International Journal of Psychosocial Rehabilitation, Vol. 24, Issue 05, 2020
ISSN: 1475-7192

Comparative Study for Catch Prefetching Algorithms

[1]Mustafa Asaad Hasan; [2]Ali Hussein Lazem ; [3]HAYDER A. JAWDHARI

Abstract— *There are many techniques that have been proposed to decrease memory latency, such as caches, locality optimizations, pipelining, out-of-order execution, and multithreading. Another such technique is called prefetching. Prefetching brings data or instructions closer to the processor before it is needed so that the processor will not have to stall and wait for the data, thereby reducing the cache miss rate and decreasing memory access latency. In this paper we have implemented one data prefetching algorithm called Next-N-Line, and three different instruction prefetching algorithms: Next Line, Target-Line, and Wrong-Path. These algorithms were implemented using a simulator created by the University of California San Diego. We compared the results of these four simulations and tested different parameters as we will show latter. Our goal in this paper is to compare the effects of data prefetching with no prefetching. These results will also help to decide which algorithm has the lowest average memory access time, miss rate, and run time among the three instruction prefetching algorithms that we implemented.*

Keywords: *Catch Prefetching, Instruction Prefetching, Data Prefetching, Miss Rate, Hit Rate.*

1. INTRODUCTION

File buffer, classic disk and cache management are interactive; accesses of the disk are begun and buffers assigned in response to application requests for file data. Several software and hardware prefetching schemes have been proposed to reduce the effect of miss penalties on processor utilization in shared memory multiprocessors. In this work we will present hardware prefetching. A significant advantage of hardware prefetching techniques over software techniques is that they do not need support from the programmer or compiler.

Processor performance seems to follow Moore's Law, increasing by 60% a year, while memory performance increases by just 7% each year as we show in figure1. This was our first motivation for looking for new and effective strategies to reduce the large gap between processor speed and memory speed. Large latency represents the second reason for our project: to decrease memory access time using multiple cache levels such as L1, L2 and L3. Caches will lose their benefit if the information that the CPU needs is not in the cache, resulting in a cache miss. Therefore, we plan to increase the effectiveness of caching by using prefetching[1][2][6].

Storage parallelism is growingly ready for use to take shape of disk arrays and striping device drivers. These hardware and software arrays pledge the I/O throughput necessary to get the balance ever-hurry CPUs through data distributing of a single file system during numerous disk arms. The advantage workloads of trivially parallel I/O immediately; major accesses well-being from the parallel transfer, and various synchronous accesses benefit from separate disk actuators.

There are two main types of prefetching: software prefetching and hardware prefetching. Software prefetching is done by the programmer or the compiler by adding lines to the code that prefetching data before it is needed inside the program. This method usually works well with regular access patterns. However, software prefetching requires more effort from the programmer. Hardware prefetching works by monitoring processor accesses to determine what information will likely be

[1]Assistant Lecturer, University of Thi-Qar, Iraq/ Mustafa.alkhafaji@utq.edu.iq
[2] Assistant Lecturer, Computer Center, University of Thi-Qar, Iraq/ alazem@utq.edu.iq
[3]Assistant Lecturer, AL Qasim Green University, Iraq/ haider.satar@uoqasim.edu.iq

used in the near future, and prefetching it automatically. There are two types of hardware prefetching: data prefetching and instruction prefetching [5][6][7][8].
Processors that have high-performance utilize aggressive offshoot foretelling mechanisms so as to take advantage of high levels of instruction-level parallelism. Unluckily, even with low branch misjudgment averages, these processors use a great number of instructions of cycles fetching from the mis-predicted program path[1][2][11].

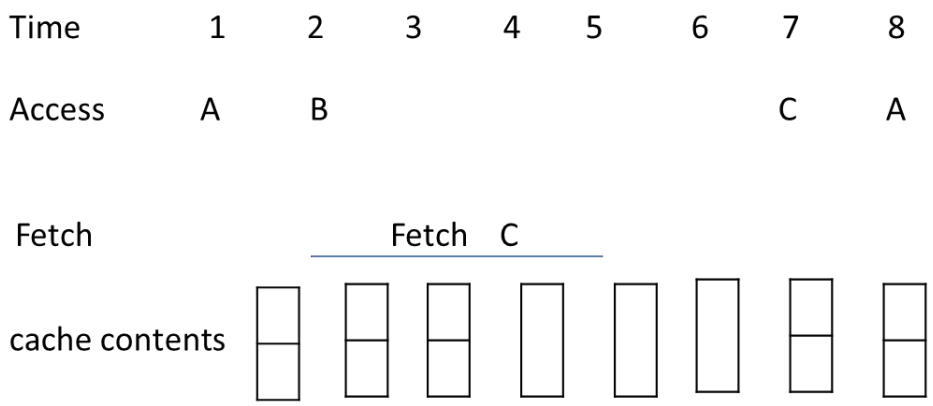


Figure 1: Example under the no-prefetch policy. Eight-time units are required. The first two references, to A and B hit; the next reference, to C, stalls for four-time units while C is fetched, and B is discarded. The final reference, to A, hits.

Look a small program which indicates blocks as stated by the Pattern “ABCA". Let we assume this file cache has two blocks, each fetching a block spends four-time units, and the initial state in the cache represented by A and B.
A no-prefetch policy may brief in figure 1, (depending on the optimal offline algorithm) may be getting a strike on the top two references, after that, Passed to the reference to C, dispose of B, and finally passed on A. The time of enforcement of the no-prefetch policy becomes eight-time units (four references, plus four units for the pass.) Differ from figure 2, a policy that prefetches if that probable picks ten-time units to execute this sequence. After the rest successful access to A, a prefetch of C is initiated, discarding A. This prefetch burrows one unit of the fetch latency, so the access to C stalls for only three cycles. Once C arrives in memory, the algorithm initiates another, bringing in A and discarding B, while accessing C. The next reference, to A, stalls for three time units, waiting for the prefetch of A to complete. This algorithm uses ten time units, one for each of the four references, plus two stalls of three time units each[1][2][3][4][6][11].

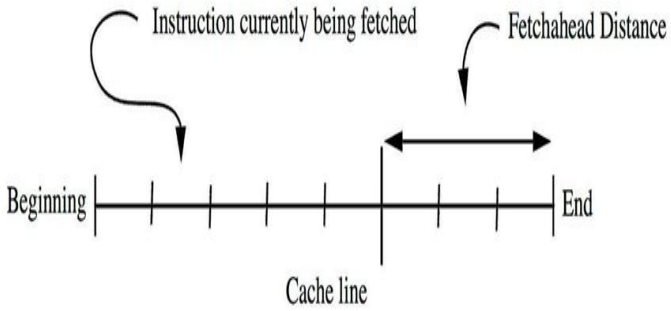
2. PREFETCHING

2.1. Instruction prefetching

Instruction prefetching attempts to prefetch future instructions before they are needed by the CPU. The following algorithms monitor the CPU’s instruction cycle to determine what instructions to prefetch.

