Journal of Instruction-Level Parallelism 13 (2011) 1-14 Submitted 3/10; published 1/11

Multi-level Adaptive Prefetching based on Performance Gradient Tracking

Luis M. Ramos LUISMA@UNIZAR.ES
José L. Briz BRIZ@UNIZAR.ES
Pablo E. Ibáñez IMARIN@UNIZAR.ES
Victor Viñals VICTOR@UNIZAR.ES
Departamento de Informática e Ingeniería de Sistemas
Instituto de Investigación en Ingeniería de Aragón (I3A)
HiPEAC Network of Excellence
Universidad de Zaragoza
Zaragoza, E-50018 SPAIN

Abstract

We introduce a multi-level prefetching framework with three setups, respectively aimed to minimize cost (*Mincost*), minimize losses in individual applications (*Minloss*) or maximize performance with moderate cost (*Maxperf*). Performance is boosted in all cases by a sequential tagged prefetcher in the L1 cache, with an effective static degree policy. In both cache levels (L1 and L2), we also apply prefetch filters. In the L2 cache we use a novel adaptive policy that selects the best prefetching degree within a fixed set of values, by tracking the performance gradient. *Mincost* resorts to sequential tagged prefetching in the L2 cache as well. *Minloss* relies on an accurate, home-made, correlating prefetcher (PDFCM, *Differencial Finite Context Method Prefetcher*). *Maxperf* maximizes performance at the expense of slight performance losses in a small number of benchmarks, by integrating a sequential tagged prefetcher with PDFCM in the L2 cache.

1. Introduction

Computer applications behave very differently when referencing memory, and therefore it is hard to find a prefetching method suitable for all of them. Aggressive sequential or stream prefetchers usually boost the average performance because they yield a good coverage, but useless blocks and pollution may occur for some unfriendly applications leading to severe performance losses. Adaptive mechanisms used to cut losses and to reduce the pressure on the system often reduce peak performance [1]. More selective approaches are prone to meager coverage or timeliness. To sum it up, designing or choosing a prefetching technique means to set clear objectives to guide unavoidable trade-offs. Reasonable targets are to minimize cost, to cut performance losses for any application, or to boost overall performance. The origin of this contribution was the 1st Data Prefetching Championship (DPC-1) [2], where achieving the maximum performance within the environment provided by the organizers was the objective. We eventually contributed with three proposals, each of them addressing one of the aforementioned targets.

All the three proposals share a common framework, where we combine prefetching in the first (L1) and second (L2) cache levels. Prefetched blocks are always stored in the caches. In the L1 cache we use a sequential tagged prefetching with a degree policy we introduced in [1]. In the L2

cache we use either an adapted PDFCM, a sequential tagged or both. PDFCM (*Prefetching based on a Differential Finite Context Machine*) is a selective correlating prefetcher that we developped in [3]. The prefetching degree in the L2 cache is adaptive, but the mechanism is new, based on performance gradient. Basic prefetch filters are applied in both levels.

The rest of the paper is organized as follows. Section 2 introduces the related work. Section 3 explains the common framework mentioned above, and details the prefetching engines, degree policies and filters. Section 4 summarizes the three proposals as variants of the common framework. Section 5 explains hardware costs. Section 6 describes the simulation environment, the baseline architecture and the workload. Section 7 provides some results, and the last Section draws some conclusions.

2. Related Work

Sequential prefetching has been well-known for three decades. It prefetches the block or blocks that follow the current demanded block, and suits programs that reference consecutive memory blocks [4]. *Sequential tagged prefetching* does only issue a prefetch upon a cache miss or when a prefetched block is referenced for the first time, and it needs an extra bit per block. Sequential prefetching can be made more aggressive by applying *degree* or *distance*. Let us consider a stream of references a program is going to demand (a_i, a_{i+1}, a_{i+2},...), where a_i has been demanded by the program. Then a prefetcher can dispatch a_{i+1},...a_{i+n}, where n is the *prefetch degree*. Alternatively, it is also possible to prefetch only a_{i+n}, and then we say that n is the *prefetch distance*. Power4 and Power5 [5][6] used this idea, storing the prefetched blocks along three cache levels. Using *stream buffers* is a way of issuing several requests in a sequence [7]. It stores the prefetched blocks in dedicated buffers until they are referenced.

Sequential and stream prefetching trigger performance losses in hostile benchmarks. Filtering mechanisms have been proposed, but they all call for non-negligible hardware, like in *Dynamic Data Prefetch Filtering* [8]. SMS *(Spatial Memory Streaming)* is a more recent prefetcher that avoids loading useless blocks into the cache —an issue for sequential prefetching and stream buffers— at the cost of using three tables plus some extra logic [9].

Losses can also be reduced by tuning the prefetching degree or distance. Sequential prefetching with adaptive degree was first proposed in Dhalgren et al. [10], on multiprocessors, focusing on prefetching usefulness. Adaptive stream prefetching is explored in Srinath et al. [11], balancing usefulness, timeliness and pollution.

