Performance Analysis of Cache for SPEC CPU2000 Benchmarks

Group member: Yuwei Huang, Shuo Fan, Peiyao Yang

1 Methodology

All functional data are collected from the Simplescalar toolset. We modified the simulator to simulate multiple caches, and report statistics every 100 million instructions. L1 instruction and L1 data cache ranging from 1KB to 256KB with 64B block size and set associativity of 1, 2, 4, 8 and full.

We modified the Simplescalar to distinguish between prefetch and load/store instructions and statistic the number of load/store instructions and prefetch instructions. We run benchmarks with and without prefectch operation and see the performance of L1 data cache.

We test four spec2000 benchmarks: the gzip, the swim, the ammp, and the applu, the gzip is an integer benchmark and the swim, ammp and the applu is the float benchmark.

2 Background

SimpleScalar is a set of tools that model a virtual computer system with CPU, Cache and Memory Hierarchy.

SimpleScalar can simulate Alpha and PISA,ARM, and x86 instruction sets

SimpleScalar tools can be built on either UNIX or Windows NTbased Operating Systems

sim-cache: This simulator can emulate a system with multiple levels of instruction and data caches, each of which can be configured for different sizes and organizations.

SPEC 2000 is one of the benchmark suites developed by the Standard Performance Evaluation Corporation (SPEC)

3 Prefetch

1 Introduction of prefetch

Prefetch is a cache optimization way to reduce the miss ratio and improve the performance of the cache. It is meant that the processor fetches a data block may be used into the cache before it is required. By this way, the processor will find the prefetched block in the cache when it is needed and avoid a cache miss, so the miss ratio can be reduced.

There are two main kinds of prefetch: the hardware prefetch and the compiler prefetch. The hardware prefetch is implemented by the hardware. Processor will fetch two blocks of data from memory when a cache miss occurs, one is the required block and the other is the prefetched block. The compiler prefetch is controlled by compiler. When the program is compiled into machine code, the compiler insert prefetch instructions to request data before the processor need it.

There are many algorithms for hardware prefetching. A simple way is the next-line prefetch, this way assumes that when a data block is required, the block next to it may be also needed in shortly future. So in this algorithm, the processor will fetch the required block and prefetch the block next to the required block. Another prefetch algorithm is the stride prefetching, this algorithm build a reference predictor table to record the previous addresses of memory instructions to determine which block to prefetch based on the historical data. This algorithm is more complex relatively, it can reduce the possibility of fetching useless data blocks and make the prefetch operation more efficiency. There are many other prefetch algorithms, but the object of this project is to compare the miss ratio with and without the prefetch, not the miss ratio of using different prefetch algorithms, we will only choose one prefetch algorithm.

2 Potential side-effect of prefetching

The prefetching operation has some side-effect that may increase the miss ratio and decrease the cache performance. We will see this phenomenon in our simulation results.

One main side-effect of prefetch is the cache pollution. If the prefetch operation fetched too many useless data that not needed by processor and replaced the more useful data in the cache, the miss ratio will increase after using prefetching operation. One way to reduce the effect of cache pollution is to use caches with large total size. When the total size of a cache is larger, the possibility of the more useful data is replaced by the prefetched data is lower, so a large total size of cache will reduce the impact of cache pollution. Another way to reduce the cache pollution is to use more efficiency prefetch algorithm and reduce the useless data prefetched by the processor.

3 Implementation of prefetch operation in Simplescalar V3.0

In this section, we will introduce the implementation of prefetch operation in Simplescalar V3.0 and correctly distinguish between prefetch instructions and load/store instructions when run SPEC2000 benchmarks with prefetch operations. Because the Simplescalar is a hardware simulator, so we will use the hardware prefetch. Among the hardware prefetch algorithms, we will choose the Next-Line prefetch algorithm.

Because the Simplescalar does not have prefetch operation itself, we need to modify the Simplescalar source code to implement prefetch operation. There are three source files of sim-cache simulator, the cache.h, the cache.c and the sim-cache.c. The cahce.h file include the declaration of variables and functions, the cache.c file include the manipulate functions of cache and the sim-cache.c file include the main function of the sim-cache simulator. We will mainly modify the cache.c file and add related variable and function declarations in the cache.h file. In the cache.c file, we add the prefetch function and the functiojn to set whether the prefetch operation will be executed. The code for prefetch operation is shown as figure 3.1.

```
/*Prefetcher Function: Nextline */
void next_line_prefetcher(struct cache_t *cp, md_addr_t addr)
{
   if (cache_probe(cp, addr + cp->bsize))      //check whether the next
block contaied in the cache
           return;//if contained, no need to prefetch

   //if next block is not in cache already, prefetch it
   cache_access(cp, Read, addr + cp->bsize, NULL, 1, (tick_t) 0, NULL,
NULL, 1);
   cp->prefetches++;
}
```

Figure 3.1: Prefetch operation code, if the data block need to be prefetched is already in cache, no need to prefetch, otherwise prefetch it.

