Evaluation of a Loop Cache Optimization for a Simulated Pipeline

John David Eriksen
EEL5764 Fall 2007
john.eriksen@yahoo.com

Abstract

A great part of the execution time of many applications is spent in the execution of small, well-defined program loops. In these applications, performance can be improved by the addition of a small instruction cache preloaded at design time with the instructions that correspond to an often-executed program loop. The proposed loop caching technique described is shown to be capable of yielding a performance savings, especially for memory hierarchies that must content with higher instruction cache miss penalties and sub-optimal instruction cache sizes. This performance improvement is experimentally verified by testing the loop cache technique against a variety of benchmarks and memory hierarchy configurations.

1. Introduction

The memory hierarchy of a processor presents an attractive target for optimization. In terms of physical size on-chip, it is often proportionately large. Since it delivers data and instructions to the processor itself, it must be as fast as possible while consuming as little power and energy as possible. One of the aspects of the memory hierarchy that is very frequently the target of optimizations, are the uppermost levels of memory. These are the L1 instruction and data caches. Since this component of the cache hierarchy is often designed to capture the majority of cache accesses, it is very frequently targeted for optimization.

In terms of program behavior, a great majority execution time is spent inside of loops. Optimizing memory architecture performance to handle loops efficiently would thus yield substantial performance, power, and/or energy enhancements. Studies have been performed that verify that it is indeed the case that studying and optimizing loop behavior can result in system improvements [3][4]. Many optimization schemes have been proposed that take advantage of this observation [1][2][5][8].

The scheme proposed here, unlike those schemes discussed in [1][2][5][8] aim at using the loop cache for improving performance in terms of execution cycles per instruction rather than reducing total energy consumption.

2. Related Work

2.1 Compiler Assisted Loop Caching Scheme

Bellas, et. al. explored L1 instruction cache optimization in their work on a hybrid software-hardware approach to optimizing L1 instruction cache access by using the compiler to select frequently executed basic blocks and loading them into a loop cache [1].

2.2 Low-Cost Program Loop Caching

An extension to the compiler-assisted loop caching scheme was proposed in [2]. This technique allowed for the caching of partial loops in the loop cache. Like 1.1, the technique involved both compiler and hardware.

2.3 Combined Dynamic and Preloaded Loop Caching

A dynamic loop caching approach attempts to make use of profiling information readily available at run-time. However, the complexity of this approach makes it difficult to support nested loops or loops with other such complex internal branching behavior. The approach explored here attempted to overcome this difficulty by allowing for the preloading of such complex loops [5].

2.4 Customized Loop Caches

Memory optimization techniques may sometimes have a tendency to offer improvements only when paired with programs that can take advantage of the benefits they provide. Sometimes, a memory optimization technique may yield very little improvement, or may even negatively impact the performance or power/energy consumption of a system. In [6] and [7], this tendency was acknowledged and a solution was proposed. A loop cache can be configured to suit a particular application through the use of design-time analysis and profiling techniques.

3. Methodology

The proposed loop cache is a software and hardware optimization. The target system is a simple five-stage pipelined system that uses a very small subset of the MIPS instruction set. In terms of memory hierarchy, it has separate instruction and data memory and a configurable L1 cache. The L1 cache has a fixed block size of a single instruction, a variable number of blocks, and a variable cache miss penalty. The loop cache itself is considered to have an optimal access time of one cycle. The loop cache optimization was tested using a small suite of benchmarks, each of which was tested with a few different cache configurations.

3.1 Benchmarks

The benchmark suite employed in evaluating the loop cache optimization consisted of four benchmarks. These are the straight, simple, busy, and nested benchmarks. The straight benchmark is a single non-looping block of instructions. The simple benchmark, is a looping block of instructions with a very short number of iteration loops. The busy benchmark takes the simple benchmark and executes the loop many more times. Finally, the nested benchmark is a complexly nested loop such as might be found in an algorithm that must traverse a three-dimensional data structure. The benchmarks contain a mix of looping and non-looping blocks of instructions. Only the looping blocks were targeted for loading into the preloaded loop cache.

