Journal of Instruction-Level Parallelism 13 (2011) 1-16 Submitted 3/10; published 1/11

Storage Efficient Hardware Prefetching using Delta-Correlating Prediction Tables

Marius Grannaes GRANNAS@IDI.NTNU.NO
Magnus Jahre JAHRE@IDI.NTNU.NO
Lasse Natvig LASSE@IDI.NTNU.NO
Department of Computer and Information Science
Norwegian University of Science and Technology
Sem Saelandsvei 7-9, 7491 Trondheim, Norway

Abstract

This paper presents a novel prefetching heuristic called Delta Correlating Prediction Tables (DCPT). DCPT builds upon two previously proposed techniques, RPT prefetching by Chen and Baer and PC/DC prefetching by Nesbit and Smith. It combines the storage-efficient table based design of Reference Prediction Tables (RPT) with the high performance delta correlating design of PC/DC. DCPT substantially reduces the complexity of PC/DC prefetching by avoiding expensive pointer chasing in the GHB (Global History Buffer) and recomputation of the delta buffer.

We evaluate this prefetcher on a simulated processor using CMP$im and the SPEC2006 benchmarks. We show that DCPT prefetching can increase performance by up to 3.7X for single benchmarks, while the geometric mean of speedups across all SPEC2006 benchmarks is 42% compared to no prefetching.

1. Introduction

Each year, exponentially more transistors can be put into a single integrated circuit [1, 2]. Moore's law is the empirical observation that the number of transistors that can be placed on an integrated circuit, with respect to minimum component cost, will double every 24 months. Increased transistor density, in turn, translates into faster computers for consumers.

Up until about the year 2002, processor performance increased by about 55% per year [3]. Since then, limitations on power, ILP and memory latency have slowed the increase in uniprocessor performance to about 20% per year. Although the capacity of DRAM increases by about 40% per year, the latency only decreases by about 6-7% per year [4]. This gap between the processor and DRAM leads to a performance problem known as the *"memory wall"* (or *"memory gap"*) [5]. Numerous techniques have been developed to tolerate or compensate for this gap, including out-of-order execution, caches and prefetching.

Prefetching is a technique to reduce the number of misses in a cache through predicting future memory references and fetching the corresponding data before it is referenced by the CPU. It is especially effective for reducing *compulsory* misses, as caches only retain previously referenced data. However, because prefetching is a speculative technique some prefetched data will not be used, which causes cache pollution and increased bandwidth usage. Most prefetching heuristics work by finding patterns in the memory access stream and use this knowledge to predict future accesses.

In this paper, we present a new prefetching heuristic called Delta Correlating Prediction Tables (DCPT). DCPT builds upon two previously proposed prefetcher techniques, combining them and refining their ideas to achieve better performance. This heuristic provides a significant speedup (42% on average for SPEC2006 benchmarks), while only needing 4KB of storage.

2. Previous Work

Many prefetching heuristics have been proposed in the past. The simplest is the *sequential prefetcher* [6], which simply fetches the next block when there is a miss in the cache. An improvement over this simple heuristic is the *tagged sequential prefetcher* which adds an extra bit per cache block (the tag). This bit is set when a block is prefetched into the cache. If there is a cache hit on a block where this bit is set, then the next cache block is fetched.

The prefetching degree is the number of cache blocks that are fetched on a single prefetching operation, while the prefetching distance is how far ahead prefetching starts. A sequential prefetcher with a prefetching degree of 2, and a prefetching distance of 5, would fetch blocks X+5 and X+6 if there was a miss on block X.

Perez, et al. [7] did a comparative survey in 2004 of many proposed prefetching heuristics and found that tagged sequential prefetching, reference prediction tables (RPT) and Program Counter/Delta Correlation Prefetching (PC/DC) were the top performers. DCPT is based on RPT and PC/DC which is covered in detail in section 2.1. and 2.2..

Instruction-based prefetchers have been shown to be very effective [7]. The problem with instruction-based prefetchers is that they require the load address, which either requires that the load-address is transmitted along with the memory request or that the prefetcher is tightly coupled with the processor core. A third option is to not use this information at all and only use the miss address.

Markov Prefetchers uses a 1-history Markov model in order to predict future references [8]. This model uses a graph where each node represents a cache block. Each transition from node X to node Y is assigned a weight representing the fraction of references to X that are followed by a reference to Y. When X is accessed, then the outgoing edges from X are examined. The weighting on the edges can be used to reject or accept prefetching to the node which the edge points to.

Another approach is to detect when there is a high level of spatial locality in the program [9]. When the program has high spatial locality, it is potentially beneficial to fetch more than a single cache block into the cache. One approach for detecting high spatial locality is to have a separate tag-array that mimics a cache with larger cache blocks. If there are more than a set threshold of hits to the same larger virtual cache block, there is a high probability of high spatial locality and a larger block of data can be prefetched into the actual cache [9]. Rather than using a separate tag-array, it is possible to use a smaller table of bit vectors or offsets to represent the same information [10, 11]. By indexing these patterns with the PC of the load, it is possible to use these bit vectors when that load misses and prefetch according to the bitvector.

Taking these ideas further, Spatial Memory Streaming (SMS) uses code correlation across loads [12]. In this approach, an initial trigger access to a spatial region starts recording subsequent accesses to the same spatial region. The blocks that are touched are stored

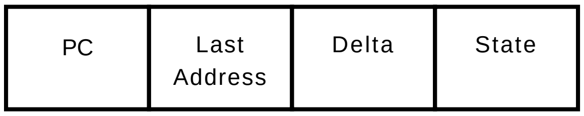

Figure 1: Format of a Reference Prediction Table entry.

in a bitvector representing the spatial region. The recording stops when the first cache block from the spatial region is evicted from the primary cache. This pattern can then be used to prefetch large spatial blocks.

