Comparing hardware prefetchers with different block sizes

As a result of processing speed outpacing memory speed, prefetchers are used to speed up memory accesses. In this paper, different common prefetching strategies are compared using the M5 simulator with the SPEC CPU2000 benchmarks. The testing reveals that the stride-based DCPT and RPT algorithms provide a speedup increase of around 1.08 when compared to no prefetching, allowing for reduced cache memory size along with the computational speedup.

I. Introduction

Modern consumer processors, such as the Intel i7 8700k, can perform instructions at 324.4 GIPS (billion instructions per second) [1]. Comparing this with a similar processor from 2008, the Intel i7 920 which performed instructions at 68.25 GIPS [2], shows that the instruction speed has been increased by a factor of 4.7 over a period of 10 years. The increase in processing speed is the result of several factors, such as smaller transistor sizes allowing for more transistors on the same chip area, allowing for more pipelined and parallel data flow. The speed increase results in a need for quick access to data at a pace which regular memory can not keep up with. Memory speed has not increased at the same pace as processing speed, causing a performance gap called ”the memory gap” [3]. This is shown in Figure 1 which depicts the increase in processor and memory performance. The use of caches, distributed over several layers closer and closer to the processor, help reduce the memory access time by loading data required by the processor when needed or through prefetching. Prefetching is process of using algorithms to try and predict what data is needed by the processor in the coming clock cycles.

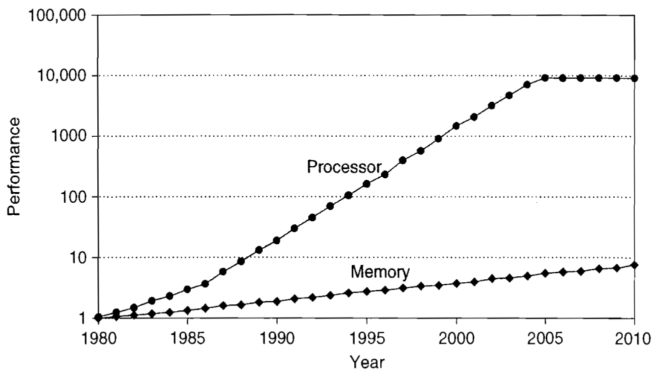

Fig. 1. Memory gap [4]

This paper aims to test and compare the speed increase and hit rate of various prefetching algorithms using different cache block sizes. The paper begins with a brief introduction to caches and prefetching, followed by the tested algorithms and how they were implemented and tested, followed by results, discussion and a conclusion.

II. Background

Prefetching can be done in two ways, either through software or hardware. Software prefetching is usually done at compiler level, with the compiler introducing prefetching functions where it finds it necessary, e.g. in loops.

Hardware prefetching is based on implementing prefetching algorithms in the physical processor. Several different prefetching algorithms exist, using varying techniques to try to predict the upcoming data stream.

Cache is memory located on or near the CPU, and is usually small in size when compared to RAM and hard drives, usually in the range from $32\,\mathrm{KiB}$ for L1 cache to $2\,\mathrm{MiB}$ per core for L3 cache [5], in modern processor architectures such as the Intel Coffee Lake CPU series. This relatively small size calls for the use of replacement policies and associativity to take advantage of the speed increase from using caches.

Cache block size is how much of the data loaded from the memory is stored in one cache block. If the cache is $1\,\mathrm{MiB}$ and the cache block size is $64\,\mathrm{B}$, the cache can hold $2^{20}/2^{6} = 2^{14} = 16384$ blocks of data. The data is mapped according to the chosen cache associativity. If the cache is directly mapped, the first $16\,\mathrm{B}$ of data are stored in block 0. Increasing cache block size means that more data is stored in the same block, so prefetching a certain block for a specific byte of data means that additional data is loaded alongside.

Memory locality is the distribution of data on the different levels of the memory hierarchy. The memory hierarchy is the composition of different memory modules, such as different levels of cache, RAM and hard drives, together making a complete memory system for data storage. Data is usually distributed according to two types of locality: Spatial locality, where address near each other are most likely referenced close together in time, and temporal locality, where the a recently used address is more likely to be reused in the future. Locality is one of the reasons why cache prefetching is efficient, as data is more likely to be distributed sequentially with constant strides, or repeating strides, due to spatial locality.

