A Framework for Data Prefetching using Off-line Training of Markovian Predictors*

Jinwoo Kim, Krishna V. Palem
Center for Research on Embedded Systems and Technology
Georgia Institute of Technology
Atlanta, GA 30332-0250
{jinwoo | palem}@ece.gatech.edu

Weng-Fai Wong
Department of Computer Science
National University of Singapore
Singapore 117543
wongwf@comp.nus.edu.sg

Abstract

An important technique for alleviating the memory bottleneck is data prefetching. Data prefetching solutions ranging from pure software approach by inserting prefetch instructions through program analysis to purely hardware mechanisms have been proposed. The degrees of success of those techniques are dependent on the nature of the applications. The need for innovative approach is rapidly growing with the introduction of applications such as object-oriented applications that show dynamically changing memory access behavior. In this paper, we propose a novel framework for the use of data prefetchers that are trained off-line using smart learning algorithms to produce prediction models which captures hidden memory access patterns. Once built, those prediction models are loaded into a data prefetching unit in the CPU at the appropriate point during the runtime to drive the prefetching. On average by using table size of about 8KB size, we were able to achieve prediction accuracy of about 68% through our own proposed learning method and performance was boosted about 37% on average on the benchmarks we tested. Furthermore, we believe our proposed framework is amenable to other predictors and can be done as a phase of the profiling-optimizing-compiler.

1. Introduction

It is a well-established fact that as processor speed increases, memory becomes a serious performance bottleneck. While the introduction of caches significantly alleviated the problem, caching alone will not bridge the increasing performance gap between multi-issue processors running at very high clock speeds and memory. Data prefetching has been proposed as an additional tool to bridge this gap. Existing hardware prefetching techniques require the prefetching hardware to perform some form of learning and prediction in real-time. This may necessitate a significant investment in hardware, or there may be an impact on the critical path of instruction processing. In the worst case, it can be both. In this paper, we propose a new paradigm that utilizes extensive profiling and powerful *off-line* learning algorithms. The main contributions of this paper are:

- A novel framework to perform off-line trace analysis that permits a wide range of learning algorithms;
- A prefetching microarchitecture that is low in hardware requirement and overhead.

Our technique showed significant improvement in prediction accuracy over existing ones.

In Section 2, we will describe some representative previous works on this subject. In Section 3, we will discuss the use of off-line learning algorithms. Our proposed architecture will be presented in Section 4 together with three learning algorithms that we tested. This is followed by experimental setup, results and a conclusion.

2. Previous Work

Research on memory hierarchy optimization can be classified into three broad categories: software approaches, hardware approaches and hybrid approaches. We will briefly mention some representative work and refer the interested reader to a detailed survey on the matter that was recently published [25].

In the field of software prefetching early work include that done by Callahan, Kennedy, and Porterfield [4], and Klaiber and Levy [16]. The former proposed the insertion of data prefetch instructions in data intensive loops while

*This work is supported in part by DARPA contract F33615-99-1499, Hewlett-Packard Laboratories and Yamacraw. Also supported in part by A*STAR Project No. 012-106-0046.

the latter studied efficient architectural support mechanisms for data prefetch instructions. Mowry, Lam and Gupta [22] showed that careful analysis and selective prefetching could provide significant performance improvements in programs with regular nested loops. Other software prefetching techniques includes Speculatively Prefetching Anticipated Interprocedural Dereference (SPAID) [19], the use of cache miss heuristics to drive prefetching [23], and the prefetching of recursive data structure proposed by Luk and Mowry [20].

Hardware approach includes Jouppi's "stream buffers" [12], Fu and Patel's prefetching for superscalar and vector processors [8, 9], and Chen and Baer's lookahead mechanism [6] and known as the *Reference Prediction Table* (RPT) [7]. Mehrota [21] 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 [5], Joseph and Grunwald [11] implemented a simple Markov model to dynamically prefetch address references. More recently, Lai, Fide, and Falsafi [18] proposed a hardware mechanism to predict the last use of cache blocks.

Hybrid approaches includes the prefetch arrays proposal by Karlsson, Dahlgren, and Stenstrom [14] and VanderWiel and Lilja's *data prefetch controller* (DPC) [24].

