Designing a Prefetcher for a Hardware Distributed Shared-Memory System

Ferran Pérez [1] , Yana E. Krasteva[1], Federico Silla[1], Holger Fröning [2] y José Duato[1]

Resumen— **El modelo de programación de memoria compartida presenta varias ventajas sobre el paradigma de paso de mensajes. De la misma manera, los sistemas de memoria compartida superan muchos de los problemas presentes en implementaciones de memoria distribuida no compartida, tales como clústers. Debido a esto, varios fabricantes de sistemas, como por ejemplo SGI, ofrecen soluciones de de memoria compartida distribuida de gran escala que agregan los recursos de memoria y CPU presentes en un clúster en una única imagen de sistema. Otros enfoques, como el recientemente propuesto MEMSCALE, proporcionan características similares con propiedades de escalabilidad mucho mejores.**

Sin embargo, en todos estos sistemas de memoria compartida distribuida, los accesos a memoria remota son penalizados debido a su latencia de acceso más alta respecto a la memoria local del nodo y, como resultado, el rendimiento de las aplicaciones se degrada. Con el fin de solucionar este problema, el bien conocido mecanismo de prefetching puede ser aprovechado. En este artículo presentamos un análisis exhaustivo de las técnicas más importantes de prefetching, centrándonos en su aplicabilidad a sistemas con fuertes características NUMA como arquitecturas de memoria compartida distribuida. Por otra parte, se proponen dos pequeñas mejoras a la técnica revisada. Basándose en la complejidad de implementación de estas técnicas de prefetching, hemos seleccionado los más adecuados y hemos realizado un análisis en profundidad de su impacto en el rendimiento de las aplicaciones. La principal contribución de este trabajo es que las aplicaciones pueden reducir su tiempo de ejecución en un 25% si el mecanismo de prefetching adecuado está configurado correctamente.

Palabras clave— **Prefetching, Distributed shared memory systems, FPGA**

I. Introducción

ACTUALMENTE , los computadores de altas prestaciones y centros de datos por lo general aprovechan ordenadores basados en x86 organizados en clúster. Realmente, la mayoría de los sistemas listados en el TOP500 son clústers basados en x86 [1].

La principal razón para elegir la arquitectura clúster para estas instalaciones grandes es simplemente el coste. Debido a los grandes volúmenes de fabricación han hecho que estos clústers sean la opción más barata para conseguir una gran potencia de cálculo.

Los clústers, sin embargo, tienen sus inconvenientes. Uno de ellos es que la memoria esta repartida entre los distintos nodos, por lo que para compartir información se deben enviar mensajes, normalment con la librería MPI, que introduce una sobrecarga no despreciable y tiene un modelo de programación mas complejo que el sistema de memoria compartida.

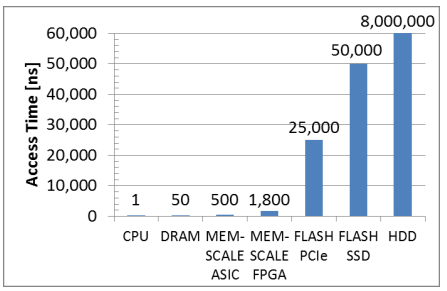

Fig. 1. Disparidad de latencias entre tecnologías de almacenamiento comunes y MEMSCALE

Para obtener una visión de memoria compartida en un clúster existen varias soluciones comerciales hardware como Altix UV de SGI [2] y el NumaChip de Numascale [3] , que proporcionan un sistema de memoria distribuida compartida (DSM) coherente del clúster. Nosotros hemos propuesto MEMSCALE[4], una nueva arquitectura hardware que proporciona memoria compartida al clúster con enfoque muy diferente. Sin embargo, todas estas propuestas tienen una alta latencia para acceder a memoria remota debido a que está ubicada en otros nodos, como se muestra en la Figura 1.

Para reducir esta latencia a memoria remota proponemos utilizar técnicas de prefetching, que pidan por adelantado el contenido de las posiciones de memoria que es probable que vayan a ser accedidas. De esta forma, cuando se accedan ya estarán en el nodo local y sufrirán una latencia mucho menor a la de acceder al nodo remoto.

En este artículo analizamos el impacto de varias técnicas de prefetching, mostrando su efecto de reducir la latencia de la memoria remota. Cabe destacar que aunque las técnicas de prefetching llevan tiempo estudiándose en distintos escenarios, éste es el primer análisis en un DSM hardware formado por ordenadores x86.

El resto del artículo se estructura en las siguientes partes: la sección siguiente proporciona los antecedentes a este trabajo. para que el lector pueda poner en contexto el análisis presentado. La Sección III resume la arquitectura MEMSCALE, sobre la que se realizará el estudio. Posteriormente, en la Sección IV introducimos el trabajo anterior relacionado con el análisis llevado a cabo. Después, en las Secciones V y VI se mostrarán las diferentes técnicas de prefetching y sus prestaciones. Por último, en la Sección VII presentamos las conclusiones finales del estudio.