2.1.1 Next-Line Algorithm

This algorithm tries to prefetch sequential cache lines before they are needed by the CPU’s fetch unit. The current cache line is the line that contains the instruction that is currently being fetched by the CPU. The next line is defined as the cache line which is located sequentially after the current line[11]. This schema works as follows: if the next line is not found in the cache, then it will be prefetched when an instruction located some distance into the current line is accessed. This specified distance, which is called the fetch a head distance, is measured from the end of the cache line as illustrated in the Figure 2.



The fetch ahead distance can drastically affect the performance of this algorithm. It shouldn't be too small in case we encounter a branch, and it shouldn't be too large because we may not be able to bring the data into the cache in time. Next-line prefetching doesn't work for conditional branches, as the execution will fall through any conditional branches in the current line and continue along the sequential path. This algorithm is simple and doesn't require any extra hardware[7][8][9][10].

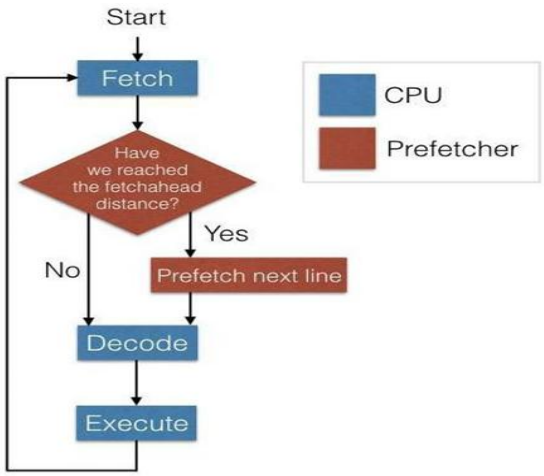


Figure 2 : Next-Line algorithm

2.1.2 Target-Line Algorithm

Target-Line prefetching is an instruction prefetching algorithm that attempts to capture more control transfer targets. In this algorithm, a target table is used to maintain a history of the paths taken for conditional branch instructions. The target table consisted of two entries: address tag and target address. This algorithm works just like the Next-Line algorithm until a conditional instruction is encountered. Once a conditional branch instruction is fetched, the path that was taken on the previous execution of the control instruction is prefetched, assuming there is an entry for this instruction in the table. After the instruction is executed, the target table is updated with the address for the executed instruction and the path taken on this execution. This algorithm makes the assumption that the path taken previously for an instruction will likely be the path taken the next time. Using this approach, the hardware will hopefully be able to prefetch on the correct path if an entry is found in the table[11].

The main advantage of this algorithm is that it generally performs better than the Next-Line algorithm, and works especially well when the execution flow of the program follows the path of the previous execution. However, this algorithm is very costly in terms of chip area and added complexity because it requires extra hardware, as well as logic for updating and searching the target table. Also, this algorithm results in cold misses in the target table until enough conditional instructions have been executed to fill the target table is filled.

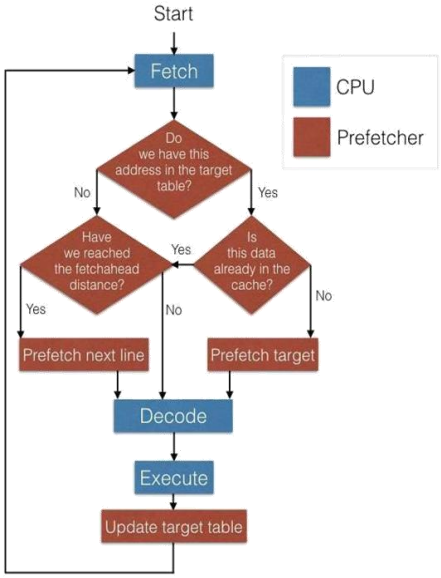


Figure 12: Target-Line algorithm

2.1.3 Wrong-Path Algorithm

The Wrong-Path algorithm is similar to Next-Line and Target-Line. Until a conditional branch instruction is encountered, Wrong-Path works just like Next-Line, bringing in the next sequential line once the fetch ahead distance has been reached in the current line. However, if a conditional branch instruction is encountered during the decode stage of the instruction cycle, the Wrong-Path algorithm prefetches the target of the branch instruction. Prefetching the target of a conditional branch takes precedence over prefetching the next sequential line[5].

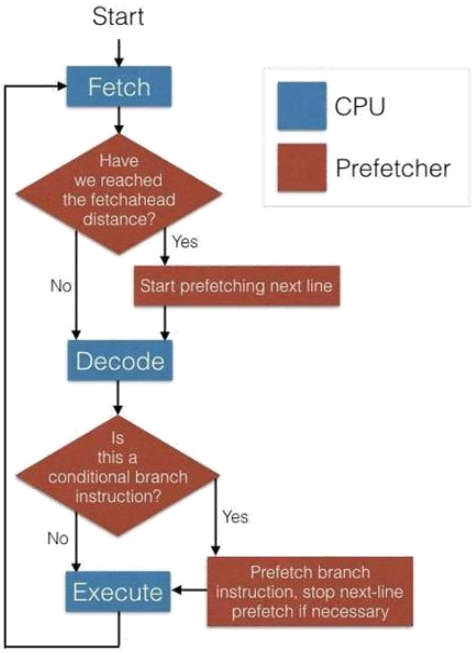


Figure 3: Wrong Path algorithm

2.2 Data Prefetching

2.2.1. Next-N-Lines:
It works automatically by prefetching consecutive blocks which follow the block that missed in the cache. When a miss occurs in the L2 cache, the subsequent K blocks are prefetched to the L2 cache if they are not already there. We will refer to K as the degree of prefetching. As an example, if we have a miss in the L2 cache for line X, we will first send the request to the RAM. Then the RAM will service the request for line X and also prefetch lines X+1, X+2, and all subsequent lines until line X+K to the L2 if they are not already there. The main advantages of this scheme are that it reduces the number of read stalls, it is a simple technique compared with other schemes, and it works well for applications with high sequentially. However, the disadvantages are the it doesn't work as well for programs with frequent branching, it can be more expensive storage-wise, and a larger value of K tends to increase memory traffic and cache pollution because it prefetches data that may not be used if the program execution path is different from the instruction prefetch path[1][3][4][5][6][8][10][11].

3. METHODOLOGY
The prefetching simulator code was written in C++, and based on code written by the University of California San Diego. Because it is written in C++, it is cross-platform and can be compiled for any operating system. Pin generator was used to generate the trace files that were used in the simulators equations.

