Trabajo Fin de Máster

Prebúsqueda adaptativa en un chip multiprocesador

Jorge Albericio Latorre

Curso 2009/2010

Director: Pablo Ibáñez Marín

Máster en Ingeniería de Sistemas e Informática

Programa Oficial de Posgrado en Ingeniería Informática

Universidad de Zaragoza

Grupo de Aquitectura de Computadores de la Universidad de Zaragoza (gAZ)

Zaragoza, septiembre 2010

Resumen

La prebúsqueda agresiva ha demostrado ser una técnica eficiente para mejorar el rendimiento de los sistemas monoprocesador. Sin embargo, en sistemas multiprocesador con un último nivel de memoria cache compartido (LLC), la actividad de prebúsqueda inducida por un núcleo consume recursos comunes como espacio en la LLC y ancho de banda. Esto puede degradar el rendimiento del resto de núcleos e incluso el rendimiento general del sistema. Por tanto, la prebúsqueda hardware en un multiprocesador que tiene un último nivel de cache compartido (LLC) es un reto.

En este trabajo presentamos ABS, un mecanismo de bajo coste que adecúa la agresividad de la prebúsqueda de cada uno de los núcleos en cada uno de los bancos de la LLC de un chip multiprocesador. El mecanismo se ejecuta de forma independiente en cada banco de la LLC usando sólo información local. A intervalos temporales regulares un núcleo es seleccionado y la tasa de fallos del banco y la utilidad de la prebúsqueda de dicho núcleo son muestreadas. Estas métricas son utilizadas para ajustar la agresividad de la prebúsqueda asociada al núcleo elegido.

Nuestros análisis con cargas multiprogramadas de SPEC2K6 muestran que el mecanismo mejora tanto las métricas de usuario (el tiempo medio de retorno un 27% y la equidad un 11%) como las de sistema (la productividad agregada mejora un 22% y el ancho de banda consumido se reduce un 14%) con respecto a un sistema base con ocho núcleos que usa prebúsqueda secuencial marcada de grado fijo. Los resultados son consistentes cuando se utiliza un sistema con dieciséis núcleos o cuando comparamos nuestro mecanismo con propuestas previas.

Índice

Resumen extendido

Una memoria *cache* es un almacén cercano al procesador que guarda datos próximos en el tiempo y/o en el espacio a los actualmente usados con la finalidad de acelerar el acceso a los mismos por parte del procesador. Normalmente este tipo de almacenes se distribuyen de manera jerárquica, siendo crecientes en tamaño y tiempo de acceso conforme están a una mayor distancia del procesador. Así mismo, estas memorias pueden ser usadas por un solo procesador o compartidas por varios. En la actualidad los procesadores comerciales pueden estar compuestos por varios *núcleos* o *cores*. Cada uno de ellos dispone de uno o dos niveles de memoria *cache* privados y un último nivel (LLC) [3][5][16][25] [32] compartido y dividido en bancos. En este contexto, cada banco de la LLC recibe un entrelazado de accesos provenientes de todos los núcleos que compiten por los bloques de *cache*. Cuando un acceso produce un fallo en la LLC, ha de accederse a recursos existentes fuera del chip (*off-chip misses*), como la memoria principal o memorias *cache* exteriores. Estos accesos acarrean elevadas latencias y a menudo son costosos en términos de rendimiento.

La prebúsqueda hardware es un mecanismo que trata de ocultar estas largas latencias. Trata de predecir las direcciones de memoria con anticipación, solicitándolas al siguiente nivel de la jerarquía y cargando los bloques en la memoria cache antes de que las peticiones reales del procesador tengan lugar. Estudios previos que consideraban chips monoprocesador fijaron su atención en cómo los patrones de fallo eran detectados, así como en incrementar la efectividad de la prebúsqueda [4][7][10][13][14][21][24][26][30]. El objetivo de la prebúsqueda en este contexto es obtener altas tasas de *cobertura*, *precisión* y *puntualidad*[1] con el propósito de reducir la *polución* en la memoria *cache* y el ancho de banda adicional requerido. Algunas formas de prebúsqueda hardware basada en *streams* han sido implementadas en procesadores comerciales [5][8][25][32].

Normalmente las prebúsquedas se generan a raíz de un fallo en el primer nivel de la jerarquía de memoria. Sin embargo, en un chip multiprocesador los primeros niveles de la jerarquía son privados y las prebúsquedas generadas por ellos llegan a una LLC compartida e interfieren con bloques pertenecientes a otros núcleos. Por lo tanto, las prebúsquedas causadas por un núcleo pueden expulsar a bloques de la LLC previamente reservados por otros núcleos; ya hubieran sido reservados por una *demanda* o una *prebúsqueda*. Además, la actividad de prebúsqueda de un núcleo reduce el ancho de banda total disponible, pudiendo incrementar la latencia que sufren las demandas y las prebúsquedas del resto de los núcleos.

La Figure 1. muestra las instrucciones por ciclo (IPC) para ocho aplicaciones de la suite SPEC CPU 2006 en un sistema con ocho núcleos y una LLC compartida de 4 megabytes. Los detalles de la simulación son

[1] Puede consultarse la definición de los términos en castellano escritos en *cursiva* en el apartado para definiciones incluido en como anexo.

presentados con posterioridad. La figura muestra cuatro barras para cada aplicación. Las dos primeras barras corresponden a la ejecución de la aplicación estando sola en el sistema (los otros siete cores no están ejecutando ningún programa) sin prebúsqueda y con prebúsqueda secuencial marcada agresiva (grado 16)[26], respectivamente. Las dos últimas barras de cada aplicación son el resultado de ejecutar ocho aplicaciones juntas sin prebúsqueda y con un prebúsqueda secuencial marcada agresiva, respectivamente. Si fijamos nuestra atención en el sistema con prebúsqueda (segunda y cuarta barras) podemos observar la significativa pérdida de rendimiento cuando los ocho núcleos comparten los recursos del sistema. Es interesante observar que la prebúsqueda no causa una pérdida de rendimiento significativas en ninguna de las aplicaciones cuando se ejecutan solas mientras que causa pérdidas en cinco de las ocho aplicaciones cuando todas son ejecutadas al mismo tiempo. Por lo tanto, se necesita un mecanismo que sea capaz de mejorar el rendimiento obtenido por la prebúsqueda cuando ésta se realiza sobre un último nivel de cache compartido.

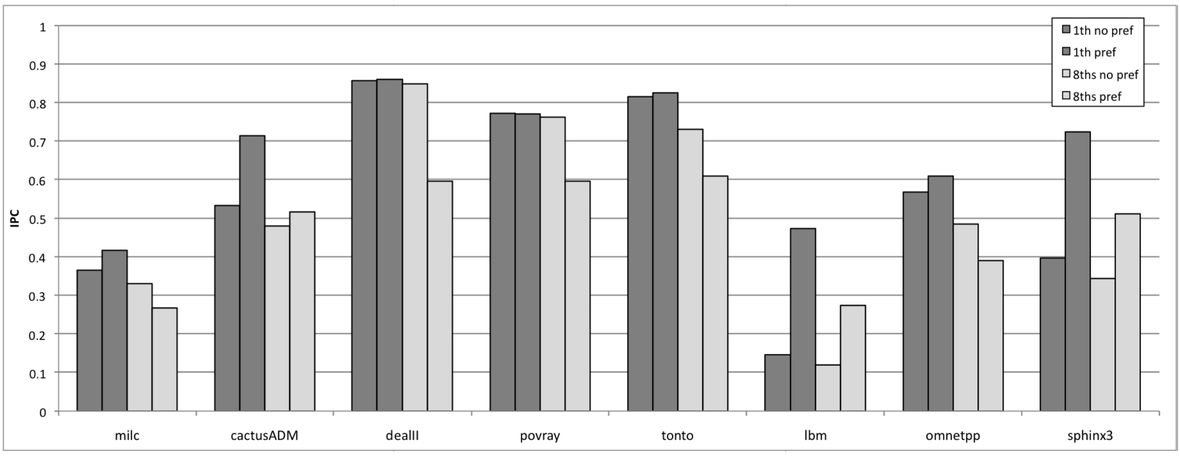


Figure 1. IPCs individuales de 8 programas de SPEC2K6 en un sistema con ocho núcleos y una LLC compartida de 4MB. Para cada grupo de barras, las dos primeras (1th no pref, 1th pref) corresponden a la ejecución de los programas solos sin o con prebúsqueda secuencial marcada de grado fijo (grado 16). Las dos barras siguientes corresponden a la ejecución de los ocho programas al mismo tiempo. En un caso en un sistema sin prebúsqueda (8ths no pref) y en el otro en un sistema dotado con la prebúsqueda anteriormente citada (8ths pref).

