

Storage Efficient Hardware Prefetching using Delta Correlating Prediction Tables

Marius Grannaes Magnus Jahre Lasse Natvig
Norwegian University of Science and Technology
HiPEAC European Network of Excellence
(grannas—jahre—lasse)@idi.ntnu.no

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 et al. 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 and recomputation of the delta buffer.

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.

I. INTRODUCTION

The performance of general purpose microprocessors continue to increase at a rapid pace, but main memory has not been able to keep up [1]. In essence, the processor is able to process several orders of magnitude more data than main memory is able to deliver on time. Numerous techniques have been developed to tolerate or compensate for this gap, including out-of-order execution, caches and prefetching.

Prefetching predicts what data the processor will need in the future, and fetch that data from main memory before it is referenced. 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 the ideas to achieve better performance. This heuristic provides a significant speedup (42% on average), while only needing 4KB of storage.

II. PREVIOUS WORK

Many prefetching heuristics have been proposed in the past. The simplest is the *sequential prefetcher* [2], 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 line (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 cacheline is fetched.

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

Perez et al. [3] 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.

A. Reference Prediction Tables

Reference Prediction Tables is a strided prefetching heuristic originally proposed by Chen and Baer in 1995 [4]. Although improvements to the original design have been proposed [5], the basic design is the same.

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.

| | | | |
|----|--------------|-------|-------|
| PC | Last Address | Delta | State |
|----|--------------|-------|-------|

Fig. 1: Format of a Reference Prediction Table entry.

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.

B. PC/DC Prefetching

In 2004, Nesbit et al. [6] 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 to 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.

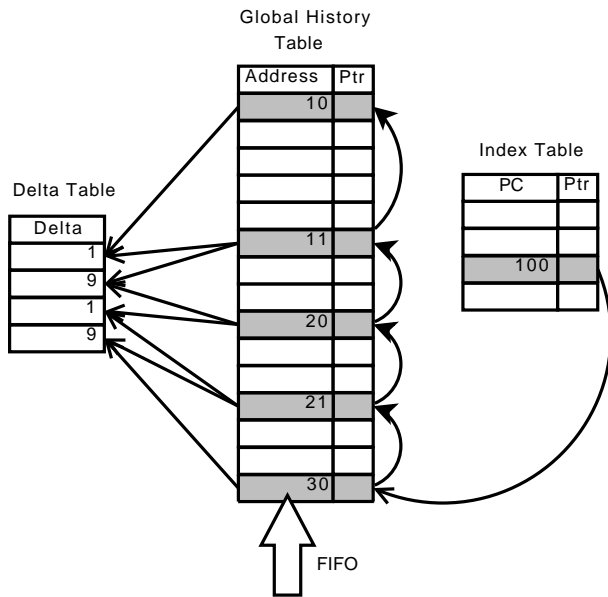


Fig. 2: Example of a Global History Buffer.

By traversing the pointers, the history of the latest misses issued by a certain instruction can be obtained.

PC/DC prefetching calculates the deltas between successive cache misses and stores them in a delta-buffer. The history in figure 2 yields the following address stream and corresponding deltas:

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

TABLE I: Example delta stream.

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.

III. 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 3.

| | | | | | | |
|----|--------------|---------------|---------|-----|---------|---------------|
| PC | Last Address | Last Prefetch | Delta 1 | ... | Delta n | Delta Pointer |
|----|--------------|---------------|---------|-----|---------|---------------|

Fig. 3: Format of a Delta Correlating Prediction Table Entry.

The *last address* field works in a similar manner as in RPT prefetching. Each delta is initially set to 0 and the delta pointer points to the first delta. 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. The buffer is only updated if the delta is non-zero. 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.

After updating the circular buffer, 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. These candidates are stored in a temporary buffer. This process is repeated for each of the deltas after the matched pair including the newly inserted deltas. If a prefetch candidate matches the value stored in *last prefetch*, the content of the prefetch candidate buffer up to this point is discarded.

In the example in table I 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.

After computing the prefetch candidate buffer, every prefetch candidate is looked up in the cache to see if it is already present. If it is not present, then it is checked against the miss status holding registers to see if a demand request for the same line has already been issued. Third, the candidate is checked against a buffer that holds other prefetch request that have not been completed. This buffer can only hold 32 prefetches, if it is full, then the prefetch is discarded. Finally, the *last prefetch* field is updated with the address of the issued prefetch.

IV. METHODOLOGY

To evaluate the performance of our prefetcher, we have used the SPEC2006 [7] benchmarks with the CMPsim simulator [8]. 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 in accordance with competition rules [9]. A maximum of two loads and one store can be issued per 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 [9], 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.

