Reducing Memory Latency Using Cache-based Enhancements

Vijay Gandrapu, Shailesh Samudrala

Department of Electrical and Computer Engineering, University of Florida
Gainesville, Florida, USA
vgandrapu@ufl.edu
shailesh2088@ufl.edu

***Abstract*— This paper implements two cache based enhancements viz. Victim Cache and Trace Cache. Victim caches are small fully-associative caches which are bound to a normal cache. It serves to reduce conflict and capacity misses in the main cache by storing data which is evicted from the main cache. Since Level 1(L1) caches are small usually employ fully-mapped mapping schemes to reduce latency and access time, chances of capacity and conflict misses are high. Therefore, we have implemented Victim Cache for L1 Instruction and L1 Data cache. Our results show an improvement in the performance of the main cache when the Victim Cache is implemented. We have also studied the effect of different configurations of the L1 cache and the Victim Cache on the performance of the system. A Trace Cache stores instructions in their dynamic order of execution, and hence provides a high bandwidth for instruction fetching. Trace Cache is a hardware structure, each line of which stores a snapshot, or trace, of dynamic instruction stream. We have implemented a Trace Cache, and the results show an increase in the performance of the system measured in terms of Cycles per Instruction (CPI), or Instructions per Cycle (IPC). The above two structures have been implemented using SimpleScalar, which is a software toolset designed for modeling and simulation of processor performance. The performance gain attained by implementing the above two features have been measured using the SPEC2000 integer and floating point benchmarks.**

1. I.INTRODUCTION

A. Victim Cache

The performance of modern day processors is limited by long memory latencies involved in fetching data from the main memory. Caches mitigate this long memory latency by storing recently fetched data so that future requests for that data can be served faster.

Cache performance is becoming increasingly important because of the major impact of memory latency on the performance of the processor. On the other hand, Cycles per Instruction (CPI) has been decreasing drastically. These two factors put together result in dramatic increase in the cost of a miss in the caches.

Victim Cache is one technique which is used to decrease the miss cost. The Victim Cache was introduced by Norman P. Jouppi[5]. 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.

Victim caches help reduce misses in the main cache by giving the evicted block a second chance. Whereas Norman P. Jouppi studies the effect of size of the main cache as well as the effect of the line size on the performance of Victim Cache, we study the effect of the associativity of the main cache and the effect of the size of the Victim Cache itself on the performance of the system.

Our results show a significant increase in the performance of the system with the implementation of Victim Cache. We implement Victim Cache by modifying the SimpleScalar toolset [11]. The performance is measured using the SPEC2000 CPU Benchmark suite. [7]

B. Trace Cache

The trend in superscalar design has been wider dispatch/issue window, more resources (i.e. functional units, physical registers, etc) and deeper speculation. However, despite these hardware enhancements, there exist bottlenecks that diminish the throughput.

One such hindrance is the execution of long noncontiguous instruction sequences that cannot be fetched in a continuous stream from traditional instruction caches because instructions are stored in a static order in which they were compiled.

The Trace Cache was proposed by Rotenberg et.al. [6] Trace Cache is a hardware structure, each line of which stores a snapshot, or trace, of dynamic instruction stream. A trace is a sequence of at most n instructions and at most m basic blocks starting at any point in the dynamic instruction stream. A trace is specified by a starting address and a sequence of up to m-1 branch outcomes which describe the path followed.

A line of trace cache is filled as instructions are fetched from the instruction cache. If the same trace is encountered again in the course of executing the program, it is fed directly to the decoder. Otherwise, fetching normally proceeds from the instruction cache. The reason Trace Cache works is because of program properties of temporal locality and easily predicted branch behavior.

Our results show a significant increase in the Instructions per Cycle (IPC) executed by the processor. We use SPEC 2000 benchmarks to evaluate the performance of the above implementation. SPEC2000 is CPU benchmark suite consisting of integer and floating point benchmarks. [7] Trace Cache is implemented using the SimpleScalar toolset. [11]

2. II. Related work

A. Victim Cache

The concept of Victim Cache was first proposed by Norman P. Jouppi [6]. In his paper, Jouppi introduced three hardware techniques to improve the performance of caches viz. Miss Caching, Victim Caching and Stream Buffers.

In Miss Caching, a small fully-associative cache is placed between a cache and its refill path. When a miss occurs, data is returned to both the main cache as well as the miss cache. If a miss occurs in the main cache but the address hits in the miss cache, then the main cache can be reloaded in the next cycle from the Miss Cache. Thus, miss caching effectively reduces the miss penalty. However, the duplication of data which exists in this technique results in wastage of valuable storage space in the miss cache.

