Multi-level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matching*

Marius Grannaes, Magnus Jahre, and Lasse Natvig

Norwegian University of Science and Technology

Abstract. This paper presents a low complexity table-based approach to delta correlation prefetching. Our approach uses a table indexed by the load address which stores the latest deltas observed. By storing deltas rather than full miss addresses, considerable space is saved while making pattern matching easier. The delta-history can predict repeating patterns with long periods by using delta correlation. In addition, we propose L1 hoisting which is a technique for moving data from the L2 to the L1 using the same underlying table structure and partial matching which reduces the spatial resolution in the delta stream to expose more patterns.

We evaluate our prefetching technique using the simulator framework used in the Data Prefetching Championship. This allows us to use the original code submitted to the contest to fairly evaluate several alternate prefetching techniques. Our prefetcher technique increases performance by 87% on average (6.6X max) on SPEC2006.

1 Introduction

In 2004, Gracia Perez et al. [1] published a paper that evaluated several prefetching techniques in a common framework. They found that several techniques were not as good as the original authors claimed. This discrepancy was due to researchers using different simulator infrastructure and benchmarks as well as the difficulty in implementing other techniques due to a lack of documentation. In this work, we avoid these problems by using the simulation infrastructure and original code from the first Data Prefetching Championship (DPC-1). This competition was similar to the earlier JILP Championship Branch Prediction Competition (CBP). In order to ensure a fair comparison of prefetcher performance, the organizers published a common simulator framework. Each prefetcher could use a maximum of 4KB of storage, but there was no limit on prefetcher complexity. Each contestant submitted their code to the competition for evaluation. This code was later published. This allows us to do a fair comparison with the top three DPC entries using their submitted code.

Our submission, Delta Correlating Prediction Tables (DCPT), used a table indexed by the PC of the load [2]. Each table entry stores a large amount of

* This work was supported by the Norwegian Metacenter for Computational Science (Notur).

history per load instruction in the form of deltas. By storing deltas rather than full miss addresses, we save a significant amount of memory and make pattern matching easier. Pattern matching is done by using Delta Correlation, originally proposed by Nesbit et al. [3]. This technique is very effective at detecting patterns with periods shorter than the amount of history stored.

In this paper, we improve DCPT by proposing DCPT-P which incorporates many of the lessons learned during DPC-1. We introduce the concept of L1 hoisting, which is a highly accurate and timely method for moving data into the L1 cache. L1 hoisting does not require complex additions to the L1 cache which could interfere with the critical path of the processor. The key idea in L1 hoisting is to first issue prefetches to the L2 cache with a high prefetch distance, thus ensuring timeliness in the L2 cache. To further increase performance, we predict when the prefetched data will soon be used and hoist it to the L1 cache.

Second, we introduce partial matching which is a technique to enhance delta correlation in hard to predict cases such as pointer chasing. Partial matching reduces the spatial resolution in the delta stream to reveal more possibilities for prefetching. Thus, this technique increases coverage at the price of reduced accuracy, for an overall increase in performance.

2 Previous Work

Because of the large gap between the latency of the processor and main memory, prefetching has a large potential for increasing processor performance. Therefore, it has been an active research topic for several decades. The simplest prefetcher is sequential (next line) prefetching, which simply fetches the next line whenever a cache line is accessed, thus exploiting spatial locality [4]. Its improvement, tagged sequential prefetching, uses an extra bit per cache line to indicate that this cache line was prefetched. When the processor subsequently hits in the cache on a cache block with this bit set, it fetches the next block.

Reference prediction tables use a table to store the recent history of a single load [5]. Each table entry is indexed by the address of the load and contains the last miss address as well as the delta (the difference between the address of the latest consecutive misses) as well as a state [6]. Then, on the next miss, the delta between the first miss address and the current is computed and stored in the table and the entry enters the training state. Finally, on the third miss, a new delta is computed. If that delta matches the one found in the table, the entry enters the prefetching state and prefetches are issued by using the computed delta.

The use of a Global History Buffer (GHB) was proposed by Nesbit et al. [3]. A GHB is essentially a FIFO containing the last misses observed by the memory system. Each entry in the GHB is linked to the previous entry of its class by a pointer. Because of the versatility of the GHB, a class can be defined in multiple ways such as belonging to the same memory region (C/DC) or originating from the same load (PC/DC) [7]. In PC/DC the entries in the GHB belong to the same class if they originate from the same load instruction.

By traversing the linked list, a miss history can be obtained for that load. This operation can be expensive in terms of energy and latency as the GHB structure is read multiple times to generate the miss history. In PC/DC, the deltas between consecutive misses are computed and stored in a delta table. This operation is repeated every time a L2 miss occurs. After the history of deltas are computed, delta correlation begins. Delta correlation means searching for the most recent pair of deltas in the delta history. If a corresponding pair is found in the delta history, the deltas after the match is used to predict future deltas.