3.1 Data Prefetch Implementation

Our data prefetching implementation required little modification of the University of California San Diego's code, and we were able to use the trace files they provided. For data prefetching, we decided to implement the Next-N-Line prefetching algorithm, which is detailed in the next section.
3.1.1 Next-N-Line implementation
To implement data prefetching, we need to answer three main questions: what to prefetch, when to prefetch, and where to put the prefetched data. To answer the first question, if we need to prefetch a request for line X, then we will prefetch X+1, X+2, and all subsequent lines until we reach line X+k. Secondly, we prefetch when a miss occurs and if the subsequent K blocks are not already in the cache. To answer the question of "where", we will prefetch the data from memory and put them in the second level cache, assuming the data is not already there. This is shown in Figure 4 below.

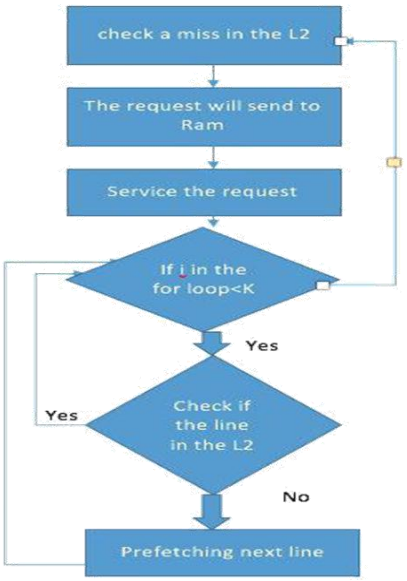


Figure 4: This diagram shows how the Next-N-Line data prefetching algorithm works.

4. INSTRUCTION PREFETCH IMPLEMENTATION
Our implementation of an instruction prefetching simulator required the following hardware to be simulated: the CPU, caches, the prefetcher, and for one of the algorithms, a target table

4.1 Trace Files

Our simulation required trace files in a different format than what were provided by the University of California San Diego. We generated our own trace files of various common programs using Pin, a dynamic binary instrumentation tool. Pin can be used to trace compiled programs[4][5], outputting information such as the address of the current instruction, whether or not the current instruction is a conditional branch, the fall-through address, the target address of conditional branches, and whether or not the conditional branch was taken. We used Pin to output this information for every instruction executed during the run of a program, and stored this information in text files in format shown in Figure 10:

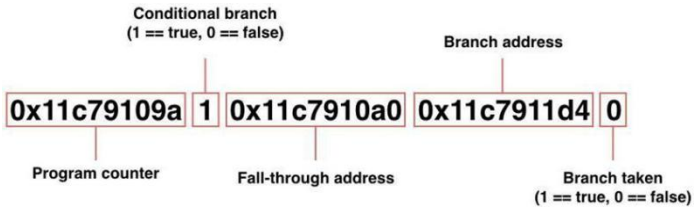


Figure 4:Format of Trace Files

4.2 Benchmarks

We have used three different benchmarks for testing data prefetching and four different benchmarks for testing instruction prefetching[1][11]. These benchmarks are described in Table

Trace	Description
ls	a Linux shell command that lists directory contents of files and directories.
g++	Gnu c++ compiler
grep	A linux shell command that searches the named input *FILE* s
rmdir	used to remove empty directories in Linux and other Unix-like operating systems

Table 1 : training Benchmarks

4. Simulation Results

The following figures show the results and analysis of our test on data prefetch.

Figure 13: Total runtime for no data prefetching and Next-N-Line data prefetching. Three different depths of prefetching were tested.

Figure 13 shows that the run time is reduced in all programs tested (g++, ls, and grep) when we use Next-N-Line data prefetching instead of no prefetching. We also see a difference when we change the depth of prefetching in that each time we increase K, the runtime is decreased in all three applications.

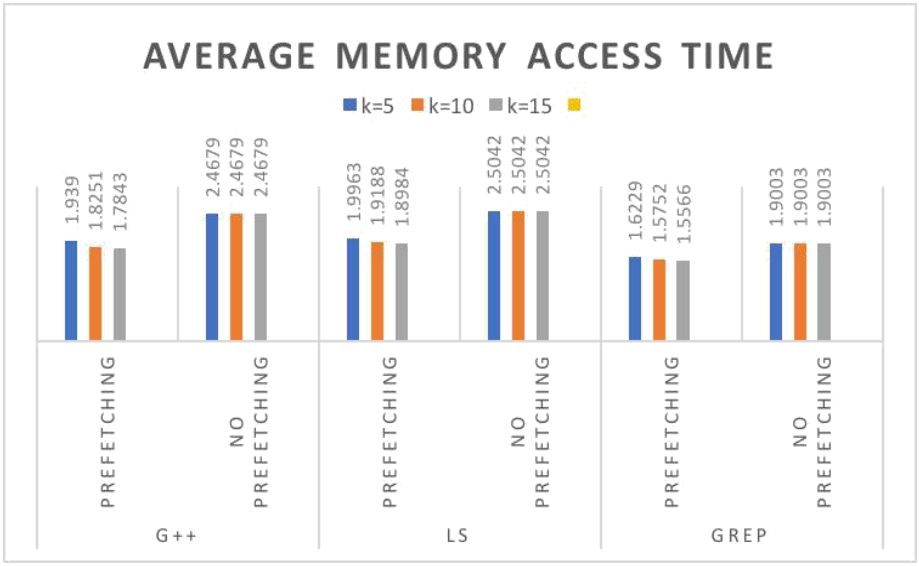


Figure 14: Average memory access time when comparing no data prefetching and Next-N-Line, and varying the depth of prefetching.

As Figure 14 shows, the average memory access time is reduced for all of the applications tested when we using Next-N-Line data prefetching compared with no prefetching. We also see a decrease in memory access time when we increase the depth of prefetching.