There are three types of misses which can occur in a cache: Compulsory-, conflict- and capacity misses. Conflict misses occur when the address associated with a block of data is not on the mapped location, i.e. a piece of data has been overwritten. This can occur in associative caches. Capacity misses occur when the cache is full. Compulsory misses occur when the requested data is not present in the cache. This miss can be reduced by using prefetching.

When a cache miss occurs, the requested data has to be fetched from a either a higher level of cache, RAM or the hard drive. Each level of storage has an increased access time, as shown in table I:

CPU registers	$1\,\text{ns}$	$512\,\text{B}$
Cache	$10\,\text{ns}$	$12\,\text{MiB}$
RAM	$100\,\text{ns}$	$8\,\text{GiB}$
Hard disk	$10\,\text{ms}$	$2\,\text{TiB}$

TABLE I
STORAGE ACCESS TIMES [6]

The access time increases from as low as $1\,\text{ns}$ to $10\,\text{ms}$. In most cases the slowest access time is that of the RAM, as most programs load the relevant data to the RAM when needed.

Cache pollution is the case were unnecessary data is loaded into the cache. This can be caused by a prefetcher loading to much unnecessary data.

III. THE ALGORITHMS

Several prefetching algorithms exist, varying in complexity, area and energy usage.

A. Sequential on miss

Sequential on miss is one of the simpler prefetcher algorithms. Every time there is a cache miss, the data for the next cycle is fetched. This is done by adding the block size to the current memory address.

B. Sequential on access

Sequential on access is similar to sequential on miss, but instead of prefetching when there is a cache miss, the prefetcher loads the next block on every cache access.

C. RPT - Reference Prediction Table

The Reference Prediction Table algorithm was introduced by Chen and Baer in 1995 [7]. The algorithm checks for cache accesses with a constant, sequential stride, where stride is the difference between the previous and current memory address at the same PC (program counter) value. As Figure 2 shows, it does this by using a table where each entry contains the PC value as a tag, the last address, the stride, and a state field.

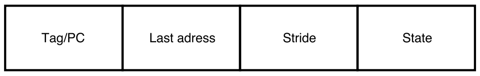

Fig. 2. Format of a RPT table entry [7]

There are four possible states; Init, Steady, Transient and No-pred, and prefetching is only performed if the state is steady. The state machine for the algorithm is shown in Figure 3. On the first miss the entry is created and the state is set to initial and the stride is set to 0. On the next cache access, the state is changed to steady if the current stride of 0 is correct, otherwise the state is changed to transient and the stride is updated. On the next access, the state changes to steady if the previous stride is the same as the current stride, otherwise the state is changed to No-pred and the stride is updated. As such, the algorithm requires two sequential strides of the same value to begin prefetching data.

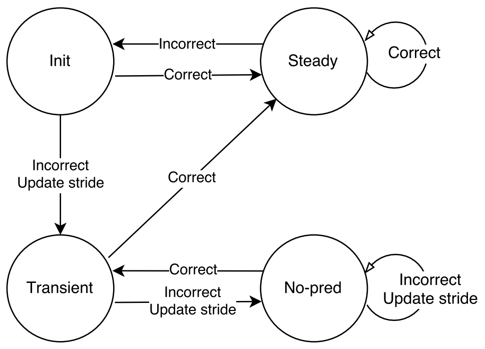

Fig. 3. RPT state transition [7]

D. DCPT - Delta Correlating Prediction Tables

DCPT is a stride-based algorithm introduced by Grannaes, Jahre and Natvig in 2011 [8]. It builds on the RPT algorithm and the PC/DC algorithm introduced by Nesbit in 2004 [9]. DCPT combines the table indexing of RPT and the delta-calculation and global history buffer of PC/DC to calculate the delta-strides, the difference between the previous and current requested address on the same program counter (PC) value. As Figure 4 shows, the table entries use the PC value as the tag.

