ECE 563 - Computer Architecture I

Implementation Of Hardware Prefetching Schemes

Final Project Report

By-
Ajinkya Padwad
Anirudh Balaji
Tejas Rajput

Guided by-
Dr. Nagmeh Karimi

Rutgers University

Department of Electrical and Computer Engineering

CONTENTS

Abstract

To enhance the instruction execution capabilities of a superscalar architecture, the aim is to attain a CPI of less than 1. Despite advancements in the recent processor performances, there has been little improvement over ability to issue instruction at a rate optimal to the performance. This has led to a lot of unused bandwidth and the processor being idle. Recent developments have shown a technique of fetching instruction or data from the memory in advance, before the processor needs it. This technique is called prefetching. It enables predicting the next likely instruction/data to be fetched and bringing that to the cache blocks.

This project discusses three major hardware prefetching schemes for instruction and data -

1. Long Cache Line
2. Next Line
3. Stride

The performance for two-way, four-way and fully-associative mapping has been analysed for the Alpha architecture using the SimpleScalar simulator tool at Ubuntu environment. Four benchmarks have been employed for the analyses - GCC, Compress95, ANAGRAM and GO. The results thus compiled show a significant improvement at number of cache misses and the overall CPI.

Problem Statement

A superscalar processor's goal is to have a CPI of less than 1. To do this, more than one functional unit (physical or virtual) is required. However, without enough instructions to execute, the superscalar processor's performance will be limited regardless of how fast it can issue and execute instructions.

To decrease the frequency of cache misses in superscalar processors, prefetching could be used to bring in data from the memory into the instruction cache before the processor needs it. There are two different types of prefetching, instruction and data prefetching.

In this project, you should simulate the system using SimpleScalar or any simulator you feel comfortable. Then you may want to choose some prefetching algorithms to compare. You can refer to following algorithms, but it is always good to find you own ones:

- Long cache line
- Next Line Prefetching
- Target Line Prefetching
- Hybrid Scheme
- Wrong Path Prefetching

You can use the benchmarks provided, but you are welcome to find other benchmarks which are suitable for your comparison.

Introduction

Reducing the cycles per instruction in a superscalar processor is essential to speed up the process and bring down the cache misses. How can we do that? We can do this by reducing the necessity to go to the main memory and bring the data. Instead, we use what is called as a cache, which contains data previously used or the data that will be needed in subsequent cycles. However, even this can cause misses. Therefore, prefetching is used to reduce cache misses. Prefetching is the process that brings data into the cache from the memory even before the processor needs it. There are two different types of prefetching: instruction and data prefetching.

1. Instruction prefetch. Instruction prefetch fetches instruction lines from memory to cache. This reduces the number of instruction misses and increases instruction issue rate.
2. Data prefetch. Data prefetch reduces data cache misses by exploiting the program data access pattern. This pattern can be done with the executable by the compiler.

Another method of classifying prefetching is through the medium of prefetch: Software prefetching and Hardware Prefetching. Software prefetching requires editing in the program source code or assembly line program code. Hence, it is much more difficult to implement. Therefore, this project mainly deals with hardware prefetching schemes. Hardware prefetching uses access patterns and storage space to perform prefetch operations.

Hardware Instruction Prefetching Schemes

In this project, there are three hardware prefetching schemes used.

1. Long Cache Line Prefetching
2. Next Line Prefetching
3. Stride Prefetching

While the first two are predominantly used as instruction prefetching schemes, the third one is a data prefetching scheme. However, these can be used alternatively as well, but the results will not be as efficient as they were for what they were originally programmed for.

SimpleScalar Simulator

SimpleScalar is powerful software infrastructure, written in C programming language which is used to model applications for the analysis of program performance, micro architectural modelling, and hardware-software verification. Basically it is a virtual computer which models CPU, cache and memory hierarchy. Simplescalar tools can be used to build applications which can simulate real programs running on a range of modern processors and systems. In addition to simulators, the SimpleScalar tool set includes performance visualization tools, statistical analysis resources, and debug and verification infrastructure.

SimpleScalar can emulate ALPHA, PISA, ARM and x86 instruction sets. ALPHA was used for implementation of this project. Since it is a Linux based software, UBUNTU was installed on windows PC using VMware and then SimpleScalar tool was installed on linux platform. The tool set itself consists of a collection of microarchitecture simulators that emulate the microprocessor at different levels of detail .The simulators that we used are sim-cache and sim-outorder

Opening SimpleScalar in Ubuntu

1. Install VMware workstation on windows.
2. Create a virtual machine and install Ubuntu as operating system.
3. Download the simplescalar tool and related files from www.simplescalar.com on Ubuntu.
4. To install and open simplescalar tools refer this video :-
   https://www.youtube.com/watch?v=kNJHX7vyKs4

Running Benchmarks

To test the performance, various parameters are to be analysed at SimpleScalar. We have employed four standard benchmarks for the analysis.

Output can be tested in two ways:

1. Default benchmark commands (with default cache configuration)
2. Benchmark command with user specified cache configuration

   cache configuration syntax :

