Architecture Tuning Study: the SimpleScalar Experience

Jianfeng Yang Yiqun Cao

December 5, 2005

Abstract

SimpleScalar is software toolset designed for modeling and simulation of processor performance. In this paper, we present our experience of using SimpleScalar to simulate several new architectures. We have implemented 3 architectural features, including a *level-3 cache*, *victim cache*, and a novel concept of *adaptive split cache*. Sample testing programs included in SimpleScalar as well as NetBench suite are used to test performance gain of these new features. We have also studied the effect of different configurations of these features. Discussions based on these observations will be given.

1 Introduction

Modeling and simulation of system performance is important in architecture design as well as software design and performance tuning. SimpleScalar tool set [1] is a system software infrastructure used to build modeling applications for program performance analysis, detailed microarchitectural modeling, and hardware-software co-verification. For software developers, SimpleScalar provides detailed performance information about program running on a variety of target platforms with a single host simulationplatform. SimpleScalar is more widely used by architecture designers and researchers. It provides sample simulators ranging from a fast functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction. Used in combination with benchmark programs such as NetBench [2] and CommBench [3], researchers can explore how each component and parameter in the complex modern processor can affect program performance for a certain type of programs. Researchers can also build their own simulator for new architecture and obtain detailed information on how it would perform in a certain application scenario, without the need to really fabricate a chip. It saves cost and time in the *try-and-modify* process of architecture design.

Compared to other simulators, SimpleScalar is superior in several factors. Firstly, it is highly flexible. SimpleScalar includes ve execution-driven processor simulators in the release. They range from an extremely fast functional simulator to a detailed, out-of-order issue, superscalar processor simulator that supports non-blocking caches and speculative execution. The inclusion of such a complex processor implies that users can rely on it to build simulator for their own modern architectures. Second, it has high extensibility. The code is designed to keep extensibility in mind so that users can change the simulator to suit their own need without much hassle. Third, it also has performance advantage over other simulators. According to the user manual, on a 200-MHz Pentium Pro, the fastest, least detailed simulator simulates about four million machine cycles per second, whereas the most detailed processor simulator simulates about 150,000 per second. And last but not least, it is highly portable, and can run on a variaty of host platforms.

We base our simulators for 3 new architectural features on stock SimpleScalar. We have made changes to the base infrastructure as well as the sample simulator, `sim-outorder`, which is an out-of-order issue, superscalar processor with detailed, precise and comprehensive statistics. The 3 architectural features we simulate are:

L3 cache Level-3 cache is an additional layer in the traditional memory hierarchy, lying between level-2 cache and main memory.

Victim cache Victim cache is a small fully-associative cache bound to a normal cache, which usually uses fully-mapping strategy. It is to give a *second chance* to replaced blocks, to reduce potential high conflict misses especially fully-mapped cache.

Adaptive split cache Adaptive split cache is a novel architecture that tries to combine the benefit of split cache and unified cache. It is used as a split cache, except that D-cache and I-cache can *donate* some blocks to each other. The goal is to reach a optimal size division between I- and D-cache, and to allow the division to change to adapt to different program characteristics or different phases of a program.

We use programs in NetBench suite as the testing programs, which is cross-compiled with the target being MIPS architecture. Netbench is a benchmarking suite for network processors. NetBench contains a total of 9 applications that are representative of commercial applications for network processors. These applications are from all levels of packet processing; small, low-level code fragments as well as large application level programs are included in the suite. NetBench is used to evaluate how well an architecture, mostly network processor architecture, can handle typical network relevant computation tasks. Selected programs included in SimpleScalar are also used to test effects of Adaptive split cache.

The rest of the paper is organized as follows. In section 2, we introduce the 3 architectural features we are studying. In section 3, selected results will be given with brief (observation-level) conclusion. In section 4, based on the observations in section3, we will discuss how the feature-relevant configurations can affect program performance. Finally in section 5, a short conclusion and prospect on future research will be presented.

2 Methods

2.1 L3 Cache

Larger caches are both slower and have better hit rates. To ameliorate this tradeoff, many computers use multiple levels of cache, with small fast caches backed up by larger slower caches. With the fast increase in size of main memory and in the speed of the fastest cache, the gap between the cache and main memory is increasing dramatically. Because of this, some processors have begun to utilize as many as three levels of on-chip cache. For example, in 2003, Itanium II began shipping with a 6MB unified level 3 cache on-chip. The IBM Power 4 series has a 256MB level 3 cache off chip, shared among several processors.