Correlating prefetchers (like SMS) are far more selective. They associate predictions to histories stored in a history table. In most of them, the recorded history stales, mega-sized tables are needed, or the number of table accesses multiplies. GHB-based prefetchers focuses on the first two problems [12]. The best performer in the family is PC/DC, where the PC of the loads missing in L2 is used as the key to index the history table. Performance is lower when using miss addresses instead of load PCs as a key. For this case, an adaptive mechanism was proposed that finds out an optimal tag size and prefetch degree [13]. We introduced PDFCM and compared it with PC/DC and other prefetchers in [3]. Like in TLB Prefetching [14] and PC/DC, PDFCM is based on deltas, but it takes fewer table references. It closely follows the DFCM value predictor [15]. A key property of this predictor is that stride sequences (deltas with the same value) take only one entry in the prediction table. We use PDFCM with little change in this proposal. There are some

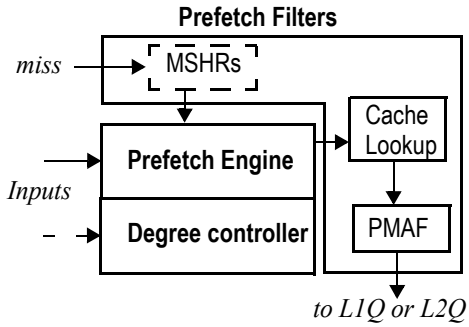

Figure 1: General organization of the prefetcher in a cache level. The MSHRs exists only in L2. PMAF: *Prefetch Miss Address File*.

meaningful differences between the PDFCM prefetcher and the DFCM value predictor [3]. For example, whereas value prediction applies on every instruction in the program, address prediction only applies to references missing in the L2 cache, considerably reducing table sizes.

We experimented several simple, cost-effective, adaptive strategies for distance or degree in [1], comparing them to PDFCM and SMS, and adaptive methods based on [10] and [11]. For this contribution we have picked up one of our proposals for the L1 cache. However, the adaptive method we use in L2 is new. It dynamically monitors the performance trend (IPC) of the workload by counting memory references per cycle. A similar idea was used to distribute resources in SMT processors [16]. We had not combined different prefetchers at different cache levels before. Details on the PDFCM and these two adaptive methods are provided in the following sections.

3. Prefetching Framework

Figure 1 shows the general setup of the prefetchers, valid for both L1 and L2 caches. We set three components at each level: a *prefetch engine*, a *degree controller* and *prefetch filters*. The prefetch engine in the L1 cache is a sequential tagged prefetcher. In the L2 cache we use PDFCM, sequential tagged prefetching, or both. The prefetch filters are similar for both L1 and L2 cache setups, except for the MSHRs, used only in the L2 cache. Whereas the degree follows a static strategy in the L1 cache, an adaptive automaton is used in the L2 cache. The next four subsections describe the prefetch engines, the two degree policies, and the prefetch filters.

3.1. Sequential Tagged Prefetch Engines

The L1 cache sequential prefetcher is fed with addresses that are either L1 cache misses or first references to a prefetched block in the L1 cache. The L2 sequential prefetcher is fed with addresses that are either L2 cache misses or first references to a prefetched block in the L2 cache, issued by either L1 cache demand misses or L1 cache prefetches. Both loads and stores are taken into account in both levels.

3.1.1. Implementing Degree in Sequential Prefetching

The degree automaton (Figure 2) increments the current block address (a_i) by one to generate the next prefetching address (a_{i+1}). When the prefetching degree is greater than 1, the degree

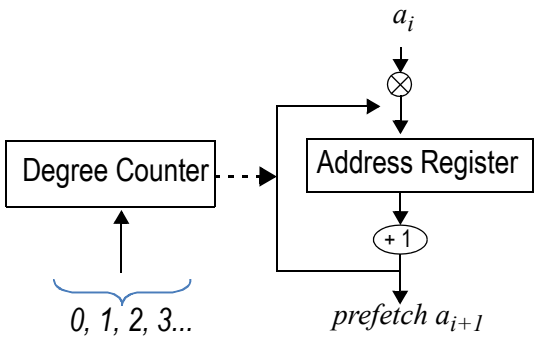

Figure 2: Sequential degree automaton.

automaton executes for as many cycles as pointed out by the degree counter, generating a single prefetch per cycle. If there is another miss, the automaton is reprogrammed.

3.2. The PDFMC Prefetch Engine

Like any markovian predictor, PDFCM aims to predict the next occurrence in a pattern. It considers sequences of differences *(deltas, δ)* between consecutive addresses issued by the same memory instruction (load or store). In this contribution, PDFCM is only trained with addresses that are either L2 cache misses or first references to a prefetched block in the L2 cache, issued by demand (i.e. not triggered by the L1 cache prefetches). We call this kind of addresses *training addresses*.

PDFCM uses the structures displayed on Figure 3.a, and operates according to the algorithms in Figure 4. Each *History Table* (HT) entry holds:

- The PC tag of a memory instruction.
- The last training address issued by a previous instance of this memory instruction (*last_addr*).
- The hashed sequence of *deltas* between recent training addresses issued by this memory instruction (*history*).
- Confidence bits.

The *history* field is used to index the *Delta Table* (DT), in order to find out the following probable delta.