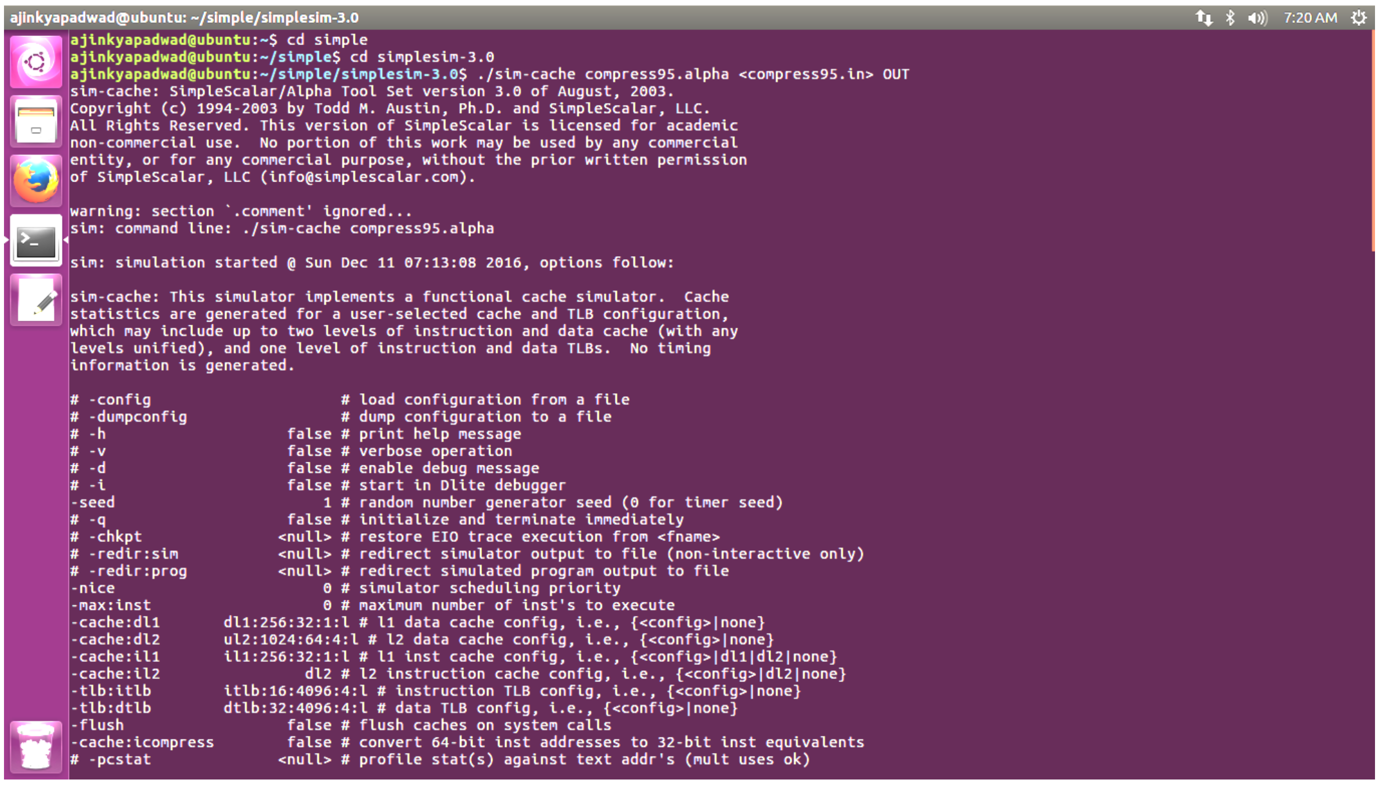
   <cache name> : <no. of sets> : <block size> : <associativity> : <replacement>

   e.g.- il1:4096:32:1:l

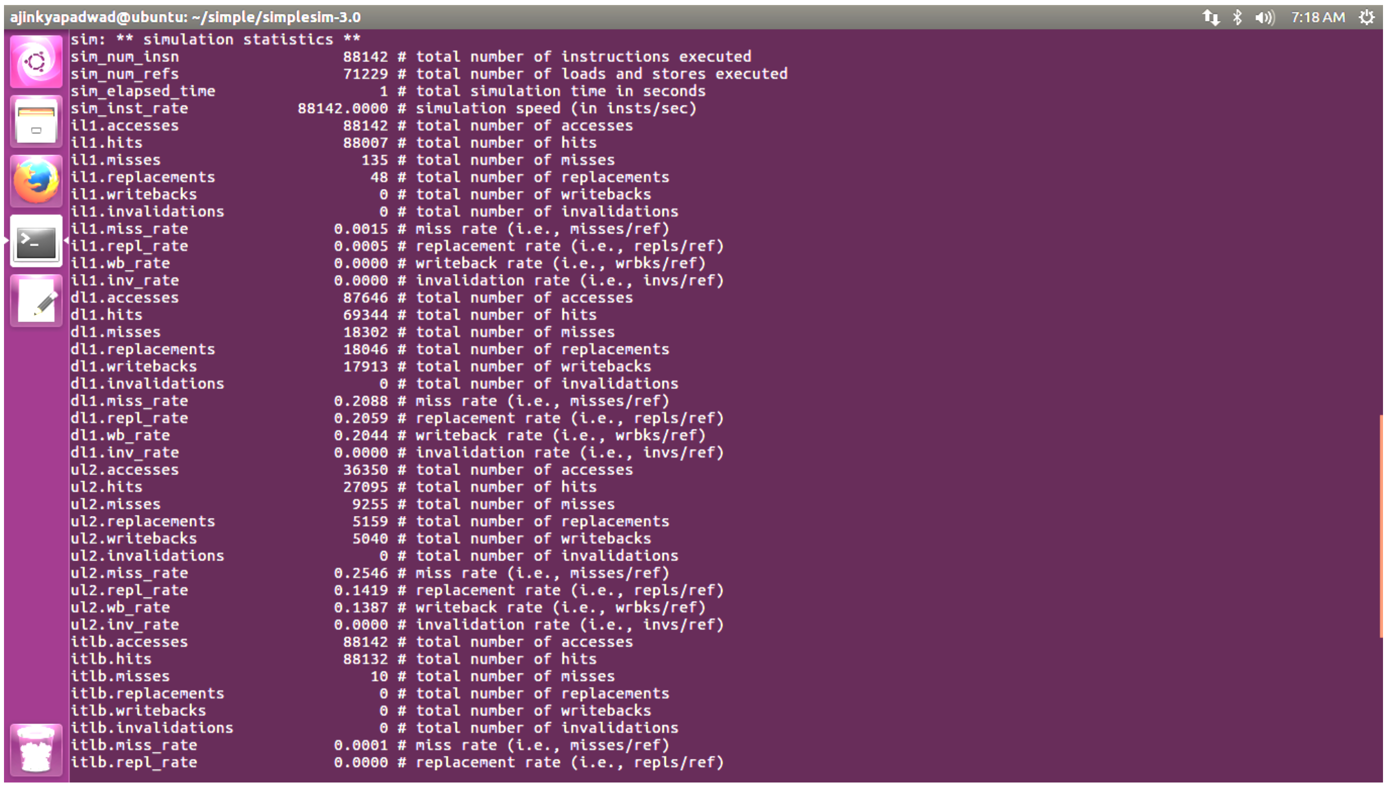
The commands for running the four benchmarks follow :

```
//GCC default benchmark command
        ./sim-cache cc1.alpha -O 1stmt.i

// GCC command with cache config for 4-way associativity
        ./sim-cache -cache:il1 il1:4096:32:4:l cc1.alpha -O 1stmt.i


// ANAGRAM default benchmark command
        ./sim-cache anagram.alpha words <anagram.in> OUT

// Compress95 default benchmark command
        ./sim-cache compress95.alpha <compress95.in> OUT

// GO default benchmark command
        ./sim-cache go.alpha 50 9 2stone9.in > OUT
```