During the first Data Prefetching Championship (DPC-1) several novel prefetcher designs were presented. Second place was awarded to GHB-LDB (Global History Buffer - Local Delta Buffer) which was proposed by Dimitrov et al. [8]. GHB-LDB improves upon the PC/DC prefetcher by also including global correlation (as opposed to the local correlation directed by the PC of the load) and most common stride prefetching. Furthermore, their prefetcher issues prefetches directly into the L1 cache.

Third place was awarded to Ramos et al. [9] for their multi-level prefetcher based on the PC/DC concept. Their PDFCM (Prefetching based on a Differential Finite Context Machine) prefetcher uses a hash-based approach with two tables. The History Table is indexed by the PC which contains a hashed representation of the recent history of that entry. This hash points to an entry in the Delta Table which contains the predicted delta. By computing new hashes based on the predicted deltas, an arbitrary prefetch degree and distance can be used.

Finally, the winner was the AMPM (Access Map Pattern Matching) prefetcher proposed by Ishii et al. [10]. Their prefetcher divides memory into hot zones similar to Czones [7]. Each hot zone is tracked by using a 2-bit vector for each cache line in that zone. This vector is then analyzed to see if there are any constant stride patterns in that zone. If there are any patterns, the predicted pattern is prefetched.

3 Delta Correlating Prediction Tables

3.1 Overview

The core of our prefetching heuristic is a table indexed by the PC of the load. Each entry has the format shown in Figure 1. In addition to the PC tag, each entry holds the last miss address, the address of the last prefetch that was issued in addition to a circular buffer containing the last n deltas. The circular buffer is managed by the *delta pointer*. This field points to the most recently added delta.

This organization has a number of advantages. Each entry holds a comparatively large history which can be used to predict any repeating pattern as long as the period is shorter than $n-2$. In addition, entries do not compete for space, thus ensuring that the amount of history per entry is monotonically increasing, which reduces the risk that prefetches are issued for the same line. Finally, by storing deltas, rather than full miss addresses it is possible to save considerably memory space.

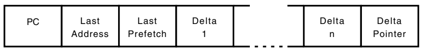

Fig. 1. Format of a single DCPT-P entry

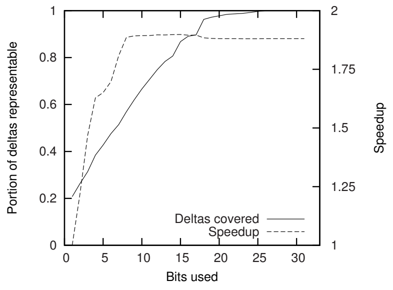

Fig. 2. Impact of increasing the numbers of bits used to represent a delta

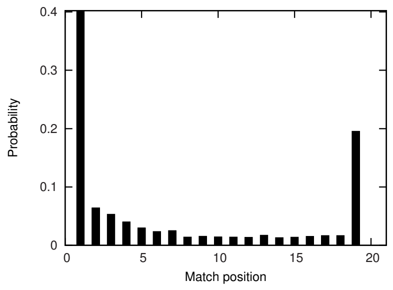

Fig. 3. Position in the circular buffer where a match is found

In Figure 2, we show the portion of deltas we observed that can be represented as a function of the number of bits used to represent each delta in the table. By far, the most common delta is one which is to be expected as this represents the common sequential pattern. As the number of bits per delta increases, the portion of the deltas we can represent increases monotonically.

Figure 2 also plots the performance impact of increasing the number of bits used to represent a single delta. Interestingly, the speedup has a much steeper slope than the coverage. Performance rises sharply as one increases the number of bits up to 12, and then trails off. Although more bits increases the information content, performance degrades because of false matches (high delta values are often generated by pointer chasing codes). Thus, performance can be improved and the memory footprint reduced by limiting the number of bits used.

3.2 DCPT-P Implementation

A basic implementation of the DCPT-P pipeline is shown in Figure 4. When there is an access to the L2 cache the same request enters the pipeline. The first step is to look up the PC of the load in the table. If a corresponding entry is not found, an old entry is replaced using a LRU replacement policy. This new entry is initialized with the miss address and the rest of the entry is initialized to zero.

