo de RunAhead ha sido implementado siguiendo [14]. La principal diferencia es que no incorporamos, en este estudio, ningún mecanismo de prebúsqueda hardware. De esta forma conseguimos aislar los beneficios del mecanismo de RunAhead.

La Tabla 1 muestra los parámetros de configuración del procesador para la configuración base. Algunos de estos parámetros se modificarán durante la evaluación.

Parámetros	*Configuración base*
Fetch Width	4
Funcional Units	6 INT / 3 FP/ 4 LD/ST
Registers	128 INT / 128 FP
ROB size	128
Issue Queues (IQ,FP,LQ)	32
Branch Predictor	Gshare (2K) + BTB (1K)
Memory System	
L1 Icache	64KB/2-way 64byte, 1 ciclo
L1 Dcache	64KB/2-way 64byte, 3 ciclos
L2 Cache	512KB/4-way 64byte, 10 ciclos
Mem Latency	500 ciclos
MSHRs	16

Tabla 1. Parámetros de configuración

Todos los experimentos fueron realizados utilizando los benchmarks de SPEC 2000 compilados con el compilador de Compaq/Alpha con todas las optimizaciones activadas. Para cada Spec se simularon 300 millones de instrucciones de una parte significativa del programa [2].

5. Evaluación

En esta sección mostraremos primero, cuan eficaz es el mecanismo de RunAhead incrementando el rendimiento y, en segundo lugar, su eficiencia ejecutando instrucciones.

5.1. Rendimiento

En las Figuras 1 y 2 se muestra el rendimiento (IPC) y el porcentaje de mejora para los programas SpecINT y SpecFP respectivamente, con los parámetros de la Tabla 1. Para cada Spec se muestra el IPC del procesador superscalar (baseline), del procesador superescalar con RunAhead (RA) y el procesador superescalar con memoria perfecta (bound), es decir, un procesador con una cache de nivel 2 con capacidad infinita, que utilizamos para mostrar el límite de la ganancia alcanzable.

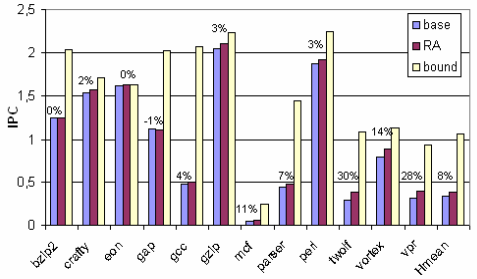

Figura 1. Rendimiento de RunAhead y SpeedUp para los programas SPEC INT 2000

En la Figura 1 se puede observar como la mejora en rendimiento con RunAhead no es muy significativa para SpecINT, aproximadamente 8%, excepto para los programas twolf (30%) y vpr (28%). Esto se debe a la gran cantidad de instrucciones inválidas que se encuentran en modo RunAhead (65%). Entre ellas, el 23% son loads, lo cual resta oportunidades a RunAhead para hacer prebúsqueda de valores.

La Figura 2 muestra los datos para los programas SpecFP. En este caso, el incremento de rendimiento obtenido es importante: 27%. Esto se debe a que el rendimiento de este conjunto de programas es más dependiente de la configuración de memoria, puesto que tienen un porcentaje mayor de fallos en L2 (36% y 62% para SpecINT y SpecFP respectivamente)., lo cual hace que la prebúsqueda de datos sea muy eficaz.

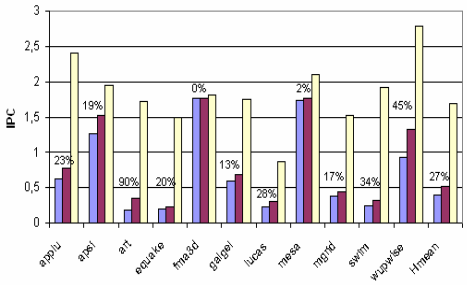

Figura 2. Rendimiento de RunAhead y SpeedUp para benchmarks SPEC FP 2000

5.2. Importancia de la ventana de instrucciones y de la latencia de memoria

Dos parámetros a tener en cuenta en el mecanismo de RunAhead son: el tamaño del reorder buffer y la latencia de memoria.

