Rutgers University

Cache Optimization

Using Anticipatory Prefetching

Anirudh Rathi

Ketan Bhave

Shriniwas Ayyer

10

ACKNOWLEDGEMENT

We take this opportunity to express our profound sense of gratitude to our Prof. Yangyong Zhang for her supervision, guidance and helping us out in various phases of our project work. We also thank TA Tinting Sun for the technical guidance, especially in the implementation of Simplescalar simulator. We thank them for their timely guidance, patience and whole hearted co-operation.

Finally we would like to acknowledge Todd. M. Austin, the inventor of Simplescalar, for providing us with such a platform where we could implement the strategies and techniques which we learned through our coursework 'Advanced Computer Architecture'.

Table of Contents

I. Introduction

Today's high performance microprocessors operate at speeds that far outpace even the fastest of the memory bus architectures that are commonly available. One of the biggest limitations of main memory is the wait state: period of time between operations. This means that during the wait states the processor wait for the memory to be ready for the next operation. The most common technique used to match the speed of the memory system to that of the processor is caching.

Cache Memory is the level of computer memory hierarchy situated between the processor and main memory. It is a very fast memory the processor can access much more quickly than main memory or RAM. Cache is relatively small and expensive. Its function is to keep a copy of the data and code (instructions) currently used by the CPU. By using cache memory, waiting states are significantly reduced and the work of the processor becomes more effective.

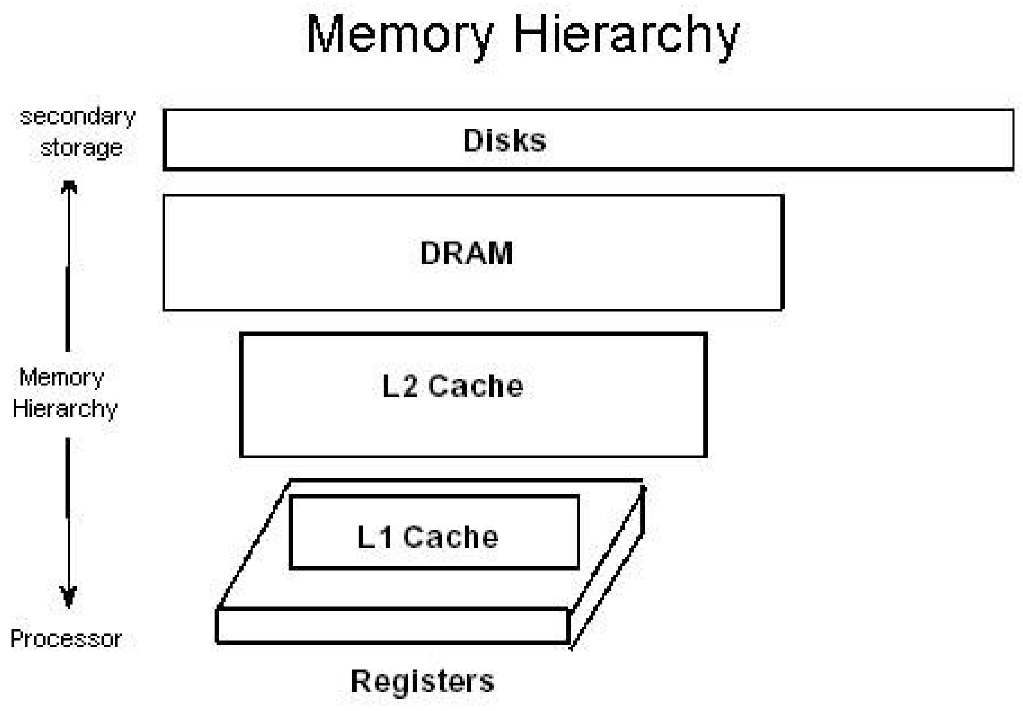


Figure 1: Memory Hierarchy

1. Principles of cache memory

In general cache memory works by attempting to predict which memory the processor is going to need next, and loading that memory before the processor needs it, and saving the results after the processor is done with it. Whenever the byte at a given memory address is needed to be read, the processor attempts to get the data from the cache memory. If the data is found in the cache (**Cache hit**), it is provided to the processor. by simply reading the cache, which is comparatively faster. Otherwise (**Cache miss**), the data has to be recomputed or fetched from its original storage location, which is comparatively slower.

The basic principle that cache technology is based upon is **locality of reference –** programs tend to access only a small part of the address space at a given point in time. This notion has three underlying assumptions– **temporal locality, spatial locality and sequentiality.**

- **Temporal locality** means that referenced memory is likely to be referenced again soon. In other words, if the program has referred to an address it is very likely that it will refer to it again.
- **Spatial locality** means that memory close to the referenced memory is likely to be referenced soon. This means that if a program has referred to an address, it is very likely that an address in close proximity will be referred to soon.
- **Sequentiality** means that the future memory access is very likely to be in sequential order with the current access.

2. Cache's effectiveness:

The effectiveness of the cache is determined by the number of times the cache successfully provides the required data. This is the place to introduce some terminology. A ***Cache hit*** is the term used when the data, required by the processor is found in the cache. If data is not found, we have a ***Cache Miss***. There are three types of misses:

- **Compulsory misses** – the first access to a block is not in the cache, so the block must be brought into the cache. These are also called cold start misses or first reference misses.
- **Capacity misses** – if the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. These are misses due to cache size.
- **Conflict misses** – if the block placement strategy is set associative or direct mapped, conflict misses will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called collision misses or interference misses.

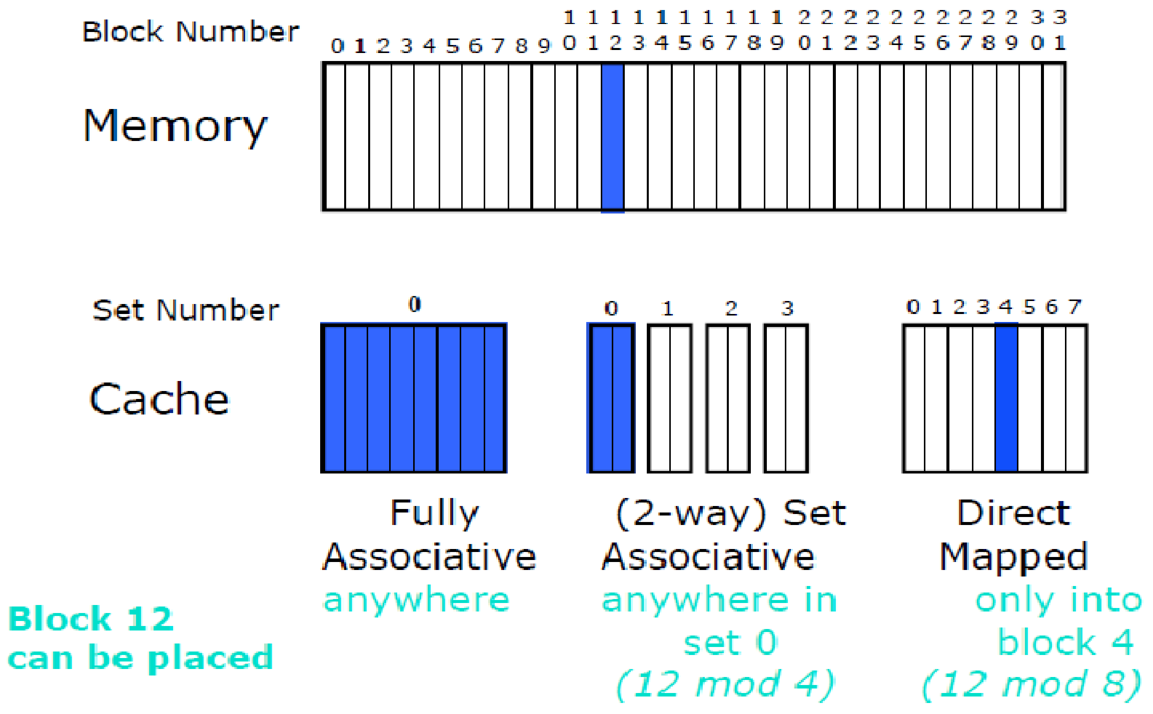
3. Elements of Cache Design

i. Cache Size

One way to decrease miss rate is to increase the cache size. The larger the cache, the more information it will hold, and the more likely a processor request will produce a cache hit, because fewer memory locations are forced to share the same cache line. However, applications benefit from increasing the cache size up to a point, at which the performance will stop improving as the cache size increases. When this point is reached, one of two things have happened: either the cache is large enough that the application almost never have to retrieve information from disk, or, the application is doing truly random accesses, and therefore increasing size of the cache doesn't significantly increase the odds of finding the next requested information in the cache. The latter is fairly rare - almost all applications show some form of locality of reference.

