Exploring the Trace Cache Design Space

Paul W. Lee, Mahesh J. Madhav, Anamaya Sullerey
{paulwlee, madhavmj, anamaya}@stanford.edu

EE482a – Prof. William Dally
Stanford University

May 30, 2000

Abstract

In this report, we investigate a number of design issues of the trace cache. We also propose an approach to enhance the performance of trace caches and conduct experiments to determine its effectiveness.

Chapter 1

Introduction

In order to achieve higher performance, the execution bandwidth of modern superscalar processors have steadily increased. However, in order to translate this increase in execution bandwith to higher performance, it is necessary to supply the execution units with more and more meaningful instructions per cycle. Conventional caches are limited in this respect, since frequent branches limit the number of instructions that can be used within a single cache line. As one solution to this problem, Peleg et al[1], and Rotenberg et al[4] propose a fetch mechanism called the trace cache.

The trace cache is a structure that stores dynamic instruction sequences gathered during runtime, in units of basic blocks. Whenever there is a hit in the trace cache, instructions can be immediately sent to the issue unit without realignment, and instructions that would normally cross cache line boundaries can be fetched in the same cycle. A number of improvements were proposed to the original trace cache in Friendly et al [2], Jacobson et al[3] and other papers.

For this project, we added the trace cache mechanism to an execution driven simulator to study the benefit of trace caches in a realistic machine, as opposed to a trace driven simulation assuming unlimited resources. With the trace cache-enabled simulator, we studied a number of design issues concerning trace caches. These were: utilization of each basic block in the trace line, the performance impact of varying the number of basic blocks per cache line, the performance gain for increasing numbers of lines in the trace cache, and performance differences between two methods of filling the trace cache entries. We also devised and implemented an approach to enhance the performance of the trace cache, and conducted experiments to determine its effectiveness.

This report consists of four parts. The next part discusses the design space, the overall simulation methods, and experiments and results for a number of different design choices. The third part introduces a possible enhancement to the trace cache and briefly explains the results of the simulation. The last part includes possible directions for additional work, insights gained from the project, including aspects of simulation methodology as well as issues concerning the trace cache itself.

Chapter 2

The basic Trace cache design space

2.1 Design space

The key design choices in the basic trace cache architecture are the number of entries in the trace cache, the number of basic blocks that can be accommodated per entry, and the replacement policy. We conducted experiments to determine the effect of the first two, and also compared two methods of obtaining trace entries from the execution path.

2.2 Simulation methods

We use the simplescalar 3.0 tool set as our base simulator. The base machine is a realistic out-of-order machine, with 8KB direct mapped instruction and data L1 caches, 64KB 4 way associative 64KB unified L2 cache, 8-wide decode, issue, and commit bandwidth.

In simulating the trace cache, a number of design choices were made. First, all types of branch or jump mark an end of a basic block. Second, the traces are truncated when the end of the trace cache line is reached, and there are no continued lines as in [1]. The size of each line is 20 instructions. The replacement policy for the trace lines is least recently used.

We ran the four SPEC95 integer benchmarks, as well as a number of linear data structure benchmarks. All benchmarks were run to completion, and statistics were gathered skipping the first 10 million instructions.

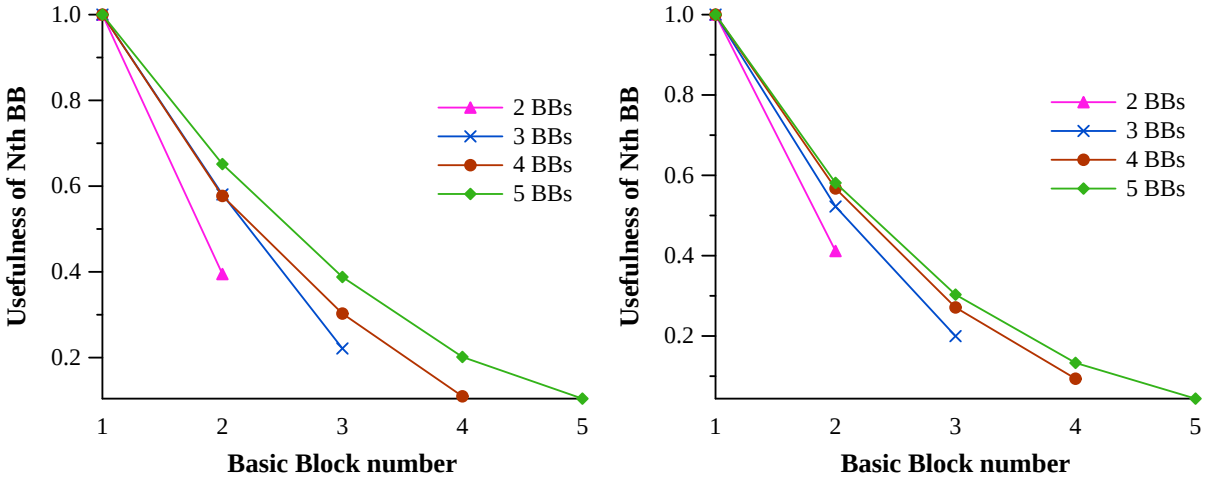
The following sections discuss the results in detail.