Benchmark Outputs



```
ajinkyapadwad@ubuntu:~$ cd simple
ajinkyapadwad@ubuntu:~/simple$ cd simplesim-3.0
ajinkyapadwad@ubuntu:~/simple/simplesim-3.0$ ./sim-cache compress95.alpha <compress95.in> OUT
sim-cache: SimpleScalar/Alpha Tool Set version 3.0 of August, 2003.
Copyright (c) 1994-2003 by Todd M. Austin, Ph.D. and SimpleScalar, LLC.
All Rights Reserved. This version of SimpleScalar is licensed for academic
non-commercial use.  No portion of this work may be used by any commercial
entity, or for any commercial purpose, without the prior written permission
of SimpleScalar, LLC (info@simplescalar.com).

warning: section `.comment' ignored...
sim: command line: ./sim-cache compress95.alpha

sim: simulation started @ Sun Dec 11 07:13:08 2016, options follow:

sim-cache: This simulator implements a functional cache simulator.  Cache
statistics are generated for a user-selected cache and TLB configuration,
which may include up to two levels of instruction and data cache (with any
levels unified), and one level of instruction and data TLBs.  No timing
information is generated.

# -config                     # load configuration from a file
# -dumpconfig                 # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -d                    false # enable debug message
# -i                    false # start in Dlite debugger
-seed                       1 # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately
# -chkpt               <null> # restore EIO trace execution from <fname>
# -redir:sim           <null> # redirect simulator output to file (non-interactive only)
# -redir:prog          <null> # redirect simulated program output to file
-nice                       0 # simulator scheduling priority
-max:inst                   0 # maximum number of inst's to execute
-cache:dl1       dl1:256:32:1:l # l1 data cache config, i.e., {<config>|none}
-cache:dl2       ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:il1       il1:256:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il2                dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-tlb:itlb        itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb        dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-flush                  false # flush caches on system calls
-cache:icompress        false # convert 64-bit inst addresses to 32-bit inst equivalents
# -pcstat              <null> # profile stat(s) against text addr's (mult uses ok)
```



```
sim: ** simulation statistics **
sim_num_insn                  88142 # total number of instructions executed
sim_num_refs                  71229 # total number of loads and stores executed
sim_elapsed_time                  1 # total simulation time in seconds
sim_inst_rate            88142.0000 # simulation speed (in insts/sec)
il1.accesses                  88142 # total number of accesses
il1.hits                      88007 # total number of hits
il1.misses                      135 # total number of misses
il1.replacements                 48 # total number of replacements
il1.writebacks                    0 # total number of writebacks
il1.invalidations                 0 # total number of invalidations
il1.miss_rate                0.0015 # miss rate (i.e., misses/ref)
il1.repl_rate                0.0005 # replacement rate (i.e., repls/ref)
il1.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
dl1.accesses                  87646 # total number of accesses
dl1.hits                      69344 # total number of hits
dl1.misses                    18302 # total number of misses
dl1.replacements              18046 # total number of replacements
dl1.writebacks                17913 # total number of writebacks
dl1.invalidations                 0 # total number of invalidations
dl1.miss_rate                0.2088 # miss rate (i.e., misses/ref)
dl1.repl_rate                0.2059 # replacement rate (i.e., repls/ref)
dl1.wb_rate                  0.2044 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ul2.accesses                  36350 # total number of accesses
ul2.hits                      27095 # total number of hits
ul2.misses                     9255 # total number of misses
ul2.replacements               5159 # total number of replacements
ul2.writebacks                 5040 # total number of writebacks
ul2.invalidations                 0 # total number of invalidations
ul2.miss_rate                0.2546 # miss rate (i.e., misses/ref)
ul2.repl_rate                0.1419 # replacement rate (i.e., repls/ref)
ul2.wb_rate                  0.1387 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
itlb.accesses                 88142 # total number of accesses
itlb.hits                     88132 # total number of hits
itlb.misses                      10 # total number of misses
itlb.replacements                 0 # total number of replacements
itlb.writebacks                   0 # total number of writebacks
itlb.invalidations                0 # total number of invalidations
itlb.miss_rate               0.0001 # miss rate (i.e., misses/ref)
itlb.repl_rate               0.0000 # replacement rate (i.e., repls/ref)
```

Long Cache Line Prefetching

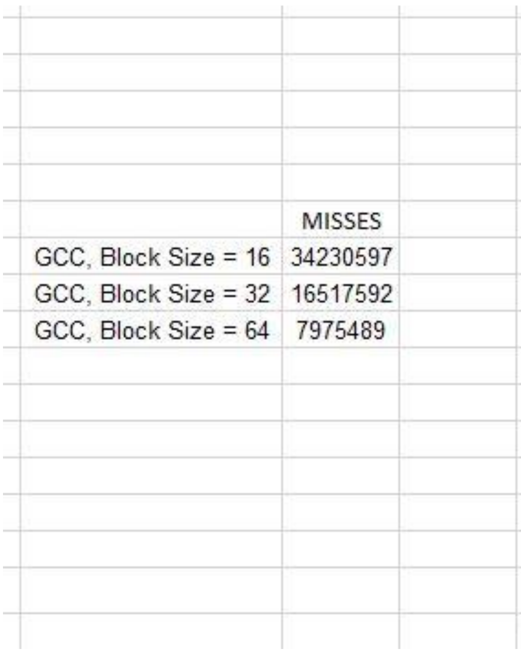
Introduction:-

One of the easiest ways to perform prefetching is by increasing the block size of the cache either during run-time or in the simplescalar code. By increasing the block size, the cache misses reduces. This increases the probability that the next instruction needed will be available in the cache increases. However, this method increases memory traffic and cache pollution.