If a corresponding entry is found, we first compute the delta between the current access and the value stored in *last address*. If the delta is not zero, then the delta is stored in the circular buffer and the *delta pointer* and *last address* is updated. In our experiments, the L2 cache uses 128 byte cache blocks. To conserve space we mask out the lower six bits (64). Thus, a delta of two represents an increment of a single cache block. As shown by Hur et al. [11],

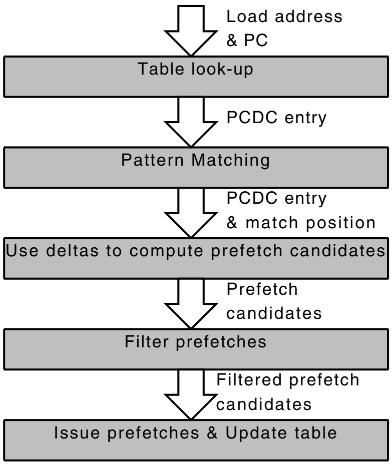

Fig. 4. DCPT-P Pipeline

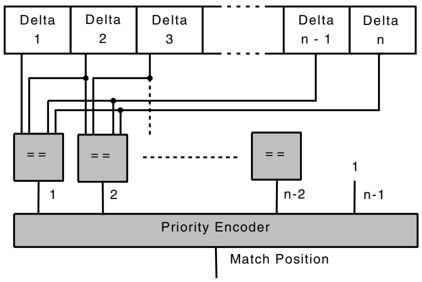

Fig. 5. Pattern matching implementation

many streams are short (2-4 cache lines). By using deltas that are smaller than a cache block we enable DCPT-P to start prefetching without waiting for too many misses to the L2. If we cannot represent a delta with the available bits, we store a zero instead (not valid). Finally, the entry is passed on to the pattern matching step.

The pattern matching logic is similar to the logic used in PC/DC [3]. In essence, we search for the first occurrence of the last pair of deltas in the circular stream. In Figure 3, we show the distribution of match locations in a 20 entry circular delta buffer. There are two peaks. The first peak is at the first possible position (the last two deltas in the circular buffer matches the first two deltas). This position represents constant strides or repeating patterns (for example 1-2-1-2-1-2). However, a match in the first possible position does not necessarily mean that the other stored deltas are redundant. Consider a blocking implementation of a matrix multiply. In this situation, the access pattern would be a series of sequential accesses followed by a large stride when the blocking algorithm moves to the next row, which in turn would be followed by a series of sequential accesses. By storing multiple deltas in this manner, this behaviour can be effectively captured by DCPT-P. The last peak (at 19) represent situations where the pattern is not found. This data point is included to illustrate the amount of times no pattern is found. Our implementation of the pattern matching step uses several comparators working in parallel in combination with a priority encoder as shown in Figure 5.

The next step is to generate prefetch candidates. The first prefetch candidate is generated by adding the first delta after the match to the current miss address. The second prefetch candidate is generated by adding the second delta after the match to the previous miss address. This is done for all deltas after the match. Thus, by increasing the number of deltas per table entry the prefetch distance is also increased.

Table 1. Example delta stream

Address:	10	11	20	21	30
Deltas:		*1*	*9*	**1**	**9**

As an example, consider the stream shown in Table 1. In this example, time increases to the right (i.e. the most recent address observed is 30). The last pair of deltas is thus (1,9) (Marked with **boldface**). We search for this pair of deltas and find the same pair of deltas in the beginning of the stream (Marked with *italics*). The next delta after this match is 1. We then add 1 to the last *last address* (30) and obtain 31. This is our first prefetch candidate. The next delta is 9. In a similar manner we add 9 to the previous prefetch candidate and obtain 40. We repeat this procedure for all the deltas in the circular buffer.

This approach generates several redundant prefetches so prefetch filtering is needed. The most important mechanism is the *last prefetch* field in each entry. This entry keeps the address of the last prefetch issued by that entry. If a candidate is made that matches the *last prefetch* field during prefetch candidate generation, all previous prefetch candidates are dropped. In the steady state, this ensures that only a single prefetch is issued.

We use a 32 entry pending prefetch buffer to store the prefetches that have been issued. This table serves a dual purpose; first it is checked prior to issuing a prefetch request, thus eliminating redundant prefetches. Second, by only allowing 32 outstanding prefetch requests we limit the amount of bandwidth used by the prefetcher and the probability of severe bandwidth contention.

3.3 L1 Hoisting

Although the greatest latency is from the last level cache to the main memory, there is a significant performance potential to prefetching into the L1 cache. However, due to its limited capacity, cache pollution becomes a significant problem. To avoid this, highly accurate and timely prefetches are needed. In addition, because the L1 cache is on the critical path it becomes much more difficult to construct large and complex prefetch heuristics that interact with the L1 access stream without degrading overall performance.