Streams of memory also exhibit temporal locality (i.e., the exact same sequence of addresses are observed in succession) [13]. This observation is exploited in Spatio-temporal memory streaming by storing the observed miss address stream in a circular buffer and using it to detect repeating patterns [13]. This approach is especially useful for programs using shared memory.

2.1. Reference Prediction Tables

Reference Prediction Tables (RPT) is a strided prefetching heuristic originally proposed by Chen and Baer in 1995 [14]. Although improvements to the original design have been proposed [15], the basic design is the same. RPT is related to the earlier Stride Directed Prefetcher (SDP) [16]. SDP used a table storing the PC of each load and the address of the last load. By storing this information, SDP could calculate the stride (delta) between the current and the previous load address. This delta is then used when computing the next address to prefetch. As the name implies, RPT prefetching is a large table indexed by the address of the load which caused the miss. Each table entry has the format shown in Figure 1.

The first time a load instruction causes a miss, a table entry is reserved, possibly evicting the table entry for an older load instruction. The miss address is then recorded in the *last address* field and the *state* is set to initial. The next time this instruction causes a miss, *last address* is subtracted from the current miss address and the result is stored in the *delta* (stride) field. *Last address* is then updated with the new miss address. The entry is now in the training state. The third time the load instruction misses a new delta is computed. If this delta matches the one stored in the entry, then there is a strided access pattern. The prefetcher then uses the delta to calculate which cache block(s) to prefetch.

2.2. PC/DC Prefetching

In 2004, Nesbit and Smith [17] proposed a different approach using a Global History Buffer (GHB). The structure of the GHB is shown in figure 2. Each cache miss or cache hit to a tagged (prefetched) cache block is inserted into the GHB in FIFO order. The index table stores the address of the load instruction and a pointer into the GHB for the last miss issued by that instruction. Each entry in the GHB has a similar pointer, which points to the next miss issued by the same instruction. By traversing the pointers, the history of the latest misses issued by a certain instruction can be obtained.

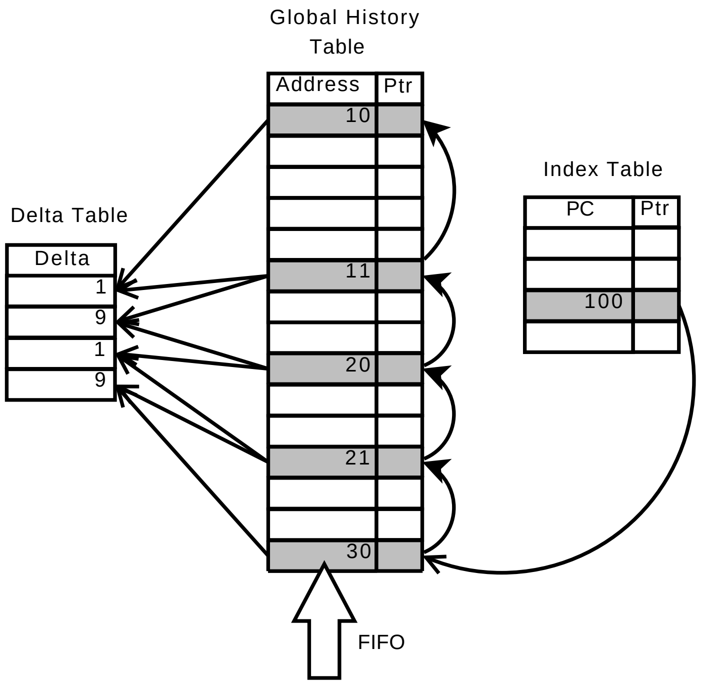

Figure 2: Example of a Global History Buffer.

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

Figure 3: Example delta stream.

PC/DC prefetching calculates the deltas between successive cache misses and stores them in a delta-buffer. The history in Figure 2 yields the address stream and corresponding deltas in Figure 3. The last pair of deltas is (1,9). By searching the delta-stream (correlating), we find this same pair in the beginning. A pattern is found, and prefetching can begin. The deltas after the pair are then added to the current miss address, and prefetches are issued for the calculated addresses.

3. Delta-Correlating Prediction Tables

Our prefetch heuristic combines the approaches of both RPT and PC/DC prefetching by using a table based approach to delta correlation. In DCPT we use a large table indexed by the address (PC) of the load. Each entry has the format shown in Figure 4. The *last address* field works in a similar manner as in RPT prefetching. The n delta fields acts as a circular buffer, holding the last n deltas observed by this load instruction and the *delta pointer* points to the head of this circular buffer.

This main flow is shown in Algorithm 1. In this pseudocode $\leftarrow$ is used as an assignment operator and $\Leftarrow$ is used as the insert into circular buffer operator. To ease notation and in-

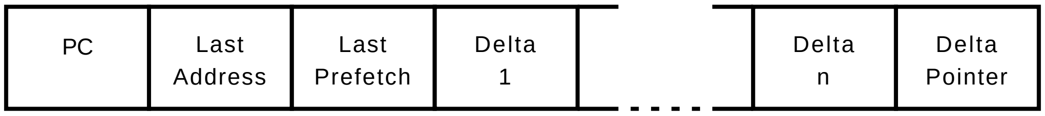

Figure 4: Format of a Delta Correlating Prediction Table Entry.

Algorithm 1 Main flow

```
1: **procedure** DCPT(Load Address $PC$, Miss address $addr$)
2:     $entry \leftarrow$ **TableLookUp**($PC$)
3:
4:     **if** $entry.pc \neq PC$ **then**
5:         $entry.pc \leftarrow PC$
6:         $entry.lastAddress \leftarrow addr$
7:         $entry.deltas[\,] \leftarrow \emptyset$
8:         $entry.lastPrefetch \leftarrow 0$
9:     **else if** $addr$ - $entry.lastAddress \neq 0$ **then**
10:        $entry.deltas[\,] \Leftarrow addr$ - $entry.lastAddress$
11:        $entry.lastAddress \leftarrow addr$
12:
13:        $candidates[\,] \leftarrow$ **DeltaCorrelation**($entry$)
14:        $prefetches[\,] \leftarrow$ **PrefetchFilter**($entry$, $candidates[\,]$)
15:        **IssuePrefetches**($prefetches[\,]$)
```

crease readability the delta buffer and the inFlight buffer is shown as pure arrays. However, both are implemented as circular buffers and if they are full, the oldest entry is discarded.