3.2 Cache Configurations

Each benchmark was tested a total of eighteen times. The configurable parameters included the loop cache itself (on or off), the instruction cache size (optimal, sub-optimal, or effectively off configurations), and the cache miss penalty (none, 10 cycles, or 20 cycles). In terms of the instruction cache size, an optimal size is simply a cache large enough to cache all instructions in a benchmark. A sub-optimal instruction cache size is half the number of instructions in a benchmark, rounded up. Finally, the effectively off instruction cache simply has size one. With a size of one, all accesses to the cache will miss.

4. Experiment

4.1 Experimental Setup

The experiment was conducted as a simulation. The simulation consisted of several complete executions of each benchmark corresponding to all available memory architecture configuration parameters.

4.2 Experimental Results

In terms of performance gains, the preloaded loop cache was shown to provide gains only in the case that the miss penalties of the L1 cache are worse than the access time to the preloaded loop cache. The greatest gains were observed in the busy and nested benchmarks with long cache miss penalties and sub-optimal L1 cache sizes. Consult the appendix for graphs of results.

5. Conclusions

The preloaded loop cache optimization explored here was shown to provide significant performance gains only when used as the uppermost level of the instruction memory architecture and only when the cache miss penalty was fairly substantial and/or the instruction cache size was sufficiently large.

6. References

[1] Low-Cost Embedded Program Loop Caching – Revisited. Lea Hwang Lee, Bill Moyer, John Arends, 1999.

[2] Energy and Performance Improvements in Microprocessor Design using a Loop Cache. Nikolaos Bellas, Ibrahim Hajj, Constantine Polychronopoulos and George Stamoulis, 1999.

[3] A Study on the Loop Behavior of Embedded Programs. Jason Villarreal, Roman Lysecky, Susan Cotterell, and Frank Vahid, 2001.

[4] Tuning of Loop Cache Architectures to Programs in Embedded System Design. Susan Cotterell and Frank Vahid, 2002.

[5] Dynamic Loop Caching Meets Preloaded Loop Caching - A Hybrid Approach. Ann Gordon-Ross and Frank Vahid, 2002.

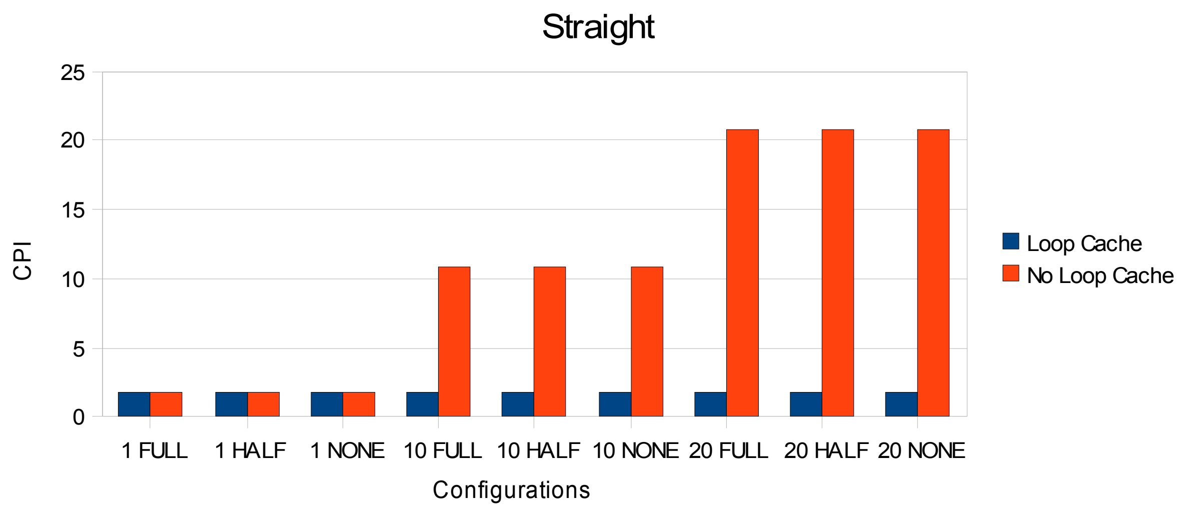
[6] Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. Ann Gordon-Ross, Susan Cotterell and Frank Vahid, 2002.

