Low-Cost Adaptive Data Prefetching

Luis M. Ramos, José Luis Briz, Pablo E. Ibáñez, and Víctor Viñals

Dpto. Informática e Ing. de Sistemas, Instituto I3A, U. Zaragoza
{luisma,briz,imarin,victor}@unizar.es

Abstract. We explore different prefetch distance-degree combinations and very simple, low-cost adaptive policies on a superscalar core with a high bandwidth, high capacity on-chip memory hierarchy. We show that sequential prefetching aggressiveness can be properly tuned at a very low cost to outperform state-of-the-art hardware data prefetchers and complex filtering mechanisms, avoiding performance losses in hostile applications and keeping the pressure of the prefetching on the cache low, turning it out into a real implementation option for current processors.

1 Introduction

Hardware data prefetching has been largely accepted as an effective way of hiding memory latency. Recent research has lead to very successful proposals like the ones based on a Global History Buffer (GHB) [21], or new stream prefetchers specially focused on servers [12][26]. However, only the simplest mechanisms have been implemented in commercial microprocessors: sequential prefetching in UltraSPARC-IIIcu and SPARC64 VI [17][29], sequential stream buffers in Power4 and Power5 [16][28], and sequential and stride prefetching in the Intel core microarchitecture [8].

Although sequential prefetching can yield the highest speedups, it triggers performance losses in hostile benchmarks and leads to a high pressure on the memory hierarchy. Filtering mechanisms have been recently applied to scheduled region prefetching [3] and sequential (always) prefetching [30]. Both of them call for non negligible hardware.

Losses can also be reduced by tuning prefetching aggressiveness. Let us consider the stream of references a program is going to demand $(a_i, a_{i+1}, a_{i+2}, \ldots)$, where a_i has been just demanded by the program. A prefetcher can dispatch $a_{i+1} \ldots a_{i+n}$, where n is the *prefetch degree*. Alternatively, it could also prefetch only a_{i+n}, and then we say that n is the *prefetch distance*. Increasing the prefetch degree or distance can either boost or ruin performance, causing pollution and exacerbating the pressure on the memory hierarchy. Sequential prefetching with adaptive degree was first proposed in [6], on multiprocessors, focusing on prefetching usefulness. Adaptive stream prefetching is explored in [27], balancing usefulness, timeliness and pollution. Both approaches need far less hardware than the aforementioned filtering proposals.

Our aim is to profit from the simplest hardware prefetcher —sequential tagged— with the smallest hardware investment. We evaluate new and known

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 327–336, 2008.

degree-distance policies along with adaptive mechanisms that can be boiled down to just a few counters, and we compare them with an optimized stride prefetcher [13], a GHB-based prefetcher [21], a correlating prefetcher [22], and a spatial memory stream prefetcher [26]. Prefetched blocks are brought into L2 in all cases. We model in great detail an on-chip memory hierarchy with high bandwidth and capacity that services an aggressive superscalar processor running SPEC CPU 2000 benchmarks. Our best simple adaptive sequential prefetcher reduces execution time 7.6% with respect to a system without prefetching in integer benchmarks (36.6% in floating point benchmarks), whereas the spatial memory stream prefetcher [26] —that performs the best among the others— saves 7.8% (int) and 32.9 % (fp), but issuing 64% more accesses to the second cache level and using a much complex hardware than the adaptive sequential prefetching.

In Sec. 2 we provide essential background and motivate the contribution. Sec. 3 introduces our proposals and all the techniques evaluated. Sec. 4 details the experimental environment. Sec. 5 analyzes the experimental results, including filtering through the Prefetch Address Buffer, and gives some hardware cost estimates. We finally draw some conclusions in the last Section.

2 Background and Motivation

Sequential prefetching prefetches the block or blocks that follow the current demanded block [25]. *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. These methods tend to issue many prefetches that are not used by the CPU (*useless prefetches*), especially when degree or distance are applied.

Conventional *stride prefetching* uses a Load Table (LT) indexed by the program counter (PC) to identify and predict accesses to memory addresses separated by a constant distance [1]. The size of the table can be much reduced without severe performance losses by applying on-miss insertion in the LT [13]. Stride prefetching can also be implemented by using stream buffers [15].