El número de entradas del reorder buffer tiene gran influencia en un procesador superescalar puesto que limita el número de instrucciones que se pueden mantener en vuelo durante un acceso de larga latencia a memoria y, por tanto, la cantidad de paralelismo a nivel de instrucción explotable. Con respecto a RunAhead, cuantas más entradas estén disponibles en el reorder buffer más se retrasará la puesta en marcha del mecanismo, ya que se tiene que esperar a que el load de larga latencia sea la instrucción más antigua y el reorder buffer esté lleno. Por otro lado, esto se puede ver contrarrestado con el aumento de paralelismo a nivel de instrucción, gracias a una ventana de instrucciones mayor.

La latencia de acceso a memoria afecta directamente al rendimiento de la configuración base, puesto que el tiempo de bloqueo del procesador es más largo al producirse un fallo de L2. En cuanto a RunAhead, una mayor latencia con memoria significa que estará mayor porcentaje de ciclos activado y, por tanto, serán ejecutadas más instrucciones en modo RunAhead.

La Figura 3 muestra el impacto del tamaño del reorder buffer y de la latencia de memoria. Para ello se muestran cuatro configuraciones: conf1 y conf2 (ambas con 128 entradas en el reorder buffer y 500 y 800 ciclos de latencia con memoria respectivamente), así como conf3 y conf4 (con 256 entradas y 500 y 800 ciclos de memoria respectivamente), en media y para SpecINT y SpecFP. Los otros parámetros de configuración no se han modificado respecto a los de la Tabla 1 excepto el tamaño de las colas de emisión, que han sido escaladas en igual proporción que el reorder buffer.

Como se puede ver en la Figura 3, modificando el tamaño del reorder buffer de 128 a 256 entradas y manteniendo la latencia con memoria (por un lado conf1 y conf3 con 500 ciclos y por el otro, conf2 y conf4, con 800 ciclos) se obtiene un aumento del incremento del rendimiento del 1,5% comparando conf1 y conf3 y 3% comparando conf2 y conf4. Esto es debido, como se comentó previamente, por que, aunque en la configuración con más entradas en el reorder buffer, las veces que se activa el mecanismo de RunAhead es menor (131.700 veces comparando conf1 y conf3, y 138.200 veces comparando conf2 con conf4), el aumento del paralelismo contrarresta el efecto (0,12 para conf1-conf3 y 0,09 para conf2-conf4 instrucciones por ciclo más son emitidas).

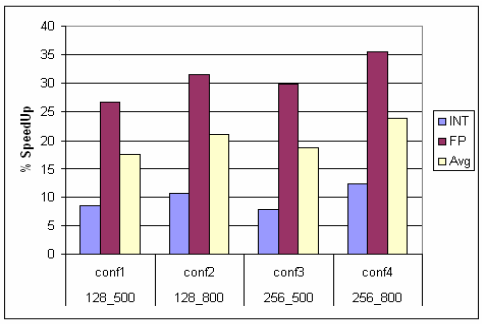

Figura 3. SpeedUp para distintas configuraciones de ROB y latencia de memoria

Por otro lado, manteniendo el número de entradas en el reorder buffer y variando la latencia de memoria de 500 a 800 ciclos el incremento de rendimiento obtenido por RunAhead es 3,5% comparando conf2 con conf1 y 5% comparando conf4 con conf3. Esto se debe a que el mecanismo está en funcionamiento un 3,8% más del tiempo de ejecución para el primer caso y 4,2% para el segundo.

La Figura 4 muestra el motivo por el cual RunAhead es capaz de incrementar el rendimiento: la prebúsqueda. Debido a la limitación de espacio, solamente se muestran el porcentaje de fallos en el segundo nivel de cache, para la configuración conf3 (128 entradas del ROB y la latencia de memoria de 800 ciclos).

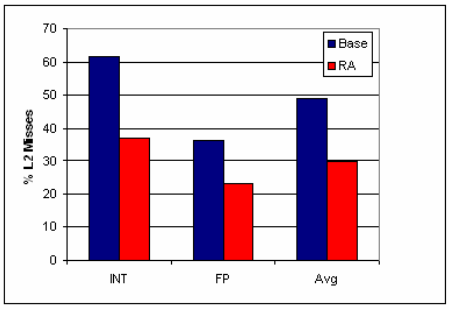

Figura 4. Porcentaje de fallos en la cache L2

En la Figura 4 se demuestra que existe una clara reducción en los accesos no especulativos a memoria. Del 49% de accesos de larga latencia que se ejecutan en el procesador superescalar sin RunAhead, el 19% han sido correctamente prebúscados en modo RunAhead.