La mayoría de las propuestas de prebúsqueda en multiprocesadores no consideran una LLC compartida así que la prebúsqueda solo se realiza sobre memorias cache privadas. Recientemente, se ha abordado el problema de la prebúsqueda en el contexto de los chip multiprocesador dotados de una LLC centralizada y compartida [9].

En este trabajo asumimos una implementación de la LLC distribuida en múltiples bancos. En este escenario presentamos el mecanismo de prebúsqueda ABS (*Adaptive prefetching for a Banked and Shared*

LLC), un mecanismo adaptativo para controlar la agresividad de la prebúsqueda de forma independiente para cada uno de los núcleos y en cada uno de los bancos de la LLC. Para conseguir esto, tanto los motores de prebúsqueda como los mecanismos de control están replicados en cada banco de la LLC, allí cada núcleo tiene un nivel particular de agresividad de la prebúsqueda. En la prebúsqueda ABS, en cada banco el mecanismo de control establece la agresividad asociada a un núcleo basándose en cálculos simples realizados a intervalos temporales regulares (épocas). Para cada banco y en cada época la agresividad de la prebúsqueda asociada a solo un núcleo es modificada con el fin de establecer una relación de causa-efecto entre el cambio en la agresividad y el cambio en el rendimiento. Al final de cada época, dicho cambio puede confirmarse o deshacerse, estableciendo un valor para la agresividad de la prebúsqueda asociada a un procesador que será mantenida hasta que ese núcleo sea de nuevo evaluado. Para cada banco de la LLC la métrica de rendimiento considerada es la tasa de fallos local. Además, ABS considera que la precisión de la prebúsqueda del procesador que está siendo evaluado ha de ser mayor que un umbral o en caso negativo reduce la agresividad de la prebúsqueda asociada a dicho procesador.

La prebúsqueda ABS es evaluada usando prebúsqueda secuencial marcada en sistemas con ocho y dieciséis núcleos. Por otro lado, usando *streams* secuenciales, ABS es comparado con el mecanismo de control para una LLC centralizada y compartida propuesto por Ebrahimi et al. [9] pero adaptado a una LLC dividida en bancos.

Bank-Based Adaptive Prefetching for Chip Multiprocessors

Jorge Albericio

Abstract

Aggressive prefetching has been demonstrated to be an efficient technique to improve single-core system performance. However, on multicore systems with a shared Last-Level Cache (LLC), prefetching induced by a core consumes common resources like cache space and memory bandwidth. This may degrade the performance of other cores and even the overall system performance. Therefore, hardware data prefetching for the shared LLC on a chip multiprocessor is a challenge.

In this paper, we introduce ABS, a low-cost mechanism that runs stand-alone in each bank of the LLC and uses only local information. At regular time intervals it selects one core in round-robin fashion and samples bank miss ratio and core prefetch accuracy. These metrics are fed-back to control the aggressiveness of the prefetcher associated to the chosen core.

Our analysis using multiprogrammed SPEC2K6 workloads shows that the mechanism improves both user-oriented metrics (Harmonic Mean of Speedups by 27% and Fairness by 11%) and system-oriented metrics (Weighted Speedup increases 22% and Memory Bandwidth Consumption decreases 14%) over an eight-core baseline system that uses aggressive sequential prefetching with fixed degree. Similar conclusions have been obtained on a sixteen-core system or when comparing with previous proposals.

1. Introduction

Several commercial chip multiprocessors have a banked, shared Last-Level Cache (LLC) [3][5][16][25] [32]. In this context, each cache bank receives an interleaving of accesses coming from all the cores, which compete for cache bank lines. Moreover, off-chip misses involve long latency accesses to either main memory or an off-chip cache and are often expensive in terms of performance.

Hardware data prefetching is a mechanism to hide such long latencies. It tries to predict memory addresses in advance, requesting them to the next level, and loading the memory blocks into the cache before the actual demand of the core takes place. Previous studies on single-processor chips focus on how to detect cache miss patterns and increase prefetching effectiveness [4][7][10][13][14][21][24][26][30]. The goal of prefetching in this environment is to attain high coverage, high accuracy and timeliness in order to reduce cache pollution and extra bandwidth.

Several commercial processors implement some form of hardware stream-based data prefetching [5][8][25][32]. Usually such prefetches are initiated from misses in the first-level caches. However, in a chip multiprocessor the first-level caches are private, and the prefetches coming from them will reach the shared LLC and interfere with each other. That is, the prefetches issued by one core may evict LLC lines previously allocated by other cores, either due to a core demand (memory instruction) or a prefetch request. In addition, the prefetching activity originated from a core can reduce the overall available bandwidth, increasing the latency seen by the demands or prefetches of the rest of cores.

Figure 1 shows instructions per cycle (IPC) for eight SPEC2K6 applications on a multiprocessor system with eight cores and a 4MB shared LLC. The simulation details are in Section 3. The figure shows four bars for each application. The first two bars correspond to run the programs alone in the system, either without prefetching or with an aggressive (degree 16) sequential tagged prefetching. The last two bars correspond to run together the eight applications on the eight cores, either all without prefetching or all with the former aggressive prefetching turned on. By focussing on the systems with prefetching (second and fourth bars) we appreciate the significant performance losses that appear when resources are shared among cores. Note that prefetching involves virtually no performance loss in any application when running alone (first and second bars), while it causes losses in 5 out of 8 applications when running all together (third and fourth bars). Therefore, in order to improve performance by using prefetching in the shared LLC, a mechanism to control the prefetch aggressiveness is called for.

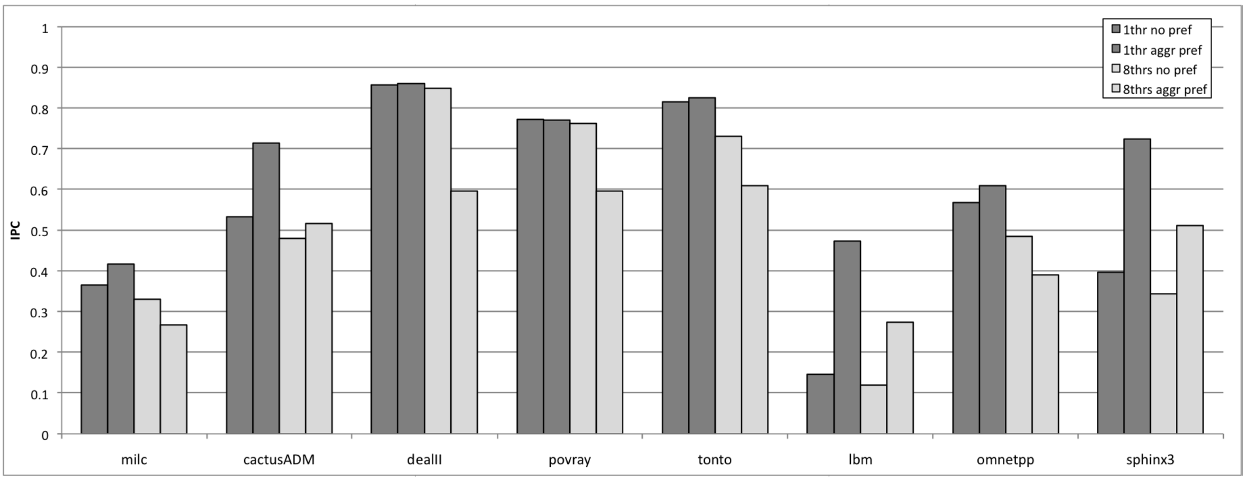


Figure 1. Individual IPC for eight SPEC2K6 programs on a system with eight cores and a 4MB shared LLC. For each bar group, the first two bars (1thr no pref, 1thr aggr pref) correspond to run the programs alone, either without prefetching or with aggressive sequential prefetching (degree 16). The next two bars correspond to run together the eight programs, either all without prefetching or all with the former prefetching turned on (8thrs no pref, 8thrs aggr pref).

