# TDT4260 Project Report

Christopher Benjamin Westlye, Kaj Palm, Raymond Selvik, and Trond Einar Snekvik

***Abstract*—**

## I. Introduction

One of the core problems in computer performance development is the growing processor - memory gap. While the processor speed has been increasing rapidly through the past 30 years, the memory access speed has been lagging behind. The major issue with memory access speed is related to the principle of locality. In order to have a lot of memory, you have to space it out, and traversing this space takes time. The prefetcher is an attempt to minimize the gap by fetching memory from RAM to cache, memory on the chip, but because of space requirements and limits to complexity, the prefetchers are still limited and subject to a lot of different algorithms and optimizations.

The goal of this project is to make a prefetcher that increases runtime performance by determining what memory block will be accessed before it's needed. Based on the DCPT algorithm described in [1], an attempt to increase performance on a collection of benchmarks running on a modified M5 hardware simulator.

## II. Backgroud

The prefetcher is based on the report on itslearning named Managing Shared Resources in Chip Multiprocessor Memory Systems by Magnus Jahre. In order to predict =======

THE performance of a modern day microprocessor is much higher than that of typical memory. Much of the computational time is thus used to access the memory of the RAM and load it into the CPU. This is a growing bottleneck in a time where microprocessors are still increasing in performance.

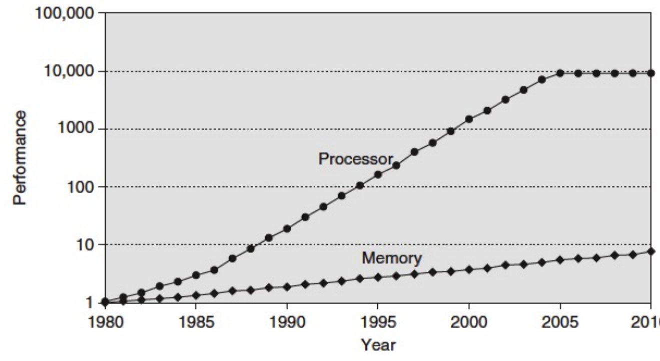

Fig. 1. Dette er ei tekning

A prefetcher reduces this bottleneck by predicting which instructions are addressed next. Memory fetches are attempted to be done before the memory is needed by the microprocessor, leading to a decreased time where the processor is stuck in a waiting state. If the prefetched memory addresses differ from what was needed, the processor needs to access the RAM anyway. This is the worst case scenario, in which the cache has no effect on performance.

## III. Related Work

The most basic prefetcher is called a sequential prefetcher. When a CPU is requesting a memory adress, the prefetcher will also fetch the memory location. The performance of this prefetcher is very limtited since it does not have any form of heuristic. An improvement of this prefetcher is called tagged sequential prefetcher, which divides the sequences into blocks.

What these have in common, is that the prefetching is not based on statistics. They are instead deterministic fetches where a fixed group or single memory adress is fetched. To improve the performance, prefetching based on statistics were developed. These prefetchers will predict the next fetch statistically.

A Reference Prediction Tables RPT is a prefetcher that is based on statistics. This type of prefetcher was first proposed in 1995 by Chan and Baer. A RPT contains a table with addresses of the missed fetches as index. Another variant of this is PC/DC prefetching.

An another type of prefetching is called Delta Correlating Prediction, as described in [1]. Their purpose was to combine the delta correlating design of PC/DC and the storage efficiency of RPT. Compared to RC/DC, the DCPT reduces the complexity of the prefetching. This is done by recomputing the delta.