5.3. Número de Instrucciones

Hasta este punto se ha presentado una evaluación de la eficacia del mecanismo de RunAhead, con diversas configuraciones de tamaño de reorder buffer y de latencia de memoria, mostrando que esta técnica incrementa, eficazmente, el rendimiento de un procesador superescalar. En esta sección se evalúa el número de instrucciones extra que se tienen que ejecutar para alcanzar tales incrementos en rendimiento y que serán la base para calcular la eficiencia del mecanismo en la siguiente sección.

Las Figuras 5 y 6 muestran el número de instrucciones extra ejecutadas debido a fallos de predicción de saltos (wrong path), y a la ejecución especulativa en modo RunAhead (RA) con respecto al total de instrucciones útiles ejecutadas (useful), para la configuración con 256 entradas en el reorder buffer y 800 ciclos de latencia de memoria. Para SpecINT (Figura 5) se observa que casi la mitad de instrucciones que se ejecutan, han sido creadas en modo RunAhead: 45%.

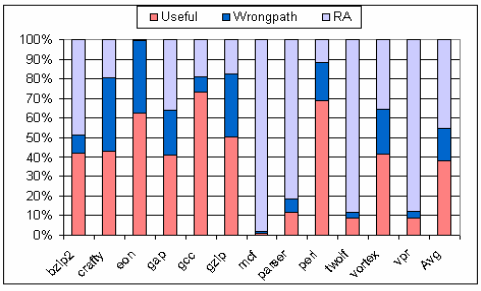

Figura 5. Instrucciones extras en RA para SpecINT

Para SpecFP (Figura 6), el porcentaje de instrucciones extra en modo RunAhead es sensiblemente menor: 32%. La disparidad de porcentajes entre SpecINT y SpecFP se debe a dos motivos: el primero es el porcentaje de tiempo que el procesador permanece en modo RunAhead (46% para SpecINT y 58% para SpecFP). El segundo motivo es el número de instrucciones especulativas que se ejecutan por período de RunAhead (2605 para SpecINT y 682 para SpecFP). Aunque el número de veces que se entra en modo RunAhead en SpecINT es sensiblemente inferior comparado con SpecFP, siempre se llega más lejos en la ventana de instrucciones en los SpecINT que en los SpecFP, lo que hace que, en proporción se ejecuten más instrucciones especulativas para SpecINT que para SpecFP.

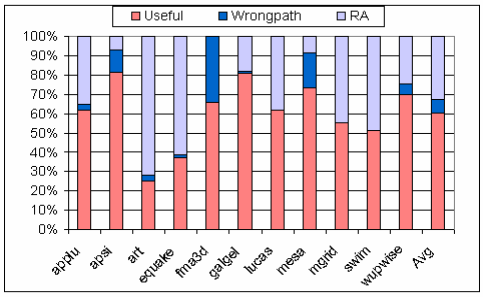

Figura 6. Instrucciones extras en RA para SpecFP.

En la Tabla 2 se muestra el número de instrucciones totales buscadas durante la ejecución normal (TotalFetch), el número total de instrucciones buscadas durante la ejecución en modo RA (FetchedRA), y de estas cuantas han hecho commit (Pseudo-Commit). Como se puede ver, el número de instrucciones tanto en modo normal como en modo RunAhead es claramente superior para SpecINT (3573 millones y 3011 millones) que para SpecFP (570 millones y 224 millones). De entre las instrucciones traídas en modo RunAhead, 2408 millones (67%) llegan a completar su ejecución en SpecINT. Para SpecFP, solamente 108 millones (39%) de instrucciones especulativas consiguen completarse. Esto se debe a que las instrucciones de coma flotante tienen, normalmente, una latencia mayor que las enteras, lo que provoca una saturación mayor de las unidades de coma flotante que puede causar que el reorder buffer se llene totalmente.