2.3 Results

2.3.1 Utilization of blocks

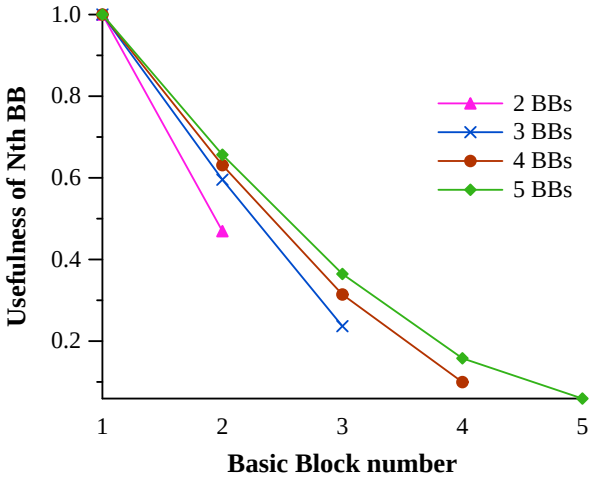
In order to gain insight on the optimal number of basic blocks for the trace cache, we ran a set of benchmarks varying the number of basic blocks. For each successive basic block in the trace cache

line, we tallied up the number of times that the basic block was used. The results are plotted in the following figures. For this experiment we took the trace cache line size of twenty instructions.

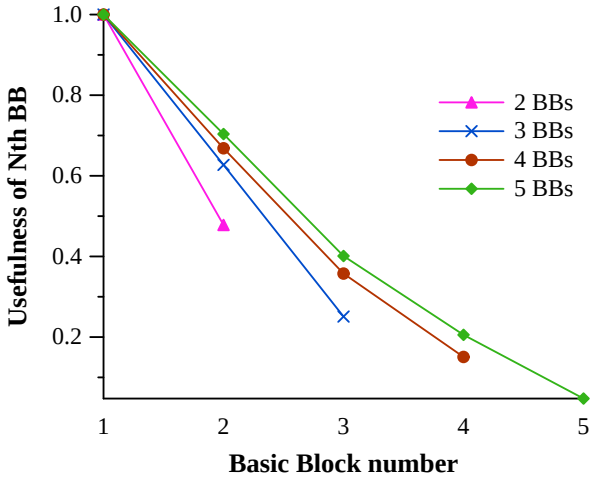


s95-compress-small

s95-gcc

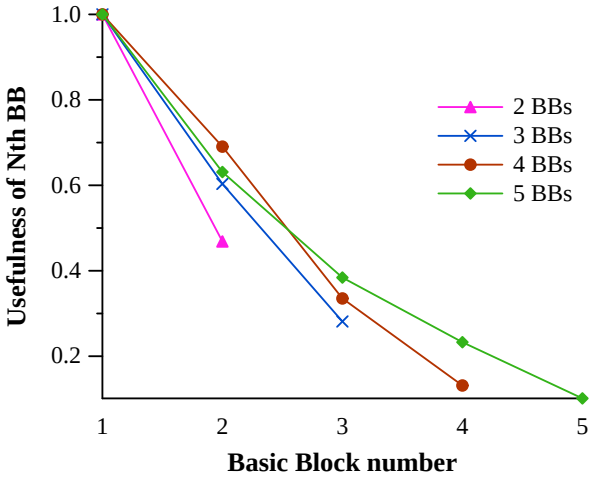


s95-li-train

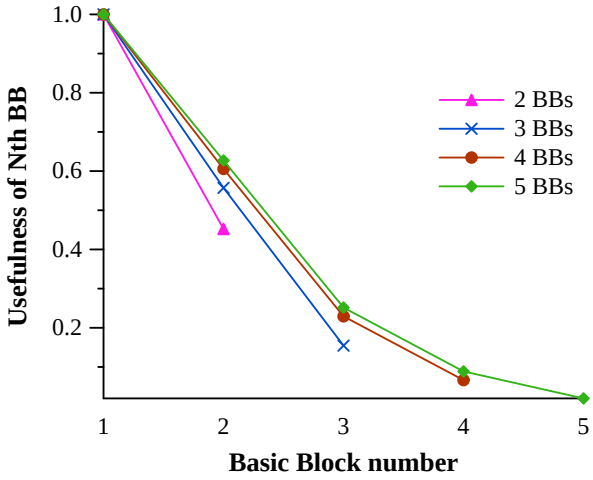


s95-m88ksim-test

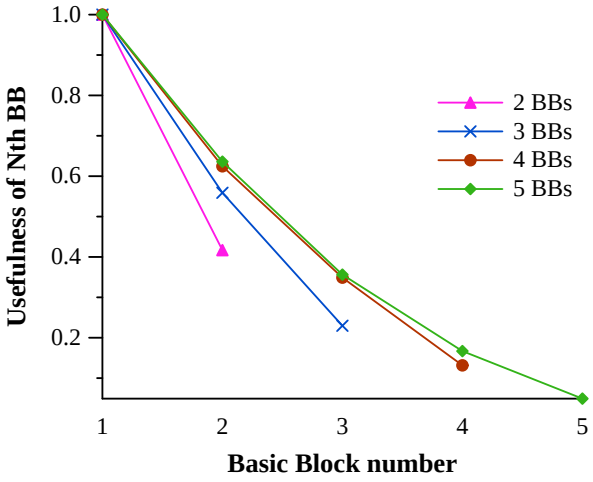
The vertical axes represent the fraction of the times that a given Nth basic block is used, given that the trace cache line is accessed. Since the first block is always accessed when there is a hit in the trace cache, the values for the first block is 1. For each configuration, these graphs tell how many times the execution reached a particular block.