The Victim Cache is an improvement on the Miss Cache, and uses a different replacement technique. On a miss in the main cache, the requested data is loaded into the main cache, whereas the victim line which is evicted from the main cache is loaded into the Victim Cache. In case of a miss in the main cache which hits in the Victim Cache, the contents of that line in the Victim Cache are swapped with the contents of the victim line in the main cache. Therefore there is no duplication of data and the memory is effectively utilized. Jouppi's results show that the performance of Victim Cache is always greater than that of Miss Cache.

The third hardware technique is the Stream Buffer. When a miss occurs, the stream buffer begins pre-fetching successive lines of data starting at the miss target. Subsequent accesses to the cache also compare their address against the first item stored in the buffer. If a reference misses in the cache but hits in the buffer the cache can be reloaded in a single cycle from the stream buffer. When a line is moved from a stream buffer to the cache, the entries in the stream buffer can shift up by one and a new successive address is fetched. Jouppi implements the stream buffer as a simple First in First out (FIFO) queue, where the elements of the buffer are strictly removed in sequence.

Yang et.al [1] also implement Victim Cache using SimpleScalar toolset. This is done by modifying the simulator code to support Victim Cache on L1 instruction and data caches. They also study the effect on this miss rate of the L1 caches with increase in the size of the Victim Cache.

Jouppi studies the effect of the size of the main cache and the effect of the line size on the performance of the Victim Cache.

We have used the above two references to implement Victim Caches to support L1 instruction and data caches using SimpleScalar. [11] We study the effect of associativity of the L1 caches on the performance of the Victim Cache, and also the increase in the size of the Victim Cache on the miss rate of the L1 caches, and in turn on the performance of the system.

B. Trace Cache

Many recent studies have focused on high bandwidth instruction fetching. Yeh et.al [8] propose a mechanism for instruction fetching by predicting multiple branch targets every cycle. This includes a feature called Branch Address Cache. A hit in the Branch Address Cache combined with multiple branch prediction produces the branch targets of several basic blocks.

Another study by Franklin et.al [9] is on similar lines but with a new method for multiple branch prediction. Rather than making a branch prediction for every path, this scheme makes one prediction for multiple paths using more accurate branch prediction mechanisms.

However, Rotenberg et.al [6] suggest that these approaches are problematic because pointers to all noncontiguous blocks must first be generated before instruction fetching can begin. Also since multiple branch prediction is used, instruction caches must support simultaneous access to noncontiguous cache lines. Even after the instructions are fetched, they must be arranged in dynamic sequence which only increases the Cycles per Instruction.

Rotenberg et.al proposed a new structure called Trace Cache. The Trace Cache overcomes the above mentioned problems by caching the dynamic sequence of instructions, called a trace. If the dynamic sequence exists in the Trace Cache, it can directly be fed into the pipeline without the need for any code re-ordering. The disadvantage of this technique is data redundancy i.e. the same instructions may reside both in the instruction cache and the Trace Cache. The Trace Cache and Fetch Unit, as proposed by Rotenberg el.al is shown in Figure 1.

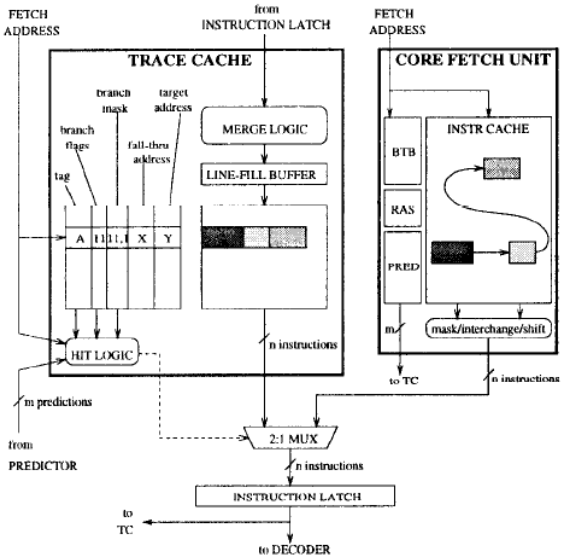

Fig.1 Trace Cache and Fetch Unit [6]

The Trace Cache is accesses in parallel to the Instruction Cache and the Branch Target Buffer using the current fetch address. At the same time, the Predictor generates multiple branch predictions. The fetch addresses, along with the

predictions are used to determine if a trace in the Trace Cache matches the predicted sequence of blocks. If the result is a hit, then the entire trace which exists in the Trace Cache is fed into the instruction latch, and the Instruction Cache is bypassed. If the result is a miss, then instruction fetching proceeds normally through the Instruction Cache.

