Instruction Pre-Processing in Trace Processors

Quinn Jacobson and James E. Smith

Department of Electrical & Computer Engineering
University of Wisconsin - Madison
{qjacobso, jes}@ece.wisc.edu

Copyright 1999 IEEE. Published in the Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5), January 9-13, 1999 in Orlando Florida. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact:

Manager, Copyrights and Permissions
IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331, USA
Telephone: + Intl. 908-562-3966

Instruction Pre-Processing in Trace Processors

Quinn Jacobson and James E. Smith

Department of Electrical & Computer Engineering
University of Wisconsin - Madison
{qjacobso, jes}@ece.wisc.edu

Abstract

In trace processors, a sequential program is partitioned at run time into "traces." A trace is an encapsulation of a dynamic sequence of instructions. A processor that uses traces as the unit of sequencing and execution achieves high instruction fetch rates and can support very wide-issue execution engines. We propose a new class of hardware optimizations that transform the instructions within traces to increase the performance of trace processors. Traces are "pre-processed" to optimize the instructions for execution together. We propose three specific optimizations: instruction scheduling, constant propagation, and instruction collapsing. Together, these optimizations offer substantial performance benefit, increasing performance by up to 24%.

1. Introduction

Trace Processors [11] are based on a microarchitecture that encapsulates dynamic instruction sequences into *traces* at run time. Each trace contains on the order of 16 dynamic instructions and commonly spans multiple basic blocks. The traces are cached [10][8] and become the fundamental unit of work for the rest of the processor. A fetch unit predicts the sequence of traces [4] and dispatches them as units to distributed processing elements. Each processing element is a modest superscalar execution engine capable of a high clock rate.

Because traces are cached, fetched, and executed as a unit, they provide a new opportunity for hardware-based optimization – at the trace-level. That is, a trace of instructions can be optimized via a "pre-process" phase prior to being placed in the trace cache.

There are a number of inherent characteristics of traces that enable pre-processing optimizations. First, each trace sequence contains basic blocks within a specific context, and optimizations applied across basic block boundaries must be valid only for that specific context. Second, the overhead of relatively complex optimizations can be amortized over many uses of a trace. Finally, pre-processing can target new, internal instructions which are not supported in the external instruction set.

In this paper we examine a number of trace optimizations. These optimizations have been chosen to exhibit the diversity of optimizations possible, and the specific ones we study improve performance by 4% to 24% across the benchmarks we studied.

1.1. Trace-Level Optimizations

The following examples illustrate the variety of optimizations that are possible. In later sections, we will discuss specific implementations and performance improvements for each.

- **Scheduling instructions** to take into account extra latency for inter-trace communication.
- **Performing constant propagation** that can not be implemented at compile time, by taking advantage of a single dynamic path and a powerful internal instruction set.
- **Collapsing data dependent chains of computation to single instructions** by adding new internal instructions that make use of very powerful execution units.

Note that pre-processing is intended to complement, not substitute for, good optimizing compilers. The dynamic nature of traces and implementation dependence allows additional transformations that would be impossible, or at least very difficult, in compile-time optimizations.

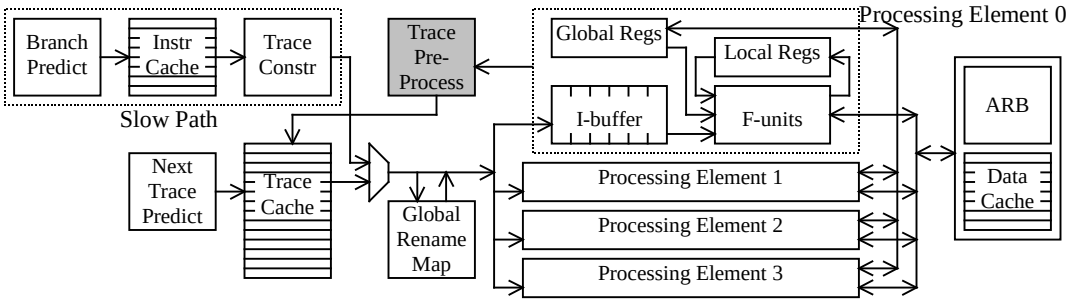

Figure 1 Trace processor hardware with pre-processing.