minimum-spanning-tree



voronoi-tri



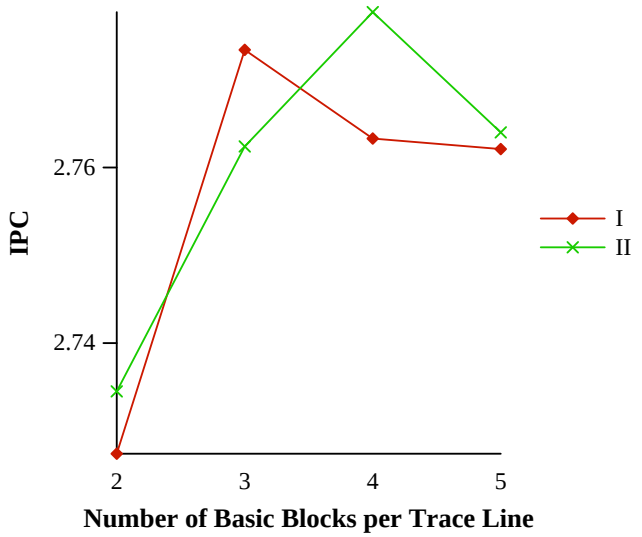
s95-perl-train

The figures show that the fifth block was used less that 10% of the time, and the fourth block 10 to 20%. The utilization increases for the third block to around 30 or 40%, suggesting a number of blocks of not more than 3 would be reasonable. One has to be careful in evaluating the results, since having less blocks does not mean there are less instructions to be found in the trace cache. Rather, for an equal amount of memory, the overall utilization of the trace cache would be higher after the initial cold misses since there is a high likelyhood of finding the block where the execution falls out of the trace line.

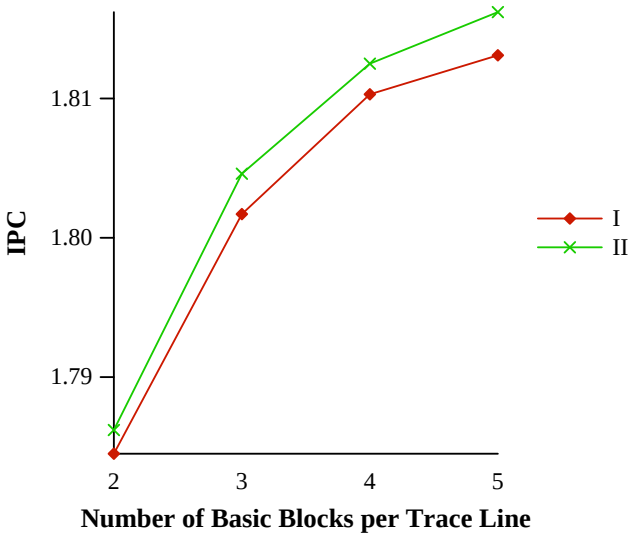
2.3.2 Impact on IPC of the number of basic blocks in the trace line

To quantitatively determine the impact on performance of the previous results, the IPC was measured for each case. The results are shown in the following set of figures.

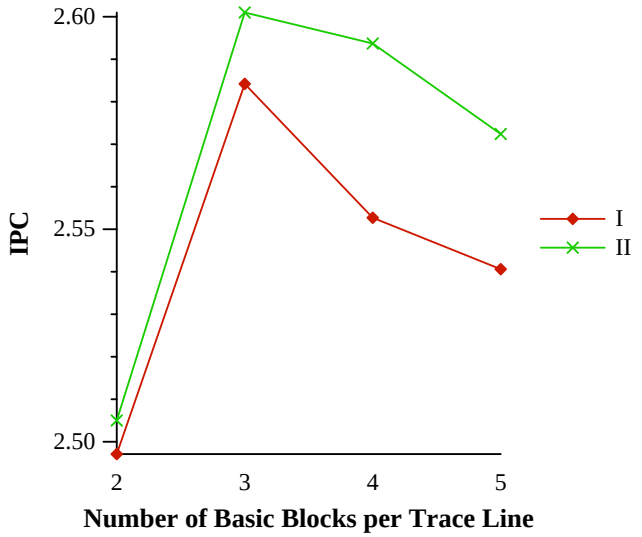
Results show a trend of improvement from 2 blocks to 3, but there are a number of benchmarks which display a degradation after 3 blocks per line. One possible explanations is the decreased level of utilization of the line due to truncation. These results suggest that 3 blocks per line is the best general choice for our line size of 20 instructions.