To overcome this problem we propose L1 hoisting. L1 hoisting is a natural addition to DCPT-P. DCPT-P is highly accurate, but issuing prefetches directly into the L1 cache brings the data in too early and displaces data that is currently needed, which in turn reduces overall performance. Our solution is prefetch hoisting. The first prefetch candidate that is generated is treated as a candidate for prefetch hoisting as well. This candidate is predicted to be the next required by the processor. In the steady state, this candidate has already been prefetched into the L2 by an earlier miss by the same load. Thus, we check if this block is present in the L2. If it is present, then the block is moved (hoisted) into the L1. Even though prefetch distance is low (only one block) it is enough to be timely, because the latency from the L1 cache to L2 cache is much lower than the latency from L1 to main memory.

```
1 for (i = x.size(); i-- > 0; ++xi) {
2    svec = const_cast<SVector*>(& A[*xi]);
3    elem = &(svec->element(0));
4    last = elem + svec->size();
5    y = vl[*xi];
6    for (; elem < last; ++elem)
7       v[elem->idx] += y * elem->val;
8 }
```

Listing 1.1. Loop from 450.Soplex

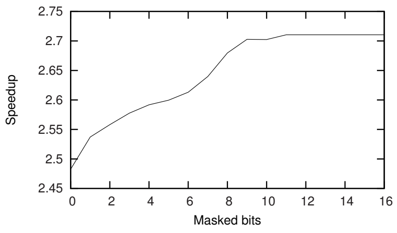

Fig. 6. Speedup of Sphinx as a function of LSB masked in partial matching

3.4 Partial Matching

DCPT captures most regular repeating patterns. However, many programs exhibit more complex and irregular patterns. Consider the code from soplex shown in Listing 1.1. Although the load in line 7 might seem hard to predict there is some structure to the addresses issued. One pattern of deltas we observed was *-2, -1, 4, -2, -3, -3, -1, 3*. In this case, there are no repeating pair of deltas, but most deltas are small. Because the observed deltas are so small, using previous deltas to issue new prefetches might be beneficial. Another pattern we observed was *9, 9, 9, 9, 9, -54, 73, 9, 9, 9*. In this case, a regular pattern is interrupted by an abrupt jump. Simply prefetching using the most common delta (9) would be preferable.

In this work, we propose a general approach to exposing such patterns called partial matching. If a pattern is not found using the exact match, we try partial matching. In essence, we reduce the spatial resolution by masking out the least significant bits and try to find a match using only the MSB's of the delta. This allows us to issue prefetches in both of the cases above.

In Figure 6, we show the speedup of the benchmark sphinx as a function of the number of LSB masked. Increasing the number of masked bits increases the number of prefetches issued. In the case of Sphinx, many of these prefetches are hits, but in other benchmarks increasing the number of masked bits increases the probability of cache pollution and wasted bandwidth.

4 Methodology

Gracia Perez et al. [1] showed that the choice of simulator and benchmarks as well as the implementation of other data cache mechanisms can severely bias the results when evaluating prefetcher performance. Therefore, to evaluate our prefetcher proposal we have used the Data Prefetching Championship (DPC-1) simulator framework [12] as well as the code submitted by the contestants to the competition.

The simulator framework is based on the CMP$im simulator [13]. This framework models a simple 15 stage, 4 wide out-of-order core with a 128-entry instruction window. The core can issue a maximum of two loads and a single store each cycle. The framework models a two level cache hierarchy, consisting of a 32KB, 8-way L1 cache with 64B cache lines. The L2 is a 2MB 16-way set-associative cache with 128 Byte cache lines and a LRU replacement policy. The second level cache has a 20 cycle latency, while main memory has a 200-cycle latency. Each cache is coupled with a queue for storing outstanding requests to the next level in the hierarchy. These queues issues requests in FIFO order and does not prioritize demand requests over prefetch requests [12]. The queue to main memory issues one request per 10 clock cycles, while the queue to the L2 issues 1 per clock cycle. This simulator setup was referred to as configuration 2 in DPC-1.

For our experiments we have generated traces for the SPEC2006 [14] benchmark suite. Each benchmark was fast forwarded by 40 billion instructions and then executed for 100 million instructions. The benchmarks were compiled with the Intel C Compiler version 10.0.

To evaluate the performance of our prefetching heuristic we have selected 5 state-of-the-art prefetchers. In the study by Gracia Perez et al. [1] mentioned earlier, Reference Prediction Tables [5] and PC/DC using a GHB [3] were found to give the highest performance. Therefore, we have implemented these two approaches with the same 4KB limitation. In addition, we have selected the top three performers from DPC-1. The contestants' prefetching code was made public after the competition so we have used their code without modification. The top performers were AMPM [10], PDFCM (Maxperf) [9] and GHB-LDB [8].

To keep within the same 4KB limit imposed on the other prefetcher implementations we have used a 95 entry table with 20 12-bit deltas. On the pattern matching pass with partial matching we mask the low 8 bits of the delta. The pending prefetch buffer can hold a maximum of 32 requests.

