TDT4260 Project Report

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

***Abstract*—**

I. Introduction

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.

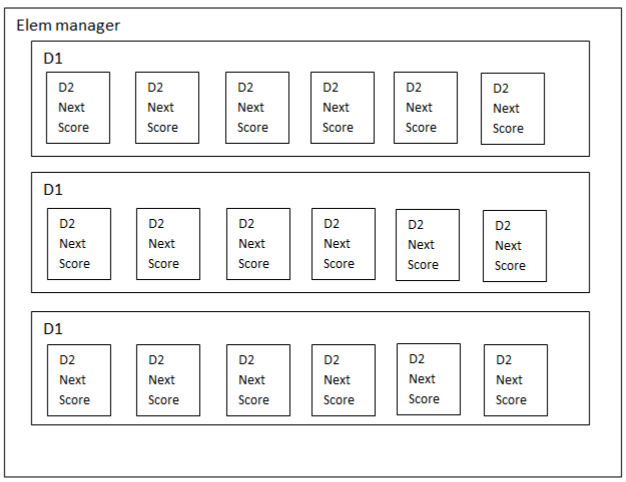

Fig. 1. Dette er ei tekning

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 memory anyway. This is the worst case scenario, in which the cache has no effect on performance.

The goal of this project was to make a prefetcher that would increase the performance of a microprocessor. The idea was also to run the prefetcher in a simulator environment with equal hardware specifications for everyone, so that everyone could work with a common interface and the results would be comparable.

The prefetcher presented in this report is based on pattern recognition implemented as a two dimensional vector structure.

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

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

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 sicne it does not have any form of pattern recognition. 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 statistic.

A Refernce Prediction Tables RPT is a prefetcher that is based on statistics. This type of prefetcher was first proposed in 1995 by Chan and Baer.

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.

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

To evaluate the prefetcher we used 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. The specific benchmarks which were executed are given in table II 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 **??**.

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

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

We also had access to a HPC cluster called "Kongull" on which we could compile the M5 simulator and run the simulations. To increase the efficiency we installed the M5 system on all of our accounts on the cluster and on a personal computer. This enabled us to run several tests concurrently as well as running shorter tests

to enable us to run several simulations concurrently. The personal computer

The framework provided with the assignment was used for

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

simulation. Our code was at first uploaded to the Kongull cluster, but this took a lot of time due to long queue times. The framework was therefore installed on one of our computers, running on Linux CentOS. This worked fine, but most of our other computers couldn't run the simulator because the GCC/G++ versions were too new.

The simulator used was the modified M5 simulator provided with the assignment.

VI. RESULTS

As seen in table II, the prefetcher does not perform as well as anticipated. Some tests provide better speedup than others, but the average speedup is only 1%. There is huge variation between the different tests, with some of the tests even reporting a speed decrease. The prefetcher was initially run on a Linux CentOS, but the results differed greatly from the Kongull Cluster results. Kongull's output was consistent while the other simulator's output varied a lot. Therefore it seemed that Kongull was the most correct simulator

Compared to different runs of the same algorithm on the Kongull Cluster, the algorithm's speedup in the various tests remain relatively constant with some being better and some slightly worse.

VII. DISCUSSION

asdf

VIII. CONCLUSION

APPENDIX A
PROOF OF THE FIRST ZONKLAR EQUATION

Appendix one text goes here.

APPENDIX B

Appendix two text goes here.

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