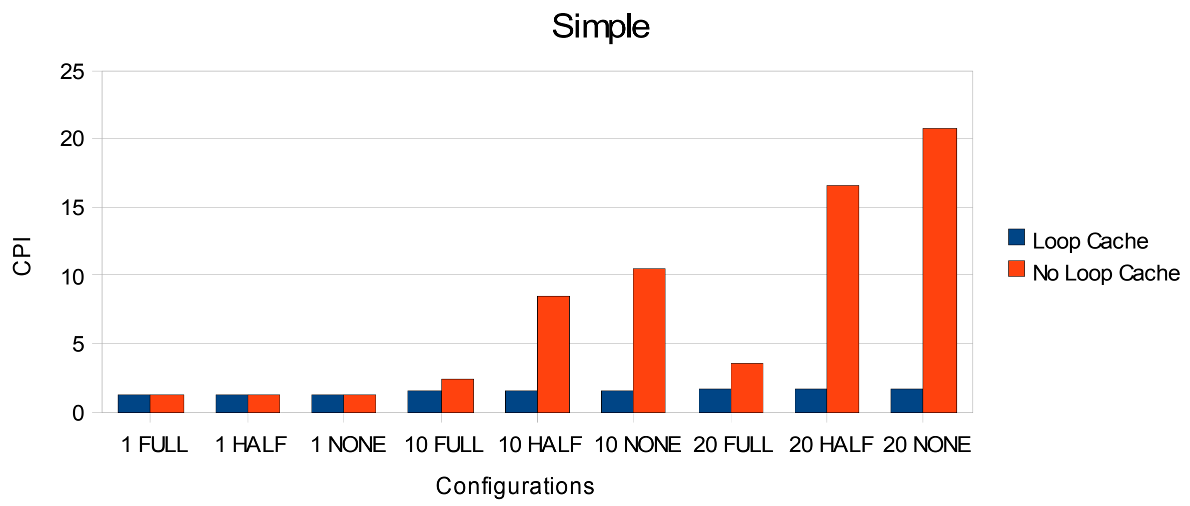
[7] Synthesis of Customized Loop Caches for Core-Based Embedded Systems. Susan Cotterell and Frank Vahid, 2002.

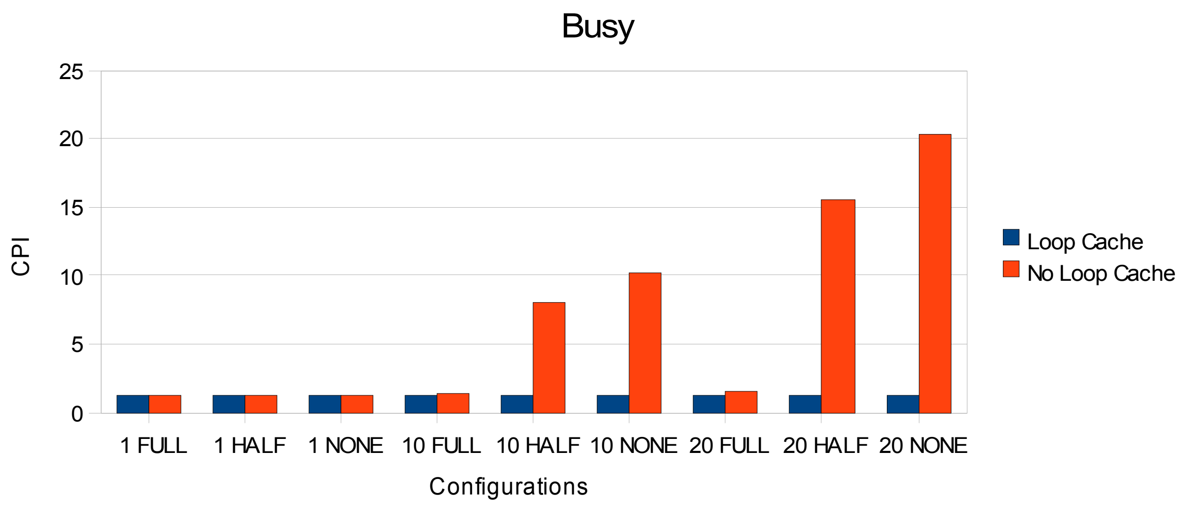
[8] Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors. Murali Jayapala, Francisco Barat, Tom Vander Aa, Francky Catthoor, Henk Corporaal, and Geert Deconinck, 2005.

