Cache Optimization Techniques

Anna DeVries, Gautham Ranganathan, and Shrikanth Showri Rajan
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA

***Abstract*— Cache performance is an important factor taken into consideration while creating a program. Utilizing the cache effectively results in reduction in latency, increase in throughput and decrease in miss rate. Several optimizations to the cache exist like different replacement policies, set associativities and varying the size of the cache.

In this paper the results and analysis of the behaviour of different cache optimization techniques over several standard benchmarks. The design-based optimization techniques include increasing set associativity, increasing the cache size, increasing the cache block size, changing the replacement algorithm and cache line coloring. The miss rate for the L1 data and instruction caches are reported using the object driven cache simulator SimpleScalar. The cache line coloring optimization is developed using gcc linkers and tested using Pin and the trace driven simulator DineroIV.

From the results, the best configuration for these benchmarks is a 128KB cache size, 32B to 64B cache block size with LRU replacement. Cache line coloring is tested on a small and simple example code which is used to demonstrate its working and the miss rate reduced from 4.54% to 4.15% for with and without using coloring respectively.

***Index Terms*—cache optimizations, cache coloring, design-based optimizations.**

I. Introduction

Each year processor performance improves while memory performance remains fairly stalemate. This gap further increases the importance of exploiting cache memory. Computer systems utilize caches as nearby memory storages, holding recently used data to prevent the costly operation of accessing main memory or secondary storage. Cache misses occur due to cache capacity, collisions and after system reboots. Designers have utilized many different methods to reduce cache miss rate, and ultimately improve memory system performance. These optimizations range from design-based to program layout on the memory space.

This project looks at both design based and code reordering/mapping algorithms for cache optimization. Specifically, the project studies the performance improvements between four unique optimization techniques over three different simple cache design configurations. Each configuration and optimization technique were tested on standard benchmarks. These performance results were then compared on effectiveness.

The paper is separated into five sections and 5 appendices. The sections include introduction, related works, methodology, results and analysis, and conclusion. The appendices include the design-based optimizations script, design-based optimizations results, cache coloring code, cache coloring results, and final project proposal.

II. Related Works

A. Cache Optimization Techniques

The first article, “Cache Memory: An Analysis on Optimization Techniques,” discusses five different cache optimization categories - replacement algorithms, cache algorithms, design based optimizations, compiler and prediction based optimization, and web based optimization [1]. The articles continues to layout a comparison of 16 specific optimization methods, comparing miss rate, miss penalty, hit time, hit rate, power consummation, access time, cost, complexity, and cycle time [1]. This article provides the team with guidance on important factors to measure during benchmark testing, such that an informed comparison can be created between the optimizations. Additionally, the team is implementing three of the optimizations discussed in the article - changing cache size, associativity and block size.

B. Cache Line Coloring Optimization

This paper discusses a link time procedure mapping algorithm which works by creating an improved program layout by color mapping considering the procedure size, cache size, cache line size and call graph. This algorithm reduces the miss rate by 40% from the original mapping. The algorithm intelligently places procedures in the cache layout by preserving color dependencies with a procedure's parents and children in the call graph, resulting in fewer instruction cache conflicts [2]. This paper implements the cache colouring algorithm and uses the ATOM tool for simulations. The paper uses programs like espresso, Perl, li, bison and gzip. The team will carry out a similar study for different workloads on a x86 machine using PIN tool by intel for generating trace.

III. METHODOLOGY

This section describes the project's experimental and testing methods. Methodology is separated into 3 subsections. Section A describes SimpleScalar setup for cache configurations and design-based optimizations, Section B discusses the cache coloring optimization setup, and Section C explains the experimental benchmark testing methods.

A. Design-Based Optimizations

This portion of the project utilized SimpleScalar [5]. This tool provides computer architecture simulations for various features including cache configurations. The first part of this project sought to compare the effectiveness of three design-based optimizations on three various base cases.

Each base case utilizes a 64 KB L1 and 64 KB L2 cache (32 KB data caches and 32 KB instruction caches for each level). Additionally, these base cases have 32 byte block sizes and utilize the FIFO replacement algorithm. The three base cases differ on associativity. There is a direct-mapped, 2-way set associative and 4-way set associative base case. These base cases provide a baseline performance for comparison with the optimizations.

The optimizations include cache size, block size and replacement algorithm. The cache size varied in powers of two, ranging from 64 KB to 32,768 KB. The data and instruction caches for both L1 and L2 levels equally incremented for each test (i.e. at the 64 KB cache size, data and instruction caches in L1 and L2 were each 32 KB in size). The block size also incremented in powers of two, from 64 KB to 32,768 bytes. The block size was also kept consistent across all instruction and data caches in both levels. Finally, the project compared two different replacement algorithms against the base case: LRU and random.

The cache configuration and data gathering script are included in Appendix A. An example is provided below. This SimpleScalar snippet simulates a direct-mapped, 64 KB cache size (32 KB instruction and 32 KB data cache) for both L1 and L2 levels, and 32-byte block size utilizing FIFO replacement algorithm. Additionally, this example uses the cc1 benchmark to gather results.

./simplesim-3.0/sim-cache -cache:dl1 dl1:1024:32:1:f -cache:dl2 dl2:1024:32:1:f -cache:il1 il1:1024:32:1:f -cache:il2 il2:1024:32:1:f simplesim-3.0/benchmarks/compress95.alpha <simplesim-3.0/benchmarks/compress95.in

B. Cache Coloring Optimization

For cache coloring the tools used were Intel Pin[7] to generate the trace to feed into DineroIV[8] which is a trace driven cache simulator. The configuration of the cache for this optimization was 256B cache block size, 1KB cache size and direct mapped.

To perform cache coloring, since this method is a link time optimization the default linker script in gcc should be modified. In the linker script, the (.test) method contains all the code and is involved in the memory mapping of the procedures in the program. By default the procedures are mapped according to the optimizations done by gcc and given by the user. This doesn't provide any control to the user on how procedures are mapped.

This paper demonstrates the working of cache coloring with an example given in the next section whose code is given in Appendix C. This involves manually changing the order of the procedures with the help of some linking switches in gcc [6]. --ffunction-sections makes every procedure referable in the linker script

and -fno-top-level-reordering prevents gcc from ordering procedures according to its default switches. This provides control over mapping each procedure according to the users choice. In the linker script the order of the procedures to be defined is as shown in Figure 1

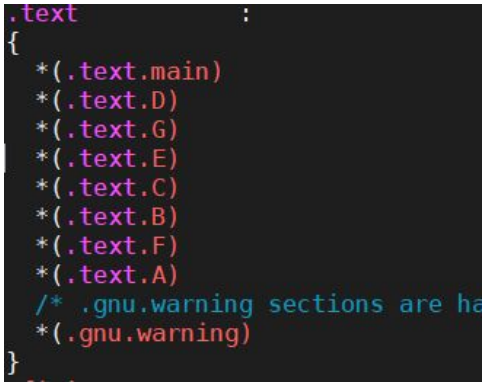


Figure 1: Text section in linker script.

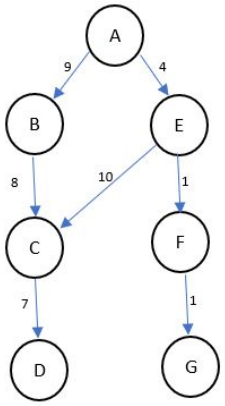
This method takes into consideration the number of times a procedure is called which is called the popularity, the size of each procedure, the order in which they are called and more. This is how the procedures are mapped in the example explained below in the following sections. The rest of the script is kept the same so as to prevent any dependency errors while linking.

Each cache line is given a color and then the procedures are mapped in such a way that no colors overlap between function calls, if they do then it's a cache conflict. Mapping using this method prevents any cache conflicts and this reduces the miss rate considerably.

C. Benchmark Testing

The design-based optimizations were tested with three different benchmarks, each provided by SimpleScalar. These benchmarks were the compress95.alpha, go.alpha and perl.alpha benchmarks [5]. Each of these benchmarks are part of the SPEC95 benchmarks. Compress95 executes approximately 400 million instructions. It generates an in-memory buffer of data, compresses this to another buffer and decompresses the data [3]. Go runs approximately 550 million instructions. It runs an AI algorithm for playing the game "Go" [3]. Finally, the perl benchmark runs various perl scripts, including primes.pl and charcoun, within the provided perl script. Primes.pl identifies primes from a list of number using a brute force algorithm and Charcount counts the numbers of characters in a file [3].