Most proposals about prefetching in multiprocessors do not consider a shared cache and so prefetching only deals with private caches [2][5][15][26][29][31][34][35]. Recently, in the context of a chip multiprocessor the problem of prefetching for a centralized and shared LLC has also been faced [9].

In this paper we assume a distributed implementation of the LLC in multiple banks. In this scenario, we introduce ABS prefetching (*Adaptive* prefetching for a *Banked*, *Shared* LLC), an adaptive mechanism to control the prefetching aggressiveness independently for each core in each LLC bank. To achieve this, the prefetch engines and the control mechanisms are replicated at each LLC bank, where each core has a particular level of prefetching aggressiveness. Under ABS prefetching, the control mechanism sets at each bank the aggressiveness associated to a particular core based on computations made at regular time intervals (epochs). For each bank at each epoch the aggressiveness of only one core (the *probed* core) is varied in order to establish a cause-effect relationship between the change in aggressiveness and the change in performance. The point is that at each epoch, the observed change in a given output metric is only due to a single aggressiveness change. At the epoch end an aggressiveness value is established for the currently probed core and this value remains unchanged until it is probed again. For each LLC bank the chosen performance metrics is just its local miss ratio[1], which is a measure of the overall performance of all the cores

[1] For each bank, its local miss ratio is the addition of the demand misses, divided by all the incoming demand requests coming from all cores.

sharing the bank. Furthermore, ABS prefetching uses the accuracy[1] of the prefetched blocks for the probed core to impose a decrease of the prefetching aggressiveness for that core if accuracy exceeds a threshold.

ABS prefetching is evaluated controlling the aggressiveness of a sequential tagged prefetcher [26] with variable degree. Our results show that using the proposed adaptive mechanism in an eight-core system running multiprogrammed SPEC2K6 workloads improve both user-oriented metrics (Harmonic Mean of Speedups (HS) by 27% and Fairness (FA) by 11%) and system-oriented metrics (Weighted Speedup (WS) increases 22% and Memory Bandwidth Consumption (BW) decreases 14%) over baseline system that uses aggressive sequential prefetching with fixed degree. The results are consistent when varying the number of cores of the considered system. Besides, it is also compared, controlling the aggressiveness of sequential streams, to the control mechanism proposed for a centralized shared LLC[2] by Ebrahimi et al. [9] but adapted for a banked LLC[3].

The paper structure follows. In Section 2 we introduce the different parts of the ABS prefetching and explain the rationale behind the considered metrics. In Section 3 we show the methodology, hardware implementation and performance indexes employed to ana lyse ABS performance. Section 4 evaluates ABS prefetching, showing results for systems with 8 and 16 cores, and using sequential tagged as prefetching engine. We also compare ABS with an adaptation of the mechanism by Ebrahimi et al. [9]. In Section 5 we review the related work and in Section 6 we conclude the paper.

[1] In each bank, the prefetch accuracy for each core is the ratio of useful to total lines prefetched on behalf of that core.

[2] Although it is internally multibanked, each cycle only one demand request can be accepted.

[3] As many demand request as banks can be accepted in one cycle.

2. ABS prefetching

ABS prefetching is an adaptive mechanism to control prefetching *aggressiveness* independently for each core in each bank of a *banked shared* LLC. In order to correlate changes in aggressiveness with changes in performance, for each bank at each epoch only a single core is probed. So, at the end of epoch, an aggressiveness value is established for the currently probed core and it remains unchanged until that core is probed again. Furthermore, prefetching aggressiveness for the probed core will be decreased if its prefetching accuracy drops below a threshold value.

Next we describe the ABS mechanism for any LLC bank, which is replicated in all LLC banks. The ABS operation can be divided into three parts: a sampling system that also selects the core to be probed, an adaptive per-core aggressiveness control, and a prefetch engine.

Core selection and sampling system. Time is divided in equal-length intervals, called epochs (Figure 2.a). At the beginning of each epoch a core is chosen in a round-robin fashion and its associated prefetch aggressiveness is changed. Then at the end of the epoch the effect of the change is evaluated by comparing the bank miss ratios observed during the *current* and the *reference* epochs. The change is undone if the current bank miss ratio is greater than the reference bank miss ratio. Otherwise, the change is confirmed and the current epoch is set as the new reference, storing a new miss ratio reference for the LLC bank.

Because the reference epoch is not changed after a probe increases the bank miss ratio, the reference epoch may become stale and not reflect the current behavior of the applications. This could occur when the bank miss ratio falls into a local minimum. In order to remedy this situation, the number of epochs elapsed without updating the reference epoch is being counted. When this counter is equal to the number of cores, the mechanism sets the last epoch as the new reference epoch. By updating the reference epoch in this way, we ensure that after probing all cores without realizing a miss ratio decrease a new value is taken as a reference.

Summarizing, the core selection and sampling system guarantees there is always only one prefetch aggressiveness change between reference and current epochs. That change corresponds to the probed core at each epoch.

Adaptive per-core miss-gradient aggressiveness control. Each core has a state which consists of a prefetch degree and a prefetch aggressiveness trend (downward or upward). At the beginning of the epoch in which a core is being proved, the state changes to eval-downward or eval-upward (Figure 2.b).

Four events are locally counted in each LLC bank during each epoch, namely bank accesses and misses from any core, prefetches from the core being probed, and hits on lines prefetched for the core being probed. At the end of an epoch two ratios are computed: *bank miss ratio* (bank misses / bank accesses) and *prefetch accuracy* for the core being probed (hits in prefetched lines / total prefetches). Then the computed bank miss ratio is compared with the bank miss ratio of the reference epoch. If it has increased, state changes to the contrary trend (from eval-downward to upward or from eval-upward to downward). Otherwise, the state changes to the initial trend and the reference bank miss ratio is updated.

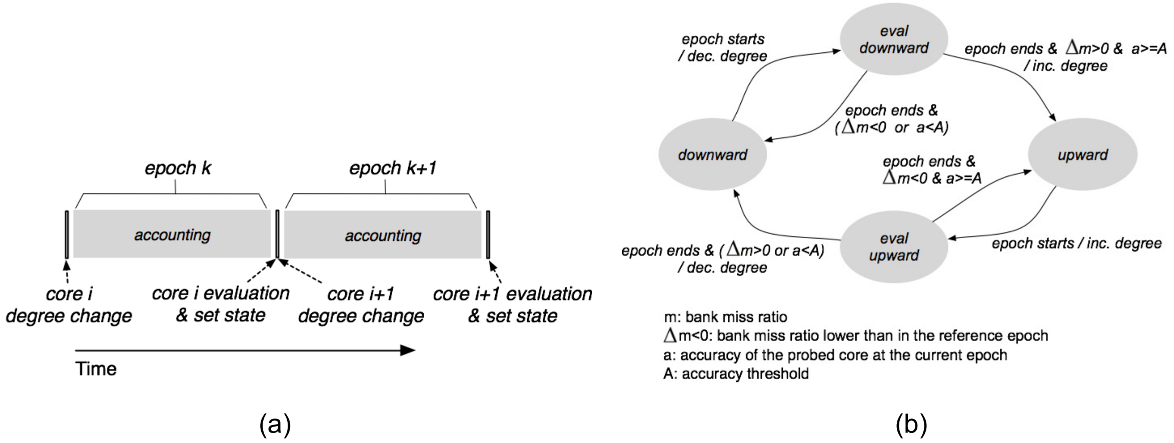


Figure 2. **(a)** In parallel in all LLC banks, cores *i* and *i+1* are probed during epochs *k* and *k+1*, respectively. At the end of the epoch *k,* the effect on the bank miss ratio of the change of prefetch degree of core *i* is evaluated, resulting either in a change of reference (prefetch degree confirmed for core *i*) or not (core *i* reverts to the former prefetch degree). **(b)** Per-core finite state machine controlling the prefetch aggressiveness (degree). State transitions occur when a core is being probed: a transition at the beginning of the epoch (to eval-x states) and another one at the end of the epoch (from eval-x states).

Accuracy is involved in the transitions that leave the *eval-x* states. It is required that the probed core has an accuracy higher than a threshold to go to the *upward* state. The rationale of this requirement is not to increase the aggressiveness of one core whose prefetches are almost useless. We have observed that the optimal threshold depends on the prefetch engine features.

