Implementation and Comparison of Two Cache Prefetching Schemes

Yashwanth Katta
Electrical and Computer Engineering
University of Florida
Gainesville, Florida
yashwantkatta@ufl.edu

Avinash Ayalasomayajula
Electrical and Computer Engineering
University of Florida
Gainesville, Florida
ayalasomayajulal.a@ufl.edu

Neel Kanjaria
Electrical and Computer Engineering
University of Florida
Gainesville, Florida
neelkanjaria@ufl.edu

Dr. Sandip Ray
Electrical and Computer Engineering
University of Florida
Gainesville, Florida
sandip@ece.ufl.edu

Karthikeyan Rajasekaran
Electrical and Computer Engineering
University of Florida
Gainesville, Florida
krajasekaran@ufl.edu

***Abstract*—As technology advances, it has become a challenge to improve the performance of a system. Increasing processor speed sufficed this need to some extent, but this could not find balance with the memory access latency. So, to further enhance the performance of a system, a cache prefetching has been introduced. 'Cache Prefetching' is an efficient technique used by the computer processors to boost the execution performance. A lot of study has been conducted in determining which way of implementation is the best way to execute this theory and people came up with a lot of ideas. There were several techniques, both hardware and software, to implement this prefetching. In a cache, we could prefetch data and instruction. Data prefetch has been widely studied and several prefetching schemes have been successfully implemented. But as performance became a pressing issue and number of misses and hit rate became quite crucial, instruction prefetching was considered.**

This project focuses on implementing instruction prefetching schemes using 'SimpleScalar'[5] and thus comparing them. The novelty of the idea intrigued us in pursuing this area for our project where a new area can be studied. The instruction prefetching schemes, Hybrid Prefetching and Wrong Path Prefetching, are compared based on miss rate, hit rate and CPI.

Keywords—Cache Prefetching, memory access latency, Instruction, Hybrid and Wrong Path Prefetching, SimpleScalar, miss rate, hit rate and CPI.

I. Introduction

Till recent times, people have focused on improving the number of processors thus increasing the processor speed for better performance of a system. But this reached a saturation where the access to the memory became a challenge. Increasing the memory size was no longer consistent with the increase in the number of processors. As you can see in Figure 1, the gap has increased and thus shows that memory and processor speed are no longer in harmony. So, some new ways of improving the performance are to be studied. Cache Prefetching is one such technique to boost the execution performance of a system. The idea is to prefetch a cache block from a slower memory to a faster memory such that when a new line of code is being executed, the data or instruction needed has already been prefetched and there is no need to go to the lower order caches to get it because when initially a miss occurred, a block would have been fetched from the lower order caches and when doing so using the principle of temporal and spatial locality, some other blocks are also fetched. Thus, for the next line of code, the probability to use this prefetched block is pretty good and thus the extra access time wouldn't be needed.

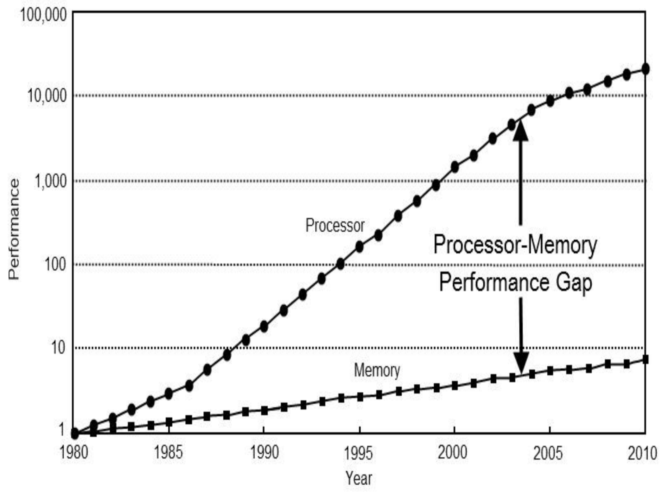


Figure 1: Processor vs Memory [1]

There are several ways of implementing 'Cache Prefetching'. Hardware and Software Prefetching[6] are two ways with several prefetching schemes to implement cache prefetching. Although there are many schemes, Hybrid and Wrong Path schemes are studied and compared. The reasons and explanation are explained further in the report.