ii. Mapping function

The mapping function gives the correspondence between main memory blocks and cache lines. Since each cache line is shared between several blocks of main memory (the number of memory blocks >> the number of cache lines), when one block is read in, another one should be moved out. Mapping functions minimize the probability that a moved-out block will be referenced again in the near future. There are three types of mapping functions: direct, fully associative, and n-way set associative.



iii. Cache Line Replacement Algorithms

When a new line is loaded into the cache, one of the existing lines must be replaced. In a direct mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache we have a choice of where to place the requested block and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set associative cache, we must choose among the blocks in the selected set. Therefore a line replacement algorithm is needed which sets up well defined criteria upon which the replacement is made. A large number of algorithms are possible and many have been implemented. Four of the most common cache line replacement algorithms are:

- *Least Recently Used* (LRU) - the cache line that was last referenced in the most distant past is replaced.
- *FIFO* (First In First Out) - the cache line from the set that was loaded in the most distant past is replaced.
- *Random* - a randomly selected line form cache is replaced

iv. Block/Line Size

Another element in the design of a cache system is that of the line size. *This is the number of bytes per cache line, sometimes also referred to as block size*. When a block of data is retrieved from main memory and placed in the cache, not only the requested word is loaded but also some number of adjacent words (those in the same block) is retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality. However, as the block becomes even bigger the hit ratio will begin to decrease because the probability of using the newly fetched information will be less than the probability of reusing the information that has been replaced. Two specific effects need to be considered:

Larger blocks reduce the number of blocks that will fit into the cache. Because each fetch overwrites older cache contents, a small number of blocks in the cache will result in data being overwritten shortly after it has been loaded.

As a block becomes larger, each additional word is farther from the requested word, and is therefore less likely to be needed in the near future (principle of locality).

4. Benchmarks

Benchmarks are individual programs or a mixture of programs that are run on a target computer to measure the overall performance of the system, or to measure more specific aspects of the performance, such as Graphics Applications, I/O Processing, Net Browsing, etc. Any aspects of computer performance that matters to the user can be benchmarked.

The role of benchmarks in computer architecture is to evaluate the performance of particular system architecture and extrapolate from the results obtained. Benchmarks are used not only to evaluate the performance of a given system under different hardware and software configurations, but also to compare the performance of different systems. The performance of different functionalities of a machine can be predicted by running a representative workload on the machine. Some benchmarks test how fast a computer can perform a specific task whereas some test how fast the computer handles different tasks at the same time. For example, *SPECint95* benchmark measures the performance of the processor only, but *SYSmark/NT* benchmark measures the overall performance of your entire PC when running real applications.

The benchmarks used in this project are:

The table below contains a brief description of the benchmarks and their SPEC reference times (used for calculating SPEC metrics). More detailed analysis and descriptions will be provided in future issues of the SPEC Newsletter.

Benchmark	**Application Area**	**Specific Task**
go	Game playing; artificial intelligence	Plays the game Go against itself.
m88ksim	Simulation	Simulates the Motorola 88100 processor running Dhrystone and a memory test program.
gcc	Programming & compilation	Compiles pre-processed source into optimized SPARC assembly code.
compress	Compression	Compresses large text files (about 16MB) using adaptive Limpel-Ziv coding.
li	Language interpreter	Lisp interpreter.
ijpeg	Imaging	Performs jpeg image compression with various parameters.
perl	Shell interpreter	Performs text and numeric manipulations (anagrams/prime number factoring).
vortex	Database	Builds and manipulates three interrelated databases.
tomcatv	Fluid Dynamics / Geometric Translation	Generation of a two-dimensional boundary-fitted coordinate system around general geometric domains.
swim	Weather Prediction	Solves shallow water equations using finite difference approximations. (The only single precision benchmark in CFP95.)
su2cor	Quantum Physics	Masses of elementary particles are computed in the Quark-Gluon theory.
.hydro2d	Astrophysics	HydrodynamicalNavier Stokes equations are used to compute galactic jets.
mgrid	Electromagnetism	Calculation of a 3D potential field.
applu	Fluid Dynamics/Math	Solves matrix system with pivoting.
turb3d	Simulation	Simulates turbulence in a cubic area.
apsi	Weather Predication	Calculates statistics on temperature and pollutants in a grid.
fpppp	Chemistry	Performs multi-electron derivatives.
wave	Electromagnetics	Solve's Maxwell's equations on a cartesian mesh.

Table 1: Description of Benchmarks

II. Project Overview

Objectives:

The two main objectives within this project are:

- ➢ Analyzing the effect of different parameters on Cache performance
- ➢ Propose our own cache optimization method.

Step 1: Evaluating Benchmark programs

We begin with evaluating the L1 cache miss rates for the benchmark programs. This evaluation will be done by varying the following parameters:

- **Cache Size:** 8KB, 16KB, 32KB and 64KB
- **Cache Organization:** direct-mapped, 2-way SA, and 4-way SA.

The benchmark programs were run on **SimpleScalar** and the cache statistics were recorded.

The statistics are displayed as below:

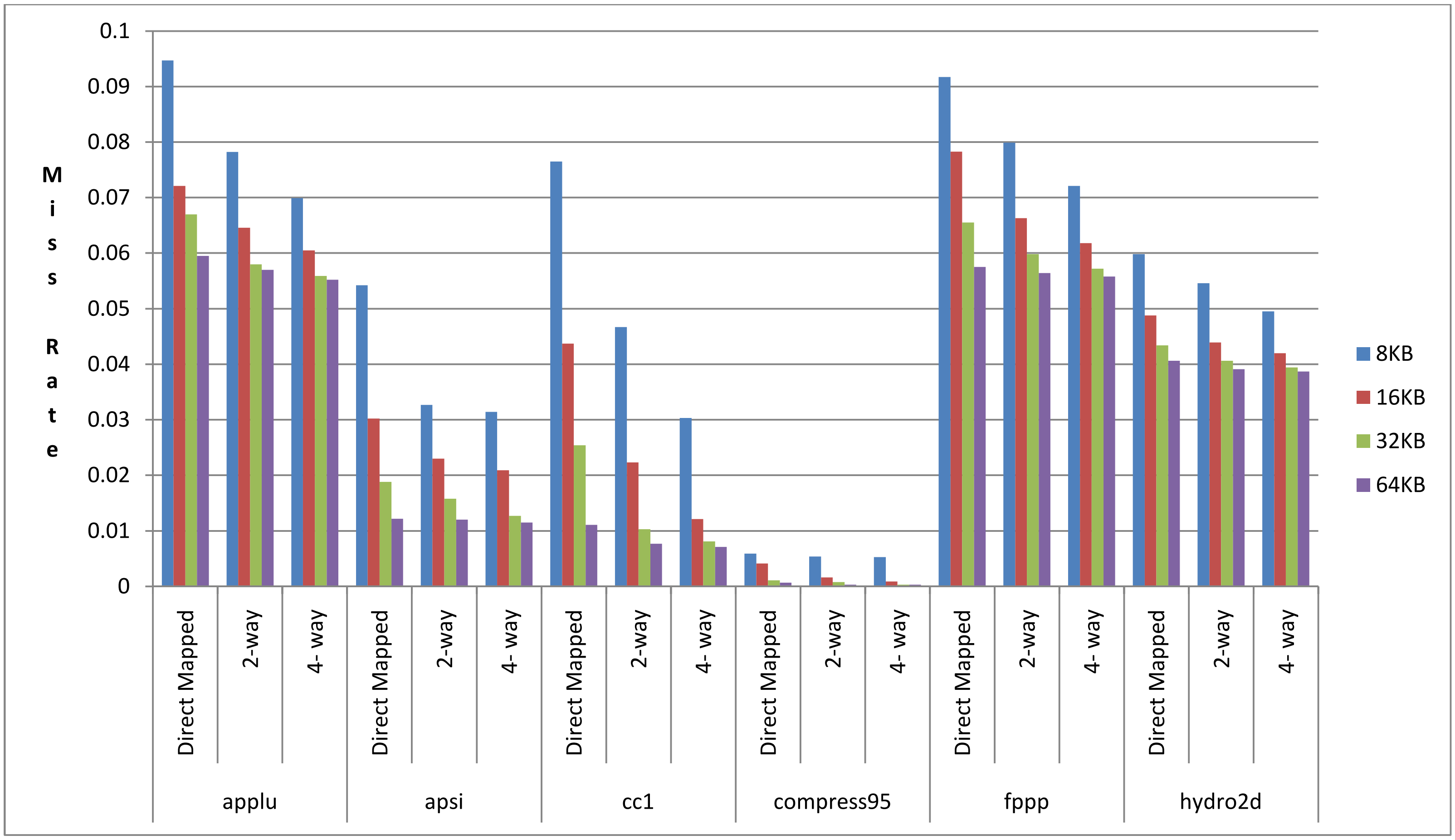


Figure 2: IL1 Step 1 Results

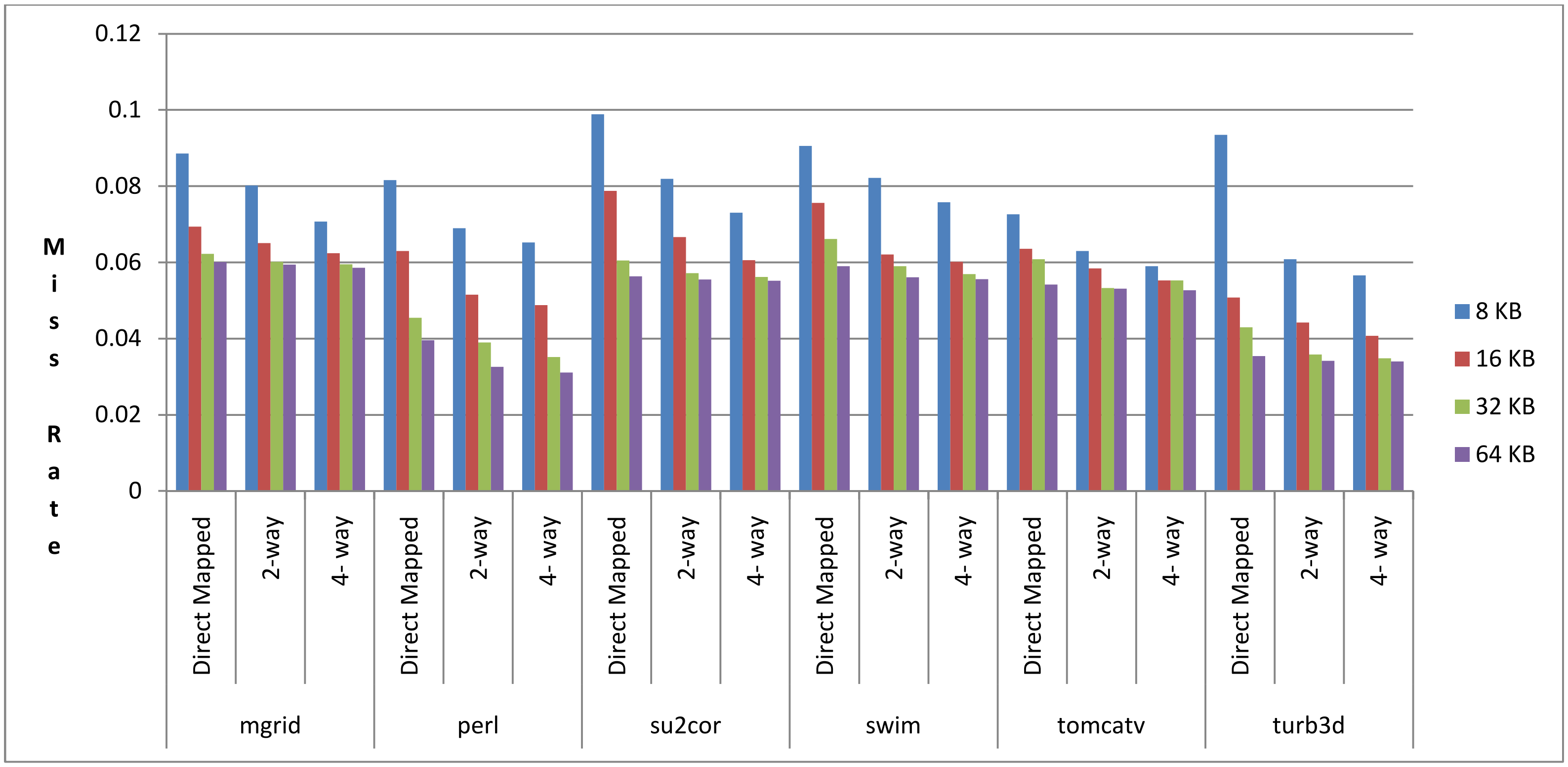


Figure 3: IL1 Step 1 Results

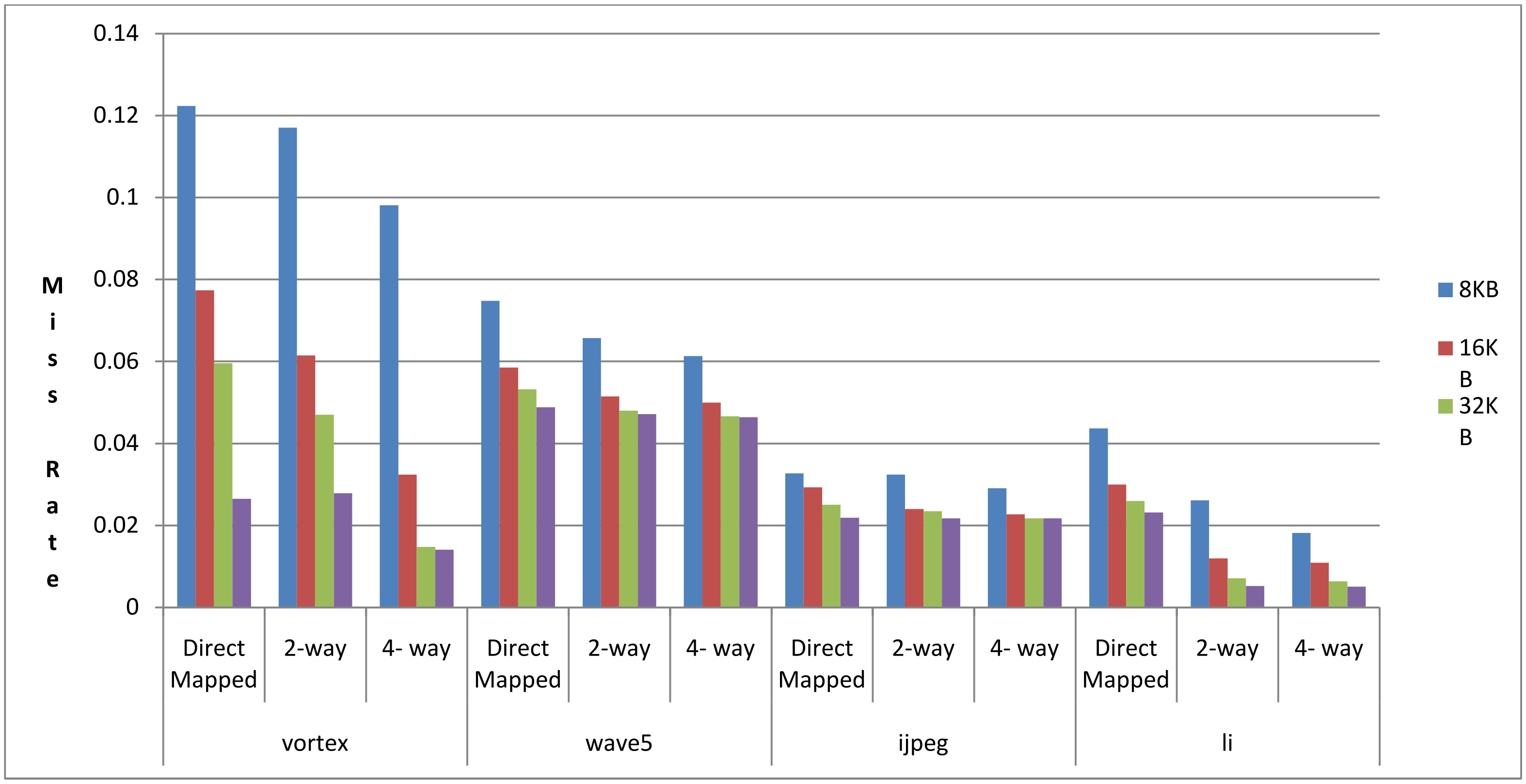


Figure 4: IL1 Step 1 Results

Here we observe that, as we increase the cache size the miss rate decreases. This is because as the cache size increases there would be lesser number of capacity misses as there is more data stored in the cache. Also, as the associativity increases, for a particular cache size, the miss rate decreases. This happens because lesser the associativity, greater the chances of a particular cache block being replaced. This would result in a cache miss if the replaced block is looked for in the cache after being flushed out. Lower the associativity of the cache, higher are the chances of data being flushed out which can lead to a cache miss at a later stage. Hence the miss rate for a given cache size in this case would be maximum when it is direct mapped and least when it is 4 way associativity.

Therefore, we decide that the optimum configuration is cache size **64KB** and **4-way set associativity**. For this configuration, we take a look at the following six benchmarks with the worst performance:

Benchmark	IL1	DL1
applu.ss	0.0552	0.0797
hydro2d.ss	0.0387	0.0596
fpppp.ss	0.0574	0.0914
mgrid.ss	0.0586	0.0918
su2cor	0.0509	0.0755
swim.ss	0.0517	0.0756

Step 2: Varying the Cache Replacement Policy

For the configuration chosen above, we evaluate the performance of the above 6 benchmark programs for the following replacement policies:

- Least Recently Used (LRU)
- First In First Out (FIFO)
- Random

The results of varying the replacement policy are:

Benchmarks	Replacement Policy		
	LRU	**FIFO**	**RANDOM**
applu.ss	0.0552	0.0552	0.0583
hydro2d.ss	0.0387	0.0387	0.041
fpppp.ss	0.0574	0.0574	0.0608
mgrid.ss	0.0586	0.0586	0.0612
su2cor	0.0509	0.0509	0.0538
swim.ss	0.0517	0.0516	0.0546

Table 2: L1 Instruction Cache Results

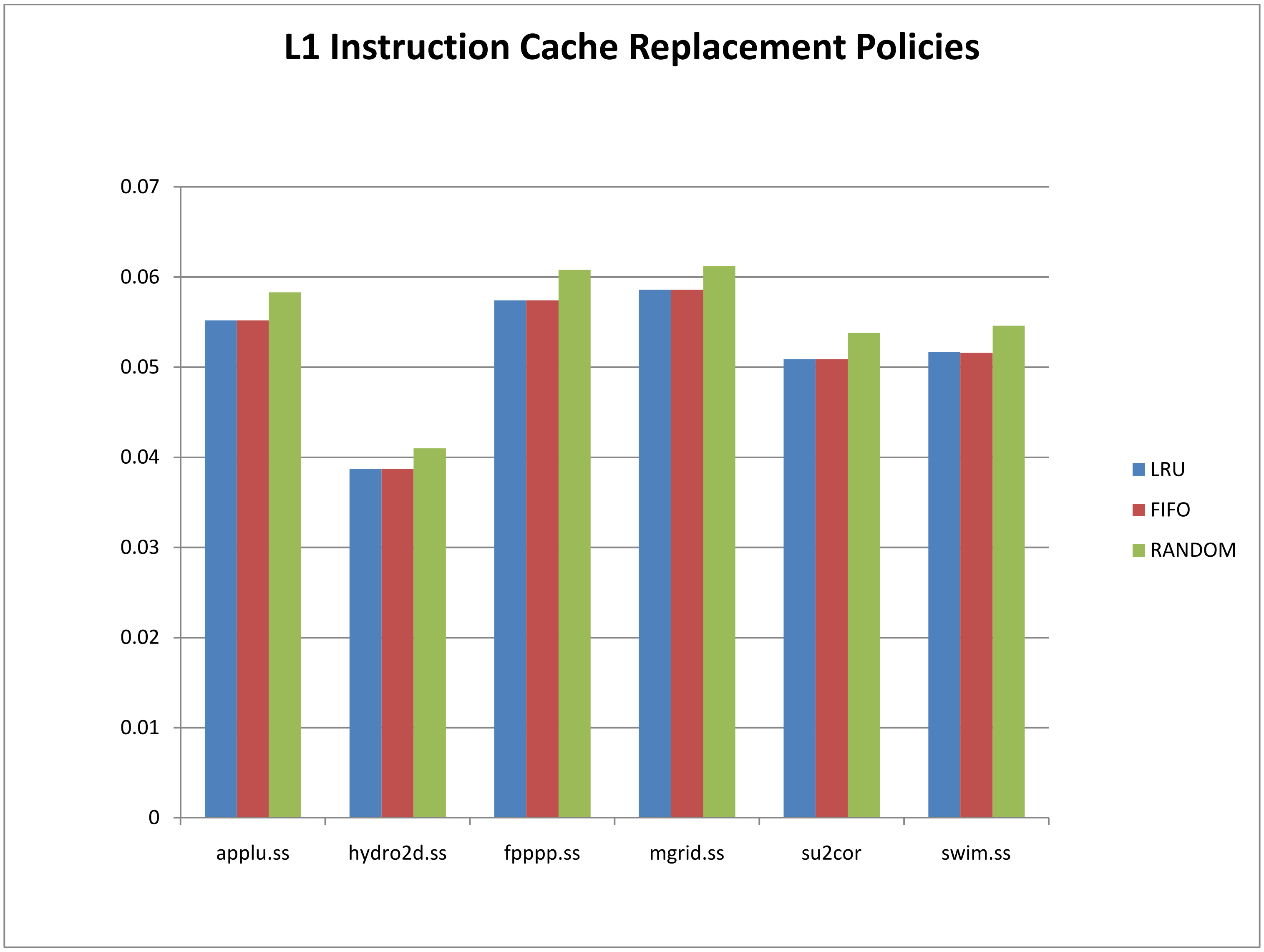


Figure 5: IL1 Cache Replacement Policy

Benchmarks	Replacement Policy		
	LRU	FIFO	RANDOM
applu.ss	0.0797	0.0797	0.081
hydro2d.ss	0.0596	0.0596	0.0611
fpppp.ss	0.0914	0.0914	0.0924
mgrid.ss	0.0918	0.0918	0.0935
su2cor	0.0755	0.0755	0.076
swim.ss	0.0756	0.0756	0.0769