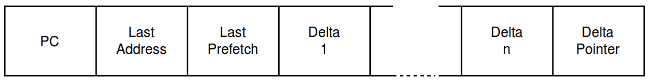

Fig. 4. Format of a DCPT table entry [8]

The previously requested memory address is stored in the entry, in addition to the address of the most recent prefetch. N number of delta's are stored in the entry in a buffer, with a delta pointer in the last space of the entry which points to the most recently calculated delta.

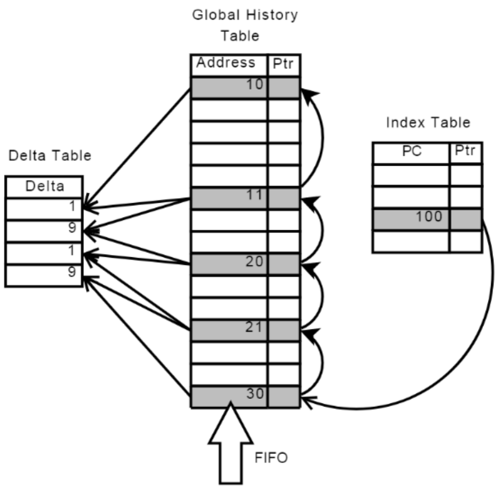

Fig. 5. Global history buffer with deltas [8]

A cycle of DCPT starts with checking whether the current PC-value has been encountered before. If not, a new entry is created. Otherwise, the delta is calculated and the delta-values of the current PC entry are updated as long as the new delta is not equal to zero. The algorithm then compares the two most recently calculated delta's to the delta buffer, traveling backwards in the buffer. If a match is found, the next delta, from newest to oldest delta, is added to the current memory address and the new address is prefetched. If a more aggressive prefetching strategy is desired, M number of deltas after the next delta can be added to the calculated memory addresses and prefetched. Figure 5 shows the global history buffer with the four most recent deltas, 1, 9, 1 and 9. The algorithm looks for the 1-9-pattern and finds it in the two previous entries. It then sets the prefetch-delta to 1 and issues a prefetch request for the data in address 31.

The prefetch candidates are looked up in the cache to see if they are already present, if they are in the miss status holding registers, which is a register for storing missed data, and whether a prefetch request already exists. If the prefetch candidate passes these checks, the last prefetch field of the entry is updated with the candidate address, and prefetch is added to the prefetch queue.

IV. Methodology

A. Simulator

The M5 simulator was used to test the different prefetcher algorithms. The simulated architecture is based on the Alpha 21264 microprocessor, and the microprocessor details are given in Table II. The L1 cache does not have a prefetcher, and the simulated prefetcher runs on the L2 cache.

L1 data cache	64 KiB	Memory bus speed	400 MHz
L1 instruction cache	32 KiB	Memory bus width	64 bit
L2 cache size	1 MiB	Memory latency	30 ns

TABLE II
Simulator setup [10]

In addition to these parameters, the algorithms were tested with cache block sizes of 16 B, 32 B, 64 B and 128 B.

The prefetching algorithms were implemented in C++ and compiled for the M5 simulator. The simulator uses the SPEC CPU2000 benchmarks for testing the efficiency of the algorithms. SPEC CPU2000 is a software benchmark product by the Standard Performance Evaluation Corporation released in 1999 [11], and includes 11 integer and 14 floating point compute-heavy applications for testing the efficiency of the hardware.

The M5 simulator measures several key performance indicators:

Speedup	Overall performance. Performance increase compared to using no prefetccher.
IPC	Instructions per cycle. How many instructions are executed per clock cycle on the microprocessor. Since the Alpha 21264 microprocessor is superscalar, this number can be greater than 1, as the processor can execute instructions in parallel.
Accuracy	Ratio of correct prefetches.
Coverage	Ratio of potential prefetches that were executed.
Identified	Number of prefetches issued by the prefetcher.
Issued	Number of prefeches actually executed, this number can be less than the `Identified` number due to duplicate prefetches already found in the prefetch queue, and prefetches dropped due to a full prefetch queue.
Misses	The total number of misses in the L2 cache.