We have used the above references to implement a Trace Cache using the SimpleScalar [11] simulator. We have modified sim-outorder, a module of SimpleScalar which simulates out of order execution of instructions to include the Trace Cache structure. The performance of the system is measured increases as the Instruction Fetch bandwidth has effectively increased, and is measured in terms of Instructions per cycle (IPC). The benchmarks used are SPEC2000. [7]

III. METHODOLOGY

A. Victim Cache

As stated in the previous section, we have implemented a fully-associative Victim Cache by modifying the SimpleScalar source code. [11] In particular, modifications were made to the sim-outorder module of SimpleScalar. The Victim Cache was defined, and initialized within this module.

Existing cache functions provided by SimpleScalar such as cach_create(), cache_accesss(), and cache_probe() were used to handle the Victim Cache implementation.

Since we planned to implement Victim Cache on L1 Instruction and Data caches, changes were made to their miss handler functions. A miss on the L1 cache, but a hit on the Victim Cache caused the victim line in the L1 cache and the hit line in the Victim Cache to be swapped. In case of a miss in both Victim Cache and L1 Cache, the line evicted from the L1 cache (if any), was stored in the Victim Cache.

If the Victim Cache was full, then Least Recently Used (LRU) replacement policy was used to store the new line. Command line options were provided to enable the user to specify whether the Victim Cache should be used. We used the cache configuration in Table 1 in our implementation.

TABLE I
CACHE CONFIGURATION

Cache	**Specifications**
L1 Instruction cache	2KB, 32B block, LRU, 1-cycle hit latency
L1 Data cache	2KB, 32B block, LRU, 1-cycle hit latency
Unified L2 cache	512KB, 128B block, LRU, 4-way, 20-cycle hit latency
Memory Latency	100 cycles for the first chunk, 5 cycles for inter-chunks
Victim cache	LRU with 1-cycle hit latency

We also studied the effect of the associativity of the L1 caches on the performance of the Victim Cache. We used direct-mapped, 2-way associative, 4-way associative, and 8-way associative L1 caches for our study. The effect of size of the Victim Cache on the miss rates of the L1 caches was also studied. Victim Cache sizes of 8 lines, 16 lines, 32 lines and 64 lines were used for this study.

Changing the size of the Victim Cache and the associativity of the L1 cache would result in larger access times for the respective caches. However, this was not factored into our implementation.

We have used SPEC2000 benchmarks to obtain the results of the above studies. These results are described and analyzed in the next section. Figure 2 depicts the place of the Victim Cache in the memory hierarchy.

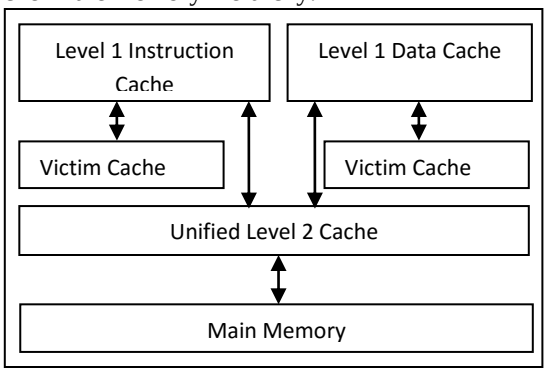

Fig. 1 Victim Cache in Memory Hierarchy

B. Trace Cache

Trace Cache was also implemented by modifying the SimpleScalar simulator source code. As described earlier, the Trace Cache is a special Instruction cache that caches dynamic instruction streams. A trace is a sequence of at most n instructions and at most m basic blocks starting at any point in the dynamic instruction stream. A trace is specified by a starting address and a sequence of up to m-1 branch outcomes which describe the path followed. A line of trace cache is filled as instructions are fetched from the instruction cache. If the same trace is encountered again in the course of executing the program, it is fed directly to the decoder. Otherwise, fetching normally proceeds from the instruction cache. Trace Cache was implemented by modifying the sim-outorder module of SimpleScalar. Trace Cache was defined and initialized in this module. sim-outorder consists of 6 major functions which simulate different stages of the pipeline. These functions are:

- ruu_commit( )
- ruu_writeback ( )
- ruu_refresh ( )
- ruu_issue ( )
- ruu_dispatch ( )
- ruu_fetch ( )

Three out these six functions have been modified to implement Trace Cache. ruu_fetch ( ) models the instruction fetch bandwidth. Inputs of this module are: Program Counter (PC), state of the predictor, and mis-prediction detection state. The output is a Fetch Instruction Queue. This function has

been modified such that, if there is a hit on the Trace Cache, instructions are fed directly from the trace cache to the Fetch Instruction Queue because the dynamic instruction stream already exists in the trace cache. If there is a miss on the Trace Cache, then instructions are fetched from the L1 instruction cache. The ruu_dispatch( ) function handles machine decode and register renaming. In this function, if mis-predicted branch instructions are detected, then instruction fetching from the Trace Cache is stopped, and the fetch is redirected to the L1 instruction cache. The function, ruu_commit ( ), models the in-order commit of instructions. The Trace Cache is filled in this function. We have implemented the Trace Cache as a direct-mapped cache with partial matches. This means that if only a few of the blocks of the Trace Cache lines hit, these blocks are fetched from the Trace Cache. The Trace Cache can also be modeled as a set-associative cache, in which case path associativity can be supported. In path associativity, multiple paths originating from the same address can be stored in the Trace Cache. Besides instructions, additional information fields needs to be stored in the Trace Cache. These fields are: tags and branch flags. A tag is the address of the first instruction in a given Trace Cache line. A branch flag is stored with every branch instruction in the Trace Cache. It indicates whether the path followed by that branch, i.e. if the branch is taken or not taken. We have implemented the Trace Cache with 64 lines, 4kB of instructions and 712 bytes for tags/branch flags. The SPEC2000 CPU Benchmark suite was used to analyze the performance of the system after the inclusion of Trace Cache. The system was analysed using the configuration in Table 2.

TABLE III
SYSTEM CONFIGURATION

Unit	Specification
Commit Bandwidth	4
RUU Size	8
IFQ Size	4
L1 Instruction Cache	64kB Direct-mapped
L1 Data Cache	64kB 4-way associative
L2 Cache	4 MB 8-way associative

These results are described and analyzed in the next section.

IV. RESULTS

A. Victim Cache

As we have stated earlier, the Victim Cache was implemented by modifying the sim-outorder module of SimpleScalar which is an out-of-order issue, superscalar processor simulator which generates detailed, precise and comprehensive statistics. The results of our studies are shown below. Figure 3 depicts the reduction in L1 Instruction and Data cache miss rates due to the implementation of an 8 line Victim Cache. Three of the SPEC2000 benchmarks are used: gcc, gzip and equake. gcc and gzip are integer benchmarks whereas equake is a floating point benchmark. The results show a drastic decrease in the miss rates with the implementation of the Victim Cache. Figure 3 clearly depicts the effectiveness of the Victim Cache in reducing the miss rates of L1 Caches, and hereby its ability to ameliorate the bottleneck of memory latency on the performance of the processor. Figure 4 represents the miss rates of the L1 Data Cache with varying size of the Victim Cache. A clear but diminishing degree of decrease in the miss rates can be observed. In the case of equake, the miss rate is almost zero when the size of the Victim Cache is greater than 32 lines. However, many more factors need to be taken into account to determine the size of the Victim Cache. Firstly, the Victim Cache should not be so large that the access-time of the Victim Cache increases so much that it negates the advantages of implementing the Victim Cache. Further, the size of the chip, power consumption, and heat dissipation, are all factors which play a part in determining the size of the Victim Cache. Figure 5 shows the effect of Victim Cache on the Cycles per Instruction (CPI) of the processor. It is clearly seen that there is a drastic reduction in the CPI with the implementation of Victim Cache. Further increase in size of the Victim Cache further decreases the CPI of the processor, but the decrease is not very significant. This can be related directly to the reduction in miss rates which is depicted in Figure 3, where the degree of reduction in miss rates decreased gradually with the increase of the Victim Cache. The effect of different associativites of the L1 cache on the performance of the Victim Cache in terms of L1 Data cache hit rates is depicted in Table 3. Direct-mapped L1 Caches have the fastest access times. As the associativity of the cache increases, the access time also increases. Therefore, L1 caches are always implemented as direct-mapped caches. The problem with direct-mapped caches is that the numbers of conflict misses are high. However the Victim Cache addresses this concern by providing the data evicted form the L1 cache a second chance. Table 3 supports the above statements. It can be seen that implementing a small Victim Cache of 8 lines is as good as using 8-way L1 associative caches. Since L1 caches have very high hit rates, caches should be designed so that the access times are as low as possible. An 8-way L1 associative cache does not serve that purpose. Implementing a Victim Cache is a better option because the latency of the Victim Cache is very less. It can also be seen that having an 8-way L1 cache supported by a 16 line Victim Cache brings up the hit rates of the L1 Data cache to almost 1. However, this may not be a feasible option because of the large access times that would be involved.