5 Results

We begin our evaluation by comparing the performance of our prefetcher to the top three DPC-1 prefetchers, Reference Prediction Tables and PC/DC with the SPEC2006 benchmark suite. The results are shown in Figure 7 and 8. In all of the results presented in this paper, speedup refers to a speedup compared to a baseline where no prefetching is performed. Because there is a wide range of speedups (up to 6.6X) we have opted to use two graphs to increase readability.

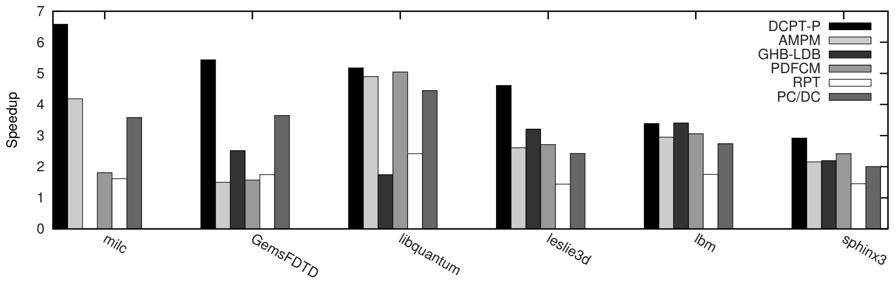

Fig. 7. 2 MB L2 cache. Benchmarks with large speedups.

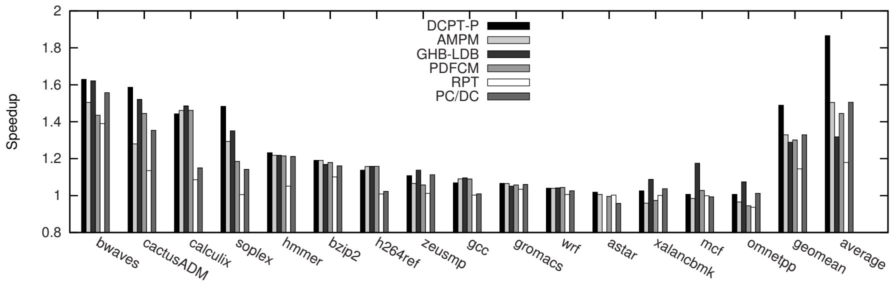

Fig. 8. 2 MB L2 cache. Benchmarks with small speedups.

In addition, we do not show the benchmarks dealII, gobmk, tonto, perlbench, sjeng, gamess, namd, povray. In all of these benchmarks, the performance impact of prefetching was less than 5% for all the prefetchers. In cases where the simulation did not terminate within 48 hours we show an speedup of 0, rather than tampering with the original code.

Overall, DCPT-P shows good performance across all benchmarks. DCPT-P is the best performing prefetcher on 11 of the 21 benchmarks shown. The good performance of both soplex and sphinx3 is due to partial matching. Leslie3d and milc benefits greatly from the L1 hoisting technique. Also, it is worth noting that GHB-LDB performs very well on xalncbmk, mcf and omentpp. This is due to the global (intra-PC) analysis done by this type of prefetcher. However, GHB-LDB performs worse than it's predecessor, PC/DC, on GemsFDTD and libquantum. Although both GHB-LDB and PDFCM both extends PC/DC, their performance is on average almost equal. Although AMPM prefetching is not the best prefetcher for any single benchmark, it nevertheless achieves signifcant speedups across the entire benchmark suite. On average, DCPT-P provides an arithmetic mean speedup of 87%. AMPM, GHB-LDB and PDFCM has speedups of 50%, 32% and 44% respectively.

In Figure 9 and 10 we reduce the L2 capacity to 512KB. This is the same configuration as config 3 in DPC-1. Overall, we observe the same general trends. The most significant changes from reducing the size of the L2 can be observed

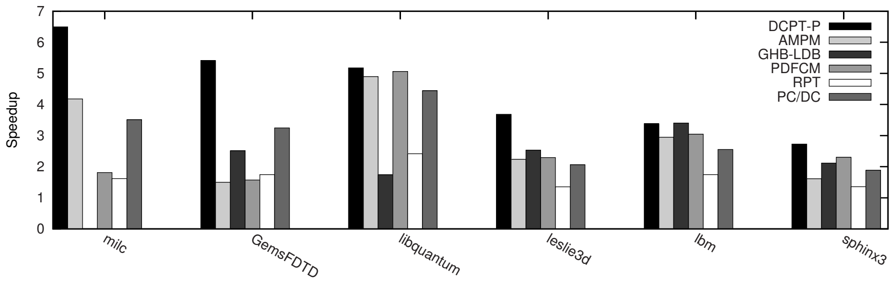

Fig. 9. 512KB L2 cache. Benchmarks with large speedups.