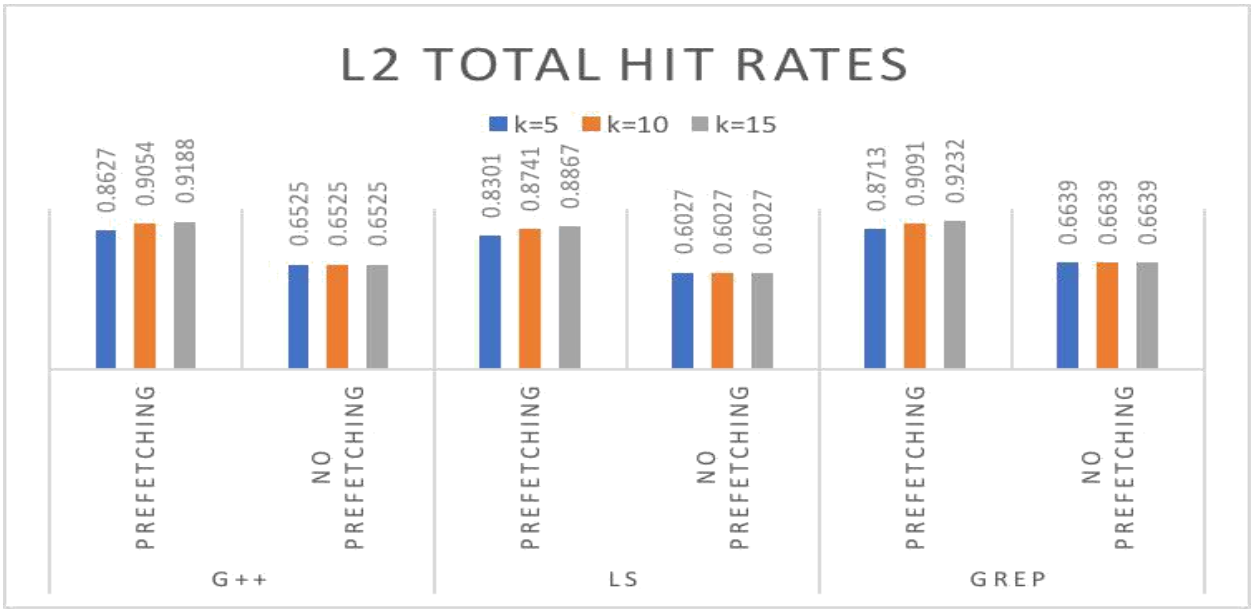


Figure 15: L2 total hit rates for no data prefetching and Next-N-Line prefetching with three different prefetching depths.

In Figure 15, the total hit rates show good improvement in all of the applications tested when we use Next-N-Line data prefetching instead of no prefetching. We also see a difference when we increase the depth of prefetching, in that each time we increase K, we increase the total hit rates in all of the programs tested. This is because our algorithm will prefetch the data to the L2.

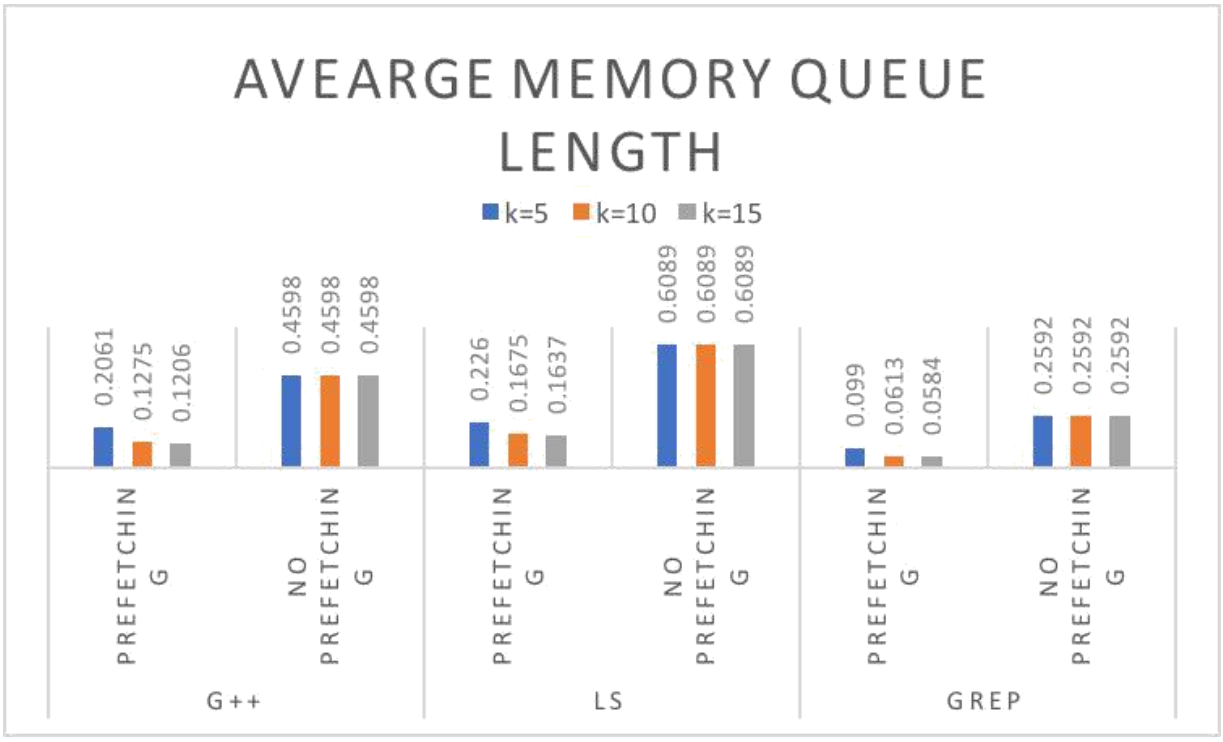


Figure 16:Average memory queue length for no data prefetching and Next-N-Line data prefetching with three different degrees of prefetching.

In Figure 16, the average memory queue length show a significant decrease for all programs tested when we implement Next-N-Line data prefetching instead of no prefetching. Each time we increase K, we could reduce the average memory queue length in all three applications.

