Data prefetching in a cache hierarchy with high bandwidth and capacity

Luis M. Ramos, José Luis Briz, Pablo E. Ibáñez, Victor Viñals
Dept. Informática e Ing. de Sistemas and Instituto de Investigación en Ingeniería de Aragón (I3A)
Univ. de Zaragoza, Spain
{luis.ramos,briz,imarin,victor}@unizar.es

Abstract

In this paper we evaluate four hardware data prefetchers in the context of a high-performance three-level on chip cache hierarchy with high bandwidth and capacity. We consider two classic prefetchers (Sequential Tagged and Stride) and two correlating prefetchers: PC/DC, a recent method with a superior score and low-sized tables, and P-DFCM, a new method. Like PC/DC, P-DFCM focuses on local delta sequences, but it is based on the DFCM value predictor. We explore different prefetch degrees and distances. Running SPEC2000, Olden and IAbench applications, results show that this kind of cache hierarchy turns prefetching aggressiveness into success for the four prefetchers. Sequential Tagged is the best, and deserves further attention to cut it losses in some applications. PC/DC results are matched or even improved by P-DFCM, using far fewer accesses to tables while keeping sizes low.

1. Introduction

Hardware data prefetching techniques are fairly efficient if data are timely prefetched and eventually demanded by the CPU, if they do not replace other contents immediately demanded, and if they do not waste bandwidth and other critical resources [6]. These conditions depend on both applications and memory hierarchy, which implies two facts. Firstly, no prefetching method succeeds for every application so far, and therefore new prefetchers are being proposed. Secondly, memory hierarchies and processors evolve: current on-chip huge memory hierarchies offer a new arena and call for the evaluation of the classic and new prefetching approaches.

The main purpose of this article is to explore the prefetching capabilities of a high performance on-chip cache hierarchy. We simulate a detailed model similar to that of Itanium 2, where sizes and bandwidth are far generous regarding the cache hierarchies often used in current prefetching evaluations. We compare four prefetchers. Sequential Tagged and Stride are almost compulsory for our purpose. Correlating prefetchers have been an attractive approach since they appeared, and we have chosen PC/DC because it is a recent proposal with a superior score [8][16][17][18]. Since PC/DC relies on a table with linked lists, we have arranged a new correlating prefetcher with a more traditional organization that needs far less acesses to tables. It is based on DFCM, a value predictor very precise for detecting *delta* pattern streams (differences between consecutive values) [7].

Partially granted by the Aragonian Government (Research Group n. 48, BOA 122, 15 Oct. 2004) and the Spanish Ministry of Education and Science /European Union FEDER, TIN2004-07739-C02-02. Authors are members of the European HiPEAC Network of Excellence.

A way of increasing the timeliness of prefetching consist of increasing the *prefetch 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 n is the *prefetch distance*. Increasing the prefetch degree has always been a good theoretical choice, yet highly limited by real resources, namely cache size and bandwidth, and for this reason we experiment with these parameters.

The rest of the paper is organized as follows. Section 2 reviews the two classic prefetchers and situates PC/DC in the context of correlating prefetchers. Section 3 introduces the new P-DFCM prefetcher. Section 4 presents the baseline architecture, focusing on our detailed cache memory model, and specifies implementation parameters and details of the four methods we compare. Section 5 presents and discuses the experimental results. We finally draw some conclusions and mention future work in the last Section.

2. Related work

Sequential prefetching has been known for three decades. It prefetchs the block or blocks that follow the current demanded block, and suits programs that reference consecutive memory blocks [24][12]. *Sequential tagged prefetching* does only issue a prefetch upon a cache miss or when a block is referenced for the first time, and it needs an extra bit per block. These methods tend to issue many prefetches that are not used by the CPU *(useless prefetches)*, but they increase performance on a broad range of applications at a low cost.

Stride prefetching identifies and predicts accesses to memory addresses separated by a constant distance. Conventional *stride prefetching* uses a Load Table (LT) indexed by the program counter (PC) that associates strides to the loads following this kind of memory access pattern [1]. When address *a* is referenced by a load that hits in the table, and the matching entry indicates that the load is following a stride pattern, the prefetch controller issues the addresses *a+s*, where *s* is the associated stride. The size of the table can be much reduced without severe performance losses by applying *on-miss insertion* in the LT [10]. In this case only a load that misses in the cache is stored in the LT, but the entry is updated whenever the load hits in the table, independently of whether it misses or not in the cache.

During the past decade, both sequential and stride prefetching were actively explored, and specialized prefetchers were also proposed for tracking other memory accesses [25]. Several proposals were specifically targeted to pointer-intensive applications [3][4][19][26]. *Correlating prefetchers* apply to a broader range of applications with good results, predicting future addresses from tables that record the past memory program behaviour. They generalize the stride table by registering the stream of addresses associated either to the load PC or to an address that misses, usually in the second cache level (L2). The seminal idea handled miss address streams as Markov's chains and was extended or modified in different ways [9][11][13][14][15]. *Markov prefetching* stores the miss address streams that followed an address that missed in L2 in the past [11]. Each table entry represents a node in a Markovian graph, and its list of addresses represents the arcs with the higher probabilities. In order to decrease the size of the table, *Tagged Correlating Prefetching* (TCP) only stores the most significant bits of each address *(tag)*, making the most of the fact that a tag can appear in one or more cache sets, while an address only appear in one [9].