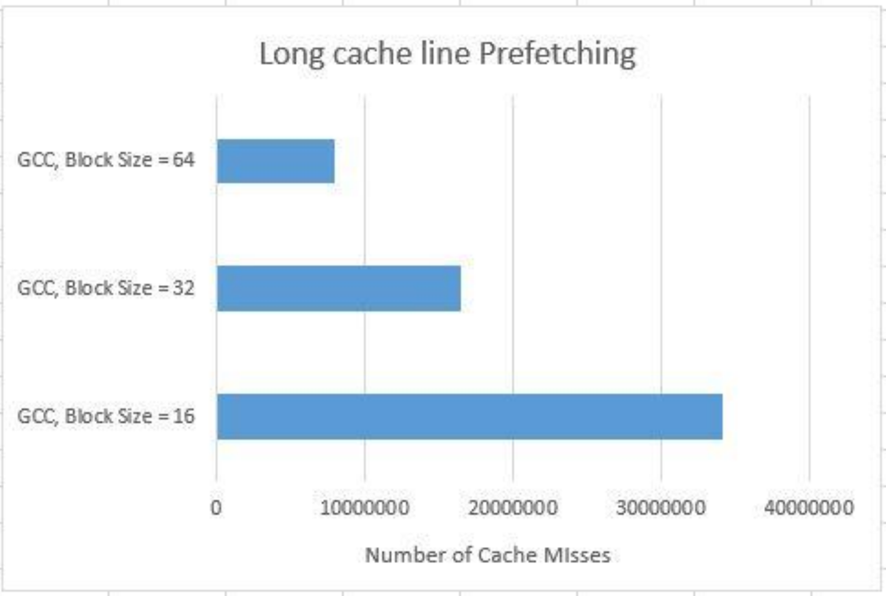
Algorithm:-

1. To reduce cache miss, the block size should be increased.
2. This increase can be done during run time by using this command in terminal:
```
./sim-cache -cache:il1 il1:4096:64:1:l cc1.alpha -O 1stmt.i
```
3. The number 64 is the block size here. The default block size is 32 for this architecture.

Performance:-



	MISSES
GCC, Block Size = 16	34230597
GCC, Block Size = 32	16517592
GCC, Block Size = 64	7975489



Next Line Prefetching

Introduction:-

Next line prefetching is not as random as long cache line prefetching. Keeping up with its name, the next cache line is prefetched automatically when the previous line is fetched if that line isn't already in the cache.

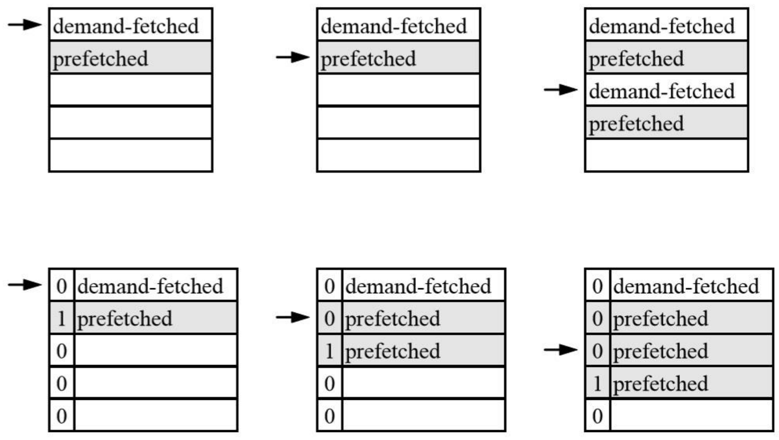
Advantages

- Simple to implement, not a lot of additional logic is required. It is just the logic of fetching the next line.
- Performance is fairly good if branches frequently execute the fall through path.

Disadvantage

- Not very useful in the case where branch is taken. In unconditional jumps and procedure calls where the branch is always taken, next line prefetching can cause increase in memory traffic and cache pollution as it is not likely that the prefetched cache lines are going to be used.

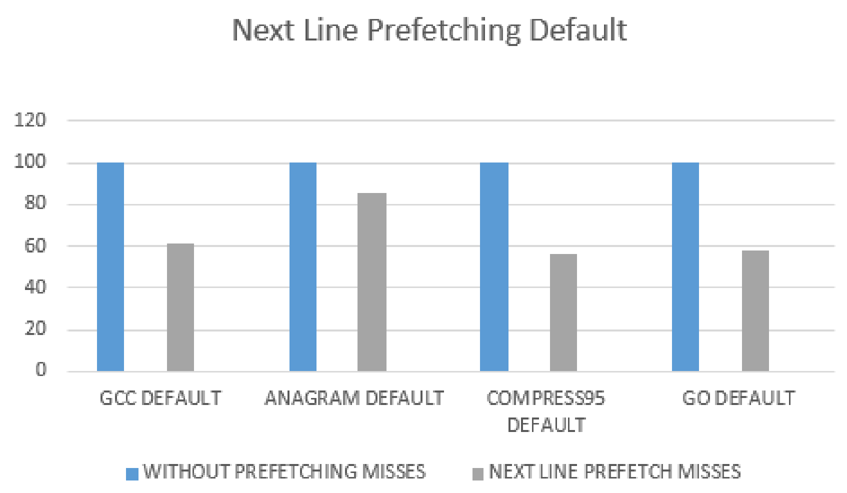
However even with the above disadvantage, next line prefetching has been shown to reduce cache misses by 20-50% Due to its ease of implementation and small cost, the next line prefetching scheme can be found in many microprocessors.



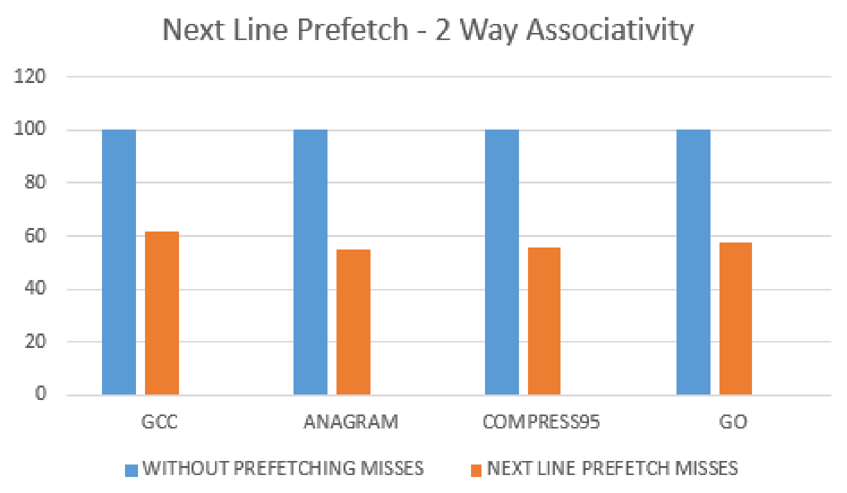
Algorithm :-

1. When an access for block b results in cache miss, next line is prefetched in cache i.e. b+1.
2. The address of the next block is obtained by summing up the address of the fetched block and the block size.
3. We have used tagged prefetch algorithm which associates a tag bit with every memory block.
4. This bit is used to detect when a block is demand-fetched or a prefetched block is referenced for the first time. In either of these cases, the next sequential block is fetched.
5. If cache is full, using replacement policy (e.g LRU, FIFO, etc.) the block is replaced with the new one i.e the prefetched block.
6. Accordingly status bit of blocks are updated.