Initially, the PC is used to look up into a table of entries. In our implementation we have used a fully-associative table, but it is possible to use other organizations as well. If an entry with the corresponding PC is not found, then a replacement entry is initialized. This is shown in lines 4-8. If an entry is found, the delta between the current address and the previous address is computed. The buffer is only updated if the delta is non-zero. The new delta is inserted into the delta buffer and the last address field is updated. Each delta is stored as a n bit value. If the value cannot be represented with only n bits, a 0 is stored in the delta buffer as an indicator of an overflow error.

Delta correlation begins after updating the entry. The pseudocode for delta correlation is shown in Algorithm 2. The deltas are traversed in reverse order, looking for a match to the two most recently inserted deltas. If a match is found, the next stage begins. The first prefetch candidate is generated by adding the delta after the match to the value found in *last address*. The next prefetch candidate is generated by adding the next delta to the previous prefetch candidate. This process is repeated for each of the deltas after the matched pair including the newly inserted deltas.

In the example in Table 3, the last pair of deltas is (1,9). Searching from the left, we find this pattern at the beginning. The first delta after the pattern is 1. This delta is then added to the last address (30), producing a prefetch request for address 31. The next delta is 9. Adding 9 to 31 yields 40, producing a prefetch request for address 40.

Algorithm 2 Delta Correlation

```
 1: procedure DeltaCorrelation(DCPT entry $entry$)
 2:     $candidates[\,] \leftarrow \emptyset$
 3:     $d1 \leftarrow entry.deltas[last]$
 4:     $d2 \leftarrow entry.deltas[last - 1]$
 5:     $address \leftarrow entry.lastAddress$
 6:
 7:     for each pair $u$, $v$ in $entry.deltas[\,]$ do
 8:         if $u = d2$ and $v = d1$ then
 9:             for each $delta$ remaining in $entry.deltas[\,]$ do
10:                 $address \leftarrow address + delta$
11:                 $candidates[\,] \Leftarrow address$
12:     Return $candidates[\,]$
```

Algorithm 3 Prefetch Filtering

```
 1: procedure PrefetchFilter(DCPT entry $entry$, $candidates[\,]$)
 2:     $prefetches[\,] \leftarrow \emptyset$
 3:     for each $candidate$ in $candidates[\,]$ do
 4:         if $candidate$ not in $inFlight[\,] \cup MSHRs \cup Cache$ then
 5:             $prefetches \leftarrow candidate$
 6:             $inFlight[\,] \Leftarrow candidate$
 7:             $entry.lastPrefetch \leftarrow candidate$
 8:         if $candidate = entry.lastPrefetch$ then
 9:             $prefetches[\,] \leftarrow \emptyset$
10:     Return $prefetches[\,]$
```

The next step in the DCPT flow is prefetch filtering. The pseudocode for this step is shown in Algorithm 3. If a prefetch candidate matches the value stored in *last prefetch*, the content of the prefetch candidate buffer up to this point is discarded. Every prefetch candidate is looked up in the cache to see if it is already present. If it is not present, it is checked against the miss status holding registers to see if a demand request for the same block has already been issued. Third, the candidate is checked against a buffer that holds other prefetch requests that have not been completed. This buffer can only hold 32 prefetches. If it is full, then prefetch is discarded in FIFO order. Finally, the *last prefetch* field is updated with the address of the issued prefetch.

4. Methodology

To evaluate the performance of our prefetcher, we have used the SPEC2006 [18] benchmarks with the CMP$im simulator [19]. Each benchmark was fast forwarded 40 billion instructions and then a memory trace of the next 100 million instructions was recorded. The simulated processor is a 15 stage, 4-wide OoO processor with a 128 entry instruction window with perfect branch prediction [20]. A maximum of two loads and one store can be issued per

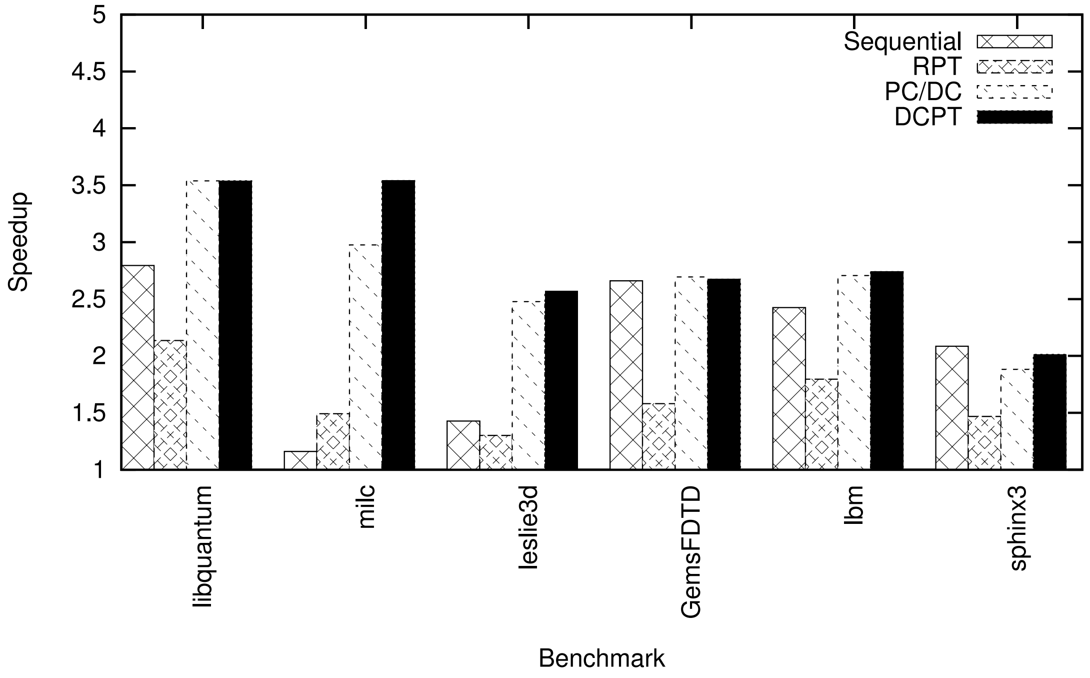

Figure 5: Speedup compared to no prefetching for benchmarks with high speedups on a 2MB cache with unlimited bandwidth.