II. Antecedentes

En el contexto de clústers de memoria compartida, las técnicas de prefetching se han analizado para DSMs software [5][6][7], que tienen una granularidad

[1]Departament d'Informàtica de Sistemes i Computadors, Universitat Politècnica de València, Spain, Email: {ferpelo,yana.krasteva}@gap.upv.es {fsilla,jduato}@disca.upv.es

[2]Computer Architecture Group, University of Heidelberg, Germany, Email: froening@uni-hd.de

de páginas de memoria. En cambio, los DSM hardware tienen una granularidad de bloques de cache.

Las implementaciones de DSM hardware comerciales implementan un protocolo de coherencia entre los nodos a parte del existente en cada nodo. Este protocolo aumenta la latencia debido la sobrecarga de la propagación de las operaciones de coherencia y a la invalidación bloques por el protocolo.

Nuestro prefetcher funcionará en DSM sin protocolo de coherencia entre nodos. Esto eliminará interferencias entre el protocolo y el prefetcher. Por lo tanto, las aplicaciones se verán restringidas a utilizar los núcleos de un solo nodo, aunque sí podrán utilizar memoria de otros nodos.

Sin embargo, ¿es realista restringir la ejecución de una aplicación paralela de memoria compartida a un solo nodo? Puede que no para todas las aplicaciones pero es un hecho que la mayoría de las aplicaciones paralelas no escalan más allá de los cores existentes en una placa base [8]. Esta cuestión se ve apoyada por los últimos desarrollos de AMD [9] e Intel [10], que aumentan el número de cores a 64 y 80 respectivamente. Sin embargo, mucha aplicaciones paralelas sí agotan la memoria presente en un nodo. Esto se ve reforzado por la tendencia a disminuir la memoria disponible para cada core [11] debido al aumento más rápido del número de cores.

Debido a esta disociación hemos presentado recientemente la nueva arquitectura de memoria compartida distribuida no coherente.

III. La arquitectura MEMSCALE

La tecnología MEMSCALE se basa en compartir la memoria entre los nodos sin degradar el rendimiento con un protocolo de coherencia. Para ello se configuran los nodos del clúster como puede verse en la Figura 2. En esta figura se muestra como cada nodo tiene asignados los procesadores presentes en su placa base y parte de la memoria, pero el resto de la memoria puede repartirse entre los nodos dependiendo de las necesidades de las aplicaciones que estén ejecutándose en el clúster. Ya que en cada nodo está mapeada la memoria remota del resto de los nodos.

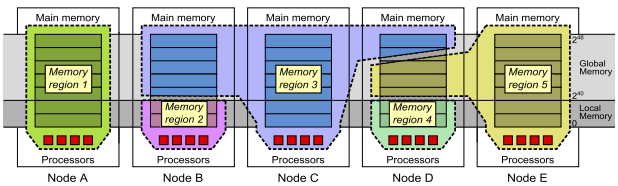

Fig. 2. Un ejemplo de compartición de memoria entre los nodos de un clúster

MEMSCALE se basa en instalar en cada nodo un controlador de memoria remoto (RMC) como se muestra en la figura 3. El RMC es una tarjeta que se conecta como un periférico con tanta memoria de I/O como haya disponible en el clúster. Cuando un nodo necesita utilizar memoria remota, envía una petición a la tarjeta y esta la reenvía al nodo correspondiente por una red dedicada. El nodo remoto completa la petición y si es necesario devuelve una respuesta.

IV. Trabajos relacionados

Aunque existen un gran número de técnicas de prefetching, solo una parte puede aplicarse a MEMSCALE. El requerimiento que ha limitado más la elección del prefetcher ha sido que el controlador de memoria remota está fuera del procesador y no tiene acceso a información imprescindible para el funcionamiento de muchos prefetchers, como qué instrucción ha provocado cada lectura [13] . La capacidad de preejecutar instrucciones para detectar futuros accesos [14] o la polución creada por el prefetecher en la cache del procesador [15].

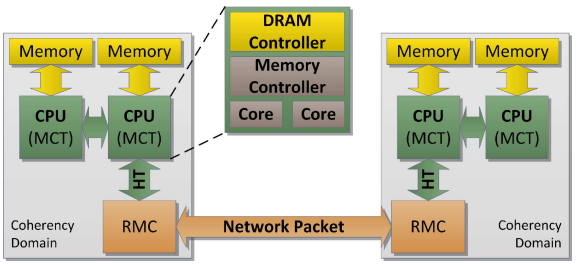

Fig. 3. Reenviando peticiones de load/store a través de la red

Existen varias técnicas de prefetching para DSM software [5] [7], pero ninguna es aplicable a MEMSCALE.

