**Three optimizations to improve cache performances based on SimpleScalar simulator**

**Abstract:**

During the last two decades, the CPU’s performance has been developed quickly, however, the speed it increase is much faster than that of memory. And the gap between the CPU and the memory became one of the bottleneck in computer architecture. Cache has been used to decrease the gap by providing the processor with data more quickly. In the other hand, many factors exists to influence the cache performance. There are multiple caching optimization strategies to improve the cache performance. The goal of this project is evaluating cache performance with various caching strategies and show its benefits using special benchmarks.

**Introduction:**

The performance of the CPU has been increasing exponentially during the last few decades; while on the other hand, the memory speed grows much slower than the development of CPU. People has been focusing on improving the performance of the cache, which is a key factor of decreasing the gap, aiming to narrow the gap between the CPU and the memory. As a result, we choose to evaluate the performance of the cache using several cache strategies and finally come up with our own caching managing optimizations.

**SimpleScalar tool set**

To accomplish our goal, we choose to use the SimpleScalar tool set to evaluate the performance. The SimpleScalar simulator is a superscalar RISC architecture simulator, offering various simulation programs from simple functional simulation to out-of-order launch of superscalar[1]. This tool sets have implemented the execution and interpretation functionally and providing a GCC centered compiler and some other parts, capable of generating object code based on the SimpleScalar architecture and running on SimpleScalar simulator.

The tool mainly used in our project is the sim-cache tool. We choose this tool because it can be used to evaluate the different cache designs.

**Benchmarks**

As for the benchmark, we use the LINPACK and the Spec2006. The LINPACK benchmark is short for the linear system package and it has been widely used for solving various mathematical and engineering problems like solving a dense system of linear equations and testing the floating-point calculations of the high-performance computer[2]. To be specific, we often choose the maximum number of floating-point calculations that the computer can complete in a second as the important indicator.

For the Spec2006, we run the bzip2 and the bwaves to test the performance. The bzip2 is to test the data compression ability of the computer and it consists of 6 parts[3]: Two small JPEG images; a program binary; several source files code in a tar; An HTML file and a "combined" file, which is representative of an archive that contains both highly compressible and not very compressible files. Then for the input, each set is compressed and decompressed at three different blocking factors and every result of the procedure will be compared to the original data after every depression step. Then the output sizes for every compression and decompression are shown to facilitate validation and the decomposition results will be compared with the input data.

The other benchmark, bwaves, numerically simulates blast waves in three dimensional transonic transient laminar viscous flow[3]. For the input, the file describes the grid size, flow parameters, initial boundary condition and number of time steps. The three data sets, test, train and ref, differ only in grid size and number of time steps. Since the transient nature of the flow and iterative solver makes bwaves a difficult problem to accomplish, the Spec2006 solves the problem by comparing 3 different outputs: The L2 norm of dq(l.i.j.k) vector after final time step; The residual for convergence after each time step and The cumulative sum of iterations for convergence for every time step.

**Optimization Methods:**

L3 cache

The traditional focus to improve cache performance is reducing the miss rate by using a larger cache, however, a larger cache has lower miss rate and higher latency []. Decreasing miss penalty becomes a new methods to improve the performance of caches for the tradeoff between miss rate and latency. Multiple levels of cache with small fast caches backed up by larger, slower caches [4] is a good way to address this tradeoff. For example, the IBM POWER4 had off chip L3 of 32MB per professor.

We extended SimpleScalar to support an iL3 cache and a dL3 cache in order to understand the benefits using L3 cache. To achieve this, we re-implemented sim-cache.c file, which include up to 2 levels of caches (instruction and data, or unified) and up to one level of TLB (instruction and data). We add L3 cache based on cache.c file between L2 cache and Memory. In this way, when a miss happened in L2 cache, the L2 cache checks L3 cache first instead of retrieving data from memory.

In the experiment, we will change different parameters of L3 cache and run some benchmarks to discuss how the different parameters influence the behavior of L3 cache.

Victim cache

The victim cache was introduced by Norman Jouppi from 1990 [4]. It is a fully associated cache whose size typically 4~16 cache lines between a direct-mapped L1 cache and L2 cache, and it is used to reduce the conflict miss. With victim cache, the evicted block can have a second chance and thus a victim cache decrease the miss caching.