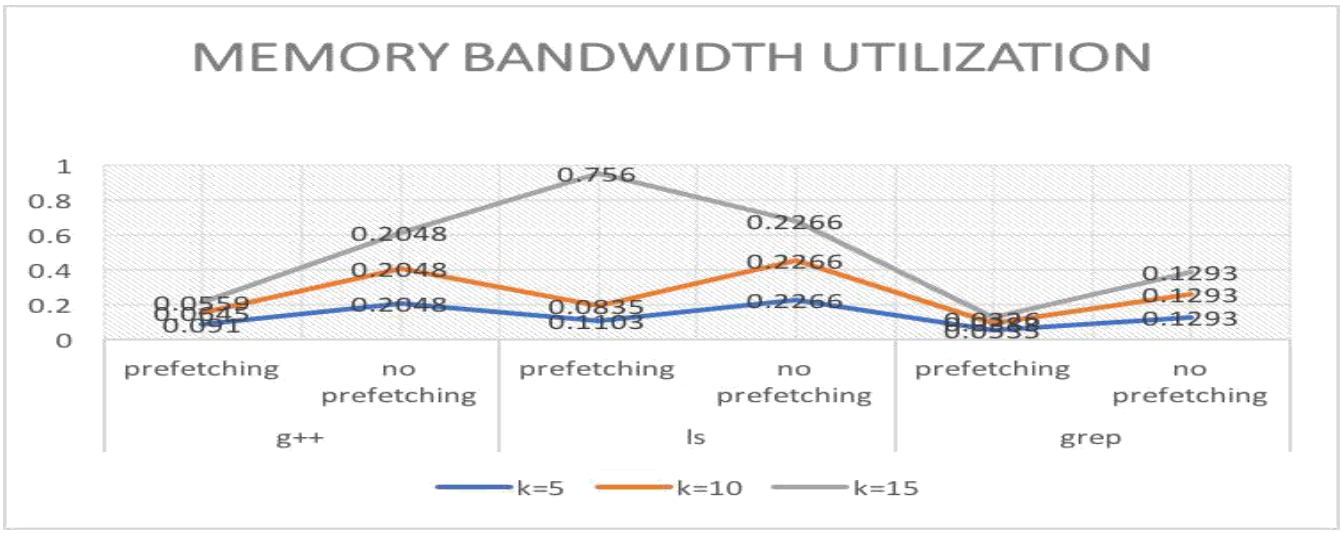


Figure 17 : Memory bandwidth utilization for no data prefetching and Next-N-Line data prefetching with three different degrees of prefetching.

In figure 17 the chart shows that how memory bandwidth utilization affected when the degree of data prefetching increased or decreased that means it is effective by the prefetching.

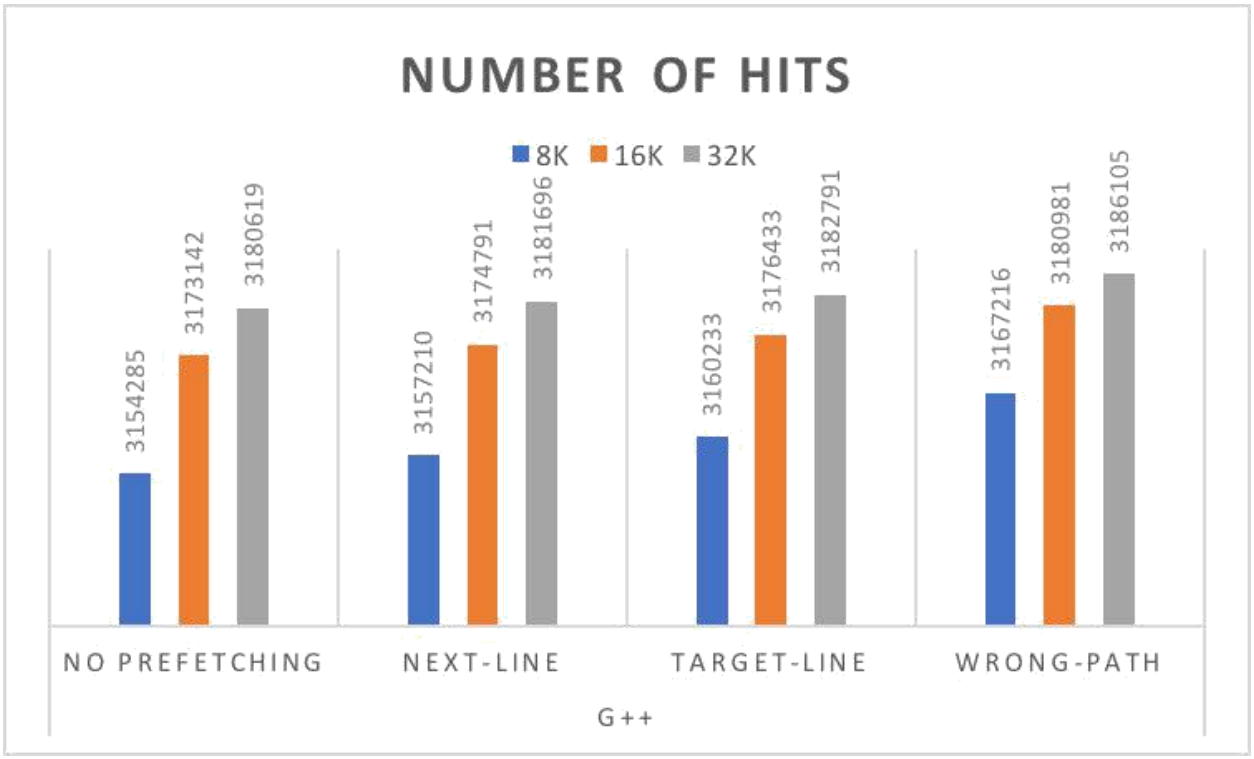


Figure 18 : Number of hits for three different instruction prefetching methods in three different cache sizes using the ls Linux command benchmark.

In the above graph we see a significant increase in the number of hits when we increase the size of the cache from 8K to 16K. We also see good improvement in the number of hits when the cache grows to be 32K.

5. Instruction Prefetching Test and Results

5.1 Parameters Tested

We have tested the following five parameters for our instruction prefetching implementation:

- Number of hits
- Number of misses
- Number of cycles
- Hit rates
- Miss rates

5.2 Cache configuration

For testing, we have implemented three different size of L1 cache with the following limitations:

- Cache size is 8K, 16K, 32K
- Direct mapped

- Block size is 32 bytes

5.3 Results Analysis:

The following charts show the number of hits when simulating different applications. Each chart shows how the number of hits change for a program when using different combinations of instruction prefetching algorithms and cache sizes.

