Multi-level Adaptive Prefetching based on Performance Gradient Tracking

Luis M. Ramos, José Luis Briz, Pablo E. Ibáñez, Víctor Viñals
Depto. Informática e Ing. de Sistemas and I3A, Univ. Zaragoza. HiPEAC Network of Excellence
{luis.ramos,briz,imarin,victor}@unizar.es

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 L1, with an effective static degree policy. In both L1 and L2, we also apply prefetch filters. In L2 we place a brand-new 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 L2 as well. *Minloss* relies on PDFCM, an accurate, home-made, correlating prefetcher. *Maxperf* maximizes performance at the cost of slight individual losses, by integrating in L2 a sequential tagged prefetcher with PDFCM.

1. INTRODUCTION

Prefetching performance depends on both the applications and the memory hierarchy. Thus, no prefetching method has succeeded for every application so far. Aggressive sequential or stream prefetchers usually boost the average performance because they yield a good coverage, but useless blocks and pollution may rocket for some unfriendly applications leading to severe performance losses. Adaptive mechanisms used to cut losses and to reduce the pressure on the system often make peak performance decrease [9]. More selective approaches tend to exhibit meager coverage or timeliness. To sum it up, designing or choosing a prefetching technique means to set down 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. In this contribution we propose three approaches, each of them addressing one of those 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 L1 we use a sequential tagged prefetching with a degree policy we introduced in [9]. In L2 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 of our own [10]. The prefetching degree in L2 is adaptive, but the mechanism is brand-new, based on performance gradient. Basic prefetch filters are applied in both levels.

Supported by D.G.A. (*Grupo Consolidado Investigación),* Spanish MEC TIN2007-66423, and the European HiPEAC NoE, FP6-IST-004408

The rest of the paper is organized as follows. Section 2 introduces the related work. Section 3 explains the aforementioned common framework, and details the prefetching engines, degree policies and filters. Section 4 sumarizes the three proposals as variants of the common framework. Section 5 gathers hardware costs, and proves how the proposals follow the contest rules. Section 6 provides some results, and the last Section draws with some conclusions.

2. RELATED WORK

Sequential prefetching has been known for three decades. It prefetches the block or blocks that follow the current demanded block, and suits programs that reference consecutive memory blocks [12]. *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 [6][15] profited from this idea, storing the prefetched blocks along three cache levels. Using *stream buffers* is a way of issuing several requests in a sequence [4]. 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* [16]. SMS *(Spatial Memory Streaming)* is a more recent prefetcher that avoids loading into the cache useless blocks —an issue for sequential prefetching and stream buffers— at the cost of using three tables plus some extra logic [13].

Losses can also be reduced by tuning degree or distance. Sequential prefetching with adaptive degree was first proposed in [2], on multiprocessors, focusing on prefetching usefulness. Adaptive stream prefetching is explored in [14], 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-

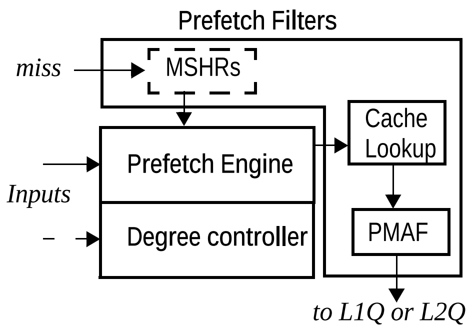

Figure 1. General organization of the prefetcher in a cache level. The MSHRs exists only in L2. PMAF: *Prefetch Miss Address File*.

sized tables are needed, or the number of table accesses multiplies. GHB-based prefetchers focused on the first two problems [7]. 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 address 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 [8]. We introduced PDFCM and compared it with PC/DC and other prefetchers in [10]. Like in TLB Prefetching [5] and PC/DC, PDFCM is based on deltas, but it takes fewer table references. It closely follows the DFCM value predictor [3]. A key property of this predictor is that stride sequences (sequences of 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 meaningful differences between the PDFCM prefetcher and the DFCM value predictor [10]. For example, whereas value prediction applies on every instruction in the program, address prediction only applies to references missing in L2, considerably lowering table sizes.