Correlating prefetchers predict future addresses from tables that record the past memory program behavior. They record the stream of addresses associated either to the load PC or to an address that misses [11][14][18]. Alternatively, differences between consecutive addresses (*deltas*) can be stored. A delta sequence can stand for many miss address sequences, hence it can predict miss addresses that did not occur in the past. A novel table structure (GHB, Global History Buffer) focuses on reducing table sizes, and can be adapted to different prefetching methods with very good results [10][21]. The best performer in the family (PC/DC) uses as index the PC of the loads missing in L2, and consecutive addresses in a linked list are subtracted to calculate deltas. Prefetching is issued when a repeating pattern of deltas is detected. Although the mechanism acts only upon L2 misses, calculating deltas and tracking patterns implies quite a few accesses to the GHB. PDFCM [22] is a more classic correlating prefetcher

based on the Differential Finite Context Method (DFCM) [9]. 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. PDFCM performs similarly to GHB with far lower table overhead [22].

SMS (Spatial Memory Streaming) is a hardware prefetcher that identifies code-related spatial access patterns and prefetch into the cache the stream of blocks inside a memory region that are likely to be used [26]. It avoids loading into the cache useless blocks, which is an issue for sequential prefetching and stream buffers, at the cost of using three tables plus some extra logic.

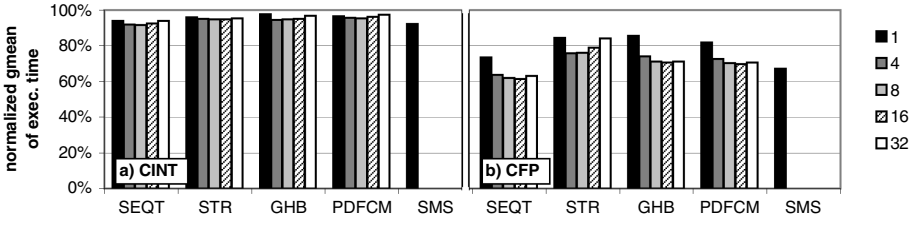

Fig. 1. SEQT, STR, GHB, PDFCM and SMS for a sample of CINT (a) and CFP (b) benchmarks. Degree ranges from 1 to 32 in SEQT, GHB and PDFCM. In STR we vary distance instead. Degree does not apply in the case of SMS.

In Fig. 1 we compare a sequential tagged prefetcher (SEQT) with an optimized stride prefetcher (STR) [13] and three state-of-the-art prefetchers (PC/DC [21], PDFCM [22] and SMS [26]). We vary the prefetching degree from 1 to 32 in all of them but in STR and SMS. We vary distance in STR because it performs better [21]. Concerning SMS, the degree depends on the length of the predicted stream inside a region. Considering the region size selected in [26] and our L2 block size (128 B), the maximum number of prefetched blocks has been set to sixteen. The graph shows the geometric mean of the normalized execution time with respect to a system without prefetching considering the selected programs from SPEC CPU 2000 (see Sec. 4).

Details on the baseline architecture and implementation of the prefetchers are given later in Sec. 4. SEQT with degree 8 yields the best results, with a negligible difference with respect to SMS in the case of integer benchmarks, and reducing the execution time an 8% more than SMS for floating point benchmarks. Fig. 2 reveals that the good performance of the SEQT and SMS implies that they pressure the cache hierarchy a lot more than the selective STR, GHB and PDFCM, concerning the rate of generated prefetch addresses, accesses to L2 and L2 misses. The plot also shows the rate of useful prefetches.

Considering the best two prefetchers (SEQT(8) and SMS), the breakdown per application in Fig. 3 reveals that SEQT degree 8 causes insignificant performance losses in twolf, that become important in ammp. These results show that it is worth looking for mechanisms to cut losses in wrong-case applications and to reduce the pressure on the memory hierarchy caused by sequential prefetchers,

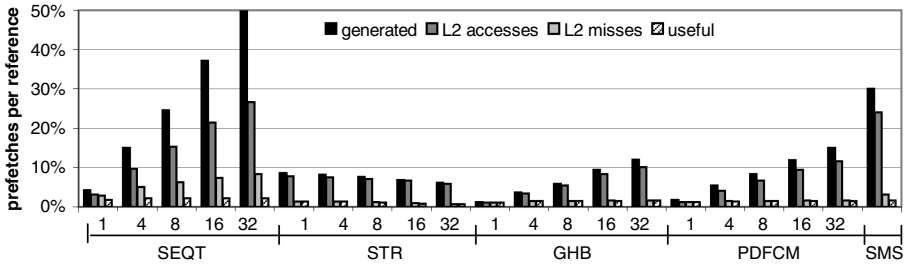

Fig. 2. Pressure on the cache hierarchy made by SEQT, STR, GHB, PDFCM and SMS. Metrics are relative to the number of committed memory references.