In the prefetch function, there are two situations. The first one is the data block need to be prefetched is already in the cache, so we do not need to fetch again. If the block is not in the cache, prefetch it. We use the cache_probe function to check whether the block next to the required block is already in the cache. When found the block is not existed, we use the cache_access function to access memory and fetch the block. In order to distinguish between prefetch instructions and load/store instructions, everytime a prefetch instruction is executed, we add the counter of prefetch instructions. So we can know how many prefetch instructions there are in the load/store instructions.

We use another function to control whether the prefetch operations will be executed, this function is shown as figure 3.2. In this function, we use a flag parameter to indicate whether the prefetch operations will be executed. The parameter is the prefetch_type. When the prefetch_type equals to 0, it represents the perfetch operation will not be executed. When the prefetch_type equals to 1, it represents the perfetch operation will be executed. We use a branch structure to implement this function. If the prefetch_type parameter equals to 1, call the next-line prefetch function to execute prefetch operation, or do nothing.

```
void generate_prefetch(struct cache_t *cp, md_addr_t addr)
{
        switch(cp->prefetch_type)
        {
                case 0:          // prefetching is not
enabled;
                   break;         // do nothing
                case 1:           //Enable prefetch
                   next_line_prefetcher(cp, addr);  //Call prefetch
function, use nextline prefetch algorithm
                   break;
        }

}
```

Figure 3.2: Function to control whether to execute prefetch operations. Type 0: without prefetch, type 1: with prefetch

In order to know the information that we collected, we need to print out the information on the screen or in the record file. So we also add the code to print related information on the screen. These code is shown as figure 3.3. We use this function to print out the total number of prefetches executed.

```
  sprintf(buf, "%s.prefetches", name);
  stat_reg_counter(sdb, buf, "total number of prefetches", &cp-
>prefetches, 0, NULL);   //Print the number of prefetches
```

Figure 3.3: The code used for print out the total number of prefetches

4 Experimental Process of Cache Simulation

In this section, we will simulate the caches with different total size and set associative using the Sim-cache simulator of the Simplescalar V3.0 toolset and run the SPEC2000 benchmarks with and without prefetch operations to see whether there is an impact on the miss ratio.

Firstly, before we run the SPEC2000 benchmarks, we will use a configuration file to set the parameter of the cache. The main part of the configuration file related to the cache parameters is shown as figure 4.1. These parameters can set the level 1 and level 2 instruction cache, data cache or unified cache. There are five parameters for each cache, they are number of sets, block size, set associative, replacement strategy and whether to execute prefetch operations.

We will simulate two caches at once, the level 1 data cache and the level 1 instruction cache. The number of sets, block size and the set associative will determine the total size of a cache together. We will keep the block size equals to 64 bytes and change the number of sets and set associative to get caches with different total size. For the replacement strategy, we will use the LRU (Least recently used) strategy for all caches. For the prefetch operation, we will use prefetch operations for data cache to see whether there is an impact on the miss ratio. For the instruction cache, the prefetch operation will not be used.

```
# l1 data cache config, i.e., {<config>|none}
-cache:dl1 dl1:128:64:4:l:1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2 none

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1 il1:128:64:4:l:0

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2 none
```

Figure 4.1: Configuration file of cache, the first parameter is the number of cache, the second is the block size (Always set to be 64 Bytes), the third is the set associative

(Change from 1,2,4,8 to full), the forth is the replacement strategy (Always set to be LRU) and the fifth is whether to execute prefetch operation (0 represent not execute prefetch and 1 represent execute prefetch).

After configuring the cache, we will run the SPEC2000 benchmarks to see the different miss ratios with different parameters. We chose for benchmarks to run, they are the Integer benchmark Gzip and the float benchmark Swim, Applu and Ammp.

5 Simulation Result

1. Distinguish prefetch instructions and Load/Store Instructions

Firstly, we simulated a level 1 data cache to run four benchmarks with prefetch operations, separately count the total number of instructions, loads/stores and prefetches to distinguish between load/store instructions and prefetches. Then we calculate the load/store rate and the prefetch rate of different benchmarks. The parameter of the cache is: Total size 32KB, 64B block size and 2-way set associative. We report the statistics every 100 million instructions. The data is recorded in the table 5.1.

Benchmark	Instructions	Loads/Stores	Load/Store Rate	Prefetches	Prefetch Rate
swim	100000000	24350876	0.243509	139814	0.001398
gzip	100000000	32664925	0.326649	1071905	0.010719
ammp	100000000	30715803	0.307158	3479400	0.034794
applu	100000000	32701906	0.327019	660152	0.006601
Average	100000000	30108377.5	0.301084	1337817.75	0.0133782

Table 5.1 Number of prefetches and Load/Store instructions of level 1 data cache with prefetch operation

According to the data of table 1, we can find that the average rate of load/store instructions in the four benchmarks don’t change a lot, from the lowest 0.2435 to the highest 0.3270, the average rate is 0.30108, about 30 per cent. But the number of prefetches is quite different between four benchmarks, from the lowest 0.001 to the highest 0.03, the difference is about 30 times. The average prefetch rate is 0.013378, about 1 per cent.

2. Miss ratio result using different benchmark

In the following diagram, y axis stands for miss rate, x axis stands for cache size. Different curve differs in set associative.