## IV. Prefetcher Description

The prefetcher is an attempt to improve the Delta-Correlating Prediction Tables (DCPT) approach by Granaes, Jahre and Natvig. The original DCPT algorithm is described in the ”Background” section. The main weakness of this approach is (arguably) that it only bases the prefetch address on the first similar delta pattern it finds. This makes the prefetcher extremely vulnerable to alternating patterns and irregularities in general. If the access pattern makes a sudden leap in an otherwise regular pattern, not only will the prefetcher miss the irregular access, it is guaranteed to miss the next access after that when the pattern goes back to normal. This example is illustrated in Table I. Access number 6 breaks the pattern, and misses. Because the next fetch is based on access 6's delta, this also misses. This problem could be solved by a more democratic approach, where the most common ”next delta” is used instead of the previous.

TABLE I

| Access | Address | Delta | Fetch Issued | Previous Result |
|---|---|---|---|---|
| 1 | ... | ... | ... | ... |
| 2 | 1000 | 10 | 1010 | Hit |
| 3 | 1010 | 10 | 1020 | Hit |
| 4 | 1020 | 10 | 1030 | Hit |
| 5 | 1030 | 10 | 1040 | Hit |
| 6 | 1050 | 20 | 1070 | Miss |
| 7 | 1060 | 10 | - | Miss |
| 8 | 1070 | 10 | 1080 | - |
| 9 | 1080 | 10 | 1090 | Hit |

This democratic approach is what the prefetcher described in this report implements. To achieve this, the structure of the reference table needs a major change. Instead of the linear array with an offset used in DCPT, this prefetcher utilizes a dynamic 2d array, resembling a sparse matrix. The algorithm is fired every time the program tries to access the cache. The access is logged, and the leap from the previous memory access address (the delta value) is stored in the database. The prefetcher then performs a lookup for the current address, and as the original DCPT, it determines the next delta from the two previous. If an entry is found for this delta combination, a prefetch is issued.

The preferred delta leap on fetch N is determined by delta(N - 1) and delta(N - 2), the length of the two leaps before this one. Delta(N - 2) is put on the Y-axis of the sparse matrix, while delta(N - 1) is mapped along the X-axis. A binary search algorithm searches along the Y-axis (implemented as a C++-std::deqeue) for the delta(N - 2) entry. This entry is implemented as a C++-class containing the delta value and a new std::deque, which contains elements representing all the delta(N-1) entries ever to appear after the given delta(N - 2). The binary search algorithm is then applied to this inner deque, and returns an X-axis element. This element contains the delta(N - 1) value, the proposed next delta and a score for this combination. The ”next delta” found in this X-axis element determines the address of the fetch. The score ensures the ”democracy” in the process. When a ”next delta” is proposed for a delta combination, this proposition gets a start score. The next time the combination appears, the proposed next delta is applied. If the prefetch is a hit, the score increases, if it’s a miss, it decreases. When a score gets below a given threshold, the proposition is replaced by the most recent candidate.

To accomodate to an 8kB memory cap, an upper element count threshold is applied. When the count exceeds this limit, the combination with the lowest score is removed from the structure, and replaced by a new element. This only applies if the lowest score is below a given kick threshold. This is to avoid altering a strong set of elements. In addition to the hit/miss scoring, all elements get deducted one point per memory access, meaning that old, unused combinations are more likely to be abandoned than fresh entries.

The entire algorithm is described in pseudo code in Algorithm 1.

**Algorithm 1** The prefetch_access(addr) implementation

```
static $delta[]$
static $total\_fetches \leftarrow 0$
$delta \leftarrow addr - prev\_addr$

$//apply\ previous\ result:$
Elem $elem \leftarrow$ find_elem($delta[prev]$, $delta[prev-1]$)
**if** $elem == NULL$ **then**
  $elem \leftarrow$ add_combination($delta[prev]$, $delta[prev-1]$)
**end if**
**if** $delta == elem.delta$ **then**
  **if** $elem.score < total\_fetches + BASE\_SCORE$ **then**
    $elem.score += total\_fetches + BASE\_SCORE$
  **else**
    $elem.score += HIT\_SCORE\_BOOST$
  **end if**
**else**
  $//check\ whether\ tochangeelem.delta$
  **if** $elem.score < (SCORE\_KICK\_THRSLD + total\_fetches)$ **then**
    $elem.delta \leftarrow delta$
    $elem.score \leftarrow BASE\_SCORE + total\_fetches$
  **else**
    $elem.score -= MISS\_SCORE\_PUNISHMENT$
  **end if**
**end if**

$//find\ next\ delta:$
Addr $next\_fetch \leftarrow$ get_delta($delta[prev]$, $delta$)
issue_prefetch($next\_fetch$)

$//prep\ for\ next\ run$
$delta[last-1] \leftarrow delta[last]$
$delta[last] \leftarrow delta$
$total\_fetches++$
```