Otros estudios [16] se basan en el compilador para que inserte instrucciones de prefetch (loads no bloqueantes). Esta técnica se ha utilizado en DSMs hardware en [17], pero no la probaremos en MEMSCALE ya que una de las características de la misma es que los programas no necesitan recompilarse debido a que MEMSCALE es completamente transparente para ellos.

Otras propuestas utilizan tablas enlazadas para almacenar patrones de acceso complejos, [18]. Estas propuestas se han descartado debido a que su complejidad es demasiado alta para ser implementada en nuestras FPGAs.

Hay varias técnicas de prefetching que sí se pueden aplicar a MEMSCALE, como el prefetcher secuencial [19], los stream buffers [20] o el secuencial adaptativo [21].

V. Los prefetchers estudiados

En la sección anterior hemos realizado una revisión del trabajo anterior sobre prefetchers, eliminando los que no son apropiados para un DSM que utiliza un RMC, como MEMSCALE.

En esta sección presentamos los distintos prefetchers que hemos seleccionado para ser analizados.

A. Prefetcher Secuencial

El prefetcher secuencial es el más simple. Junto a cada acceso a memoria envía la petición de los N bloques siguientes (siendo N el grado del prefetcher). Debido a la alta latencia de la memoria remota y para evitar que se envíen lecturas a una dirección de memoria remota a la que ya se ha enviado una lectura, se ha implementado el *FetchingBit*, un bit asociado a cada bloque de la cache del RMC y que indica que el bloque se ha pedido pero aun no ha llegado.

B. Prefetcher Adaptativo

Este prefetcher es una ampliación del anterior, donde el grado de prefetch se cambia dinámicamente dependiendo de lo útiles que sean los bloques que

está pidiendo el prefetcher en cada momento. Necesita las siguientes estructuras adicionales:

- **PrefetchBit:** un bit asociado a cada bloque de la cache que se activa cuando se recibe un bloque del prefetcher y se desactiva la primera vez que se lee.
- **PrefetchDegree:** un registro con el grado actual del prefetcher
- **UsefulCounter:** cuenta los PrefetchBit que se desactivan, es decir, cuantos bloques traídos por el prefecher se han usado.
- **PrefetchCounter:** cuenta cuantos bloques carga el prefetcher. Cuando llega a un límite, comprueba el UsefulCounter y dependiendo de su valor actualiza el PrefetchDegree. Después borra el PrefetchCounter y el UsefulCounter.

C. Prefetcher Stream Buffer

El stream buffer es más complejo que los prefetchers anteriores. Está formado por colas FIFO dedicadas que almacenan los próximos accesos de los flujos de datos detectados. Para poder detectar los flujos de datos, se guarda un historial con las direcciones accedidas y detecta los flujos a partir de las distancias entre los nuevos accesos y los almacenados en el historial. Cuando detecta un flujo, reserva un stream buffer para ese flujo, que almacena en una cola FIFO de capacidad N los próximos accesos del flujo detectado, cuando se consume un bloque, se carga el bloque N en la FIFO.

VI. Resultados Experimentales y Análisis

Para analizar los beneficios obtenidos por cada uno de los prefetchers en nuestro DSM hardware, hemos utilizado un simulador propio. En las secciones siguientes describimos el banco de pruebas para la simulación y realizamos un análisis de los principales resultados obtenidos.

A. Entorno de Simulación

Para comparar los diferentes prefetchers se ha utilizado un simulador basado en eventos. El simulador lee trazas que contienen las direcciones de memoria accedidas durante la ejecución de programas reales. Estos programas son los benchmarks PARSEC y TPC-H. El benchmark PARSEC se ha ejecutado para cuatro, ocho y dieciseís hilos, mientras que el TPC-H se ha ejecutado para cuatro hilos solamente. Para generar las trazas se ha utilizado la herramienta de simulación Simics. Para cada una de las ejecuciones, se recogieron varios archivos de traza. Cada uno de ellos contiene el flujo de accesos de uno de los núcleos involucrados.