Instead of addresses, differences between consecutive addresses *(deltas)* can be alternatively stored. A delta sequence can stand for many miss address sequences. Therefore we can predict miss addresses that did not occur in the past, and a stride reference pattern is a particular case where all deltas have the same value. *Distance Prefetching* was firstly proposed for prefetching TLB entries. A delta is used to index in the correlation table the stream of deltas that previously followed [13].

DBCP *(Dead-block Correlating Prefetching)* correlates and predicts miss streams per block frame. An entry in the *History Table* encodes a list of prior memory addresses mapped to the block frame along with the sequence of PCs of the last instructions that have referenced that frame in the first cache level (L1). An entry in the *Dead-block Table* stores the sequence of instruction PCs that resulted in a block eviction some time in the past, the addresses of the evicted block, and the address of the block that replaced it [14]. Whenever a sequence of PCs referencing block A matches a sequence stored in the Dead-block Table that also ends with the eviction of A, the block B that replaced A in the past (recorded in the table) is prefetched. The History Table can be moderate in size, but we need a multi-megabyte Dead-block table hierarchy on- and off-chip, with high associativities.

Often times in correlating prefetchers the recorded history stales, mega-sized tables are needed, or the number of table accesses multiplies. A novel table structure (GHB) focuses on the first two problems [8][16][17][18]. GHB prefetching organizes the history table as a circular buffer (GHB, Global History Buffer). Miss addresses are inserted in the GHB as they appear in the global L2 miss stream. GHB entries are linked into address lists. Head pointers for each address list are stored in the Index Table (IT), indexed by a key. The PC of a missing load, any address that misses in L2, or some combination, may act as a key. This structure can be adapted to different prefetching methods, named on a X/Y basis, where X is the prefetch key and Y the method for detecting address patterns. The best performer in the family is PC/DC: the PC of the loads missing in L2 is used as the key, and consecutive addresses in a linked list are substracted to calculate *deltas*. Prefetching is issued when a repeating pattern of *deltas* is detected. [17]. Calculating *deltas* and tracking patterns implies several accesses (cycles) to the GHB, but the mechanism acts only upon L2 misses. 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 [18].

A GHB indexed by PC was evaluated and reported to outperform all the former prefetchers using the MicroLib platform in [8]. Our contribution here differs in scope and experimental procedures, specially in the processor and memory hierarchy. In addition, we use *LT on-miss insertion*, trained with the memory reference stream, instead of the stride prefetcher, trained with L2 misses, used in [8] and also in the original papers of GHB ([16][17][18]). Moreover the GHB prefetcher in [8] issues up to four prefetches per L2 miss, but sequential, stride and the other prefetchers seem to operate with a degree of one. We explore different prefetching degrees and distances across the four prefetchers, and analyze total and useful prefetches per reference, for establishing a precise parametrization of ports in the three table-based methods.

3. P-DFCM: prefetch based in DFCM

DFCM *(Differential Finite Context Method)* [7] is a value predictor based on *deltas* that can be adapted for prefetching with some advantage. It uses a table indexed by PC, where each entry holds the last value produced by the instruction, and the differences *(deltas)* between recent values. Deltas are hashed for indexing a second table, to find out the following probable *delta*. In this way, stride sequences (constant *delta*) only occupy one entry in the second table. A prediction can be done as soon as the PC is

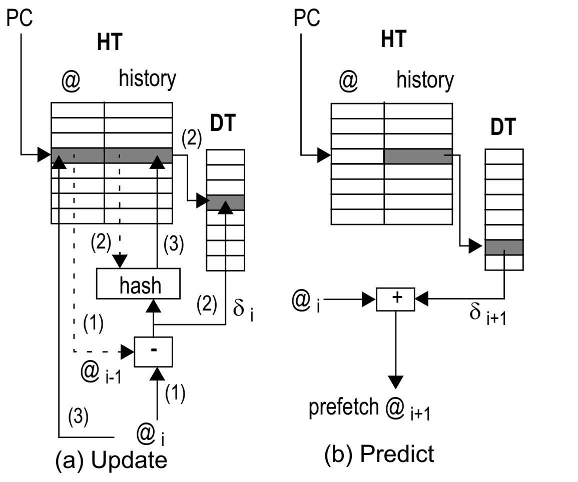

Fig. 1. Prefetch based on DFCM. HT: History Table; DT: Delta Table.

available, and the table is updated once the final value is known.

Differences between the P-DFCM prefetcher and the DFCM value predictor are as follows. On the one hand, value prediction applies on every instruction in the program, but address prediction for prefetching does not. It only applies to loads missing in L2, considerably lowering table sizes. On the other hand, the most remarkable difference with DFCM lies in the way of updating and predicting values or addresses. We could predict the address a load is going to produce as soon as the load PC is known. However, it makes little sense to prefetch a datum that is going to be demanded by a load just a few cycles later. In P-DFCM a prefetch is issued as soon as the data address is known, on behalf of a later instance of the same load. Let us consider Figure 1a. The History Table (HT) is indexed by PC. Each entry holds the last address produced by a load instruction, and the differences (*deltas*) between recent addressees issued by this load. Deltas are hashed for indexing the Delta Table (DT), to find out the following probable delta. Once the data address $@_i$ is known, it is substracted from the address issued in the previous instance of the load ($@_{i-1}$), which is stored in HT, for calculating δ_i (Figure 1a, (1)). This delta is stored in the DT indexed by the load history of deltas, while the new delta sequence is hashed (Figure 1a (2)), and the proper HT entry is next updated (Figure 1a (3)). Now, this new delta sequence indexes the next (predicted) delta in the sequence, δ_{i+1}, used to produce the prefetching address (Figure 1 b). All in all, each L2 miss requires one read and one write in HT, plus one read and one write in DT.