Cache coloring is tested on an example code which is modeled out of the explanation provided in [2] which is given in Appendix C. The code follows the same number of calls and contains dummy code so as to increase the size of each procedure to fulfill its cache line requirement as mentioned in the procedure table. This size constraint was verified using the readelf command which provides with the necessary information regarding size and memory address. The itrace.so example from Intel Pin is used to generate the instruction trace, then since DineroIV accepts the traces only in din format the pin trace should be modified such that all the '0x' at the beginning of each line is replaced with '2 ' which refers to instruction fetch and this was done with sed. No benchmarks are tested for coloring since only the working of cache coloring and its effectiveness is tested and demonstrated. The call graph and procedure table are shown in Figure 2.



Procedure	# Cache Lines
A	1
B	1
C	2
D	2
E	2
F	1
G	2

Figure 2: Call graph and Procedure Table.

The code is compiled with each procedure is aligned to 256B, since the cache is configured to 256B cache block size, each function will either take one cache line or two depending on the procedure table. As mentioned in the previous section the default linker script is modified according to the ideal mapping mentioned in [2] and compiled with that. Therefore two setups are tested, one without cache coloring and the other with. These object files were tested in DineroIV with the trace generated above and the miss rate in the L1 instruction cache is recorded for both with and without cache coloring.

IV. Results and Analysis

This section presents the project's results and analyzes these results. It's divided into design-based optimizations and cache line coloring.

A. Design-Based Optimizations

The complete data set collected during these optimizations is provided below in Appendix B. The results were graphed on six separate graphs: L1 instruction cache miss rate versus cache size, L1 data cache miss rate versus cache size, L1 instruction block miss rate versus cache size, L1 data block miss rate versus cache size, replacement algorithm L1 cache miss rate versus associativity, and replacement algorithms L2 cache miss rate versus associativity. The L2 miss rates varied based upon the replacement algorithms (improving from the base case's FIFO algorithm); however, the L2 miss rates constantly held 100% on all other design-based optimizations. Due to this static miss rate, the L2 caches were not graphed for cache size or block size optimizations.

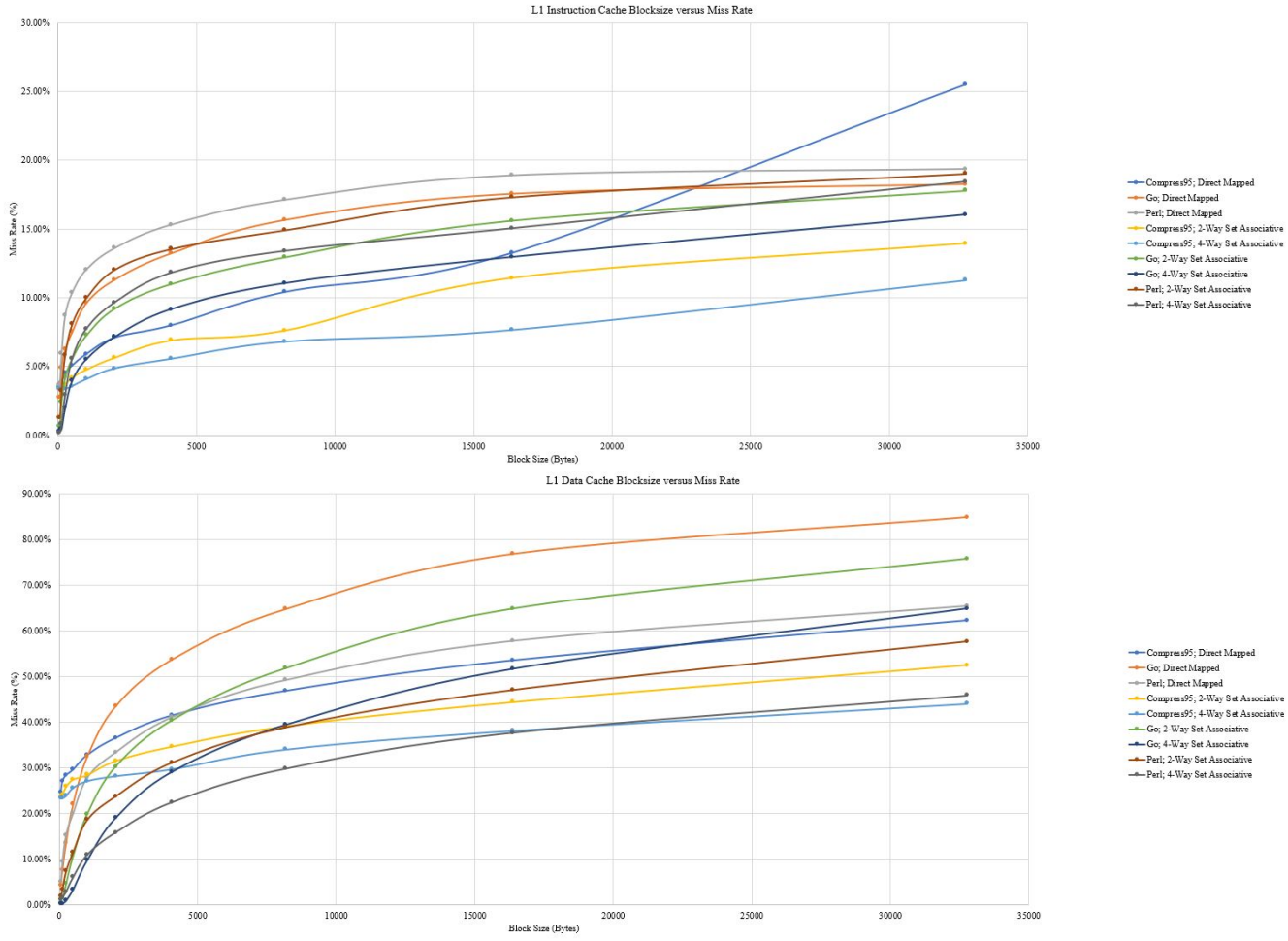


Figure 3: L1 cache blocksize versus miss rate.

Figure 3 above illustrates the L1 instruction and data cache miss rate versus increasing block size. The miss rate is represented as a percentage of misses / cache accesses; the block size is measured in bytes and represents the block size configured for each cache. For both caches, the general trend consisted of an increasing miss rate as blocksize increased.

Larger blocksize allows users to exploit spatial locality, capturing data the program needs and the data surrounding it; however, keeping the cache size static while increasing blocksize decreases the number of blocks within a cache [4]. This leads to fewer sets and, therefore, fewer hits. This also allows for greater cache pollution, polling unneeded data into cache. Given the 32 KB cache size, smaller blocksize is necessary for accuracy.