B. Análisis

En esta sección analizamos los resultados obtenidos con el simulador. Dada la gran cantidad de configuraciones evaluadas, y con el fin de ajustar las leyendas de las gráficas de un espacio razonable, los nombres de las configuraciones de los prefetchers simulados se han abreviado como sigue:

- **Sp:** Sequential Prefetcher. Prefetcher secuencial.
- **Fb:** El FetchingBit se está usando. En el prefetcher adaptativo siempre se usa.
- **D:** Degree. Grado del prefetcher secuencial. Cuando se usan grados negativos se indica con *Dgrado_positivo-grado_negativo*.
- **S:** Size. Tamaño en KB de la cache privada del RMC. Por defecto tiene 512KB.
- **Ap:** Adaptive Prefetcher. Prefetcher adaptativo.
- **C:** Contador. Umbral del PrefetchCounter para desencadenar la actualización del PrefetchDegree.
- **L:** Lower. Un UsefulCounter más bajo o igual a este valor causa un decremento en el PrefetchDegree.
- **H:** Higher. Un UsefulCounter más alto o igual a este valor causa un incremento en el PrefetchDegree.
- **I:** Increment. Cuanto aumenta elPrefetchDegre cada vez que se incrementa.
- **str:** STReam buffer. El siguiente número indica el número de stream buffers.
- **Nc:** Not Confirmed. Los stream buffers son reemplazados sin esperar a que se confirmen los posibles flujos.
- **SmCPU:** Small CPU cache. La cache L3 se reduce de 2MB asociativa de 32 vías (configuración por defecto) a 512KB asociativa de 8 vías.
- **Om:** Only on Miss. El prefetcher solo se activa cuando hay un fallo en la cache del RMC.

Por otro lado, en las gráficas siguientes, los diferentes benchmarks ejecutados se conocen como *nombre_número*, donde nombre es una abreviación del benchmark real: bla: blackscholes (M denota el tamaño medio del problema mientras S denota el tamaño pequeño del problema), bod: bodytrack, can: canneal, fer: ferret, flu: fluidanimate, fre: freqmine, ray: raytrace, str: streamcluster, swa: swaptions, vip: vips, tcph1: TPC-H) y número se refiere a la cantidad de núcleos implicados en la ejecución (4, 8 ó 16). También se ha añadido la media de cada prueba, etiquetada como *Average*.

Para permitir una comparación homogénea, cada una de las ejecuciones del simulador representadas en las siguientes gráficas se ha normalizado al caso que tiene un mayor tiempo de ejecución para esa gráfica en particular, que suele ser el caso de no utilizar ningún Prefetcher. Este caso se ha etiquetado como *noPref*, y representa la cota máxima del tiempo de ejecución. Además, se ha incluido una cota inferior con el *perfectPref*, el cual siempre tiene los bloques en la cache del RMC.

En la Figura 4 se muestran los tiempos de ejecución de prefetcher secuencial. Como puede verse, el prefetcher más simple (SpD01) resulta en un incremento significativo de prestaciones (casi un 20% en la media). Además, hay un punto (alrededor de $D22$) a partir del cual las mejoras de incrementar el grado son insignificantes.

A partir de este momento, debido a la gran cantidad de casos distintos simulados, y para mejorar la legibilidad de las figuras, solo se mostrarán las configuraciones con cuatro threads para cada benchmark. Téngase en cuenta que la diferencia con las configuraciones de 8 y 16 hilos son muy pequeñas en

