Targeted Data Prefetching

Weng-Fai Wong

Department of Computer Science, and Singapore-MIT Alliance
National University of Singapore
3 Science Drive 2, Singapore 117543
wongwf@comp.nus.edu.sg

Abstract. Given the increasing gap between processors and memory, prefetching data into cache becomes an important strategy for preventing the processor from being starved of data. The success of any data prefetching scheme depends on three factors: *timeliness*, *accuracy* and *overhead*. In most hardware prefetching mechanism, the focus has been on accuracy - ensuring that the predicted address do turn out to be demanded in a later part of the code. In this paper, we introduce a simple hardware prefetching mechanism that targets *delinquent loads*, i.e. loads that account for a large proportion of the load misses in an application. Our results show that our prefetch strategy can reduce up to 45% of stall cycles of benchmarks running on a simulated out-of-order superscalar processor with an overhead of 0.0005 prefetch per CPU cycle.

1 Introduction

The growing gap between processors and memory has led to performance not growing with improvements in processor technology in a commensurate way. The introduction of caching has alleviated somewhat the problem though not eliminating it entirely. In many state-of-the-art processors, a miss that requires going to the memory can cost hundreds of cycles. This problem can only be made worse by the introduction of chip multiprocessors which only serve to increase the demand for the timely delivery of data.

Data prefetching is an important strategy for scaling this so-called memory wall. There are two main classes of data prefetching strategies. In the pure hardware approach, additional hardware is added to the processor that monitors the execution of the program, launching prefetch requests when the opportunity arises. The key advantage of this approach is its transparency. Neither the source code nor any special knowledge about the application is needed. In the hybrid software-hardware approach, the hardware exposes its prefetching mechanism to the compiler or programmer. The compiler is then suppose to take advantage of some knowledge about the application to maximize the utility of these hardware mechanisms. An example of such mechanisms could be a simple prefetch instruction which serves as a hint to the processor to promote a piece of datum higher up the memory hierarchy. The disadvantage of the hybrid approach is that the source code of the application is often needed for the compilation process. In this paper, we are concerned with hardware prefetching.

For any prefetch mechanism to work, three criteria have to be met. The prefetch has to be *timely*. Data have to be brought in just in time to met the demand of it. Bringing

in data too early risks having it being evicted from the cache when it is needed, and of course, performing the prefetch after the original load has been issued renders the prefetch useless. Any prefetch requires the ability to predict the address of the data that will be needed ahead of its time. This prediction has to be *accurate* in order that the prefetch not be wasted. Finally, the *overhead* associated with performing an overhead has to be small. This overhead may be measured in terms of the amount of hardware needed to support a hardware prefetcher or the pressure exerted by the prefetches on the processor's resource.

Most prefetch schemes suggested in the literature thus far has been focused on the issue of accuracy. In this paper, we propose a strategy for data prefetching that addresses the question of overhead without sacrificing accuracy and timeliness. The key insight exploited by this mechanism is that most of the cache misses experienced by an application are due to a small number of *delinquent loads* [9, 27]. By focusing the prefetch effort on these delinquent loads, one can reduce the number of prefetches issued without sacrificing much in terms of accuracy or timeliness.

The rest of the paper is organized as follows. Section 2 will survey the related works in the field. Section 3 will introduce our proposed mechanism. This is followed by our experimental setup, the results and a discussion of the results. The paper ends with a conclusion and some ideas for future works.

2 Related Work

There has been a lot of research into data prefetching [30]. Early work on software prefetching focused on prefetching for data intensive loops [4, 19]. With the proper analysis, regularly nested loops can benefit tremendously from data prefetching. [26]. Beyond loops, prefetching for procedure parameters [21] and recursive data structures [23] have also been proposed. Karlsson, Dahlgren, and Stenstrom [17] proposed the use of prefetch arrays while VanderWiel and Lilja proposed a *data prefetch controller* (DPC) [29]. More recently, there has been some interest in using a 'helper' hardware thread to prefetch ahead of the main thread [18]. Rabbah et. al. [28] proposed the use of spare resources in an EPIC processor to accomodate the prefetch thread instead of using a hardware thread.