We have extended SimpleScalar to support L3 cache. This is achieved by creating an L3 cache before L2 cache is created, and set L2's next level memory to L3 rather than memory. In this way, when a miss happens in L2, instead of retrieving data from memory, L2 will consult L3, which will further handle potential misses and bring data to upper layer of memory hierarchy.

Our experiment relies on practical assumption on latency at different cache layers. Based on realistic product specification, we set L1 latency to 1 cycle, L2 latency to 6 cycles and L3 latency to 12 cycles. The performance is tested with `crc` in NetBench suite.

2.2 Victim Cache

A victim cache is a cache used to hold blocks evicted from a CPU cache due to a conflict or capacity miss. The victim cache lies between the main cache and its refill path, and only holds blocks that were evicted from that cache on a miss. This technique is used to reduce the penalty incurred by a cache on a miss.

For L1 cache, which usually employs fully-mapped mapping scheme due to latency concern, conflict misses can be very common, because for each memory block, there is only one placement choice in this cache. If 2 blocks that share the same cache block but are used at the same time, the cache will keep on replacing the same cache block. In this case, victim cache eliminates this scenario by giving the evicted block a second chance: every

evicted block will be placed in the victim cache temporarily with the hope that it might be used by the main cache soon.

We extend SimpleScalar to support victim cache on L1 D-cache and L1 I-cache. Command line options are added to allow user to specify the size of victim caches. The victim cache itself is implemented as a fully-associative cache using LRU replacement scheme. Both the function (misses, miss_rate, etc) and the timing (latency) are simulated.

2.3 Adaptive Split Cache

Adaptive split cache is novel cache architecture. The basic idea is to combine the advantages of unified cache and split cache.

A unified cache caches both instructions and data. The advantage of this architecture is more flexible use of cache storage. However, since CPU accesses instructions and data in different pattern, caching these together may pollute the cache and degrade cache performance for instructions and/or data. Therefore, 2 split cache for instructions and data are used, to best utilize the accessing pattern. A disadvantage of this architecture, however, is that the expensive cache storage may not efficiently be used because the division of I-cache and D-cache may not be optimal. For example, it is possible that miss rate at I-cache is very low while that at D-cache is very high, and if I could transfer some blocks to D-cache, the overall performance can be improved. Another potential problem of the split cache is its static nature: when program goes through different phases with different memory access patterns, the split between I-cache and D-cache cannot adapt to this change of program patterns.

Based on this consideration, we propose a new architecture called *Adaptive Split Cache*. In this new architecture, I-cache and D-cache are physically on the same chip, but logically accessed separately. In this way, they perform exactly like normal split caches and thus have all the advantages of split caches; besides, the two split caches can *donate* a block to the other.

To allow it to adapt to different programs, several counters are built in each cache to record the number of misses. After a specific number of memory references, the cache will look at its miss records to decide whether the performance is bad and needs change of division of I- and D-cache. Decision on which cache should be the donator/donatee will be based on the previous miss record, the new miss record, and the previous decision on donator/donatee.

Donating part of cache is non-trivial, because it concerns the flush of cache blocks, and change of addressing and replacement of cache blocks. Complex logic in cache is not acceptable, because of the latency consideration at a component so near to CPU. Thus, we use a simple approach based on set-associate cache.

In N-way set-associative cache, the cache is divided into N separate

sub-cache, each of which can be viewed as a fully-mapped cache. Each block to be placed into this cache has N choices of placement. If N-way set-associative cache is used for both I- and D-cache, I-cache can donate *one way* to D-cache so that I-cache becomes $(N-1)$-way set-associative cache, and D-cache becomes $(N+1)$-way set-associative cache, and there is only small change at addressing and replacement in I- and D-caches.

Using this method, the main overhead is due to flush cache blocks in the donated cache *way*. Therefore, the decision of whether to initiate a donation of *way* should also consider this overhead.

The main challenge of this technique is to derive a appropriate decision algorithm of *way donation*, on both whether to initiate a donation and who should be the donator. Due to time restriction, we only implement a naive decision algorithm, motivated by *Newton's method*: *way donation* happens after every K cycles, if the previous donation benefits the performance, repeat that donation and if it hurts the performance, undo that donation.

The most obvious advantage of this technique is to combine the flexible usage of cache storage and separated storage of instructions and data to respect their different access patterns. This technique will hopefully find a best division of I- and D-caches for different programs, and can adapt to different phases of a single program.