Table 3: L1 Data Cache Results

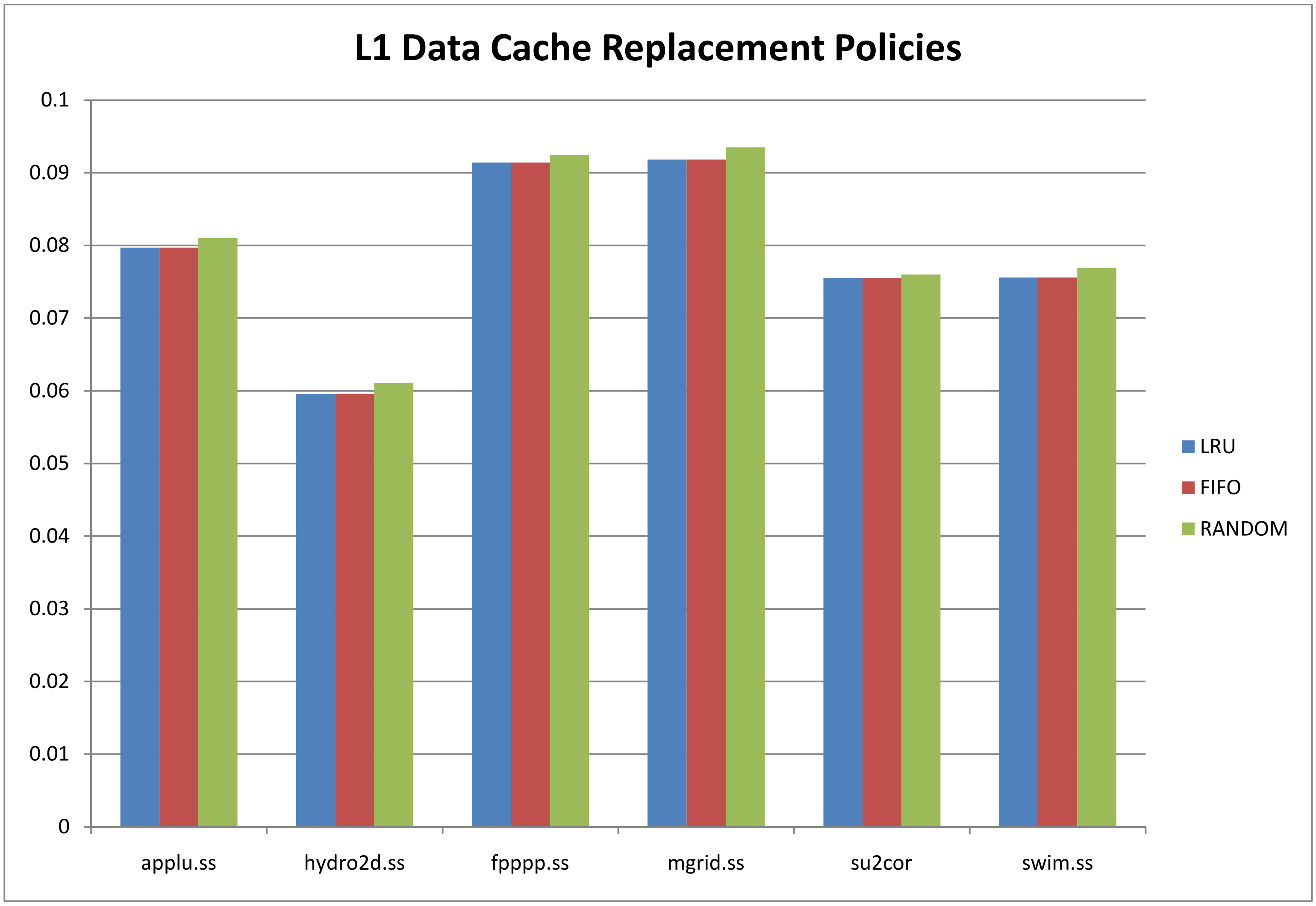


Figure 6: DL1 Cache Replacement Policies

Based on the results above, we conclude that **LRU** is the replacement policy which gives us the least miss rate.

Step 3: Performance Analysis

From the above step, we choose the benchmark program with the worst performance for cache size 64 KB, 4-way set associativity and LRU replacement policy. For the set of benchmarks observed, we choose **mgrid.ss.**

The analysis of the poor performance is as follows:

Cache misses = Capacity misses + Conflict misses + Compulsory misses

The effect of capacity misses was reduced on all the benchmark programs by increasing the cache size.

The effect of conflict misses was reduced on all the benchmark programs by increasing the associativity.

Thus the poor performance is due to the ***Compulsory misses.***

Step 4: Optimization

Based on the analysis, we conclude that the low miss rate was due to compulsory misses.

Compulsory misses occur due to the first time access to a block. Cache size and associativity make no difference to the number of compulsory misses

Increasing the block size decreases the compulsory misses, but it is not a way of optimization.

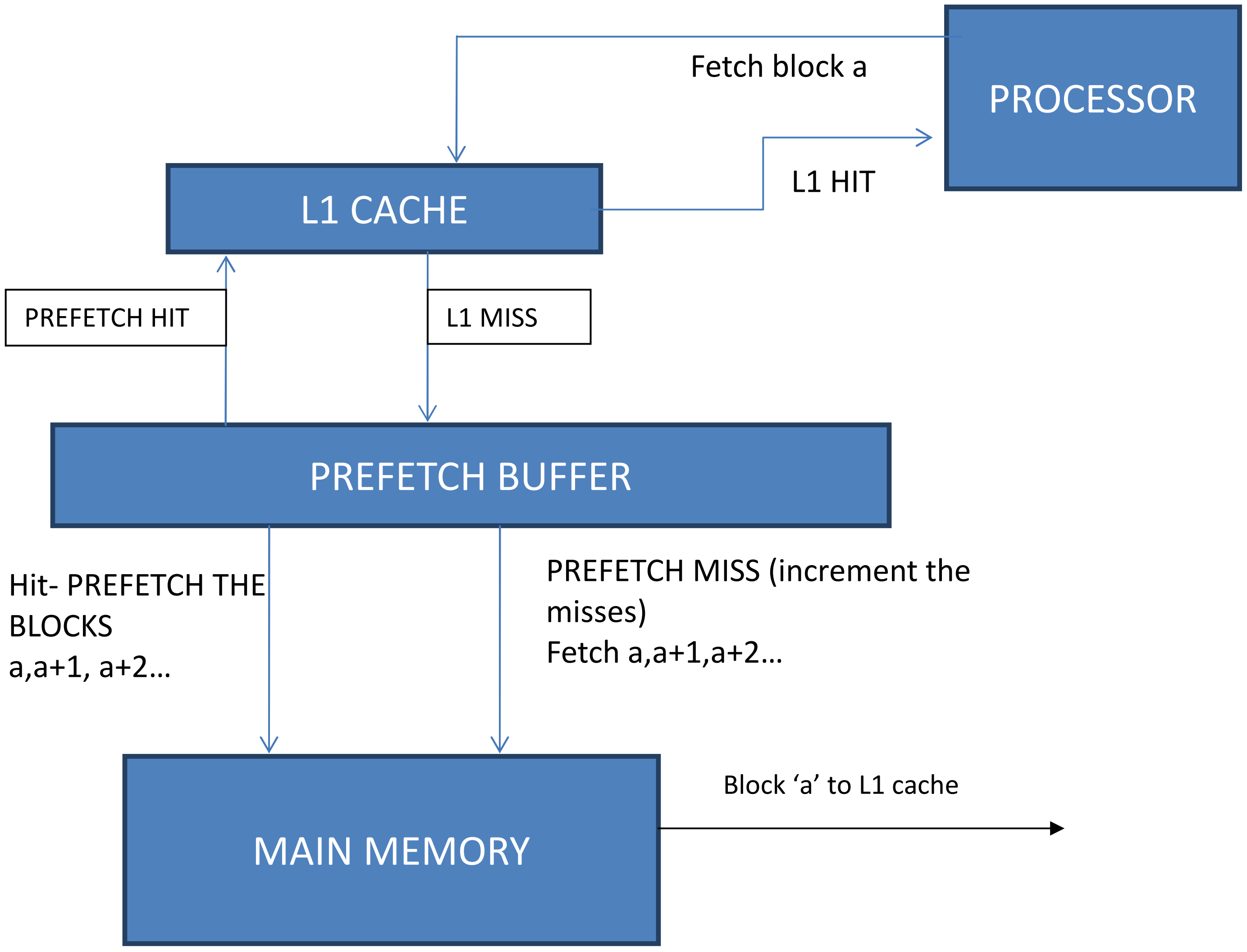
Whenever a block is fetched from memory, if the next few sequential blocks are fetched and stored somewhere from which they could be accessed, and then the number of compulsory misses decreases. This technique is known as "**Prefetching**".

Thus, focusing on the common case we implement a new hardware: ***Prefetch Buffer*** to optimize the performance of the cache.