Pure hardware prefetching schemes includes Jouppi's "stream buffers" [16], Fu and Patel's prefetching for superscalar and vector processors [11, 12], and Chen and Baer's lookahead mechanism [7] and known as the *Reference Prediction Table* (RPT) [8]. Mehrota [25] proposed a hardware data prefetching scheme that attempts to recognize and use recurrent relations that exist in address computation of link list traversals. Extending the idea of correlation prefetchers [6], Joseph and Grunwald [15] implemented a simple Markov model to dynamically prefetch address references. More recently, Lai, Fide, and Falsafi [20] proposed a hardware mechanism to predict the last use of cache blocks.

On load address prediction, Eickemeyer and Vassiliadis first proposed the use of a stride predictor to predict the address of a load [10]. However, it is not a prefetch as the prediction is verified before any memory system activity is triggered. This idea was extended by others by means of operand-based predictors [2] and last-address predic-

tors [22]. The use of a stride predictor in the launch of actual prefetches was proposed by Gonzalez and Gonzalez [14]. A more elaborate scheme to enhance prediction accuracy was proposed by Bekerman et. al. [3]. The latter also expressed concern on the negative effects spurious prefetches have on the cache and made special provisions to avoid these from polluting the cache.

The delinquent load identification hardware scheme is similar to the local hit-miss predictor proposed by Yoaz et. al. [31]. However, they used it to predict the dynamic latencies of loads. The predicted load latencies are used to help the instruction scheduler more accurately schedule load instructions. For their purpose, a higher degree of accuracy is required as compared to ours which is used to spot potential delinquent loads. Furthermore, we applied the filter only to misses as we only launch prefetches on level one data cache misses. As far as we know, the use of the simple delinquent load identification scheme for the purpose of prefetching is novel.

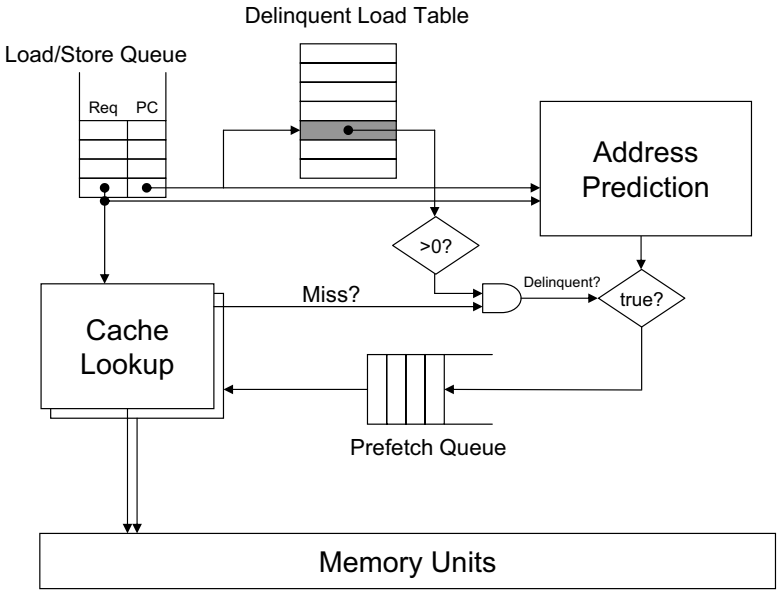

Fig. 1. Proposed prefetching architecture.

3 Proposed Prefetching Mechanism

Data prefetching involves two important decisions, namely:

– *When* should a prefetch be triggered? It is not desirable to launch prefetches too early or too late. Neither is it desirable to launch more prefetches than what is