1 Result of Integer benchmark Gzip

Table 5.2 GZIP instruction cache without prefetch

IL1					
	1	2	4	8	full
1KB	0.04778048	0.02357519	0.03320474	0.02670338	0.03511525
2KB	0.01624342	0.01651416	0.00003071	0.00003121	0.0000334
4KB	0.00764555	0.00002214	0.00002234	0.00002223	0.00002168
8KB	0.00763643	0.00001254	0.00001249	0.0000127	0.00001396
16KB	0.00763386	0.00000638	0.00000512	0.00000504	0.00000495
32KB	0.00763256	0.00000508	0.00000497	0.00000495	0.00000495
64KB	0.00000512	0.00000506	0.00000495	0.00000495	0.00000495
128KB	0.00000495	0.00000495	0.00000495	0.00000495	0.00000495
256KB	0.00000495	0.00000495	0.00000495	0.00000495	0.00000495

Figure 5.1 Miss ratio change in GZIP instruction cache without prefetch

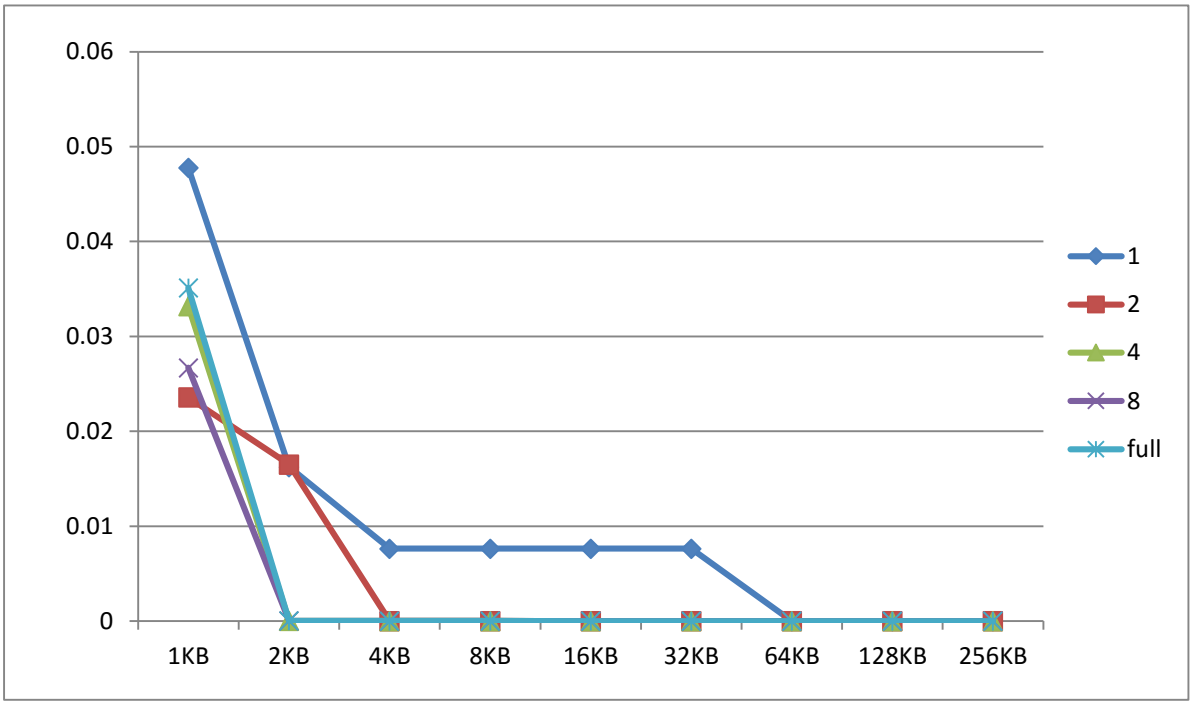


Table 5.3 GZIP data cache without prefetch

DL1					
	1	2	4	8	full
1KB	0.2008	0.1698	0.1322	0.1188	0.1229
2KB	0.1098	0.0853	0.0689	0.056	0.0487
4KB	0.0751	0.053	0.0441	0.0424	0.0413
8KB	0.0546	0.0412	0.0389	0.0384	0.0381
16KB	0.0436	0.036	0.0353	0.0352	0.035
32KB	0.0348	0.0306	0.0304	0.0304	0.0303
64KB	0.0253	0.0224	0.0222	0.0222	0.022
128KB	0.0124	0.0106	0.0096	0.0095	0.0093
256KB	0.0032	0.0023	0.0018	0.0018	0.0019