Another description of the idea We can also view this method in the following ways. As in Figure 1, two caches, cache A and cache B, are common split caches. Another cache, cache C, is a "shared" cache, i.e. part of it (C_a) can belong to cache A and the remain of it (C_b) can belong to cache B. Yet at the same time any part of cache C can only belong to one of cache A and cache B. The combination of Cache C_a and cache A are called cache Aa. The combination of Cache C_b and cache B are called cache Bb. For the CPU, there are only two separate cache, Aa and Bb. We can dynamically adjust the partition of cache C according to real need.

Partition policy and address mapping:

- Initiate the partition of cache C. Define the direction of is positive, if C_a increases; define the direction is negative is C_b increases.
- In every fixed interval, the hardware will decides whether the current partition of cache C is as good as before.
- When the hardware decides the current partition of cache C is not as good as before, the partition will be changed in the reverse direction;if the hardware decides that the current partition of cache C is better before, the partition will be changed in the same direction.
- If the direction is positive, increase C_a and decrease C_b; if the direction is negative, decrease C_a and increase C_b. That is to say, each time the change of the size of C_a and C_b is a fixed step. Actually, the change

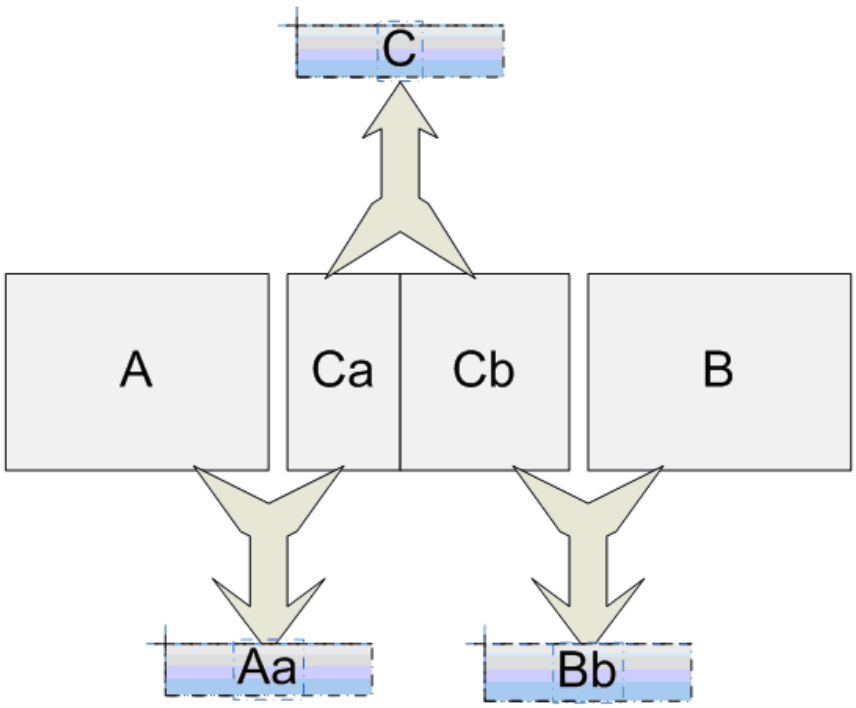

Figure 1: Anatomy of adaptive split cache

means increasing or decreasing the "way" of cache Aa and Bb. So, the index of the block will keep the same.

- Go to 2.

3 Result

In this section, we present results of experiments with our new architectures. Our simulator is based on `sim-outorder`, and is designed to preserve precise timing and function simulation. The statistics is obtained from statistics module derived from that of SimpleScalar. Automatic scripts are used to postprocess the raw output.

3.1 L3 Cache Performance

Our L3 cache performance test is based on the cache organization in Table 1:

We observe on average CPI with fetch queue sizes, and compare that with average CPI with normal 2-level cache architecture with the same L1 and L2 cache settings. The result in shown in Figure 2.

	L1	L2	L3	Memory
latency	1 cycle	6 cycles	12 cycles	150 cycles (and 2 cycles on consecutive access)
size	16K I-cache and 16K D-cache	256K I- and 256K D-cache	2M I- and 2M D-cache	N/A