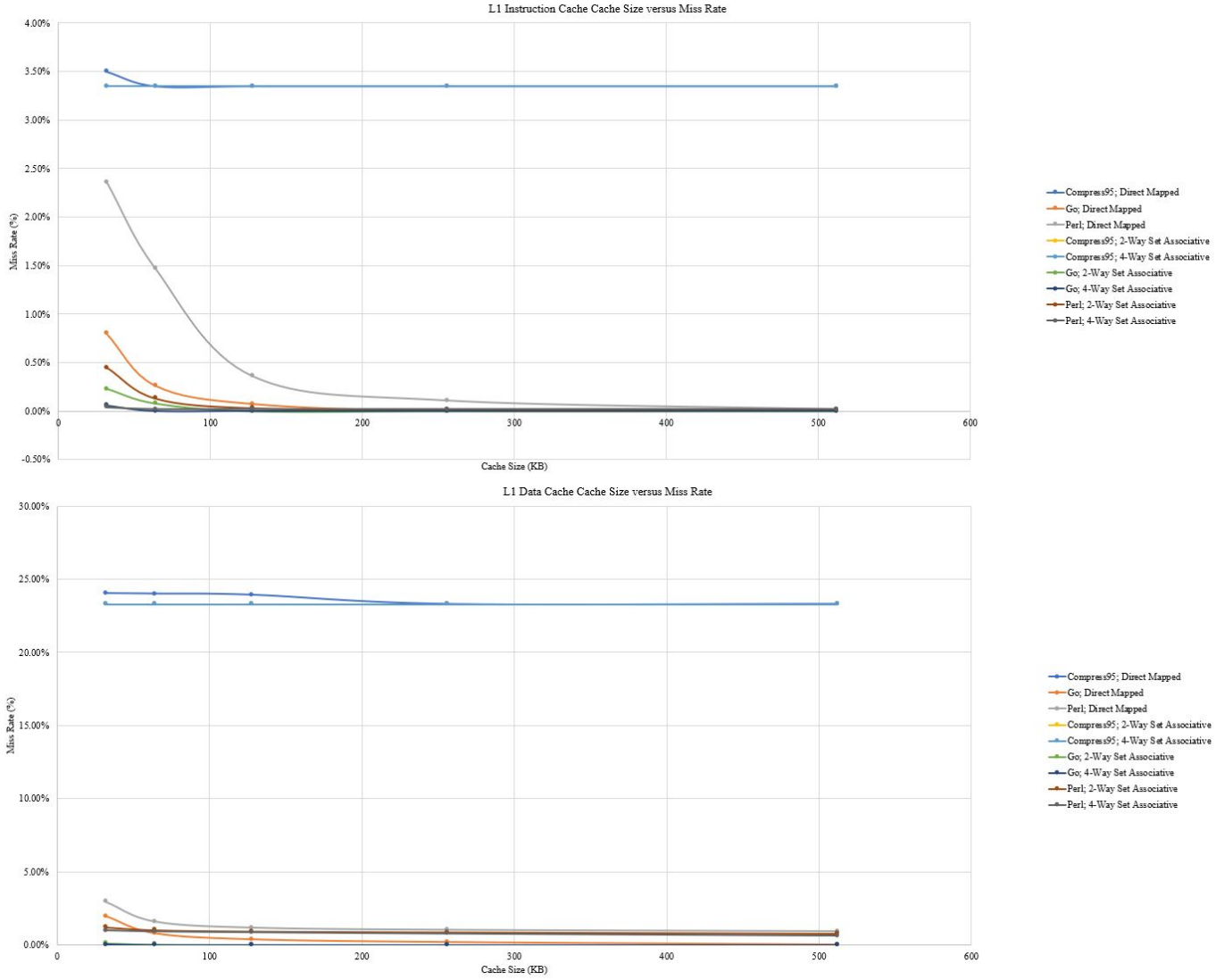


Figure 4: L1 cache size versus miss rate.

Figure 4 above illustrates the L1 instruction and data cache miss rate versus increasing cache size. The cache size incremented in powers of two, ranging from 32 KB to 16384 KB (i.e. 32 KB L1 data cache and 32 KB L1 instruction cache to 16384 KB L1 data cache and 16384 KB L1 instruction cache). The graph is zoomed in between 32 KB and 512 KB because the miss rates following the 512 KB cache sizes yielded the same results. The general trend showed a decrease in miss rate for increased cache size; however, the logarithmic trend eventually showed diminishing returns on the larger cache size.

Larger caches allow for more information to be stored near the processor. While this creates less misses, appearing to increase performance, every optimization comes with tradeoffs. Large cache sizes increase complexity, cost, and access times. Miss rate decreases, but access time and miss penalty will increase.

Additionally, increased cache size only improved performance up to a size of 128 KB (64 KB data and instruction caches). After 128 KB, the performance remained steady with a miss rate of approximately 3.35% for the majority of the tests. The program size may contribute to this diminishing returns in performance. The additional size is only as useful as the amount of memory being grabbed and utilized again. Larger cache sizes may perform better with more aggressive predictive caching and larger program sizes.

Figure 5 below illustrates the L1 and L2 caches' miss rates based upon replacement algorithms. The y-axis contains miss rate as a percentage and the x-axis denotes the level of associativity (1 = direct mapped, 2 = two-way set associative, 4 = four-way set associative).

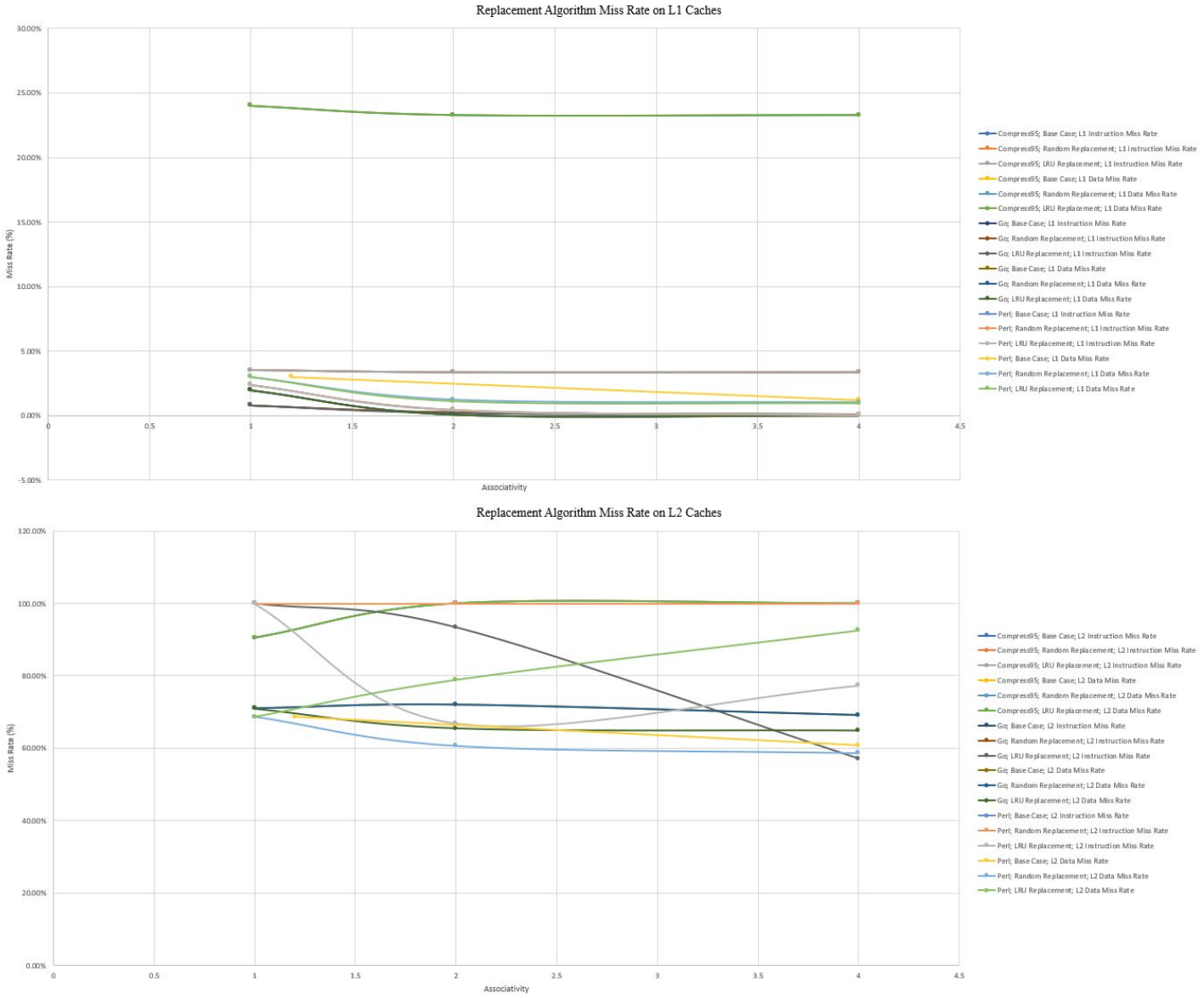


Figure 5: Replacement algorithm miss rate.

In these benchmarks, the replacement algorithms created similar results on the L1 caches, with the exception of the LRU replacement algorithm on the L1 data cache during the compress95 benchmark. The general trend showed an increase in performance with higher levels of associativity for each replacement algorithm and an increase in performance using the LRU replacement algorithm over FIFO or random. For example, during the go benchmark, the L1 instruction cache produced a miss rate of 0.80% on a direct-mapped cache and a miss rate of 0.05% on a four-way set associative cache with an LRU replacement algorithm. The FIFO and random replacement algorithms produced a miss rate of 0.80% on a direct-mapped cache as well but a 0.06% miss rate on a four-way set associative cache.

Random replacement refers to randomly selecting which blocks of data to remove from the cache. FIFO stands for first-in first-out. This algorithm behaves similar to a queue; the blocks are evicted in the order they were added. The FIFO algorithm does not account of how many times or how recently the data was accessed, only the order it was first added. However, LRU (which stands for least recently used) evicts blocks from the cache that were least recently accessed. In general, LRU outperforms FIFO or random replacement algorithms due to the nature of real-world data -- recently used data has a higher chance of being used again in a program, independent of when it was added to the cache.

The results of the design-based optimizations indicate that a cache size of 128 KB, block size ranging between 32 - 64 bytes and an LRU replacement algorithm produce the lowest miss rate.

B. Cache Coloring Optimization

The trace is generated from each of the compiled objects using the itrace code in Intel Pin. This trace is piped into the DineroIV cache simulator with the configuration of 256B cache block size, 1KB cache size and direct mapped.

For the code compiled without cache coloring it can be observed from Figure 6 below that the miss rate was 4.54% and with cache coloring it is 4.15% as provided in Appendix D. This proves that re ordering procedures using color mapping helps in reducing cache conflicts which in turn reduce miss rate.