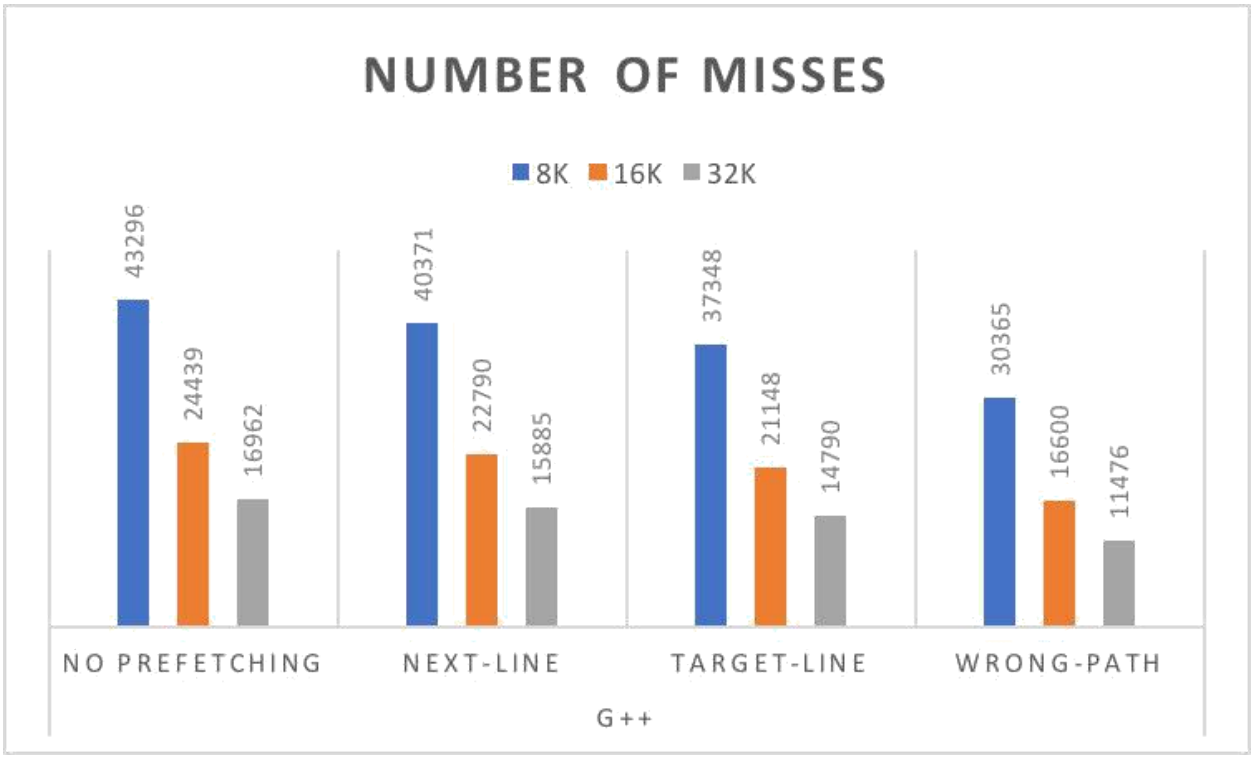


Figure 20: Number of misses of three different instruction prefetching methods in three different cache sizes using the g++ compiler benchmark.

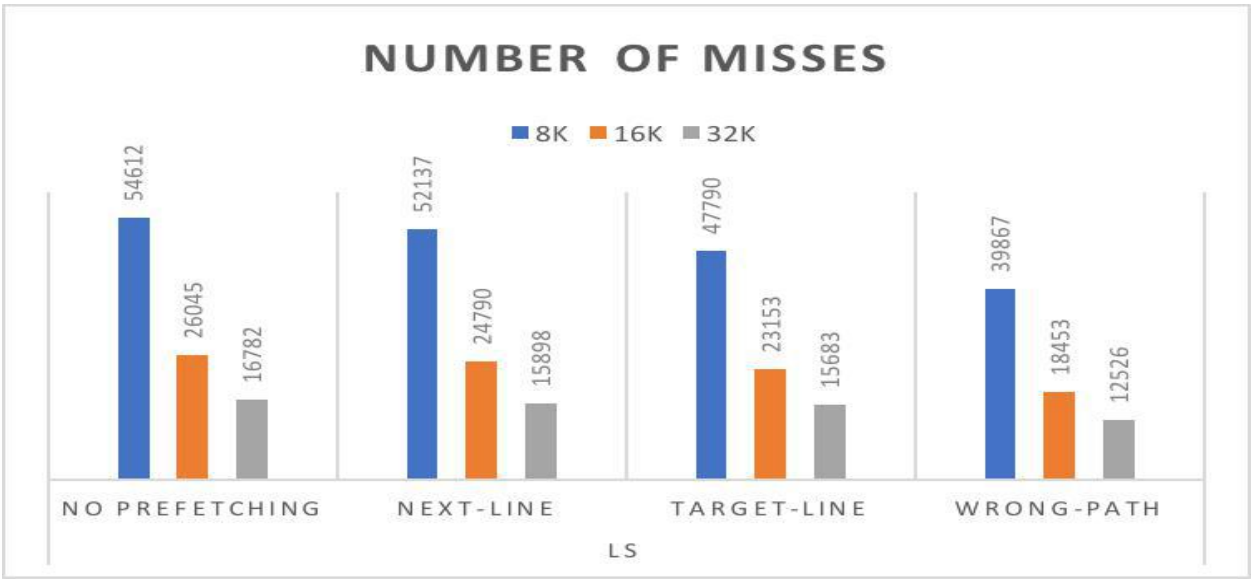


Figure 21: Number of misses for three different instruction prefetching methods in three different cache sizes using the ls Linux command benchmark.

The number of cycles for different applications are displayed in the following charts. As you can see, increasing the cache size reduces the number of misses since the cache can hold more instructions, and the different instruction prefetching algorithms affect the number of cycles as well.

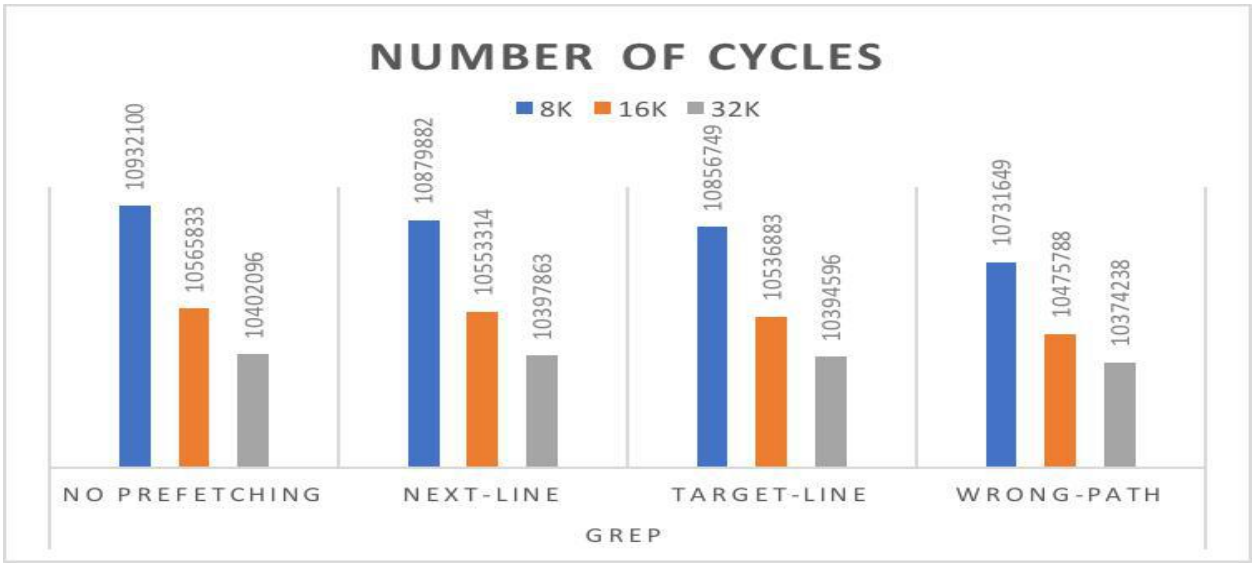


Figure 22: Number of cycles of three different instruction prefetching methods in three different cache sizes using grep Linux command benchmark.