Prefetch engine. ABS prefetching can work with any prefetch engine able to generate a variable number of memory references. In this work, the base system uses a sequential tagged with variable degree as prefetch engine. Given an initial address and a degree *k*, it is asked to generate *k* sequential references in the next *k* cycles. In Section 4.3 we also evaluate ABS using the sequential streams implementation suggested by Ebrahimi et al. in [9] as prefetch engine. However, in other context the prefetch engine may generate non-consecutive references belonging to a fixed-stride stream, or a reference stream generated by a context predictor like GHB [21].

2.1. Example

Figure 3 shows an example of how ABS works in a LLC bank of a system with four cores. The degree scale (0, 1, 4, 8, 16) represents the prefetch aggressiveness. Time moves from left to right and it is divided into regular time intervals as shown in row *Epoch*. The four rows designated as *degree & trend* (P0, P1, P2 and P3) show the level of prefetch degree and trend associated with each core at each epoch. For instance, 4↑ means prefetch degree of 4 and upward trend. The core identifier and the arrow over the dashed arrows indicate the change applied between two consecutive epochs. For example, from E0 to E1 the degree of P1 is being changed, from 4 to 1. The row *bank miss ratio accounting* shows the miss ratio at each epoch. Finally, the last row shows the comparison result between the reference and the current bank miss ratios. Next we show the positive and the negative cases.

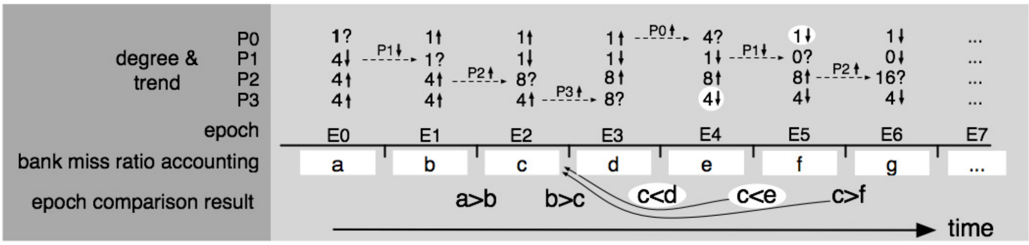


Figure 3. Example of ABS prefetching working in a LLC bank of a 4-core system (cores P0 to P3). Numbers 1-16 with arrows and question marks represent prefetch degrees.

Positive aggressiveness change. At the beginning of E1, aggressiveness of P1 core is changed from a degree of 4 to 1 following its downward trend. The question mark next to P1 row at E1 epoch (1?) means that the change is being evaluated. At the end of E1 we observe a decrease on the bank miss ratio (a >b) respect to the reference epoch (a). Therefore, the change on degree and the current trend of P1 are confirmed. E1 becomes the new reference epoch. The same happens at the epochs E2 and E5.

Negative aggressiveness change. White circles surround negative evaluations. At the begin of E3, P3 degree is changed from 4 to 8 following its upward trend. At the end of E3, an increment on the bank miss ratio (c<d) is observed, therefore the change on P3 degree is undone, becoming 4, and its trend is reversed to downward. The reference epoch is not changed. At the begin of E4, P0 degree is changed from 1 to 4. Note the epoch of reference at E4 is E2. The only difference between E2 and E4 is the change on the degree of P0, this is the one under evaluation at E4. At this epoch, the bank miss ratio also is higher than at E2. Therefore, at the end of E4 the change on P0 degree is undone, its trend is reversed and E2 continues as reference at E5.

2.2. Miss ratio as a good metric to guide aggressiveness

Prefetch related metrics as coverage, accuracy, timeliness, pollution or consumed bandwidth have been often proposed to evaluate the quality of a prefetch engine because an aggregate figure such as the miss ratio do not allow to distinguish the net effect of individual prefetches [12][24][34].

These same metrics have been used later in other works to guide prefetch aggressiveness [5][30]. However, these metrics are not directly related with system performance. Moreover, in this context they face two important problems: *i)* some of them are hard to compute online, and *ii)* they are difficult to aggregate in a single number in order to take a decision.

In this paper we use the LLC bank miss ratio as the main metric to guide prefetch aggressiveness. The penalty of the off-chip misses is large in processor cycles and so a miss ratio decrease has a great potential to reduce Cycles per Instruction (CPI) and improve performance. Therefore, we expect the LLC miss ratio to be a good measure of performance. Moreover, the prefetch engine associated with each LLC bank can locally count the number of misses in the bank. In consequence, our proposal establishes the feedback loop without requiring communication among LLC banks.

From a performance standpoint, the prefetch related metrics are highly correlated with the miss ratio. In fact, some of these metrics are only valuable when they really correlate with the miss ratio. For instance, a prefetched line is considered useful if it is used along its lifetime in cache, and useless otherwise (accuracy metric). However, a useful prefetch does not always have a positive impact on performance. Indeed, it only will increase performance if it gets a reduction in the miss ratio. In particular, if the block evicted by the prefetch is referenced before the prefetched block itself, despite having a useful prefetch, the miss ratio does not change and therefore no performance increase will be seen.

2.3. Hardware cost

Our proposal is low-cost in hardware. In each bank, ABS needs 4 bits per core in order to keep its prefetching state (1 bit for trend + 3 bits for aggressiveness level). It also needs four 16 bits counters to store the number of bank misses, accesses to the bank, prefetch requests sent to main memory, and hits on prefetched blocks. Additionally, it needs one counter of 3 bits (4 bits in a 16-core system) to keep the number of epochs without having a successful change. The reference is stored in a 32 bits register. Finally, ABS needs 15 bits to divide time into 32K cycles intervals.

Summarizing, for a 8-core system with a 4-bank LLC like the one of our baseline system ABS needs 539 bits (less than 68 Bytes).

3. Methodology, hardware implementation and performance indexes

3.1. Experimental setup

We use Simics [26], a full-system execution-driven simulator to evaluate the proposal. We also use the GEMS toolset and Ruby to model the memory hierarchy with a high degree of detail [19]: coherence protocol, on-chip network, communication buffering, contention, etc. We have integrated the prefetch system into the coherence mechanism and we have added a detailed DDR3 DRAM model.

We use multiprogrammed SPECCPU 2k6 workloads for our experimental evaluation. In order to discard the less demanding memory applications and locate the end of the initialization phase, we use hardware counters on a real machine and run until completion all the SPARC binaries with the reference inputs. After doing so, 8 applications were discarded and 21 selected, which appear in the first column of Table 1

.

	mix1	**mix2**	**mix3**	**mix4**	**mix5**	**mix6**	**mix7**	**mix8**	**mix9**	**mix10**
bzip2					1.8		1.7			
bwaves	20.7		20.7		20.7	20.7	20.7			20.7
mcf	33.9			37.5	30.7	33.2	20.2			
milc	16.4	16.2		34.2						
zeusmp	16.7						8.1	9.1	7.3	13.8
gromacs			3.9						4.0	
cactusADM		4.2		4.1		4.2	4.2	4.4		
leslie3d				28.3	32.5	36.4			14.4	
gobmk			1.5	1.5					1.4	1.5
dealII		0.0		0.1				0.2	0.3	
soplex	3.0			4.0				3.6		
povray		0.3	0.3				0.3		0.3	0.3
calculix						0.5			5.9	0.5
GemsFDTD	32.5		26.9				32.5	26.8		
libquantum					28.8	85.5	65.6			
tonto		3.4	1.5						1.6	
lbm	36.1	47.6	36.1		36.1					36.1
omnetpp	0.7	5.4			0.7	0.7		0.6		0.7
wrf					3.0			0.5		0.5
sphinx3		12.5		12.9						
xalancbmk			1.8			1.8		1.5		

Table 1. Ten mixes of 8 application selected for individual analysis in Section 4. Values in the cells are the misses per kilo-instruction of each application in each mix when the application runs alone.

For an eight-core system, we have produced a set of 30 random mixes of 8 programs each, taken the programs from the previously selected 21 SPECCPU 2k6 programs (no effort has been made to distinguish between integer and floating point). In the next Section, we usually show averages over that set of 30 mixes, but in order to get a deeper insight into individual mix behaviors, sometimes a subset of 10 mixes is showed. This randomly chosen subset of mixes appears in Table 1, along with the misses per kilo-instruction (mpki) of each application in each mix when the application runs alone, that is, it runs using all the shared memory resources and with the remaining seven cores stopped.

Notice that, in general the same application appearing on two different mixes does not have the same mpki value. This is because the number of executed instructions before creating a multiprogrammed checkpoint is given by the application with the longest initialization phase in the mix. So, the results for a particular application, in general, can not be compared between mixes.