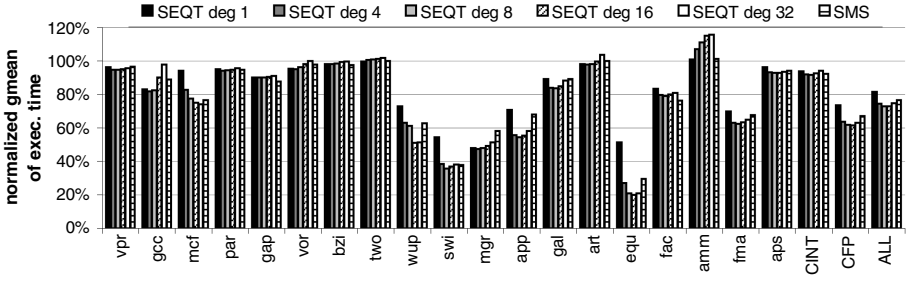

Fig. 3. Breakdown per application for SEQT with degree 1 to 32 and SMS

as long as the hardware needs are kept low. The next section introduces several proposals to handle this problem.

3 Degree-Distance Policies

Table 1 summarizes the options we evaluate in this paper (note that *1st use* refers to a prefetched block). All of them but Deg-dist(x) and Ad5(x) constitute new proposals. Deg(x) and Dist(x) policies are straightforward (see Section 2). Deg-dist(x) and Deg($1 - x$) are just as explained in the table. The Deg-dist(x) policy is described in [7], and is quite similar to the stream buffers proposed in [15]. Prefetch performance depends on the *usefulness* and *timeliness* of the prefetched blocks, and on the *pollution* caused by them in the cache. Looking for simplicity, most of our adaptive policies focus only on usefulness. All of them use three counters, *degree*, *up* and *down*. The first one holds the current degree. *Up* increases whenever a prefetched block is demanded for the first time, and when it reaches a threshold *degree* is increased. The *down* counter has the opposite effect and relies on a different threshold. It counts useless prefetches (replacements of tagged blocks). Both the two thresholds determine how many events we let happen before deciding to increase or decrease the degree.

Our Ad1(x) policy prefetches with degree 1 on a demand miss, and with the degree indicated by the counter on the first use of a prefetched block, up to

Table 1. Degree - Distance Policies

	Policy	Description
Fixed	Deg(x)	Degree fixed to x
	Dist(x)	Distance fixed to x
	Deg-dist(x)	On miss, prefetch with degree x. On 1st use, prefetch with distance x.
	Deg($1-x$)	On miss, prefetch with degree 1. On 1st use, prefetch with degree x
Adaptive	Ad1(x)	On miss, prefetch with degree 1. On 1st use, prefetch with variable degree $[0 \ldots x]$
	Ad2(x)	On miss, prefetch next and previous block and set their *direction bit*. On 1st use, prefetch with variable degree $[0 \ldots x]$ according to the *direction bit* of the block
	Ad3(x)	Behaves like Ad1(x), but timeliness and pollution are taken into account
	Ad4(x, y)	Ad2(x) applied to y memory regions
	Ad5(x)	Follows [6]

a maximum degree x. Ad2(x) prefetches both the next and the previous block on every demand miss, and tags them with the direction they were prefetched (forward or backward). Then, on the first use of a prefetched block *degree* blocks are prefetched following the direction indicated by the block accessed. Therefore, Ad2(x) requires an extra bit per block, besides the one needed by any sequential tagged prefetcher. The Ad4(x, y) policy applies Ad2(x) to y memory regions. It needs 3 counters per memory region to adapt the degree independently.