Figure 5.2 Miss ratio change in GZIP data cache without prefetch

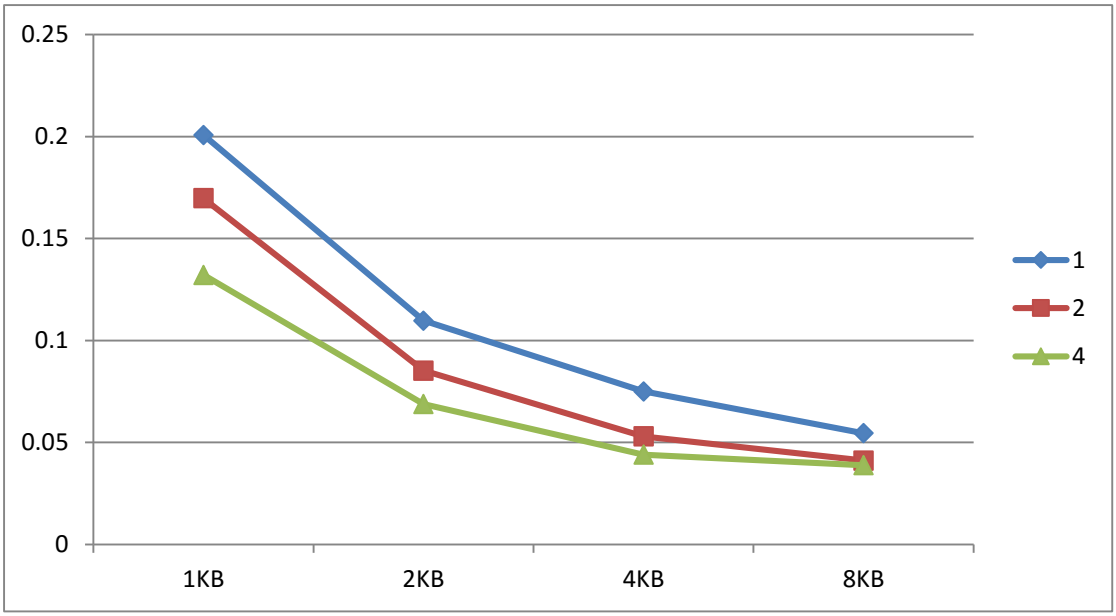
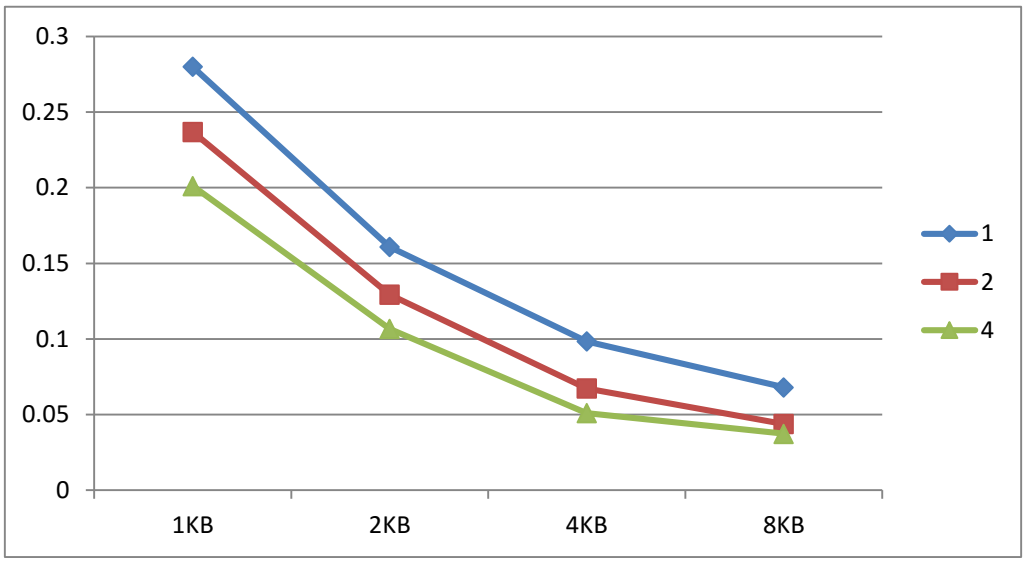


Table 5.4 GZIP data cache with prefetch

DL1 Prefetch					
	1	2	4	8	full
1KB	0.2801	0.237	0.2012	0.1919	0.1954
2KB	0.1609	0.1294	0.1067	0.0944	0.0823
4KB	0.0985	0.0673	0.0511	0.0467	0.0405
8KB	0.0681	0.0439	0.0374	0.0356	0.0352
16KB	0.0476	0.0338	0.0319	0.0317	0.0315
32KB	0.0346	0.0273	0.0268	0.0267	0.0266
64KB	0.0232	0.019	0.0188	0.0188	0.0186
128KB	0.01	0.0073	0.0065	0.00649	0.0065
256KB	0.0019	0.0005	0.00029	0.00026	0.00029

Figure 5.3 Miss ratio change in GZIP data cache with prefetch



For the miss ratio with and without the prefetch, we can find prefetch decrease the miss ratio in bigger size cache. When the cache is bigger, the efficiency is better. But the miss ratio in smaller size cache increase, this may be resulted from cache pollution.