Summarizing, we have created 30 checkpoints of 8 applications each. Every checkpoint guarantees no application is in its initialization phase. The cycle-accurate simulation starts at those checkpoints, warming the memory hierarchy for 500 million cycles with prefetching disabled, and then we collect statistics for the next 500 million cycles.

3.2. Baseline system

The baseline system has eight in-order cores with small first-level caches. The shared LLC has four banks. A MOSI protocol maintains the memory system coherent. A crossbar communicates the first level caches and the shared LLC banks. There is a single DDR3 memory channel. The DRAM memory bus runs at a quarter of the core frequency. Table 2 gives additional implementation details.

private L1 I/D	16KB, 4-way, 64B line size, 1cycle
shared L2	4MB inclusive, 4 banks. 1MB/bank, interleaved by OS page of 4KB. 64B line size Each bank: 16-way, 2 cycles TAG access, 4 cycles data access. 16 demand + 16 prefetch MSHR
Bank prefetcher	Sequential tagged: degree 16
DRAM controller	1 memory channel, demand first [16]
DRAM	1 rank, 16 banks, 4KB page size, Double Data Rate (DDR3 1333Mhz)
DRAM bus	667Mhz, 8B wide bus, 4 DRAM cycles/block, 16 processor cycles/block

Table 2. Implementation details of the 8-core baseline system

3.3. Prefetching framework

Aggressive prefetchers like a high-degree sequential tagged prefetcher can generate a significant number of prefetches. We assume the hardware cost of the prefetcher is lowered by sharing the same tag lookup port for demand and prefetch requests (demand requests have higher priority). Furthermore, only one adder is provided to each LLC bank in order to generate prefetches, and so prefetch addresses are computed at rate of one per cycle. The generation of a burst of prefetch requests after a cache miss or cache hit to a tagged block is cut-off if another event initiate a new burst of prefetches.

After been generated, each prefetch is sent to the prefetch address buffer (PAB) and it waits queued until the LLC tag port is available (Figure 4). Because the PAB has a finite number of entries it is managed in FIFO order (oldest prefetch request is dropped first). Before inserting a prefetch address, the PAB is checked for an already allocated entry with the same address. That is, the PAB does not have duplicated requests.

Prefetch and demand Miss Status Holding Registers (MSHRs) keep the requests to main memory pending the arrival of the corresponding cache lines. There are no duplicates between MSHRs. When the LLC bank tag port is available, the request at the head of PAB lookups the bank tags and both MSHRs. Only on a miss in all of them, the request is sent to memory and inserted in the prefetch MSHRs.

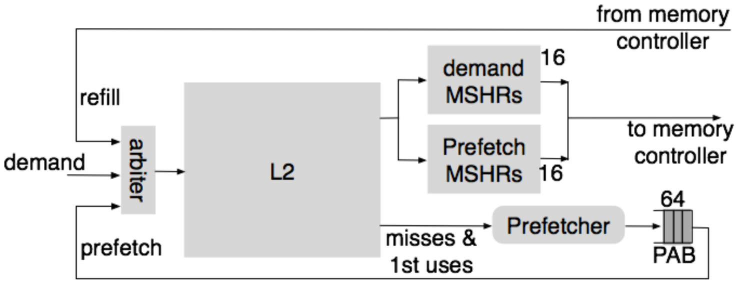


Figure 4. Components of an LLC bank

Regular demands have higher priority than prefetch requests at every arbiter in the hierarchy. Moreover, a regular demand can arrive to an LLC bank asking for a line that is being prefetched but whose data are not loaded yet into cache. In that case a *prefetch upgrade* message is sent to the memory controller. If the request is queued at the controller that message will upgrade the prefetch request giving it demand priority. The rationale of this mechanism is to prevent that an aggressive prefetching damages regular memory instructions.

3.4. ABS prefetching parameters

ABS prefetching works on top of the aforementioned prefetching framework. We use epochs of 32K cycles (other durations were tested without significant variation) and accuracy threshold of 0.6. The prefetch degree for the sequential tagged prefetcher varies along the following scale: 0, 1, 4, 8, 16.

3.5. Performance indexes

We assume that an aggressive sequential prefetch delivers good results when only one program is executed in the system as previous work has shown [1]. But in Section 1 we pointed out this assumption is no longer true on a multiprocessor system because of resources are shared among cores that interfere each other. So, the ABS goal is to control the prefetch aggressiveness in a per-core basis in such a way that programs running together in the multicore system attain similar performance than running alone with an aggressive sequential prefetch. In consequence, our performance indexes use as references the IPC of the programs running alone on a system with a sequential tagged prefetcher with a fixed degree of 16.

In order to evaluate ABS in the multiprocessor system we use two system-oriented performance indexes, namely weighted speedup [16], and main memory bandwidth consumption, and two user-oriented performance indexes, namely harmonic mean of speedups [26], and fairness [21], see Table 3. The weighted speedup (WS) quantifies the number of jobs completed per unit of time. The harmonic mean of speedups (HS) is the inverse of the average normalized turnaround time [10]. To determine whether the co-execution in the multicore system benefits or harms some programs more than others we use the fairness (FA).

Weighted Speedup [16]	$WS = \sum_{i=1}^{n} \frac{IPC_i^{MP}}{IPC_i^{SP}}$
Harmonic Mean of Speedups [26]	$HS = n / \left(\sum_{i=1}^{n} \frac{IPC_i^{SP}}{IPC_i^{MP}} \right)$
Fairness [21]	$FA = \frac{min(IS_1, IS_2, ..., IS_n)}{max(IS_1, IS_2, ..., IS_n)}$, where $IS_i = \frac{IPC_i^{MP}}{IPC_i^{SP}}$

IPC_i^{MP} is the IPC of program *i* when other applications are running in the rest of cores

IPC_i^{SP} is the IPC of program *i* running alone in the system and with a fixed degree-16 sequential tagged prefetch

Table 3. Performance indexes.

4. Results

4.1. Results for an eight-core system

Figure 5 shows ABS working in the program mix #2 used in the motivation of Section 1. The 8 programs run simultaneously and we keep the bars corresponding to execution without prefetching (no pref) and with fixed-degree(16) aggressive prefetching (aggr pref). We add a new bar corresponding to prefetching under ABS control (ABS).

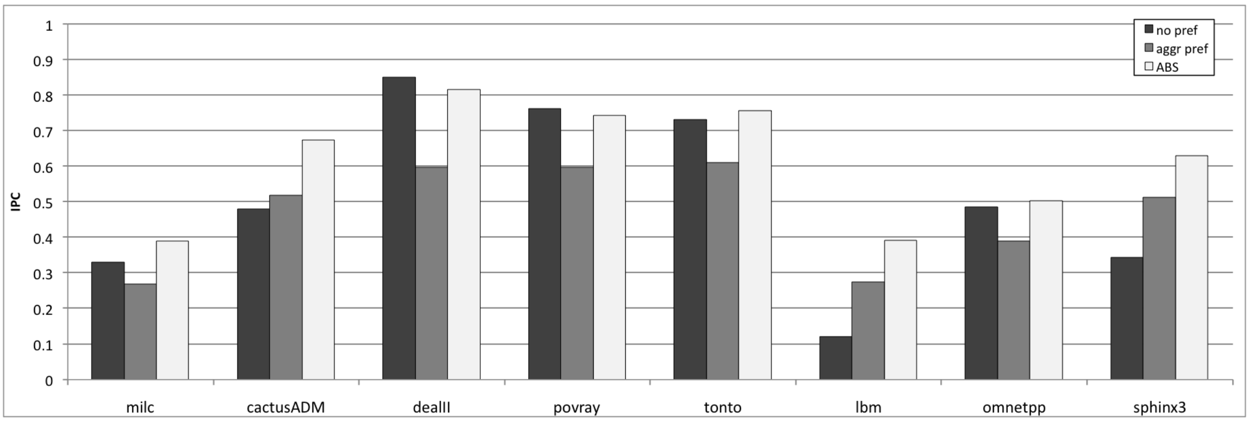


Figure 5. Individual IPC for mix #2 on a system with eight cores and a 4MB shared LLC.