We experimented several simple, cost-effective adaptive strategies for distance or degree in [9], comparing them to PDFCM and SMS, and adaptive methods based on [2] and [14]. For this contribution we have picked up one of our proposals for L1. However, the adaptive method we use in L2 is brand-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 [1]. 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. We set three components at each level: a *prefetch engine*, a *degree controller* and *prefetch filters*. The prefetch engine in L1 is a sequential tagged prefetcher. In L2 we use PDFCM, sequential tagged or both. The prefetch

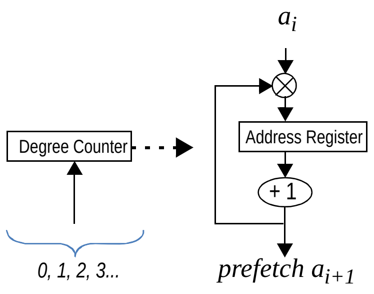

Figure 2. Sequential degree automaton

filters are similar for both L1 and L2 setups, except for the MSHRs, used only in L2. Whereas the degree follows a static strategy in L1, an adaptive automaton is used in L2. The next four subsections describe the prefetch engines, the two degree policies, and the prefetch filters.

3.1. Sequential Tagged prefetch engines

The L1 sequential prefetcher is fed with addresses that are either L1 misses or first references to a prefetched block in L1.

The L2 sequential prefetcher is fed with addresses that are either L2 misses or first references to a prefetched block in L2, issued by either L1 demand misses or L1 prefetches.

Both loads and stores are taken into account in both levels.

Implementing degree in sequential prefetching.- The degree automaton (Figure 2) increases 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 automaton executes for as many cycles as pointed out by the degree counter, generating a single prefetch by cycle.

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 misses or first references to a prefetched block in L2, issued by demand (i.e. not triggered by L1 prefetches). In which follows, we call this kind of addresses *training addresses*.

PDFCM uses the structures displayed on Figure 3.a. Each *History Table* (HT) entry holds: a) the PC tag of a memory instruction, b) the last training address (a_{i-1}) issued by a previous instance of this memory instruction (LA), c) the hashed sequence of *deltas* between recent training addresses issued by this memory instruction (*history*), and d) confidence bits. The *history* field is used to index the *Delta Table* (DT), in

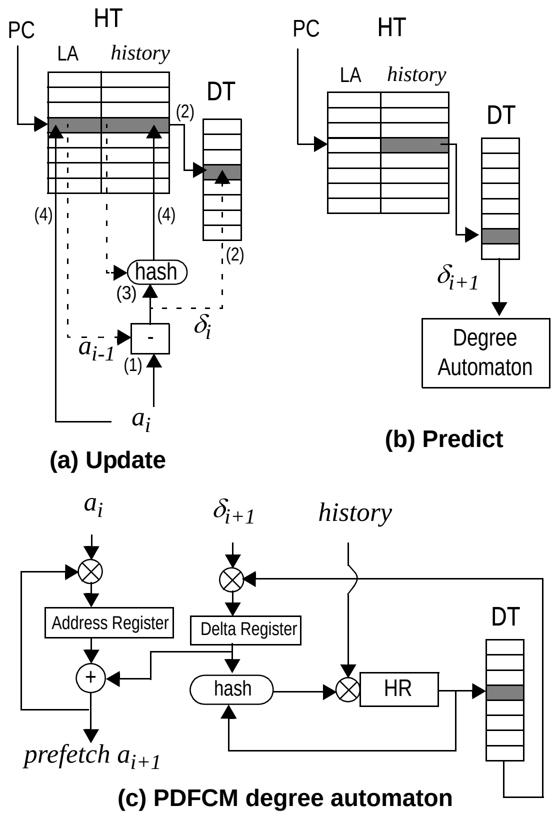

Figure 3. Prefetch based on DFCM. HT: *History Table;* **DT:** *Delta Table*. **LA:** *last address field in HT.* **HR:** *History Register.*

order to find out the following probable delta.

PDFCM operation is as follows. When a memory instruction issues a training reference a_i, HT is indexed with the corresponding PC. If miss, the HT entry is replaced and the operation ends. Otherwise, the PDFCM predictor is updated in the following way. First δ_i is computed by subtracting from a_i the last training address of the same instruction (a_{i-1}, read from HT, see Figure 3.a, (1)). On the other hand, DT is indexed with that HT entry's history to read the delta predicted for that sequence. If this delta does not match δ_i, it is replaced with δ_i (Figure 3.a (2)), and the confidence counter is decreased. Otherwise, the confidence is increased. A new history is now computed by hashing the last history and δ_i (Figure 3.a (3)). The HT entry is next updated (Figure 3.a (4)). All in all, each PDFCM update takes up one read and one write in HT, plus one read and one write in DT. The prediction takes place if the confidence read from the HT entry is above a threshold. In that case the new history indexes DT to get the next delta in the sequence (δ_{t+1}, Figure 3.b), which is then used to produce the prefetching address a_{i+1}.