III. Implementing Cache Optimization Strategy

1. Logic

The cache optimization strategy used by us is implementing an "***Anticipatory prefetching''*** technique using a Variable Size Prefetch Buffer. On a miss for block 'a' at the L1 cache, the prefetch buffer is checked for block 'a'. If there is a prefetch hit, then the block is moved to L1 cache and the prefetch buffer is filled with sequentially blocks 'a','a+1','a+2' and so on. On a prefetch miss, which is a ***true miss,*** block 'a' is fetched into the L1 cache and the prefetch buffer is filled with blocks 'a', 'a+1', 'a+2' and so on. Thus we are anticipating that if the processor is using block 'a', it might need blocks 'a', 'a+1', 'a+2', and so on, thus exploiting spatial locality.



The Prefetch buffer comes into picture only when there is a L1 cache miss, thus not interfering with any cache operations. The implementation of Prefetch Buffer reduces the Compulsory misses as well as reduces the access time since the processor need not access the main memory if there is a Prefetch Hit.

The result after implementing the prefetch buffer is as shown:

Benchmarks	Replacement Policy with prefetch buffer of size 32		
	LRU	FIFO	RANDOM
applu.ss	0.0069	0.0069	0.0117
hydro2d.ss	0.0043	0.0044	0.0078
fpppp.ss	0.0071	0.0071	0.0111
mgrid.ss	0.0058	0.0058	0.099
su2cor	0.0064	0.0064	0.0116
swim.ss	0.0060	0.0066	0.0106