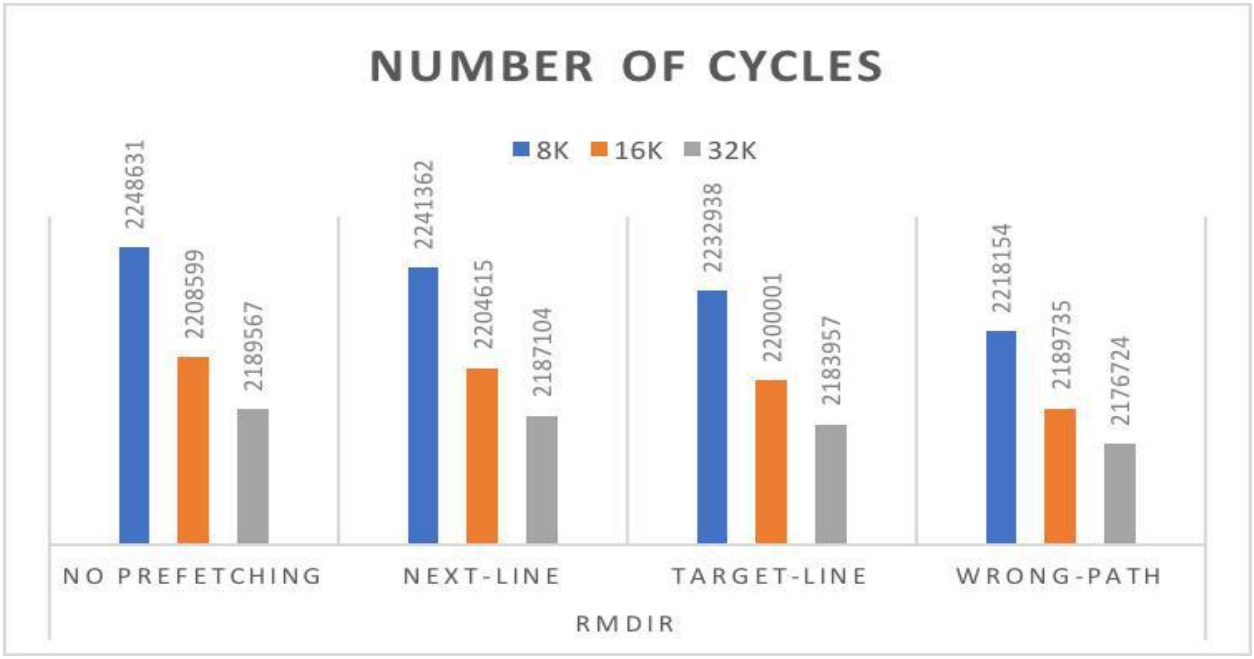


Figure 23: Number of cycles of three different instruction prefetching methods in three different cache sizes using rmdir Linux command benchmark.

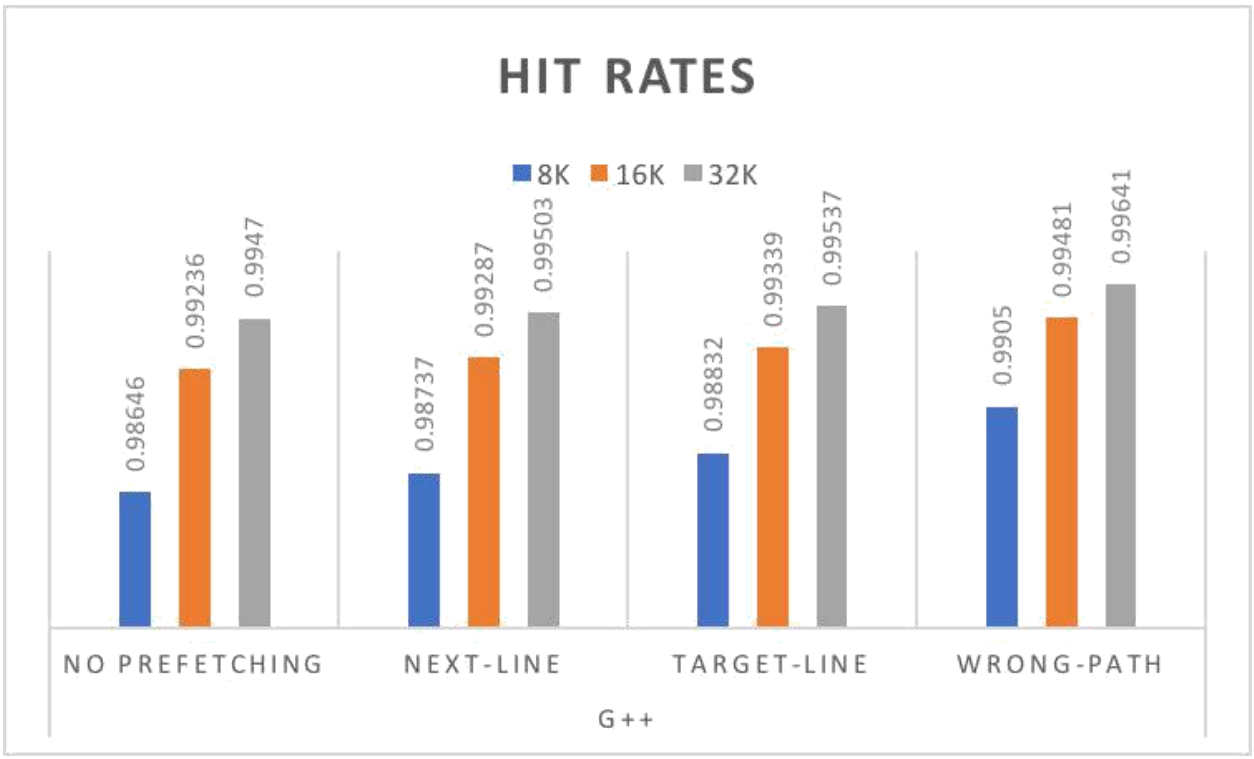


Figure 24: Hit rates of three different instruction prefetching methods in three different cache sizes using g++ compiler benchmark.

The hit rates and miss rates of each application are simply the number of hits divided by the total number of cache accesses, and the number of misses divided by the total number of cache accesses, respectively.

Figure 25: Hit rates of three different instruction prefetching methods in three different cache sizes using ls linux command benchmark.