2 Result of float benchmark swim

Table 5.5 Swim instruction cache without prefetch

IL1					
	1	2	4	8	full
1KB	0.00006432	0.00004885	0.00004938	0.00004711	0.00004594
2KB	0.00004965	0.00004414	0.00004356	0.0000415	0.00004157
4KB	0.00004009	0.00003984	0.00003956	0.0000381	0.00003887
8KB	0.00003263	0.00003176	0.00003227	0.00003217	0.00003166
16KB	0.00002368	0.00001933	0.00001717	0.0000153	0.00001416
32KB	0.00001758	0.00001415	0.00001169	0.00001076	0.0000103
64KB	0.00001342	0.0000109	0.00000964	0.00000945	0.00000927
128KB	0.00001081	0.00000988	0.00000931	0.00000927	0.00000927
256KB	0.0000101	0.00000966	0.00000927	0.00000927	0.00000927

Figure 5.4 Miss ratio change in swim instruction cache without prefetch

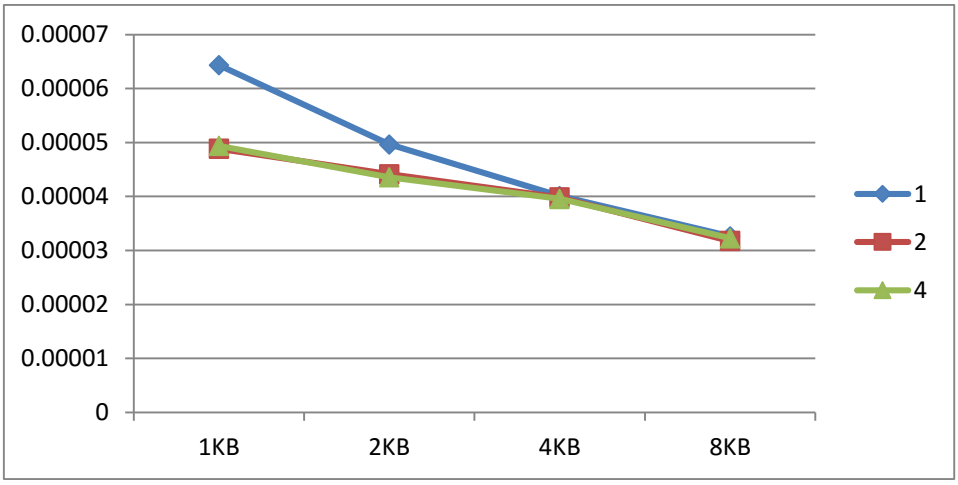


Table 5.6 Swim data cache without pretech

DL1					
	1	2	4	8	full
1KB	0.203	0.2146	0.1911	0.2059	0.0212
2KB	0.1157	0.1128	0.0575	0.015	0.0166
4KB	0.0944	0.0143	0.0069	0.0063	0.0057
8KB	0.0331	0.0124	0.0057	0.0056	0.0056
16KB	0.0175	0.0059	0.0056	0.0056	0.0056
32KB	0.0167	0.0057	0.0056	0.0056	0.0056
64KB	0.0162	0.0056	0.0056	0.0056	0.0056
128KB	0.0058	0.0056	0.0056	0.0056	0.0056
256KB	0.0057	0.0056	0.0056	0.0056	0.0056