3. Off-line Learning

Hardware predictors operate in two phases - a *learning* phase and a *prediction* phase. In the learning phase, the prediction facility is trained. Typically, this involves the updating of a prediction table or automaton. In the prediction phase, the learned table or automaton is used to make prefetch requests. In some schemes, during the prediction phase, the prediction table or automaton may also be updated, i.e. the learning and prediction phases are interleaved.

A major drawback of existing hardware schemes is the need to perform learning and prediction both at run time. This severely limits the type of learning schemes that one can use. We propose overcoming this limitation by taking the learning phase off-line. By using sample traces collected from an application, prediction tables and automata can be trained off-line. This rests on the important assumption that the sample traces used for the training do correctly reflect the behavior of the application during its actual run. The success of hardware prefetch mechanisms, all of which are based on learning past patterns to predict future references, provides strong circumstantial evidence for this.

The factors determining the success of a prefetch scheme are *accuracy*, *timeliness*, *overhead* and *coverage*. Accuracy refers to the percentage of prefetch requests issued are actually used. An accurately predicted prefetch request is useless if it is issued too early or too late relative to the actual use of the data. Any prefetch mechanism will have an associated overhead (which may be in the form of additional instructions, hardware investment, or increased bus utilization) that must not be too significant. Finally, the scheme must be able to cover most of the load misses. Unlike online schemes, off-line schemes can consider a significantly larger window of the sample trace and/or use more complex analysis and learning algorithms. This generally improves the accuracy of the prediction. Furthermore, by staying focus on program hotspots, coverage is improved. The issue of timeliness and overhead will be discussed when we outline our architectural solution.

4. Markovian Predictors

In this section, we shall describe our proposed Markovian predictors. Training traces of the application of interest are collected. In our experiments, these traces are first processed through a cache simulator so that we obtained only the miss traces. It should be emphasized that we used a trace generated by using a *different* input for the application in our experiments. During the sample trace collection phase, the application is also profiled to identify the "hotspots" - sections of code in which most of the load misses occur. These training sequences are then fed to a learning/analysis algorithm that outputs a prediction model for a particular hotspot. The prediction model is essentially a table with entries $(x, y_1, y_2, ..., y_n)$ where upon encountering miss address x, prefetch requests are issued for address $y_1, y_2, ..., y_n$. In subsection 4.4, we will describe how we encode the entries in the table so as to reduce the size of the prediction tables.

4.1. Simple Markov Predictor

This simple predictor is similar to the one used by Joseph and Grunwald [11]. Let T be the sample miss trace of an application. For two miss addresses, $x, y \in T$ say, the probability $P(y|x)$, i.e. the probability of x being followed immediately by miss address y, is computed. For each $x \in T$ in the miss trace, we compute $N(x) = \{P(y|x)\}$ where , $x, y \in T$ and $y \neq x$. In addition, from the trace we compute $f(x)$ which is the frequency of occurrences of x in T.

Next, we fix the size of the prediction table. This allows us to control the amount of hardware support needed. Since in practice, not all miss addresses can be accomodated, we need a hashing algorithm to access the table. Let $h(x)$ be the hashing function that maps x to its entry in the prediction table. We used a lookup mechanism that is similar to cache tag checking. We iterate through the rows of the prediction table. Of all the miss addresses that map to the same row,

we pick the one with the highest frequency of occurrences in the sample trace. Let p be the number of prefetch request entry per row. Having selected x, we simply use the p miss addresses of $N(x)$ with the highest probabilities. The value of p varies from 1 to 4 depending on the encoding described in section 4.4.

4.2. Window Markov Predictor

This is a new predictor. Instead of considering only the miss addresses that immediately follows x, we use a window of size w and consider all miss addresses within the window. In other words, if $y_1, y_2, ...$, is the sequence of miss addresses that follows x, then for the windowed Markov predictor, we use $N'(x) = \{P(y_i|x) \mid i \leq w,\ x, y_i \in T\}$. For our experiments, we chose w to be five. Another important modification is that we do not necessarily use up all p prefetch request slots.