B. Trace Cache

Trace Cache was implemented using the sim-outorder module of the SimpleScalar simulator. The sim-outorder module contains 6 major modules which model different stages of the pipeline. These modules are listed in the previous section. Three out of these six functions were modified to implement the Trace Cache viz. ruu_fetch( ), ruu_commit( ), and ruu_dispatch ( ). Figure 6 depicts the

increase in the Instructions per Cycle (IPC), with the implementation of the Trace Cache. SPEC2000 benchmarks: gcc, vpr and bzip2 are used to measure the performance. Figure 6 clearly proves the usefulness of the Trace Cache. The Trace Cache caches the dynamic order of instructions for future use. The advantage is that if the same trace of instructions needs to be accessed again, code reordering and scheduling need not be done, thus resulting in fewer numbers of steps during the Instruction Fetch stage. Since the Trace Cache is responsible for increasing the Instruction Fetch Bandwidth, the processor is able to execute more number of instructions per cycle. The Trace Cache can be accessed in parallel to the Instruction Cache, and if the Trace is formed, it is fed directly into the Fetch Instruction Queue, bypassing the Instruction Cache. The Trace Cache implemented here is a direct-mapped cache with partial matches. We have implemented the Trace Cache with 64 lines, 4kB of instructions and 712 bytes for tags/branch flags.

V. Conclusions and Future Work

The results which were analyzed in the previous section clearly show the advantages of the two cache optimizations implemented in this paper viz. Trace Cache and Victim Cache. They also show how these optimizations result in increased processor performance which is seen through the increase in Cycles per Instruction (CPI) or through the decrease in Instructions per Cycle (IPC). With the IPC of processors increasing continuously, the importance of cache based enhancements should also increase. Future work could include further research in the above 2 fields, or could also involve developing better, faster techniques which address the memory latency problems present in today's computers. Future work on Victim Caches could involve determining an ideal size of Victim Caches taking into account access times, chip size, power consumption and heat dissipation factors. Further research in the field of Trace Cache may focus upon the effectiveness of a direct-mapped Trace Cache versus the effectiveness of a set-associative Trace Cache and the effect of size of the Trace Cache. Branch Prediction techniques also have an effect on the Trace Cache efficiency, and a prospective area for future research. The implementation of a Victim Cache to support a Trace Cache is also an interesting area to work on.