Figure 5.5 Miss ratio change in swim data cache without prefetch

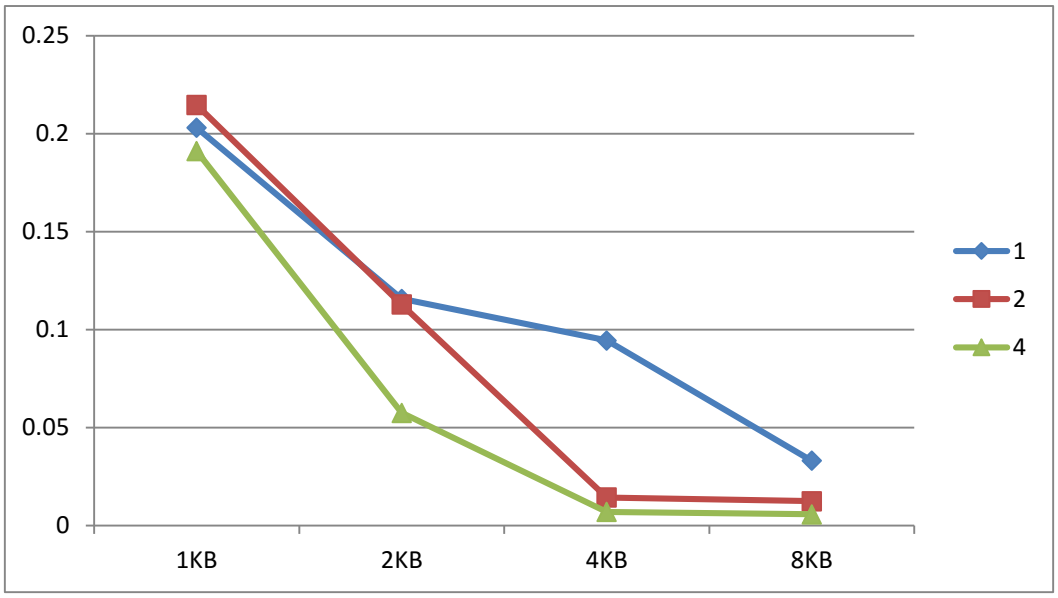
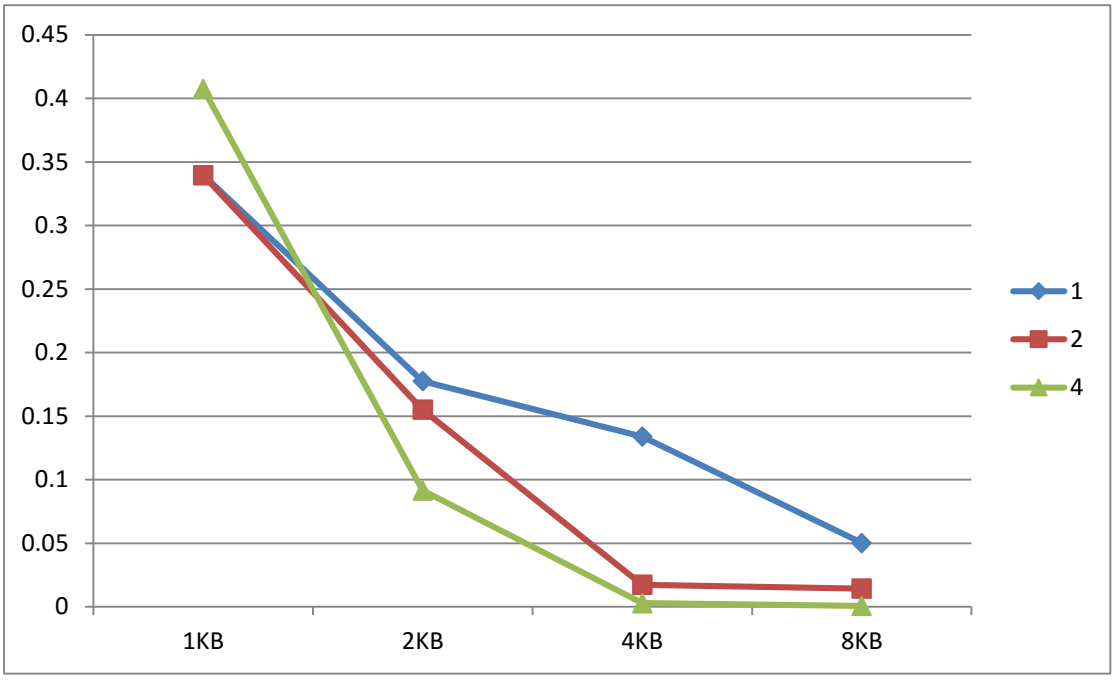


Table 5.7 Swim data cache with prefetch

DL1 Prefetch					
	1	2	4	8	full
1KB	0.3397	0.3395	0.4075	0.4141	0.4046
2KB	0.1775	0.1551	0.0915	0.0094	0.0082
4KB	0.1339	0.0173	0.0028	0.0016	0.0007
8KB	0.0502	0.0143	0.0007	0.0007	0.0007
16KB	0.0235	0.0011	0.0007	0.0007	0.0007
32KB	0.0225	0.0008	0.0007	0.0007	0.0007
64KB	0.0219	0.0007	0.0006	0.0006	0.0006
128KB	0.0009	0.0007	0.0006	0.0006	0.0006
256KB	0.0008	0.0006	0.0006	0.0006	0.0006

Figure 5.6 Miss ratio change in swim data cache with prefetch



The result of swim is the similar to gzip.

3 Result of float benchmark Applu

Table 5.8 Applu instruction cache without prefetch

IL1					
	1	2	4	8	full
1KB	0.01339245	0.04920297	0.04788744	0.04806435	0.0492
2KB	0.04624102	0.0471646	0.03572161	0.04271863	0.0427
4KB	0.01339065	0.01339166	0.01339245	0.01339347	0.0134
8KB	0.0133441	0.01334272	0.0133441	0.01334144	0.0133
16KB	0.01332484	0.01285858	0.01332484	0.01332286	0.0133
32KB	0.00001182	0.00001437	0.00001224	0.00001187	0.00001182
64KB	0.00001182	0.00001194	0.00001182	0.00001182	0.00001182
128KB	0.00001267	0.00000208	0.00001182	0.00001182	0.00001182
256KB	0.00001222	0.00001182	0.00001182	0.00001182	0.00001182