PDFCM's operation is as follows. Throughout this explanation, the numbers in parenthesis refer to steps pointed out in both Figure 3 and Figure 4. When a memory instruction issues a training reference a_i, HT is indexed with the corresponding PC (0). In the instance of a miss, the HT entry is replaced and the operation ends. Otherwise, the PDFCM predictor is updated in the following way. First act_δ is computed by subtracting from a_i the last training address of the same instruction (a_{i-1}, read from HT) (1). On the other hand, DT is indexed with that HT history entry to read the last delta predicted for that sequence ($last\_\delta$) (2). If this delta does not match $actual\_\delta$, it is replaced with $actual\_\delta$ (3), and the confidence counter is decreased. Otherwise, the confidence is increased. A new history (H) is now computed by hashing the last history and $actual\_\delta$ (4). The HT entry is next updated (5). All in all, each PDFCM update takes up one read and one write in the HT, plus one read and one write in DT. If the confidence read from the HT entry is above a

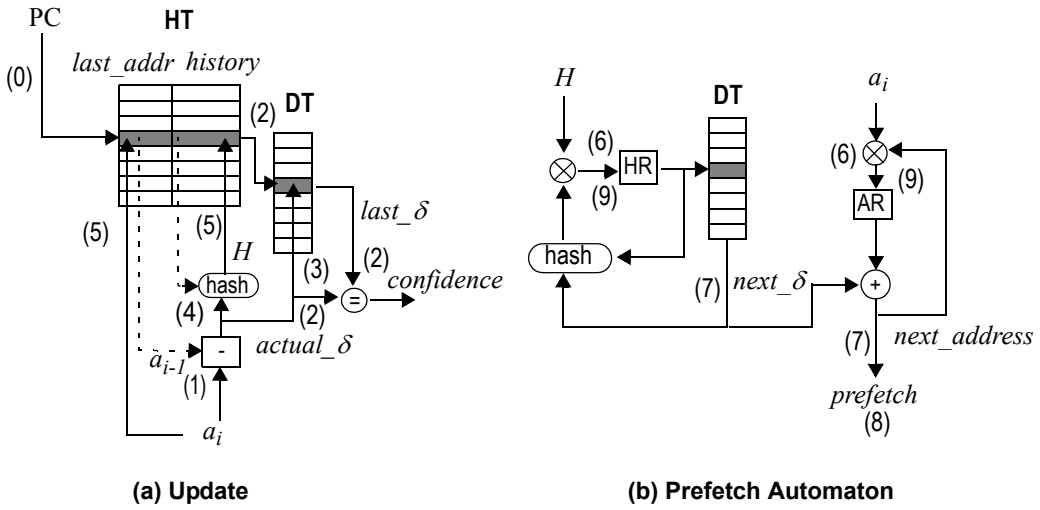

Figure 3: Prefetch based on DFCM. HT: *History Table,* DT: *Delta Table,* HR: *History Register,* AR: *Address Register.* The numbers in parenthesis refer to the steps described in the text explanation and match the ones used in the algorithms of *Figure 4.*

threshold, the Prefetch Automaton (Figure 3.b) is programed with the new history (HR=*H*) and the current address (AR=a_i) (6).

3.2.1. Implementing Degree in PDFCM

The Prefetch Automaton indexes DT with the history (HR) to get the next delta in the sequence (*next_δ*) (7) which is added to the current address (AR) to generate the next prefetching address (*next_address*) (8). The predicted delta (*next_δ*) is hashed with the content of the *History Register* (HR) (9). This yields a new (speculative) history that is stored again in the HR. The predicted address (*next_address*) is stored in the AR, considering it a speculative current address.

When the prefetching degree is greater than 1, the automaton executes the same steps ((7), (8) and (9)) at each processor cycle, for as many cycles as indicated by the degree. Note that only DT is accessed (one access per processor cycle for *degree* cycles).

We apply the hashing function FS R-5 used in DFCM, that yields the best results for finite context predictors [17]. In this function the length of the history *(order)* is a function of the logarithm of the number of DT entries (order = $\lceil n/5 \rceil$, with $n = 9$ in this proposal).

3.3. Degree Controllers

3.3.1. Degree 1-x (L1)

We apply in the L1 cache our static prefetching degree policy *Degree (1-x)*: on miss, prefetch with degree 1; on first use of a prefetched block, prefetch with degree *x* [1]. In this implementation, *x=4*.

```
Global types and variables:
    HT_entry fields: {tag; last_addr; history; confidence;}
    DT_entry DT[DT_SIZE];
    HT_entry HT[HT_SIZE];
    // Prefetch Automaton (actual registers in the implementation)
    int AR; //Address Register
    int HR; // History Register
Update (PC, a_i)
    PC; // address of the ld or st instruction
    a_i; // primary miss or 1st use of prefetched block
{   // local variables: are not registers in the implementation
    HT_entry hte; // temporal struct to hold an HT entry
    int H;  // temporal variable to hold the new history
    int last_δ; // temporal variable to hold last predicted δ
    int act_δ; // temporal variable to hold actual δ
(0) Look up HT with PC;
    if ( hit ) { // read HT entry and save it to a register
      hte = HT[PC];
    } else {
      replace HT entry; exit;
    }
    // check last predicted δ and update confidence
(1) act_δ = a_i - hte.last_addr;
(2) last_δ = DT[hte.history];
(2) if ( act_δ == last_δ ) {
      hte.confidence++;
    } else {
      hte.confidence--;
      // update DT
(3)   DT[hte.history] = act_δ;
    }
    // calculate new history
(4) H = hash(hte.history, act_δ );
    // update HT entry
(5) hte.history = H;
(5) hte.last_addr = a_i;
(5) HT[PC] = hte;
    if ( hte.confidence > THRESHOLD) {
      // program Prefetch Automaton
(6)   AR = a_i;
(6)   HR = H;
    }
} // Update end

Predict (void){
(7) next_δ = DT[HR];
(7) next_address = AR + next_δ;
(8) Prefetch (next_address);
(9) AR = next_address;
(9) HR = hash(HR, next_δ );
} // Predict end
```

Figure 4: Algorithms for implementing PDFCM. The numbers in parenthesis refer to the steps described in the text explanation and match the ones used in *Figure 3*.