On a miss in L1 cache, we check the victim cache before retrieving the next level cache directly. If a hit happened in the victim cache, the matching line is replaced to the L1 cache, at the same time, the data contends in conflict L1 cache line is thrown out to the victim cache. If there is a miss in victim cache, the data is accessed to the next level of cache, the L2 cache, and moving the current block to the victim cache[5]. Figure 1 is the position of a victim cache.

Processor

L1 cache

Victim

Cache

L2 cache

Memory

L1 cache system

Address

**Figure 1: The position of a victim cache**

We extended the SimpleScalar to support the victim cache between L1 and L2 cache by re-implementing the sim-cache.c file. The algorithm is as follows:

/\* il1 / dl1 cache block miss handler function \*/

If (victim cache) {

Find a line to replace in L1 cache

If (hit in victim cache) {

Swap the ‘hit’ to L1 cache

Move the conflicted line to victim cache

}

Else {

Add replaced block to victim cache

Continue

}

}

If (L2 cache) {

…

}

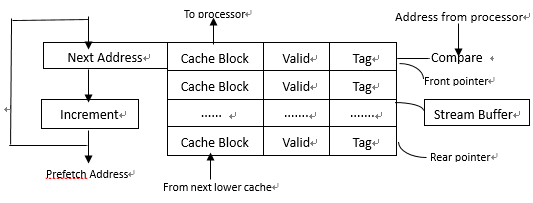
For the victim cache is fully associated, we assume the 16 associativity as fully associated. In the experiment part, we will discuss the effect of different parameters on victim cache performances.

Stream Buffer

Stream buffer, in a broad sense, is kind of cache whose main function is to avoid cache pollution caused by pre-reading[5]. When using this mechanism, the processor will put pre-read data sequence into the Stream buffer instead of into the cache. If the data used by the processor did not hit in the cache, the processor will first look in the stream buffer. It can eliminate the pollution of the read-ahead Cache by using this method, but also thereby increasing the complexity of the system design.

A stream buffer is composed of a plurality of entries, an entry can be stored in one or more cache block, also contains a number of status bits. Each entry of stream buffer consists of cache block, valid bit and with the corresponding address tag. Valid bit which indicates the current data in cache block is valid, and the address tag used for c address comparison. The stream buffer is similar to FIFO, started from the front pointer, the new data will fill the position of rear pointer[6].

When cache miss occurs, the micro-architecture will first start looking for data in the front of the stream buffer, if hit, the data would pre-read into the cache, so as not to cause cache pollution, at the same time the pre-reading data would be removed from the front of stream buffer cache. Then the micro-architecture obtain cache block from the next lower cache based on pre-fetch address, and fill in the tag information of entry corresponding rear pointer, when data returns, the processor will fill in the corresponding cache block , and valid bit is valid. If the data in the Stream buffer Miss, but the system has only one stream buffer, the stream buffer will be refreshed, and trying to establish a new pre-reading sequence.



**Figure 2: Stream Buffer**

Then we extend to simplescalar to support the stream buffer between the L1 and L2 cache by implementing the cache.c and sim-cache.c files, the algrithom is shown below:

In cache.c file, we will create a function associated with stream buffer:

Buffer () {

Compare the address tag and the valid bit

If (hit) {

the stream buffer hits will increase by 1, then remove the head and put it back to L1, add a new data to the tail, we consider the cache as FIFO }

if (miss) {

the stream buffer misses will increase by 1, then we will flush this cache and request the next address and fill the stream buffer with the new stream

}

In sim-cache.c file, we will use the function we create in the cache.c:

/\* il1/dl1 cache block miss handler function \*/

if (stream buffer){

We enter into the stream buffer to look for the address

if (hit in the stream buffer){

the instruction L1 cache miss decreases by 1

the instruction L1 cache hit increases by 1

}

}

**Experiment**

Configurations

* **L3 cache:**
* **Origin Configuration:**

Il1:64:32:1:l dl1:64:32:1:l

Il2:128:32:1:l dl2:128:32:1:l

* **Configuration1:**

Add split L3 caches with block size 32B, 64B and 128B

* **Configuration2:**

Add a unified L3 cache with block size 32B, 64B and 128B

* **Victim cache**
* **Configuration3:**

Add victim cache (4 entries) between il1 and il2 cache with block size 32B, 64B and 128B