4.3. Hidden Markov Model (HMP) Predictor

The Hidden Markov Model (HMP) is a well-known technique that has a wide range of applications [10, 17, 23]. Essentially, it is a Markov chain where each state generates an observation. HMP are known to be very useful for time-series modeling since the discrete state-space can be used to approximate many non-linear, non-Gaussian systems. There are established algorithms to train a HMP such as the Viterbi and Baum-Welch algorithms [13]. We extract the table from the HMP by examining the state transition and output probabilities.

4.4. Encoding the Prediction Table Entries

In order to reduce the size of the prediction tables, we used a stride-based encoding scheme. Given an entry from the predictive table derived above, where x is the miss address to start the prefetch process and y_1, y_2, y_3, and y_4 are the four prefetch targets. Without loss of generality, we shall assume that they are sorted in the probability of the prediction. Each of the methods above computes these probabilities. Consider the four displacements $d_1 = (x - y_1), ..., d_4 = (x - y_4)$.
There are four cases for the encoding:

- Case 1: All four displacements are in the integer interval [-128, 127]. We store all four displacements in a 4-byte word.

- Case 2: One of the four displacements cannot be stored in an 8-bit byte. The one that cannot be held in a byte is discarded.

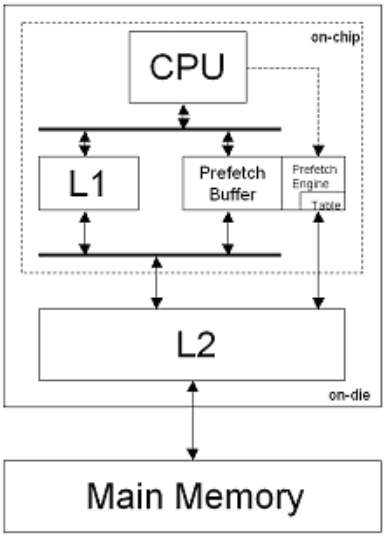

Figure 1. L1 Prefetch Architecture.

- Case 3: At least two of the four displacements are in the integer interval [-256, 255]. The two of the higher probabilities are stored in a 4-byte word.

- Case 4: None of the above. y_1 is stored as a full 4-byte address.

Two additional bits are needed to distinguish the case of the entry. This encoding scheme sacrifices on accuracy but results in a very compact table. The actual table size per hotspot is shown in Table 3. On the average, the prediction table for each hotspot is about 8Kbyte with about 2.8 predictions per table entry.

5. The Proposed Hardware Architecture

In this section, we will describe the proposed architectures in which the off-line prediction tables can be effectively deployed. The techniques described can be used to prefetch data into the L1 data cache or the L2 data cache. We begin by assuming a canonical machine with the non-blocking L1 data cache on-chip, a small prefetch buffer, and a L2 data cache that is off-chip but on-die. Fig. 1 and Fig. 2 show the proposed architecture for L1 and L2 prefetching, respectively. We have described how the prediction tables are constructed off-line, and shall now describe how the scheme will work at runtime. By means of the training trace, a special "**load-predictor [table-addr]**" instruction is inserted into earliest branch that, in the trace, leads to a new hotspot as shown in Fig. 3. An important issue is whether there is sufficient time to preload the predictor table. If we assume that the table is 8Kbyte, and the bus width

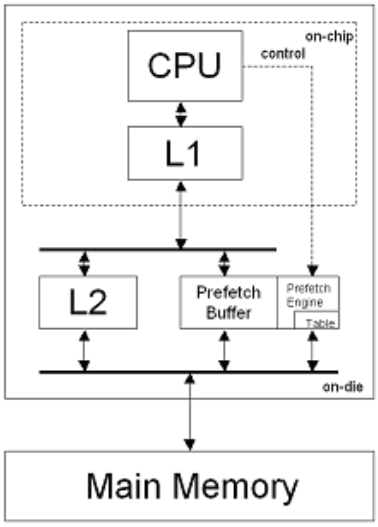

Figure 2. L2 Prefetch Architecture.