3.3.2. Adaptive Degree by Tracking Performance Gradient (L2)

The prefetching degree in the L2 cache is dynamically tuned by tracking the program performance gradient. The rationale behind the mechanism is that we keep the trend (either increasing or decreasing the prefetching degree) as long as it helps performance, and we change it otherwise. Since we stick to the simulation environment provided by the DPC-1 we couldn't track the instruction count. We get around this problem by considering the references (to the L1 cache) issued per cycle as the performance metric. Other metrics could be tried in a different environment. We use an automaton with just two states (*increasing degree* and *decreasing degree*).

A *cycle counter* determines the duration of an *epoch* (a fixed number of cycles) whereas a *reference counter* records the number of demand references (loads and stores) issued (to the L1 cache) by the CPU over the epoch. A *previous epoch* register holds the count of the previous epoch. At the end of an epoch, the automaton switches the state between *increasing* and *decreasing* if *previous epoch* > *reference counter*. Then, it updates (increases / decreases) the prefetch degree as pointed out by the state. The degree fluctuates non-linearly, according to the following values: *0, 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64*. We have tried different number of cycle counts to define an epoch. Since we observed little change in the outcome, we have fixed it to 64 K cycles.

3.4. Prefetch Filters

The simulation environment provided by the DPC-1 contest did not provide any mechanism to filter secondary misses. Thus, *n* references to the same missing block produce *n* references to the next memory level, all of which will appear in the stream of references that feeds the prefetcher. The aim of using the MSHRs as a filter is to remove the secondary misses from that stream. We don't use MSHRs in the L1 cache, on account of the limit imposed by the DPC-1 contest rules. Our MSHRs hold up to 16 block addresses that have missed in the L2 cache and are being serviced from memory. Only references that miss in both the L2 cache and the MSHRs are allocated a new MSHR (following a FIFO policy), and feed the prefetcher.

Cache Lookup and *PMAF* (*Prefetch Memory Address File*) filter the output of the prefetcher. *Cache Lookup* eliminates prefetches to blocks that are already in the cache. PMAF eliminates prefetches to blocks that have already been issued to the next memory level. We use a dedicated port in both the L1 and L2 cache directories for prefetch lookups, a possibility left open by the contest rules. PMAF is a FIFO structure similar to the MSHRs. It holds up to 32 prefetch addresses that have already been issued but have not been serviced yet. Only prefetch addresses missing in the PMAF are allocated a new PMAF entry and sent to the next memory level. PMAF entries only keep a tag with the least significant 16 bits of the block address.

4. Three Goals, Three Proposals

We summarize our three proposals as three different setups of the framework introduced above, subject to our three objectives. All of them share a Sequential Tagged prefetch engine with degree policy *1-x* in the L1 cache, *adaptive degree by tracking performance gradient* in the L2 cache, and the filters explained in subsection 3.4. Only the L2 cache prefetch engine changes:

- ***Minimizing cost (Mincost)—*** Sequential Tagged.

- ***Minimizing losses (Minloss)—*** PDFCM.
- ***Maximizing performance (Maxperf)—*** This setup just matches the DPC-1 contest target itself. Here, the prefetch engine in L2 (Figure 1) embodies a PDFCM, along with a conventional sequential tagged prefetcher. The rest of the elements do not change in either level, excepting the lookup and PMAF filters in L2, that call for an extra port. Both the PDFCM and the sequential tagged prefetcher in L2 follow the same prefetching degree, managed by the adaptive degree controller used in L2 (subsection 3.3.2). This configuration yields the best performance considering the average of all the applications), and its hardware cost is negligible. However, it causes performance losses in some individual applications.

5. Hardware Cost

The DPC-1 contest set down a budget of 32 Kbits per proposal. The overall budget for ***Mincost***, ***Minloss*** and ***Maxperf*** respectively amounts to 1255 bits, 20784 bits, and 20822 bits. The two following subsections break down these figures, according to the cost of the components used in the L1 and L2 cache prefetchers.

5.1. L1 Cache Prefetcher

Sequential prefetch engine.- None.

Implementing 1-4 degree in sequential prefetch (35 bits).- A counter for monitoring the degree (3 bits) and an address register (32 bits).

Filtering mechanisms (512 bits).- There are no MSHR in the L1 cache. L1 cache lookup needs no storage. PMAF1 entries keep only a tag with the least significant 16 bits of each block address. PMAF1 takes up 512 bits (32 entries, 16 bits per entry).