The Ad3(x) policy takes also into account timeliness and pollution. Here, *up* also increases with *late prefetches* (demand misses on blocks that are being prefetched but that have not reached L2 yet) trying to make up for them. In Ad3(x) the *down* counter also accounts for pollution due to prefetch (demanded blocks replaced by prefetched blocks and causing a demand miss later on). To track this last event we use a Bloom filter like in [27].

Ad5(x) uses a *prefetch* counter, a *useful* counter and a *degree* counter. It closely follows [6] except in that all original counters are four bits long and hence degree range is $[0 \ldots 15]$ whereas we let degree increase up to 32.

4 Experimental Environment

The simulation environment is based on SimpleScalar 3.0 using Alpha binaries [2]. SimpleScalar was modified to model in detail a superscalar processor with a three-level on-chip cache memory (Fig. 4). Table 2 shows the 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. Store-load dependences go through a perfect predictor. L2 follows the Itanium 2 model. L2Q holds all data references to the sixteen banks. Refill of the L1d critical block proceeds in parallel with refill in L2. When a load references L1d, its dependent instructions are speculatively issued. L2 tags and L1d are accessed in parallel in the first memory stage.

We run the simple Simpoints, warming caches and branch predictor during 200 million instructions [24]. We selected those SPEC CPU 2000 applications that achieve a speedup greater than 2% with an ideal L2. Table 3 shows the characteristics of the benchmark programs.

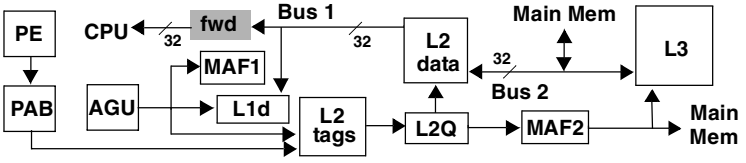

Fig. 4. Main components of the memory hierarchy. AGU: *Address Generation Unit*; L1d: *1st-level data cache*; L2 (tags/data): *2nd-level cache*: MAF1 /MAF2: Miss Address File; L3: 3rd-level cache; fwd: forwarding crossbar; PE: Prefetch Engine; PAB: Prefetch Address Buffer.

Table 2. Baseline architecture: parameters

Fetch & Dec.	8 instructions per cycle
Issue / Retire / ROB	8 int + 4 fp / 16 instructions per cycle / 256 entries
IQ / Exec. Units	64 int + 32 fp / 8 int ALU, 2 int MUL, 4 fp ALU, 4 fp MUL
Store Buffer	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, Miss Addr. File (MAF1): 16 entries
Cache L1 i	Ideal
Cache L2	256 KB, block 128 B, 8 way, 16 banks 16B-interleaved. Serial access: tag 1 cycle + data 2 c. (ld/use lat: 8 c.) Write-back alloc., L2Q 32 entries, WB 6 entries, MAF2: 8 entries
Cache L3	4 MB, block 128 B, 16 way; write-back alloc.; WB 2 entries, Serial access: tag 2 cycle pipelined + data 4 c. (ld/use lat: 13 c.)
Memory	Latency 200 cycles; bandwidth 1/20 cycles

Concerning the implementation of the five tested prefetchers, we do not prefetch addresses beyond the physical page limit (8 KB), and data are always brought into L2. We selected optimal table sizes for each prefetching method setting the prefetch degree to one and varying table configuration over a wide range. The number of entries per table are 32 in STR, 256(IT)×256(GHB) in PC/DC, 256(HT)×256(DT) in PDFCM and 32 (Accumulation table) / 64 (FT) / 1024×16 (PHT) in SMS. A Prefetch Address Buffer (PAB) holds addresses issued for prefetching, as many as indicated by the maximum degree. When the prefetch degree is greater than one, the second and following prefetches are issued at a one-per-cycle rate in all prefetchers. Prefetches are not issued if less than 5 free entries are left in MAF2. This precaution dramatically cut losses in all aggressive prefetchers. In STR the LT is read in the address generation stage for every reference. Prefetches are issued in the first memory stage. LT entries are always updated (or assigned) in the Commit stage only for references that hit in LT (or miss in L2). The PC/DC predictor is updated in the first memory stage at a maximum rate of one per cycle. Update and predict activities in PDFCM are also carried out in the memory stage at a maximum rate of one per cycle. SMS matches the implementation given in [26].