A. *Motivation*

The modern processors are mostly superscalar where the CPI (Cycles per Instruction) for such processors is less than 1. But, for that, more than one functional unit are required and having so we must make sure that this inclusion gives us the effective CPI < 1 without degrading the performance. One way to achieve this is to reduce the number of cache misses because when there is a cache miss, there is some delay in the execution, hence increasing the CPI. So, reducing the probability of cache misses is one of the ways to optimize the performance. This can be done using ‘Prefetching’**.** In prefetching we can prefetch data and instruction. Data Prefetching has been successfully implemented and there are several established techniques.

But as the need for better performance increases, newer ways of tackling this issue had to be explored. Instruction prefetching is one such area that showed great promise in solving this issue to an extent. So, this was studied extensively. This field is a new approach and since there are very few promising schemes to implement, there are a still more theories that can be studied and improved over the others.

B. *Proposal*

Initially we made a proposal where we told to implement Hybrid Prefetching and Wrong Path Prefetching. These are instruction prefetching techniques implemented on hardware, here we use SimpleScalar [5]. We proposed to compare these two schemes based on cache pollution and miss penalty. Eventually, we wanted to establish the efficiency of each scheme over no prefetching scenario and over each other. So, throughout this report, it would be clearly explained about how the prefetching schemes have been implemented and finally a histogram comparing the cache parameters would be shown.

II. Prefetching Schemes

Prefetching can be both Hardware Prefetching and Software Prefetching. Both techniques have their pros and cons. In this project ‘Hardware Prefetching’ is implemented and under that we study about instruction prefetching. There are several instruction prefetching schemes of which Hybrid and Wrong Path prefetching are used for this project. To implement these, we must further implement next line and target line prefetching.

A. *Hybrid Prefetching*

Hybrid prefetch scheme combines two of the basic prefetching schemes namely ‘Next-Line Prefetching’ and ‘Target Line Prefetching’ [2]. This offers us a double protection against an instruction miss for a branch instruction.

a) NEXT LINE: Next line is one of the most basic prefetching schemes. In next line prefetching we prefetch the next sequential cache line to the cache line that is currently being fetched by the processor. This is an apt approach if we don’t consider branches.

b) TARGET LINE**:** Target prefetching removes the roadblock placed by next line prefetching for cache containing control instructions. So, we prefetch the based on bimodal or 2-bit saturating counter method. The bimodal or 2-bit saturating method is a state machine with four states and a counter of 2 bits. Branch prediction is based on which state is active and once the branch is evaluated the states change between ‘Weakly Not-Taken’, ‘Strongly Not-Taken’, ‘Weakly Taken’, ‘Strongly Taken’. Target Line is better over Next Line in the sense that it can implement branches and if the program flow is sequential. But for this to execute efficiently we need to execute the instruction at least once before the actual execution for the target buffer to store target branches. This proves to be exhausting in most of the scenarios. [2]

Hybrid Prefetching combines these two schemes and overcomes the branch limitation in Next Line and large buffer size in Target line. The performance is the combined gain of these two schemes. But, the on-chip area is quite expensive.

B. *Wrong Path Prefetching*

Wrong Path is quite different from all the above schemes in the sense that it prefetches on the simplest wrong path. It prefetches the target lines that are not taken in the hopes that they might be needed for the future correct predictions. It prefetches two cache lines. One is the sequential cache line to the cache line being fetched by the processor. The second cache line is the target of the branch instruction contained within the current cache line. There is no dependency for the fetching of the target of the control instruction. This scheme is most effective if the target branch is not taken. This is when we get to store this target, and which can later be of use. [2]
It is better in terms that very minimal hardware is required and the gap between the CPU and memory speed doesn’t matter as the instruction prefetched may not execute immediately unlike the previous ones. But there is a catch. In the worst-case scenario if none of the target branches that are stored are used, then this might prove to be inefficient.

III. Implementation

This section below describes the implementation methods for both hybrid and wrong path prefetching methods. The basic idea is to make changes to the code to introduce prefetching into the cache. SimpleScalar 3.0 tool was used for simulation of these prefetch methods. SimpleScalar consists of four processor simulators namely ‘Sim-Fast’, ‘Sim-Cache’, ‘Sim-profile’ and ‘Sim-outorder’. Sim-Cache is a cache simulator that simulates a memory. It also generates one- and two-level hierarchy statistics and profiles. Sim-outorder is an Out-of-Order processor timing simulation. It combines both the ‘Sim-Cache’ simulator with branch prediction for an out-of-order issue. All the following prefetch methods have been implemented only for the level-1 instruction cache