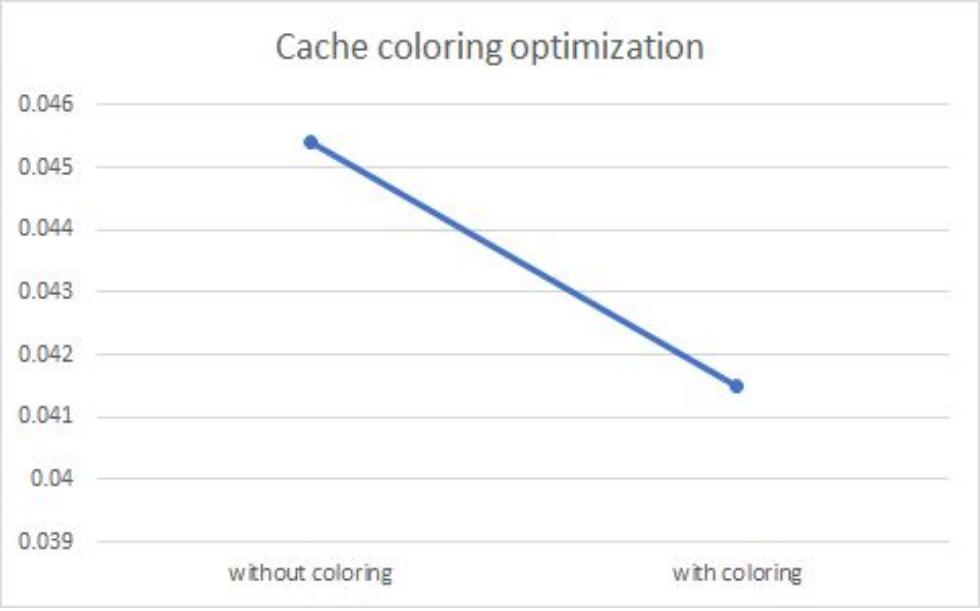
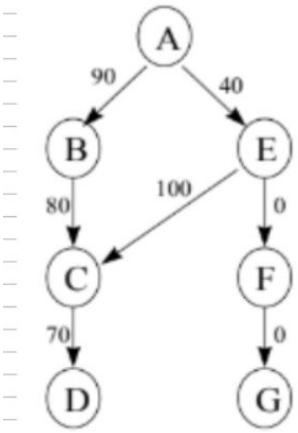


Figure 6: Cache coloring result.

The explanation below demonstrates the working with and without cache coloring for the example code used.



Random ordering of functions in address space

	A	C	E	F	B	D	G

Program Flow	Cache contents				Replacement	Miss/Hit
A	A	X	X	X		Compulsory Miss
A → B	B	X	X	X	B replaces A	Compulsory Miss
B → C	B	C	X	X		Compulsory Miss
C → D	B	D	X	X	D replaces C	Compulsory Miss
D → A	A	D	X	X	A replaces B	Conflict Miss for A
A → E	A	D	E	X		Compulsory Miss
E → C	A	C	E	X	C replaces D	Conflict miss for C
C → D	A	D	E	X	D replaces C	Conflict miss for D
D → A	A	D	E	X		Cache hit for A
Cache Hits	1		Conflict Miss	3		

After Reordering functions in address space (ROM address)

	A	B	C	D	F	E	G

Program Flow	Cache contents				Replacement	Miss/Hit
A	A	X	X	X		Compulsory Miss
A → B	A	B	X	X		Compulsory Miss
B → C	A	B	C	X		Compulsory Miss
C → D	A	B	C	D		Compulsory Miss
D → A	A	B	C	D		Cache hit for A
A → E	A	E	C	D	E replaces B	Compulsory Miss
E → C	A	E	C	D		Cache hit for C
C → D	A	E	C	D		Cache hit for D
D → A	A	E	C	D		Cache hit for A
Cache Hits	4		Conflict Miss	0		

Figure 7: Reordering in address space

First, we show the effect of reordering subroutines in address space based on the program flow. As an example, let's consider the program flow shown in Figure 7 and simulate how the cache is populated with these functions as they are executed. We assume each function occupies only one cache block for this example and a cache size of 4 blocks. In case 1, we place the functions randomly in address space and simulate how they are populated in cache and in case 2 we reorder based on program flow and place the caller next to the calling function.

After reordering, we observe that the number of conflict misses is lesser. Different workloads might have different nature of program flow and more paths between subroutines. To optimize our placement in address space for such cases, we calculate and assign a weight-age to each edge of the graph. This serves as a proof of concept that we can reduce the cache miss rate by placing the Calling functions and called functions in calculated locations of the ROM so that we can minimize the occurrences of them thrashing each other in the cache. This is likely to cause some large gaps in the memory and we will need some mechanism to stitch these distant memory locations together. A strong compiler logic to place the less popular functions to fill these gaps can help us resolve this.

V. Conclusion

This project determined that the optimal cache design utilizes a 128 KB cache size, 32 - 64 byte block size, 4-way set associative with LRU replacement algorithm and utilizing cache coloring. We see that implementing cache line coloring in a program can improve hit rate but this comes at the expense of spatial locality, the functions in the program are not in calling order in the memory but the mapping is done in such a way that the reduction in miss rate outweighs spatial locality. Furthermore, this project illustrated the diminishing returns each optimization gave. This illuminates that the optimizations should be combined in new ways to further exploit the cache memory.

One recommendation for future testing on design-based optimizations would be to compare more aggressive to more conservative predictive caching on varying sizes of cache. For systems with larger caches, aggressive predictive caching may increase performance. However, this system would be more expensive and require large enough programs to effectively improve memory.

For future work on cache line coloring the granularity of mapping can be increased from procedures to basic blocks or loops in the program so as to provide a more effective utilization of the cache. This method can be extended to set associative caches as increasing associativity to a limit can reduce the miss rate and implementing coloring on top of this can decrease the miss rate further.

References

[1] M. W. Ahmed and M. A. Shah, “Cache Memory: An analysis on Optimization Techniques,” *International Journal of Computer and Information Technology*, vol. 4, no. 2, pp. 414 - 418, March 2015.

[2] A. H. Hashemi, D. R. Kaeli, and B. Calder, “Efficient Procedure Mapping Using Cache Line Coloring,” *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[3] “SPEC95 Benchmarks for SimpleScalar,” University of Washington Computer Science & Engineering. https://courses.cs.washington.edu/courses/cse471/09sp/sim/benchmark-guide.pdf

[4] J. Hennessy and D. Patterson, “Computer Architecture: A Quantitative Approach Sixth Edition,” 2019 Elserview Inc: Cambridge, MA.