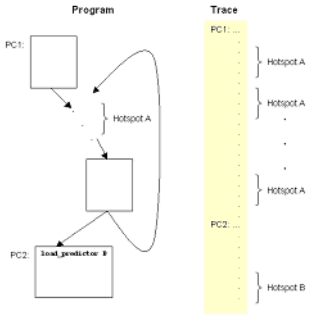

Figure 3. Insertion of load-predictor instruction

for the L1 and L2 architecture is 256 bits and 128 bits, then for a 2 GHz processor using 400MHz quad-pumped bus, we estimate that it will take about 1500 CPU cycles (inclusive of startup latencies) to load in a 8Kbyte table. Table 2 shows average distance in terms of clock cycles between neighboring hotspots. It should be noted that the distance between neighboring hotspot candidates was also an important consideration in the final choice of hotspots.

Once the table is loaded, the prefetch engine will examine the miss addresses reported by the cache unit. Using the standard tag checking mechanism, the prefetch engine will probe the prediction table. When there is a hit in the prediction table, the prefetch engine will decode the entry and issue the prefetch requests.

The mechanism for L2 prefetch is a variation of the L1 mechanism except that instead of requiring an additional port to L2 memory, the table is fetched by cycle-stealing from the main memory bus.

	Training input	Testing input
130.li	input1/train.lsp	input2/*.lsp
181.mcf	input_train/inp.in	input_ref/inp.in
183.equake	input_train/inp.in	input_ref/inp.in
164.gzip	input_train/input.combined 32	input_test/input.compressed 2
188.ammp	input_train/ammp.in	input_test/ammp.in
mst	1024 1	684 6
treeadd	20 1	40 6
bisort	250000 0	19600 4
tsp	1000000 0	3000000 0
health	5 500 1	3 250 2

Table 1. Training and testing inputs (arguments) for each benchmark tested

6. Experimental Setup

We use the Trimaran compiler-EPIC architecture simulation infrastructure [3] to evaluate the performance of our proposed system and of each of the three off-line learning algorithms outlined above. We compared the performance of our system against that of using larger caches, and the RPT hardware prefetch scheme of Chen and Baer [7]. For evaluation, we used 130.li of SPEC 95, 181.mcf, 183.equake, 164.gzip, 188.ammp all from the SPEC 2000 suite [2] and bisort, mst, treeadd, tsp, health from well-known Olden Pointer Benchmark suite.

Our baseline setup is an IA64-like EPIC machine [15] with four integer, two floating point and two memory units and a 32Kbyte L1 cache and a 256Kbyte L2 cache. We computed stall cycles for L1 and L2 load misses when L1 cache size is 32K, 64K and 128K with 256K L2 cache.

Our main metric for characterizing the performance of the memory system is *stall cycles*. Stall cycles account for a significant portion of the execution times of data intensive applications. Most of the memory stall cycles come from load misses and hence reducing load misses has a significant impact in improving overall performance. Our EPIC machine is an in-order machine, and we assumed a "stall-upon-use" latency model. In this stalling model, a load instruction that causes a cache miss will not immediately block the pipeline. The pipeline is stalled only at the earliest attempt to use the data that is to be loaded.

We first built the prediction table per each hot spot for each benchmark we tested using training input sets through offline learning methods. Then we ran the simulation again using different input sets and generated load miss traces for level 1 and level 2 cache misses. The training and testing input for the experiments are described in Table 1.

For each benchmark, we selected certain basic blocks where most load misses occurred through profiled information and assign them as candidates of hot spots. To be chosen as a hot spot, there should be a large enough gap be-

tween the neighboring hotspots. For example, the Treeadd benchmark of Olden Pointer benchmark suite comprises of 33 total basic blocks and load miss occurred in only 11 of those 33 blocks. Moreover 75% of entire load misses came from one particular basic block, basic block number 4 of treeadd procedure. We chose this basic block as our first hot spot and next candidate for hotspot was block number 6 of treeadd procedure where 20% of entire load misses came from. But the average latency between this block and block number 4 was just 388 cycles which was less than our threshold of 5000 for choosing hotspots, so even though basic block number 6 was one with second most load misses, it was not chosen as our hot spot during our experiments. Basic block number 6 of treealloc procedure was chosen as our second hotspot since its average latency to the chosen hotspot was 190,826 cycles ensuring that there is enough time to load the prediction table for this hotspot during runtime. The total number of hot spots ranges from 2 (treeadd) to 19 (130li) as seen in Table 2. The table also reports the average distances of neighboring hot spots.