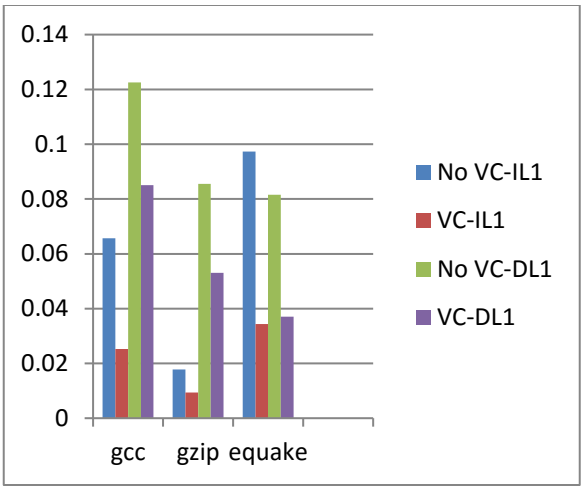

Fig. 3 L1 Cache Miss rates with an 8 line Victim Cache

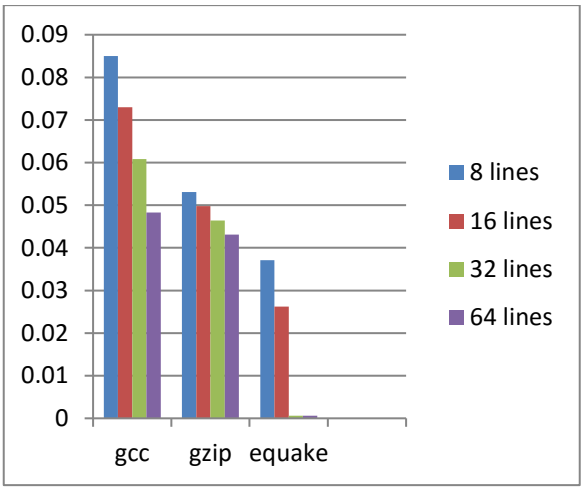

Fig. 4 Miss Rates of L1 Data Cache for Different sizes of Victim Cache

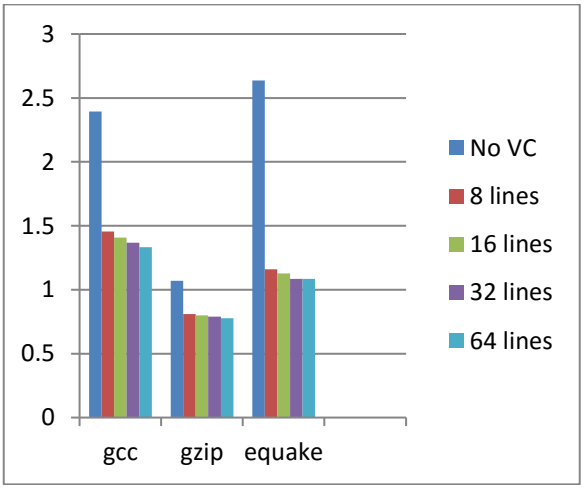

Fig. 5 CPI of Processor for varying Victim Cache Size

TABLE IIIII
EFFECT OF ASSOCIATIVITES ON VICTIM CACHE

Level1 Caches	**Direct-mapped**	**2-way**	**4-way**	**8-way**
No victim cache	0.9184	0.9487	0.9552	0.9665
8 entries	0.9629	0.969	0.9729	0.9911
16 entries	0.9738	0.9841	0.998	0.999

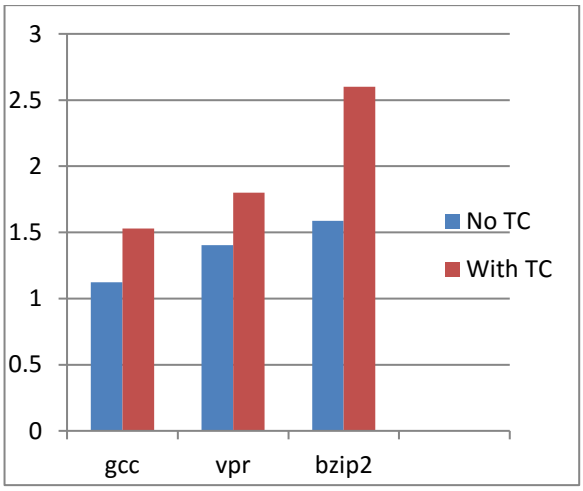

Fig. 6 CPI of the Processor with Implementation of Trace Cache

ACKNOWLEDGMENTS

We would like to thank Dr. Ann Gordon Ross for providing us with valuable guidance which helped us immensely throughout this project. We would also like to thank Akshay V Kale, who helped us out with some valuable advice when we were in some difficulty.

REFERENCES

[1] "Architecture Tuning Study: the SimpleScalar Experience" Jianfeng Yang and Yiqun Cao

[2] Trace Cache and Multiple Branch Prediction: Wangli Yang, Chen Zhou

[3] 2007 Doug Burger and Todd M. Austin, The SimpleScalar tool set version 2.0, user guide of SimpleScalar,
URL: http://www.simplescalar.com

[4] J. Hennessy and D. Patterson, Computer Architecture A Quantitative Approach, 4th Ed., Elsevier Inc., 2007

[5] Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers: Norman P. Jouppi

[6] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. 29th International Symposium on Microarchitecture, Dec. 1996.

[7] SPEC2000 CPU Benchmark suite.
URL: http://www.spec.org/cpu2000/

[8] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache.

[9] S. Dutta and M. Franklin. Control flow prediction with treelike subgraphs for superscalar processors.

[10] The MIRV SimpleScalar/PISA Compiler Matthew Postiff, David Greene, Charles Lefurgy, Dave Helder and Trevor Mudge.

[11] SimpleScalar Toolset:
URL: www.simplescalar.com