[5] www.simplescalar.com

[6] https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

[7] https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/

[8] http://course.ece.cmu.edu/~ece548/tools/dinero/man/dinero.htm

Appendix A: Data Gathering Automation

```
#!/bin/bash

# Note: This simulation prints to stderr
# Array of all benches to run
allBenches=('compress95' 'go' 'perl') # removed
cc1 and anagram

# Set Associativity
allAssoc=(1 2 4)

# Iterate through each bench test
for bench in ${allBenches[@]};
do
       if [ $bench = "cc1" ]; then
       extra='-o /dev/null'

       elif [ $bench = 'compress95' ]; then

extra='<simplesim-3.0/benchmarks/compress95.in'
       elif [ $bench = 'go' ]; then
       extra='50 9
simplesim-3.0/benchmarks/2stone9.in'
       elif [ $bench = 'perl' ]; then

extra='simplesim-3.0/benchmarks/perl-tests.pl'
       fi

       echo "Benchmark: " $bench

       # Title file
       file_name=results_$bench.csv

       # Add bench.alpha
       bench=$bench.alpha

       # Empty file
       > $file_name

       # Place headers
       { echo
"title,cache_size,set_number,block_size,replacemen
t_type,associativity,il1.miss_rate,il1.repl_rate,i
l2.miss_rate,il2.repl_rate,dl1.miss_rate,dl1.repl_
rate,dl2.miss_rate,dl2.repl_rate"; } > $file_name

       for asso in ${allAssoc[@]};
       do
       echo "Associativity: " $asso

       # Replacement algorithm
       repl='f'

       # Base Case
       echo " "
       echo "Printing Base Case"
       echo "-------------------------"

       # Variables
       title=base_case
       cache_size=32
       set_number=1024
       bsize=32

       { printf
"$title,$cache_size,$set_number,$bsize,$repl,$asso
,"; } >> $file_name
       ./simplesim-3.0/sim-cache -cache:dl1
dl1:$set_number:$bsize:$asso:$repl -cache:dl2
dl2:$set_number:$bsize:$asso:$repl -cache:il1
il1:$set_number:$bsize:$asso:$repl -cache:il2
il2:$set_number:$bsize:$asso:$repl
simplesim-3.0/benchmarks/$bench $extra 2> >( grep
"il1.miss_rate\|il1.repl_rate\|il2.miss_rate\|il2.
repl_rate\|dl1.miss_rate\|dl1.repl_rate\|dl2.miss_
rate\|dl2.repl_rate" ) | awk "{print \$2}" | tr
'\n' ',' | sed 's/,\?$/\n/' >> $file_name


       # Optimization: Larger Cache
       echo " "
       echo "Printing Optimization: Larger Cache"
       echo "-------------------------"

       # Variables
       title=cache_size
       set_number=512
       bsize=32

       for i in `seq 1 10`;
       do
            set_number=$(($set_number<<1))

cache_size=$(($(($set_number/1024))*32));
            { printf
"$title,$cache_size,$set_number,$bsize,$repl,$asso
,"; } >> $file_name
            ./simplesim-3.0/sim-cache -cache:dl1
dl1:$set_number:$bsize:$asso:$repl -cache:dl2
dl2:$set_number:$bsize:$asso:$repl -cache:il1
il1:$set_number:$bsize:$asso:$repl -cache:il2
il2:$set_number:$bsize:$asso:$repl
simplesim-3.0/benchmarks/$bench $extra 2> >( grep
"il1.miss_rate\|il1.repl_rate\|il2.miss_rate\|il2.
repl_rate\|dl1.miss_rate\|dl1.repl_rate\|dl2.miss_
rate\|dl2.repl_rate" ) | awk "{print \$2}" | tr
'\n' ',' | sed 's/,\?$/\n/' >> $file_name
       done


       # Optimization: Larger Block Size
       echo " "
       echo "Printing Optimization: Larger Block
Size"
       echo "-------------------------"

       # Variables
       title=block_size
       set_number=1024
       cache_size=32
       bsize=32

       for i in `seq 1 10`;
       do
            set_number=$(($set_number>>1))

block_size=$(($((32*1024))/$set_number));
            { printf
"$title,$cache_size,$set_number,$bsize,$repl,$asso
,"; } >> $file_name
            ./simplesim-3.0/sim-cache -cache:dl1
dl1:$set_number:$bsize:$asso:$repl -cache:dl2
dl2:$set_number:$bsize:$asso:$repl -cache:il1
il1:$set_number:$bsize:$asso:$repl -cache:il2
```

```
il2:$set_number:$bsize:$asso:$repl
simplesim-3.0/benchmarks/$bench $extra 2> >( grep
"il1.miss_rate\|il1.repl_rate\|il2.miss_rate\|il2.
repl_rate\|dl1.miss_rate\|dl1.repl_rate\|dl2.miss_
rate\|dl2.repl_rate" ) | awk "{print \$2}" | tr
'\n' ',' | sed 's/,\?$/\n/' >> $file_name
        done


        # Replacement Algorithm Optimization
        # Replacement algorithm
        repl='r'

        # Base Case
        echo " "
        echo "Printing Replacement Algorithm
Optimization: Random"
        echo "-------------------------"

        # Variables
        title=random_case
        cache_size=32
        set_number=1024
        bsize=32

        { printf
"$title,$cache_size,$set_number,$bsize,$repl,$asso
,"; } >> $file_name
        ./simplesim-3.0/sim-cache -cache:dl1
dl1:$set_number:$bsize:$asso:$repl -cache:dl2
dl2:$set_number:$bsize:$asso:$repl -cache:il1
il1:$set_number:$bsize:$asso:$repl -cache:il2
il2:$set_number:$bsize:$asso:$repl
simplesim-3.0/benchmarks/$bench $extra 2> >( grep
"il1.miss_rate\|il1.repl_rate\|il2.miss_rate\|il2.
repl_rate\|dl1.miss_rate\|dl1.repl_rate\|dl2.miss_
rate\|dl2.repl_rate" ) | awk "{print \$2}" | tr
'\n' ',' | sed 's/,\?$/\n/' >> $file_name


        # Replacement Algorithm Optimization
        # Replacement algorithm
        repl='l'

        # Base Case
        echo " "
        echo "Printing Replacement Algorithm
Optimization: LRU"
        echo "-------------------------"

        # Variables
        title=lru_case
        cache_size=32
        set_number=1024
        bsize=32

        { printf
"$title,$cache_size,$set_number,$bsize,$repl,$asso
,"; } >> $file_name
        ./simplesim-3.0/sim-cache -cache:dl1
dl1:$set_number:$bsize:$asso:$repl -cache:dl2
dl2:$set_number:$bsize:$asso:$repl -cache:il1
il1:$set_number:$bsize:$asso:$repl -cache:il2
il2:$set_number:$bsize:$asso:$repl
simplesim-3.0/benchmarks/$bench $extra 2> >( grep
"il1.miss_rate\|il1.repl_rate\|il2.miss_rate\|il2.
repl_rate\|dl1.miss_rate\|dl1.repl_rate\|dl2.miss_
rate\|dl2.repl_rate" ) | awk "{print \$2}" | tr
'\n' ',' | sed 's/,\?$/\n/' >> $file_name
        done
done

# deubbing: ./simplesim-3.0/sim-cache -cache:dl1
dl1:1024:32:1:f -cache:dl2 ul2:1024:32:1:f
-cache:il1 dl1:1024:32:1:f -cache:il2
dl2:1024:32:1:f simplesim-3.0/benchmarks/cc1.alpha
-O simplesim-3.0/benchmarks/1stmt.i
```

Appendix B: Design-Based Optimization Data

Benchmark: Compress95										
Direct-Mapped Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	3.50%	0.57%	100.00%	16.21%	24.03%	2.97%	90.49%	11.19%
32	32	r	3.50%	0.57%	100.00%	16.21%	24.03%	2.97%	90.49%	11.19%
32	32	l	3.50%	0.57%	100.00%	16.21%	24.03%	2.97%	90.49%	11.19%
32	32	f	3.50%	0.57%	100.00%	16.21%	24.03%	2.97%	90.49%	11.19%
64	32	f	3.35%	0.00%	100.00%	0.00%	24.00%	2.60%	91.50%	9.92%
128	32	f	3.35%	0.00%	100.00%	0.00%	23.92%	2.45%	92.00%	9.43%
256	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
512	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
1024	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
2048	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
4096	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
8192	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
16384	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	64	f	3.50%	0.84%	100.00%	24.16%	24.63%	9.92%	73.91%	29.77%
32	128	f	3.82%	1.79%	100.00%	46.78%	27.04%	17.53%	66.06%	42.83%
32	256	f	4.50%	3.21%	100.00%	71.26%	28.31%	23.55%	60.67%	50.48%
32	512	f	5.07%	4.38%	100.00%	86.50%	29.53%	27.15%	58.93%	54.19%
32	1024	f	5.90%	5.56%	100.00%	94.20%	32.76%	31.58%	59.72%	57.55%
32	2048	f	7.10%	6.93%	100.00%	97.59%	36.48%	35.88%	61.18%	60.19%
32	4096	f	7.99%	7.90%	100.00%	98.93%	41.46%	41.16%	63.02%	62.56%
32	8192	f	10.41%	10.36%	100.00%	99.59%	46.84%	46.69%	65.30%	65.10%
32	16384	f	13.25%	13.23%	100.00%	99.84%	53.53%	53.45%	66.99%	66.90%
32	32768	f	25.52%	25.51%	100.00%	99.96%	62.30%	62.26%	67.95%	67.91%
2 Way-Set Associative Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	32	r	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	32	l	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
64	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%

128	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
256	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
512	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
1024	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
2048	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
4096	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
8192	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
16384	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	64	f	3.35%	0.05%	100.00%	1.60%	23.48%	0.85%	96.93%	3.53%
32	128	f	3.38%	0.27%	100.00%	7.91%	24.07%	6.87%	80.50%	22.98%
32	256	f	3.69%	1.28%	100.00%	34.78%	25.85%	16.34%	66.60%	42.11%
32	512	f	4.19%	2.83%	100.00%	67.60%	27.45%	22.70%	60.23%	49.80%
32	1024	f	4.77%	4.09%	100.00%	85.65%	28.42%	26.04%	58.31%	53.43%
32	2048	f	5.64%	5.29%	100.00%	93.93%	31.50%	30.31%	59.14%	56.90%
32	4096	f	6.90%	6.73%	100.00%	97.52%	34.66%	34.06%	60.04%	59.01%
32	8192	f	7.62%	7.54%	100.00%	98.88%	39.08%	38.78%	61.81%	61.34%
32	16384	f	11.44%	11.40%	100.00%	99.63%	44.35%	44.21%	64.23%	64.01%
32	32768	f	13.96%	13.93%	100.00%	99.85%	52.45%	52.38%	67.43%	67.34%

4 Way-Set Associative Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	32	r	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	32	l	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
64	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
128	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
256	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
512	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
1024	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
2048	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
4096	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
8192	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
16384	32	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	64	f	3.35%	0.00%	100.00%	0.00%	23.29%	0.00%	100.00%	0.00%
32	128	f	3.35%	0.00%	100.00%	0.00%	23.33%	0.26%	98.90%	1.10%
32	256	f	3.35%	0.04%	100.00%	1.28%	23.81%	5.76%	83.79%	20.26%