Benchmark	Cache	Total number of hotspots	Average distance between hot spots (cycles)
130.li	L1	12	48,405
	L2	15	54,885
181.mcf	L1	14	75,694
	L2	14	92,289
183.equake	L1	17	42,448
	L2	18	97,291
Mst	L1	8	8,012
	L2	12	12,830
treeadd	L1	2	190,826
	L2	2	210,211
bisort	L1	11	100,215
	L2	14	113,356
Tsp	L1	12	89,402
	L2	15	114,129
health	L1	18	40,918
	L2	22	69,206

Table 2. Characteristics of hotspots in the benchmark.

As explained in Section 4.4, we used a stride-based encoding scheme to get a realistic size of prediction tables. Table 3 shows the result of applying this scheme to our implementation. For each benchmark, we measured the average percentage of each 4 cases after Hidden Markov and Window Markov predictors' learning phase ends and they each provide the prediction table. As shown in Table 3, the Hidden Markov Predictor shows the tendency of having prediction addresses that are far apart in comparison to the Window Markov Predictor of window size 5. This eventually led to a lesser number of prediction addresses in the prediction table because many addresses that are far away are discarded in the final prediction table. The result show Window Markov Model not only contains more prediction addresses for particular miss address in the encoded prediction table but also its prediction accuracy was much higher than Hidden Markov Model. In Fig. 4, we tested our Window Markov Model with different window size and the best result came from window size 5. Performance deteriorated as window size is increased. Those results strongly show the existence of data locality characteristics even in pointer intensive applications.

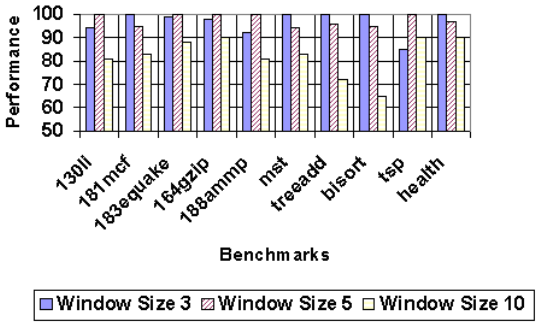

Figure 4. Window Markov Predictor with various window size.

Table 4 gives the detailed breakdown of the performance of the Window Markov Predictor. It shows that the predictor do indeed reduce the overall number of load misses in the applications. Columns 6 and 7 report the coverage of the predictor. This is the percentage of load misses in the hotspots that hit the prediction table causing prefetch requests to be sent out. The last two columns is the ratio of wasted prefetch requests (i.e. mispredictions) for each (overall) load miss. We argue that although we did not simulate actual bus transactions and bandwidth, these ratios indicate that the overhead caused by prefetch requests is low. We attribute this to the good accuracy and coverage of the predictor.

Fig. 5 shows the effect of increasing miss penalties on the various schemes that we tested on the 188.ammp benchmark. In the top diagram, the L2 miss penalty is fixed at 93 cycles. This was obtained from the actual measurements reported [1]. L1 miss penalty was varied from 12 to 38. In the lower diagram, a L1 penalty of 25 is assumed while the L2 penalty was varied from 30 to 162. What is interesting to note is that the slope for the Window Markov Predictor is gentler than that of others. The gap in memory and processor speed is increasing resulting in larger miss penalties. The Window Markov Predictor seems to show more promise than the other schemes in tolerating larger penal-