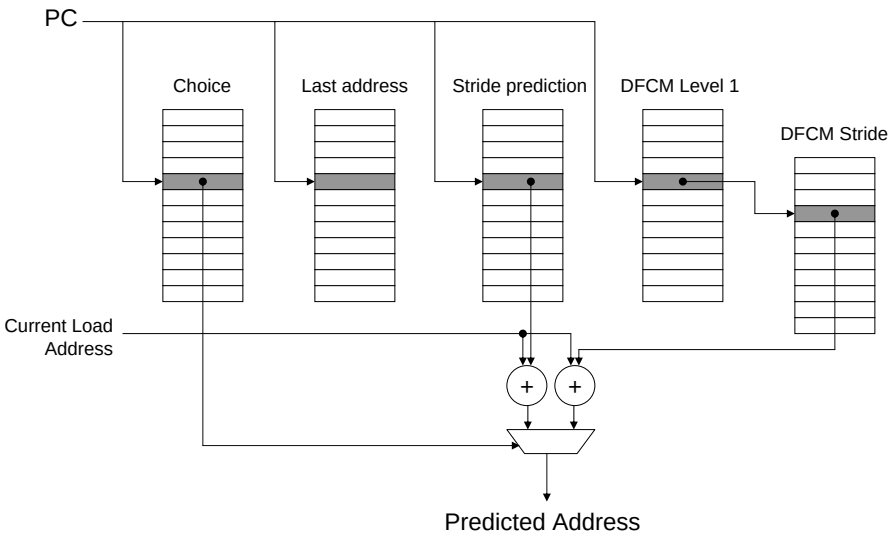

Fig. 2. Hybrid load address predictor used.

required because there are fixed overheads associated with the processing of each prefetch.

– *What* should be prefetch? It is necessary to predict the address(es) of the data items required by the processor in the near future. As with many successful applications of predictions, here, past history serves as a good indicator for future use. The accuracy of the prediction is key to the success of such schemes. Some prefetch schemes performs pre-computation instead of prediction [1]. These schemes are generally much more accurate then mere prediction. However, the hardware investments for this can be substantial as portions of the pre-computation may turn out to be redundant.

We propose a prefetch scheme that addresses these issues as follows. The first key insight our scheme takes advantage of is that only a small number of load instructions, known as delinquent loads, are responsible for a large number of the cache misses. It is for these loads that we will launch prefetches. In order to identify delinquent loads, we hash the program counter for a load instruction into a table of saturating counters. Let PC stand for the program counter. Let h(PC) be the 3-bit saturating counter obtained by hashing PC into the delinquent load table, a table of saturating counters. If the counter is non-zero, then the load is identified as a delinquent load. The counter is updated as follows: if the current load misses the cache, the counter is incremented (up to 7, beyond which it cannot be incremented further). If the current load hits the cache, the counter is decremented (down to zero, below which it cannot be decremented further). The main idea is for the counter to track if, in the recent history of this particular load, there were more misses than hits. If so, it is classified as delinquent. The length of the

counter affects how much of the past history is considered. The choice of 3-bit counters is along the lines of the experiences gained in the branch prediction community.

The proposed architecture is shown in Fig. 1. In an out-of-order processor, a *load-store queue* serves as the reorder buffer for loads and stores. In our proposed architecture, we require the program counter to be noted alongside the load requests. When a load reaches the head of the load-store queue for processing, in parallel with cache lookup, a delinquent table lookup is performed. If it is known that the load misses the (L1) cache, a prediction is performed. Not shown in Fig. 1 is how the delinquent load table is also updated according to the description above. The predicted load address forms a prefetch request that enters the prefetch queue. This queue competes with the main load-store queue for service by the memory ports. Prefetch requests that hit the cache, however, are discarded.

When a delinquent load misses, prefetching may then occur. In order to contain the amount of resources needed to realize the prefetch mechanism, we chose a prediction based scheme. In particular, we chose to use a hybrid *stride* and *differential finite context method* predictor [13]. This predictor is shown in Fig. 2. First, a last value table records the previous address which the current load accessed. After the current prediction is completed, it is updated with the current load address. The stride predictor records most recently seen stride for the current load. Its predicted address is formed by adding the last seen stride to the current load address. The predictor is updated by recording the difference between the current address and the last seen address of this load which is recorded in the last address table. The differential finite context method is another stride predictor that makes use of a two-level table. In doing so, it is able to account for the context information under which a stride was last seen. We refer the reader to the original paper on DFCM [13] for a detailed discussion of the rationale and working of this state-of-the-art predictor. The last address table is also needed for the update of this predictor.

The idea of hybridizing two predictors comes from the branch prediction community [24]. Based on the past history of success, a table of saturating counters is checked to see which of the two predictors were more successful for a particular load and to use that predictor's prediction in the current prefetch. To update the predictor choice table, the last seen address is added to the strides predicted by both predictors. These are the previous predictions made by the predictors. These are then compared with the current load address. If the stride predictor is correct, the counter is decremented. If the DFCM predictor is correct, the counter is incremented. The next choice of predictor will depend on whether the counter is greater or less than zero. If it is exactly at zero, an arbitrary choice is made. In our experiments, the prediction of the DFCM is taken.