32	512	f	3.57%	1.02%	100.00%	28.44%	25.59%	16.08%	66.00%	41.48%
32	1024	f	4.07%	2.71%	100.00%	66.40%	27.08%	22.33%	59.90%	49.38%
32	2048	f	4.86%	4.17%	100.00%	85.90%	28.16%	25.78%	57.91%	53.02%
32	4096	f	5.56%	5.22%	100.00%	93.85%	29.68%	28.49%	57.90%	55.58%
32	8192	f	6.81%	6.64%	100.00%	97.49%	33.99%	33.40%	59.80%	58.76%
32	16384	f	7.66%	7.57%	100.00%	98.88%	38.08%	37.78%	61.38%	60.90%
32	32768	f	11.28%	11.24%	100.00%	99.62%	43.98%	43.83%	63.90%	63.68%

Benchmark: Go										
Direct-Mapped Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	0.80%	0.80%	100.00%	99.98%	1.97%	1.97%	70.95%	70.93%
32	32	r	0.80%	0.80%	100.00%	99.98%	1.97%	1.97%	70.95%	70.93%
32	32	l	0.80%	0.80%	100.00%	99.98%	1.97%	1.97%	70.95%	70.93%
32	32	f	0.80%	0.80%	100.00%	99.98%	1.97%	1.97%	70.95%	70.93%
64	32	f	0.26%	0.26%	100.00%	99.86%	0.79%	0.79%	73.94%	73.85%
128	32	f	0.07%	0.07%	100.00%	98.97%	0.37%	0.37%	70.75%	70.39%
256	32	f	0.00%	0.00%	100.00%	46.06%	0.18%	0.18%	71.68%	70.21%
512	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.01%	90.93%	55.03%
1024	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	99.77%	10.31%
2048	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	10.03%
4096	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	10.03%
8192	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	10.03%
16384	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	10.03%
32	64	f	2.74%	2.74%	100.00%	100.00%	4.22%	4.22%	70.83%	70.83%
32	128	f	4.90%	4.90%	100.00%	100.00%	7.65%	7.65%	72.17%	72.17%
32	256	f	6.23%	6.23%	100.00%	100.00%	13.54%	13.54%	73.87%	73.87%
32	512	f	7.52%	7.52%	100.00%	100.00%	22.04%	22.04%	74.85%	74.85%
32	1024	f	9.63%	9.63%	100.00%	100.00%	32.46%	32.46%	76.53%	76.53%
32	2048	f	11.32%	11.32%	100.00%	100.00%	43.47%	43.47%	78.29%	78.29%
32	4096	f	13.24%	13.24%	100.00%	100.00%	53.77%	53.77%	79.77%	79.77%
32	8192	f	15.66%	15.66%	100.00%	100.00%	64.77%	64.77%	81.22%	81.22%
32	16384	f	17.56%	17.56%	100.00%	100.00%	76.83%	76.83%	82.38%	82.38%
32	32768	f	18.27%	18.27%	100.00%	100.00%	84.90%	84.90%	83.19%	83.19%
2 Way-Set Associative Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size	Block Size	Replacement	Miss Rate	Replacement	Miss Rate	Replacement	Miss Rate	Replacement	Miss Rate	Replacement

(KB)	(bytes)	Algorithm		Rate		Rate		Rate		Rate
32	32	f	0.23%	0.23%	100.00%	99.84%	0.10%	0.10%	71.95%	71.28%
32	32	r	0.23%	0.23%	100.00%	99.84%	0.10%	0.10%	71.95%	71.28%
32	32	l	0.22%	0.22%	93.40%	93.23%	0.09%	0.09%	65.44%	64.68%
32	32	f	0.23%	0.23%	100.00%	99.84%	0.10%	0.10%	71.95%	71.28%
64	32	f	0.08%	0.08%	100.00%	99.07%	0.04%	0.04%	67.83%	64.90%
128	32	f	0.00%	0.00%	100.00%	38.35%	0.02%	0.02%	68.72%	56.25%
256	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	84.57%	18.91%
512	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	99.71%	2.80%
1024	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
2048	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
4096	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
8192	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
16384	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
32	64	f	0.65%	0.65%	100.00%	99.97%	0.42%	0.42%	67.32%	67.24%
32	128	f	2.45%	2.45%	100.00%	100.00%	1.46%	1.46%	68.46%	68.45%
32	256	f	4.20%	4.20%	100.00%	100.00%	4.52%	4.52%	69.57%	69.57%
32	512	f	5.59%	5.59%	100.00%	100.00%	10.67%	10.67%	71.37%	71.37%
32	1024	f	7.27%	7.27%	100.00%	100.00%	19.84%	19.84%	73.64%	73.64%
32	2048	f	9.22%	9.22%	100.00%	100.00%	30.24%	30.24%	75.71%	75.71%
32	4096	f	11.01%	11.01%	100.00%	100.00%	40.43%	40.43%	77.37%	77.37%
32	8192	f	12.95%	12.95%	100.00%	100.00%	51.78%	51.78%	79.09%	79.09%
32	16384	f	15.61%	15.61%	100.00%	100.00%	64.82%	64.82%	81.01%	81.01%
32	32768	f	17.79%	17.79%	100.00%	100.00%	75.78%	75.78%	82.29%	82.29%
4 Way-Set Associative Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	0.06%	0.06%	100.00%	98.79%	0.02%	0.02%	69.12%	63.47%
32	32	r	0.06%	0.06%	100.00%	98.79%	0.02%	0.02%	69.12%	63.47%
32	32	l	0.05%	0.05%	57.13%	55.74%	0.02%	0.02%	64.81%	57.99%
32	32	f	0.06%	0.06%	100.00%	98.79%	0.02%	0.02%	69.12%	63.47%
64	32	f	0.00%	0.00%	100.00%	23.87%	0.01%	0.01%	68.88%	43.95%
128	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	88.60%	11.58%
256	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
512	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
1024	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
2048	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%

4096	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
8192	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
16384	32	f	0.00%	0.00%	100.00%	0.00%	0.01%	0.00%	100.00%	0.00%
32	64	f	0.26%	0.26%	100.00%	99.85%	0.08%	0.08%	72.60%	71.76%
32	128	f	0.46%	0.46%	100.00%	99.96%	0.22%	0.22%	67.92%	67.77%
32	256	f	1.98%	1.98%	100.00%	100.00%	0.86%	0.86%	66.37%	66.35%
32	512	f	3.98%	3.98%	100.00%	100.00%	3.31%	3.31%	67.42%	67.42%
32	1024	f	5.52%	5.52%	100.00%	100.00%	9.72%	9.72%	70.23%	70.22%
32	2048	f	7.15%	7.15%	100.00%	100.00%	19.00%	19.00%	73.09%	73.09%
32	4096	f	9.17%	9.17%	100.00%	100.00%	29.14%	29.14%	75.30%	75.30%
32	8192	f	11.05%	11.05%	100.00%	100.00%	39.42%	39.42%	76.93%	76.93%
32	16384	f	12.95%	12.95%	100.00%	100.00%	51.65%	51.65%	78.91%	78.91%
32	32768	f	16.03%	16.03%	100.00%	100.00%	64.87%	64.87%	81.09%	81.09%

Benchmark: Perl										
Direct-Mapped Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	2.36%	2.35%	100.00%	99.83%	2.98%	2.97%	68.65%	68.39%
32	32	r	2.36%	2.35%	100.00%	99.83%	2.98%	2.97%	68.65%	68.39%
32	32	l	2.36%	2.35%	100.00%	99.83%	2.98%	2.97%	68.65%	68.39%
32	32	f	2.36%	2.35%	100.00%	99.83%	2.98%	2.97%	68.65%	68.39%
64	32	f	1.47%	1.47%	100.00%	99.46%	1.61%	1.58%	63.21%	62.32%
128	32	f	0.36%	0.35%	100.00%	96.31%	1.18%	1.13%	60.08%	57.77%
256	32	f	0.11%	0.09%	100.00%	82.42%	1.03%	0.94%	59.41%	54.17%
512	32	f	0.02%	0.00%	100.00%	0.00%	0.93%	0.74%	60.41%	48.57%
1024	32	f	0.02%	0.00%	100.00%	0.00%	0.82%	0.45%	67.06%	37.19%
2048	32	f	0.02%	0.00%	100.00%	0.00%	0.67%	0.03%	99.29%	5.05%
4096	32	f	0.02%	0.00%	100.00%	0.00%	0.66%	0.03%	99.99%	4.29%
8192	32	f	0.02%	0.00%	100.00%	0.00%	0.66%	0.03%	99.99%	4.29%
16384	32	f	0.02%	0.00%	100.00%	0.00%	0.66%	0.03%	99.99%	4.29%
32	64	f	3.64%	3.64%	100.00%	99.94%	4.89%	4.89%	71.38%	71.30%
32	128	f	5.93%	5.93%	100.00%	99.98%	9.40%	9.39%	73.93%	73.91%
32	256	f	8.70%	8.70%	100.00%	99.99%	15.25%	15.25%	72.11%	72.10%
32	512	f	10.37%	10.37%	100.00%	100.00%	20.01%	20.01%	71.04%	71.04%
32	1024	f	12.05%	12.05%	100.00%	100.00%	27.81%	27.81%	71.46%	71.46%
32	2048	f	13.60%	13.60%	100.00%	100.00%	33.35%	33.35%	71.35%	71.35%

