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.
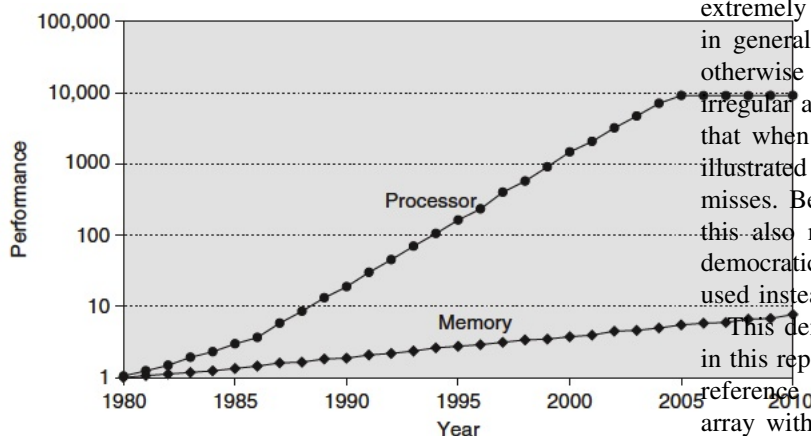

Fig. 1. Dette er ei tekning

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 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. A RPT contains a table with adresses of the missed fetches as index.

An another type of prefetching is called Delta Correlating Prediction, as described in ¡INSERT REF HERE!!!¿. This prefetcher is based on the RPT and PC/DC.

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

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

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.

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

---

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($next\_fetch$)

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

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

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.

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	3256
applu	1.014	0.523	0.681	0.068	662896	3366
apsi	1.015	1.507	0.628	0.020	60110	5477
art110	0.999	0.122	0.519	0.006	1728334	2019
art470	0.999	0.122	0.519	0.006	1728334	2019
bzip2_graphic	1.055	1.390	0.945	0.320	46207	750
bzip2_program	1.021	1.515	0.962	0.128	10422	924
bzip2_source	0.997	1.705	0.964	0.209	10327	2282
galgel	0.998	0.443	0.388	0.008	249284	7106
swim	0.978	0.669	0.309	0.019	2033005	4134
twolf	0.997	0.423	0.632	0.001	1366	86
wupwise	1.059	0.791	0.343	0.187	279083	264236

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 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 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].

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 and independently from one another. We also compiled M5 on a personal laptop with a dualcore 1.73GHz Intel Celeron processor running 32 bit CentOS 6.5 with g++ 4.4.7 and python 2.6.6.

VI. RESULTS

As seen in table III, the prefetcher does not provide a significant speedup. 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. Especially the benchmarks that use a lot of incrementation suffer bad results. The prefetcher is not able to cache something before it has seen it. If all requests are unique, prefetches will never occur.

The results above 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.

The results without a cache size limit, shown in table IV resulted in an average speedup of 1.5%. This is still low, but an increase from when the limit was imposed. As mentioned above, these results are only included for comparison.

The dcpt algorithm was also implemented and tested for comparison. Results are shown in table

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 are done in any of the tests, it became obvious that attempting to log all patterns seen in such a big address space simply introduced complexity.

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

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

[3] SPEC, "SPEC CPU2000 Benchmark Descriptions", 2007. http://www.spec.org.