V. Results

A. Sequential on miss

Test	IPC	Speedup	Accuracy	Coverage
ammp	0.05	0.88	0.00	0.00
applu	0.48	1.11	0.84	0.27
apsi	1.54	1.02	0.06	0.02
art110	0.12	0.96	0.35	0.13
art470	0.12	0.96	0.35	0.13
bzip2_graphic	1.42	1.02	0.57	0.31
bzip2_program	1.20	1.02	0.69	0.34
bzip2_source	1.66	1.00	0.70	0.36
galgel	0.70	1.04	0.54	0.10
swim	0.72	1.01	0.34	0.21
twolf	0.45	1.03	0.42	0.18
wupwise	0.79	1.06	0.58	0.29
Overall speedup	1.01			

TABLE III
CPU2000 simulation results for sequential on miss prefetching with a cache block size of 64 B.

Table III shows the results of a simulation of the sequential on miss prefetcher using a cache block size of 64 B. The algorithm performs worse compared to not prefetching on ammp, art110 and art470. The cause is that the algorithms fill the cache with unnecessary data, replacing useful data. On the ammp test, the prefetcher recognises less than 1% of prefetch candidates. The prefetcher does provide a speedup increase on 9 of the 12 programs, resulting in an overall speed increase of 1%.

Cache block size [B]	Speedup
16	0.96
32	0.96
64	1.01
128	1.00

TABLE IV
Comparison of speedup with different cache block sizes for sequential on miss prefetching.

As Table IV shows, the prefetching speedup becomes worse than not prefetching at all for cache block sizes of 16 B and 32 B, and overall offers no improvement for a block size of 128 B.

B. Sequential on access

Table V shows the results of a simulation of the sequential on access prefetcher with a cache block size of 64 B. The

Test	IPC	Speedup	Accuracy	Coverage
ammp	0.05	0.88	0.00	0.00
applu	0.48	1.11	0.83	0.36
apsi	1.54	1.03	0.13	0.04
art110	0.11	0.94	0.34	0.14
art470	0.11	0.94	0.34	0.14
bzip2_graphic	1.43	1.03	0.64	0.53
bzip2_program	1.23	1.05	0.76	0.62
bzip2_source	1.66	1.00	0.80	0.64
galgel	0.71	1.05	0.71	0.23
swim	0.72	1.01	0.46	0.33
twolf	0.45	1.03	0.48	0.39
wupwise	0.82	1.10	0.46	0.37
Overall speedup	1.01			

TABLE V
CPU2000 SIMULATION RESULTS OF SEQUENTIAL ON ACCESS PREFETCHING WITH A CACHE BLOCK SIZE OF 64 B.

algorithm performs in a similar fashion to the sequential on miss prefetcher, improving on the results of the miss prefetcher in tests such as apsi and bzip2_graphic, resulting in an overall speedup of 1.01. The algorithm also decreases speedup on art110 and art470 when compared to the sequential on miss prefetcher. This is most likely caused by the test having a memory access pattern which is not sequential. The opposit is the case for tests such as applu and wupwise, which most likely feature more data which is located with constant strides, usually in the next data block.

Cache block size [B]	Speedup
16	0.96
32	0.96
64	1.01
128	1.01

TABLE VI
COMPARISON OF SPEEDUP WITH DIFFERENT CACHE BLOCK SIZES FOR SEQUENTIAL ON ACCESS PREFETCHING.

Table VI shows the overall speedup increase of the prefetching algorithm for the various cache block sizes. The prefetcher performs similarly to the sequential on miss prefetcher, only increasing the speedup for a cache block size of 128 B.

C. RPT

Test	IPC	Speedup	Accuracy	Coverage
ammp	0.09	1.52	0.78	0.63
applu	0.45	1.05	0.95	0.38
apsi	1.55	1.03	0.36	0.15
art110	0.13	1.05	0.86	0.46
art470	0.13	1.05	0.86	0.46
bzip2_graphic	1.41	1.01	0.69	0.32
bzip2_program	1.22	1.04	0.86	0.38
bzip2_source	1.66	1.00	0.78	0.42
galgel	0.73	1.08	0.37	0.20
swim	0.72	1.00	0.43	0.33
twolf	0.43	1.00	0.42	0.01
wupwise	0.90	1.21	0.67	0.67
Overall speedup	1.07			

TABLE VII
CPU2000 SIMULATION RESULTS OF RPT PREFETCHING WITH A CACHE BLOCK SIZE OF 64 B.