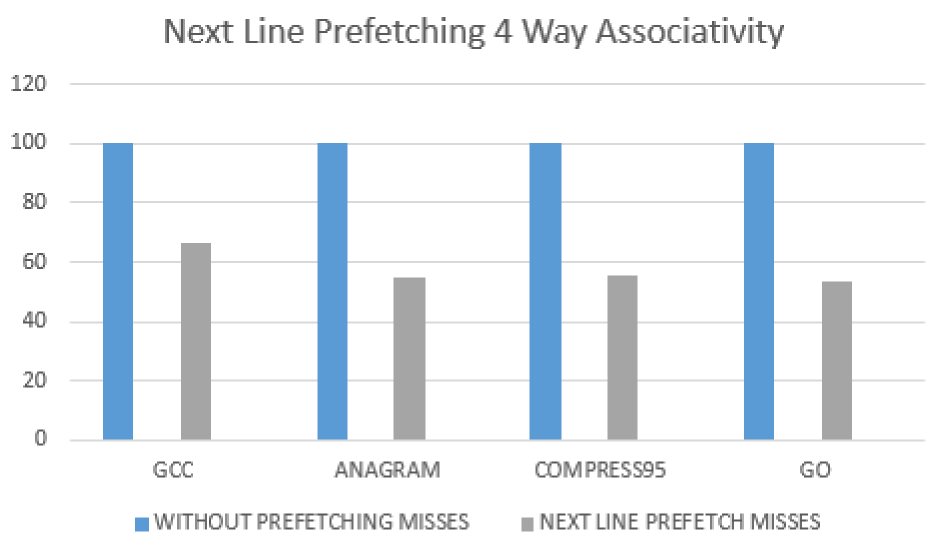
Performance:-



	WITHOUT PREFETCHING		NEXT LINE PREFETCH	
DEFAULT	MISSES	%	MISSES	%
GCC	16517592	100	10191567	61.7012879
ANAGRAM	46906	100	40089	85.466678
COMPRESS95	239	100	135	56.4853556
GO	26717717	100	15574169	58.2915412



WITHOUT PREFETCHING			NEXT LINE PREFETCH	
2 -WAY SET	MISSES	%	MISSES	%
GCC	557817	100	344002	61.6693288
ANAGRAM	851	100	470	55.2291422
COMPRESS95	234	100	130	55.5555556
GO	10644	100	6110	57.4032319



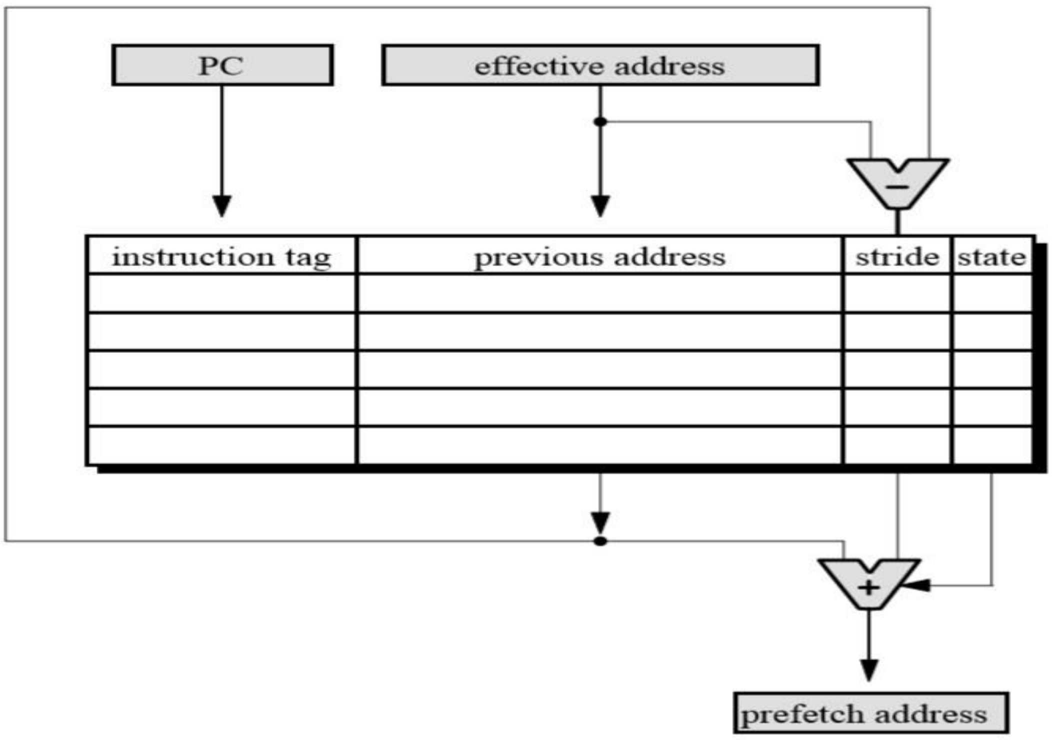
WITHOUT PREFETCHING			NEXT LINE PREFETCH	
4 WAY SET	MISSES	%	MISSES	%
GCC	139630	100	92811	66.4692401
ANAGRAM	851	100	470	55.2291422
COMPRESS95	234	100	130	55.5555556
GO	7226	100	3865	53.4874066

Stride Prefetching

Introduction:-

Stride prefetching is a data prefetching algorithm where the program access pattern is used for data to logically prefetch the next data in the cache. Therefore, this helps to overcome the disadvantage of next line prefetching. In stride prefetching, the prefetching is done in strides, i.e., if a_1 is the present address and the next two address references are a_2 and a_3, the prefetcher takes a stride to a_2 from a_1. Hence, instead of just prefetching the next line, it prefetches the next data requirement logically. This makes stride prefetching highly efficient for data prefetching.