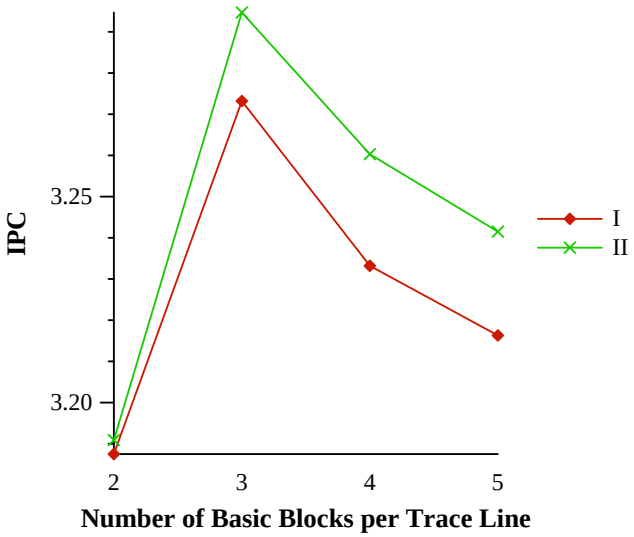
s95-compress-small



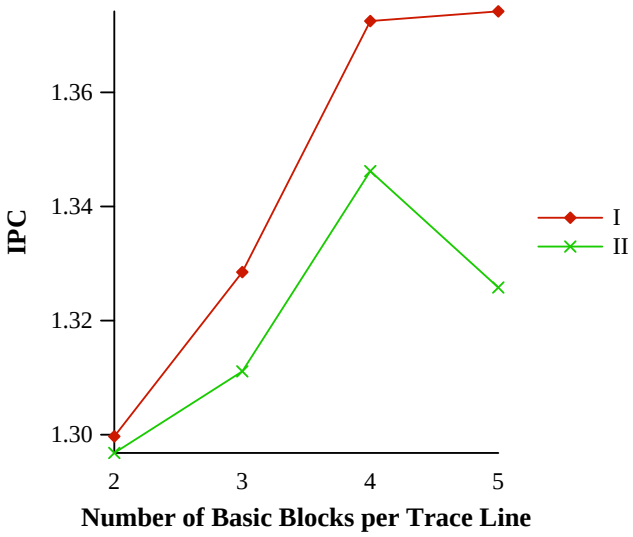
s95-gcc



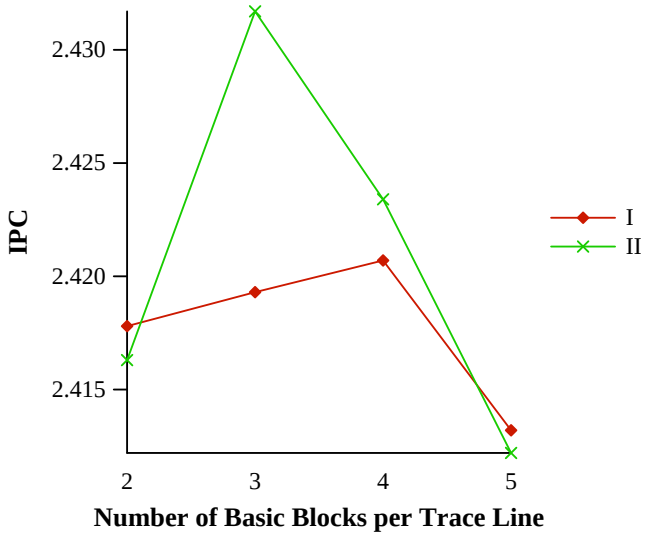
s95-li-train



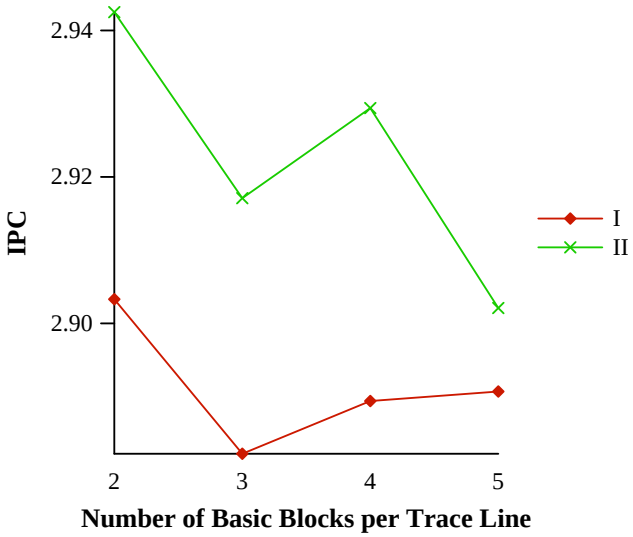
s95-m88ksim-test



minimum-spanning-tree



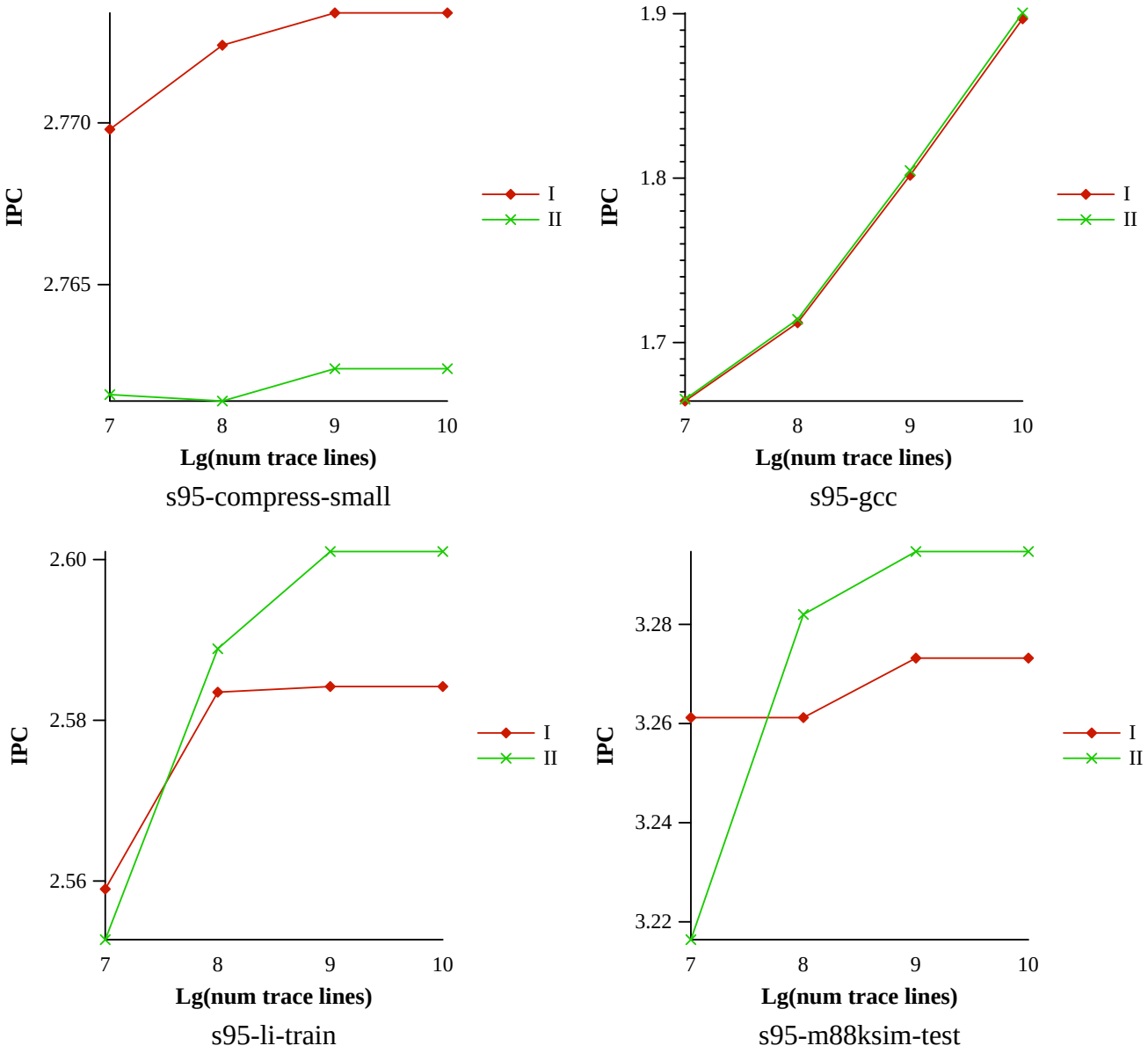
voronoi-tri



s95-perl-train

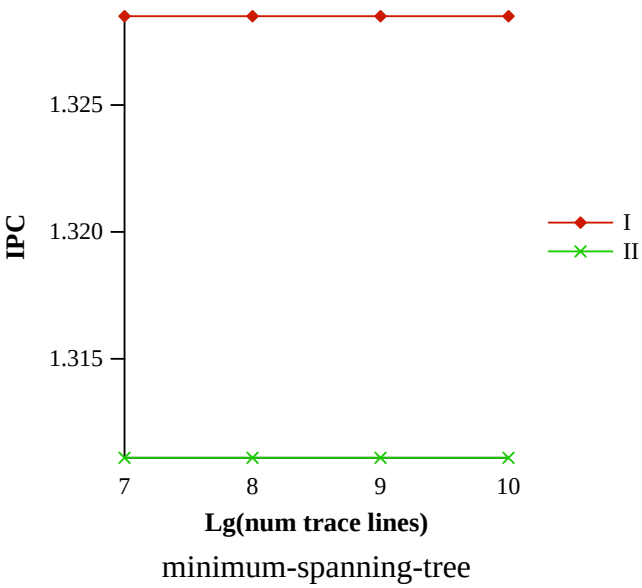
2.3.3 Impact on IPC of the number of trace lines

Equipped with information on the number of basic blocks, we then investigated the impact of the number of cache lines on