32	4096	f	15.31%	15.31%	100.00%	100.00%	41.07%	41.07%	71.62%	71.62%
32	8192	f	17.11%	17.11%	100.00%	100.00%	49.18%	49.18%	72.51%	72.51%
32	16384	f	18.89%	18.89%	100.00%	100.00%	57.73%	57.73%	72.87%	72.87%
32	32768	f	19.36%	19.36%	100.00%	100.00%	65.39%	65.39%	72.97%	72.97%
2 Way-Set Associative Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	0.45%	0.44%	100.00%	98.21%	1.21%	1.18%	60.72%	59.57%
32	32	r	0.45%	0.44%	100.00%	98.21%	1.21%	1.18%	60.72%	59.57%
32	32	l	0.39%	0.39%	66.89%	64.83%	1.09%	1.07%	78.87%	77.64%
32	32	f	0.45%	0.44%	100.00%	98.21%	1.21%	1.18%	60.72%	59.57%
64	32	f	0.13%	0.12%	100.00%	88.67%	1.02%	0.98%	58.87%	56.26%
128	32	f	0.03%	0.01%	100.00%	26.76%	0.93%	0.84%	58.26%	52.60%
256	32	f	0.02%	0.00%	100.00%	0.00%	0.87%	0.69%	59.07%	46.72%
512	32	f	0.02%	0.00%	100.00%	0.00%	0.78%	0.42%	66.12%	35.39%
1024	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.01%	99.31%	0.86%
2048	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
4096	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
8192	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
16384	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
32	64	f	1.27%	1.27%	100.00%	99.68%	1.82%	1.81%	65.03%	64.63%
32	128	f	3.23%	3.23%	100.00%	99.94%	3.40%	3.40%	69.34%	69.22%
32	256	f	5.83%	5.83%	100.00%	99.98%	7.35%	7.34%	71.42%	71.40%
32	512	f	8.10%	8.10%	100.00%	99.99%	11.52%	11.52%	71.46%	71.45%
32	1024	f	9.98%	9.98%	100.00%	100.00%	18.61%	18.61%	70.92%	70.92%
32	2048	f	12.04%	12.04%	100.00%	100.00%	23.74%	23.74%	70.63%	70.63%
32	4096	f	13.54%	13.54%	100.00%	100.00%	31.16%	31.16%	70.06%	70.06%
32	8192	f	14.91%	14.91%	100.00%	100.00%	38.83%	38.83%	71.09%	71.09%
32	16384	f	17.30%	17.30%	100.00%	100.00%	47.01%	47.01%	72.69%	72.69%
32	32768	f	19.01%	19.01%	100.00%	100.00%	57.61%	57.61%	72.51%	72.51%
4 Way-Set Associative Cache			L1 Instruction Cache		L2 Instruction Cache		L1 Data Cache		L2 Data Cache	
Cache Size (KB)	Block Size (bytes)	Replacement Algorithm	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate	Miss Rate	Replacement Rate
32	32	f	0.04%	0.03%	100.00%	63.62%	1.00%	0.96%	58.73%	56.07%
32	32	r	0.04%	0.03%	100.00%	63.62%	1.00%	0.96%	58.73%	56.07%
32	32	l	0.03%	0.02%	77.37%	32.05%	0.92%	0.88%	92.56%	89.74%
32	32	f	0.04%	0.03%	100.00%	63.62%	1.00%	0.96%	58.73%	56.07%
64	32	f	0.02%	0.00%	100.00%	9.18%	0.93%	0.84%	58.15%	52.49%

128	32	f	0.02%	0.00%	100.00%	0.00%	0.87%	0.68%	58.68%	46.36%
256	32	f	0.02%	0.00%	100.00%	0.00%	0.78%	0.42%	65.88%	35.32%
512	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	99.91%	0.16%
1024	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
2048	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
4096	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
8192	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
16384	32	f	0.02%	0.00%	100.00%	0.00%	0.64%	0.00%	100.00%	0.00%
32	64	f	0.17%	0.17%	100.00%	95.36%	1.15%	1.12%	59.98%	58.79%
32	128	f	0.83%	0.83%	100.00%	99.51%	1.54%	1.53%	63.29%	62.83%
32	256	f	2.94%	2.94%	100.00%	99.93%	2.80%	2.79%	68.69%	68.55%
32	512	f	5.59%	5.59%	100.00%	99.98%	6.00%	6.00%	72.34%	72.31%
32	1024	f	7.74%	7.74%	100.00%	99.99%	10.93%	10.93%	71.77%	71.76%
32	2048	f	9.66%	9.66%	100.00%	100.00%	15.80%	15.80%	70.99%	70.99%
32	4096	f	11.82%	11.82%	100.00%	100.00%	22.46%	22.46%	69.46%	69.46%
32	8192	f	13.40%	13.40%	100.00%	100.00%	29.81%	29.81%	69.38%	69.38%
32	16384	f	15.05%	15.05%	100.00%	100.00%	37.67%	37.67%	70.86%	70.86%
32	32768	f	18.45%	18.45%	100.00%	100.00%	45.86%	45.86%	72.71%	72.71%

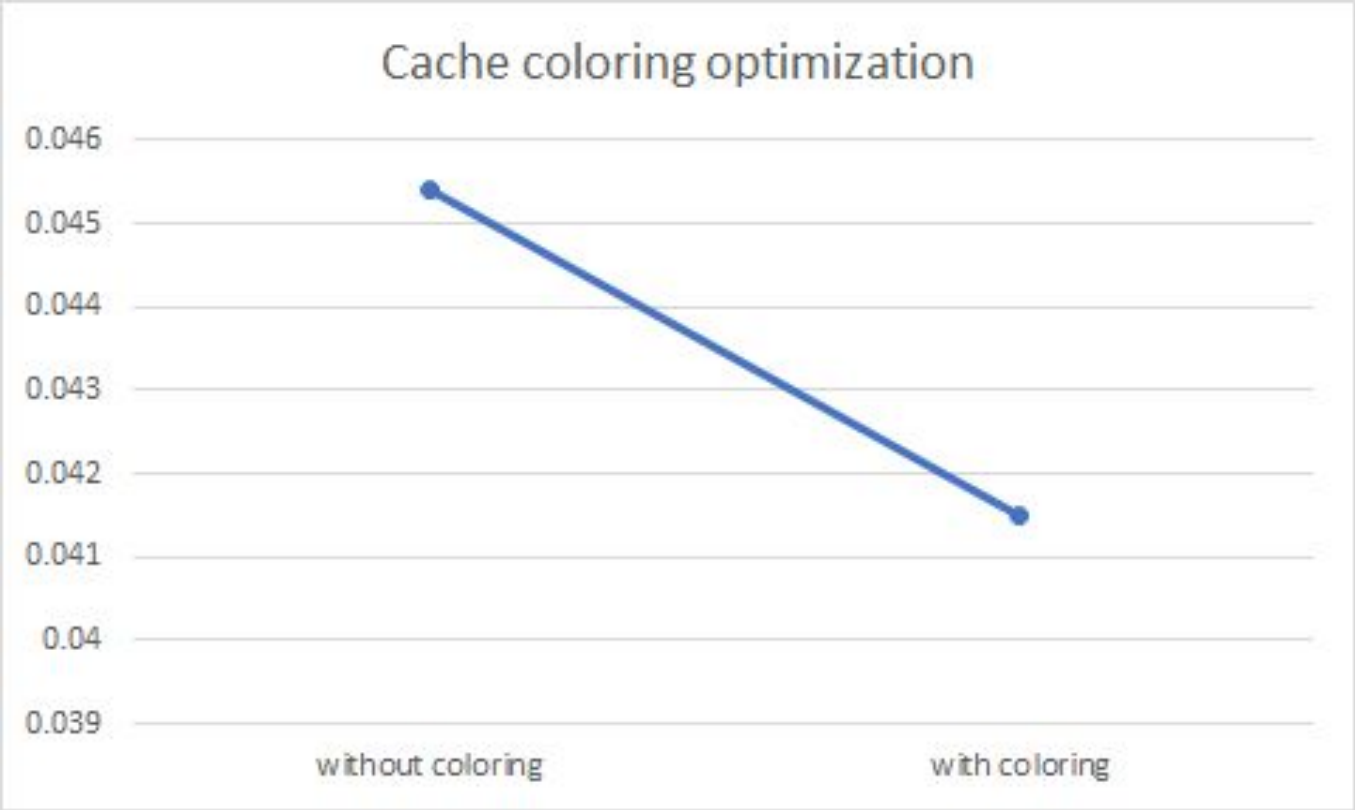
Appendix C: Cache Coloring Code

```
#include<stdio.h>
void A(){
	B();B();B();B();B();B();B();
	E();E();
}
void B(){
	C();C();C();C();C();C();
}
void C(){
	D();D();D();D();D();
	int a=16;int b=16;int c=32;int d=64;
	 c = a+b;
	 c = c-b;
	 a = c*b;
	 b = a/c;
	 c = a*b;
	 d = d*c;
	 d = d-a;
	 a = d*b;
	 c = a*d;
}
void D(){
	printf("D\n");
	int a=16;int b=16;int c=32;int d=64;
	 c = a+b;
	 c = c-b;
	 a = c*b;
	 b = a/c;
	 c = a*b;
	 d = d*c;
	 d = d-a;
	 a = d*b;
	 c = a*d;
	 c = a+b;
}
void E(){
	C();C();C();C();C();C();C();C();
	F();
	int a=16;int b=16;int c=32;int d=64;
	 c = a+b;
	 c = c-b;
	 a = c*b;
b = a/c;
	 c = a*b;
	 d = d*c;
	 d = d-a;
	 a = d*b;
	 c = a*d;
	 c = a+b;
	 c = c-b;
}

void F(){
	G();
}

void G(){
	printf("G\n");
	int a=16;int b=16;int c=32;int d=64;
	 c = a+b;
	 c = c-b;
	 a = c*b;
	 b = a/c;
	 c = a*b;
	 d = d*c;
	 d = d-a;
	 a = d*b;
	c = a*d;
	 c = a+b;
	 c = c-b;
	 a = c*b;

}

int main(){
	A();
	return 0;
}
```