## V. Methodology

The prefetcher was evalued using the given framework which contained a modified version of the M5 open source hardware simulator system. This system uses a selected set of the SPEC CPU2000 bencmarks to evaluate the prefetchers performance [3]. The specific benchmarks which were executed are given in table III in the results section. The benchmarks in this table are selected from both the integer and the floating point components of the CPU2000 benchmark, which generate the data to be compared to a set of reference prefetchers.

In the framework there is also a python script included which decides the arguments and parameters for the simulation. These predefined arguments are given in table II.

The architecture M5 is simulating is loosely based on the Alpha 21264 microprosessor from the DEC Alpha Tsunami system, which is a superscalar out-of-order CPU. The L1 prefetcher is split in a 32kB instruction cache and a 64kB data cache. Each cache block is 64B. The L2 cache size is

TABLE II
PYTHON SCRIPT COMMAND LINE OPTIONS

| Option | Description |
|---|---|
| --detailed | Detailed timing simulation |
| --caches | Use caches |
| --l2cache | Use level two cache |
| --l2size=1MB | Level two cache size |
| --prefetcher=policy=proxy | Use the C-style prefetcher interface |
| --prefetcher=on_access=True | Have the cache notify the prefetcher on *all* accesses, both hits and misses |

1MB, as defined in table II, also with a cache block size of 64B. The L2 prefetcher is notified on every access to the L2 cache, both hits and misses, which is also defined in table II. There is no prefetching for the L1 cache. The memory bus is defined to run at 400MHz with a width of 64 bits and a latency of 30ns [2].

A HPC cluster called ”Kongull” was also avalible to facilitate the M5 system and run the simulations. To increase the efficiency the M5 system was installed on several accounts on the cluster and on a personal computer. This enabled several tests to be run concurrently and independently from one another. The M5 system was also compiled M5 on a personal laptop with a dual-core 1.73GHz Intel Celeron processor running 32 bit CentOS 6.5 with g++ 4.4.7 and python 2.6.6. Regarding the implementation of the actual prefetcher there were some limits to take into account. First of all the allowed size of the prefetcher was limited to a total of 8kB. Since the algorithm is using a set of four integers to store the various delta values and their corresponding data it uses 16B for each entry to the element pool. Taking this into concideration the limit for the numbers of stored elements was set to 448, which totals 7.2kB. This leave 1024B for housekeeping variables, which should be more than enough space to facilitate them.

## VI. RESULTS

As seen in table III, the prefetcher does not provide a significant speedup. Some benchmarks provide better speedup than others, but the average speedup is only 1%. There are some rather noteworthy variations regarding how the prefetcher performs on the different benchmarks, with some even reporting a decrease in speedup. These variations will be discussed further in the following section.

Some minor changes was also performed on the weighting in the scoring system, but the algorithm's speedup in the various tests remained relatively constant with negligible differences in the 0.1% range.

The results in table III adhere to the project guideline of keeping cache size under 8 Kb. Due to the strict limit this imposes on the prefetcher compared to the size of the address space, results from a run without these limits are included in table IV for comparison. As can be seen the results are fairly similar and provided an average speedup of 1.5%. This is still low, but an increase from the implementation when the limit was imposed. As mentioned above, these results are only included for comparison. Common and noteworthy for both of these results are that the prefetcher performs rather poor on the benchmarks that have long series of sequencial fetches. This will be further discussed in the next section.