the IPC. The simulations were run with 3 basic blocks, with varying number of trace cache lines.

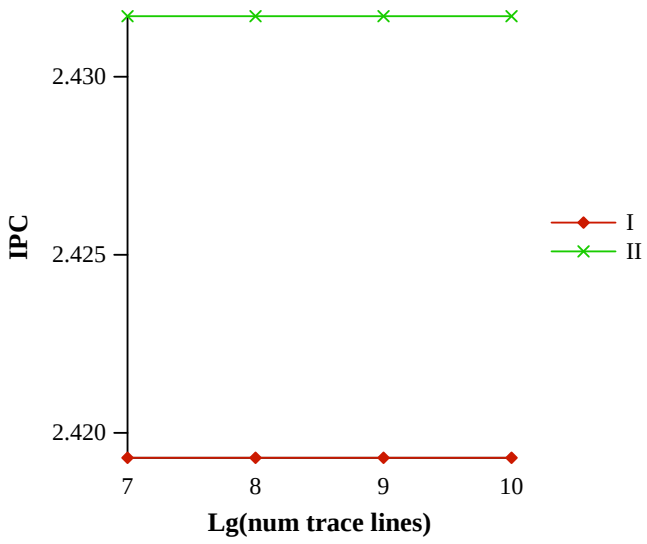


The figures show the results. The horizontal axis is the number of trace cache lines in base 2 logarithmic scale. Two different methods of filling the trace cache were used, and these are shown as seperate plots. In method I, only one trace is entered into the trace cache in each pass, whereas in method II, each block starts off a new trace cache line to be entered into the trace cache. Method II