A. *Next Line Prefetching*

Next line prefetching is done using the ‘Sim-Cache’ as no branch prediction is required. We introduce a variable called “prefetch_blk_count” in cache.c. This allows the user to tell the simulator the number of sequential blocks to be prefetched by the simulator. This introduces a new parameter ‘<prefetchblkcount> to the cache configuration parameters that are described by the user. We also introduce a function called “cache_prefetch_blk”, this function is accessed when a cache is accessed to check for the cache line being requested by the processor. On a hit nothing happens. But when there is a miss and the cache accesses the main memory for the cache line, the ‘cache_prefetch_block’ function fetches the next sequential cache line to the current cache line being fetched. [2]

B. *Hybrid Prefetching*

SimpleScalar 3.0 provides us with the bimodal or 2-bit saturating counter branch prediction scheme. Sim-outorder allows us to combine both the upgraded cache file with next line prefetching and bimodal branch prediction to achieve hybrid prefetching. We make bimodal branch prediction as the default branch prediction scheme, which returns the target of any conditional instruction in the cache line being fetched based on which state the bimodal state machine is in. Along with this the ‘cache_prefetch_block’ returns the next sequential address on a miss. [2]

C. *Wrong Path Prefetching*

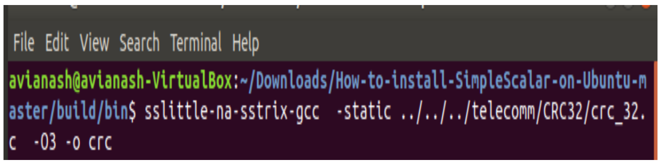
SimpleScalar 3.0 provides us with a ‘branch taken’ branch prediction scheme. Sim-outorder allows us to combine both the upgraded cache file with next line prefetching and ‘branch taken’ branch prediction to achieve hybrid prefetching. We make branch taken branch prediction as the default branch prediction scheme, which returns the target of any control instruction present within the cache line that is being fetched. Along with this the ‘cache_prefetch_block’ returns the next sequential address on a miss. [2]

Introducing prefetching into the cache code, which was already provided in SimpleScalar, did show the advantages over no prefetching execution. For all the above schemes a standard procedure has been followed in generating the output file and executing this output file using a benchmark.

IV. Compilation Procedure

The following steps have been followed for the execution of the prefetching code for a specific benchmark.

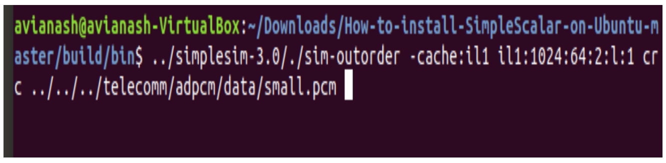
1) Copy the updated “Cache.c”, “Cache.h”, “Bpred.c”, “Bpred.h”, “Sim-Cache.c”, “Sim-Bpred.c”, “Sim-outorder.c” files to the Simplesim-3.0 folder.
2) Then we “Make” these files to check errors.
3) We get an output saying “My work is done here…”
4) We open terminal through build/bin file present within the SimpleScalar installation file.
5) We then compile the benchmarks here using “sslittle-na-sstrix-gcc” as follows for the ‘Telecomm_CRC32’ benchmark as follows



```
avianash@avianash-VirtualBox:~/Downloads/How-to-install-SimpleScalar-on-Ubuntu-m
aster/build/bin$ sslittle-na-sstrix-gcc  -static ../../../telecomm/CRC32/crc_32.
c  -O3 -o crc
```

Figure 2 : Compilation Command for Telecomm_CRC32 Benchmarks

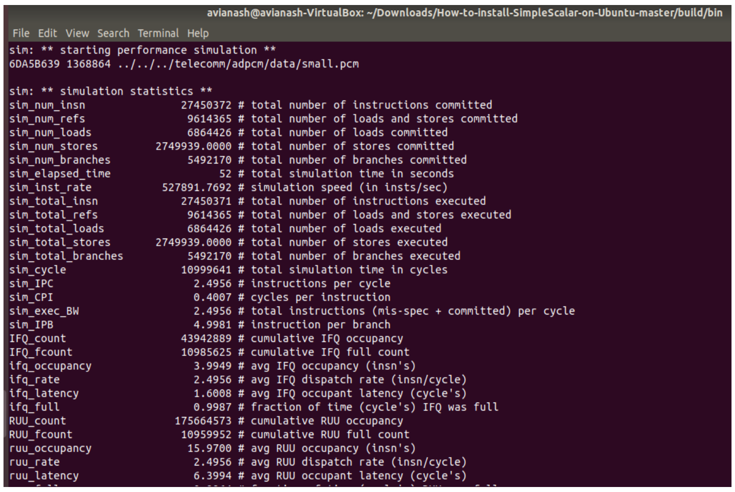
6) Then we run the simulation through “Sim-outorder” simulator for a small data of inputs to the ‘Telecomm_CRC32’ benchmark and a level instruction cache with 1024 sets, 64bit cache line, an associativity of 2 and prefetch block count of 1 as follows.



```
avianash@avianash-VirtualBox:~/Downloads/How-to-install-SimpleScalar-on-Ubuntu-m
aster/build/bin$ ../simplesim-3.0/./sim-outorder -cache:il1 il1:1024:64:2:l:1 cr
c ../../../telecomm/adpcm/data/small.pcm
```