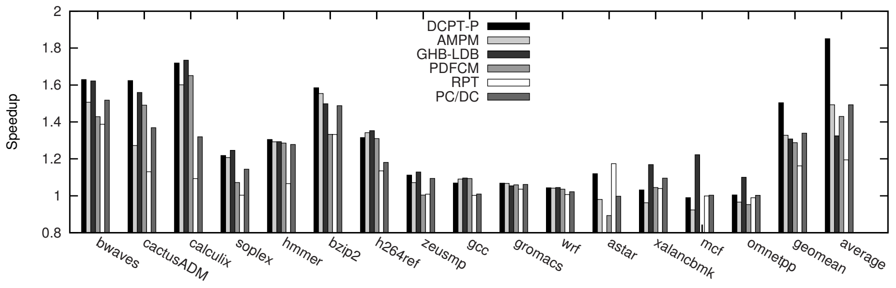

Fig. 10. 512KB L2 cache. Benchmarks with small speedups.

on leslie3d, calculix, bzip2 and h264ref. In this configuration PDFCM causes performance degradation on astar and omnetpp. Surprisingly, RPT prefetching is the best prefetcher on astar. On this benchmark, most of the other prefetchers has very high miss rates, especially when prefetching into the L1 cache. Thus, the more conservative prefetcher performs well. Additionally, the benefits of GHB-LDB on mcf, omnetpp and xalncbmk increases.

Figure 11 and 12 provides insight into the relative performance benefits of the three techniques proposed in this work. Undoubtedly, the basic DCPT design is responsible for most of the performance gain. This is because it is responsible for bridging the last level cache to main memory gap and thus has the most potential. Both Partial matching and L1 hoisting contribute to the overall performance. Interestingly, the effects of the two does not seem to be cumulative, but rather synergistic. For instance, on libquantum, switching off partial matching reduces performance somewhat. Switching off L1 hoisting reduces performance even more, but there is no difference between this configuration and switching both L1 hoisting and partial matching off. On both omnetpp and astar we see that partial matching actually causes a performance degradation. This effect is due to the much lower accuracy of partial matching, which in turn causes bandwidth saturation.

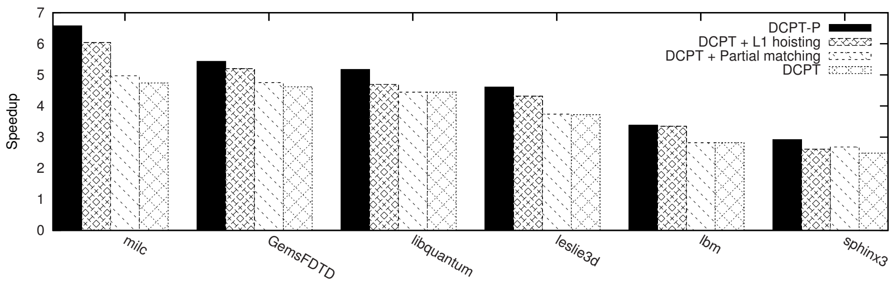

Fig. 11. Breakdown of performance contribution of DCPT-P. Benchmarks with large speedups.

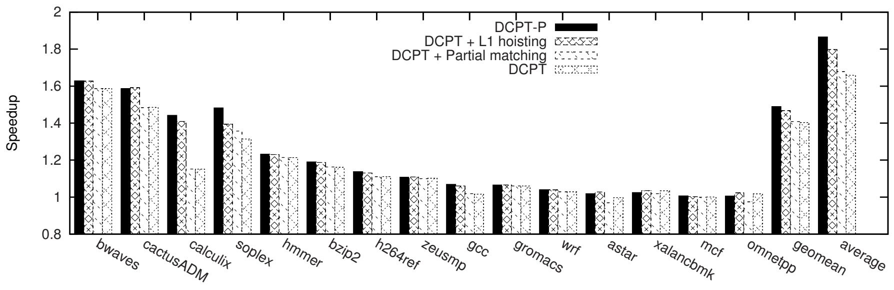

Fig. 12. Breakdown of performance contribution of DCPT-P. Benchmarks with small speedups.

5.1 Area and Performance Trade-Offs

So far, we have focused our attention on performance. However, it is possible to optimize for area as well. The largest structure in DCPT-P is the table holding the entries. In this section, we explore the area and performance trade-off of changing some of the key table parameters. In Figure 13, we show the performance impact of increasing the number of deltas in each entry. The speedups are reported relative to the same case with no prefetching. Although the unlimited bandwidth case has higher absolute performance, the relative speedup of prefetching is lower. Increasing the number of deltas has three distinct effects. Firstly, it increases the probability of a match, thus the number of prefetches increases. Secondly, it increases the effective prefetch distance. Finally, it increases power and area as the number of comparators has to be increased. Although DCPT-P is highly accurate, a large prefetch distance can cause problems, because blocks are fetched too soon. This poses a problem because the blocks may be either evicted before they are used and/or displace other data that is currently needed. This effect can be seen by examining the difference between the 2MB and 512K cases in Figure 13. In the 512K case, performance starts to drop after about 18 deltas, and declines faster than in the 2MB case. Additionally, the

