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 proposes to implement two cache based enhancements viz. Victim Cache and Trace Cache, and measure the improvement in performance of the system in terms of the reduction in memory latency.**

I. INTRODUCTION

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-based enhancements improve the performance of Caches, thus further improving the performance of the processor. This paper proposes to implement two cache-based enhancements-a Trace Cache and a Victim Cache.

II. VICTIM CACHE

Victim cache is a small fully-associative cache bound to a normal cache, which usually uses fully-mapping strategy. It serves to reduce conflict or capacity misses in the main cache by storing data recently evicted from the main cache. 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. L1 caches usually employ fully-mapped mapping schemes to reduce latency. Thus, conflict misses can be very common in L1 caches because each block of memory has only one placement choice in the cache. Victim caches help reduce these conflict misses by giving the evicted block a second chance. [1][8]

III. 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. 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. 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.[6][7]

IV. IMPLEMENTATION AND SIMULATION

A. Victim Cache

We will use SimpleScalar to implement Victim Cache bound to L1-D Cache and L1-I Cache. sim-outorder, which is an out-of-order issue, superscalar processor with detailed, precise and comprehensive statistics, will be used to measure the performance of the Victim Cache. The victim cache is implemented as a fully-associative cache using LRU replacement scheme. The performance will be measured in terms of miss-rate of the main cache, as well as the CPI of the processor. The Base configuration of the system will be as follows:

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

The access times can be modeled using sim-outorder module of SimpleScalar. We will then measure the effect on performance of the victim cache by varying the associativity of the L1 cache. [1][2]

B. Trace Cache

Trace Cache will be implemented using SimpleScalar Simulator. Trace Cache is added beside the conventional instruction cache to capture the dynamic instruction sequences. The Trace Cache will be implemented with 64 lines, 712 bytes for tags/control and 4kB of instructions. [2][6]

C. Benchmarks

NetBench Benchmarks will be used to measure the performance of the victim cache. SPEC 2000 Benchmarks will be used to test the performance of the Trace Cache. [5][7]

V. RELEVANCE

Cache-based enhancements are important because of the adverse effect of long memory latencies on the performance of the Processor. The two techniques discussed in this paper are widely used and are known to improve the performance of the processor by a big factor.

VI. ANTICIPATED RESULTS

We hope to establish the fact that the above two techniques greatly improve the performance of the processor. We will also study the performance of the Victim Cache for different associativities of the L1 cache.

ACKNOWLEDGMENT

We would like to thank Dr. Ann Gordon-Ross for her guidance and for giving us the opportunity to work on this paper.

REFERENCES

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

[2] 2007 Doung Burger and Todd M. Austin, The SimpleScalar tool set version 2.0, user guide of SimpleScalar, URL: http://www.simplescalar.com

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

[4] Memory Hierarchy, URL: http://en.wikipedia.org/wiki/Memory_hierarchy

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

[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] Trace Cache Bing Chen , Musawir Ali.

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