Table 1: Cache organization in L3 cache test

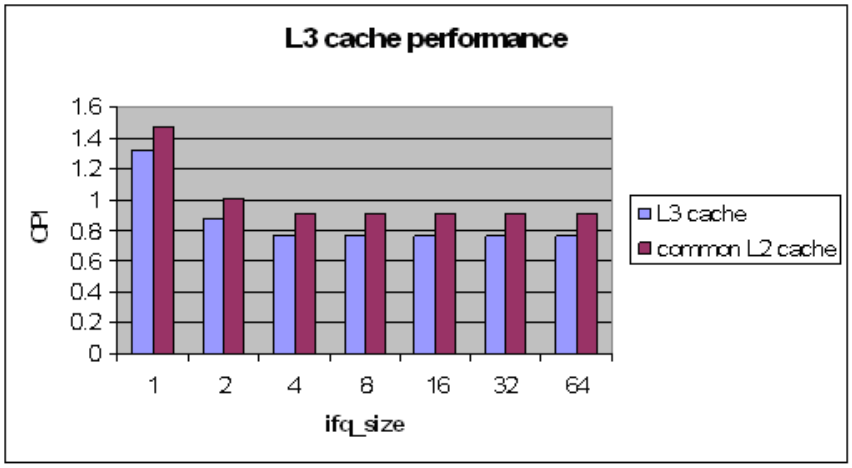

Figure 2: Performance comparison of 2-level and 3-level cache architecture

From Figure 2 it can be observed that , by adding an L3 cache, the CPI decreases as large as 9%. This is because the memory latency is more than 100 cycles, 0.1% more memory accesses will cause about 10% increase in CPI. So the CPU performance can be largely improved by decreasing the main memory access. It is seen from our experiments adding L3 cache can fulfill this reduction in average CPI. For example, when fetch queue has a size of 1, average CPI is reduced to 1.324 from 1.469 by adding the L3 cache.

3.2 Victim Cache

In this test, we add victim cache to level-1 I-cache and/or D-cache, and experiment with different size of victim cache to examine how victim cache can access the performance.

We firstly test the victim cache attached to level-1 I-cache and see how the size of the victim cache can affect the miss rate at that I-cache. The result is shown in Figure 3.

From Figure 3 we can see that the miss rate at level-1 I-cache is reduced with increased victim cache. There is steep reduction when victim size is

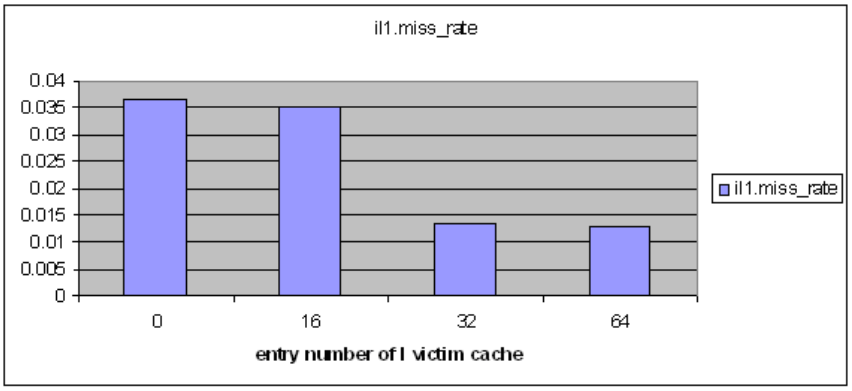

Figure 3: Miss rate at level-1 I-cache is reduced with increased victim cache attached

changed to 32 entries and stays relatively unchanged for further increase in size. This implies that for this application and level-1 I-cache size, 32 entries is an appropriate size of its victim cache.

Similarly, we test the victim cache attached to level-1 D-cache. The result is shown in Figure 4.

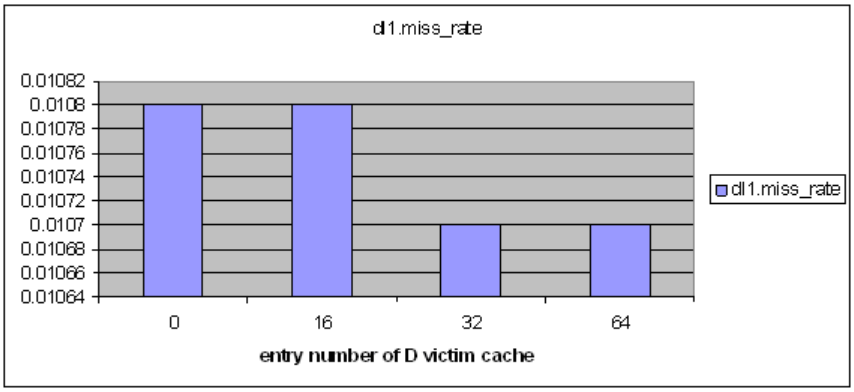

Figure 4: Miss rate at level-1 I-cache is reduced with increased victim cache attached

From Figure 4, it can be observed that 32 entries is also an appropriate size.