We apply the same hashing function used in DFCM, FS R-5, that yields the best results for finite context predictors [22]. In this function the length of the history *(order)* is a function of the logarithm of the number of DT entries ($\text{order} = \lceil n/5 \rceil$). Confidence counters are used in HT.

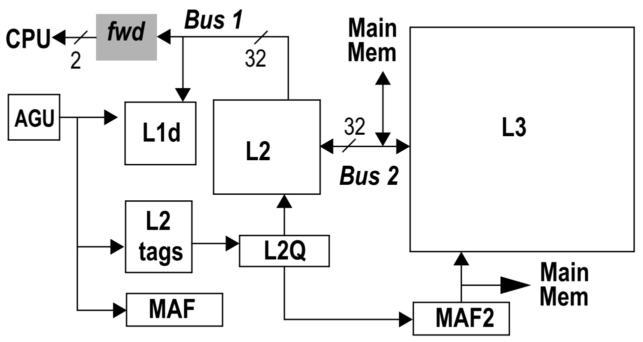

Fig. 2. Baseline architecture: main components of the memory hierarchy. AGU:Address Generation Unit; L1d: first-level data cache; L2 (tags/-): second-level cache (tags /data): MAF /2: Miss Address File in first /second level; L3: third-level cache; *fwd*: forwarding crossbar.

4. Experimental framework

4.1. Baseline architecture

The simulation environment is based on Simple Scalar 3.0 using Alpha binaries [2]. Simple Scalar was modified to model in detail a superscalar processor with a three-level on-chip cache memory (Figure 2). Table I shows baseline architecture parameters. The first-level data cache (L1d) supports up to four loads, one store and up to two loads, or two stores, and includes a store buffer, replicated for supporting four lookups by cycle. Four Coalescing Write Buffers allow to write in spare cycles. Store-load dependences go through a perfect predictor. L2 follows the Itanium 2 model. It is organized in 16 banks, and L2Q holds all data references to the banks. Refill of the L1d critical block proceeds in parallel with refill in L2. Relevant details on the memory pipeline are given in Figure 3. When a load references L1d, its dependent instructions are speculatively issued. L2 tags and L1d are accessed in parallel in the M1 stage.

Table II shows the characteristics of the benchmark programs. We selected those applications that achieve a speedup greater than 2% with an ideal L2 among Olden [20], SPEC CPU 2000 and IAbench [19]. Olden and IAbench applications were run up to completion. For SPEC CPU applications, we run the simple Simpoints, warming caches and branch predictor during 200 million instructions [23]. Software prefetching instructions were ignored during the simulations.

4.2. Implementation details of the prefetchers

The four prefetchers we compare aim to reduce L2 misses, and consequently data are brought into L2 in all cases. However, each prefetch controller uses information from different levels, as we explain below. We selected optimal table sizes for each prefetching method setting a

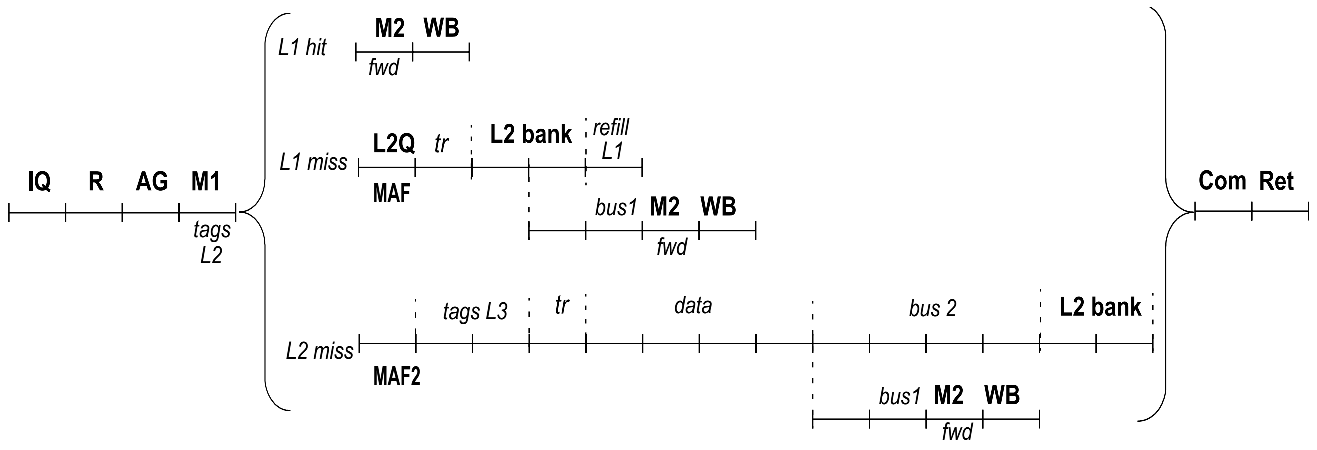

Fig. 3. Baseline architecture: memory pipeline. Stages: IQ (*Instruction Queue*), R (*Read registers*); AG (*Address generation*); M1 (*L1 data access and L2 tag lookup*); M2 (*data forwarding*); WB (*Write data in register*); Com (*Commit: architecturally complete*); Ret (*Retire: leave* ROB). Other stages indicate the component accessed. *tr* stands for *transport cycle*.