4 Experiment Setup

We realized the proposed architecture through modifying the SimpleScalar simulator for an out-of-order superscalar processor [32]. The machine parameters used in our experiments are listed in Table 1. We used a delinquent load table of 2,048 3-bit counters. All other tables, i.e. the last value table (which is 4 bytes per entry), the stride predictor (2 bytes per entry), DFCM level 1 table (10 bits per entry), DFCM level 2 stride table

Parameter	Value
Instruction fetch queue size	4 instructions
Branch predictor	Combined predictor
Decode width	4 instructions
Issue width	4 instructions
Commit width	4 instructions
Load-store queue length	8 requests
Prefetch queue length	8 requests
Memory ports	2 or 4
Integer ALU	4
Integer multiplier	1
FP ALU	4
FP multiplier	1
Number of registers	32
L1 inst cache	16K, 32-byte, direct, LRU
L1 data cache	16K, 32-byte, 4-way, LRU
L1 hit latency	1 cycle
L2 data cache	256K, 64-byte, 4-way, LRU
L2 hit latency	10 cycle
L2 miss latency	min. 100 cycle

Table 1. Simulated out-of-order machine used in the experiments.

(2 bytes per entry), and the choice predictor (3 bits per entry), are 1,024 entries each. The total table size is about 10.3 Kbytes. Note that this is significantly less than the 16 Kbyte L1 data cache especially when tag storage is also considered.

We evaluated the performance of our proposed prefetching scheme using benchmarks from the SPEC [33] and the Olden [5] benchmark suite. In detail, the benchmarks used were a Lisp interpreter (`130.li`), a JPEG encoder (`132.ijpeg`), gzip compression (`164.gzip`), quantum chromodynamics (`168.wupwise`), shallow water modeling (`171.swim`), a multigrid solver (`172.mgrid`), a neural network (`179.art`), combinatorial optimization (`181.mcf`), seismic wave propagation simulation (`183.equake`), computational chemistry (`188.ammp`), word processing (`197.parser`), an object oriented database (`255.vortex`), BZIP2 compression (`256.bzip2`), electromagnetic wave propagation in a 3D object (`em3d`), and the traveling salesman problem (`tsp`).

As a comparison, we implemented the standard RPT scheme that attempts prefetching on every load instruction. In addition, we also implemented Joseph and Grunwald's Markovian prefetcher [15]. The latter will launch four prefetches for each delinquent load. These four addresses are basically the last four addresses previously accessed by a particular load and missed the cache.

5 Performance Evaluation Results

Fig. 3 shows the result of our experiments for a machine with two memory ports. It shows the ratio of simulated machine cycles of the RPT, Markovian (denoted by 'Jos-Grun') and our proposed prefetch schemes against a baseline machine that do not per-

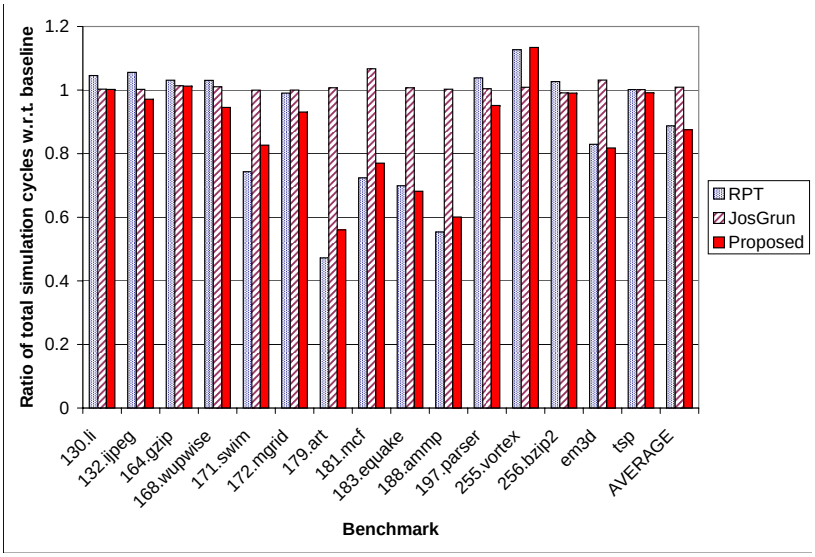

Fig. 3. Performance of various prefetching schemes on a simulated out-of-order superscalar processor with two memory ports.