Table 4: Instruction Cache Results with Prefetch buffer

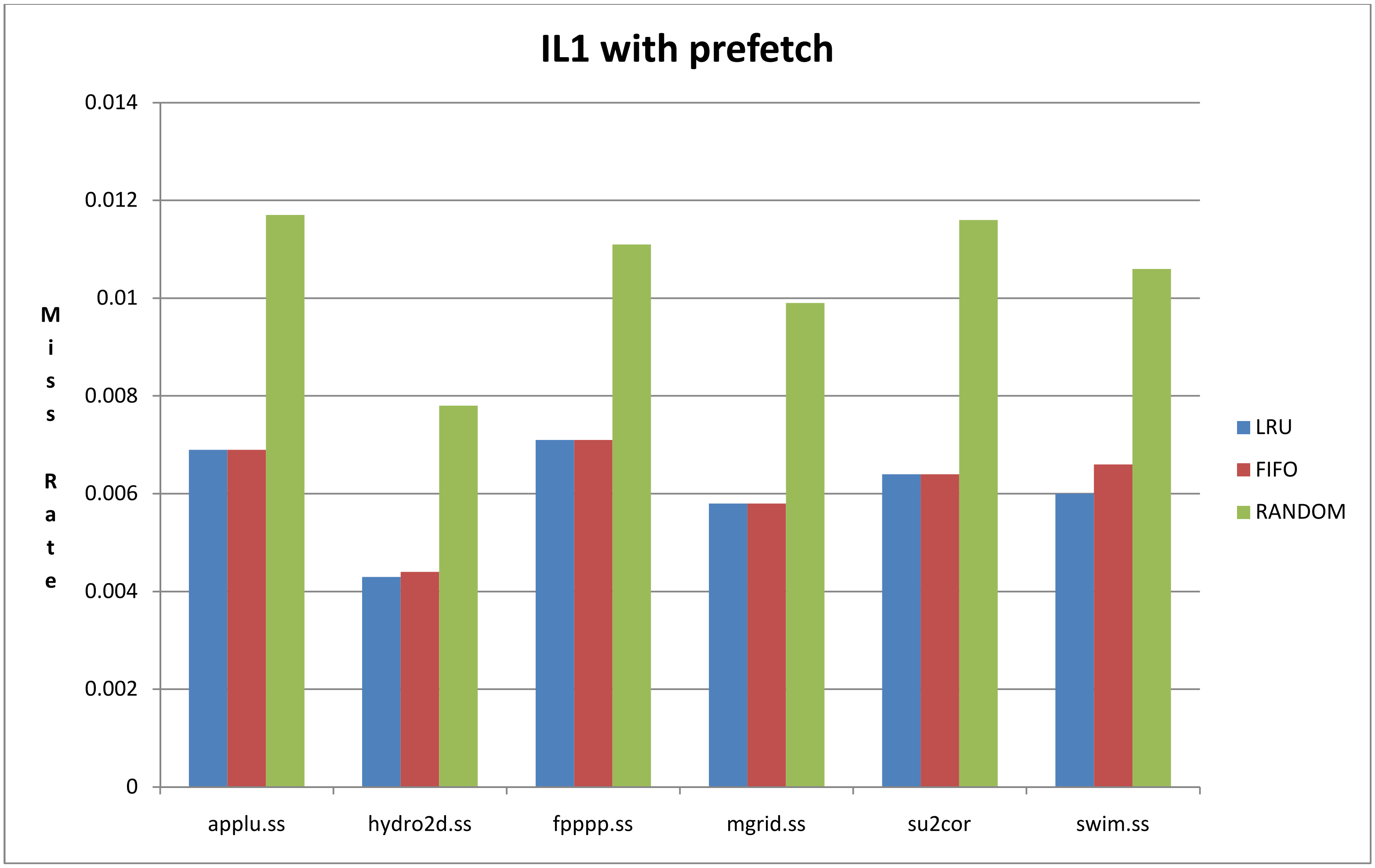


Figure 7: Graph showing Instruction Cache Miss Rate with Prefetch Buffer

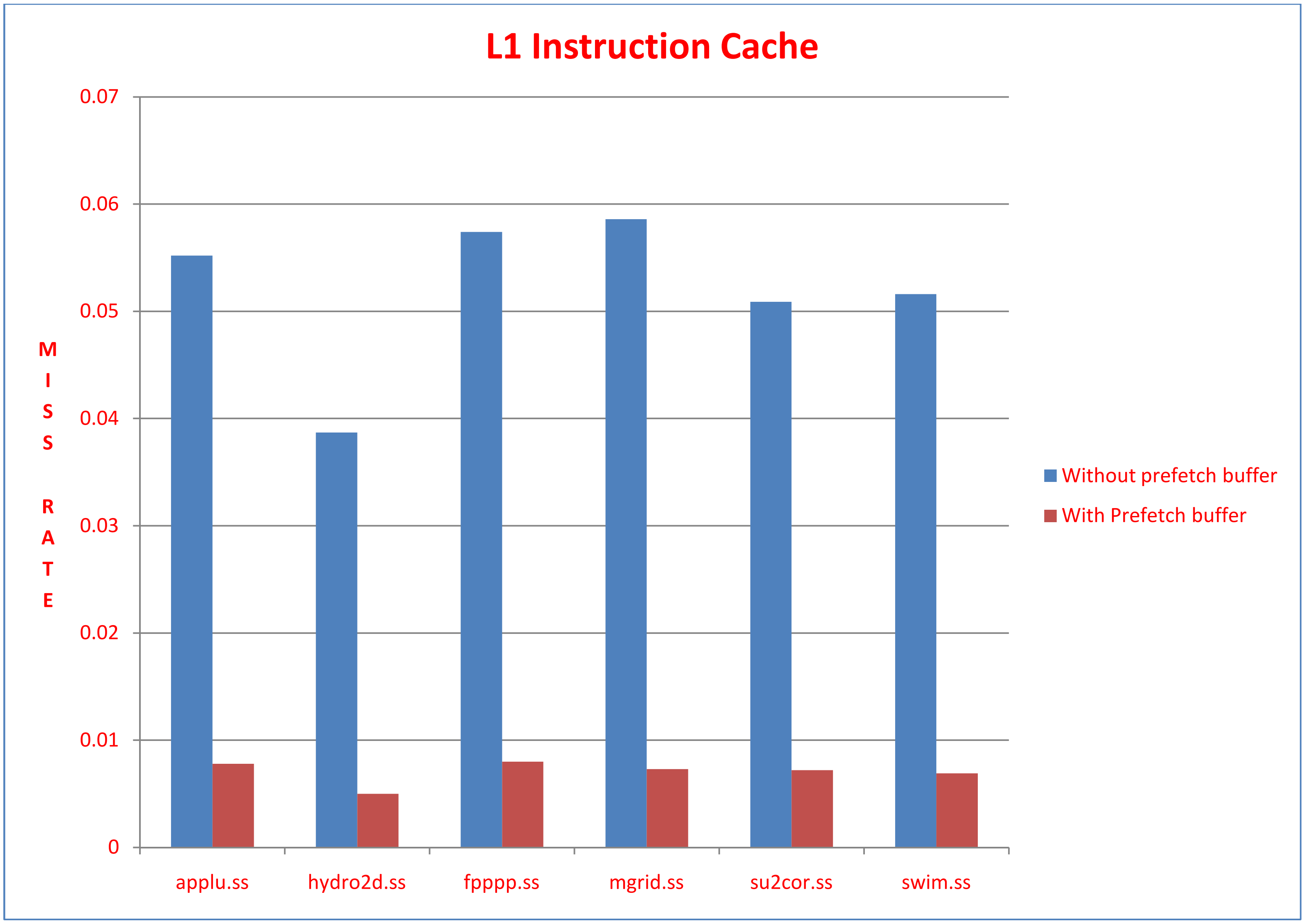


Figure 8: Graph Shoing Improvement in L1 Instruction Cache

Benchmarks	Replacement Policy with prefetch buffer of size 32		
	LRU	FIFO	RANDOM
applu.ss	0.0072	0.0072	0.0135
hydro2d.ss	0.0052	0.0052	0.0105
fpppp.ss	0.0079	0.0079	0.0137
mgrid.ss	0.0061	0.0061	0.0121
su2cor	0.0078	0.0078	0.0125
swim.ss	0.0050	0.0050	0.0111