1.2. Trace Processor Overview

The main components of the trace processor frontend are the next-trace predictor and the trace cache (see Figure 1). Next-trace prediction [4] implicitly performs branch prediction and branch target prediction with sufficient bandwidth to take advantage of the potentially high fetch rate of the trace cache. During normal operation, the next-trace predictor and the trace cache provide a stream of instructions to the processor's execution engine. When the trace cache can not provide the needed instructions, the *slow path* is used.

We model a trace processor with a distributed backend, based on the design proposed in [11]. It is composed of 8 processing elements (PEs) each with a register file, instruction window and execution units. In trace processors, the unit of work dispatched to each processing unit is a trace. Each processing element has full bypasses internally and can support back-to-back dependent operations. It takes a full cycle for register values to be broadcast to other processing elements. Synchronizing of dependences through memory is enforced by special hardware [2] in the memory subsystem. We model a 2-cycle, 4 ported, 64Kbyte level-one data cache and a 10-cycle level-two data cache.

The maximum trace length is 16 instructions. Traces are forced to end at all jump indirect or return instructions. The trace cache has 2K entries and there is a 64Kbyte instruction cache.

1.3. Previous Work

This work is similar to recent work done independently by Friendly et. al. [3]. It builds on the work in trace caches [8][10] and trace processors [11][15].

The optimizations of scheduling and constant propagation are well studied in the context of compilers [1]. Dynamically scheduling groups of instructions was also studied in [7]. The optimization of data collapsing builds on a large body of work [5][9][12][13][14] in this area.

2. Implementing Trace Pre-Processing

When the next trace of a program is not in the trace cache, the slow-path hardware constructs it. This newly constructed trace is dispatched to an idle processing element to be executed and is also placed in the trace cache (in non-optimized form). Most of the work of pre-processing is performed in parallel with the initial execution of the trace. When the trace is completed, the pre-processor performs the final transformations on the trace. We assume a two-cycle latency for this transformation. The new optimized trace is then inserted into the trace cache. Using this model, the latency of trace pre-processing is not on the critical path.

2.1. Instruction Scheduling

The first optimization considered is instruction scheduling. As a practical matter, we do not schedule by literally moving the instructions within the trace. Rather, we assign priorities to instructions in a trace and rely on out-of-order issue logic to complete the scheduling task. When an instruction is identified for increased priority, instructions it depends on must also be increased in priority.

There are many possible heuristics that could be incorporated into assigning priorities. We focus on trace-level communication. Instructions that use values generated in previous traces are decremented in priority and instructions that produce values used in other traces are incremented in priority (incrementing priority has precedence over decrementing). This is an optimization that can not be performed at compile time.

2.2. Constant Propagation

The second pre-processing optimization considered is constant propagation within traces. There are many cases where computation chains with no inputs are used to generate constant values. These can occur because of encoding limitations in the instruction set and because of

control dependences that affect the value being produced. With traces, both of these restrictions can be overcome.

Constant propagation leads to performance improvement in two ways. First, there are some instructions that do not need to be executed after constant propagation. Second, constant propagation reduces the length of dependence chains.

Constant propagation can be incorporated into the initial execution of a trace. This is performed by adding a bit to the local register file indicating whether a value was generated from constant operands only. Instructions propagate the bit to their destination if all their sources have the bit set. Instructions producing a constant value record the value and are marked to inhibit their execution in the future. Register source operands with the constant bit set are replaced by the constant value for future execution.