ABS increases performance compared with aggressive prefetching for all programs. The increase varies from 23% in *sphinx3* to 40% in *milc*. With regard to the system without prefetch, ABS only affect slightly the performance of *deall* and *povray*, while the aggressive prefetching leads to significant losses in five of the programs. In contrast, in six of the eight programs of the mix, ABS outperforms the system without prefetch achieving improvements that range between 3% in *omnetpp* and 225% in *lbm*.

Figure 6 shows HS, WS, FA and consumed bandwidth for systems without prefetching, with aggressive prefetching and with ABS prefetching for the ten mixes of Table 2. In each plot the rightmost bar group represents the average of the 30 mixes (AVG30).

The HS values show a nonuniform pattern across the different mixes (Figure 6.a). Aggressive prefetching increases the average HS 4% with respect of no prefetch, but causes losses in six of the 10 mixes. Under the ABS control, aggressive prefetching improves in 9 of the 10 mixes (up to 60% in mix6) and produces small losses in the other mix (0,5% on mix9). On the other hand, ABS always increases performance compared to no prefetch, between 20% and 50% in mix3 and mix6, respectively. On average ABS improves the system without prefetch by 35%.

In terms of WS (Figure 6.b), aggressive prefetching causes losses in seven of the 10 mixes and performs on average 3% worse than the system without prefetch. ABS improves aggressive prefetching in 9 of the 10 mixes (up to 47% in mix4) and produces negligible losses in the other mix (1% on mix9). On the other hand, ABS improves the system without prefetch in 9 of the 10 mixes, with improvements ranging from 14% in mix9 to 27% in mix3. In mix5, WS results in a reduction of 0.3%. On average, ABS improves the system without prefetch by 18%.

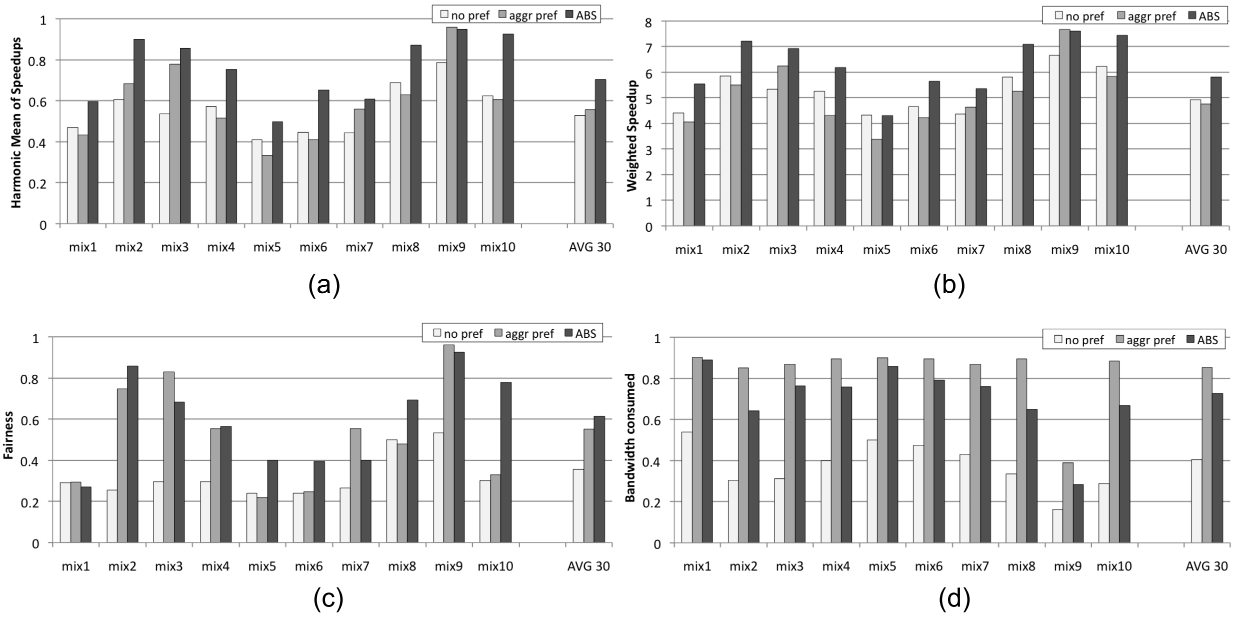


Figure 6. Performance indexes for the ten mixes of Table 2 and the average behavior for all the thirty mixes (AVG 30). **a)** HS index. **b)** WS index. **c)** FA index. **d)** Bandwidth consumed.

Notice that the performance indexes HS and WS not always correlate; in mix2, for instance, the HS index points out that aggressive prefetch is better than no prefetch while the WS index indicates the contrary. The following discussion on fairness will give a deeper insight about what is happening.

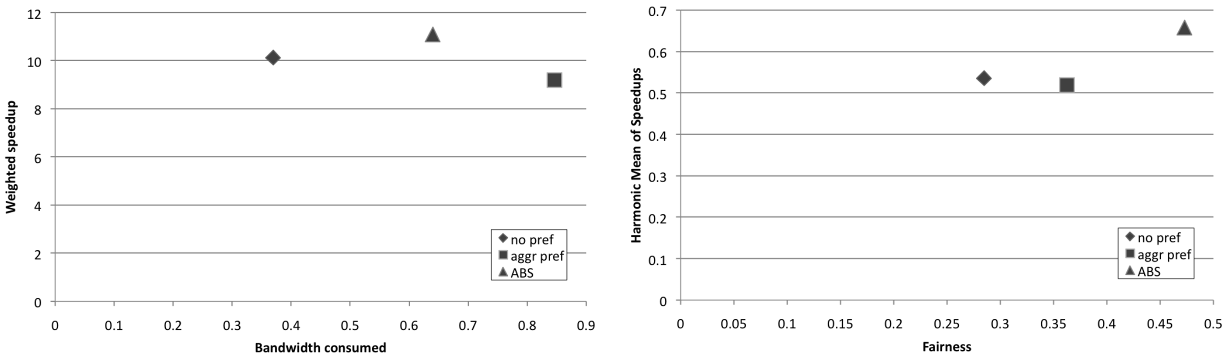
Figure 6.c plots the FA values, showing that the system with aggressive prefetching is significantly more fair than the system without prefetch. This is because the performance indexes use as reference a system with prefetching as we have seen in Section 3.5. Therefore, in the system without prefetch we see the unfairness introduced by the lack of prefetch itself, plus the unfairness due to the interferences among the eight cores. In Figure 6.c, we observe the low fairness of mix 2 on the system without prefetching. As the HS index includes some notion of fairness in its definition and WS is a pure throughput index, the previous issue about mix2 becomes clear. On average, the ABS system is more fair than the system with aggressive prefetching (0.62 and 0.56, respectively). ABS is more fair in 6 of the 10 mixes (differences between 1% and 140%), and less fair in the remaining 4 (differences between -3% and -30%).

Finally, in Figure 6.d we see that the main memory bandwidth consumption of the system without prefetch is very uneven among the different mixes, varying between 18% and 55% of the maximum bandwidth. However, the common pattern is that aggressive prefetching greatly increases bandwidth consumption with respect to the system without prefetch (on average, from 40% to 85% of maximum bandwidth), and ABS removes a significant portion of that increase lowering it to 70% of maximum bandwidth.

Summarizing, in a 8-core chip with a shared 4-MB LLC, ABS prefetching introduces significant benefits over a system with aggressive prefetching. On average, throughput (WS), the inverse of the turnaround time (HS), and fairness (FA) increase 27%, 23% and 11%, respectively while memory bandwidth consumption decreases 18%.

4.2. Results for a 16-core system

In this section we analyze the behavior of prefetching in a 16-core system. The LLC is not modified, so that the increase in the number of cores results in an increased pressure on the LLC. However, communication with main memory is expanded from one to two DDR3 channels. We run 30 random mixes of 16 applications randomly selected among the 21 SPEC CPU 2006 most active applications on the LLC. For the shake of brevity, we only present the average of each index over the 30 mixes of 16 programs each. Figure 7.a combines in a single Y-X plot the system-oriented metrics, WS and bandwidth, while Figure 7.b combines the user-oriented metrics, HS and FA.



(a) (b)

Figure 7. Comparison between average indexes for sequential tagged aggressive prefetching and ABS on a 16-core system. **a)** System-oriented metrics: WS and consumed bandwidth. **b)** User oriented metrics: HS and Fairness.