clock cycle. The L1 is a 32KB 8-way set associative cache with a latency of 1 cycle. In this paper we use either a 512KB or a 2MB L2 cache, both 16 way set associative with a 20 cycle latency. Main memory has a 200 cycle latency.

The tagged sequential prefetcher was configured with a prefetching degree of 5, and a distance of 4. The RPT prefetcher has a 256 entry table, a prefetching degree of 16 and a distance of 4. To keep within the 32 Kbit limit set by the competition [20], the PC/DC prefetcher has a 702 entry GHB and a 32 entry delta buffer. Our prefetcher was set up with a 98 entry table with 19 12-bit deltas. These parameters were found experimentally to maximize performance on each prefetcher.

5. Results

In Figures 5 and 6, we compare the performance of the 4 prefetchers relative to no prefetching in a system with unlimited bandwidth. Prefetching has very little impact on performance ($< 2\%$) for the benchmarks *perlbench*, *gcc*, *gobmk*, *sjeng*, *gamess*, *namd*, *dealII*, *povray* and *tonto* and are not shown to conserve space. However, they are included in computed the geometric mean for the benchmark suite. Furthermore, the results have been split into two graphs so that the benchmarks showing large speedups do not dwarf the others and the geometric mean of speedups.

Although DCPT and PC/DC prefetching share the same underlying pattern recognition engine, DCPT is able to capture more of the potential due to a more space-efficient implementation. Because there is no penalty for issuing several prefetches, sequential prefetching

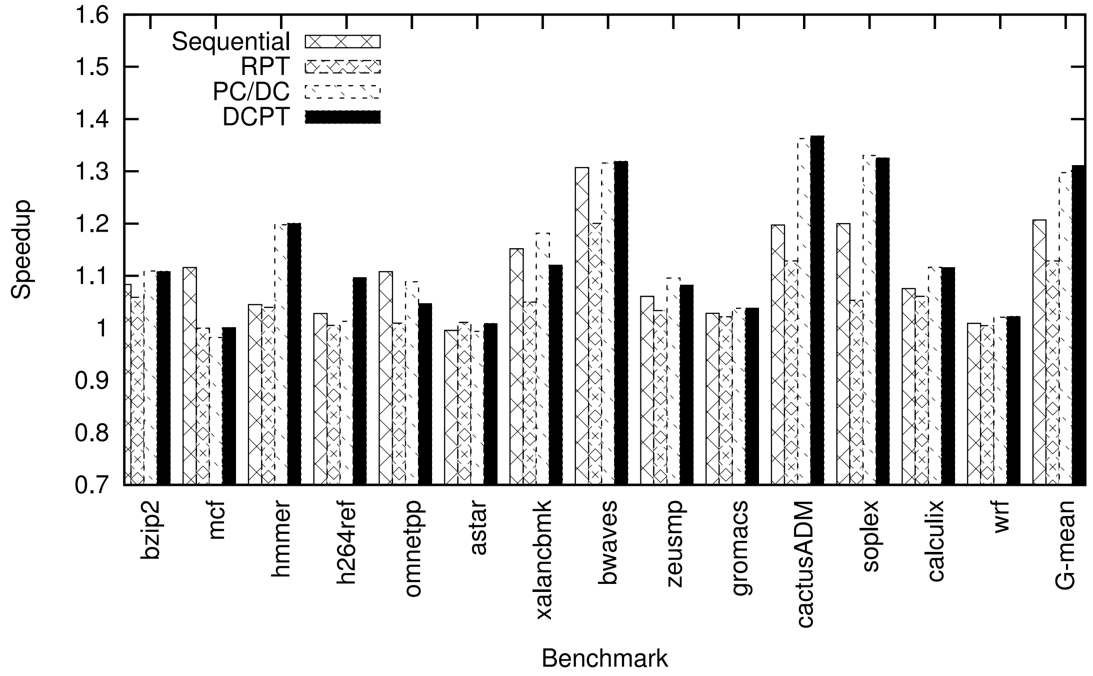

Figure 6: Speedup compared to no prefetching for benchmarks with low speedups on a 2MB cache with unlimited bandwidth.

performs quite well on several benchmarks but is unable to capture the patterns observed in *milc* and *leslie3d*. Overall, DCPT prefetching acheives a geometric mean of speedups of 1.31, while PC/DC achieves 1.29.

Our second experiment, shown in Figures 7 and 8, limits the bandwidth to one request per 10 clock cycles. In this configuration there is a more significant performance difference between DCPT (1.38 geometric mean speedup) and PC/DC (1.32 geometric mean speedup), even though there are several benchmarks where prefetching has no effect. Again there is a marked difference between DCPT and PC/DC in both *milc* and *leslie3d*.

In Figures 9 and 10 we reduce the size of the L2 cache to 512KB. In this case, sequential prefetching causes a severe slowdown on *mcf* and *astar*. However, it is also the top performer on *GemsFDTD*. Again, DCPT outperforms the other prefetchers. Surprisingly, RPT prefetching does not perform very well. This is mainly due to it being too conservative with respect to bandwidth and at the same time not being able to detect the same access patterns as PC/DC and DCPT. In this configuration, DCPT achieves a geometric mean speedup of 1.42 vs 1.33 for PC/DC.