* **Configuration4:**

Add victim cache (4 entries) between dl1 and dl2 cache with block size 32B, 64B and 128B

* **Configuration5:**

Add victim cache (8 entries) between il1 and il2 cache with block size 32B, 64B and 128B

* **Configuration6:**

Add victim cache (8 entries) between dl1 and dl2 cache with block size 32B, 64B and 128B

* **Configuration7:**

Add victim cache (16 entries) between il1 and il2 cache with block size 32B, 64B and 128B

* **Configuration8:**

Add victim cache (16 entries) between dl1 and dl2 cache with block size 32B, 64B and 128B

* **Stream buffer**
* **Configuration9:**

Add stream buffer between dl1 and dl2 with block size 32B, 64B, and 128B

* **Configuration10:**

Add stream buffer between il1 and il2 with block size 32B, 64B, and 128B

* **Victim cache & Stream Buffer**
* **Configuration11:**

Victim cache between dl1 and dl2, has 16 entries with block size 128B

Stream buffer between il1 and il2, with block size 128B

* **Configuration12:**

Victim cache between il1 and il2, has 16 entries with block size 128B

Stream buffer between dl1 and dl2, with block size 128B

**Result and Analysis**

* L3 cache:

Figure 3: L3 miss rate of bzip2

Figure 4: L3 miss rate of bwaves

Figure 5: L3 miss rate for Linpack

Due to the results we obtained in the experiment, the miss rate of L1 and L2 cache showed nothing changed, therefore we mainly study the performance of the L3 cache. From the three figures shown above, just in terms of the block size, with the increasing block size, the L3 miss rate of all benchmarks are decreasing exactly as what we expected. The reason why is that larger block size indeed takes advantage of spatial locality, leading to the reduction of the compulsory misses. Even though larger block size means that fewer blocks will be in cache which will increases capacity misses and conflict misses. Relatively, the reduction of compulsory misses is more than the increase of capacity misses and conflict misses.

Next we take the factor of split and unified into consideration, from the figures, the miss rate of unified L3 cache is less than the miss rate of the split L3 cache. In my view, for the split cache need to use the half cache only for instruction and the other half only for data, while combined cache does not have to limited by this. The unified cache does not divide the numbers of entries in cache that the split cache have to do. What’s more, if the current program needs more data references than instruction references, the cache will accommodate, vice versa. Thus, the unified cache performs better than split cache on account of miss rate.

* Victim Cache

As for bzip2 benchmark, we first set a victim cache with different block size between il1 cache and il2 cache and increase the number of victim cache entries to show the effect of different parameters of the victim cache. The result is showed in the Figure 6. Then, we change the position of victim cache to data cache and the result is showed in the Figure 7.

Figure 6: il1 miss rate of bzip2

Figure 7: dl1 miss rate of bzip2

In the Figure 6, we can see the il1 cache’s performance benefits obviously with the higher block size. The 4 entries victim cache with 32B block size also did a great job in decreasing the miss rate and the miss rate decrease up to 25% to 50% of the origin miss rate, however, the change is not apparently for those victim caches with 64B, 128B block size for there are no changes between 4 entries to 16 entries with 64B, 128B block size.

From Figure 7, we can see the il1 cache’s performance benefits obviously with the higher block size, however, there are nearly no changes between 4 entries to 16 entries with 64B, 128B block size. On the other hand, the 4 entries victim cache with 32B block size decrease the miss rate clearly and the miss rate decrease up to 67% of the origin miss rate.

Figure 8: il1 miss rate of bwaves

Figure 9: dl1 miss rate of bwaves

For bwaves benchmark, we repeat the above steps to discuss how the parameters influenced the performance of victim cache. The result is showed in Figure 8 and Figure 9.

From Figure 8, we can see the il1 cache’s performance benefits obviously with the higher block size, however, there are nearly no changes between 4 entries to 16 entries with 64B, 128B block size. On the other hand, the 4 entries victim cache with 32B block size decrease the miss rate clearly and the miss rate decrease 42% to 67% of the origin miss rate.