2.3. Instruction Collapsing

The third pre-processing optimization considered is instruction collapsing. Recently, there has been much discussion about the impact of the increase in wire delays relative to gate delay [6]. The flip side of the relative increase in wire delays is the relative decrease in gate delays. While much current research is motivated by the problems posed by long wire delays, there has been little work performed to investigate the opportunities posed by short gate delays. Developing more sophisticated execution resources can significantly reduce the latency of executing a program. With the relative decrease in the delays of gates, adding a few levels of logic to execution resources may not impact the cycle time.

More complex operations can be used to collapse small chains of dependent operations into a single operation, thus reducing the latency. In general, data collapsing turns a chain of dependent operations into a single operation with more operands.

Most of the benefits of collapsing can be obtained by adding a new operation described below. This operation still has two register source operands and a single register destination operand. This is very important as it does not change the demands on the register file and data bypassing configurations, both of which are communication intensive and are usually timing critical. The new operation consists of shifting each of the register source operands left by a small immediate amount (0 to 3), inverting either or both operands, and adding the two values to a third immediate value to produce the result. This operation is extremely powerful but adds only a couple of gate delays over the traditional two-operand add instruction.

After collapsing is performed, an instruction may become redundant, as all instructions that depend on its result have been replaced with a compound instruction that encompasses the redundant instruction. We investigate the performance impact of either executing or not executing these redundant instructions. Executing the redundant instruction simplifies maintaining precise state for recovering from control mispredictions or traps.

Dependence collapsing is implemented in two phases. The first phase is implemented during the initial execution of the trace. A small type field is added to the local register file specifying the type of the instruction that creates the register value. Dependent instructions detect whether they can be collapsed with producing instruction(s). In this way the candidates for collapsing are determined. The second phase takes dependent chains of instructions and forms a new compound instruction that captures the same functionality but has lower total latency.

3. Performance Results

Five of the SPECint95 benchmarks (compress, gcc, go, list and m88ksim) are used to study the benefits of incorporating pre-processing. Each benchmark is run for 100 million instructions. We study the benefit of incorporating each optimization in isolation as well as in conjunction with the other optimizations.

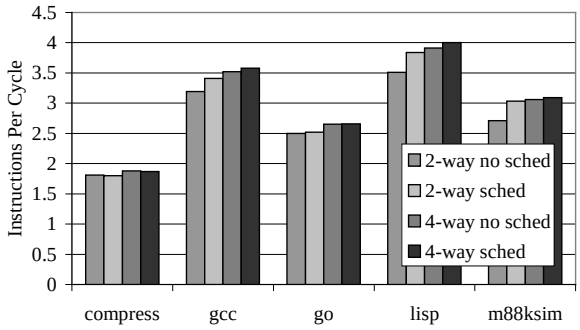

Figure 2 Performance gains from scheduling.

The performance benefit of instruction scheduling for trace-level communication is studied for two processor configurations (2-way issue per PE and 4-way issue per PE). Three of the five benchmarks see a notable performance gain from instruction scheduling in the case of the narrower issue configuration (see Figure 2). With narrower issue there is a greater chance that more instructions will be ready to issue then there are issue slots, so priorities can influence the order of execution. The benefit of scheduling is not significant with the wider issue processors. Scheduling becomes more important when applied in conjunction with collapsing.

Table 1 Instructions removed by constant propagation.

Benchmark	% of insns removed by constant propagation
Compress	4.8
Gcc	9.2
Go	12.0
Lisp	4.1
M88ksim	9.5

Constant propagation removes computation that need not be performed each time a trace is reused. A significant fraction of instructions can be optimized away (see Table 1). This reduces the demand for execution resources in the processor.

Although a significant fraction of instructions are removed by constant propagation the speedup is minimal, less than 1%. The reason is that with the large instruction window and accurate prediction, computation based only on constants can usually be computed before the values are needed. This is especially true in the trace processor configuration where execution resources are dedicated to traces. Constant propagation may have a larger impact if small pools of execution resources are shared by multiple processing elements.

Table 2 Breakdown of instructions effected by instruction collapsing.