Figure 3 :Simulation Command

We get an output as follows :


avianash@avianash-VirtualBox: ~/Downloads/How-to-install-SimpleScalar-on-Ubuntu-master/build/bin
File Edit View Search Terminal Help

```
sim: ** starting performance simulation **
6DA5B639 1368864 ../../../telecomm/adpcm/data/small.pcm

sim: ** simulation statistics **
sim_num_insn                 27450372 # total number of instructions committed
sim_num_refs                  9614365 # total number of loads and stores committed
sim_num_loads                 6864426 # total number of loads committed
sim_num_stores           2749939.0000 # total number of stores committed
sim_num_branches              5492170 # total number of branches committed
sim_elapsed_time                   52 # total simulation time in seconds
sim_inst_rate             527891.7692 # simulation speed (in insts/sec)
sim_total_insn               27450371 # total number of instructions executed
sim_total_refs                9614365 # total number of loads and stores executed
sim_total_loads               6864426 # total number of loads executed
sim_total_stores         2749939.0000 # total number of stores executed
sim_total_branches            5492170 # total number of branches executed
sim_cycle                    10999641 # total simulation time in cycles
sim_IPC                        2.4956 # instructions per cycle
sim_CPI                        0.4007 # cycles per instruction
sim_exec_BW                    2.4956 # total instructions (mis-spec + committed) per cycle
sim_IPB                        4.9981 # instruction per branch
IFQ_count                    43942889 # cumulative IFQ occupancy
IFQ_fcount                   10985625 # cumulative IFQ full count
ifq_occupancy                  3.9949 # avg IFQ occupancy (insn's)
ifq_rate                       2.4956 # avg IFQ dispatch rate (insn/cycle)
ifq_latency                    1.6008 # avg IFQ occupant latency (cycle's)
ifq_full                       0.9987 # fraction of time (cycle's) IFQ was full
RUU_count                   175664573 # cumulative RUU occupancy
RUU_fcount                   10959952 # cumulative RUU full count
ruu_occupancy                 15.9700 # avg RUU occupancy (insn's)
ruu_rate                       2.4956 # avg RUU dispatch rate (insn/cycle)
ruu_latency                    6.3994 # avg RUU occupant latency (cycle's)
```

Figure 4 :Output File for Telecomm_CRC32 Benchmark

V. Results

The main aim of this project is to compare the Hybrid Prefetching and Wrong Path Prefetching based on various cache parameters. Since we are implementing instruction prefetching schemes, we would be looking at the instruction level cache (IL1 cache). Once the entire sim-cache.c file or any corresponding file is run, it would generate an output file which was then tested on various benchmarks to estimate the performance improvement as a result of using that specific prefetching scheme.

Let us look at the following Figures, which show the cache parameter comparison for different benchmarks. The three benchmarks used are NETWORK_PATRICIA, SECURITY_SHA and TELECOM_CRC32

A. *Comparison of Hit Rate*

Figure 5 shows the Hit rate comparison for the prefetching schemes on different benchmarks.