Stat	*INT*	*FP*
TotalFetch	3.573.954.281	570.334.978
FetchedRA	3.011.686.360	224.325.965
Pseudo-commit	2.408.649.828	108.743.847
% Valid	35%	66%
% INV	65%	34%

Tabla 2. Estadísticas de instrucciones en RA

Por último, la Tabla 2 también muestra el porcentaje de instrucciones válidas (%Valid) y el de instrucciones que son eliminadas del procesador (%INV) por que dependen directa o indirectamente del load que causó la entrada en modo RunAhead. El porcentaje de instrucciones válidas en SpecFP es del 66% mientras que para enteros es del 35%. Este mayor porcentaje nos indica que para SpecFP hay más posibilidades de efectuar prebúsquedas válidas, puesto que se consigue calcular más direcciones efectivas de loads. Como consecuencia, se obtiene un mejor beneficio de RA, y por tanto, una mayor ganancia de rendimiento en SpecFP que en SpecINT.

5.4. Eficiencia

Por último, en esta sección evaluamos si el mecanismo de RunAhead es eficiente ejecutando instrucciones. Para medir la eficiencia de los mecanismos, normalmente se utiliza la métrica Energy-Delay^2 [15] que relaciona el consumo de un procesador con su rendimiento. En nuestro caso, mediremos el consumo en número de instrucciones ejecutadas. Aunque el consumo depende de la clase de instrucción ejecutada supondremos, para simplificar el estudio, que todas las instrucciones consumen la misma cantidad de energía.

Por otro lado, el retardo (delay^2) lo mediremos en CPI^2 medio. Por tanto, la fórmula que emplearemos será:

$$ED^2 = \#Instrucciones\text{-}ejecutadas \times CPI^2$$

Mediante esta fórmula compararemos cuantas *unidades* de energía se consumen por instrucción ejecutada. De esta forma, nos dará una aproximación de cuan eficiente, en términos de energía y consumo, es un procesador con RunAhead ejecutando instrucciones.

Las Figuras 7 y 8 muestran los valores de ED^2 del mecanismo de RunAhead normalizados al procesador superescalar sin este mecanismo para una configuración con un reorder buffer de 256 entradas y 800 ciclos de latencia con memoria. En estas Figuras los valores por encima de 1 significan que se consume más energía por instrucción ejecutada que en la configuración base.

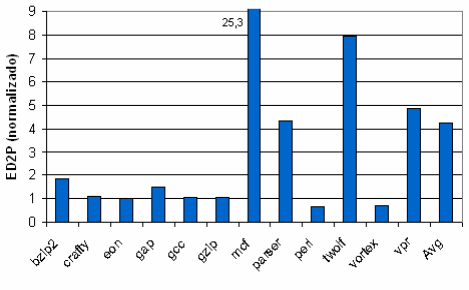

Figura 7. Energy-Delay^2 SpecINT

Para SpecINT se observa claramente que el mecanismo es poco eficiente ejecutando instrucciones. En media, se necesita 4 veces más energía para ejecutar una instrucción en comparación con la configuración base.

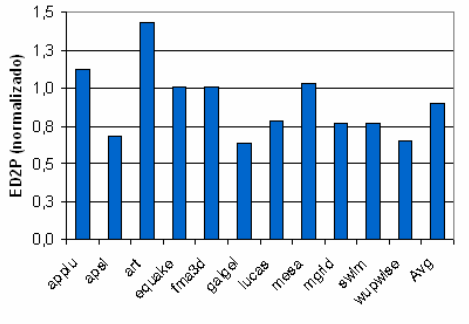

Figura 8. Energy-Delay^2 SpecFP

Por el contrario, para los SpecFP, el mecanismo es más eficiente, obteniendo 0,9 ED^2. Esto significa que, en promedio, se necesita consumir menos energía por instrucción retirada en el procesador con el mecanismo de RunAhead que sin este mecanismo.

Por tanto, en determinados escenarios, RunAhead puede ser al mismo tiempo eficiente y eficaz. Sin embargo, no hay que olvidar que RunAhead está lejos de ser eficaz en otros escenarios. Por tanto, la eficacia de esta técnica es un dato muy importante que debe ser analizado y tenido en cuenta por los diseñadores de procesadores.