5.2. L2 Prefetcher

PDFCM prefetch engine (19530 bits).- HT entry fields: PC tags and last_addr (16 bits each); history (9 bits); confidence (2 bits). 256 HT entries take up 11008 bits. DT has 512 entries (deltas), each 16 bits long (8192 bits in all). The code that implements this engine uses 14 local variables that add up 314 bits. Some of them could be shortened or even eliminated in a hardware implementation, though.

Implementing degree in PDFCM (64 bits).- Counter for monitoring the degree (7 bits), Address Register (32 bits), Delta Register (16 bits) and History Register (9 bits).

Implementing degree in sequential prefetch (38 bits).- Degree counter (6 bits) and address register (32 bits).

Adaptive degree by tracking performance gradient (131 bits).- Cycle counter (16 bits), instruction memory counter (16 bits), instruction count of the last epoch (*previous epoch,* 16 bits), array of degree values (13 entries x 6 bits), index register of the array (4 bits) and one state bit (*increasing / decreasing degree)*.

Filtering mechanisms (512 bits).- We do not consider the cost of the 16 MSHRs, as specified in the contest mail list. L2 cache lookup needs no storage. PMAF2 entries keep only a tag with the least significant 16 bits of each block address. PMAF2 takes up 512 bits (32 entries, 16 bits/entry).

Issue window	128-entry fully out-of-order
Issue rate	4 (max. 2 loads and 1 store)
Pipelining	15 stages, 1 lat inst. (except cache misses)
Branch prediction	perfect
L1 cache	32 KB, LRU, 8 way, no MSHRs
L2 cache	512 KB, LRU, 16 way, ld/use latency 20 cycles, bandwidth 1/cycle, no MSHRs
Memory	Latency 200 cycles, bandwidth 1/10 cycles
L1 cache & L2 cache request queues	1024 entries, FIFO, prefetches not prioritized vs. demands

Table 1: Baseline architecture parameters.

6. Evaluation Framework

The simulation environment used in this work is just the one provided by the DPC-1 organizers [2]. It is based on the CMP$im simulator [18]. The framework models a simple out-of-order core whose principal characteristics are shown in Table 1. Our baseline system features the most restricted configuration among the three ones considered in the DPC-1: 512KB L2 cache; one request per cycle from the L1 to the L2 cache, and a maximum of one request per every 10 cycles from the L2 cache to Memory. We have used the SPEC 2006 benchmark. We skip the first 40 billion instructions, and then the next 100 million instructions are executed. .

7. Results

Figure 5 and Figure 6 plot the speedups obtained by our three proposals with respect to the baseline system without prefetching. The simulation environment and the selected workload have been just described in the previous section. The rightmost group of columns stands for the geometric mean of speedups. Results for the other two configurations considered in the DCP are relatively similar. As it could be expected, *Mincost* and *Maxperf* respectively peak at the lowest (28.3%) and highest (32.5%) speedup figures, whereas *Minloss* performance situates just in between (30.7%). It must be considered that there are twelve applications whose global miss ratio falls below 0.2% in the baseline system (*perlbench, gobmk, sjeng, h264ref, gamess, gromacs, namd, deal II, povray, calculix, tonto,* and *wrf*). They roughly match the ones where prefetching does not achieve great results or even shows performance losses. Exceptions are *h264ref* and *tonto*, where little improvements appear. *Minloss* only shows losses in *astar* and *povray*, but they are definitely negligible. Indeed, *Minloss* is the only option that virtually cuts any losses in *astar*.

Figure 7 and Figure 8 focus on the effect of the adaptive degree policy we propose for the L2 cache prefetcher. We only show results for *Maxperf*. The cache and bandwidth configuration are the same we used in the previous experiment. The adaptive mechanism (bar *Adaptive*) achieves the higher performance in average, and it helps cutting losses in unfriendly cases. The rest of the bars

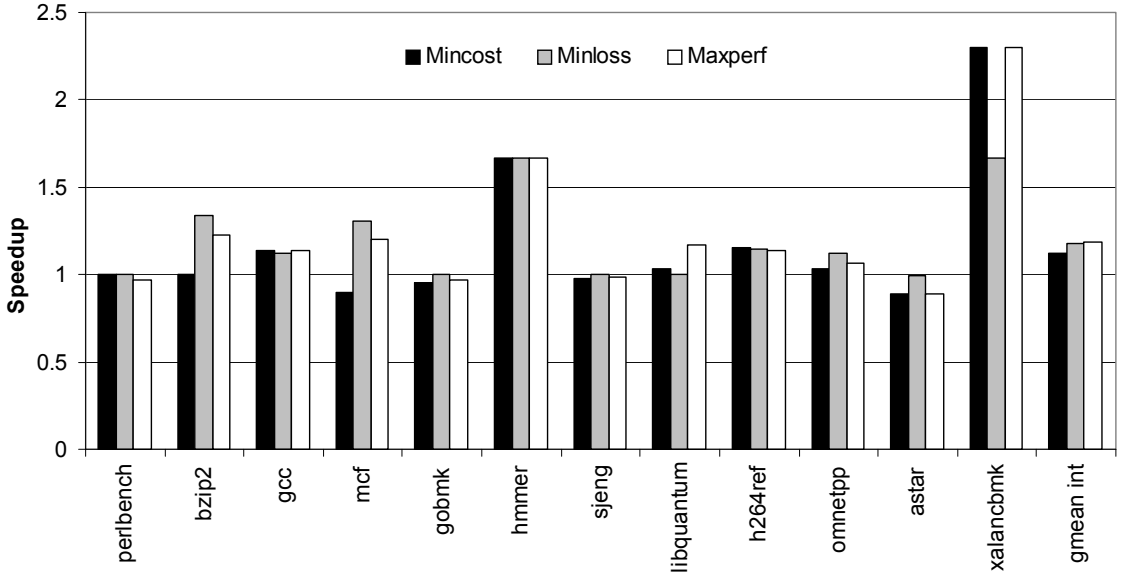

Figure 5: Breakdown of INT application speedups for the three proposals. The rightmost group of bars shows geometric means.