We then increase victim caches of I-cache and D-cache simultaneously and observe the change of CPI. The result is shown if Figure 5.

From Figure 5, it can be observed that 32 entries is also an appropriate size when CPI is considered.

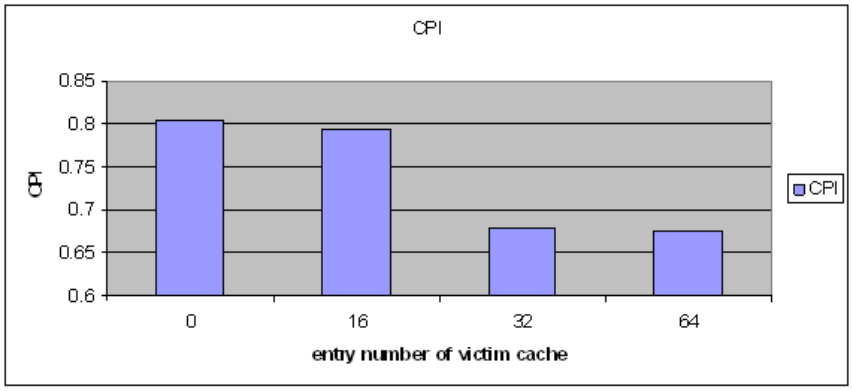

Figure 5: CPI is reduced with increased victim cache attached

	Without ASC	with ASC
CPI	1.2595	1.2601

Table 2: Performance is degraded with Adaptive split cache technique

3.3 Adaptive Split Cache (ASC)

We simulate this technique in 2 different ways. In this first way, the cache will be fully flushed at each *donation* of *way(s)*; in the other way, the *donation* is implemented by changing the cache implementation code at SimpleScalar and will flush only the donated cache blocks. In both ways, we employ the naive decision algorithm described in section 2.3.

Table 2 shows the negative performance gained generated by this technique. The test results show that the performance of the new cache architecture is not as good as that of normal 2-level cache architecture. The reason can be that the initial parameters are not good, that the repartition policy is not good enough or that the statistics is not complete.

Using the 2nd implementation, we generate 3 Figures, listed in Figure 678. In these figures, 4 different architectures are compared. `Unified cache` denotes the architecture using a unified L2 cache; `separate cache` denotes the architecture using a split L2 cache; `scalable cache(1)` denotes the architecture using ASC, and `scalable cache(2)` denotes the same architecture but the intial *denotation* donator and donatee are swapped. The overall L2 cache size is the same among all the above architectures.

From the above figures we can see that the performance of ASC technique is not so good as common L2 cache architectures; and the initial donator/donatee can affect the performance dramatically.

In fact, the bad performance is largely attributed to the naive decision algorithm we are using. Due to time constraint, we cannot derive and im-