Figure 5.7 Miss ratio change in Applu instruction cache without prefetch

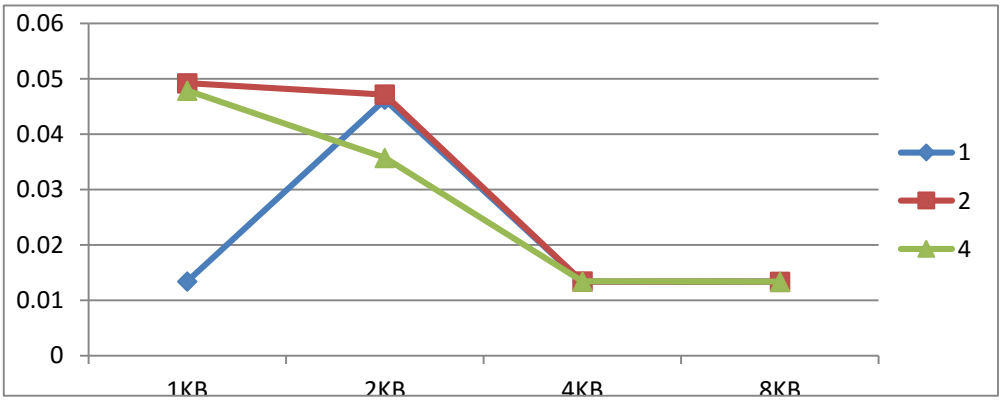


Table 5.9 Applu data cache without prefetch

DL1					
	1	2	4	8	full
1KB	0.24762843	0.18413517	0.16364961	0.1568547	0.0492
2KB	0.2136	0.18413517	0.09234822	0.09021	0.0905
4KB	0.10236198	0.07934399	0.0762	0.078589	0.0113
8KB	0.03028741	0.03828342	0.03445502	0.034402	0.0342
16KB	0.02110912	0.02378743	0.0211	0.0203425	0.0201
32KB	0.02017428	0.02085033	0.0201482	0.02013152	0.0201
64KB	0.02013086	0.0203575	0.02013086	0.0201307	0.0201
128KB	0.02214557	0.02014871	0.0201304	0.020130336	0.0005
256KB	0.01843899	0.02005759	0.020125749	0.02012844	0.0201

Figure 5.8 Miss ratio change in Applu data cache without prefetch

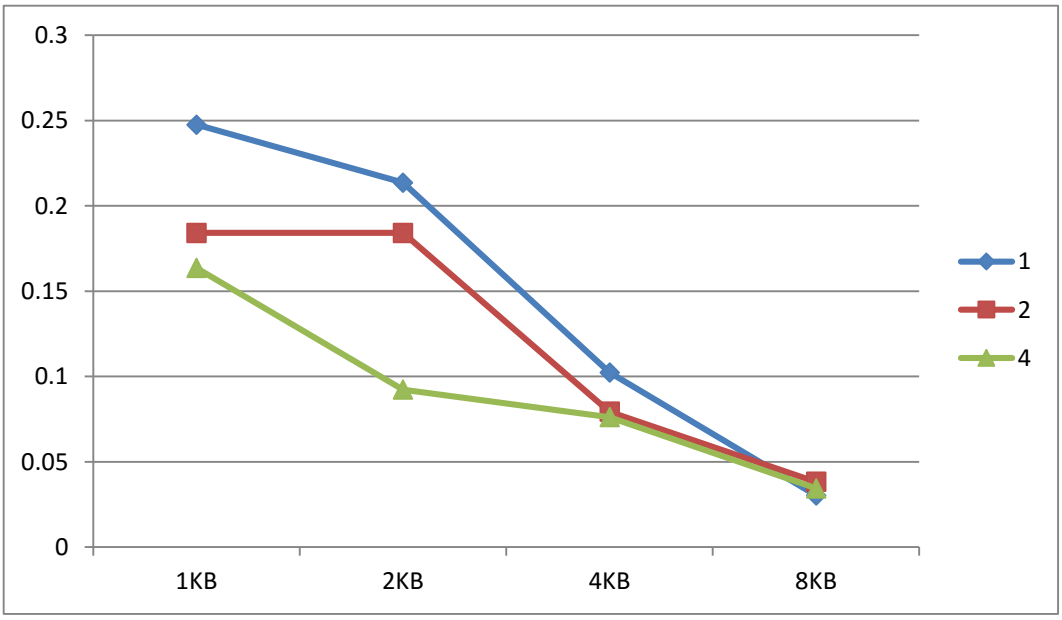
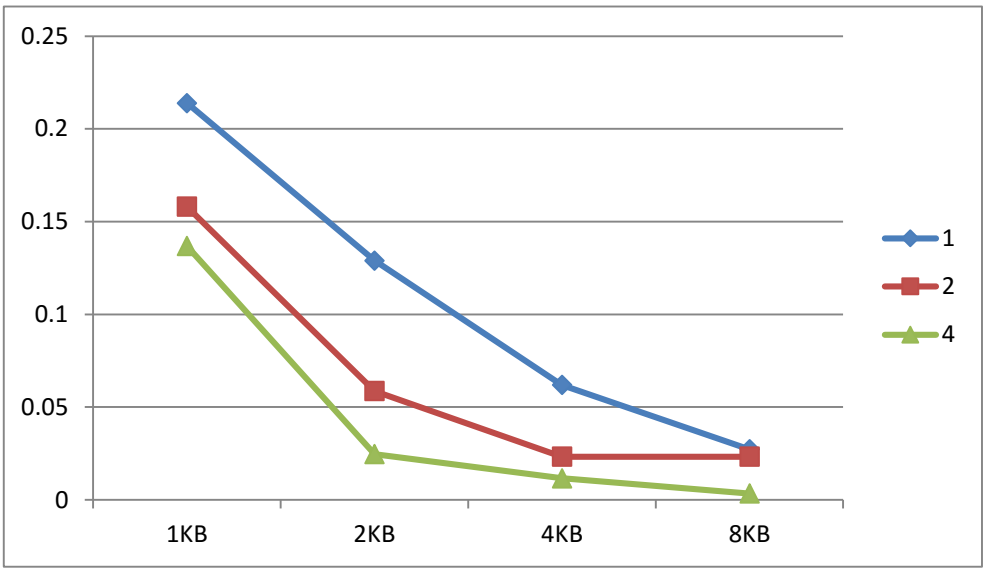