The dcpt algorithm originally proposed in [?] was also implemented and tested. For comparison the results are presented in table V. As can be seen from the table this implementation is conciderably faster on most benchmarks, and provided an overall avarage speedup of 5.5%.

## VII. DISCUSSION

The prefetcher did not result in any significant speed increase. It seemed like a good algorithm until it was tested. Seeing how many prefetches were done in each tests, it became obvious that attempting to log all patterns seen in such a big address space simply introduced unnecessary complexity. There are extremely many address combinations, reducing the chance that the same patterns are visited multiple times. While the sequences probably will be visited again in the future, many others will take up a lot of the time, rendering the potential speed increase and accuracy negligible.

One could compare the prefetcher detailed in this report with a more simple prefetcher based on the locality principle. While such a prefetcher makes a probabilistic assumption that conceding fetches will be related in some spatial or temporal manner, the prefetcher detailed here attempts to be as deterministic as possible. The margin of error allowed is implicitly much lower in the prefetcher detailed in this report, with only directly mapped sequences being prefetched. The use of spatial and temporal locality therefore seems superior, both in terms of simplicity and results.

Table IV shows an attempt to increase the speedup. It was thought that increasing the cache size would cause many more combinations to be stored, and thus avoid having sequences deleted before they were used due to the massive amount of possible sequences. The lack of speed increase shows that the algorithm was in the wrong from the beginning, and tries to predict too much. It assumes that things happen periodically, something there is no guarantee for in the benchmark tests.

A possible reason for the low increase in speed can be the large overhead of the system. Even though the simulator should not take this into account, seeing as it is a software simulation, The computational delay will cause an increase in runtime. This is of course a tradeoff with prefetchers. While the use of them might make the computation process faster, they still often require computation which steal away CPU cycles.

TABLE III
PREFETCHER RESULTS FROM THE KONGULL CLUSTER

| Test | Speedup | IPC | Accuracy | Coverage | Identified | Issued |
|---|---|---|---|---|---|---|
| ammp | 0.999 | 0.082 | 0.060 | 0.000 | 13765492 | 32526 |
| applu | 1.014 | 0.523 | 0.681 | 0.068 | 662896 | 233366 |
| apsi | 1.015 | 1.507 | 0.628 | 0.020 | 60110 | 3777 |
| art110 | 0.999 | 0.122 | 0.519 | 0.006 | 1728334 | 182419 |
| art470 | 0.999 | 0.122 | 0.519 | 0.006 | 1728334 | 182419 |
| bzip2_graphic | 1.055 | 1.390 | 0.945 | 0.320 | 46207 | 31750 |
| bzip2_program | 1.021 | 1.515 | 0.962 | 0.128 | 10422 | 7394 |
| bzip2_source | 0.997 | 1.705 | 0.964 | 0.209 | 10327 | 7282 |
| galgel | 0.998 | 0.443 | 0.388 | 0.008 | 249284 | 7166 |
| swim | 0.978 | 0.669 | 0.309 | 0.019 | 2033005 | 144134 |
| twolf | 0.997 | 0.423 | 0.632 | 0.001 | 1366 | 862 |
| wupwise | 1.059 | 0.791 | 0.343 | 0.187 | 279083 | 236426 |