Table I. Baseline architecture: parameters

Fetch & Decode	8 instructions/cycle
Issue	8 int + 4 fp
Retire	16 instructions/cycle
ROB	256 entries
Execution Units	8 int ALU, 2 int MUL, 4 fpALU, 4 fpMUL
IQ	64 int + 32 fp *(1 cycle for reading operands)*
Store Buffer (STB)	128 entries
Branch Pred.	hybrid bi-modal, gshare (16 bits)
Cache L1 d	16 KB, block 32 B, 2-way, lat. 2 cycles write-through non-allocate, 4 Coalescing Write Buffers (CWB) Mem. Address File (MAF): 16 entries
Cache L1 i	ideal
Cache L2	256 KB, block 128 B, 8-way 16 banks 16B -interleaved bank access lat.: 2 cycles; ld/use lat.: 8 cycles, write-back allocate L2Q 32 entries, WB 6 entries, MAF2: 8 entries
Cache L3	4 MB, block 128 B, 16-way tag access: 2 cycles, pipelined data access: 4 cycles; ld/use lat.: 13 cycles write-back allocate; WB 2 entries
Memory	Latency 200 cycles; bandwith 1/20 cycles

Table II. L1, L2 and L3 miss rates and IPC for the different benchmarks.

Progr.	L1 mr	L2 mr	L3 mr	IPC	Parameters and group	
vpr	7,2%	2,5%	0,3%	1.29	SPEC CPU 2000 int (CINT)	
gcc	2,4%	0,5%	0,1%	5.19		
mcf	34,1%	19,6%	13,2%	0.24		
parser	7,6%	0,8%	0,0%	2.27		
gap	1,4%	0,1%	0,1%	1.74		
vortex	2,5%	0,3%	0,1%	4.72		
bzip2	3,1%	1,2%	0,0%	2.44		
twolf	12,6%	4,3%	0,0%	1.96		
wupwise	3,3%	0,8%	0,7%	2.88	SPEC CPU 2000 fp (CFP)	
swim	23,8%	5,0%	5,0%	0.81		
mgrid	7,4%	1,8%	0,9%	1.94		
applu	13,8%	3,0%	2,9%	1.33		
galgel	15,7%	3,3%	0,2%	3.31		
art	73,7%	41,5%	0,0%	2.22		
equake	19,3%	3,4%	3,2%	0.50		
facerec	4,5%	2,2%	0,2%	2.07		
ammp	12,1%	4,6%	0,1%	2.74		
fma3d	3,0%	0,5%	0,4%	2.45		
apsi	1,2%	0,1%	0,1%	4.57		
em3d	27,7%	3,8%	0,0%	2.92	em3d 2000 10 100 100	Olden
mst	14,1%	10,9%	9,4%	0.76	mst 1024 1	
perimeter	4,3%	0,5%	0,6%	1.60	perimeter 10 1	
treeadd	4,7%	0,6%	0,6%	1.63	treeadd 20 1	
tsp	1,9%	0,2%	0,1%	2.60	tsp 100000 0	
csrlite	12,4%	3,1%	2,8%	0.72	csrlite 30 20 gre__115.rua impcol_c.rua mcca.rua west0156.rua	IA bench
sparse	37,1%	12,3%	0,4%	0.64	(500x500 sparse matrix with 5000 non-zero elements)	

prefetch degree of one and varying table configuration over a wide range (Table III). A Prefetch Address Buffer holds up to eight addresses issued for prefetching. In all prefetchers, when the prefetch degree is greater than one, the second and following prefetches are issued at a one-per-cycle rate.

Sequential tagged prefetching operates at the second level cache, and therefore a bit was added to each entry in L2tags. We have observed experimentally that placing it in the first level exacerbates its aggressiveness.

Stride prefetching is implemented as *LT on-miss insertion* [10], and use information from L1 and L2: a) Loads are inserted in the LT only upon L2 miss, but note that b) every load that hits in LT is trained with the address issued to memory (i.e to L1). Whenever a load hits in the LT, if the confidence counter indicates that a stride pattern has been found, a prefetch is issued. LT is read in the AG stage for every load. Prefetches are issued in the M1 stage. LT entries are always updated (or assigned) in the Commit

Table III. Table configurations

Prefetching method	Entries per table	Size order[a]
Stride with on-miss insertion	32	512 Bytes
PC/DC	256 (IT) x 256 (GHB)	4 KB
P-DFCM	256 (HT) x 512 (DT)	5 KB

a. Including tags and rounded to the ceiling power of two.

stage only for loads that hit in LT (or miss in L2). The table has four read ports and one write port, for supporting up to four lookups and one update in a cycle.