We apply the same hashing function used in DFCM, FS R-5, that yields the best results for finite context predictors [11]. In this function the length of the history *(order)* is a function of the logarithm of the number of DT entries (order $= \lceil n/5 \rceil$, $n = 9$ in this proposal).

Implementing degree in PDFCM.- The degree automaton (Figure 3.c) adds up the current address (a_i) and the predicted delta (δ_{i+1}) to generate the next prefetching address (a_{i+1}). When the prefetching degree is greater than 1, the degree automaton executes the following steps each processor cycle, for as many cycles as indicated by the degree. The predicted delta (δ_{i+1}) is hashed with the content of the *History Register* (HR, Figure 3). This yields a new (speculative) history. The latter is used to select a new (speculative) delta in DT. Taking a_{i+1} as the next current address, it produces a new prefetching address. Note that only DT is accessed (one access per processor cycle for *degree* cycles).

3.3. Degree controllers

3.3.1 Degree 1-x (L1)

We apply in L1 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* [9]. In this implementation, *x=4*.

3.3.2 Adaptive degree by tracking performance gradient (L2)

The prefetching degree in L2 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 don't have the instruction count, we have considered references (to L1) 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 L1) by the CPU over an 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. 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 got little change in the outcome, we have fixed it to 64 K cycles.

3.4. Prefetch Filters

The simulation environment given for this contest does 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 L1, on account of the limit imposed by the contest rules. Our MSHRs hold up to 16 block addresses that have missed in L2 and are being serviced from memory. Only references that miss in both L2 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 L1 and L2 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 keep only a tag with the least significant 16 bits of the block address.

4. THREE GOALS, THREE PROPOSALS

We sumarize here 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 L1, *adaptive degree by tracking performance gradient* in L2, and the filters explained in subsection 3.4. Only the prefetch engine changes.

- ***Minimizing cost (Mincost)—*** Sequential Tagged.
- ***Minimizing losses (Minloss)—*** PDFCM in L2
- ***Maximizing performance (Maxperf)—*** Since maximum performance is targeted in the contest rules, we also submit an additional code setup where the prefetch engine in L2 (Figure 1) comes with 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 a little more average performance at an almost negligible cost, but it might cause performance losses in some individual applications.

5. HARDWARE COST AND RULE COMPLIANCE

The two following subsections break down the cost of the componentes used in the L1 and L2 prefetchers. The last subsection sumarizes the overall cost for each one of the three proposals.

5.1. L1 prefetcher *(547 bits)*

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 L1. L1 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 LA (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 ruled out 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 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).

5.3. Overall budget per proposal

According to the previous details, the overall budget for ***Mincost***, ***Minloss*** and ***Maxperf*** respectively amounts to 1255 bits, 20784 bits, and 20822 bits.