6. Conclusiones

El consumo de energía, una característica no contemplada en la evaluación de muchos mecanismos propuestos en la literatura, empieza a ser, cada vez más, un factor limitante en los procesadores actuales. Un incremento importante en el rendimiento de las aplicaciones, no tiene por qué justificar la inclusión de un mecanismo en el diseño de un procesador, si esto conlleva un incremento en el consumo de energía.

Como caso práctico, en este artículo, hemos evaluado en términos de rendimiento y eficiencia una técnica para aliviar el problema de las largas latencias causada por los accesos a memoria: RunAhead. Este mecanismo incrementa un 24% el rendimiento de un procesador con 256 entradas en el reorder buffer y 800 ciclos de latencia con memoria, gracias a la prebúsqueda especulativa de datos para instrucciones de larga latencia. Como contrapartida, la ejecución de instrucciones especulativas aumenta el consumo de energía del procesador.

En el caso de SpecINT, para obtener un incremento medio de rendimiento del 12% se necesita aproximadamente 4 veces más energía por instrucción que en el procesador sin este mecanismo. Esto lleva a la conclusión de que el rendimiento obtenido podría no ser justificable, en función de las características y objetivos del procesador, debido a la gran cantidad de instrucciones extra que tienen que ser ejecutadas.

Agradecimientos

Este trabajo ha estado apoyado por el Ministerio de Educación y Cultura de España bajo el contrato TIN2004-07739-C02-01, la Red Europea de Excelencia HiPEAC, y el BSC-Centro Nacional de Supercomputación.

Referencias

[1] A. Cristal, D. Ortega, J. Llosa and Mateo Valeo. *Out-of-Order commit processors*. HPCA-10, Febrero 2004.

[2] A. Falcón et al. *Selecting Where to Simulate SPEC2000 using stream analysis.* XV Jornadas de Paralelismo, Almeria, España.

[3] Alan J. Smith. *Cache Memories*. ACM Computing Surveys, Septiembre 1982.

[4] A. R. Lebeck et al. *A large, fast instruction window for tolerating cache misses*. 29th ISCA-2002.

[5] C-K Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. 28th ISCA- 01.

[6] D. Callahan et al. *Software prefetching*. 4th ASPLOS, 1991.

[7] D. Joseph and D. Grunwald. *Prefetching using Markov predictors*. 24th ISCA-97

[8] J. Dundas and T. Mudge. *Improving data cache performance by pre-executing instructions under a cache miss*. ICS, Julio 1997.

[9] J.D. Collins et al., "Dynamic Speculative Precomputation". *34th International Symposium on Microarchitecture* (Micro-34), 2001..

[10] H. Akkary, R. Rajwar and S.T. Srinivasan. *Checkpoint processing and recovery*: Towards scalable large instruction window processors. MICRO-36, 2003.

[11] M. Dubois and Y. Song, *Assisted Execution*, technical report CENG 98-25, Dept. EE-Systems, Univ. Southern California, 1998.

[12] M. Wilkes. *Slave memories and dynamic storage allocation*. IEEE Transactions on Electronic Computers, 1965.

[13] Norman P. Jouppi. *Improving direct-mapped cache performance by the addition of small fully-associative cache and prefeth buffers.* (ISCA 90).

[14] O. Mutlu, J. Stark, C. Wilkerson and Y. Patt. RunAhead execution: *An alternative to very large instruction windows for out-of order processors*. HPCA-09, Febrero 2003.

[15] R. Gonzalez and M. Horowitz, *Energy Dissipation in General Purpose Microprocessors*, IEEE J. Solid-State Circuits, Vol. 31, No. 9, Sept. 1996.

[16] S.T. Srinivasan et al. *Continual Flow Pipelines*. ASPLOS, 2004.

[17] T.C. Mowry et al. *Design and evaluation of a compiler algorithm for prefetching*. 5th International, ASPLOS 1992.

[18] T. Sherwood et al. *Predictor-Directed Stream Buffers*. 33rd ACM/IEEE International Symposium on Microarchitecture, MICRO-00

[19] W.A. Wulf and S.A. McKee, *Hitting the Memory Wall: Implications of the Obvious.* ACM SIGARCH Computer Architecture News, vol. 23, no. 1, March 1995, pp. 20-24.

[20] D.M. Tullsen. *Simulation and Modeling of a Simultaneous Multithreading Processor.*, 22nd Annual Computer Measurement Group Conference, December, 1996