In the Figure 9, we can see the il1 cache’s performance benefits obviously with the higher block size. The 4 entries victim cache with 32B block size also did a great job in decreasing the miss rate and the miss rate decrease up to 69% of the origin miss rate, however, the change is not apparently for those victim caches with 64B, 128B block size.

Figure 10: il1 victim cache misses of Linpack

Figure 11: dl1 miss rate of Linpack

As for Linpack benchmark, we repeat the above steps and show the results in Figure 10 and Figure 11.

In the Figure 10, we use the number of misses of the victim cache instead of the miss rate because of the 0 miss rate we obtained. The number of misses reduce with the larger block size and the higher entries of the victim cache.

From Figure 11, we can see the il1 cache’s performance benefits obviously with the higher block size, however, there are nearly no changes between 4 entries to 16 entries with 64B, 128B block size. On the other hand, the 4 entries victim cache with 32B block size decrease the miss rate clearly and the miss rate decrease up to 4% of the origin miss rate.

* Stream buffer

Figure 12: il1 miss rate of bzip2 Figure 13: dl1 miss rate of bzip2

Figure 14: il1 miss rate of bwaves Figure15: dl1 miss rate of bwaves

Figure 16: il1 miss rate of Linpack Figure 17: dl1 miss rate of Linpack

To analysis the effect of the stream buffer, we set a stream buffer with different block size between L1 and L2 cache, both instruction cache and data cache. The results are showed in Figure 12 to Figure 17.

We can see a better performance with the increasing block size from the above figures. The benefit appears more obviously in instruction cache than the data cache. The stream buffer can decrease the miss rate up to 95% in the instruction cache compared 43% in the data cache.

* Victim cache & Stream Buffer

To measure the position of victim cache and stream buffer, we set the victim cache between the data caches and stream buffer between the instruction caches first and it is the configuration 11. The configuration 12 is that the stream buffer between the data caches and the victim cache between the instruction caches. The result is showed in the Table 1.

From Table 1, we can see the configuration 11 has better benefits though the instruction caches’ miss rate may increase a bit.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | conf11\_il1 | conf11\_dl1 | conf12\_il1 | conf12\_dl1 |
| bzip2 | 0.01 | 33.10 | 0.02 | 33.13 |
| bwaves | 1.57 | 1.68 | 1.23 | 7.67 |
| Linpack | 0.04 | 0.50 | 0 | 1.09 |

Table 1: the miss rate of the configuration 11 and configuration 12

**Conclusion**

In this project, we mainly study three cache optimization method: victim cache, stream buffer and L3 cache. From the results above, all of three cache optimization provide better performance than original settings when running on the SimpleScalar simulator.

The performance judgment in the project based on the value of the cache miss rate and there are two main factor to influence the miss rate: block size and the number of entries. The results of 12 configurations show the relationship between the factors and performance, with the increasing cache block size or the number of entries, the miss rate of cache will decrease correspondently. However, the miss rate would not decrease all the time. When the value comes to the threshold, the performance achieves its limitation. For instance, during the victim cache experiment, the miss rate remains unchanged when the block size reach to 128B.

Based on the limited time, our project is done here. There might be more future work can be done to appreciate the cache optimization. For instance, the stream buffer can be set to multi-way stream buffer, and victim cache could be added after L2 cache and the set associativity of L2 cache could be set to 4, 8 and fully associative.

**References:**

[1] SimpleScalar version 2.0 tutorial www.simplescalar.com

[2] http://en.wikipedia.org/wiki/LINPACK

[3] “SPECCPU2006 Benchmark Descriptions”, Edited by John L. Henning,

www.spec.org/cpu2006/publications/

[4] http://en.wikipedia.org/wiki/CPU\_cache

[5] Norman P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. SIGARCH Comput. Archit. News 18, 3a

[6] Xiaomei Ji, Dan Nicolaescu. 1940. Compiler-Directed Cache Assist Adaptivity. ISHPC 200, LNCS 1940, pp. 88-104.

[7] Alaa Alameldeen, Haitham Akkary. 2012. Cache and prefetching.