5.1. DCPT Parameters

One of the main differences between DCPT and PC/DC is that DCPT stores deltas, while PC/DC stores entire addresses in its GHB. Because the deltas are usually quite small, fewer bits are needed to represent a delta than a full address. In Figure 11, we show the average portion of deltas that can be represented with a given amount of bits across

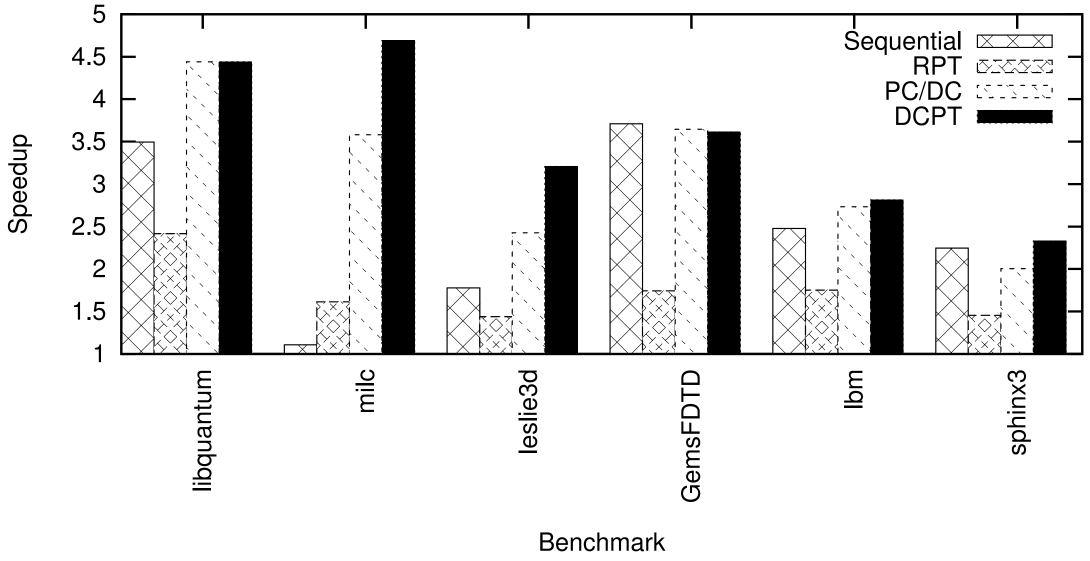

Figure 7: Speedup compared to no prefetching for benchmarks with high speedups on a 2MB cache with limited bandwidth.

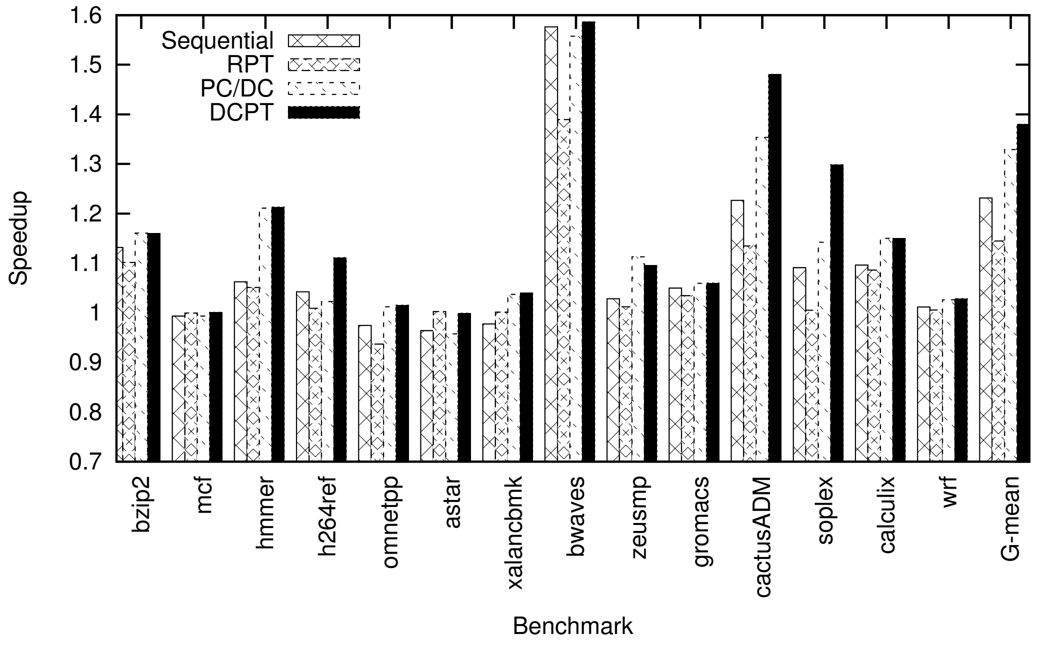

Figure 8: Speedup compared to no prefetching for benchmarks with low speedups on a 2 MB cache with limited bandwidth.