Methods based in GHB can hardly be trained with the stream of references to memory, as we do in LT on-miss insertion, because the GHB should be much greater for holding the greater history. According to the original proposal, insertion and prediction in PC/DC is made upon a miss in L2. To keep the demand address stream unaltered, prefetched lines are tagged. The loads that hit on lines with a set prefetch tag update the tables as they were L2 misses. IT and GHB are assumed to have one port, enough to support the stream of L2 misses. The PC/DC predictor is updated in M1 at a maximum rate of one per cycle (see Figure 3). This requires reading IT, updating GHB, and next updating IT. Along the next cycles the GHB is walked looking for a pattern. Every table access takes a cycle, and the activity of the GHB state machine when serving an L2 miss occurrence is overridden if a subsequent L2 miss shows up [17][18]. Update and predict activities in P-DFCM are also carried out in M1 at a maximum rate of one per cycle. Details were given in Section 3.

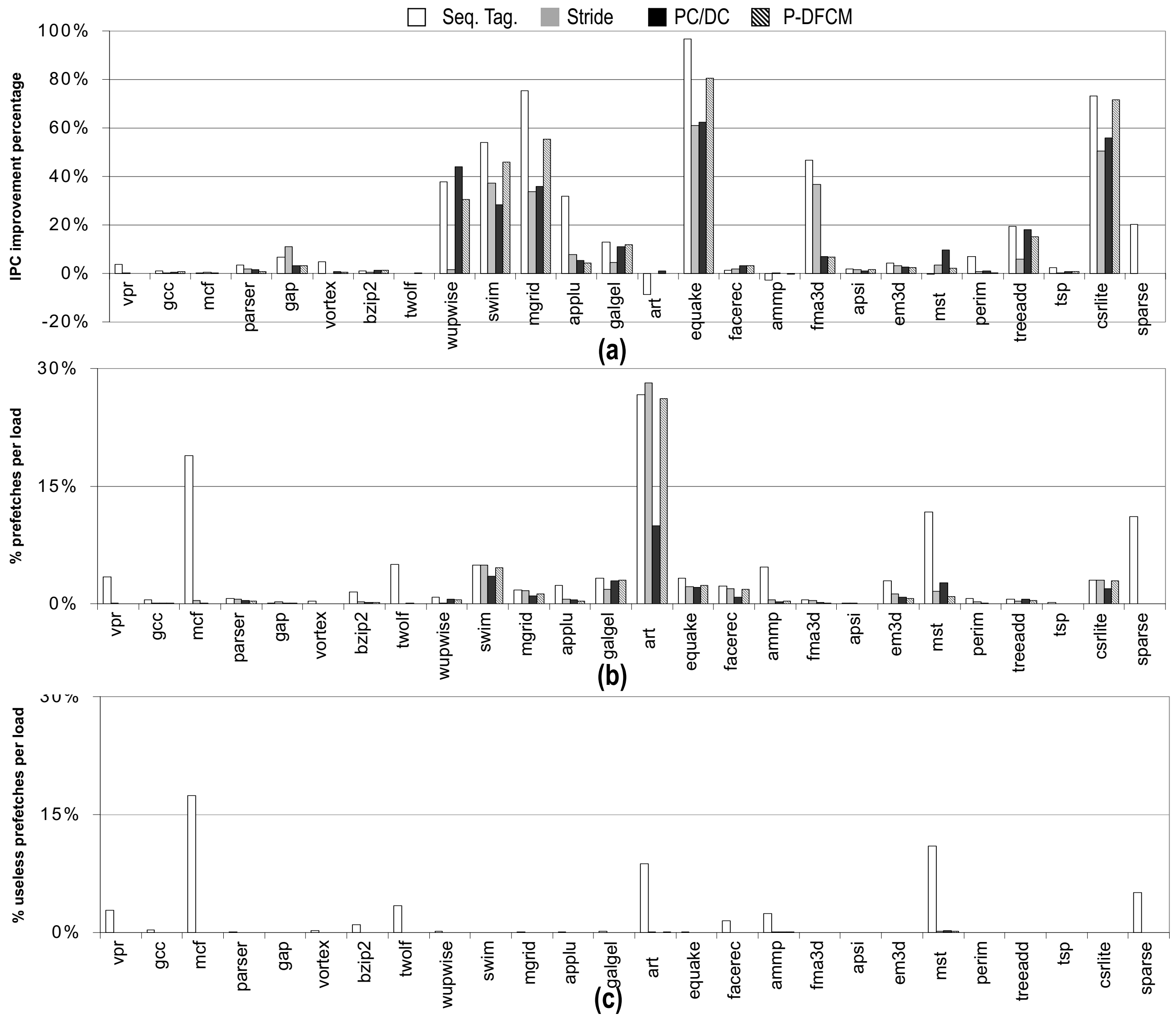

Fig. 4. Performance comparison of the four prefetchers, as IPC improvement percentage over the baseline system with no prefetch (a). Total (b) and useless (c) prefetches per committed load.