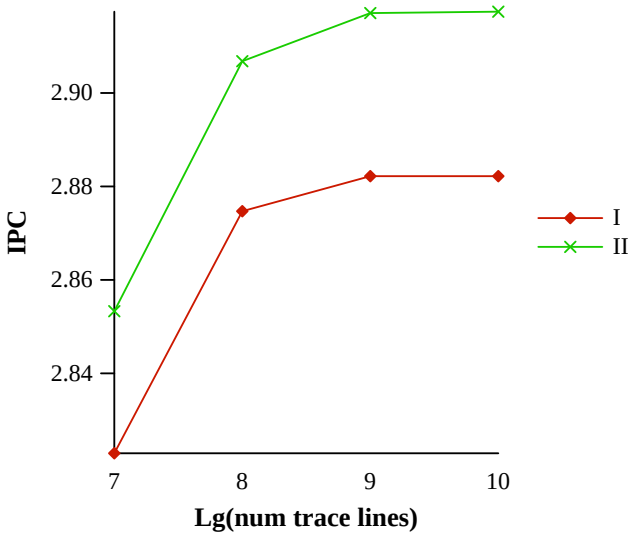
is similar to the approach described in [1].



minimum-spanning-tree



voronoi-tri



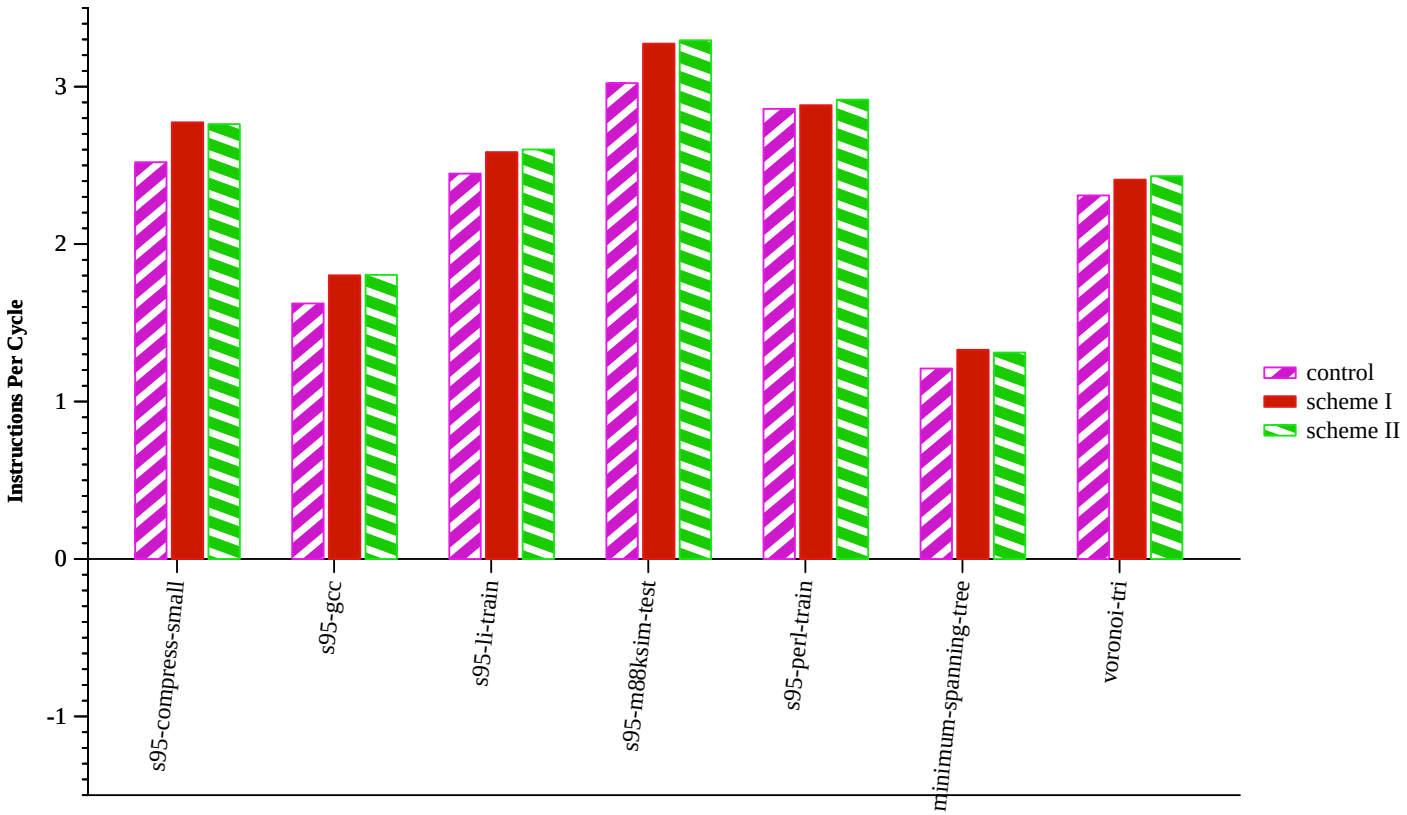
s95-perl-train

The performance profile varies greatly among the benchmarks. This is probably due to the difference in the size of the working sets, and for compress-small, voronoi and minimum spanning tree the working set is clearly smaller than the minimum size of 128 trace lines. Since the performance differs with the application it is difficult to ascertain a 'correct' number of lines, and tradeoffs with other performance enhancing schemes would have to be considered.

2.3.4 Comparison of filling method I and II

This figure compares the two methods of filling the trace cache, as explained previously. The control is the ordinary I-cache baseline architecture described in section 2. The degree of improvement over the control is small. The probable reason for the discrepancy with the reasonable improvements obtained in [4] is the difference in the issue widths used. For our simulations, a width of 8 was used as opposed to 16 in [4]. With fewer instructions issued per cycle, the benefit of trace caches are reduced considerably.

Given this small difference in performance, it is difficult to obtain detailed information on the differences. The main difference in the two filling methods is in the "learning time" of the cache. The first scheme possibly requires multiple passes of the trace to fill the trace cache with the traces that start with each basic block that occur in the execution trace, whereas the second scheme only needs a single pass. This added learning time would incur more cold-start misses on the trace cache. However, with many passes, the two would tend to converge, and the difference between the two would become small. One advantage of the first method, is more efficient usage of trace lines, which would be advantageous for cases where there is a large amount of non-frequent traces that pollute the trace cache.



Comparison of IPC across benchmarks

Chapter 3

Trace Cache with non-predicted path address table and prefetch buffer

3.1 Concept