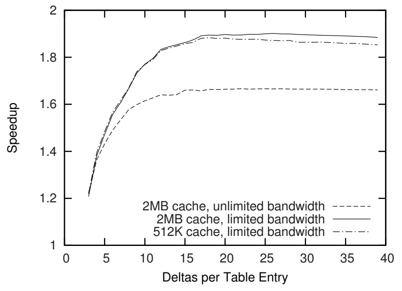

Fig. 13. Average speedup as a function of the number of deltas in each entry

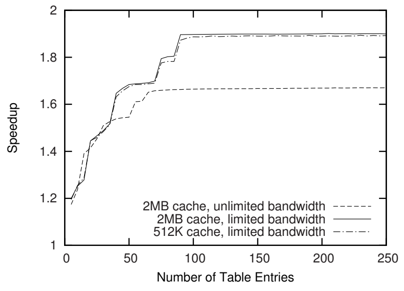

Fig. 14. Average speedup as a function of the number of table entries

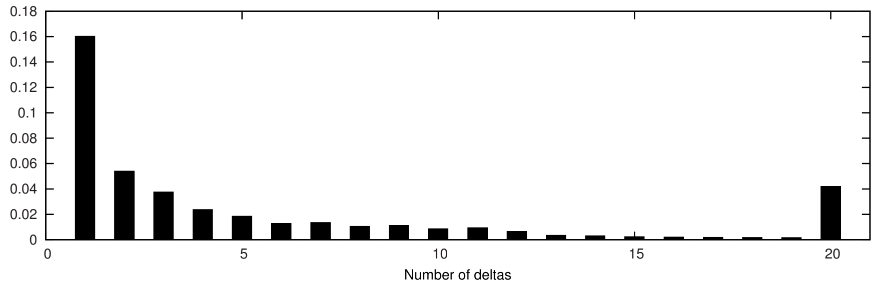

Fig. 15. Distribution of the number of deltas registered in a table entry upon replacement

knee in the graph in the bandwidth unlimited case is shifted to the left compared to the bandwidth limited cases. This suggests that a higher prefetching distance can mask some transient bandwidth contention as well.

In Figure 14, we show the average speedup as a function of the number of entries in the DCPT-P table. Performance increases as the number of entries is increased. After roughly 100 entries there is no performance gain in increasing the size of the table.

6 Discussion

In the design of DCPT-P we have omitted several interesting design ideas, either because they provide little performance benefit or that they will increase the overall complexity of the design and obscure the more central mechanisms in DCPT-P. In this section, we will discuss some of these design options.

In Figure 15, we show the distribution of the number of deltas that has been registered in a entry when it is replaced. DCPT-P requires at least three deltas before it can begin prefetching. As such, the wast majority of table entries are never used for actual prefetching. Thus, much of the table space is wasted on inactive table entries. A possible solution is to use two tables. The first table is a smaller version of the DCPT table, that can hold up to two deltas. If the entry produces more deltas, then that table entry is promoted into the larger table. A second approach is to modify the simple LRU replacement policy in the table to give increased weight to entries with more deltas.

We observed that several of the patterns are simple repeating patterns with a short period. It is possible to capture much of the benefit of DCPT-P by using fewer deltas and analyze the delta pattern to see if it repeats. If it does, then the pattern can be extrapolated. In addition to decreasing the storage requirements by requiring fewer deltas, this approach also gives the possibility of varying the prefetch distance dynamically [15,16].

The pattern matching step is at the core of the DCPT-P heuristic. It is possible to implement this step in a variety of ways depending on the performance and area requirements. Our implementation uses several comparators to examine every possible match location in parallel. To reduce the number of comparators, it is possible to split this step into multiple stages. Consequently, pattern matching can be performed in an iterative fashion by reusing the comparators. As previously shown in Figure 3, the probability of finding a match in the beginning of the delta stream is high. This is because of the prevalence of repeating patterns with short periods. Thus, the probability of finding a match during the first few iterations is high, reducing the average latency.

Another possibility is to limit the search to a subset of the deltas, thus reducing the number of comparators or iterations needed. We investigated limiting the number of deltas searched for a match. As expected, reducing the probability of finding a match decreases overall performance because patterns with long periods are not detected.

Partial matching increases coverage at the cost of decreased prefetcher accuracy. In our implementation we treat prefetches generated by full and partial matching equally. In a more bandwidth-constrained environment it might be beneficial to not treat them equally and only issue prefetches generated by partial matching if there is ample off-chip bandwidth available [17].