In a system with 16 processors, aggressive prefetching produces losses with respect to no prefetch in terms of throughput (WS decrease 9%) and turnaround time (HS decrease 4%). The memory bandwidth consumption greatly increases from 36% to 85%. Only fairness improves from 0.28 without prefetch to 0.36 with aggressive prefetching.

The control of aggressiveness leads to improvements in all metrics. Compared to aggressive prefetching, ABS increases the HS index 27% (22% compared to no prefetch), increases the FA index to 0.48, and also improves the system throughput index with a WS increase of 25% (14% compared to no prefetch). The bandwidth consumption decreases significantly compared to aggressive prefetching, from 85% to 62% of the maximum, but it is still greater than without prefetch which only requires 36% of the maximum.

Summarizing, in a 16-core chip with a shared 4-MB LLC, ABS prefetching introduces significant benefits over a system with an aggressive prefetching in all the performance indexes tested. Comparing 16 with 8 cores, the increase in the WS index is similar but the improvement in the rest of indexes —HS, fairness and memory bandwidth— is much higher. This result is consistent because the pressure on the memory hierarchy in a 16-core chip is larger than in an 8-core, resources are more scarce, and therefore controlling the prefetching aggressiveness becomes more important.

4.3. HPAC comparison

Next we compare ABS with the Hierarchical Prefetcher Aggressive Control mechanism (HPAC) introduced by Ebrahim et al. in [9]. To the best of our knowledge, this is so far the only work that proposed to adjust prefetch aggressiveness on a shared LLC.

HPAC works on a centralized LLC with a single access port although internally is organized in banks to support several concurrent accesses. Its base proposal uses sequential streams as the prefetch engine and a local control of aggressiveness for each core: Feedback-Directed Prefetching (FDP) [30]. HPAC adds a global interference feedback in order to coordinate the prefetchers of the different cores and throttle their aggressiveness.

We have implemented HPAC and FDP adapting them to a banked LLC, where each bank has an access port. We have used the thresholds indicated in the published proposals for both mechanisms. In addition, we have implemented sequential streams as described in the referred works [9][30]. We simulate 32 streams per core and LLC bank. Each stream launches sequential prefetches with a degree and distance from a starting address. We implement five levels of aggressiveness that correspond to degrees 1, 1, 2, 4, and 4 and distances 1, 4, 16, 32, and 64, respectively. The aggressiveness control mechanism (HPAC or FDP) decides the aggressiveness level associated to each core. In this subsection we change the accuracy threshold of ABS from 0.6 to 0.3 because now the prefetch engine, sequential streams, is more accurate than sequential tagged with variable degree.

Figure 8 plots results for HPAC and ABS on an 8-core system. Both mechanisms use sequential streams as the prefetch engine. Performance indexes have been computed using as references the IPCs of the programs running alone on a system with a sequential stream prefetcher with a fixed level of aggressiveness (distance 64 and degree 4). We only show average indexes computed over the 30 mixes already used in Section 4.1.

Figure 8.a shows the system-oriented metrics, WS and bandwidth, while Figure 8.b shows the user-oriented metrics HS and Fairness.

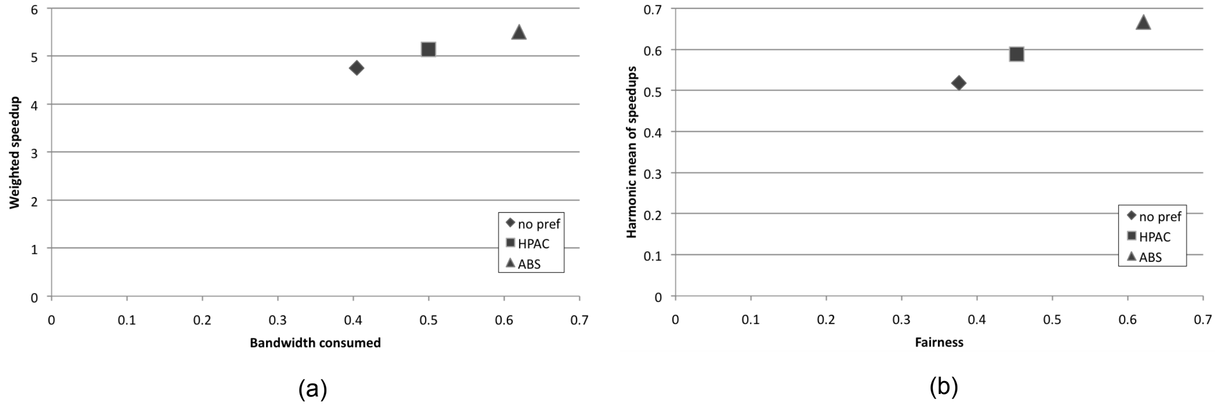


Figure 8. Average performance indexes for HPAC and ABS on a 8-core system. **a)** System-oriented metrics, WS and consumed bandwidth. **b)** User oriented metrics, HS and fairness.

ABS gets better results than HPAC in all metrics except in the consumed bandwidth. ABS improves WS index by 8%, HS index by 14% and fairness by 40%. Bandwidth consumption is higher in ABS (62%) than in HPAC (50%).

Figure 9 extends the results to a 16-core system. Performance indexes have been averaged over the 30 mixes already used in Section 4.2.

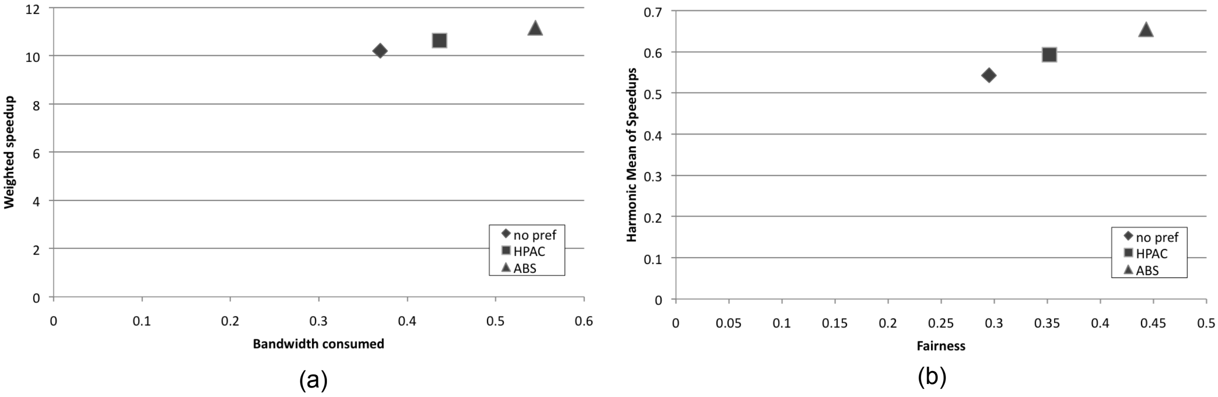


Figure 9. Average performance indexes for HPAC and ABS on a 16-core system. **a)** System-oriented metrics, WS and consumed bandwidth. **b)** User oriented metrics, HS and fairness.

The results for a 16-core system are similar to those obtained for a 8-core system. ABS also gets better results than HPAC in all metrics except consumed bandwidth. ABS improves WS index by 7%, HS index by 11%, and fairness by 29%. Bandwidth consumption is higher in ABS (54%) than in HPAC (44%).

Summarizing, ABS gets better performance than HPAC at the expense of some increase in consumed bandwidth. In addition, it succeeds with a very low implementation cost.

5. Related work

Several works address prefething in CC-NUMA multiprocessor systems having only private cache memories[2][5][15][26][29][31][34][35]. Dahlgren et al. suggest to determine the prefetch aggressiveness at each private cache by counting the number of useful prefetches every given number of issued prefetches (an epoch) [5]; the prefetch aggressiveness is increased or decreased taking into account two usefulness thresholds. Tcheun et al. add a degree selector to a sequential prefetching scheme [31]; when the selector detects useful prefetches along a sequential sub-stream it increases the prefetch aggressiveness of next sub-stream belonging to the same stream. Cantin et al.[2] and Wallin et al. [34] aim to identify private memory regions not shared by the other processors, starting to prefetch only in these memory regions in order to avoid that shared data prefetching may hurt performance. Koppelman et al. use the instruction history to compute an area around the demanded data, which can be prefetched [15]. Somogyi et al. predict memory accesses that exhibit a repetitive layout (spatial streaming); they propose a predictor that correlate the memory access patterns with instructions addresses [26]. Wenisch et al. propose temporal streaming [35], which is based on the observation that recent sequences of shared data accesses often recur in the same precise order; therefore they propose to move data to a sharer in advance of its demand. Somogyi et al. leverage the ideas of the two previous works and propose a predictor that exploit both temporal and spatial correlations [29].