Table 3. L1, L2 and L3 miss rates and IPC for the selected benchmarks

		vpr	gcc	mcf	parser	gap	vortex	bzip2	twolf
CINT	**L1 mr**	7.2%	2.4%	34.1%	7.6%	1.4%	2.5%	3.1%	12.6%
	L2 mr	2.5%	0.5%	19.6%	0.8%	0.1%	0.3%	1.2%	4.3%
	L3 mr	0.3%	0.1%	13.2%	0.0%	0.1%	0.1%	0.0%	0.0%
	IPC	1.29	5.19	0.24	2.27	1.74	4.72	2.44	1.96

		wupwise	swim	mgrig	applu	galgel	art	equake	facerec	ammp	fma3d	apsi
CFP	**L1 mr**	3.3%	23.8%	7.4%	13.8%	15.7%	73.7%	19.3%	4.5%	12.1%	3.0%	1.2%
	L2 mr	0.8%	5.0%	1.8%	3.0%	3.3%	41.5%	3.4%	2.2%	4.6%	0.5%	0.1%
	L3 mr	0.7%	5.0%	0.9%	2.9%	0.2%	0.0%	3.2%	0.2%	0.1%	0.4%	0.1%
	IPC	2.88	0.81	1.94	1.33	3.31	2.22	0.50	2.07	2.74	2.45	4.57

Thresholds for the *up* and *down* counters in the adaptive policies are preset to 100 and 50 events respectively. We experimented with different values, but results were hardly affected as long as the threshold ratio was kept.

5 Results

We have already seen that SMS is the state-of-the-art prefetcher that gives the best results in the preliminary experiments (Sect. 2), and therefore we will use it as the reference for evaluating our proposals.

We do not show the breakdown of results per application for the sake of space. The only benchmarks showing performance losses due to prefetch are ammp — where Deg-dist(x) and Deg(x), on the one hand, and Deg($1-x$) and Dist(x), on the other hand, are paired in terms of losses— and art (Deg(32)). No loss shows up in any application for any adaptive method.

Figure 5 gathers the geometric mean of the normalized execution times related to the base system of all the policies we consider in this work (Table 1). In general, the best options for CINT are those with degree $x = \{4, 8\}$, while $x = \{8, 16\}$ yields better results in CFP. Ad5(x) and Dist(x) show the poorest performance in both CINT and CFP groups, hence we will not consider them in which follows. The rest of the techniques perform similar to Deg with $x = \{4, 8\}$ in CINT and to Deg with $x = \{8, 16\}$ in CFP. Among the techniques without losses, the best choice in CINT is Ad4(8, 32) (91.9%) but differences are below 1% with Ad2(4), Ad2(8), Ad4(4, 32), Ad4(16, 32) and SMS. However SMS is widely outperformed in CFP by all the adaptive ones (except Ad5(x)). The best one is now Ad3(x) (62.9%, while SMS amounts to 67.1%), but Ad1(8), Ad2(8), and Ad4(8, 32) are all of them less than 1% above Ad3(x). All in all, Ad4(8, 32) and Ad2(8) seem to be the best tradeoffs on average (INT, CFP).

Figure 6 shows the pressure each technique makes on the memory hierarchy in terms of generated prefetches, lookups in L2, L2 misses and useful prefetches. The difference between the last two bars indicates the percentage of useless prefetches. Ad1(x) and Ad3(x) keep useless prefetches below SMS, whereas Ad2(x) and Ad4(x, y) are a little bit less efficient than SMS concerning this subject.