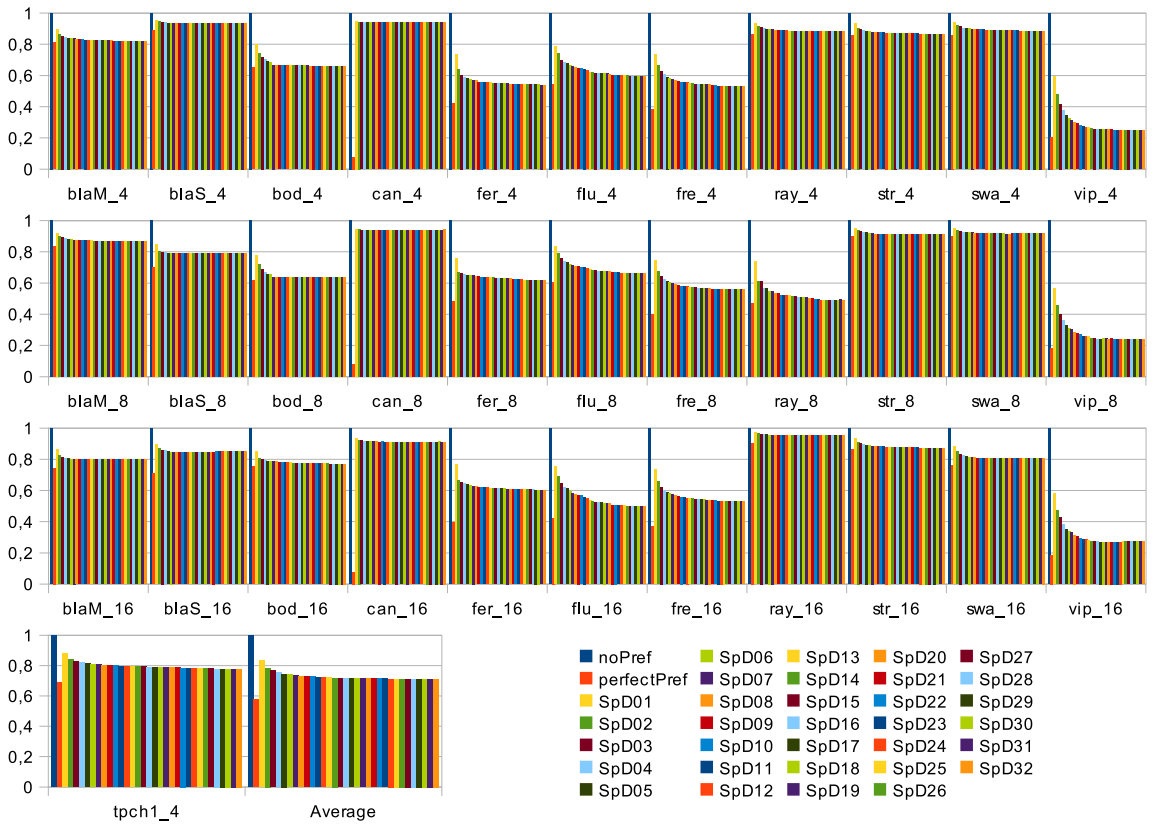

Fig. 4. Tiempo de ejecución normalizado para el prefetcher secuencial variando el grado de prefetch

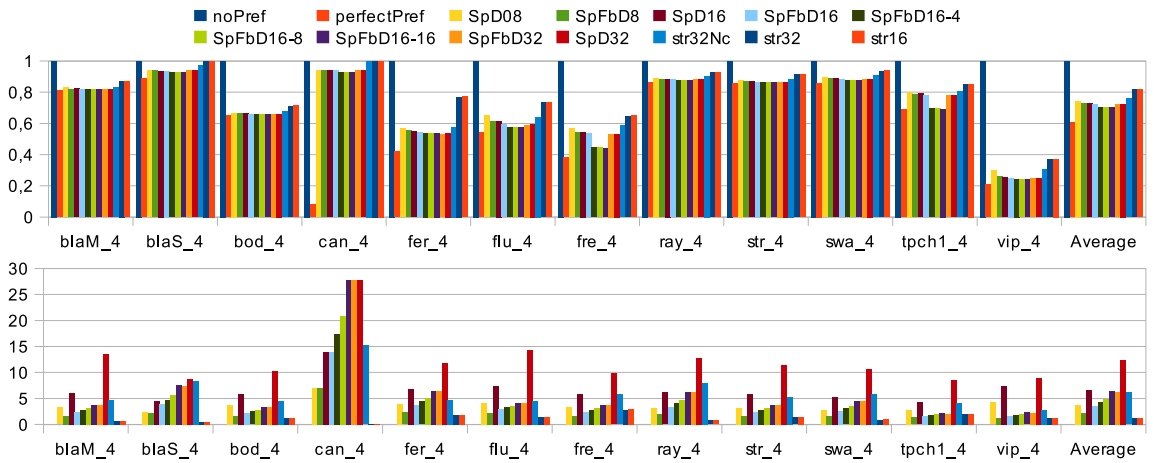

Fig. 5. Resultados reunidos usando el prefetcher secuencial mejorado (usando FetchingBit y grados negativos) y el stream buffer. Gráfica superior: tiempo de ejecución normalizado. Gráfica inferior: cantidad media de bloques traídos por acceso al RMC

la mayoría de los casos.

En la Figura 5, los efectos de las mejoras sugeridas para el prefetcher secuencial y los resultados más significativos de los stream buffers son mostrados. Las mejoras que se han sido la inclusión del FetchingBit y el uso de grados negativos. Téngase en cuenta que estas dos mejoras se proponen en este documento por primera vez.

Como puede verse en la figura, algunos benchmarks, como ferret o TPCH1, presentan una mejora significativa cuando se usan los grados negativos debido a la gran cantidad de accesos secuenciales de índices altos a índices bajos de vectores que presentan en su código. De hecho el uso de índices negativos tiene el menor tiempo de ejecución, incluso con un índice tan pequeño como 4. También puede verse que *FetchingBit* mejora significativamente el número de bloques traídos. En cambio los stream buffers, aunque tienen un tráfico bajo, su rendimiento es significativamente menor al secuencial.