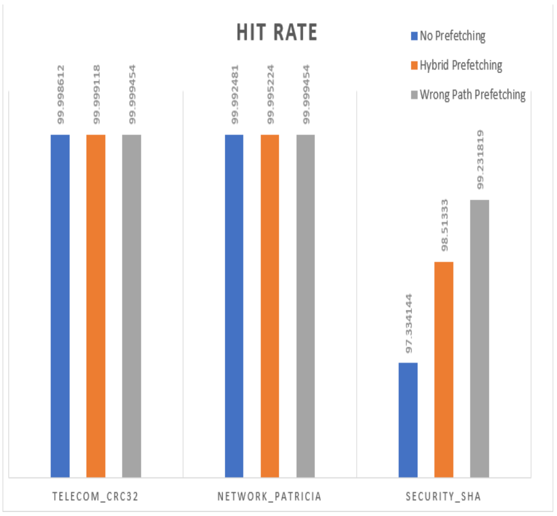


Figure 5 : Hit Rate Comparision

As we can see from the above histogram, there is an increase in the hit rate as a result of using the prefetching schemes. From the benchmarks Telecom_CRC32 and Network_Patricia a huge difference is not seen as there intial hit rates are better over the third benchmark. But there is an improvement over the no prefetching scenario but it is minimal. But when we consider the benchmark [3] Security_Sha, we can see a substantial increase in the hit rate from no prefetching to Hybrid to Wrong Path Scenario. It clearly shows that using prefetching has boosted the performance of the application in terms of hit rate.

On a whole, it is evident that the hit rate increases from No Prefetch to Hybrid to Wrong Path, with Security_Sha showing the largest improvement.

B. *Comparison of Miss Rate*

As we see in the above case for the hit rate, there was an improvement. Now cache misses become crucial in determining the system performance. The below Figure 6 describes the variation of miss rate for the three different benchamrks on using the prefetching schemes.

From the histogram it is quite evident that the number of misses have reduced substantially from No Prefetch to Wrong Path Prefetch with Wrong Path Prefetch having the least number of misses followed by Hybrid and then No Prefetch. Even here we see that the first two benchmarks have very few misses without any prefetch. But once we introduce Hybrid Prefetching, there is approximately 50% reduction in the miss rate. When we use Wrong Path, it reduces the miss rate by another 50% over the Hybrid Prefetching scheme.

For the third benchmark [3], Security_Sha, we can see that the intial miss rate is quite high before prefetching, but once we use prefetch there is clear improvement in the miss rate.

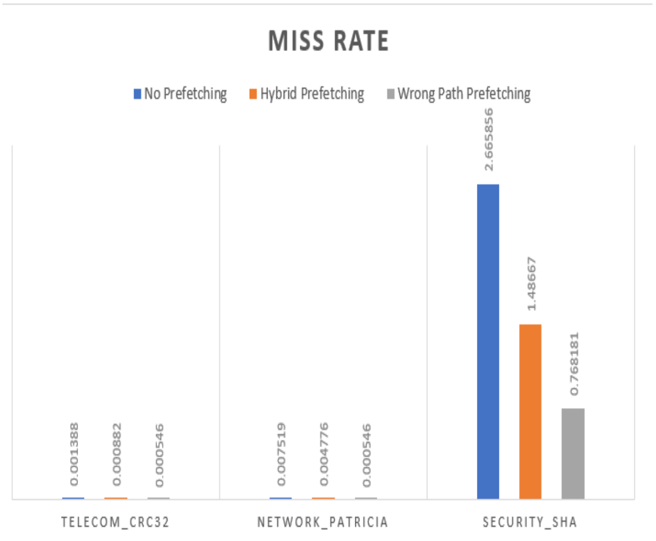


Figure 6 : Miss Rate Comparision

From comparing miss rate and hit rate we see that cache prefetching has helped to improve the performance of the system in terms of higher number of cache hits and lower number of cahce misses. Let us now compare one last paramter.

C. *Comparison of CPI (Cycles per Instruction)*

Figure 7 shows the CPI comparison for the prefetching schemes on different benchmarks.

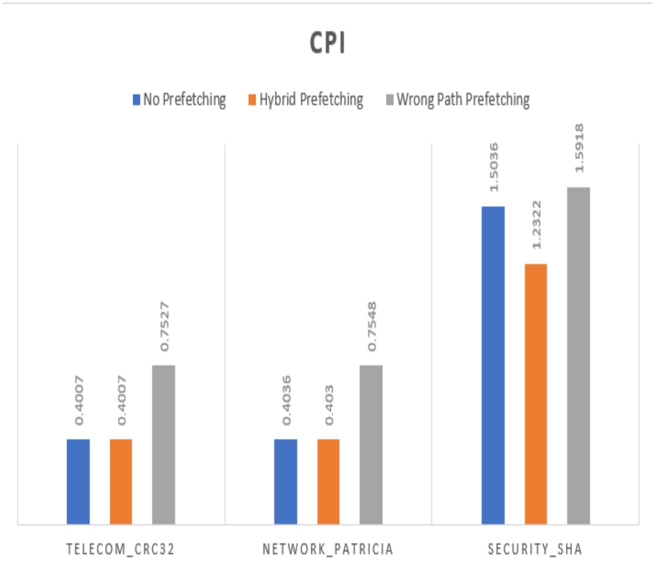


Figure 7 : Comparision of CPI (Cycles per Instruction)

Here we see that CPI for Wrong Path Prefetching is more than the other schemes. On the other hand Hybrid prefetching has a CPI of almost same as no prefetch condition in the first two benchmarks but when we move to the third one, we see a change. The CPI for Hybrid is lesser compared to both No Prefetch and Wrong Path. But Wrong Path has a CPI a bit more compared to the other two.