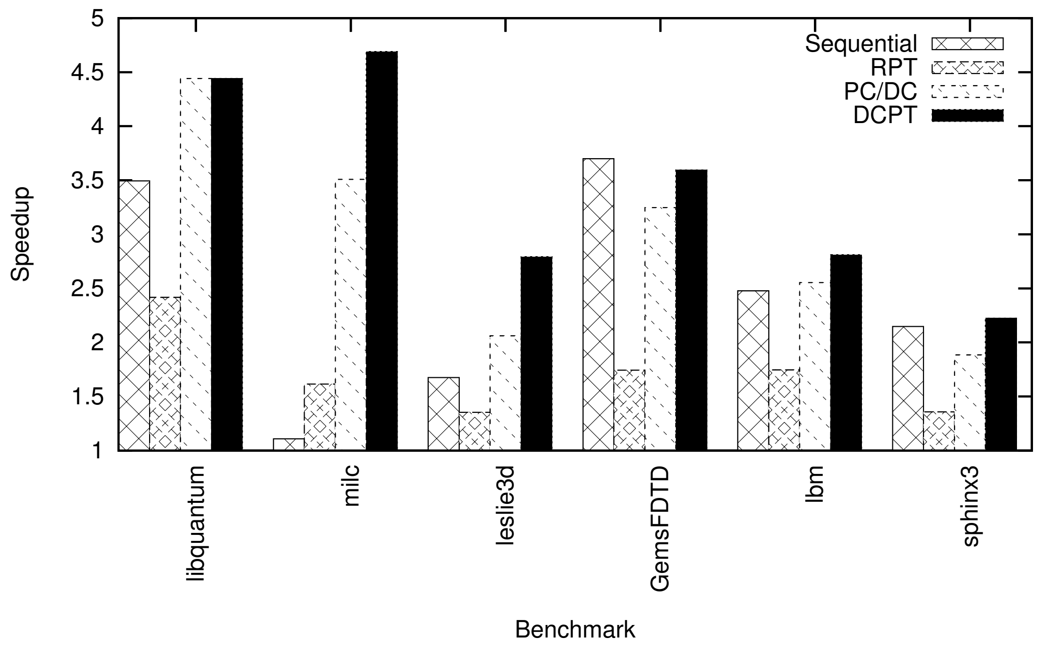

Figure 9: Speedup compared to no prefetching for benchmarks with high speedups on a 512KB cache with limited bandwidth.

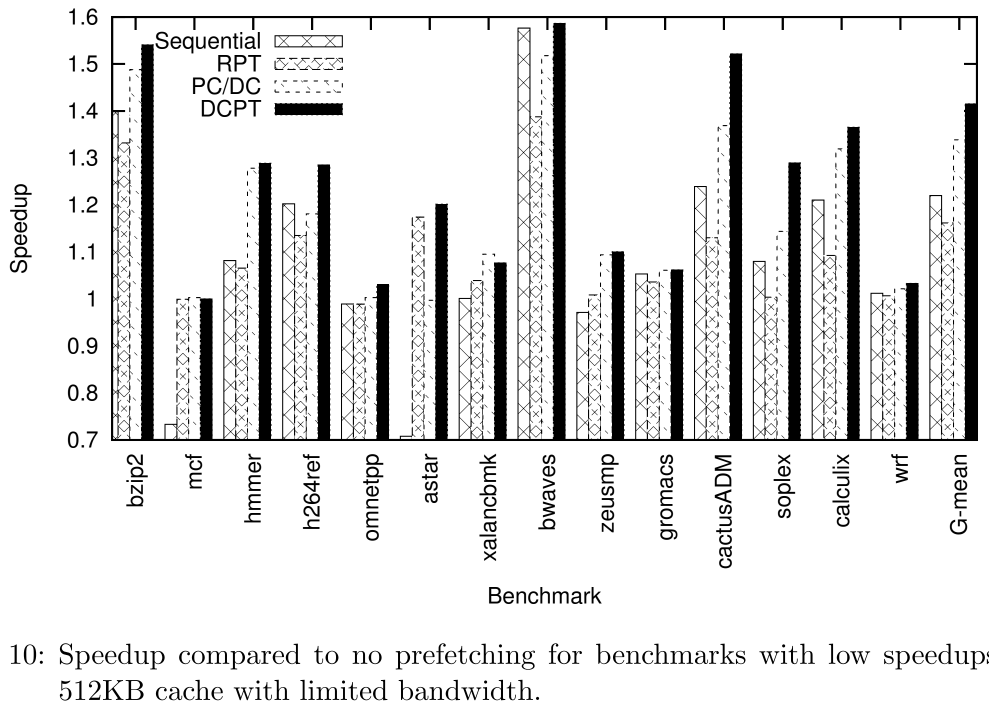

Figure 10: Speedup compared to no prefetching for benchmarks with low speedups on a 512KB cache with limited bandwidth.

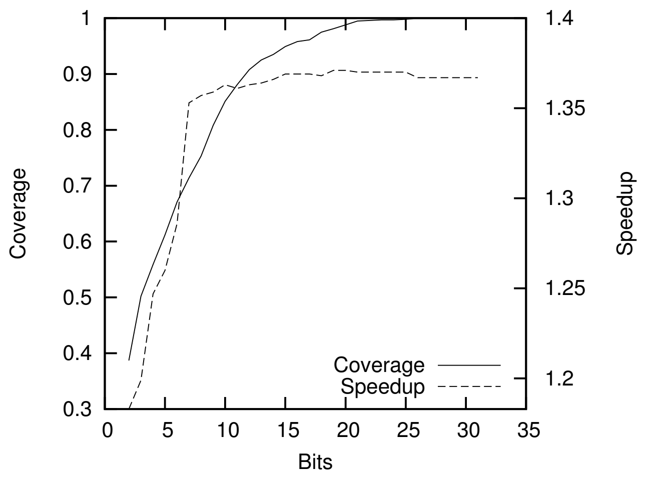

Figure 11: Coverage and speedup as a function of the number of bits used to represent a delta.

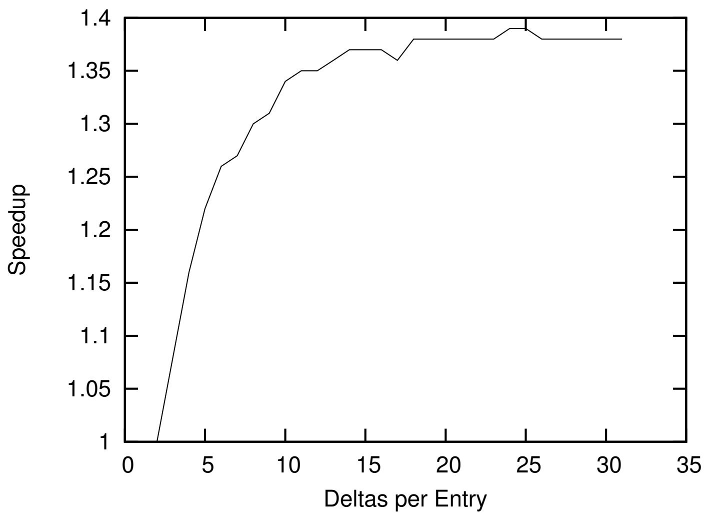

Figure 12: Speedup vs. the number of deltas per entry.

all SPEC2006 benchmarks. Additionally, the geometric mean of speedups is plotted as a function of the number of bits used per delta. In this experiment, we have used a 256 entry DCPT prefetcher with 16 deltas per entry. Although the coverage steadily increases with the amount of bits used, speedup has a distinct knee at around 7 bits. Thus, high deltas are not useful for prefetching.