La Figura 6 muestra las prestaciones con el prefetcher adaptativo. El tiempo de ejecución no tiene una gran variabilidad, pero el número de bloques traídos sí que varia significativamente. Los factores L y H son los que más afectan a las prestaciones. Valores bajos de estos parámetros resultan en un grado de prefetch alto que provoca un mayor tráfico en la red. Aumentar estos parámetros disminuye el número de bloques traídos, pero si se aumentan demasiado el tiempo de ejecución aumenta. El parámetro I se introdujo para intentar que el grado aumentara más rápidamente después de un periodo de baja utilidad del prefetcher, pero no tiene un

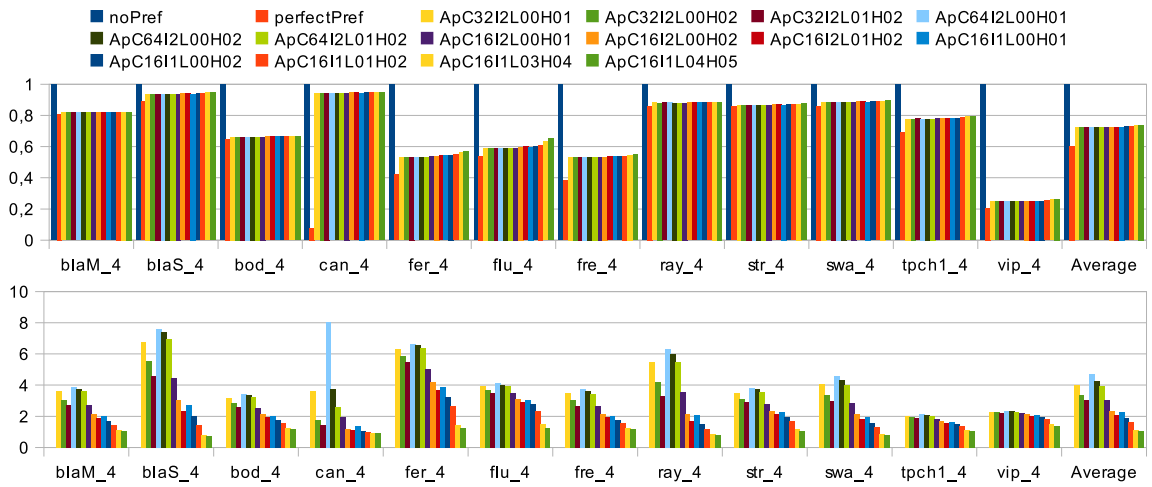

Fig. 6. Resultados de prestaciones cuando se usa el prefetcher adaptativo. Gráfica superior: tiempo de ejecución normalizado. Gráfica inferior: cantidad media de bloques traídos por acceso al RMC

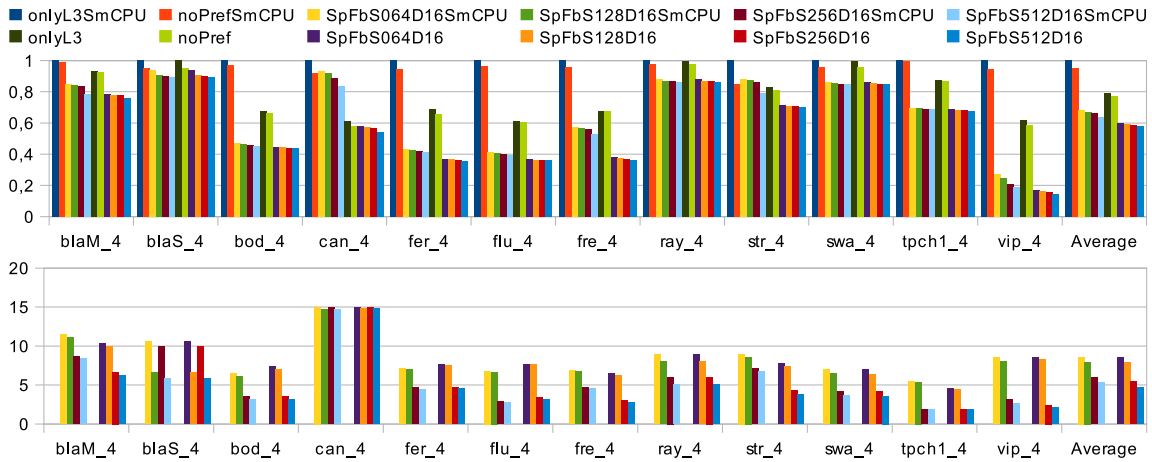

Fig. 7. Efecto del tamaño de las caches en el procesador y el RMC. La cache de la CPU tiene un tamaño de 2MB o 512KB (indicado con *Sm*). El tamaño del la cache privada del RMC varia entre 64, 128, 256 o 512KB. Gráfica superior: tiempo de ejecución normalizado. Gráfica inferior: cantidad media de bloques traídos por acceso al RMC