TABLE IV
PREFETCHER RESULTS WITHOUT CACHE SIZE LIMIT

| Test | Speedup | IPC | Accuracy | Coverage | Identified | Issued |
|---|---|---|---|---|---|---|
| ammp | 1.000 | 0.082 | 0.210 | 0.001 | 13890679 | 51830 |
| applu | 1.021 | 0.527 | 0.606 | 0.111 | 1508604 | 420247 |
| apsi | 1.015 | 1.508 | 0.680 | 0.052 | 70265 | 9120 |
| art110 | 0.997 | 0.122 | 0.607 | 0.033 | 13203897 | 981292 |
| art470 | 0.997 | 0.122 | 0.607 | 0.033 | 13203897 | 981292 |
| bzip2_graphic | 1.087 | 1.431 | 0.915 | 0.356 | 52025 | 36610 |
| bzip2_program | 1.038 | 1.540 | 0.771 | 0.197 | 17858 | 14172 |
| bzip2_source | 1.000 | 1.709 | 0.768 | 0.259 | 14667 | 11336 |
| galgel | 0.994 | 0.442 | 0.432 | 0.028 | 310736 | 21353 |
| swim | 0.978 | 0.669 | 0.311 | 0.020 | 2125494 | 149677 |
| twolf | 1.003 | 0.426 | 0.450 | 0.115 | 426956 | 264981 |
| wupwise | 1.059 | 0.791 | 0.343 | 0.187 | 279083 | 236426 |

TABLE V
DCPT PREFETCHER RESULTS

| Test | Speedup | IPC | Accuracy | Coverage | Identified | Issued |
|---|---|---|---|---|---|---|
| ammp | 1.000 | 0.082 | 0.148 | 0.000 | 13045 | 11424 |
| applu | 1.074 | 0.554 | 0.333 | 0.124 | 3424217 | 831105 |
| apsi | 1.084 | 1.609 | 0.498 | 0.248 | 60278 | 59727 |
| art110 | 1.056 | 0.129 | 0.918 | 0.377 | 14786042 | 5651241 |
| art470 | 1.056 | 0.129 | 0.918 | 0.377 | 14786042 | 5651241 |
| bzip2_graphic | 1.090 | 1.435 | 0.888 | 0.806 | 89642 | 85079 |
| bzip2_program | 1.059 | 1.571 | 0.793 | 0.588 | 42511 | 41080 |
| bzip2_source | 0.992 | 1.696 | 0.779 | 0.680 | 30397 | 29344 |
| galgel | 1.013 | 0.450 | 0.564 | 0.287 | 239107 | 166359 |
| swim | 1.046 | 0.716 | 0.600 | 0.475 | 3102812 | 1792901 |
| twolf | 0.997 | 0.423 | 0.380 | 0.002 | 6042 | 4990 |
| wupwise | 1.240 | 0.927 | 0.667 | 0.667 | 556492 | 433877 |

The prefetcher is not able to cache something before it has seen it. If all requests are unique, prefetches will never occur. This is especially clear in test ammp shown in table III. The test is in large part incremental. Memory addresses are for the most part only accessed once, so the prefetching has little to no effect here.

While it is not reflected in the test, the prefetcher could do better in a desktop environment.

## VIII. CONCLUSION

All in all the prefetcher is comparable to a deterministic implementation of the more stochastic principle of locality approach. The idea of mapping possible sequences directly to their assumed following address might work in a predictable environment, but does not seem to function well in practice. A more useful approach would be to cache the most frequently accessed addresses, or simply the address area around a request. Another reason for the prefetcher's low performance is the size of the accessed memory spaces. With so many combinations, it is not too likely that the same sequences will be frequented a lot. If the memory space was smaller, the same sequences might have been frequented more, resulting in higher accuracy. With the amount of addresses being high, the actual hits drown in the vast amount of possible sequences.

## References

[1] M. Grannaes, M. Jahre and L. Natvig *Storage Efficient Hardware Prefetching using Delta-Correlating Prediction Tables*, Journal of Instruction-Level Parallelism 13 (2011) 1-16

[2] *M5 simulator system TDT4260 Computer Architecture User documentation*, Assignment text

[3] (2014, Apr) The SPEC website. [Online]. Available: http://www.spec.org/

[4] John L. Hennessy, Stanford University and David A. Patterson, University of California, Berkeley *Memory Hierarchy Design - Part 1. Basics of Memory Hierarchies*, 2012. http://www.edn.com/design/systems-design/4397051/2/Memory-Hierarchy-Design-part-1.