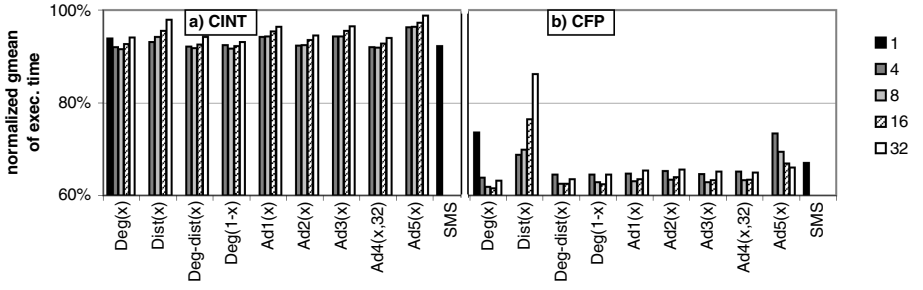

Fig. 5. Comparison of the best adaptive policies in the selected applications

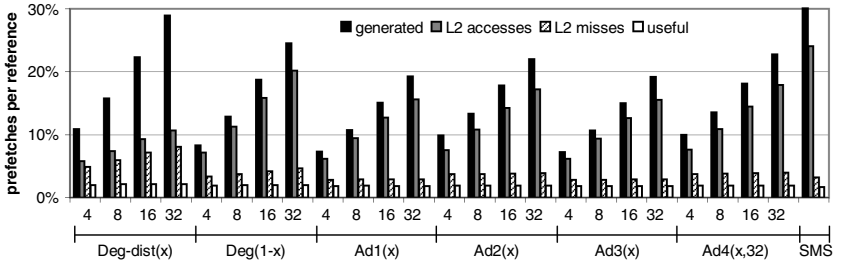

Fig. 6. Pressure on the memory hierarchy. Percentages refer to the number of committed memory references.

Using the Prefetch Address Buffer as a filter. We propose to reduce the pressure on the second cache level (L2) by using the Prefetch Address Buffer (PAB) as a filter. The PAB keeps prefetch requests after they are generated by the PE and until they lookup the L2 tags. Whenever new prefetch addresses are generated, the prefetch unit accesses the PAB to check if they are already on the buffer, to avoid issuing them again. Our proposal maintains the prefetch addresses on the PAB after the lookup on L2 tags, replacing them only when a new PAB entry is needed in FIFO order. New prefetch addresses check all the PAB entries, so they are filtered if they match any of the last N prefetch addresses generated by the PE, where N is the PAB size. Following this strategy, L2 lookups are reduced by 2% for Deg-dist(x), and between 25% and 40% for the rest of policies shown in Fig. 6, leaving performance unaffected. The higher reduction is achieved in SMS (49%), but L2 lookups are still above the figures for the rest of prefetchers. Thus, our best choices in Fig. 5 (Ad4$(8, 32)$ and Ad2(8)) give 7.5%, well below the 12.3% for SMS.

Hardware cost. All the adaptive methods we have proposed require three counters, and an extra bit per cache line –because of the underlying tagged sequential mechanism– accounting for 2Kbit (256 B). Ad2(x) requires another bit to record the direction of the prefetching. Ad3(x) needs a 4 Kbit array (512 B) to implement the Bloom filter. Counters in Ad4(x, y) are per region, hence

the best point (8, 32) needs a 64 B table. The approximated table sizes for STR, PC/DC, PDFCM and SMS are respectively 512 B, 4 KB, 5 KB and 33 KB.

6 Conclusions

We have proposed here different simple ways of tuning the aggressiveness of a sequential prefetcher so that it can perform similar to or better than the best state-of-the-art prefetcher, SMS [26]. Among the options we propose, Ad2(8) and Ad4(8, 32) perform the best. Both are adaptive mechanisms that vary the sequential prefetching degree (up and down) according to prefetching usefulness. Ad2(8) prefetches forward and backward with variable degree. Ad4(8, 32) splits memory into 32 regions and keeps separated counters for each region. Both of them equal SMS in CINT and outperform it in CFP, with 60% less lookups in L2. Ad2(8) needs just two bits per cache line and Ad4(8, 32), additionally, a 64 B table. This is far less than the 33 KB needed by SMS tables.

We also propose a simple filtering technique using the Prefetch Address Buffer that helps to reduce significatively the pressure of prefetching on the second level cache. It reduces the L2 lookups generated by Ad2(8) and Ad4(8, 32) in 30%.

Considering the prefetchers implemented by manufacturers so far, ours are a feasible choice showing no losses on typical integer and floating point workloads at a really low hardware cost.

Acknowledgments. Supported by Diputación General Aragón *Grupo Consolidado Investigación*, Spanish MEC TIN2007-66423, and the European HiPEAC-2 (FP7/ICT 217068).