7. Appendix

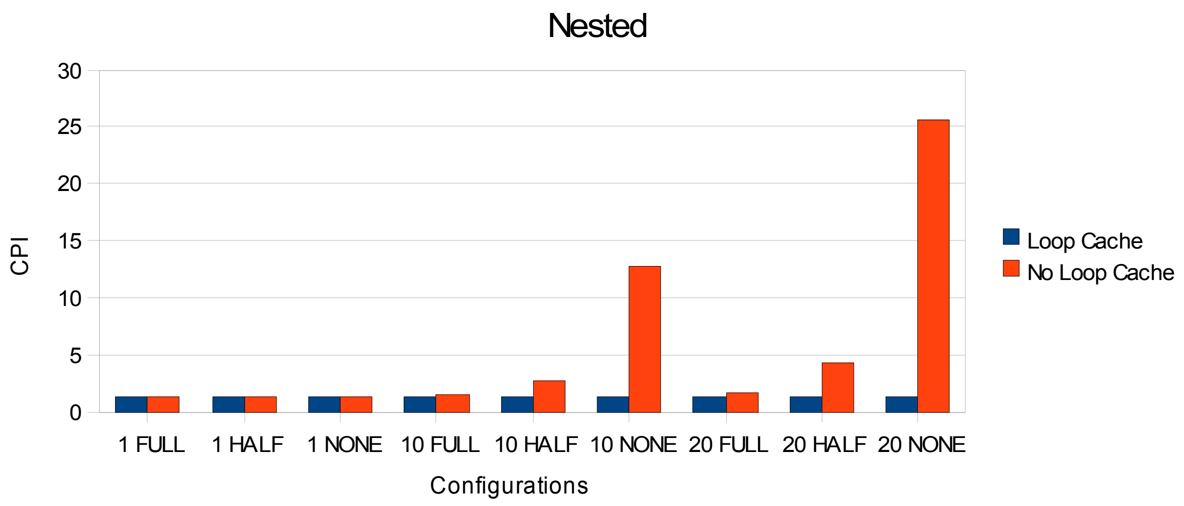
The graphs below present the experimental results in graphical form. There are four benchmarks. These are named straight, simple, busy and nested. Each graph shows the execution time of the benchmark in terms of CPI on the Y axis, as well as the cache configurations explored on the X axis. The configurations are code named 1 FULL, 10 HALF, 20 NONE, etc. The numerical digit indicates the L1 cache miss penalty, and FULL, HALF, or NONE indicates the L1 cache size relative to the total size of a benchmark in terms of instructions.







Busy
CPI
25
20
15
10
5
0
1 FULL
1 HALF
1 NONE
10 FULL
10 HALF
10 NONE
20 FULL
20 HALF
20 NONE
Configurations
Loop Cache
No Loop Cache



Nested
CPI
30
25
20
15
10
5
0
1 FULL
1 HALF
1 NONE
10 FULL
10 HALF
10 NONE
20 FULL
20 HALF
20 NONE
Configurations
Loop Cache
No Loop Cache