Benchmark	% of instruction removed by collapsing			% of instructions replaced by collapsed instruction		
	Agr	Mod	Con	Agr	Mod	Con
Compress	28.5	27.8	0.0	16.4	14.4	14.4
Gcc	20.8	20.4	0.0	9.0	8.3	8.3
Go	21.2	20.6	0.0	10.4	9.9	9.9
Lisp	18.9	18.9	0.0	5.3	5.3	5.3
M88ksim	20.7	17.5	0.0	8.1	4.9	4.9

Instruction collapsing removes data dependence delays by replacing instructions with new compound instructions. We consider three policies for collapsing instructions. The first case, labeled "Aggressive," allows any dependent chain of logical operations (AND, OR,…) to be collapsed together as long as the total number of register operands is four or fewer. Also, any chain of additions and subtractions, including address calculations and set instructions, can be collapsed together as long as the total number of register operands is four or fewer. For chains of either logical or arithmetic operations, left shifts of immediate counts of 3 or fewer are allowed anywhere in the computation chain. Instructions no longer needed after collapsing is performed are not executed unless there is a branch mispredict or an exception within the trace.

The second case, labeled "Moderate," limits collapsed instructions to addition/subtraction of up to two register operands and one immediate operand. Either of the register operands can be shifted left by an immediate counts of 3 or fewer. This restriction limits the needed hardware support to a relatively simple arithmetic unit and requires no additional register ports or bypass logic. This new instruction can encode a number of possible chains of additions and left shifts. As before, instructions no longer needed after collapsing is performed are not executed unless there is a branch mispredict or an exception within the trace.

The third case, labeled "Conservative," is similar to the Moderate case except that instructions no longer needed after collapsing are still executed. This simplifies the logic for branch mispredict recovery and traps. This also simplifies the logic for performing the pre-processing, as it does not require the analysis of which instructions can be safely removed after collapsing.

Table 2 gives the breakdown of instructions effected by pre-processing. The first three columns present the fraction of instructions that do not need to be executed after collapsing for each of the collapsing policies. The next three columns present the fraction of instructions replaced by a new compound instruction.

A significant amount of collapsing takes place in all the benchmarks, with 5% to 16% of instructions being replaced by new compound instructions. The aggressive and moderate approaches, which remove unnecessary computation after collapsing, remove about 20% of instructions for all the benchmarks. The simpler approaches can capture a majority of the collapsing that the aggressive model can. Only in one benchmark, m88ksim, does the number of instructions replaced by compound instructions vary substantially between the aggressive and the simpler approaches. Even in this case, the simpler approaches get 60% of the collapsing of the aggressive approach.

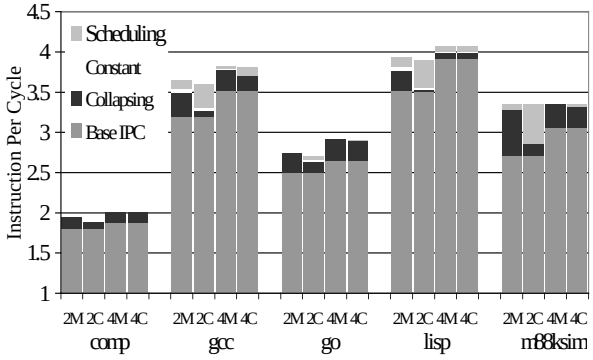

Figure 3 Performance gains from pre-processing
(2M = 2-way issue PEs and moderate collapsing, 2C = 2-way issue PEs and conservative collapsing, 4M = 4-way issue PEs and moderate collapsing, 4C = 4-way issue PEs and conservative collapsing).

Figure 3 gives the performance of the processor as successive pre-processing optimizations are added. Four processor configurations are studied, 2-way and 4-way issue per processor with both the conservative and moderate collapsing approaches. There is a notable benefit, up to a 20% increase in performance, by

incorporating the collapsing optimization. When constant propagation is incorporated in addition to collapsing, there is no significant increase in performance.