For chip multiprocessors with a centralized and shared L2 cache Wenisch et al. evaluate to store meta-data off chip for an address-correlating prefetcher in a four-core chip [36]. Their problem is how to save the large amount of information that requires their prefetcher to get good results in commercial applications. In contrast, our proposal uses a very low cost prefetcher similar to that implemented in several commercial processors and aims to achieve good results in multicore systems with shared resources.

To our knowledge, only the work of Ebrahimi et al. faced the problem of adjusting prefetch aggressiveness on a shared LLC (HPAC)[9]. We see two main differences with it. 1) In HPAC, the LLC used is a monolothic cache with only one channel between it and the first levels of the hierarchy. In our work, a multibank LLC with one access port per bank is used. 2) HPAC throttles the auto-regulated prefetch engines associated to each core. In the original paper, FDP [21] was used as auto-regulated prefetch engine. However, ABS sets directly the aggressiveness level of the local prefetchers. To collect the control parameters in a banked LLC, HPAC requires several hardware structures per bank. On the contrary, our proposal requires very little hardware to collect the control parameters. A performance analysis of both mechanisms was presented in Section 4.

6. Conclusions

In this paper we focus in the shared, last level cache (LLC) of a multicore chip. We assume LLC has a distributed implementation in multiple banks. In this scenario, we introduce ABS prefetching (*Adaptive* prefetching for a *Banked*, *Shared* LLC), an adaptive mechanism to control the prefetching aggressiveness independently for each core in each LLC bank. The mechanism runs stand-alone at each LLC bank and uses only local information. Bank miss ratio and prefetch accuracy are sampled at regular time intervals (epochs) and used as control variables. For each bank at each epoch the aggressiveness of only one core is varied in order to establish a cause-effect relationship between the change in aggressiveness and the change in the control variables.

We evaluate ABS prefetching controlling the aggressiveness of a sequential tagged prefetcher with variable degree. Our analysis using multiprogrammed SPEC2K6 workloads shows that the mechanism improves both user-oriented metrics (Harmonic Mean of Speedups by 27% and Fairness by 11%) and system-oriented metrics (Weighted Speedup increases 22% and Memory Bandwidth Consumption decreases 14%) over an eight-core baseline system that uses aggressive sequential prefetching with fixed degree. Similar results have been obtained on a sixteen-core system.

Besides, ABS is also compared to the control mechanism proposed by Ebrahimi et al. for a centralized shared LLC [9], but adapted for a banked LLC. For this comparison the selected prefetch engine has been a variable-aggresiveness sequential stream prefetcher. ABS performs better in all the performance indexes but in required bandwidth, which is somewhat greater.

Summarizing, ABS prefetching is able to control aggressive prefetch engines in a distributed fashion and with very low implementation cost. Its distributed nature assures scalability in the number of cores and cache banks for future multicore chips.

References

[1] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In Supercomputing '91, pp. 176-186, 1991.

[2] J. F. Cantin, M. H. Lipasti and J. E. Smith. Stealth prefetching. In ASPLOS Conference, pp. 274-282, 2006.

[3] J. Casazza, First the Tick, Now the Tock: Next Generation Intel Microatchitecture (Nehalem), Intel® Xeon® processor 3500 and 5500 series. In white paper, 2009

[4] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. In MICRO-40, pp. 301-313, 2007.

[5] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. In IEEE Micro, vol. 30, no 2, pp. 16-29, 2010.

[6] F. Dahlgren et al. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In ICPP-22, pp. 56-63, 1993.

[7] P. Diaz and M. Cintra. Stream chaining: Exploiting multiple levels of correlation in data prefetching. In ISCA, pp. 81-92, 2009.

[8] J. Doweck. Inside Intel Core microarchitecture and smart memory access. In Intel White Paper, 2006.

[9] E. Ebrahimi, O. Mutlu, C. Joo Lee, and Y. N. Patt. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In MICRO, pp. 316-326, 2009.

[10] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. In IEEE Micro, volume 28(3), pp. 42–53, 2008.

[11] Hur and C. Lin. Memory prefetching using adaptive stream detection. In MICRO-39, pp. 397-408, 2006.

[12] D. S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. ICS, pp. 1-11, 2004.

[13] Joseph and D. Grunwald, Prefetching using Markov predictors, IEEE Transactions on Computers, vol. 48, no. 2, pp. 121–133, 1999.

[14] NN. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In ISCA, pp. 364-373, 1990.

[15] D. M. Koppelman, Neighborhood Prefetching on Multiprocessors Using Instruction History, In PACT, pp.123-132 , 2000.

[16] C. J. Lee et al. Prefetch-aware DRAM controllers. In MICRO-41, pp. 200-209, 2008.

[17] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In ISPASS, pp. 31-38, 2001.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A.

Moestedt, B.Werner. Simics: A Full System Simulation Platform. In Computer, vol. 35, no. 2, pp. 50-58, 2002.

[19] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. In Computer Architecture News, pp. 92–99, Sept. 2005.

[20] K. J. H. McGhan, Niagara2 Opens the Floodgates. In Microprocessor Report, December 2006

[21] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In MICRO-40, pp. 146-160, 2007.

[22] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. In HPCA, pp. 96-106, 2004.

[23] Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In PACT, pp. 135-145, 2004.

[24] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In ISCA-21, pp. 24-33, 1994.

[25] B. Sinharoy, R.N. Kalla, J. M. Tendler, R. J. Eickemeyer, J. B. Joyner, POWER5 System microarchitecture. IBM Journal of Research and Development, volume 49(4/5), pp. 505-521, 2005

[26] A. J. Smith. Cache memories. In ACM Computing Surveys, volume 14(3), pp. 473-530, 1982.

[27] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In ASPLOS-IX, pp. 234-244, 2000.

[28] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos. Spatial Memory Streaming. In ISCA, pp. 252-263, 2006.

[29] S. Somogyi, T. F. Wenisch, and et. al. Spatio-temporal memory streaming. In ISCA, pp. 69-80, 2009.

[30] S. Srinath et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In HPCA-13, pp. 63-74, 2007.

[31] M. K. Tcheun, H. Yoon and S. R. Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In ICPP, pp. 306-313, 1997.

[32] J. M. Tendler, J. S. Dodson, J.S. Fields, H. Le, B. Sinharoy, POWER4 system microarchitecture. IBM Journal of Research and Develop-ment, volume 46(1), pp. 5-25, 2002

[33] D. M. Tullsen and S. J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In ISCA-20, pp. 278-288, 1993.

[34] D. Wallin and E. Hagersten. Miss Penalty Reduction Using Bundled Capacity Prefetching in Multiprocessors. In IPDPS, pp. 12.1, 2003.

[35] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In ISCA, pp. 222-233, 2005.

[36] T.F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In HPCA, pp. 79-90, 2009.

Definiciones

Se definen aquí términos en castellano que aparecen en este documento teniendo un significado distinto del habitual.

Cobertura: Medida que refleja la cantidad de fallos, que en un sistema sin prebúsqueda sufriría un programa, que es capaz de eliminar el prebuscador.

$$C = \frac{\text{fallos eliminados por prebúsqueda}}{\text{fallos en el sistema sin prebúsqueda}}$$

Demanda: Petición de un bloque generada por una instrucción de acceso a memoria perteneciente al programa en ejecución.

Núcleo: Cada una de las unidades de procesamiento incluidas en un chip multiprocesador.

Precisión: Medida que refleja cuán bueno es el prebuscador prediciendo las direcciones de memoria de los datos que serán necesitados por el procesador.

$$A = \frac{\text{prebúsquedas útiles}}{\text{prebúsquedas lanzadas}}$$

Puntualidad: Medida que refleja cuánto se ajusta la llegada del bloque prebuscado a la instrucción de acceso a memoria que lo necesita. Se define como:

$$L = \frac{\text{prebúsquedas tardías}}{\text{prebúsquedas útiles}}$$

Polución: Medida que refleja cuánto molesta la prebúsqueda en la LLC. Se define como:

$$P = \frac{\text{fallos provocados por la prebúsqueda}}{\text{prebúsquedas lanzadas}}$$