This case shows that prefetching can be very much useful in boosting the system performance in a few benchmarks but also that there are a few applications where having no prefetching proves to be efficient.

VI. Proposal vs Implementation

Hybrid Prefetching and Wrong Path Prefetching have been successfully implemented using SimpleScalar. The comparison of the two prefetching schemes over each other and with No Prefetching scenario has been established. In the proposal, we promised to implement Hybrid and Wrong Path Prefetching using SimpleScalar. It is now been fulfilled and is quite evident from the results that have been provided with this report. On a whole, we implemented Next Line Prefetching and Target Line Prefetching. Now using these two and merging them we implemented Hybrid Prefetching. Further using Next Line and Branch Prediction we implemented Wrong Path Prefetching. To this point we met the requirements stated in our proposal.

Comparison of cache parameter needs some clarification. We stated that we would be comparing cache pollution and miss penalty. When the benchmarks have been tested with the code (after prefetching), SimpleScalar's output file showed the hit rate, miss rate and CPI for IL1 cache. Unfortunately, it didn't provide us with the other parameters. But cache pollution means that the cache has been prefetched either a bit early or late than at the required time. So, in this case the cache must be fetched again thus wasting some clock cycles. This can be seen from the CPI comparison.
Also, for miss penalty it depends on the kind of program code the user is running. Estimating miss penalty is quite tricky but it is okay even if the miss penalty is not shown in this case. These prefetching schemes have been implemented to compare the cache parameters for a benchmark. Comparing the miss rate, hit rate and CPI suffices the need to evaluate system performance.

VII. Conclusion

Instruction Prefetching for a cache has been successfully implemented using SimpleScalar. The cache parameters – Miss rate, hit rate and CPI have been compared for different benchmarks using Hybrid Prefetching, Wrong Path Prefetching and no Prefetch in the cache. Prefetching instruction has a lot of approaches, but they are not efficient on all kinds of benchmarks. As, we can see from our results, for the benchmark Security_Sha, the prefetching proved to be of significance. For the other benchmarks the improvement was very minimal. Every scheme is better than the other in some application. So, we can conclude that the results are quite promising for the used benchmarks and the comparison of the cache prefetching schemes ended in providing some valuable data that led to deduce the performance.

VIII. Future Extensions

We could explore more prefetching schemes under instruction prefetching. These comparisons have been done for a few benchmarks. There are benchmarks where these prove completely inefficient. For such cases a challenge arises for us to develop new schemes or at the best develop a generic scheme that could be applied to most of the applications.

Acknowledgment

It's our privilege to thank all the people who contributed to making the commencement of this project a reality. It was a great opportunity for learning and educational development.

We would like to use this opportunity to express our deepest gratitude and special thanks to our project mentor and guide Dr. Sandip Ray, Professor, Electrical and Computer Engineering Department, University of Florida, Gainesville, for the effort taken to provide us with this opportunity and for his guidance throughout this project.

We would also like to express our gratitude to Mr. Karthikeyan Rajasekaran, Teaching Assistant, Computer Architecture Course, Electrical and Computer Engineering Department, University of Florida, Gainesville, for his continuous support and guidance in making this project a success.
Finally, we would like to thank everyone who had a vital role in this commencement of making our visualized concepts a dream come true.

References

[1] Computer Architecture: A Quantitative Approach by John L. Hennessy, David A. Patterson, Andrea C. Arpaci-DusseauJ. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] http://web.engr.oregonstate.edu/~benl/Projects/prefetch_report/final.html

[3] http://vhosts.eecs.umich.edu/mibench/source.html

[4] http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/lab1.htm

[5] http://www.simplescalar.com/docs/hack_guide_v2.pdf

[6] http://home.eng.iastate.edu/~zzhang/cpre581/lectures/Lecture17-1p.pdf

[7] https://en.wikipedia.org/wiki/Cache_prefetching