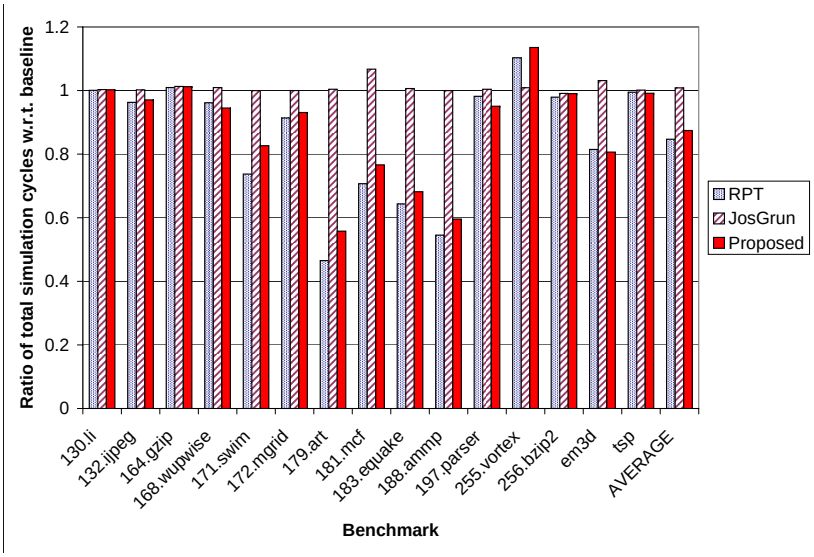

Fig. 4. Performance of various prefetching schemes on a simulated out-of-order superscalar processor with four memory ports.

form any prefetching. In the best case (`179.art`), it took only 56% of the total machine cycles of the baseline to complete the application with our prefetch scheme. In the worst case, however, it took 13% more cycles than the baseline to complete with our prefetching scheme. This happens when the prediction turned out to perform badly. Overall, our proposed scheme reduced the baseline cycle count by about 12.5%. This is marginally better than RPT. On the other hand, except in a few instances, the Markovian predictor performed poorly.

For a machine with four memory ports, the results are shown in Fig. 4. They are similar to that for that for two memory ports. The noticable difference is that here RPT performs better for our scheme.

Benchmark	**RPT**	**Fract. of Total Cyc.**	**JosGrun**	**Proposed Scheme**
130.li	177	41.88%	17 (9.60%)	5 (2.82%)
132.ijpeg	12,405	44.88%	346 (2.79%)	103 (0.83%)
164.gzip	528	29.52%	98 (18.56%)	45 (8.52%)
168.wupwise	5,255	32.96%	281 (5.35%)	47 (0.89%)
171.swim	12,751	18.91%	1,939 (15.21%)	526 (4.13%)
172.mgrid	420	51.13%	10 (2.38%)	2 (0.48%)
179.art	4,998	10.88%	2,322 (46.46%)	495 (9.90%)
181.mcf	427	17.37%	318 (74.47%)	88 (20.61%)
183.equake	34,739	34.60%	6,370 (18.34%)	1,638 (4.72%)
188.ammp	905	9.81%	865 (95.58%)	185 (20.44%)
197.parser	1,579	31.29%	254 (16.09%)	108 (6.84%)
255.vortex	6,936	21.34%	562 (8.10%)	365 (5.26%)
256.bzip2	1,548	40.15%	57 (3.68%)	26 (1.68%)
em3d	96	11.04%	59 (61.46%)	16 (16.67%)
tsp	8,723	3.87%	404 (4.63%)	110 (1.26%)
Average			(25.51%)	(7.00%)

Table 2. Number of prefetches launched (in millions). The ratio of the number of prefetches launched by a particular scheme over that for RPT is shown inside parenthesis.

RPT performs better because it covers all loads, which includes all delinquent loads, whereas our delinquent load predictor can still at times miss out on certain delinquent load. This improved performance for RPT, however, comes at a price. Table 2 shows the number of prefetches launched by RPT, the Markovian predictor and our scheme. The second column is the ratio of the number of prefetches launched and the total machine cycles for the application running on the baseline machine with four memory ports. This highlights the key point of the paper: with our delinquent load identification scheme, we can achieve a performance gain competitively comparable with RPT but with *only* 7% of the prefetches launched by RPT. This is significant in processor designs where there are more constraints on resources or where power-energy is an issue.