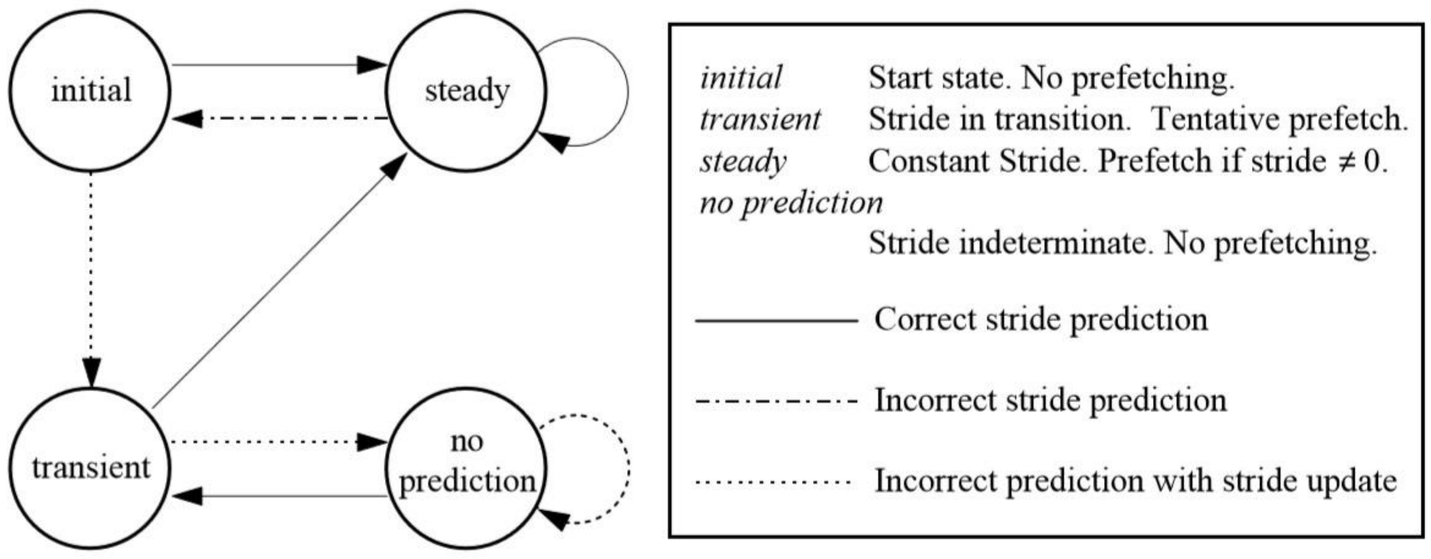
The Reference Prediction Table (RPT) is essential for this prefetching because it has the predictions made in the previous iterations to predict what happens in the next iteration. Therefore, four states are brought about.



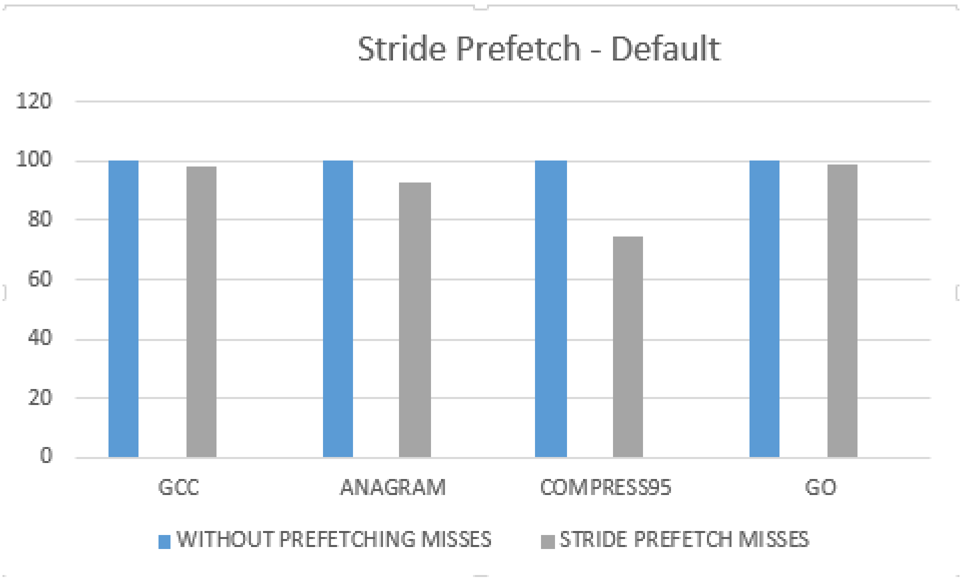
Instruction tag: Has the address of load/store instruction.
Previous address: Contains the last address when the PC reached that instruction.
Stride: Difference between the previous address and the instruction tag.
State: It is a 2-bit address encoding having the prediction for prefetching.

Algorithm:-

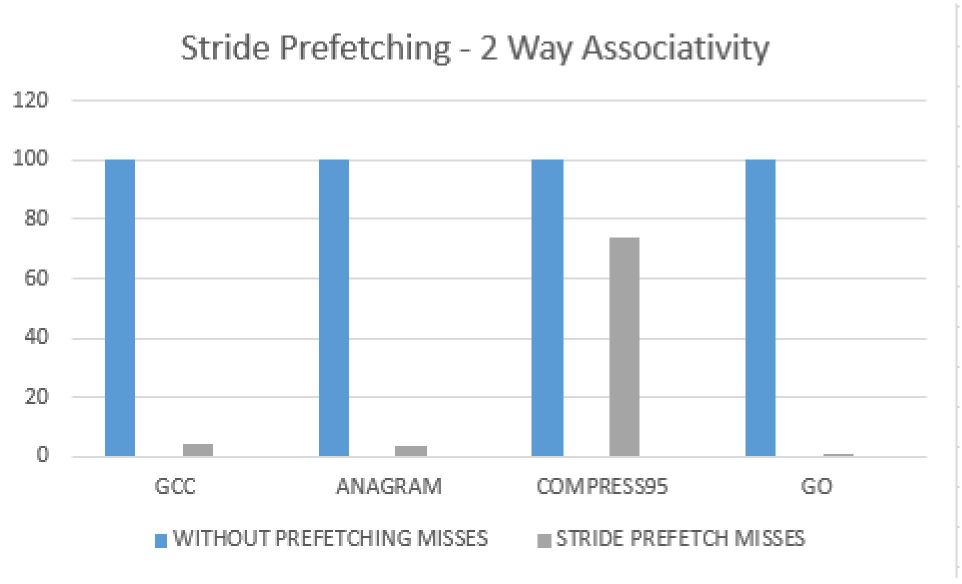
1. When an access for address a_1 results in a cache miss, the cache pointer goes to the main memory to fetch the data.
2. Once it is done fetching that data, then it goes to the next address from the program sequence, i.e., a_2, and prefetches that data for usage.
3. There are four states once the data prefetching is done: initial, transient, steady, and no prediction.
4. When the prefetched data enters, it is in the initial state, where it is checked if a_1-a_2=0. If it is zero, then it goes directly to steady as the stride is zero and it is at the same address.
5. If the two addresses are different, then there is a stride, and hence prefetching should be done.
6. Now the RPT is used to predict an outcome.
7. If the RPT can predict the outcome, then a_2 is prefetched and it goes to the steady state. Otherwise, it goes to no prediction state.