Table 5: Data Cache Results with Prefetch Buffer

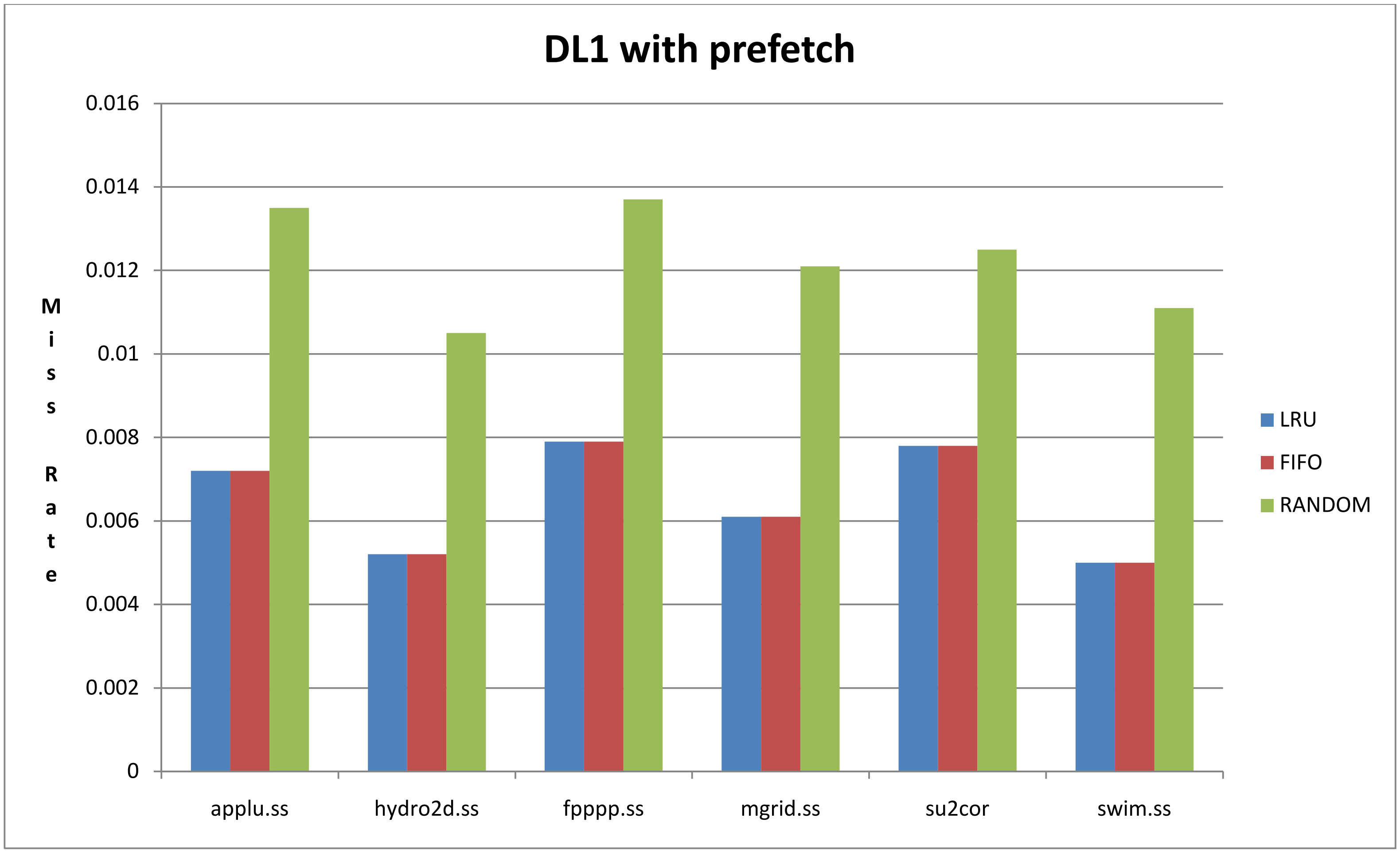


Figure 9: Graph showing Data Cache Miss Rate with Prefetch Buffer

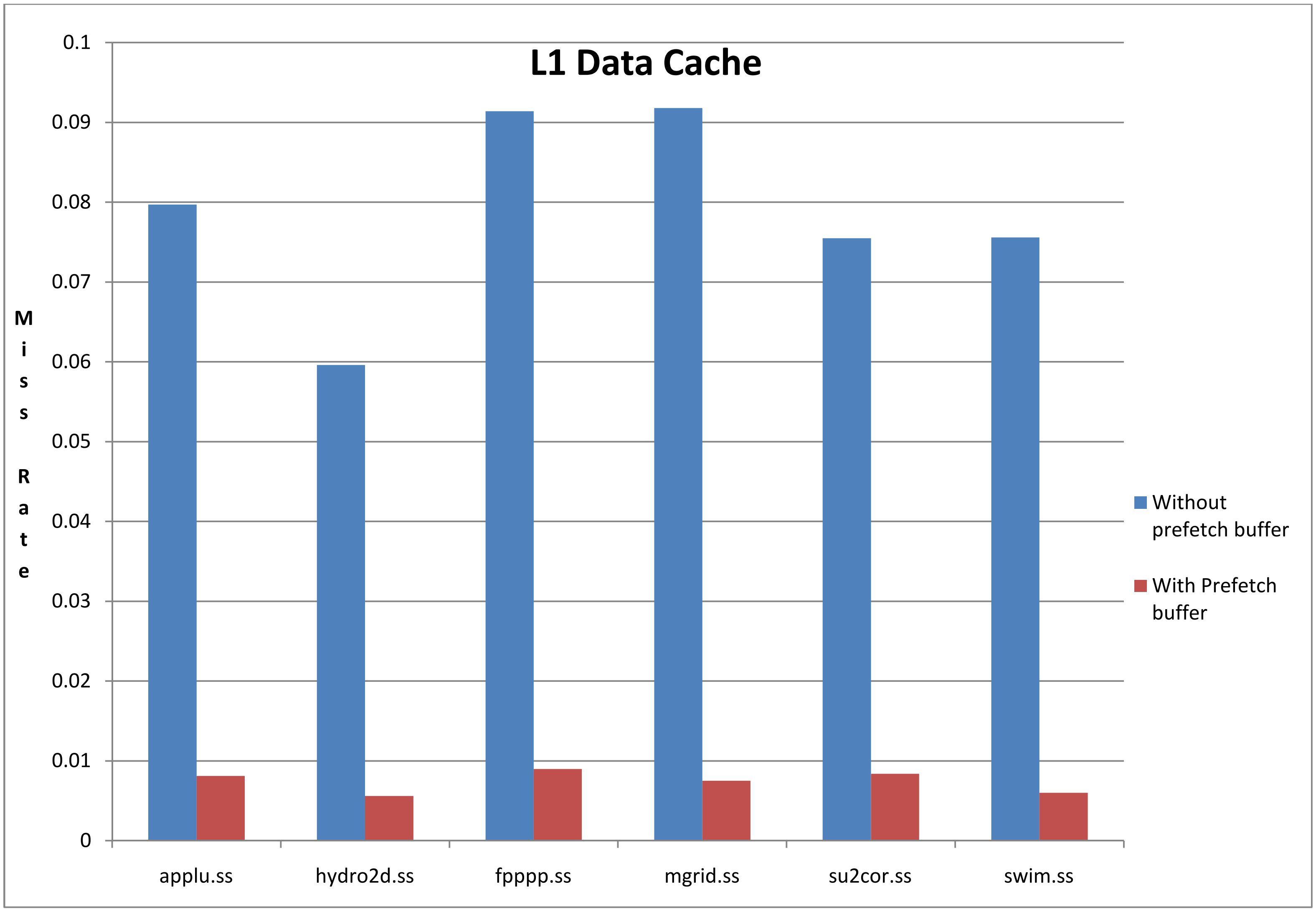


Figure 10: Graph Showing L1 Data Cache Improvement

From the results we see that, the implementation of Prefetch Buffer has **reduced the miss rate by about ~88% for all the benchmark programs**, which is a significant improvement. As the buffer size increases the miss rate drops further low. Thus we see that the programs were largely affected due to Compulsory Misses, which was overcome by '***Anticipatory Prefetching'***.

2. PREFETCH ACCURACY

To analyze the usefulness of the Prefetch Buffer, we have defined a parameter called ***'Prefetch Accuracy'*** which is the ratio of no. of prefetch hits upon no. of prefetch reads.

$$\text{Prefetch Accuracy} = \frac{\text{Number of prefetch hits}}{\text{Number of prefetch reads}}$$

The implemented variable size prefetch buffer has accuracy of about ~88% i.e. 88% of the times the prefetch buffer supplies the required block to the processor whenever the processor accesses it. The accuracy of the prefetch buffer increases as the buffer size increases as shown:

SIZE	IL1 ACCURACY	DL1 ACCURACY
1	85.24	91.76
2	86.1	91.91
4	86.85	92.38
8	87.72	92.69
16	89.01	93.16
32	90.62	93.93

Table 6: Prefetch Accuracy for different size of the Prefetch Buffer

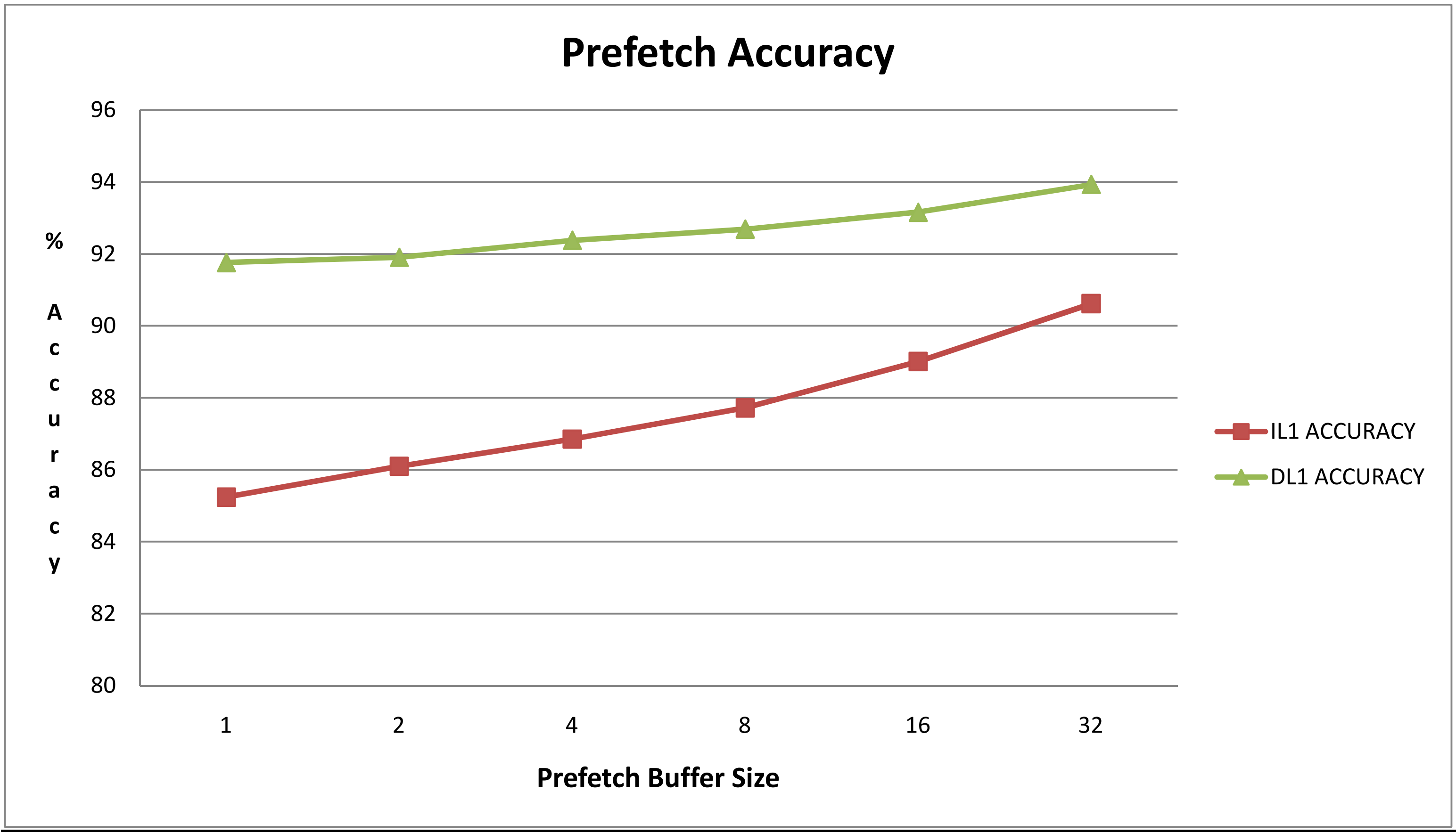


Figure 11: Graph showing Prefetch Accuracy for different sizes of prefetch buffer

IV. Conclusion

We studied how the cache has been implemented in Simplescalar and performed the analysis of miss rate for a suite of benchmark programs. We observed the effect of varying the Cache Size, Cache Associativity and the Cache Replacement policy on the miss rate.

We executed the benchmark programs for varying cache sizes, associativity, and replacement policies. We noted down the resulting miss rates and plotted the graphs. The analysis of the results gave us a lot of insights into the cache working and performance.

We chose the best configuration of size, associativity and replacement policy and tried to optimize it for further reducing the miss rate. We have incorporated a Prefetch Buffer which exploits spatial locality and decreases the compulsory misses. We executed the benchmark programs after the optimization and noted down the results.

The comparison of results show a significant improvement in the miss rate which leads to a conclusion that the compulsory misses were reduced to a lot of extent. There is always a scope for further improvement by implementing multiple prefetch buffers and sophisticated prefetching techniques. Also, a victim cache could be implemented for reducing the conflict misses.

V. References

- Lecture 8 of ‘Advanced Computer Architecture Course’ by Prof. Yanyong Zhang at Rutgers University.
- *Implementing Predictor Directed Stream Buffers in Simplescalar*ByBoulosHarb, Ryan McDonald, Nick Montfort and Kilian Weinberger.
- *Improving Direct-Mapped Cache Performance by Addition of small fully associative cache and Prefetch Buffer* By Jouppi,N.

- Koopman, Philip. *Memory System Architecture*. Lecture notes, Carnegie Mellon, 1998.
- Sing, Joel. Computer Technology – Cache Memory. http://ironbark.bendigo.latrobe.edu.au/subjects/int11ct/2002/lectures/l17/cache.html
- Hennessy, John L. et al. Computer Organization and Design, 2nd Edition. 1998 Morgan Kaufmann Publishers, Inc., p. 552, p.576
- www.cs.ucf.edu/courses/cgs3269.sum01/mem3.ppt