In Figure 12 we show the geometric mean of speedups as a function of the number of deltas per table entry. In this experiment, we used 16 bits deltas and 256 table entries. In effect, increasing the number of deltas increases the prefetch distance of DCPT. Thus, the optimal choice will be both processor and program dependant. However, there is a clear trend that performance flattens after about 14 deltas per table entry.

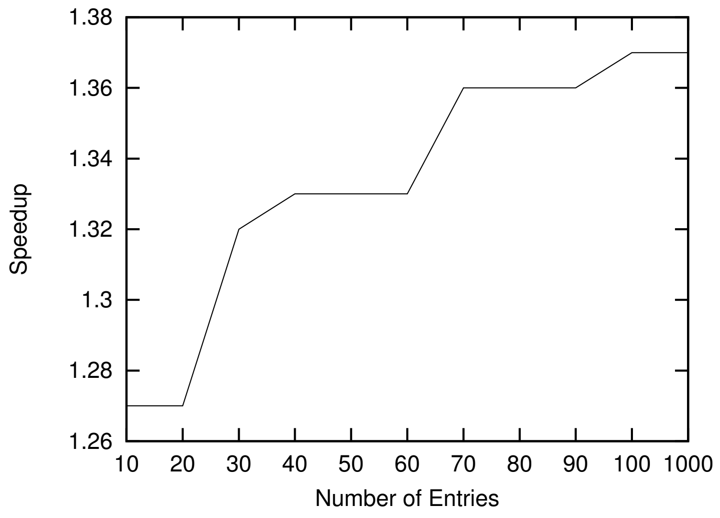

Figure 13: Speedup vs table size.

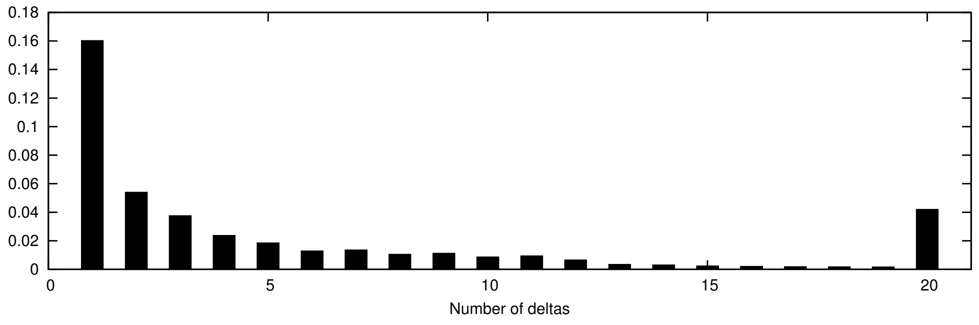

Figure 14: Number of deltas stored in entry at replacement.

Figure 13 shows the geometric mean of speedups as a function of the number of table entries. There is a steady performance improvement up to about 100 table entries. After this point, there is virtually no gain in adding extra entries.

Finally, we examined the average number of deltas stored in each entry at replacement. Our results are shown in Figure 14. Many of the entries are only used to store the initial entry and then discarded. If these could be detected, the number of entries required would be reduced.

6. Discussion

Our proposed prefetching technique is storage efficient. However, the complexity of calculating each prefetch is high. Each calculation involves searching the entire length of the deltas for possible matches and then adding the remaining deltas. Thus, each calculation has a fixed latency as each delta is either used in a comparison or an addition which lends itself well to pipelining. However, as shown in Figure 15 in most cases the first pair of deltas

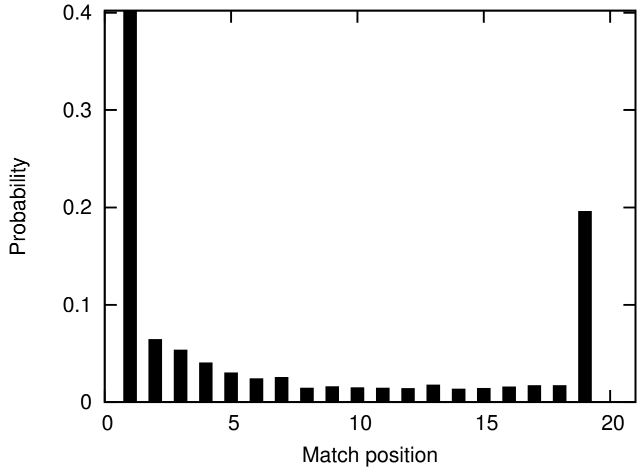

Figure 15: Number of entries examined before a match is found in delta correlation. Note that the peak at 19 represents misses.