References

1. Baer, J.L., Chen, T.F.: An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In: ICS, pp. 176–186 (1991)
2. Burger, D., Austin, T.: The SimpleScalar Toolset, v. 3.0, `www.simplescalar.org`
3. Burger, D., et al.: Filtering Superfluous Prefetches Using Density Vectors. In: ICCD, p. 124 (2001)
4. Charney, M.J., Reeves, A.P.: Generalized correlation-based hardware prefetching. TR EECEG-95-1, School of Electrical Engineering, Cornell Univ. (February 1995)
5. Cooksey, R., et al.: A Stateless, Content-Directed Data Prefetching Mechanism. In: ASPLOS-X, S. Jose, CA, pp. 279–290 (October 2002)
6. Dahlgren, F., et al.: Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors. In: ICPP, pp. 156–163. CRC Press, Boca Raton (1993)
7. Dahlgren, F., Stenström, P.: Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. IEEE Trans. Parallel and Distributed Systems 7(4), 385–398 (1996)
8. Doweck, J.: Inside Intel Core Microarchitecture and Smart Memory Access. White Paper, Intel Corporation (2006)
9. Goeman, B., et al.: Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In: HPCA-7, Monterrey, Mexico, pp. 207–218 (2001)

10. Gracia, D., et al.: MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. MICRO-37, 43–54 (2004)
11. Hu, Z., et al.: TCP Tag Correlating Prefetchers, HPCA-9 (2003)
12. Hur, I., Lin, C.: Memory Prefetching Using Adaptive Stream Detection. MICRO-39, 397–408 (2006)
13. Ibáñez, P., et al.: Characterization and Improvement of Load/Store Cache-based Prefetching. In: ICS, Melbourne, Australia, pp. 369–376 (July 1998)
14. Joseph, D., Grunwald, D.: Prefetching Using Markov Predictors. IEEE Trans. on Computer Systems 48(2), 121–133 (1999)
15. Jouppi, N.: Improving direct-mapped cache performance by addition of a small fully associative cache and prefetch buffers. In: ISCA-17, Seattle, WA (1990)
16. Kalla, R., et al.: IBM Power5 chip: A dual-core multithreaded processor. IEEE Micro. 24(2), 40–47 (2004)
17. Krewell, K.: Fujitsu Makes SPARC See Double. Microproc. Report (November 2003)
18. Lai, A., et al.: Dead-Block Correlating Prefetchers. In: ISCA-28, pp. 144–154 (2001)
19. Lin, W.F., et al.: Filtering superfluous prefetches using density vectors. In: ICCD 2001, Washington D.C., USA, pp. 124–132. IEEE Comp. Society, Los Alamitos (2001)
20. Nesbit, K.J., Smith, J.E.: Data Cache Prefetching Using a Global History Buffer. In: HPCA-10, Madrid, Spain, pp. 96–105 (2004)
21. Nesbit, K.J., Smith, J.E.: Data Cache Prefetching Using a Global History Buffer. IEEE Micro. 25(3), 90–97 (2005)
22. Ramos, L.M., et al.: Data prefetching in a cache hierarchy with high bandwidth and capacity. SIGARCH Comput. Archit. News 35(4), 37–44 (2007), http://doi.acm.org/10.1145/1327312.1327319
23. Sair, S., et al.: A Decoupled Predictor-Directed Stream Prefetching Architecture. IEEE Trans. on Computers 52(3), 260–276 (2003)
24. Sherwood, T., et al.: Automatically Characterizing Large Scale Program Behaviour. In: ASPLOS-X (October 2002)
25. Smith, A.J.: Sequential Program Prefetching in Memory Hierarchies. IEEE Trans. on Computers 11(12), 7–21 (1978)
26. Somogyi, S., et al.: Spatial Memory Streaming. In: ISCA-33, pp. 252–263 (2006)
27. Srinath, S., et al.: Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In: HPCA-13, pp. 63–74.
28. Tendler, J.M., et al.: Power4 system microarchitecture. IBM Journal of Research and Development 46(1), 5–26 (2002)
29. UltraSPARC III Cu - User's Manual.Sun Microsystems (January 2004), http://www.sun.com/processors/manuals/USIIIv2.pdf
30. Zhuang, X., Lee, H.-H.S.: Reducing Cache Pollution via Dynamic Data Prefetch Filtering. IEEE Trans. on computers 56(1), 18–31 (2007)