Ideally we would like to feed instructions solely from the trace cache. But the trace cache has limited capacity, and its contents depend on the pattern of the last instruction. If the branches taken are not very biased, the execution will take different paths different times, the trace entries will change frequently. To solve this problem, one can have multiple trace entries per address, and use trace prediction to choose one of those entries. [3] Another way to solve this problem is to have instructions from both sides of the execution path. [2] However, storing instructions from multiple paths in the trace cache would be undesirable, as problems similar to those that arise in I-caches, such as alignment will become an issue.

In our experiment, we have a not-predicted path prefetch buffer. Associated with each entry in the trace cache, there was an entry in a separate table which could store as many addresses as basic blocks in the trace. Each trace ends with a control instruction. Instructions from the previously taken path are stored in the trace itself. The address for the other path was stored in this buffer. Whenever the trace cache was accessed, the prefetch buffer prefetches instructions from the I-cache for the paths that were not in the trace execution. These addresses are obtained from the table. During execution, if the path in the trace and the branch predictor outcome did not match, the trace was abandoned. Then the prefetch buffer was checked for its contents. On a hit, the instructions were fetched from this buffer. In case of a miss, the table is updated and this new address is added.

3.2 Simulation

For our simulations, we chose a baseline architecture in which once trace was added to a trace cache, it was only evicted due to a capacity miss. This allowed the table to make a stable copy of the alternate

paths. For our modified scheme, we ran two simulations. In the first scheme, it took one event to learn the alternate path and in the second scheme (oracle) the alternate paths were always known. The latency to the I-Cache was 1 cycle, while there was no penalty in accessing prefetch buffer. The prefetch buffer could hold eight instructions for each path.

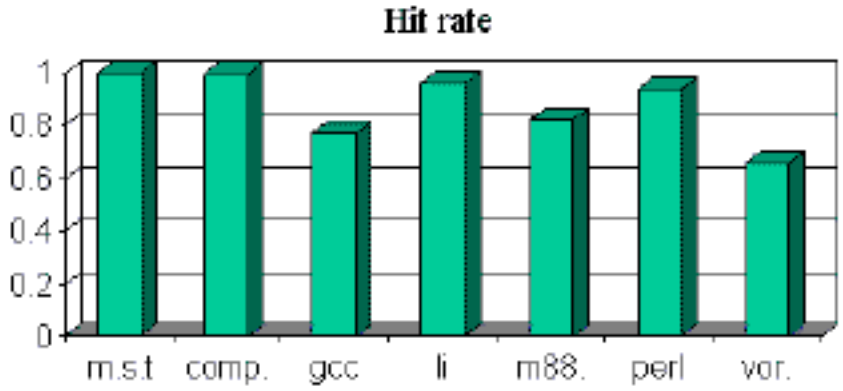
We modelled our prefetch buffer in the simulation environment by making two changes on the baseline architecture. First, whenever the execution did not follow the trace, the instructions were fetched from the prefetch buffer in the same cycle. Apart from that, if there was miss in the I-Cache for the instruction fetched by the prefetch buffer, the penalty would be less by the number of cycles prefetch buffer was ahead of simulation. Since we were not sure of this number, we chose it to be two for first scheme. For the oracle, we assumed that the prefetch buffer removed all the memory latencies.