Table 5.10 Applu data cache with prefetch

DL1 Prefetch					
	1	2	4	8	full
1KB	0.213927	0.1581	0.1368	0.0904	0.0984
2KB	0.128979	0.0587	0.024665	0.019093	0.0191
4KB	0.06197	0.0233	0.011676	0.011222	0.0041
8KB	0.027255	0.023261	0.003449	0.003483	0.0041
16KB	0.0144	0.00167	0.000795	0.000588	0.0005
32KB	0.009	0.007586	0.000509	0.00049	0.0005
64KB	0.0028	0.000583	0.000489	0.000489	0.000489
128KB	0.0017	0.000498	0.000489	0.000489	0.0005
256KB	0.0009	0.000489	0.000487	0.000488	0.0005

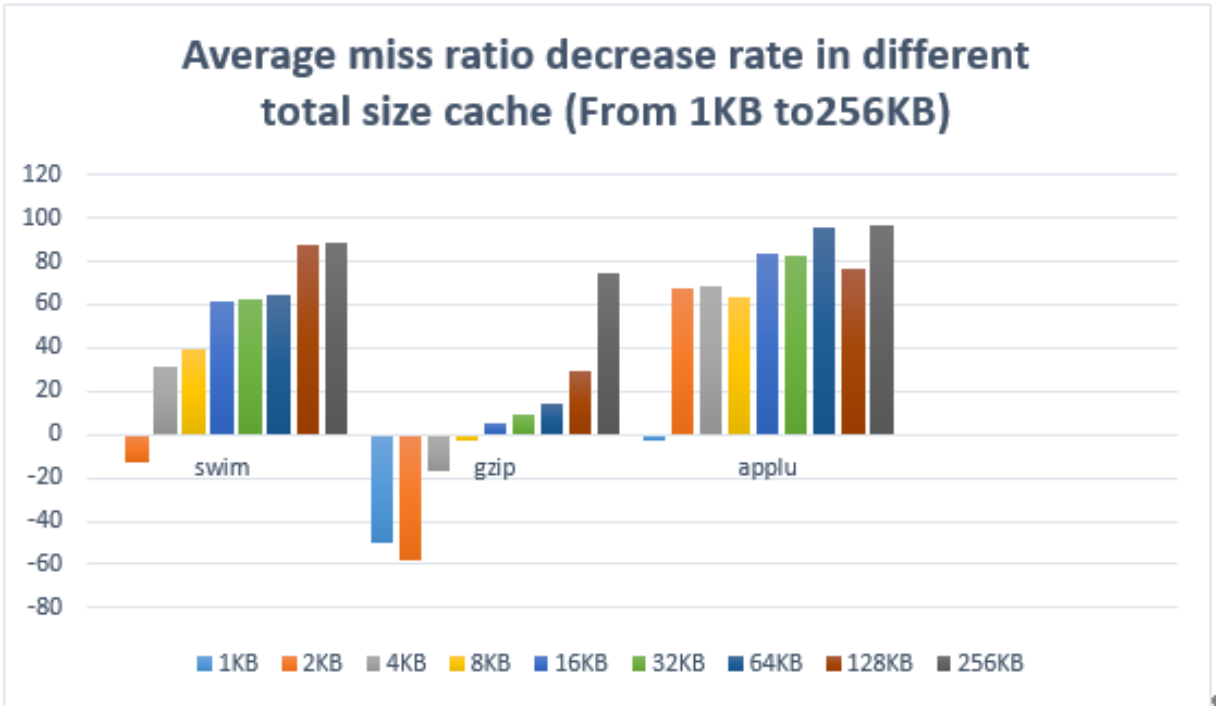
Figure 5.9 Miss ratio change in Applu data cache with prefetch



3 The impact of prefetch operation in different total cache size

The average miss ratio decrease rate in different total cache size is shown as figure 5.10.

Figure 5.10



From this figure, we can find in small total size cache, the cache pollution is quite serious and may decrease the cache performance. In larger total size cache, the efficiency of prefetch operation is quite good. Generally, larger the total cache size, better the efficiency of prefetch operation.

6 Conclusion

1 When the set associative increases, the miss ratio of cache generally decreases. This is because when the set associative increase, it will reduce the conflict misses and improve the cache performance.
2 When the cache size increases, the miss ratio generally decreases. This is because when the cache size becomes larger, it will reduce the capacity misses.
3 The efficiency of prefetch operation depends on the prefetch algorithm, if the prefetch algorithm designed well and can avoid prefetching too much useless data, it will improve the miss ratio significantly, or it may decrease the performance.
4 When using the next line prefetching algorithm, it has quite good efficiency in larger size cache and may decrease the performance of small size cache.

References

1.www.simplescalar.com
2.http://www.ecs.umass.edu/ece/koren/architecture/Simplescalar/SimpleScalar_introduction.htm