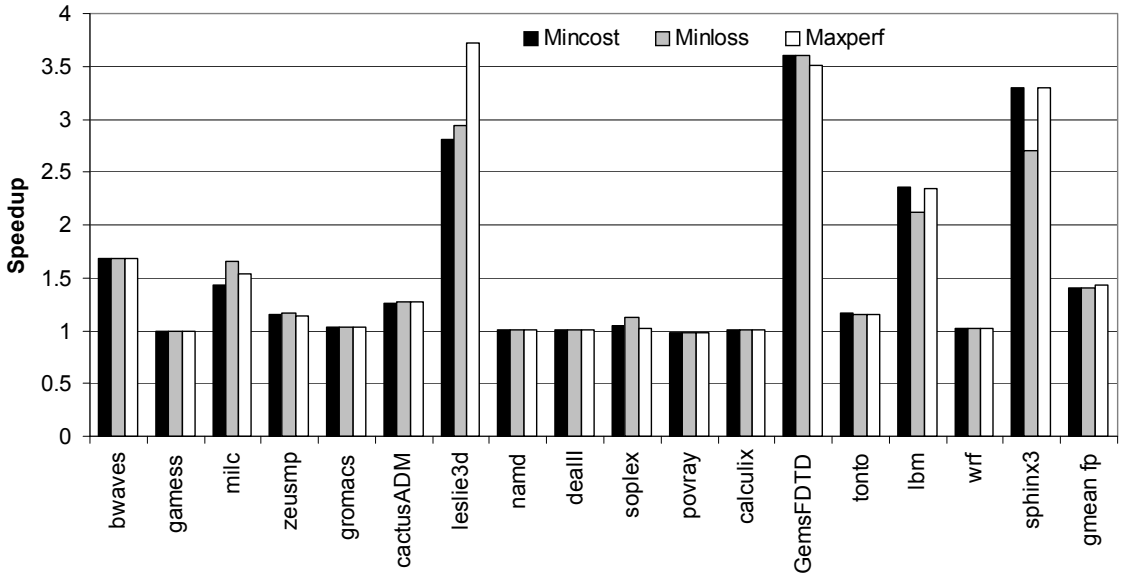

Figure 6: Breakdown of FP application speedups for the three proposals. The rightmost group of bars shows geometric means.

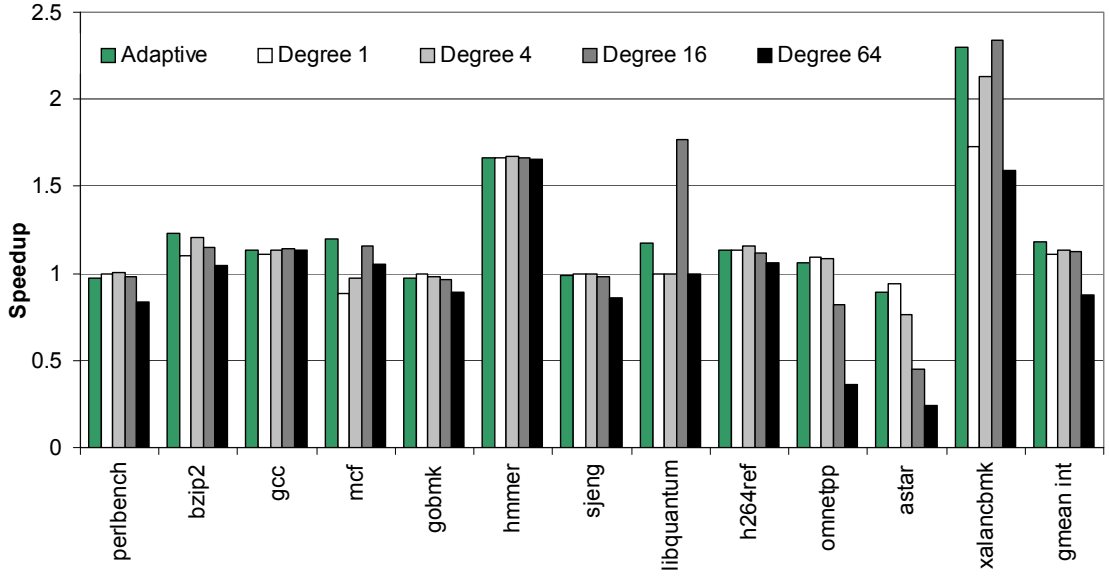

Figure 7: Breakdown of INT application speedups respect the baseline system for *Maxperf* with different degree policies in L2. The bar *Adaptive* matches the *Maxperf* bar in *Figure 5*.