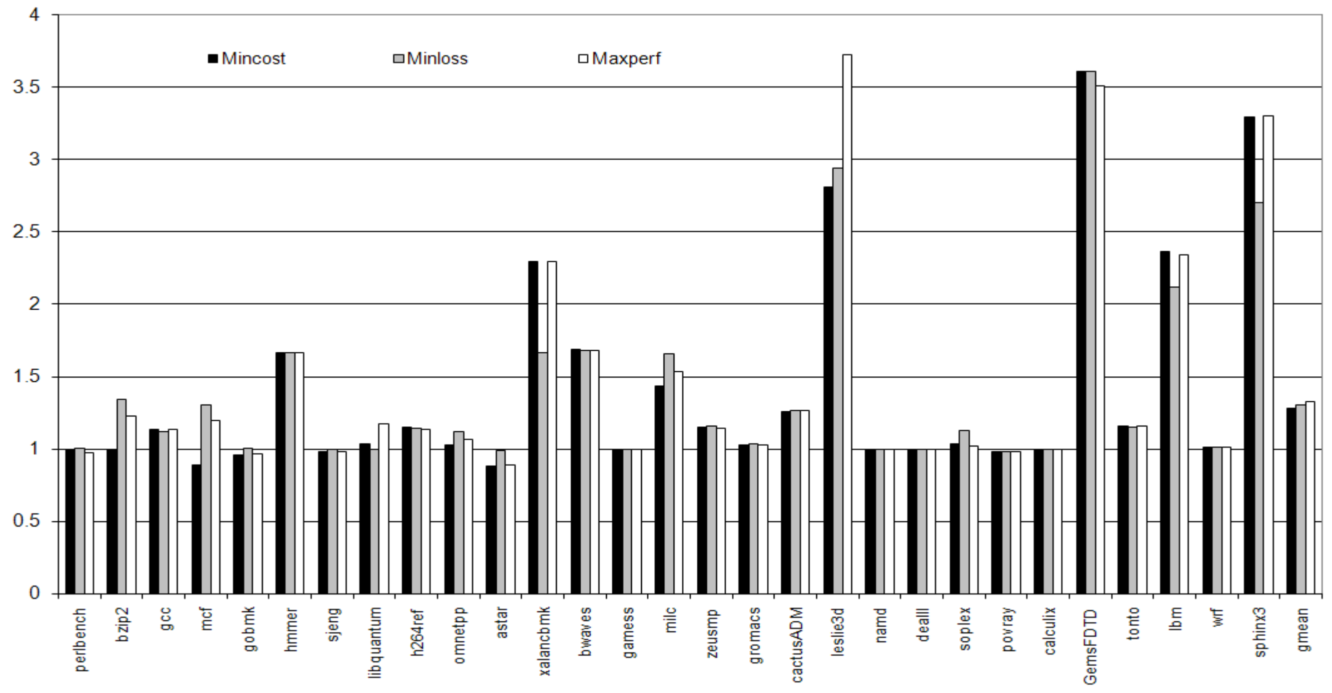

Figure 4. Breakdown of application speedups for the three proposals. The rightmost group of bars shows geometric means.

6. RESULTS

Figure 4 plots speedups obtained by our three proposals with respect a baseline system without prefetching. We used the environment provided by the DPC-1, and SPEC CPU 2006 as the workload. We apply a warming period of 40 billion instructions, and then the next 100 million instructions are executed. The baseline system features the most restricted configuration among the three ones considered in the DPC-1 (512KB L2; one request per cycle from L1 to L2, and a maximum of one request per every 10 cycles from L2 to Memory). The rightmost group of columns stands for the geometric mean of speedups. Results for the other two configurations considered in the DCP-1 are relatively similar. As it could be expected, *Mincost* and *Maxperf* respectively peak at just the lowest (28.3%) and highest (32.5%) speedup figures in average, 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 really negligible. Indeed, *Minloss* is the only option that virtually cuts any losses in *astar.*

Figure 5 focuses on the effect of the adaptive degree policy we propose for the L2 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* in Figure 5) achieves the higher performance in average, and it helps cutting losses in unfriendly cases. The rest of the bars (degree n = 1, 4, 16, 64) show the speedup of the same prefetching scheme but applying a fixed prefetching degree n in L2. The optimal degree fluctuates among 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.

7. 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 prefetfching chances for improving performance are really slim (SPEC CPU 2006 includes twelve applications out of twenty nine with negligible L2 cache misses, for example). However, if we boil it down to average results, prefetching performs pretty well, since dramatic speedups can be achieved in friendly applications. All in all, prefetching is worth implementing especially if we keep cost low.

For this reason, we propose in this contribution a common multi-level prefetching framework that can fit three differ-