5. Results

A preliminary comparison of the four prefetchers is given in Figure 4. The upper graph (a) plots the IPC speedups achieved regarding the baseline architecture without prefetching. There are thirteen applications where at least one prefetcher gets a 5% IPC speedup (*gap, wupwise, swim, mgrid, applu, galgel, equake, fma3d, mst, perim, treadd, tsrlite,* and *sparse*). Sequential Tagged is the best in ten out of these thirteen applications, but also the only one having significative IPC losses *(art, ammp)*. The graphs below show the total (b) and useless (c) prefetches issued per non-speculative load. It can be observed that Sequential Tagged is the most active prefetcher, but also the one that issues more useless prefetches, yielding irregular results with great losses some times. Among the other three prefetchers (table-based) in this Figure 4, the best performer is Stride in *gap*, *applu*, and *fma3d*, PC/DC in *wupwise*, *mst* and *treeadd*, and P-DFCM in *swim*, *mgrid*, *galgel*, *equake* and *csrlite*. PC/DC performance is similar to P-DFCM, in spite of the fact that walking the linked lists in the GHB multiplies by four, on average, the number of accesses to the correlation tables performed by P-DFCM. Note that Stride is the only prefetcher in this experiment getting a speedup greater than 5% in a CINT application (*gap*). It is also interesting to observe in Figure 4 (b and c) that Stride, PC/DC and P-DFCM are certainly selective prefetchers: most prefetches are useful.

Figure 5 shows the impact of varying distance and degree for each one of the four prefetchers. Increasing distance or degree is beneficial for all, but note that losses in Sequential Tagged are very high in *amp* and *mst* (Figure 5 a), because useless prefetches increase by a factor of 2.8x on average. Stride Prefetching speedups are quite comparable when increasing either distance or degree, but increasing distance seems a better option because it can be implemented without additional cost, and far less prefetches are issued. In the case of PC/DC there is little advantage in increasing the distance, and some losses can appear. Increasing the prefetching degree is always positive, and seems the best option because in this method implies little extra effort. Similar conclusions apply to P-DFCM. The cost of increasing the degree is a bit higher in this method, but similar to increasing the distance, and results appear slightly more favorable when increasing the degree.

Finally, we gather all prefetchers again in Figure 6 but selecting the most reasonable option for each of them: distance four for Stride, and degree four for the rest. Let us now focus on the fourteen applications where at least one prefetcher gets a 5% IPC speedup (the same thirteen that before plus *parser*). Sequential Tagged turns out to be the best in ten of them. Stride clearly outperforms the two correlating prefetchers in four applications (*parser*, *gap*, *mgrid* and *fma3d*). If we compare the two correlating prefetchers, considering differences greater than 5% in IPC speedup, PC/DC is better than P-DFCM in *wupwise* and *mst*, but falls behind P-DFCM in *swim*, *mgrid*, *equake* and *csrlite*. Increasing degree or distance does not bias the outcome in favor of one of them. The number of references to the correlation tables keeps higher in PC/DC (2.6x on average with respect to P-DFCM).

It is worth noting that speedups on CINT applications keep low, and that only Sequential Tagged and Stride get some success with them. In many cases L3 serves most data or IPC is kind of high (*gcc*, *vortex,* see Table II), but it should be noted that few prefetches are issued in those applications. With a low base IPC and high miss ratios on the three levels, *mcf* remains a challenge for the four prefetchers: not even the two correlating prefetchers manage to deal with the erratic behavior of its object pointers, although only a 10% of memory references follow an unknown pattern [21].

6. Final remarks

We have simulated in detail a three-level on chip cache hierarchy, serving an 8-way processor capable of issuing up to four memory references per cycle. We evaluate two classic prefetchers (Sequential Tagged and Stride) and two correlating prefetchers: PC/DC, a recent method with a superior score and low-sized tables, and P-DFCM, a more conventional approach inside the class. It is similar to PC/DC in that it focuses on local delta sequences, but it takes the most of the simpler approach of the DFCM value predictor, while keeping table sizes low.

Our experimental results show that incrementing the prefetch degree and distance up to a value of four is beneficial in the arena of a high-performance cache hierarchy. Sequential Tagged performs notably well. It is worth devoting some research again to filter its useless prefetches and to cut its losses. Stride prefetching falls short in recent comparisons [17][8]. However, here we show that when used with *on-miss insertion* in LT it offers results comparable to smart correlating prefetchers, but using negligible resources. In addition, degree or distance can be increased in Stride with little effort. P-DFCM demonstrates that the performance of PC/DC can be achieved or even improved avoiding costly linked list traversals along the correlation tables while keeping sizes low.

Beside from we have just pointed out regarding Sequential Tagged, our future work includes exploiting additional opportunities of P-DFCM and measuring the effects of different prefetchers at different hierarchy levels, including hybrids that can cope with the still challenging applications Further delving into simulation results taking into account the characteristics of some application programs may help to understand why prefetchers do or do not work in each case.