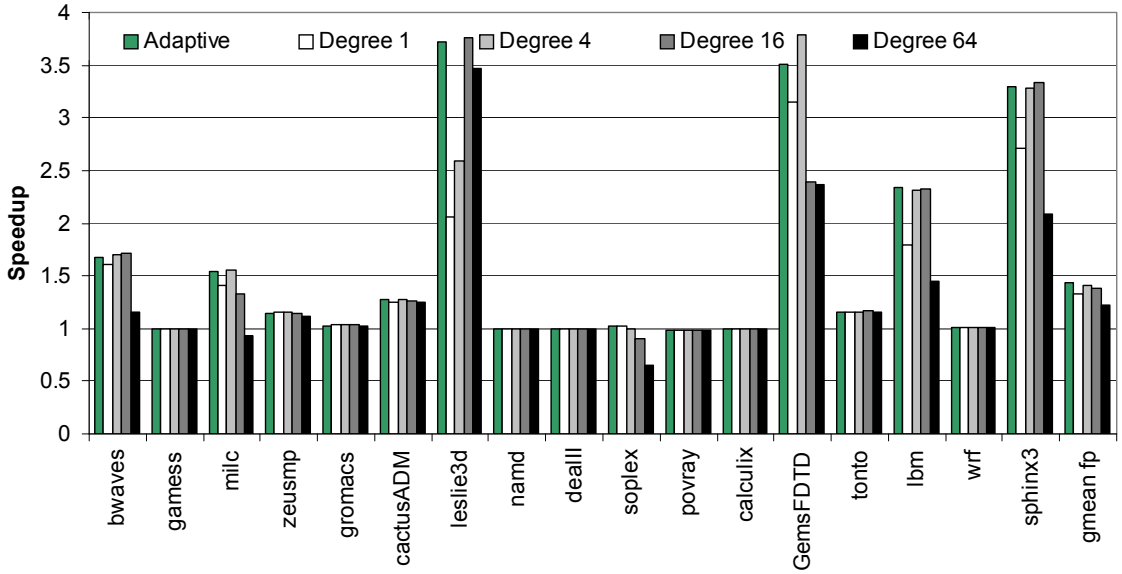

Figure 8: Breakdown of FP application speedups respect the baseline system for *Maxperf* with different degree policies in L2. The bar *Adaptive* matches the *Maxperf* bar in *Figure 6*.

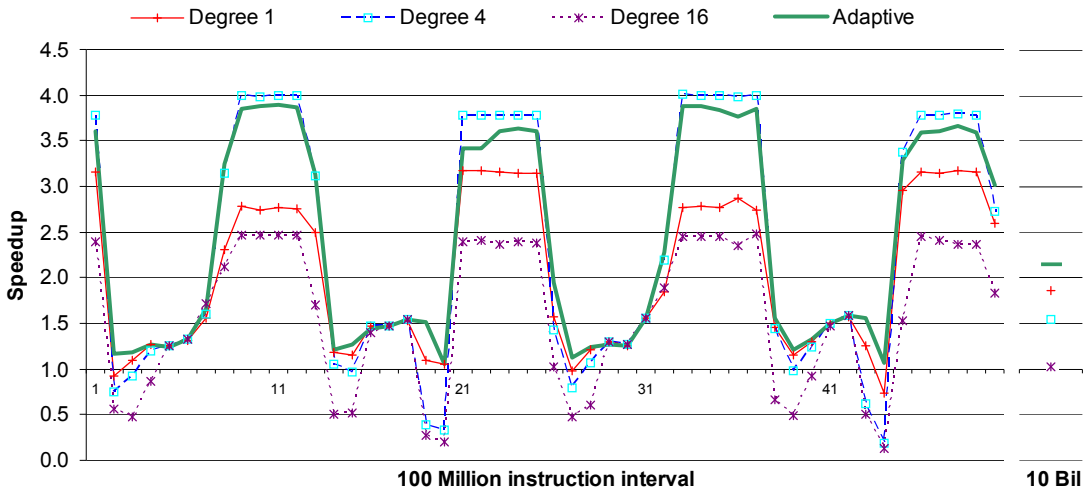

Figure 9: Speedups of *GemsFDTD* in 100 Million instruction intervals for Maxperf proposal with different degree policies. The last point is the speedup obtained in the 10 Billion instruction interval.

(degree n = 1, 4, 16, 64) show the speedup of the same prefetching scheme but applying a fixed prefetching degree n in the L2 cache. The optimal degree fluctuates along the different benchmarks. Thus, the best degree policy in *astar*, *GemusFDT* and *leslie3d* is respectively 1, 4 and 16. The speedup achieved by the Adaptive policy in all the benchmarks closely follows the best speedup achieved by a fixed degree policy in each case.

Actually, if the optimal degree changes along the different phases of a given program, the adaptive degree policy can even outperform the best fixed-degree option for that program. For example, we have increased the number of simulated instructions from 100 millions in the former base experiment, up to 10 billion. Figure 9 shows, for *GemsFDTD*, the speedup obtained with fixed-degree policies (1, 4 and 16) and with the adaptive policy. We show results per 100 million instruction interval. Intervals are consecutive, and start at the same execution point where we started the former experiments.