Benchmark		Av. % of Case 1	Av. % of Case 2	Av. % of Case 3	Av. % of Case 4	Encoded Tab. Sz.	Accuracy
130.li	HMP L1	1.9%	5.3%	26.1%	66.7%	9.3 KB	39.2%
	HMP L2	2.3%	2.7%	13.7%	81.3%	11.8 KB	38.3%
	WMP L1	46.4%	38.5%	4.6%	10.5%	7.8 KB	61.6%
	WMP L2	48.6%	32.9%	7.8%	10.7%	7.2 KB	78.4%
181.mcf	HMP L1	1.6%	3.4%	12.9%	82.1%	9.7 KB	28.6%
	HMP L2	1.9%	5.7%	9.2%	83.2%	7.2 KB	26.9%
	WMP L1	50.6%	40.2%	1.4%	7.8%	7.5 KB	76.2%
	WMP L2	13.7%	52.2%	17.7%	17.4%	8.4 KB	75.5%
183.equake	HMP L1	1.3%	2.9%	6.6%	89.2%	9.1 KB	8.1%
	HMP L2	1.7%	3.1%	7.5%	87.7%	8.4 KB	11.1%
	WMP L1	48.7%	31.4%	8.6%	11.3%	5.3 KB	66.8%
	WMP L2	72.8%	20.7%	2.1%	4.4%	4.6 KB	48.6%
164.gzip	HMP L1	5.4%	10.2%	21.4%	63.0%	9.7 KB	8.9%
	HMP L2	8.2%	18.9%	16.4%	56.5%	9.4 KB	7.1%
	WMP L1	38.5%	42.2%	10.6%	8.7%	6.3 KB	54.9%
	WMP L2	43.1%	37.9%	5.1%	13.9%	5.8 KB	58.8%
188.ammp	HMP L1	8.3%	3.2%	21.6%	66.9%	8.1 KB	13.1%
	HMP L2	5.7%	8.2%	16.8%	69.3%	8.0 KB	9.4%
	WMP L1	44.7%	25.5%	4.6%	25.2%	6.3 KB	70.3%
	WMP L2	53.8%	30.7%	2.1%	13.4%	5.9 KB	64.8%
bisort	HMP L1	3.6%	7.2%	67.1%	22.1%	5.3 KB	15.9%
	HMP L2	2.1%	3.1%	39.6%	55.2%	4.3 KB	17.4%
	WMP L1	37.8%	41.2%	3.0%	4.9%	6.0 KB	87.9%
	WMP L2	41.2%	32.8%	13.3%	12.7%	7.6 KB	82.8%
mst	HMP L1	1.2%	1.7%	7.8%	89.3%	8.5 KB	21.7%
	HMP L2	2.5%	3.9%	11.2%	82.4%	7.3 KB	14.9%
	WMP L1	80.3%	6.0%	12.4%	1.3%	12.2 KB	68.2%
	WMP L2	68.1%	8.9%	21.6%	1.4%	10.6 KB	62.2%
treeadd	HMP L1	1.2%	1.4%	18.8%	78.6%	3.1 KB	18.6%
	HMP L2	2.3%	4.8%	30.2%	62.7%	2.8 KB	12.4%
	WMP L1	78.3%	5.4%	14.5%	1.8%	3.9 KB	75.9%
	WMP L2	70.1%	22.8%	4.6%	2.5%	3.6 KB	66.6%
tsp	HMP L1	1.3%	4.9%	27.8%	66.0%	7.7 KB	38.0%
	HMP L2	3.2%	2.0%	25.9%	68.9%	5.8 KB	36.4%
	WMP L1	39.6%	16.8%	41.4%	2.2%	9.9 KB	43.7%
	WMP L2	30.1%	15.0%	52.0%	2.9%	8.3 KB	40.9%
health	HMP L1	3.9%	2.1%	25.5%	68.5%	13.3 KB	18.7%
	HMP L2	1.8%	3.5%	28.6%	66.1%	11.3 KB	15.5%
	WMP L1	43.5%	31.1%	12.4%	13.0%	8.1 KB	73.6%
	WMP L2	41.2%	28.6%	17.3%	12.9%	7.6 KB	78.2%

Table 3. Statistics for Table Encoding obtained for each Benchmark.