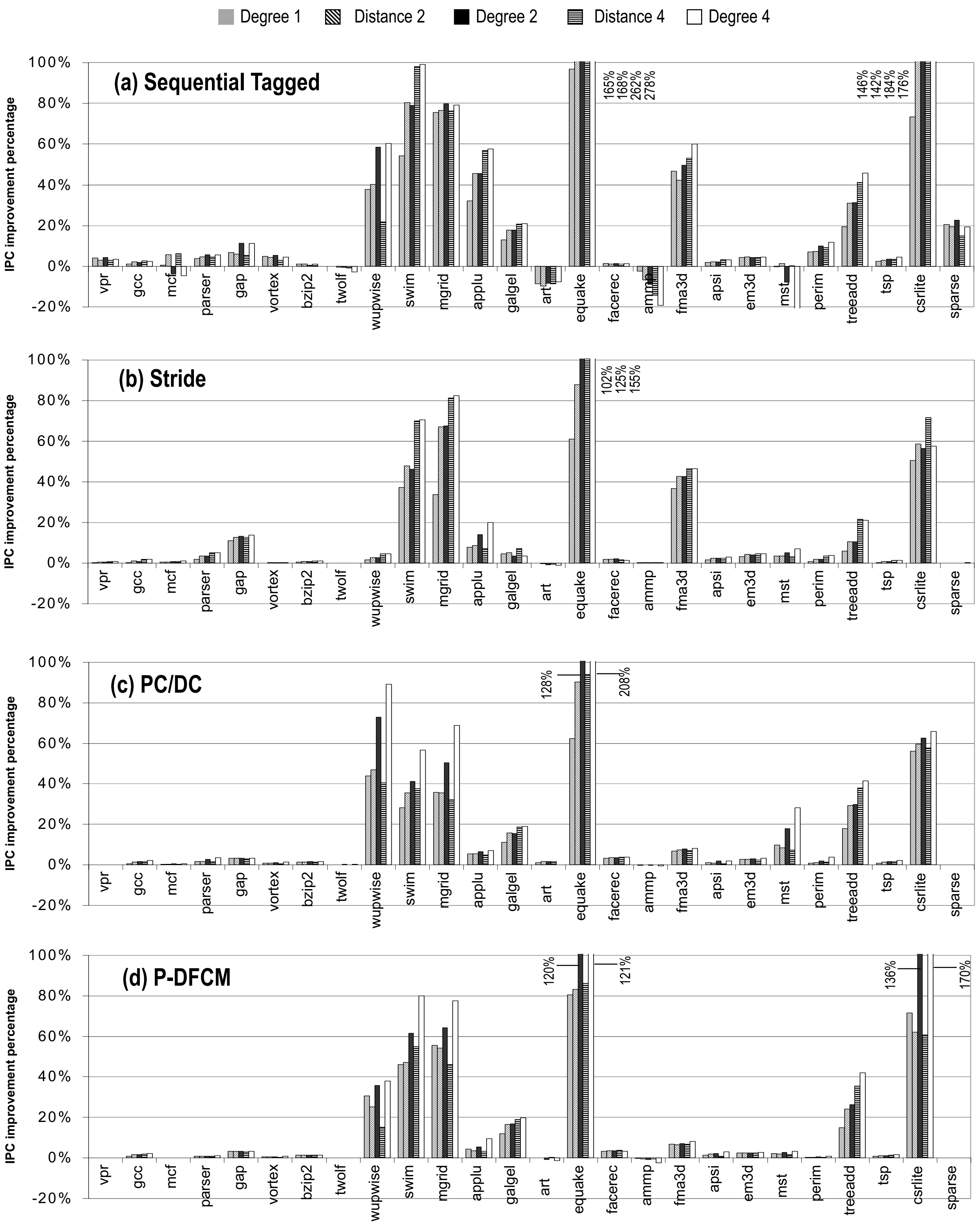

Fig. 5. IPC speedups over the baseline system with no prefetch when varying the prefetch degree and distance for Sequential Tagged (a), Stride (b), PC/DC (c) and P-DFCM (d). Each group of bars, from left to right, stands for degree 1, distance 2, degree 2, distance 4 and degree 4.

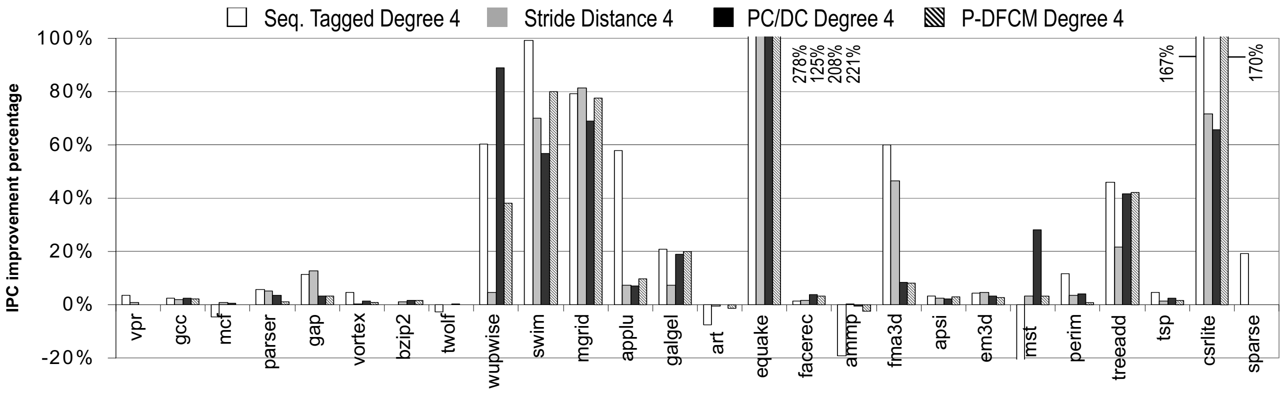

Fig. 6. IPC speedups comparing the selected point of each prefetcher regarding prefetch degree or distance. Stride operates with distance 4, and Sequential Tagged, PC/DC and P-DFCM with degree 4.