The first simulation interval in Figure 9 matches the data of Figure 7 and Figure 8. The best fixed degree is 4 (speedup 3.78), followed by degree 1 (speedup 3.15) and degree 16. The adaptive degree policy scores a speedup of 3.61, close to the best speedup achieved by a fixed-degree policy. However, if we track the lines along the rest of the plot, the optimal degree fluctuates along the program run. The fixed-degree policy with degree 4 does not always keeps the best results, whereas the adaptive degree policy adapts to each phase, and gets results close to the best option of each interval.

On the right side of Figure 9 we show the speedup of each policy considering the complete execution of 10 billion instructions. The adaptive degree gets a speedup of 2.15, well above the 1.86 figure achieved by the best fixed-degree policy.

8. Conclusions

Prefetching performance depends on both the memory hierarchy and the workload, so different targets lead to different designs. Moreover, there are many applications where prefetching has a slim chance of improving performance, e.g. SPEC CPU 2006 includes 12 applications with negligible L2 cache misses. However, on average, prefetching performs pretty well, since dramatic speedups can be achieved in friendly applications. Therefore, prefetching is worth implementing especially if we keep cost low.

In this paper, we propose in this contribution a common multi-level prefetching framework that can fit three different prefetching engines, respectively targeted to minimize cost (*Mincost*), performance losses (*Minloss*) or to maximize performance (*Maxperf*), although all of them keep cost fairly low. *Mincost* yields the lowest performance but it takes just over 1 KB. *Minloss* closely follows *Maxperf* performance, but it virtually cuts all performance losses. If average maximum performance matters, *Maxperf* costs just one hundred bits more than *Minloss*, but performance losses appear in some applications. The new adaptive policy we introduce to manage the prefetching degree at the second cache level proves to be effective, and merely takes 131 bits.

Acknowledgements

This work was supported in part by grants TIN2007-66423 and TIN2010-21291-C02-01 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-2 NoE (European FP7/ ICT 217068).

References

[1] L. M. Ramos, J. L. Briz, P. E. Ibañez and V. Viñals, “Low-cost adaptive hardware prefetching”, *LNCS* 5168, pp. 327–336, Springer-Verlag Berlin Heidelberg, 2008.

[2] L. M. Ramos, J.L. Briz, P.E. Ibáñez and V. Viñals, “Multi-level Adaptive Prefetching based on Performance Gradient Tracking”, *1st JILP Data Prefetching Championship*, Internet: www.jilp.org/dpc, Raleigh, Feb 2009.

[3] L. M. Ramos, J. L. Briz, P. E. Ibañez and V. Viñals, “Data prefetching in a cache hierarchy with high bandwith and capacity”, *ACM Computer Architecture News* 35 (4), Sept. 2007, pp. 37-44.

[4] A.J. Smith, “Sequential program prefetching in memory hierarchies”, *IEEE Transactions on Computers*, 11(12), pp.7-21, Dec. 1978.

[5] R. Kalla, B. Sinharoy and J. Tendler, “IBM Power5 chip: a dual-core multithreaded processor”, *IEEE Micro*, 24(2), pp. 40-47, 2004.

[6] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le and B. Sinharoy, “Power4 system microarchitecture”, *IBM Journal of Research and Development* 46(1), pp. 5-26, 2002.

[7] N. Jouppi, “Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers”, in *Procs. of the 17th ISCA*, Seattle, WA, 1990.

[8] X. Zhuang and H. S. Lee, "Reducing cache pollution via dynamic data prefetch filtering", *IEEE Trans. on Computers* 56 (1) Jan. 2007, pp. 18-31.

[9] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos, "Spatial memory streaming", in *Proc. of ISCA-33*, pp. 252-263, 2006.

[10] F. Dahlgren, M. Dubois and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared-memory multiprocessors", in *Proc. of the International Conference on Parallel Processing*, CRC Press, Boca Raton, Fla.,1993, pp. 156-163.

[11] S. Srinath, O. Mutlu, H. Kim and Y. N. Patt, "Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers", in *Proc. of HPCA-13*, pp. 63-74.

[12] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer", *IEEE Micro* 25 (3), pp. 90-97, May/Jun. 2005.

[13] K. J. Nesbit, A. S. Dhodapkar and J. E. Smith, "AC/DC: An adaptive data cache prefetcher", in *Proc. of the 13th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Sept. 2004.

[14] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: an application-driven study", in *Procs. of the 29th ISCA*, May 2002.

[15] B. Goeman, H. Vandierendonck and K. Bosschere, "Differential FCM: increasing value prediction accuracy by improving table usage efficiency", in *HPCA-7*, pp. 207-218, Monterrey Mexico, 2001.

[16] S. Choi and D. Yeung, "Learning-based SMT processor resource distribution via hill-climbing", in *Proc. International Symposium on Computer Architecture ISCA-33*, pp: 239-251, 2006.

[17] Y. Sazeides and J. E. Smith, "Implementations of context based value predictors", TR ECE97-8, Dept. of Electrical and Computer Engineering, Univ. Wiscosin-Madison, Dec. 1997.

[18] A. Jaleel, R. S. Cohn, C. K. Luk and B. Jacob. "CMP$im: A Pin-based on-the-fly multi-core cache simulator", in Proc. Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), pages 28–36, Beijing, China, 2008.