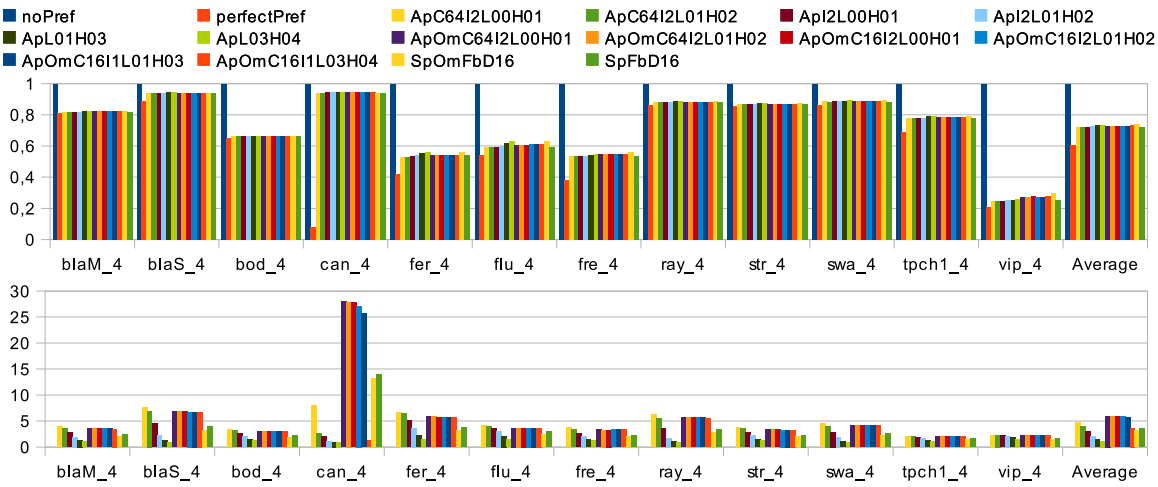

Fig. 8. Efecto de activar el prefetcher solo en los fallos de cache (*Om*) o en todos los accesos a memoria remota (opción por defecto) para las mejores configuraciones anteriores, Gráfica superior: tiempo de ejecución normalizado. Gráfica inferior: cantidad media de bloques traídos por acceso al RMC

efecto significativo y aumenta el número de bloques traídos. Un valor alto de C permite que el prefetcher tenga en cuenta más bloques para calcular el nuevo grado, haciendo modificaciones más precisas, pero si se incrementa mucho reduce el grado de recuperación a las adaptaciones aumentado el tiempo de ejecución.

La Figura 7 muestra las prestaciones del prefetcher secuencial cuando se varía el tamaño de las cachés del procesador y RMC. Las pruebas marcadas como *OnlyL3* se corresponden con configuraciones sin caché ni prefetcher en el RMC, solo la cache del procesador. Como puede verse, aunque el tamaño de la caché del procesador tiene un efecto significativo si se desactiva el prefetcher, cuando el prefetcher está activado el tamaño de las dos cachés es poco significativo para el rendimiento del mismo.

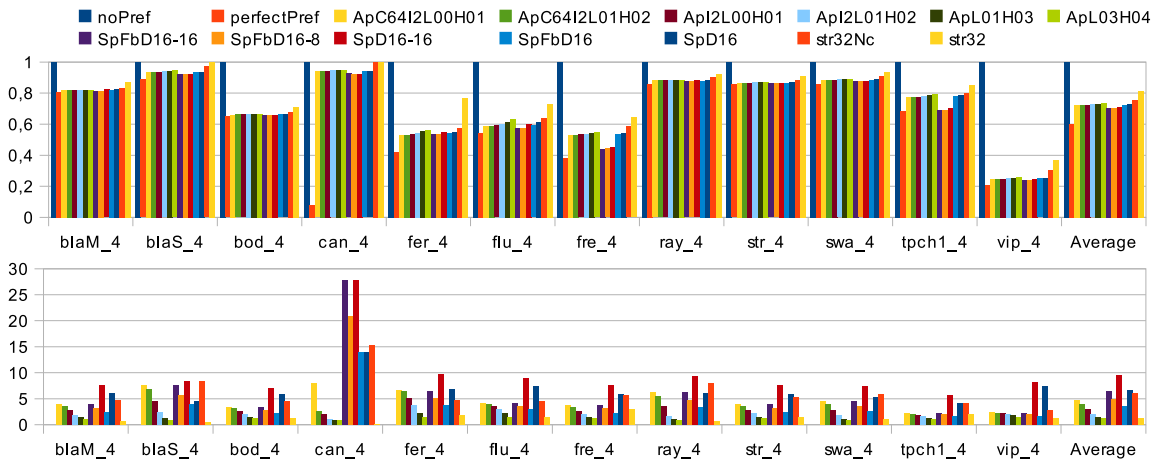

Fig. 9. Comparación entre las mejores configuraciones de prefetchers. Gráfica superior: tiempo de ejecución normalizado. Gráfica inferior: cantidad media de bloques traídos por acceso al RMC