Completing the execution of an application early is often advantageous from a total energy consumption perspective. However, each prefetch consumes energy, thus our

proposed prefetching scheme will save a significant amount of energy most of the time. Furthermore, in many high performance processors where heating is a serious concern, the lower amount of additional processor activity due to prefetching is also conducive to maintaining the temperature profile of the processor.

Benchmark	Total num. of loads	Baseline D1 miss rate	Delinquency
130.li	211	0.93%	1.30%
132.ijpeg	14,594	0.55%	0.39%
164.gzip	595	2.70%	3.63%
168.wupwise	6,600	0.40%	0.41%
171.swim	13,340	4.26%	1.98%
172.mgrid	426	0.70%	0.26%
179.art	6,134	9.18%	4.03%
181.mcf	594	8.10%	8.01%
183.equake	36,772	3.81%	2.89%
188.ammp	1,015	12.33%	9.29%
197.parser	2,151	1.86%	2.42%
255.vortex	8,181	0.84%	2.82%
256.bzip2	1,999	0.79%	0.65%
em3d	132	7.01%	0.63%
tsp	10,785	0.44%	0.48%

Table 3. Delinquency in our benchmarks.

Table 3 shows the number of load instructions, level 1 data cache miss rate and the percentage loads identified as delinquent using our scheme. The delinquency rate is generally significantly lower than the miss rate. In some instances, the miss rate is lower. However, the miss rate is computed over all memory references including writes so the two values cannot be directly compared but merely serves as an indication of the selectivity of our scheme.

6 Conclusion

In this paper, we introduced an architecture for hardware prefetching that targets delinquent loads. When coupled with a hybrid load address predictor, our experiments showed that the prefetching scheme can reduce the total machine cycles by as much as 45% by introducing a low overhead. For the latter, the ratio of prefetches launched by our proposed scheme over the total machine cycles for the baseline out-of-order machine with four memory port ranges from 0.05% to 3.6%. This contrasts with the 3.8% to 51% overhead for RPT which achieves similar performance gains. We therefore argue that our proposed scheme fulfils the three criteria of a good prefetch scheme discussed in the Introduction, namely timeliness, accuracy and low overhead.

The key contribution of this work is in pointing out that even with a simple scheme, prefetching can be targeted very specifically to the load instructions that matter, and this yields significant practical benefits.

On a final note, the proposed architecture is general enough to work with any load address prediction scheme. It would be interesting to see if other prediction schemes, perhaps even ones that are uniquely designed for different applications so as to optimize area-performance, say, can benefit from it.

References

1. Annavaram, M., Patel, J.M., Davidson, E.S.: Data prefetching by dependence graph precomputation. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001) 52–61
2. Austin, T.M., Sohi, G.S.: Zero-cycle loads: Microarchitecture support for reducing load latency. In: Proceedings of the 28th International Symposium on Microarchitecture. (1995) 82 – 92
3. Bekerman, M., Jourdan, S., Ronen, R., Kirshenboim, G., Rappoport, L., Yoaz, A., Weiser, U.: Correlated load-address predictors. In: Proceedings of the 26th Annual International Symposium on Computer Architecture. (1999) 54–63
4. Callahan, D., Kennedy, K., Porterfield, A.: Software prefetching. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. (1991) 40–52
5. Carlisle, M.C.: Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines. PhD, Princeton University Department of Computer Science (1996)
6. Charney, M., Reeves, A.: Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University (1995)
7. Chen, T.F., Baer, J.L.: A performance study of software and hardware data prefetching schemes. In: Proceedings of International Symposium on Computer Architecture. (1994) 223 – 232
8. Chen, T.F., Baer, J.L.: Effective hardware-based data prefetching for high-performance processor computers. IEEE Transactions on Computers **44-5** (1995) 609 – 623
9. Collins, J.D., Wang, H., Tullsen, D.M., Hughes, C., Lee, Y.F., Lavery, D., Shen, J.P.: Speculative precomputation: Long-range prefetching of delinquent loads. In: Proceedings of the 28th International Symposium on Computer Architecture. (2001) 14–25
10. Eickemeyer, R.J., Vassiliadis, S.: A load-instruction unit for pipelined processors. IBM Journal of Research and Development **37** (1993) 547–564
11. Fu, J.W.C., Patel, J.H.: Data prefetching strategies for vector cache memories. In: Proceedings of the International Parallel Processing Symposium. (1991)
12. Fu, J.W.C., Patel, J.H.: Stride directed prefetching in scalar processors. In: Proceedings of the 25th International Symposium on Microarchitecture. (1992) 102 – 110
13. Goeman, B., Vandierendonck, H., Bosschere, K.D.: Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In: Proceedings of the 7th International Sysmposium on High-Performance Computer Architecture. (2001) 207 – 216
14. Gonzalez, J., Gonzalez, A.: Speculative execution via address prediction and data prefetching. In: Proceedings of the 11th International Conference on Supercomputing. (1997) 196–203
15. Joseph, D., Grunwald, D.: Prefetching using markov predictors. In: Proceedings of the 24th International Symposium on Computer Architecture. (1997) 252 – 263