Benchmark	L1 load misses w/o WMP prefetch	L2 load misses w/o WMP prefetch	L1 load misses with WMP prefetch	L2 load misses with WMP prefetch	L1 load coverage	L2 load coverage	L1 prefetch overhead	L2 prefetch overhead
130.li	11.17M	2.13M	6.19M	0.83M	44.6%	60.8%	0.48	0.37
183.equake	977.89M	472.61M	564.83M	156.53M	42.2%	66.9%	0.39	0.96
164.gzip	581.75M	142.81M	241.43M	61.58M	58.5%	56.9%	0.74	0.66
188.ammp	408.57M	178.33M	155.58M	74.72M	61.9%	58.1%	0.51	0.57
181.mcf	774.65M	429.93M	350.76M	204.22M	54.7%	52.5%	0.36	0.36
Tsp	1.78M	0.61M	1.10M	0.40M	38.3%	34.8%	0.60	0.58
Treeadd	0.54M	0.26M	0.25M	0.13M	54.0%	50.2%	0.36	0.47
Mst	3.15M	2.54M	1.84M	1.55M	41.5%	39.1%	0.37	0.41
Health	18.19M	9.64M	9.69M	4.98M	46.7%	48.4%	0.34	0.30
Bisort	2.47M	1.05M	0.90M	0.39M	63.5%	62.3%	0.21	0.30

Table 4. Coverage of L1 and L2 load misses of WMP predictor.

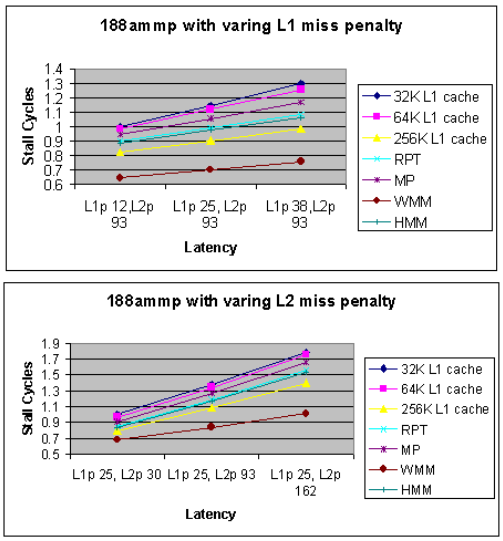

Figure 5. Effect of increasing miss penalty.

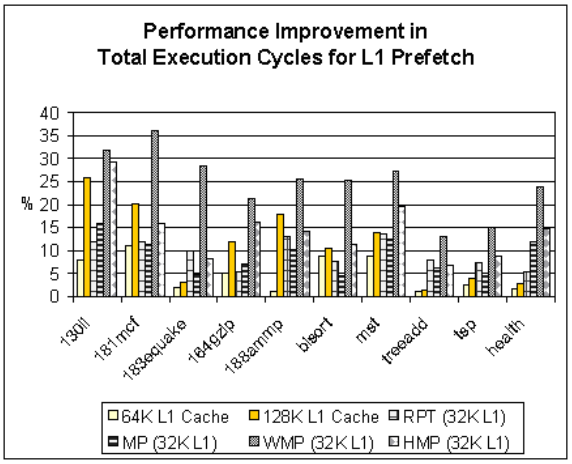

Figure 6. Performance improvement in total cycles for L1 Prefetching (normalized by 32K L1 cache and 256K L2 cache without using any prefetching scheme).

ties, especially in the L2 cache.

The percentage performance improvements for L1 and L2 prefetching are shown in normalized graphs of Fig. 6 and Fig. 7 with the base case being that of a machine with 32KByte L1 cache and 256KByte L2 cache without using any prediction scheme. We measured performance improvement by dividing total execution cycles after certain prefetching scheme was applied by total execution cycles without any prefetching scheme. The results shows that increasing L1 cache size does not necessarily improve performance especially for data intensive applications using dynamic data structures like pointers. In one instance, a 47% improvement in performance was recorded using Window Markov Predictor. In almost all cases, the use of off-line learning algorithms gave a pronounced performance improvement over that of simply increasing the cache size or a hardware prefetch scheme like RPT. In particular, the Window Markov Predictor gives the best performance.