initial
steady
transient
no prediction
initial Start state. No prefetching.
transient Stride in transition. Tentative prefetch.
steady Constant Stride. Prefetch if stride ≠ 0.
no prediction Stride indeterminate. No prefetching.
Correct stride prediction
Incorrect stride prediction
Incorrect prediction with stride update

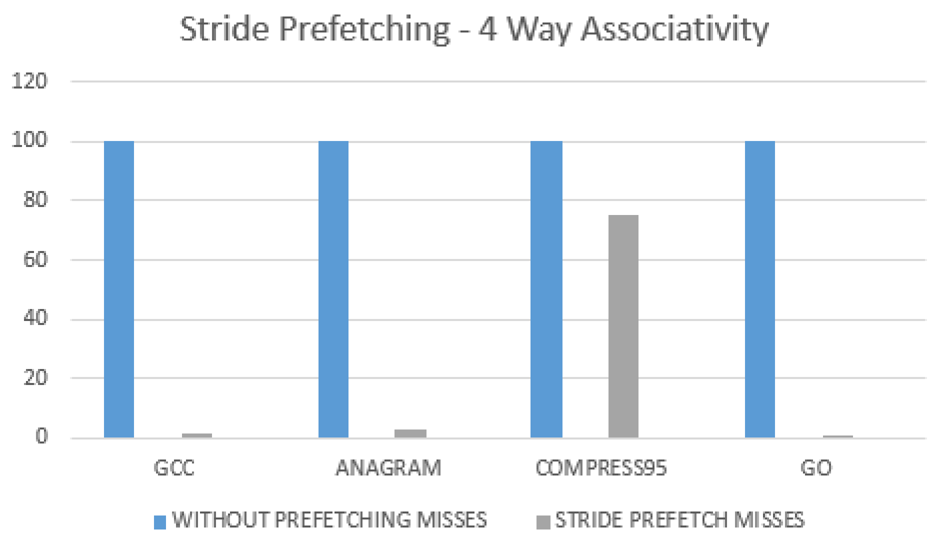
Performance:-



	WITHOUT PREFETCHING		STRIDE PREFETCH	
DEFAULT	MISSES	%	MISSES	%
GCC	8891196	100	8757854	98.5002917
ANAGRAM	245859	100	228502	92.9402625
COMPRESS95	18302	100	13695	74.8278877
GO	16404182	100	16192339	98.7086037



	WITHOUT PREFETCHING		STRIDE PREFETCH	
2 WAY SET	MISSES	%	MISSES	%
GCC	8891196	100	345810	3.88935302
ANAGRAM	245859	100	9175	3.73181376
COMPRESS95	18302	100	13572	74.15583
GO	16192339	100	29113	0.1797949



	WITHOUT PREFETCHING		STRIDE PREFETCH	
4 WAY SET	MISSES	%	MISSES	%
GCC	8891196	100	118731	1.33537715
ANAGRAM	245859	100	6017	2.4473377
COMPRESS95	18302	100	13752	75.139329
GO	16192339	100	8087	0.04994337

Performance Analysis

SIM-CACHE SIMULATION						
ORIGINAL – WITHOUT PREFETCH						
	IL1		DL1		UL2	
	MISSES	MISS RATE	MISSES	MISS RATE	MISSES	MISS RATE
GCC DEFAULT – IL1	16517592	0.049	8891196	0.0728	877230	0.0313
GCC 2-way set asso – IL1	557817	0.0017	8891196	0.0728	613945	0.051
GCC 4-way set asso – IL1	139630	0.0004	8891196	0.0728	341848	0.0294
ANAGRAM DEFAULT	46906	0.0018	245859	0.0265	11130	0.0323
ANAGRAM 2 WAY IL1	851	0	245859	0.0265	9522	0.032
ANAGRAM 4 WAY IL1	851	0	245859	0.0265	9522	0.032
COMPRESS95 DEFAULT	239	0.0027	18302	0.2088	9275	0.2544
COMPRESS95 2 WAY IL1	234	0.0027	18302	0.2088	9275	0.2544
COMPRESS95 4 WAY IL1	234	0.0027	18302	0.2088	9275	0.2544
GO DEFAULT	26717717	0.049	784886	0.765	147108	0.003
GO 2 WAY IL1	10644	0	784886	0.765	16060	0.0007
GO 4 WAY IL1	7226	0	784886	0.765	14620	0.0007
NEXT LINE PREFETCHING– FOR INSTRUCTION						
	IL1		DL1		UL2	
	MISSES	MISS RATE	MISSES	MISS RATE	MISSES	MISS RATE
GCC DEFAULT	10191567	0.0302	8891196	0.0728	827891	0.0382
GCC 2-way set asso – IL1	344002	0.001	8891196	0.0728	588505	0.0497
GCC 4-way set asso – IL1	92811	0.003	8891196	0.0728	334763	0.0289
ANAGRAM DEFAULT	40089	0.0016	245859	0.0265	10896	0.0323
ANAGRAM 2 WAY IL1	470	0	245859	0.0265	9385	0.0315
ANAGRAM 4 WAY IL1	470	0	245859	0.0265	9385	0.0315
COMPRESS95 DEFAULT	135	0.0015	18302	0.2088	9255	0.2546
COMPRESS95 2 WAY IL1	130	0.0015	18302	0.2088	9252	0.2546
COMPRESS95 4 WAY IL1	130	0.0015	18302	0.2088	9252	0.2546
GO DEFAULT	15574169	0.0285	16192339	0.765	141122	0.0037
GO 2 WAY IL1	6110	0	16192339	0.765	15676	0.0007
GO 4 WAY IL1	3865	0	16192339	0.765	14354	0.0006