16. Jouppi, N.P.: Improving direct-mapped cache performance by the addition of a smally, fully associative cache and prefetch buffers. In: Proceedings of the 17th International Symposium on Computer Architecture. (1990) 364 – 373
17. Karlsson, M., Dahlgren, F., Stenstrom, P.: A prefetching technique for irregular accesses to linked data structures. In: Proceedings of the 6th International Sysmposium on High-Performance Computer Architecture. (2000) 206 – 217
18. Kim, D., Liao, S.S., Wang, P.H., del Cuvillo, J., Tian, X., Zou, X., Wang, H., Yeung, D., Girkar, M., Shen, J.P.: Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In: Proceedings of the International Symposium on Code Generation and Optimization. (2004) 27–38
19. Klaiber, A.C., Levy, H.M.: An architecture for software-controlled data prefetching. In: Proceedings of the 18th International Symposium on Computer Architecture. (1991) 43 – 53
20. Lai, A.C., Fide, C., Falsafi, B.: Dead-block prediction and dead-block correlation prefetchers. In: Proceedings of the International Parallel Processing Symposium. (2001) 144 – 154
21. Lipasti, M., Schmidt, W., Kunkel, S., Roediger, R.: Spaid: Software prefetching in pointer and call-intensive environment. In: Proceedings of the 28th International Symposium on Microarchitecture. (1995) 231 – 236
22. Lipasti, M.H., Wilkerson, C.B., Shen, J.P.: Value locality and load value prediction. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. (1996) 138–147
23. Luk, C.K., Mowry, T.: Compiler-based prefetching for recursive data structures. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. (1996) 222 – 233
24. McFarling, S.: Combining branch predictors. Technical Report TN-36, DEC WRL (1993)
25. Mehrota, S., Luddy, H.: Examination of a memory classification scheme for pointer intensive and numeric programs. Technical Report CRSD Tech. Report 1351, CRSD, University of Illinois (1995)
26. Mowry, T.C., Lam, M.S., Gupta, A.: Design and evaluation of a compiler algorithm for prefetching. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems. (1992) 62 – 73
27. Panait, V.M., Sasturkar, A., Wong, W.F.: Static identification of delinquent loads. In: Proceedings of the International Symposium on Code Generation and Optimization. (2004) 303–314
28. Rabbah, R.M., Sandanagobalane, H., Ekpanyapong, M., Wong, W.F.: Compiler orchestrated prefetching via speculation and predication. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. (2004) 189–198
29. Vanderwiel, S., Lilja, D.: A compiler-assisted data prefetch controller. In: Proceedings of the International Conference on Computer Design. (1999) 372 – 377
30. Vanderwiel, S., Lilja, D.J.: Data prefetch mechanisms. ACM Computing Survey **32** (2000) 174 – 199
31. Yoaz, A., Erez, M., Ronen, R., Jourdan, S.: Speculation techniques for improving load related instruction scheduling. In: Proceedings of the 26th Annual International Symposium on Computer Architecture. (1999) 42–53
32. The SimpleScalar toolkit.: http://www.simplescalar.com. (2005)
33. The SPEC benchmarks.: http://www.spec.org. (2000)