The RPT algorithm was expected to perform on par with or better than the sequential prefetchers, as it has an adaptive stride which can perform the same prefetching as the sequential, loading data with a stride equal to the cache block size. Table VII shows the simulation results with a cache block size of 64 B. Overall, the prefetcher performs better than the sequential prefetchers, providing a significant increase in speedup on the ammp and wupwise tests, resulting in an overall speedup of 1.07. The prefetcher does however perform worse than the sequential prefetchers on some tests, such as the applu test. The cause of this might be irregular stride patterns, resulting in the RPT algorithm staying in the No-prediction state as shown in figure 3, which the sequential prefetchers are not affected as they, in general, perform more prefetches.

Cache block size [B]	Speedup
16	1.07
32	1.07
64	1.07
128	1.07

TABLE VIII
COMPARISON OF SPEEDUP WITH DIFFERENT CACHE BLOCK SIZES FOR RPT PREFETCHING.

Table VIII shows that the overall speedup remains stable independent of cache block sizes. This is most likely a result of the algorithm predicting, relatively accurately, the data required in the future as well as performing few unnecessary prefetches filling the cache with useless data.

D. DCPT

Similar to RPT, but improved in regards to stride recognition, the DCPT prefetcher is expected to perform on par with or better than RPT. As table IX shows, it performs as expected on certain tests, especially ammp, providing a speedup of 1.67, a 0.17 increase from RPT. It does, however, perform slightly worse on some tests, such as apsi. Nevertheless, the total speedup is increased to 1.08 with a cache block size of 64 B. The cause of the increase probably comes a result of the prefetcher being able to handle different, but still sequential, stride patterns, such as 1-4-1-4.

Test	IPC	Speedup	Accuracy	Coverage
ammp	0.10	1.67	0.55	0.52
applu	0.45	1.05	0.97	0.34
apsi	1.52	1.01	0.38	0.09
art110	0.13	1.05	0.86	0.47
art470	0.13	1.05	0.86	0.47
bzip2_graphic	1.42	1.02	0.68	0.32
bzip2_program	1.22	1.04	0.87	0.37
bzip2_source	1.66	1.00	0.79	0.42
galgel	0.71	1.05	0.43	0.13
swim	0.72	1.00	0.50	0.35
twolf	0.43	1.00	0.30	0.00
wupwise	0.91	1.23	0.67	0.67
Overall speedup	1.08			

TABLE IX
CPU2000 SIMULATION RESULTS OF DCPT PREFETCHING WITH A CACHE BLOCK SIZE OF 64 B.

Table X shows that the overall prefetching speedup remains stable at 1.08 except for a cache block size of 32 B where it falls to 1.06. Why this happens is unclear, as the overall speedup of 1.08 is achieved for a block size of 16 B. Table XI shows the two most significant speedup differences with

Cache block size [B]	Speedup
16	1.08
32	1.06
64	1.08
128	1.08

TABLE X
COMPARISON OF SPEEDUP WITH DIFFERENT CACHE BLOCK SIZES FOR DCPT PREFETCHING.

a cache block size of 16 B when compared to the rest of the DCPT simulations. The cause of the reduction might be that the prefetcher fetches unnecessary data along with the correctly predicted data as a result of the block size, replacing useful data in the cache. The block size is just big enough to cause extra misses compared to block sizes of 16 B and 64 B for the programs in question.

Block size [B]	Test	IPC	Speedup	Accuracy	Coverage
32	ammp	0.09	1.43	0.89	0.67
	wupwise	0.87	1.17	0.69	0.53
64	ammp	0.10	1.67	0.55	0.52
	wupwise	0.91	1.23	0.67	0.67

TABLE XI
MOST SIGNIFICANT SPEEDUP DIFFERENCES WITH DCPT PREFETCHING WITH CACHE BLOCK SIZES OF 32 B AND 64 B.

E. Comparison

Table XII shows the overall speedup for the different algorithms with a cache block size of 64 B. All algorithms provide an overall speedup increase, with the stride-based prefetchers performing significantly better than the sequential prefetchers. The cause of this is that they have adaptive strides instead of constant strides. Still, the sequential prefetchers performed better in some tests. This is most likely due to the fact that they, in general, perform more prefetches than the stride-based prefetchers.