Finally, we looked at allowing partial matching to issue multiple prefetches per delta. Because partial matching reduces spatial resolution, the deltas after the match also have reduced resolution. It is possible to compensate for this reduced resolution by issuing multiple prefetches covering the range of possible LSBs. However, because partial matching reduces overall accuracy, we found that issuing multiple prefetches quickly saturated off-chip bandwidth which resulted in reduced performance.

The simulation framework we have opted to use has some limitations. For instance, the look-up time of the predictor is not accounted for. Furthermore, a very simple DRAM model is used, the 4KB storage limit is somewhat arbitrary and techniques which can deal with large off-chip meta-data has been developed [18]. Overall, we chose to use this framework so that a fair comparison with previously proposed prefetchers could be conducted.

7 Conclusion

In this paper, we have presented a novel low-complexity prefetching heuristic called DCPT-P. DCPT-P uses a table indexed by the PC of the load. Each table entry stores a large amount of history per load instruction in the form of deltas. By storing deltas rather than full miss addresses, we save a significant amount of memory and make pattern matching easier. Pattern matching is done by using Delta Correlation, originally proposed by Nesbit et al. [3]. This technique is very effective at detecting patterns with periods shorter than the amount of history stored.

We also introduce the concept of L1 hoisting. L1 hoisting is a technique that combines with DCPT-P to issue highly accurate and timely prefetches into the L1 cache. To deal with several real-world problems with prefetching, we have introduced a mechanism called partial matching which reveals previously hidden patterns by reducing spatial resolution.

Our technique builds upon and expands several ideas presented during the first data prefetching championship (DPC-1). We have examined the top performers extensively and extracted key properties of these prefetchers and improved upon their ideas and synthesised them into a low complexity, storage efficent and high performance prefetcher. By using the code submitted to the DPC-1 contest we can be confident that the comparison with other prefetching techniques is accurate. On average, DCPT-P provides an arithmetic mean speedup of 87% on the SPEC2006 benchmark suite.

References

1. Perez, D.G., Mouchard, G., Temam, O.: Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In: MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 43–54. IEEE Computer Society, Los Alamitos (2004)
2. Grannaes, M., Jahre, M., Natvig, L.: Storage efficient hardware prefetching using delta correlating prediction tables. In: Data Prefetching Championships (2009)
3. Nesbit, K.J., Smith, J.E.: Data cache prefetching using a global history buffer. In: International Symposium on High-Performance Computer Architecture, p. 96 (2004)
4. Smith, A.J.: Cache memories. ACM Comput. Surv. 14(3), 473–530 (1982)
5. Chen, T.F., Baer, J.L.: Effective hardware-based data prefetching for high-performance processors. IEEE Transactions on Computers 44, 609–623 (1995)
6. Dahlgren, F., Stenstrom, P.: Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems 7(4), 385–398 (1996)
7. Nesbit, K.J., Dhodapkar, A.S., Smith, J.E.: AC/DC: An adaptive data cache prefetcher. In: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques, pp. 135–145 (2004)
8. Dimitrov, M., Zhou, H.: Combining local and global history for high performance data prefetching. In: Data Prefetching Championship-1 (2009)

9. Ramos, L.M., Briz, J.L., Ibáñez, P.E., Viñals, V.: Multi-level adaptive prefetching based on performance gradient tracking. In: Data Prefetching Championship-1 (2009)
10. Ishii, Y., Inaba, M., Hiraki, K.: Access map pattern matching prefetch: Optimization friendly method. In: Data Prefetching Championship-1 (2009)
11. Hur, I., Lin, C.: Feedback mechanisms for improving probabilistic memory prefetching. In: HPCA 2009: Proceedings of the 15th International Symposium on High-Performance Computer Architecture, pp. 443–454 (2009)
12. DPC-1: Data prefetching championship rules, `http://www.jilp.org/dpc/framework.html`
13. Jaleel, A., Cohn, R.S., Luk, C.K., Jacob, B.: CMP$im: A pin-based on-the-fly multi-core cache simulator. In: MoBS (2008)
14. SPEC: Spec 2006 benchmark suites (2006), `http://www.spec.org`
15. Srinath, S., Mutlu, O., Kim, H., Patt, Y.N.: Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. Technical report, University of Texas at Austin, TR-HPS-2006-006 (May 2006)
16. Grannaes, M., Natvig, L.: Dynamic parameter tuning for hardware prefetching using shadow tagging. In: CMP-MSI: 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (2008)
17. Grannaes, M., Jahre, M., Natvig, L.: Low-cost open-page prefetch scheduling in chip multiprocessors. In: IEEE International Conference on Computer Design, ICCD (2008)
18. Wenisch, T., Ferdman, M., Ailamaki, A., Falsafi, B., Moshovos, A.: Practical off-chip meta-data for temporal memory streaming. In: High Performance Computer Architecture, HPCA (2009)