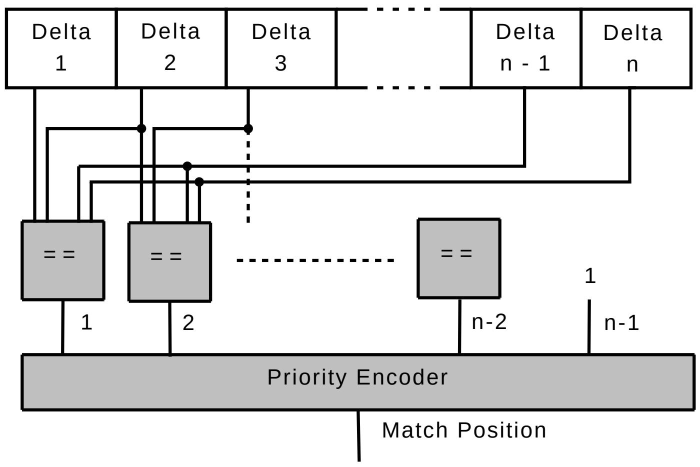

Figure 16: Parallel implementation of the pattern matching step of DCPT.

examined is a match. This is due to the prevalence of simple strided or sequential patterns. Note that the peak at 19 are misses.

Because of the relative infrequency of L2 misses, several design points are available depending on the needed performance and available power and area. At one end of the spectrum, a single comparator and a single adder is sufficient to implement DCPT in addition to the memory storage. At the other end, the pattern matching step can be performed in a single cycle, provided enough comparators as shown in Figure 16.

In all our experiments, we unrealistically assumed that the calculation would not take any time to perform. We experimented with increasing the delay of the calculation from 1 to 100 clock cycles. Although 100 clock cycles is an unrealistic delay, it nevertheless indicates the maximum impact of delay. Our results are shown in Figure 17. It shows that there was only a minor ($< 1\%$) performance impact of additional delay. However, this penalty can

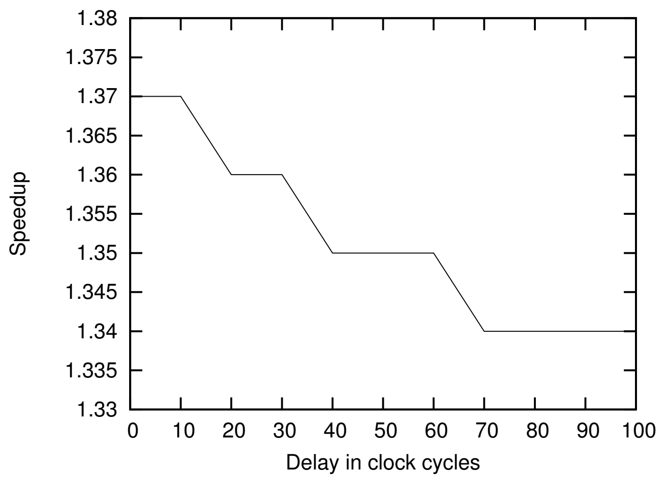

Figure 17: Speedup vs calculation delay.

be offset by increasing the number of deltas per entry - indirectly increasing the prefetch distance and thus timeliness.

In most cases, the patterns observed are quite simple, as they often repeat themselves after only a few deltas (see Figure 15). We did some initial experimentation with storing fewer deltas per entry and extrapolate the pattern from those deltas. However, there was little to gain from this technique, and we chose to eliminate it from the final design to keep it simpler. Furthermore, because most memory access patterns are relatively stable, the last prefetch candidate is often the only one that is not filtered out by the *last prefetch* entry. This observation can be exploited by only calculating the last possible prefetch candidate.

In our experiments, we used n bits to represent the delta range, representing the values between 2^{n-1} and -2^{n-1}. We observed more positive deltas than negative deltas, leading us to believe that adding a bias to the delta might be beneficial. We did not explore this further as the maximum potential of this technique would be equal to adding a single extra bit to each delta.

7. Conclusion

In this paper, we have presented a new prefetching heuristic called Delta Correlating Prediction Tables (DCPT). DCPT builds upon two previously proposed techniques, Reference Prediction Tables by Chen and Baer [14] and PC/DC prefetching by Nesbit and Smith [17]. It combines the table based design of RPT and the delta correlating design of PC/DC with some improvements.

We show that DCPT prefetching can increase performance by up to 3.7X, while the geometric mean of speedups across all benchmarks is 42%. This is an improvement over PC/DC prefetching by 27.2% in the 512KB cache configuration with limited bandwidth.

Acknowledgments

This work was supported by the Norwegian Metacenter for Computational Science (Notur). Lasse Natvig is a member of HiPEAC2 NoE.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, Apr. 1965.

[2] International Technology Roadmap for Semiconductors, "ITRS roadmap," 2007. `http://www.itrs.net/Links/2007ITRS/Home2007.htm`.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition.* Morgan Kaufmann Publishers, 2007.

[4] D. A. Patterson, "Latency lags bandwith," *Commun. ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[5] W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," *ACM Computer Architecture New*, vol. 23, march 1995.

[6] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.

[7] D. G. Perez, G. Mouchard, and O. Temam, "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, 2004.

[8] D. Joseph, "Prefetching using markov predictors," in *The 24th Annual International Symposium on Computer Architecture*, pp. 252–263, 1997.

[9] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu, "Run-time spatial locality detection and optimization," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 57–64, 1997.

[10] C. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *High Performance Computer Architecture, HPCA-10. Proceedings. 10th International Symposium on*, pp. 276–287, Feb. 2004.

[11] D. M. Koppelman, "Neighborhood prefetching on multiprocessors using instruction history," *Parallel Architectures and Compilation Techniques, International Conference on*, 2000.

[12] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pp. 252–263, 2006.

[13] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 69–80, 2009.

[14] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, pp. 609–623, May 1995.

[15] F. Dahlgren and P. Stenstrom, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, pp. 385–398, Apr. 1996.

[16] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pp. 102–110, 1992.

[17] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *High-Performance Computer Architecture, International Symposium on*, vol. 0, p. 96, 2004.

[18] SPEC, "Spec 2006 benchmark suites," 2006. `http://www.spec.org`.

[19] A. Jaleel, R. S. Cohn, C. K. Luk, and B. Jacob, "CMP$im: A pin-based on-the-fly multi-core cache simulator," in *MoBS*, 2008.

[20] DPC-1, "Data prefetching championship rules."