STRIDE PREFETCHING – FOR DATA PREFETCH						
	IL1		DL1		UL2	
	MISSES	MISS RATE	MISSES	MISS RATE	MISSES	MISS RATE
GCC DEFAULT	16517592	0.049	8757854	0.0717	874266	0.0313
GCC 2-way set asso – DL1	16517592	0.049	345810	0.0028	706748	0.0414
GCC 4-way set asso – DL1	16517592	0.049	118731	0.001	544905	0.0325
ANAGRAM DEFAULT	46906	0.0018	228502	0.0247	9602	0.0293
ANAGRAM 2 WAY DL1	46906	0.0018	9175	0.001	4830	0.0811
ANAGRAM 4 WAY DL1	46906	0.0018	6017	0.0006	4028	0.0761
COMPRESS95 DEFAULT	239	0.0027	13695	0.1563	9239	0.2901
COMPRESS95 2 WAY DL1	239	0.0027	13572	0.1549	11288	0.4754
COMPRESS95 4 WAY DL1	239	0.0027	13752	0.1549	7736	0.4974
GO DEFAULT	26717717	0.049	784600	0.0775	139317	0.0028
GO 2 WAY DL1	26717717	0.049	29113	0.0001	22610	0.0008
GO 4 WAY DL1	26717717	0.049	8087	0	10447	0.0004
LONG CACHE LINE						
GCC, Block Size = 16	34230597					
GCC, Block Size = 32	16517592					
GCC, Block Size = 64	7975489					

SIM-OUTORDER – FOR CPI		
ORIGINAL – WITHOUT PREFETCH		CPI
	GCC	0.8102
	ANAGRAM	0.4572
	COMPRESS95	0.5541
	GO	0.7569
NEXT-LINE PREFETCHING		
	GCC	0.7444
	ANAGRAM	0.4564
	COMPRESS95	0.5436
	GO	0.6985
STRIDE PREFETCHING		
	GCC	0.6283
	ANAGRAM	0.4544
	COMPRESS95	0.5524
	GO	0.7565
LONG CACHE LINE		
(BLOCK SIZE=64, DEFAULT=32)	GCC	0.6571
	ANAGRAM	0.4507
	COMPRESS95	0.5475
	GO	0.6344

Conclusion

In the project discussed above, three prefetching schemes were implemented using the SimpleScalar toolset for Alpha architecture at Ubuntu environment.

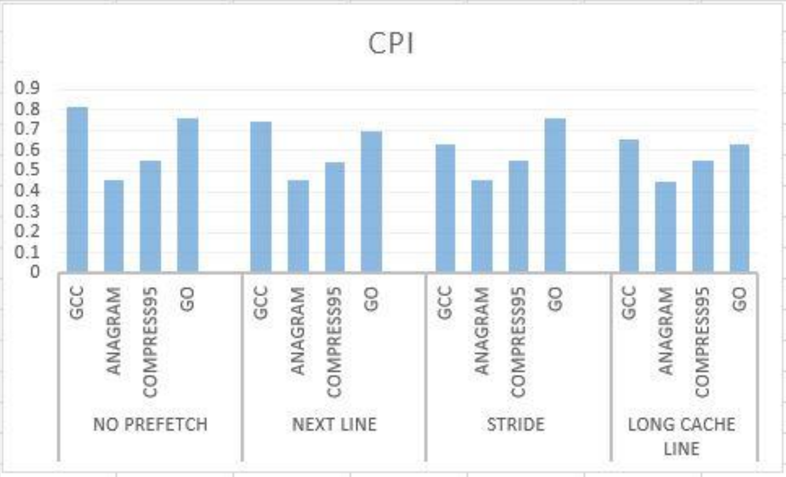
The long cache line prefetching scheme is the most trivial technique which involves simply increasing the size of cache block so that more number of instructions can be accommodated while fetching from memory.
Next-line prefetching method involves fetching the next sequential block from the main memory after the one that is demand fetched. This increases the possibility of cache hits.
Stride prefetching made use of a logical distance called stride that is employed to predict the next cache line to be fetched from the memory.

The performance of four benchmarks was analysed for two-way, four-way and fully-associative cache mapping types. All the data was compiled at Excel sheet and plotted for analysis.

It is observed that miss rate is brought down considerably for every prefetching scheme. Four way associative cache has the lowest number of misses. While next line improves the instruction hit rate, stride improves the data hit rate, however, it can be observed that the CPI is quite less for all the three algorithms while compared to the original. So, through selection of a proper prefetching scheme, the effective cache performance can be increased keeping in mind the cache traffic and cache pollution factors.

Prefetching is hence an efficient technique of improving the CPI for superscalar architectures.

		CPI
NO PREFETCH	GCC	0.8102
	ANAGRAM	0.4572
	COMPRESS95	0.5541
	GO	0.7569
NEXT LINE	GCC	0.7444
	ANAGRAM	0.4564
	COMPRESS95	0.5436
	GO	0.6985
STRIDE	GCC	0.6283
	ANAGRAM	0.4544
	COMPRESS95	0.5524
	GO	0.7565
LONG CACHE LINE	GCC	0.6571
	ANAGRAM	0.4507
	COMPRESS95	0.5475
	GO	0.6344



References

1. '*SimpleScalar Hacker's Guide SimpleScalar Hacker's Guide(for tool set release 2.0) (for tool set release 2.0)*', Todd Austin, SimpleScalar LLC.
2. *'SimpleScalar Tutorial (for release 4.0) (for release 4.0)'*, Todd Austin, Dan Ernst, Eric Larson, Chris Weaver University of Michigan.
3. *'The SimpleScalar Tool Set, Version 2.0.'*, Doug Burger, University of Wisconsin, Todd M. Austin, SimpleScalar LLC.
4. *'The Basics of Caches',* Sat Garcia.
5. '*Computer Architecture Lecture 24: Prefetching*', Prof. Onur Mutlu, Carnegie Mellon University.
6. 'Effective Hardware Based Data Prefetching for High performance Processors', Tien-Fu Chen, Jean-Loup Baer, IEEE.
7. *'Data Prefetch Mechanisms',* Steven P. VanderWiel, David J. Lilja, University of Minnesota.