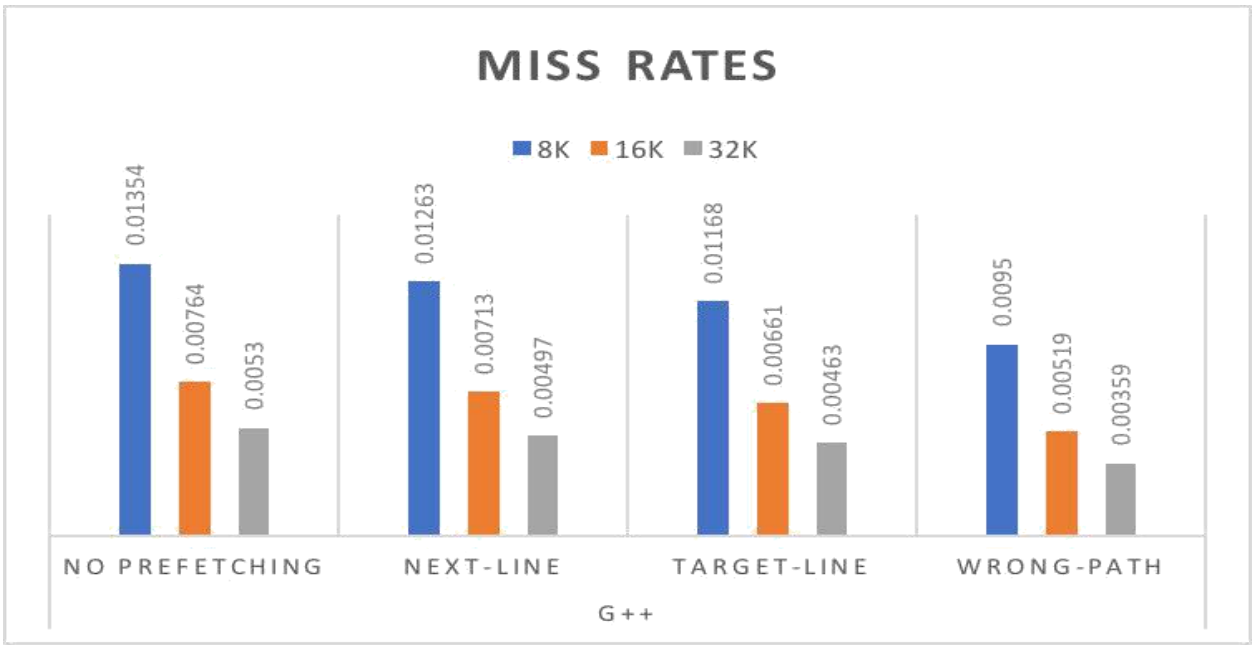


Figure 26: Miss rates of three different instruction prefetching methods in three different cache sizes using g++ compiler benchmark.

Figure 27: Miss rates of three different instruction prefetching methods in three different cache sizes using grep linux command benchmark.

6.Conclusion

In our paper, we have shown four different hardware prefetching algorithms. Three of these algorithms are instruction prefetching hardware algorithms, and one is a data prefetching algorithm. Our aim was to compare and test these algorithms after implementing them to show the best algorithm that produces the shortest latency and/or lowest cache miss rate. Also, we wanted to see the improvement in run time, miss rate, average memory access time, and hit rate when we implemented hardware prefetching compared with no prefetching.

If we were to continue with this work in future, we would combine the properties of Next-Line, Target-Line, and Wrong-Path instruction algorithms to create an optimized instruction algorithm that will improve the worst-case performance. We would also be interested in simulating both data prefetching and instruction prefetching together in one simulator.

References

[1] Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D. and Zelenka, J., 1995, December. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles* (pp. 79-95).

[2] Huizinga, D.M. and Desai, S., 2000, March. Implementation of informed prefetching and caching in linux. In *Proceedings International Conference on Information Technology: Coding and Computing (Cat. No. PR00540)* (pp. 443-448). IEEE.

[3] Zelenka, J., 1995. Informed prefetching and caching. In *Proceedings of the*.

[4] Tomkins, A., Patterson, R.H. and Gibson, G., 1997. Informed multi-process prefetching and caching. *ACM SIGMETRICS Performance Evaluation Review*, *25*(1), pp.100-114.

[5] Pierce, J. and Mudge, T., 1996, December. Wrong-path instruction prefetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29* (pp. 165-175). IEEE.

[6] *Luk, C.K. and Mowry, T.C., 1998, December. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture (pp. 182-193). IEEE.*

[7] Dahlgren, F., Dubois, M. and Stenstrom, P., 1993, August. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *1993 International Conference on Parallel Processing-ICPP'93* (Vol. 1, pp. 56-63). IEEE.

[8] Lipasti, M.H., Schmidt, W.J., Kunkel, S.R. and Roediger, R.R., 1995, December. SPAID: Software prefetching in pointer-and call-intensive environments. In *Proceedings of the 28th annual international symposium on Microarchitecture* (pp. 231-236). IEEE.

[9] Cherng, C. and Ladner, R.E., 2005. Cache efficient simple dynamic programming. In *Discrete Mathematics and Theoretical Computer Science* (pp. 49-58). Discrete Mathematics and Theoretical Computer Science.

[10] Tang, Y., You, R., Kan, H., Tithi, J.J., Ganapathi, P. and Chowdhury, R.A., 2015, January. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 205-214).

[11] Thomas H.. Cormen, Leiserson, C.E., Rivest, R.L. and Stein, C., 2001. *Introduction to algorithms* (Vol. 5, pp. 2-2). Cambridge: MIT press.

Mustafa Asaad Hasan Alkhafaji received the B.S degree in computer science from University of Thi-Qar, Nasiriyah, Iraq in 2011 and the M.S. degree in computer science from University of Colorado Denver, United States, in 2017. He is a member faculty at University of Thi-Qar in computer science and mathematics college. He published three papers before about IT and computer science technologies. He is studying PhD philosophy in computer science at University of Babylon, in IT College (Software Branch). His research interest and focused on Computer Networking, Data Mining, neural networks, Image Processing and Video Tracking.

Ali Hussein Lazem received the B.S degree in computer science from University of Thi-Qar, Nasiriyah, Iraq in 2010 and the M.S. degree in Computer Science from Troy University, United States, in 2017. He is a member faculty in University of Thi-Qar in Computer Center. His research interest and focused on AI, Machine Learning, neural networks (CNN,LSTM), Reinforcement Learning, Computer Vision and Deep Learning.

HAYDER A. JAWDHARI received the M.Sc. degree in computer science from ISTANBUL TECHNICAL University, TURKEY, in 2017. He is currently working toward the Ph.D. degree in the college of information technology, University of Babylon. He was Manager of computer center in AL-Qasim Green university from year 2017 to 2019. He is a Lecturer at AL Qasim Green University now. His research interests include the theory and applications of sparse representations, hyperspectral image compression and video processing.