7. References

[1] J.L. Baer and T.F. Chen. "An Effective On-chip Preloading Scheme to Reduce Data Access Penalty". In *Int. Conf. on Supercomputing* (ICS) pp.176-186, 1991.

[2] D. Burger and T. Austin, The SimpleScalar Toolset, v. 3.0. `http://www.simplescalar.org`.

[3] J. Collins, S. Sair, B. Calder and D. M. Tullsen. "Pointer Cache Assisted Prefetching". In *Procs. 35th Int. Symp. on Microarchitecture* (MICRO-35) pp. 62-73, Nov. 2002

[4] R. Cooksey, S. Jordan, D. Grundwald. "A Stateless, Content-Directed Data Prefetching Mechanism". In Proc. of *10th Int. conf. on Architectural support for programming languages and operating systems* (ASPLOS X) pp. 279 - 290 San José, California, Oct. 2002.

[5] A. S. Dhodapkar and J. E. Smith. "Managing Multi-Configuration Hardware via Dynamic Working Set Analysis". In Proc. of the *29th Ann. Intl. Symp. on Computer Architecture*, (ISCA) pp. 233-245. May 2002.

[6] P. G. Emma, A. Harstein, T. R. Puzac and V. Srinivasan. "Exploring the limits of prefetching". *IBM Journal of Res. and Dev.* 49 (1) pp. 127-144, Jan. 2005.

[7] B. Goeman, H. Vandierendonck and K. De Bosschere. "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency". In *Procs. of the 7th Int. Symp. on High-Performance Computer Architecture* (HPCA) pp. 207-218. Monterrey, Mexico 2001.

[8] D. Gracia, G. Mouchard and O. Temam. "MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms". Proc. of the 37th Int. Symp. on Microarchitecture (MICRO-37), pp. : 43-54. December 2004.

[9] Z. Hu, M. Martonosi, S. Kaxiras, "TCP Tag Correlating Prefetchers", In Proceedings of the *9th Int. Symposium on High Performance Computer Architecture* (HPCA), 2003.

[10] P. Ibáñez, V. Viñals, J.L. Briz, and M.J. Garzarán. "Characterization and Improvement of Load/Store Cache-based Prefetching". In *Proc. of Int. Conf. on Supercomputing* (ICS) Melbourne, Australia. pp.369-376 July 1998.

[11] D. Joseph and D. Grunwald. "Prefetching Using Markov Predictors". *IEEE Trans. on Computer Systems*, 48(2), pp. 121–133, 1999."

[12] N. Jouppi. "Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers". In *Procs. of the 17th International Symposium on Computer Architecture* (ISCA), Seattle, WA, 1990.

[13] G. B. Kandiraju and A. Sivasubramaniam. "Going the Distance for TLB Prefetching: An Application-driven Study". In *Procs. of the 29th Int. Symposium on Computer Architecture* (ISCA), May 2002.

[14] A. Lai, C. Fide and B. Falsafi. Dead-Block Correlating Prefetchers". In Procs. of the *28th Intl. Symp. on Computer Architecture* (ISCA) pp. 144-154, 2001

[15] Mark J. Charney and Anthony P. Reeves. "Generalized correlation-based hardware prefetching". TR EECEG-95-1, School of Electrical Engineering, Cornell University, February 1995.

[16] K. J. Nesbit and J. E. Smith. "Data Cache Prefetching Using a Global History Buffer". In Procs. of the *10th Annual Int. Symp. on High Performance Computer Architecture* (HPCA) pp: 96-105, Madrid, Spain 2004.

[17] K. J. Nesbit and J. E. Smith. "Data Cache Prefetching Using a Global History Buffer". *IEEE Micro* 25 (3), pp. 90-97. May/June 2005.

[18] 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.

[19] L. Ramos, P. Ibáñez, V. Viñals and J.M. Llabería. "Modelling Load Address Behaviour Through Recurrences". In Proc. of *Int. Symp. on Performance Analysis of Systems and Software* (ISPASS), Austin, Texas. pp. 101-108 April, 2000.

[20] A. Rogers, M. Carlisle, J. Reppy and L. Hendren. "Supporting Dynamic Data Structures on Distributed Memory Machines". *ACM Trans. on Programming Languages and Systems*, March 1995.

[21] S. Sair, T. Sherwood and B. Calder. "Quntifying load stream behavior". In Proc *8th. Annual International Symposium on High Performance Computer Architecture* (HPCA) 2002.

[22] 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.

[23] T. Sherwood et al., "Automatically Characterizing Large Scale Program Behaviour," ASPLOS X, Oct. 2002.

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

[25] S.P. Vanderwiel and D.J. Lilja.- "Data Prefetch Mechanisms". *ACM Computing Surveys* 32 (2) June 2000.

[26] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt and C. C. Weems. "Guided Region Prefetching: A Cooperative Hardware/Software Approach". In *Proc. 30th Int. Symp. on Computer Architecture* (ISCA) 2003.