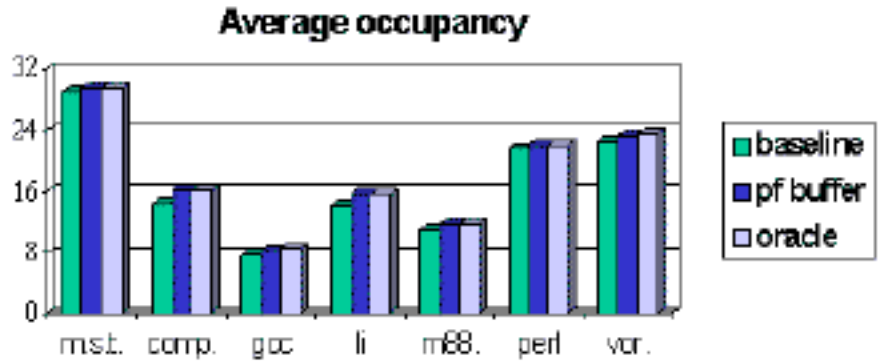
3.3 Results

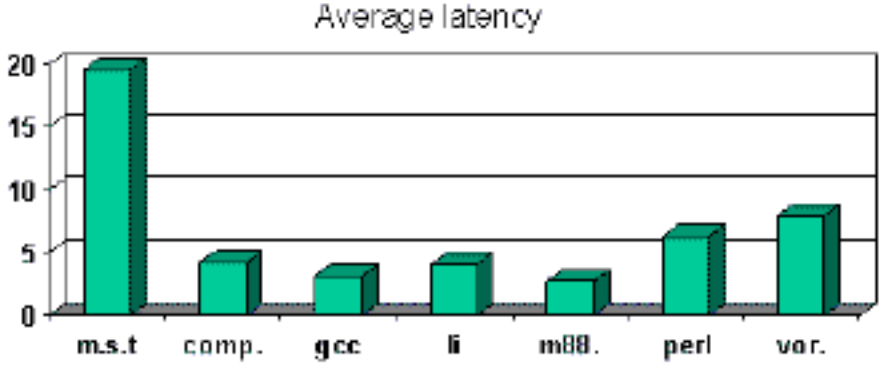
The hit rate, which is fraction of times a required line was found out when prefetched buffer was checked for its contents, the address table ranges from 65% to 100%, with most benchmarks showing more than 80% hit rate. This means that the table is effective in storing the address of the not taken path for all the branches.

The improvement in performance resulting from the proposed were very small for our simulations. The IPC remains almost unchanged for all of the three simulations.

The absence of improvement can be attributed to a number of reasons. One reason may be that the line stored in the prefetch buffer may include branch conditions itself, and may not be getting fully utilized. More important issue is of the baseline architecture used in the experiment. Statistics show that the average instruction queue latency is 2.62(m88ksim) to 19.55(minimum-spanning-tree) cycles. This means that a fetched instruction remains in the instruction fetch queue for long before getting executed. Also, average occupancy of the instruction fetch ranges from 8.26 (gcc) to 29.5 (minimum-spanning-tree), which is higher than the abndwith supported by the decode stage. This shows that the bottleneck was not the frontend, and hence any improvement in fetch bandwidth would not necessarily translate into IPC. More experiments need to be done to determine the effectiveness of the scheme. A more rigorous experiment would use a cycle level simulator, which models the complex latency issues, but this was out of scope of our project. Also, frontend performance needs to be isolated to get the idea of real improvement from this scheme.


Hit rate
1
0.8
0.6
0.4
0.2
0
m.s.t
comp.
gcc
li
m88.
perl
vor.


Average occupancy
32
24
16
8
0
m.s.t.
comp.
gcc
li
m88.
perl
vor.
baseline
pf buffer
oracle


Average latency
20
15
10
5
0
m.s.t
comp.
gcc
li
m88.
perl
vor.

Chapter 4

Concluding Remarks

Our simulations show that the degree of speedup resulting from the trace cache is quite small for typical processors without additional ways to improve the parallellism of the back end of the processor. Many experiments involving the front end typically use a trace-driven simulator to isolate away any performance degradation not attributable to the front end itself. This type of approach is desirable in that the performance changes are easily attributable to a given set of variables. However, in order to capture possible interactions in the entire system, and obtain 'real world numbers', it is also desirable to conduct experiments with non-ideal configurations. For these experiments, it is often difficult to find a range of parameter values that return a meaningful result, and many more passes are required to obtain a representative set of results.

More experiments are needed with different base configurations of the underlying basic structures to gain a thorough understanding of the issues involved. The following is a list of experiments that could be done in this respect.

- Performance gain over base vs. issue/decode width
- Impact of different branch prediction accuracy on the effectiveness of the trace cache

Bibliography

[1] A. Peleg, U. Weiser "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line", *US Patent number 5,381,533, Intel Corporation*, 1994.

[2] D. Friendly, S. Patel, and Y. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism", in Proceedings of the *30th International Symposium on Microarchitecture, Novemeber 1997.*

[3] Q. Jacobson, E. Rotenberg, and J. Smith, "Path-Based Next Trace Prediction," in *Proceedings of the 30th International Symposium on Microarchitecture*, Novemeber 1997.

[4] E. Rotenberg, S. Bennett, and J. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996