The scheduling optimization has a large effect when it is applied in conjunction with instruction collapsing. Instruction collapsing increases the amount of instruction level parallelism and therefore increases the chances of exceeding available issue bandwidth. Even with the wider issue processor, instruction scheduling is important. Instruction scheduling has a significantly larger effect with conservative instruction collapsing, where redundant instructions are not removed. With instruction scheduling, the conservative collapsing policy performs almost as well as the moderate collapsing policy.

The proposed pre-processing optimizations enable the processor to make the best utilization of available issue bandwidth. The narrower issue configuration with conservative collapsing and instruction scheduling optimizations perform as well as, or better than, the wider issue configurations without pre-processing. By better utilizing limited resources, intelligent pre-processing can significantly improve the price/performance of a processor.

4. Summary

We have proposed a new set of hardware optimizations for processors with trace caches. These optimizations take advantage of the intermediate program representation encoded in traces. The instructions within traces are pre-processed to optimize them for execution as a group. Three optimizations are studied: instruction scheduling, constant propagation and instruction collapsing. Together, these optimizations can increase performance by 4% to 24% across the benchmarks we studied.

The trace pre-processing optimizations allow for better utilization of execution resources. Trace pre-processing enables a given performance level to be achieved by a trace processor implementation with reduced total issue bandwidth. Trace pre-processing can therefore significantly improve not only the performance, but also the price/performance of a processor.

Acknowledgments

Quinn Jacobson is supported by a Graduate Fellowship from Intel. This work was supported in part by NSF Grant MIP-9505853 and the U.S. Army Intelligence Center and Fort Huachuca under Contract DAPT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the U.S. Army Intelligence Center and For Huachuca, or the U.S. Government.

References

[1] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Co., 1986.

[2] M. Franklin, G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Memory Disambiguation," *IEEE. Transactions on Computing*, pp. 552-571, Feb. 1996.

[3] D. Friendly, S. Patel, Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," To appear in the *Proceedings of the 31st International Symposium on Microarchitecture*, Nov. 1998.

[4] Q. Jacobson, E. Rotenberg, J. E. Smith, "Path-Based Next Trace Prediction," *Proc. of the 30th Int'l Symposium on Microarchitecture*, pp. 14-23, Dec. 1997.

[5] N. Malik, R. Eickemeyer, S. Vassiliadis, "Interlock Collapsing ALU for increased Instruction-Level Parallelism," in *Proceedings of the 25th International Symposium on Microarchitecture*, Sept 1992.

[6] D. Matzke, "Will Physical Scalability Sabotage Performance Gains," *IEEE Computer* Volume 30, Number 9, pp. 37-39, Sep. 1997.

[7] R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduling Groups," in *Proc. of the 24th Int'l Symposium on Computer Architecture*, pp. 13-25, June 1997.

[8] S. Patel, D. Friendly and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism." University of Michigan Technical Report CSE-TR-335-97, 1997.

[9] J. Phillips, S. Vassiliadis, "High Performance 3-1 interlock collapsing ALU's," *IEEE Transactions on Computers*, pp. 825-839, March 1994.

[10] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, Dec. 1996.

[11] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. E. Smith, "Trace Processors," *Proc. of the 30th Int'l Symposium on Microarchitecture*, pp. 138-148, Dec. 1997.

[12] Y. Sazeides, S. Vassiliadis, J. E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing," in *Proceedings of the 29th International Symposium on Microarchitecture*, Dec 1996.

[13] S. Vassiliadis, B. Blaner, R. Eickemeyer, "Scism: A Scalable, Compound Instruction Set Machine Architecture," *IBM Jounal of Research and Development*, pp. 59-78, Jan 1993.

[14] S. Vassiliadis, J. Phillips, B. Blaner, "Interlock Collapsing ALU's," *IEEE Transactions on Computers*, pp. 825-839, July 1993.

[15] S. Vijapeyam, T. Mitra, "Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences," Proc. *of the 24th Int'l Symposium on Computer Architecture*, pp. 1-12, June 1997.