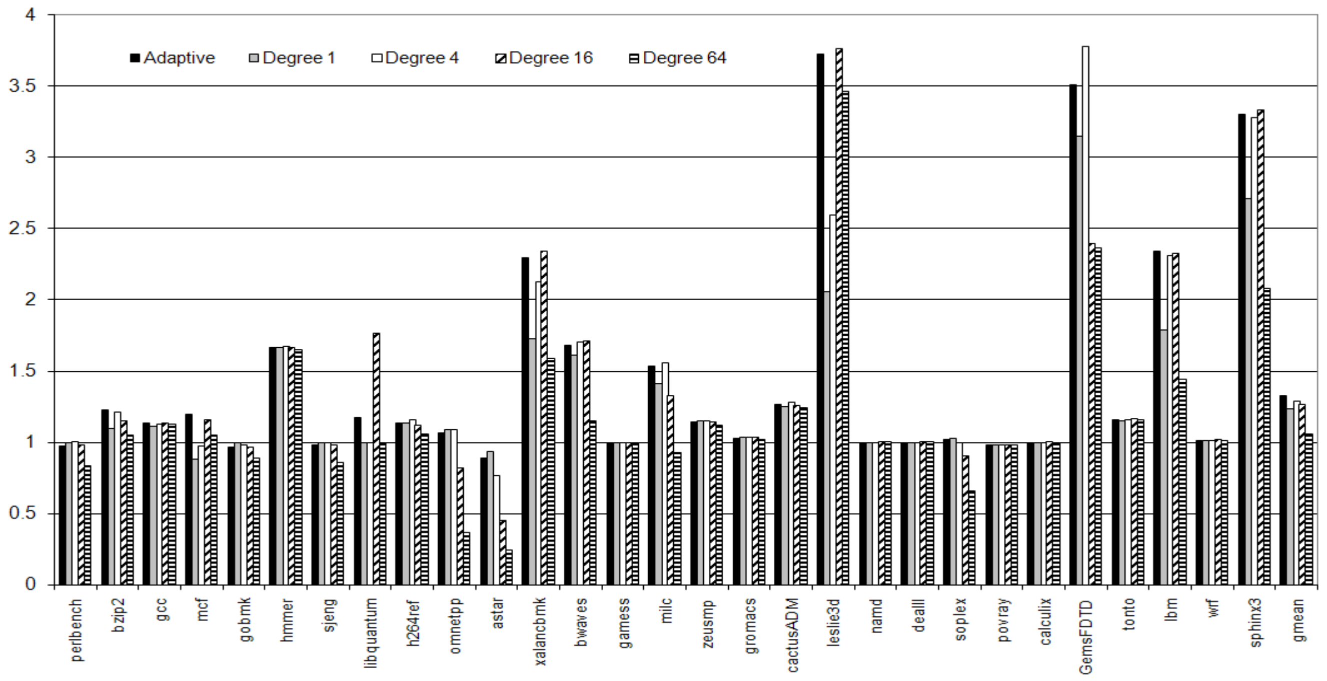

Figure 5. Breakdown of speedups respect the baseline system for Maxperf with different degree policies in L2. The bar Adaptive matches the Maxperf bar in Figure 4.

ent 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 lower performance but it takes just over 1 KB. *Minloss* closely follows *Maxperf* performance, but cutting almost all performance losses. If average maximum performance matters, *Maxperf* costs just one hundred bits more than *Minloss*, but performance losses appears in some applications. Last but not least, 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.

REFERENCES

[1] S. Choi and D. Yeung. “Learning-based SMT processor resource distribution via hill-climbing”. ISCA-33, pp: 239-251, 2006.

[2] F. Dahlgren et al, Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors. ICPP, CRC Press, Boca Raton, Fla.,1993, pp. 156-163.

[3] B. Goeman et al. “Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency”. In HPCA-7 pp. 207-218. Monterrey, Mexico 2001.

[4] 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.

[5] G. B. Kandiraju and A. Sivasubramaniam. “Going the Distance for TLB Prefetching: An Application-driven Study”. In Procs. of the 29th ISCA, May 2002.

[6] R. Kalla et al. IBM Power5 chip: A dual-core multithreaded processor. IEEE Micro, 24(2): 40-47, 2004.

[7] 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.

[8] K. J. Nesbit et al. “AC/DC: An Adaptive Data Cache Prefetcher”. In Proc. of the 13th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT) Sept. 2004.

[9] Ramos, L.M et al. "Low-cost Adaptive Hardware Prefetching". LNCS 5168, pp. 327–336. Springer-Verlag Berlin Heidelberg 2008.

[10] Ramos, L.M et al. "Data prefetching in a cache hierarchy with high bandwith and capacity".ACM Computer Architecture News 35 (4) Sept. 2007: 37-44.

[11] 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.

[12] A.J. Smith, “Sequential Program Prefetching in Memory Hierarchies”, IEEE Transactions on Computers, 11(12), pp.7-21, Dec. 1978.

[13] Somogyi, T. F. et al. “Spatial Memory Streaming”. ISCA-33 pp. 252-263.2006.

[14] S. Srinath et al. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. HPCA-13 pp. 63-74.

[15] J. M. Tendler et al. Power4 system microarchitecture. IBM Journal of Research and Development, 46(1): 5-26, 2002.

[16] X. Zhuang and Hsien-Hsin S. Lee. Reducing Cache Pollution via Dynamic Data Prefetch Filtering. IEEE Trans. on Computers 56 (1) Jan. 2007: 18-31.