Prefetcher	Speedup
Sequential on miss	1.01
Sequential on access	1.01
RPT	1.07
DCPT	1.08

TABLE XII
SPEEDUP COMPARISON OF THE SIMULATED PREFETCHERS WITH A CACHE BLOCK SIZE OF 64 B.

This is reflected in the accuracy and coverage values. Comparing the accuracy of all prefetchers on the bzip2_program test shows that the stride-based prefetchers have an accuracy of around 0.86 while the sequential prefetchers have an accuracy of 0.69 and 0.64 for the miss and access prefetchers, respectively. A more extreme case can be found for the ammp test, where the stride based prefetchers have an accuracy of 0.78 and 0.55 for RPT and DCPT, respectively, while both sequential prefetchers have an accuracy under 1%.

VI. DISCUSSION

Hardware prefetching provides, in most cases, a speedup increase. This comes at a cost, though, as hardware prefetchers require more space on the chip, power and bandwidth. Unless the prefetcher has 100% coverage and accuracy, the prefetcher will use more power and bandwidth than no prefetching as unnecessary data is loaded from the memory to the cache. This is especially the case for the sequential prefetchers as they have relatively low accuracy and coverage. The stride based prefetchers require more area and power than the sequential prefetchers, but provide a substantial speedup increase, as well as performing fewer unnecessary fetches.

Area usage is critical in chips, and cache memory requires a relatively large area when compared to the rest of the components on a CPU. Reducing the cache block size is similar to reducing the cache memory size as the prefetchers have to be more accurate in their predictions. The sequential prefetchers provide reduced speedup in this case, while both stride based prefetchers perform well, with RPT remaining stable across all tested block sizes.

VII. CONCLUSION

The goal of the paper was to simulate the efficiency of several hardware prefetching algorithms. The results show that stride-based prefetching algorithms provide speedup increases of over 6% across all tested cache block sizes. This could allow for reduced cache memory size as unnecessary data is replaced with data required in the future, as well as increasing the computational speed as the delay resulting from loading data from higher memory levels is reduced. Even though more advanced and efficient hardware prefetchers require more area than simpler prefetchers, the possible reduction in cache memory and computational speedup makes it worthwhile.

REFERENCES

[1] M. Chiapatta, “Intel core i7-8700k and core i5-8400 review: Coffee lake - more cores, performance and value,” https://hothardware.com/reviews/intel-core-i7-8700k-and-core-i5-8400-coffee-lake-processor-review, accessed 1/3/2018.

[2] J. Apong, “Intel core i7 920 nehalem 2.66ghz processor review,” http://www.pcstats.com/articleview.cfm?articleid=2582&page=6, accessed 28/3/2018.

[3] C. Carvalho, “The gap between processor and memory speeds,” in *Proceedings of 3rd. Internal Conference on Computer Architecture*, Braga, Portugal, Jan. 2002.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach, 5th ed.*, 2011.

[5] Intel, “Products formerly coffee lake,” https://ark.intel.com/products/codename/97787/Coffee-Lake, accessed 8/3/2018.

[6] AVNET, “Storage hierarchy,” http://www.as.techdata.eu/uk/images/StorageHierarchy.jpg, 2017, accessed 6/3/2018.

[7] T.-F. Chen and J.-L. Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE TRANSACTIONS ON COMPUTER*, vol. 44, 1995.

[8] M. Grannaes, M. Jahre, and L. Natvig, “Storage efficient hardware prefetching using delta-correlating prediction tables,” *Journal of Instruction-Level Parallelism*, vol. 13, 2011.

[9] K. J. Nesbit and J. Smith, “Data cache prefetching using a global history buffer,” *High-Performance Computer Architecture, International Symposium on*, vol. 0, 2004.

[10] “M5 simulator system tdt4260 computer architecture user documentation,” 2017.

[11] S. P. E. Company, “Spec cpu2000 press release faq,” https://www.spec.org/cpu2000/press/faq.html, 1999, accessed 29/3/2018.