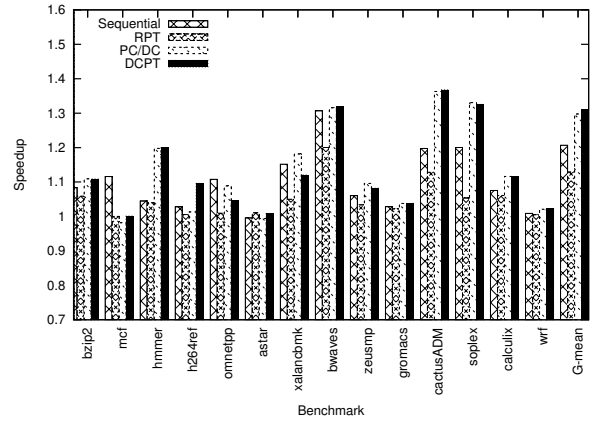
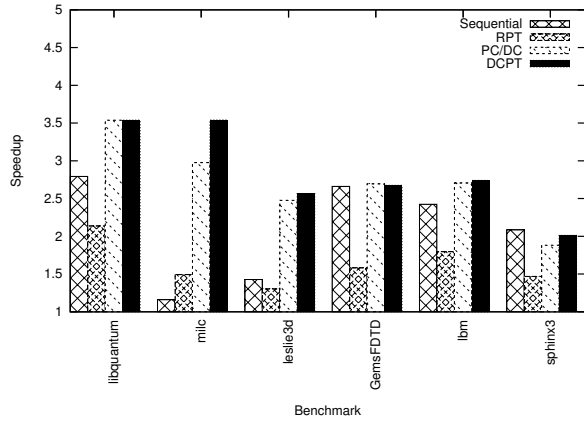


Fig. 4: Speedup compared to no prefetching. 2 MB L2 cache with unlimited bandwidth.

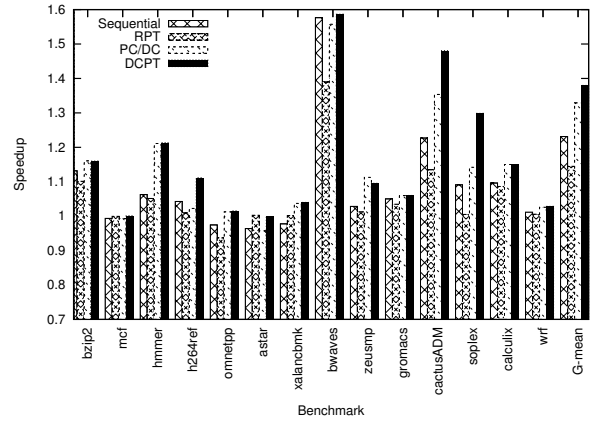
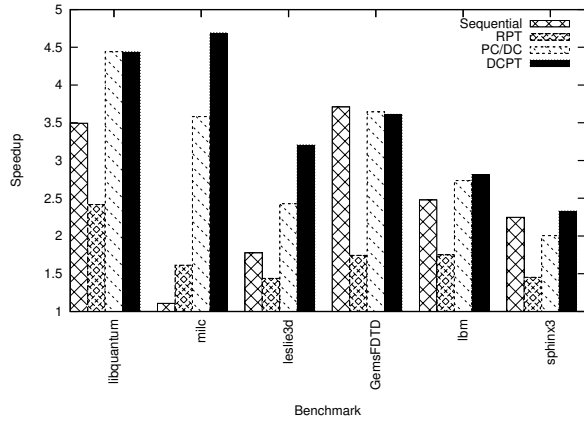


Fig. 5: Speedup compared to no prefetching. 2 MB L2 cache with limited bandwidth.

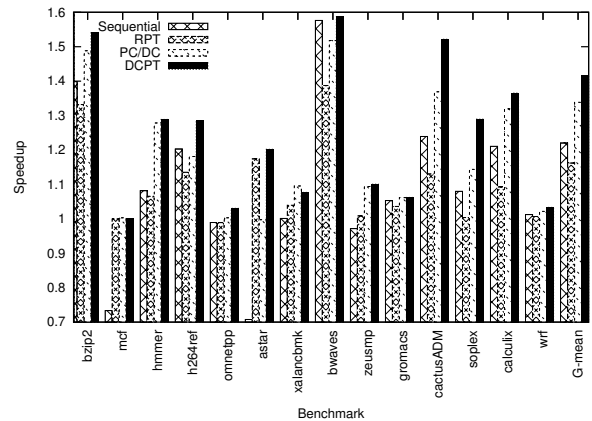
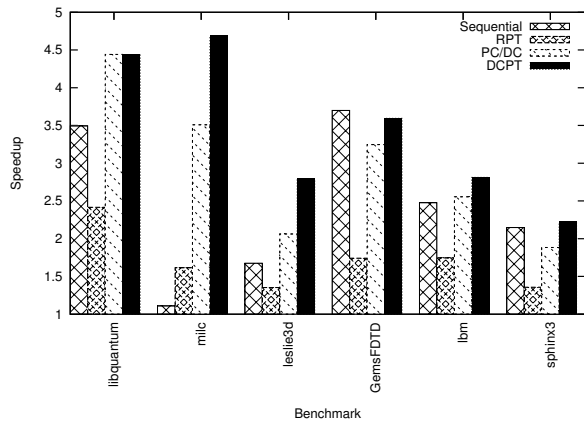


Fig. 6: Speedup compared to no prefetching. 512KB L2 cache with limited bandwidth.