La Figura 8 muestra los efectos de activar el mecanismo prefetch solo cuando se produce un fallo en la cache del RMC (Om) o en todos los accesos (comportamiento por defecto en este estudio). Como puede verse, el prefetcher secuencial (*SpOmFbD16*), tiene un tiempo de ejecución mayor debido a los fallos de caché necesarios para que se active el prefetcher. Este efecto no es visible en el adaptativo, que no disminuye el tiempo de ejecución y aumenta el número de bloques precargados cuando de activa Om. Esto es debido a que cuando el prefetcher se activa suele pedir el bloque siguiente al fallo, que es el más probable que vaya a ser utilizado. En cambio, el adaptativo original suele pedir el bloque $addr+PrefetchDegree$ porque los anteriores ya están en la cache. Por este motivo unos valores más altos de L y H son necesarios para que el grade de prefetch disminuya correctamente, como se ve en *ApOmC16i1L03E04*.

Para terminar, ls Figura 9 compara las mejores configuraciones analizadas. Puede verse claramente que los stream buffers (*str32Nc, str32*) tienen unas prestaciones peores. Por otra parte, el prefetcher secuencial con Fetchbit y grado negativo (*SpFbD16-8*) tiene el mejor tiempo de ejecución. Sin embargo, el decremento en tiempo de ejecución sobre el prefetcher adaptativo conlleva un aumento considerable de bloques precargados, debido a que el secuencial no detecta cuando está trayendo bloques que están siendo utilizados.

VII. Conclusiones

En este artículo hemos simulado y analizado diversos prefetchers adecuados para ser incluidos en una implementación de RMC. Los resultados de nuestro estudio muestran que el prefetcher adaptativo es el más adecuado a nuestras necesidades debido a su bajo coste hardware y alto rendimiento cuando está sintonizado correctamente. Además, el tráfico en la red que genera es menor que para los otros prefetchers analizados. Este prefetcher se implementará en nuestro prototipo MEMSCALE de 64 nodos basado en FPGAs y un análisis posterior se realizará con el fin de averiguar los parámetros más óptimos para nuestro entorno.

Agradecimientos

Este trabajo fue financiado por Spanish MICINN, Plan E funds, under Grant TIN2009-14475-C04-01.

Referencias

[1] Top 500. [Online]. Available: http://www.top500.org
[2] Altix UV. SGI. [Online]. Available: http://www.sgi.com/products/servers/altix/uv/
[3] NUMAChip. [Online]. Available: http://www.numachip.com/
[4] H. Montaner *et al.*, "Getting rid of coherency overhead for memory-hungry applications," CLUSTER '10.
[5] S.-H. Lu *et al.*, "Design issues of prefetching strategies for heterogeneous software dsm, CCGRID'06.
[6] H. Liu and W. Hu, "A comparison of two strategies of dynamic data prefetching in software dsm, IPDPS '01.
[7] H.-H. Wang *et al.*, "On the design and implementation of an effective prefetch strategy for dsm systems," *J. Supercomput.*, vol. 37, no. 1, pp. 91–112, Jul. 2006.
[8] J. Gray *et al.*, "Scientific data management in the coming decade," *SIGMOD Rec.*, vol. 34, no. 4, pp. 34–41, Dec. 2005.
[9] P. Conway *et al.*, "Blade computing with the AMD Opteron processor (magny-cours)," in *Hot chips 21*, aug 2009.
[10] S. Kottapalli and J. Baxter, "Nehalem-ex cpu architecture," in *Hot chips 21*, aug 2009.
[11] K. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers, ISCA '09.
[12] H. Fröning *et al.*, "A case for FPGA based accelerated communication, ICN '10.
[13] A. Mahjur and A. Jahangir, "Two-phase prediction of L1 data cache misses," *Computers and Digital Techniques, IEE Proceedings* -, vol. 153, no. 6, pp. 381 –388, nov. 2006.
[14] H. Cain and P. Nagpurkar, "Runahead execution vs. conventional data prefetching in the IMB Power6 microprocessor, ISPASS '10
[15] S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers, HPCA '07.
[16] X.-H. Sun *et al.*, "Server-based data push architecture for multi-processor environments," *J. Comput. Sci. Technol.*, vol. 22, no. 5, pp. 641–652, Sep. 2007.
[17] A. Agarwal *et al.*, "The MIT Alewife machine," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 430 –444, mar 1999.
[18] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, Jan. 2005.
[19] E. D. M. Ordonez, "Performance evaluation of the fixed sequential prefetching on a bus-based multiprocessor: preliminary results, ISPAN '96.
[20] T. Sherwood *et al.*, "Predictor-directed stream buffers, MICRO '00.
[21] F. Dahlgren and P. Stenström, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 4, pp. 385–398, Apr. 1996.