Appendix D: Cache Coloring Results

	Example Code	
	Without Cache line Coloring	With Cache line coloring
Miss rate	0.0454	0.0415

Table: Results of cache coloring



Miss Rate v Cache coloring

Appendix E: Final Project Proposal

Research Problem

Analysis and implementation of different cache optimization techniques.

Background

The gap in performance between processor speed and memory continues to widen; new architectures are released every year with increased capabilities but memory systems remain stalemate, unable to cope with new demands. In this regard, it has become increasingly important to exploit cache memory efficiently.

Project Description

This project studies the performance improvements between four unique optimization techniques and three different simple cache design configurations. Each configuration and its optimization's performance on a standard benchmark are compared between implementations. This comparison provides an indication of which technique is the most effective.

Literature Review

Cache Optimization Techniques

The first article, "Cache Memory: An Analysis on Optimization Techniques," discusses five different cache optimization categories - replacement algorithms, cache algorithms, design based optimizations, compiler and prediction based optimization, and web based optimization [1]. The articles continues to layout a comparison of 16 specific optimization methods, comparing miss rate, miss penalty, hit time, hit rate, power consummation, access time, cost, complexity, and cycle time [1]. This article provides the team with guidance on important factors to measure during benchmark testing, such that an informed comparison can be created between the optimizations. Additionally, the team is implementing three of the optimizations discussed in the article - changing cache size, associativity and block size.

Cache Line Coloring Optimization

This paper discusses a link time procedure mapping algorithm which works by creating an improved program layout by color mapping considering the procedure size, cache size, cache line size and call graph. This algorithm reduces the miss rate by 40% from the original mapping. The algorithm intelligently places procedures in the cache layout by preserving color dependencies with a procedure's parents and children in the call graph, resulting in fewer instruction cache conflicts [2]. This paper implements the cache colouring algorithm and uses the ATOM tool for simulations. The paper uses programs like espresso, Perl, li, bison and gzip. The team will carry out a similar study for different workloads on a x86 machine using PIN tool by intel for generating trace.

Methodology

This project consists of three main tasks: architecture design feature implementation, cache design based optimization, and program to memory layout optimization. The team will use C++ for all architecture and optimization designs; benchmarks are in C. The project will compare the various architecture optimizations with three different benchmarks (chosen from either CacheBench, STREAM[3], Linpack, gcc, Whetstone, or other SPEC benchmarks). The team will collect 3 result parameters for each optimization: miss rate (MR), hit time (HT), and hit rate (HR) [1]. Each optimization will be implemented individually and results compared with configuration base case.

First step, the team will implement a simple cache. The simple cache, or base case, will consist of the following architecture specifications: 64 KB cache size (32 KB instruction cache and 32 KB data cache), 32-byte block size with 32-bit addresses (x86 represented). The base case with utilize a first-in-first-out (FIFO) replacement algorithm and consist of three configurations (direct mapped, 2-way set associative and 4-way set associative).

Second step, the team will implement 3 different cache design based optimization techniques on each simple cache configuration. The different optimization techniques include improving the replacement algorithm (least recently used LRU and least frequently used LFU), increasing cache size, and increasing block size. The team expects a higher level of associativity in the cache configurations to reduce capacity and conflict misses, an increase in cache size to reduce miss rates but increase access time, and an increase in block size to decrease compulsory misses while increasing conflict misses [1]. The team also expects LRU to decrease conflict misses by using a more advanced algorithm to remove memory in cache.

Third step, the team will implement cache coloring as a program to memory cache optimization. We will simulate the three-base case configurations in C++. All benchmarks will be in compiled from C with GCC. We will create an instruction trace of each benchmark using SimpleScalar and/or Pin. An example of the results table the team will create for each cache configuration is provided below.

Table 1: Expected result table.

Direct-Mapped Cache			
	Total Demand Fetches	Miss Rate	Hit Rate
Base Case (no optimizations)			
LFU Replacement Algorithms			
LRU Replacement Algorithm			
Larger Cache (128 KB cache size)			
Larger Block Size (128-byte block size)			

Cache Coloring			

Expectations

The expected relation between gathered results and associated grade is shown below in the table. The team will work together to complete every task but each member is assigned a primary and secondary role of responsibility for one aspect of the project: architecture feature design, design based optimizations, and program-memory layout optimization.

The primary method for the team's communication is in-person, secondary through Slack messaging, and tertiary through email. The team's version control system (VCS) will be Git, hosted on GitHub.com. The team's research and analysis notation tool will be LaTeX, hosted on Overleaf.com. In general, the team will follow a waterfall type methodology for completing the project - requirements, design, implementation, and verification.

Individual members' tasks are provided below in the responsibility matrix. The project is split into three parts. The first task includes implementing a simple cache for each configuration (direct-mapped, two-way, and four-way set associative). The second task includes creating and implementing three optimization features (increasing cache size, increasing block size and improving the replacement algorithm). Additionally, this task is responsible for creating a program to gather all necessary data from benchmark tests and running all benchmark experiments. The third task includes implementing cache coloring.

Table 2: Results with expected associated grade.

A	Implemented three simple cache configurations, implemented four optimization features, and tested three benchmarks.
A-	Implemented three simple cache configurations, implemented three optimization features, and tested three benchmarks.
B+	Implemented two simple cache configurations, implemented three optimization features, and tested two benchmarks.
B	Implemented two simple cache configurations, implemented two optimization features, and tested two benchmarks.
B-	Implemented two simple cache configurations, implemented one optimization feature, and tested two benchmarks.
C+	Implemented one simple cache configuration, implemented one optimization feature, and tested one benchmark.
C	Implemented one simple cache configuration, and tested one benchmark.

Table 3: Responsibility Matrix

	Primary Responsibility	Secondary Responsibility
Architecture Design Feature (simple cache implementation for direct-mapped, 2-way and 4-way set associative)	Anna DeVries	Gautham Ranganathan
Design Based Optimizations, Data Gathering Script and Benchmark Testing (increase cache size, increased block size, improved replacement algorithm)	Gautham Ranganathan	Shrikanth Showri Rajan
Program-Memory Layout Optimization (cache coloring)	Shrikanth Showri Rajan	Anna DeVries

References

[1] M. W. Ahmed and M. A. Shah, "Cache Memory: An analysis on Optimization Techniques," *International Journal of Computer and Information Technology*, vol. 4, no. 2, pp. 414 - 418, March 2015.

[2] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient Procedure Mapping Using Cache Line Coloring," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[3] https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/memory/membench_bm readme.html#CB%20Required%20Problems.