V. RESULTS

In figure 4, 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*, *dealll*, *povray* and *tonto* and are not shown to conserve space. Furthermore, the results have been split into two graphs so that the benchmarks showing large speedups does 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 performs quite well on several benchmarks but is unable to capture the patterns observed in *milc* and *leslie3d*. Overall, DCPT prefetching achieves a geometric mean of speedups of 1.31, while PC/DC achieves 1.29.

Our second experiment, shown in figure 5, 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 figure 6 we reduce the size of the L2 cache to 512KB. In this case, sequential prefetching causes a severe slowdown on *mcfl* 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.

A. 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 7 we show the average portion of deltas that can be represented with a given amount of bits across 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 8 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

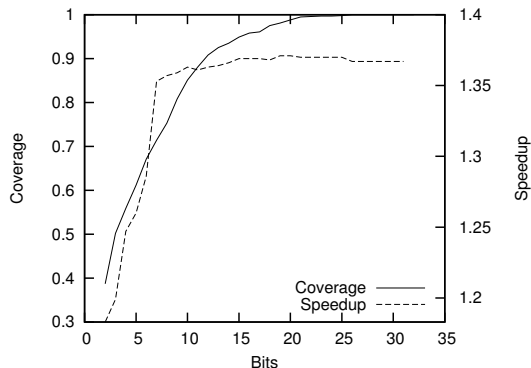


Fig. 7: Coverage and speedup as a function of the number of bits used to represent a delta.

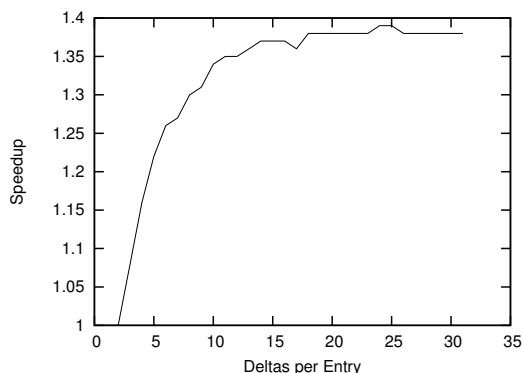


Fig. 8: Speedup vs. the number of deltas per entry.

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.

Finally, in figure 9 we show 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.

VI. DISCUSSION

Our proposed prefetching technique is quite memory 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.

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

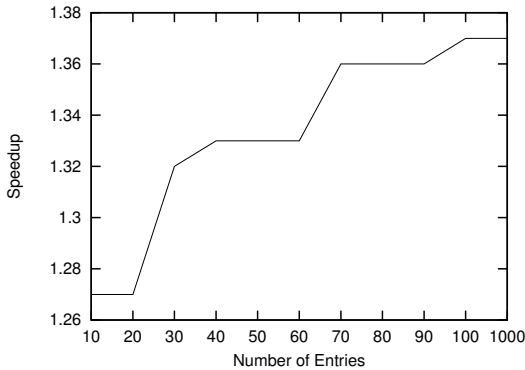


Fig. 9: Speedup vs table size.

in a single cycle, provided enough comparators, while the additions can be performed by several pipelined adders.

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, and found there was only a minor ($< 1\%$) performance impact in the same configuration. However, this penalty can 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. 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.

VII. 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 [4] and PC/DC prefetching by Nesbit et al. [6]. It combines the table based design of RPT and the delta correlating design of PC/DC, as well as improving upon the ideas.

We show that DCPT prefetching can increase performance by up to 3.7X, while the average speedup across all benchmarks is 42%. This is an improvement over PC/DC prefetching by 27.2%.

REFERENCES

- [1] D. A. Patterson, "Latency lags bandwidth," *Commun. ACM*, vol. 47, no. 10, pp. 71–75, 2004.
- [2] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, 1982.
- [3] 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: IEEE Computer Society, 2004, pp. 43–54.
- [4] 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.
- [5] 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, no. 4, pp. 385–398, Apr. 1996.
- [6] 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.
- [7] SPEC, "Spec 2006 benchmark suites," 2006, <http://www.spec.org>.
- [8] 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.
- [9] DPC-1, "Data prefetching championship rules." [Online]. Available: <http://www.jilp.org/dpc/framework.html>