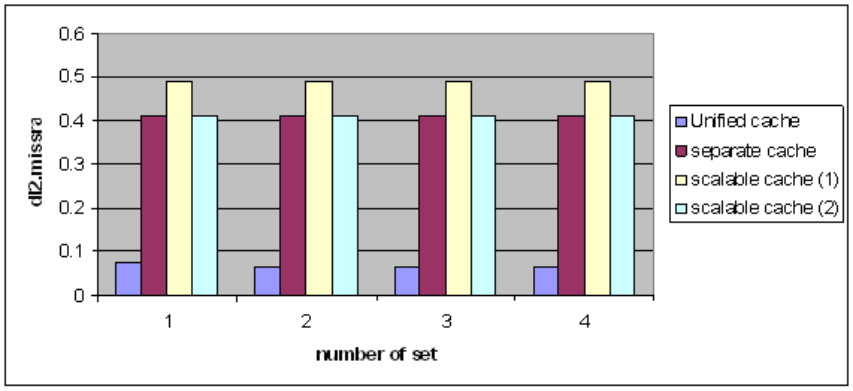

Figure 6: Miss rate at L2 I-cache

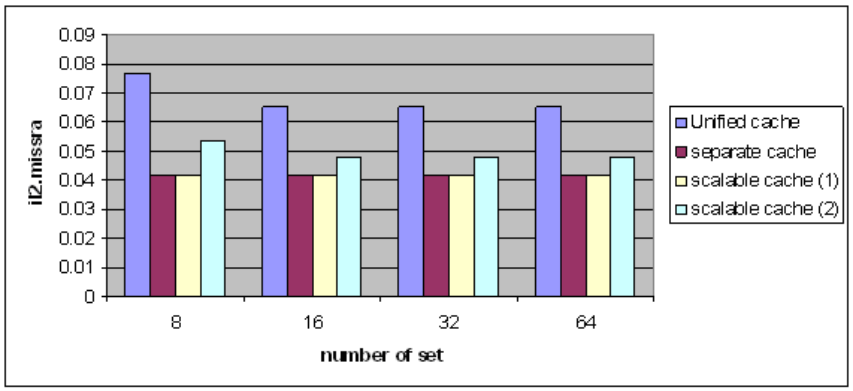

Figure 7: Miss rate at L2 D-cache

plement better decision algorithms.

It is also seen that different applications benefit from ASC to a different level. We have tested ASC with the testing program `test-printf` built in SimpleScalar, and this specific program benefits from ASC techique even with our naive decision algorithm. The performance comparison is listed in Table 3.

From Table 3 we see that ASC technique reduces the misses at L2 I-caches from 1605 to 1041 just by sacrificing misses at L2 D-caches, which increases from 300 to 350. The overall effect is that the CPI decreases from 0.7800 to 0.7725. This shows that ASC can be useful, by looking for a better division between I- and D-cache.

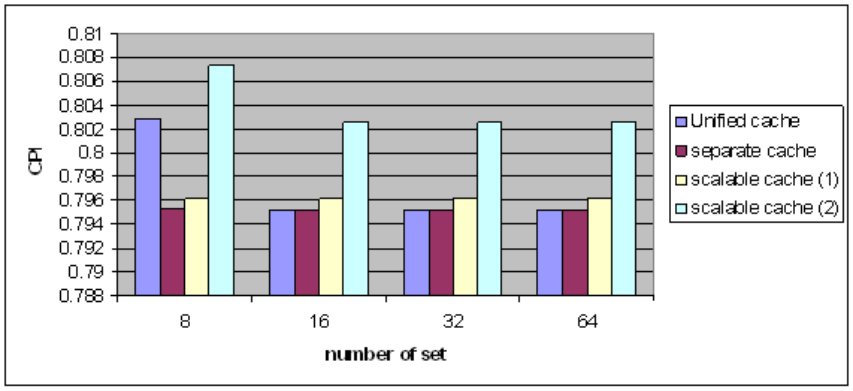

Figure 8: CPI comparison among different architecture

	without ASC	with ASC
CPI	0.7800	0.7725
misses at L2 I-cache	1605	1041
misses at L2 D-cache	300	350

Table 3: test-printf benefits from ASC technique with naive decision algorithm

4 Discussion

L3 cache is between the main memory and L2 cache. We introduce L3 cache because there is a great gap between the latency of the main memory and that of L2 cache. With a L3 cache, the memory access can be decreased a lot, which will decrease CPI and increase the overall performance.

Victim cache is an additional small cache attached to a normal main cache. It can help reduce conflict misses and decrease CPI.

Adaptive split cache is a cache with both advantages of unified cache and split cache. It can take good advantage of the capacity of cache. Meanwhile, it can keep the two ports access, and keep the caching of instructions and data separate from each other. Thus, it should reduce capacity miss as well as conflict miss. However, the effect of adaptive split cache depends on good decision algorithm on cache block donation. So far, our naive algorithm does not benefit the performance, but it shows that this technique can be useful and effective.

5 Conclusion and Future Work

Cache is a critical component of CPU. With L3 cache and victim cache, we can improve the performance of cache, thus improve the performance of CPU. L3 cache aims at decreasing the miss penalty. Victim cache aims at decreasing the miss rate. In addition, we propose a kind of size-scalable cache, which aims at merging the advantages of unified cache and separate cache. Hopefully, size-scalable cache can efficiently use the capacity of cache and have two ports, which will decrease the miss rate further. Though the available result is worse than we expected, we believe it will work if we can find the proper repartition policy.

References

[1] URL: `http://www.simplescalar.com`

[2] Gokhan Memik and William H. and Mangione-Smith and Wendong Hu, NetBench: A Benchmarking Suite for Network Processors, *ICCAD 2001*

[3] Tilman Wolf and Mark Franklin, CommBench A Telecommunications Benchmark for Network Processors, *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, Austin, TX, Apr. 2000

[4] Doung Burger and Todd M. Austin, The SimpleScalar tool set version 2.0, *user guide of SimpleScalar*, URL: `http://www.simplescalar.com`