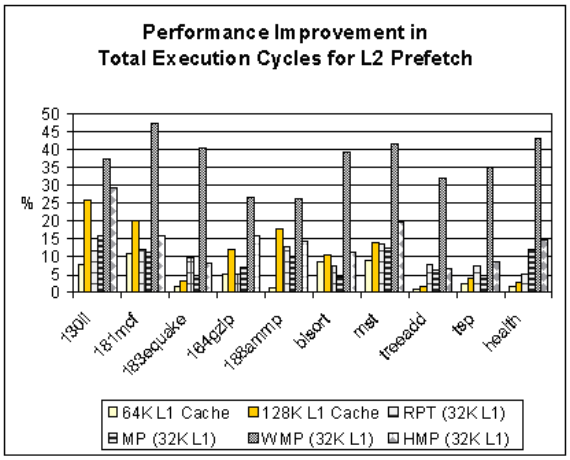

Figure 7. Performance improvement in total cycles for L2 Prefetching (normalized by 32K L1 cache and 256K L2 cache without using any prefetching scheme).

7. Conclusion

In this paper, we proposed a paradigm and architectural framework for the use of off-line learning algorithms in the prefetching of data. In all the benchmarks that we tested, our off-line learning scheme gave improved performance more significantly than other schemes such as increasing the cache sizes. The off-line approach allows for even more aggressive analysis and prediction schemes. Our future research seeks to develop more powerful learning module. Furthermore, we believe that off-line learning can also be adapted to software prefetching, and we are currently

also examining that approach which will require even lesser hardware support.

References

[1] The Calibrator: a cache-memory and TLB calibration tool. http://www.cwi.nl/ manegold/calibrator/db/db.shtml.

[2] The SPEC benchmarks. http://www.spec.org.

[3] The Trimaran Compiler Infrastructure. http://www.trimaran.org.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[5] M. Charney and A. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, February 1995.

[6] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proceedings of International Symposium on Computer Architecture*, pages 223 – 232, 1994.

[7] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processor computers. *IEEE Transactions on Computers*, 44-5:609 – 623, May 1995.

[8] J. W. C. Fu and J. H. Patel. Data prefetching strategies for vector cache memories. In *Proceedings of the International Parallel Processing Symposium*, 1991.

[9] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 102 – 110, 1992.

[10] F. Jelinek. Self-organized language modeling for speech recognition. Technical report, IBM T. J. Watson Research Center, 1985.

[11] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 252 – 263, 1997.

[12] N. P. Jouppi. 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*, pages 364 – 373, 1990.

[13] J. D. Jr., J. Proakis, and J. Hansen. *Discrete-time Processing of Speech Signals*. MacMillan, 1993.

[14] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 7th International Sysmposium on High-Performance Computer Architecture*, pages 206 – 217, 2000.

[15] V. Kathail, M. Schlansker, and B. Rau. Hpl-pd architectural specifications: Version 1.1. Technical Report Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, Revised 2000.

[16] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43 – 53, 1991.

[17] A. Krogh, S. Mian, and D. Haussler. A hidden markov model that finds genes in e. coli dna. *Nucleic Acids Research*, 22:4769 – 4778, 1994.

[18] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlation prefetchers. In *Proceedings of the International Parallel Processing Symposium*, pages 144 – 154, 2001.

[19] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. Spaid: Software prefetching in pointer and call-intensive environment. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 231 – 236, 1995.

[20] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222 – 233, 1996.

[21] S. Mehrota and H. Luddy. Examination of a memory classification scheme for pointer intensive and numeric programs. Technical Report CRSD Tech. Report 1351, CRSD, University of Illinois, December 1995.

[22] T. C. Mowry, M. S. Lam, and A. Gupta. 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*, pages 62 – 73, 1992.

[23] A. Nadas. Estimation of probabilities in the language model of the ibm speech recognition system. *IEEE Transactions on Acoustics, Signal and Speech Processing*, 32(4):859 – 861, 1984.

[24] S. Vanderwiel and D. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of the International Conference on Computer Design*, pages 372 – 377, 1999.

[25] S. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Survey*, 32(2):174 – 199, June 2000.