Increasing Instruction-Level Parallelism with Instruction Precomputation

Joshua J. Yi, Resit Sendag, and David J. Lilja

Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota - Twin Cities
Minneapolis, MN 55455
{jjyi, rsgt, lilja}@ece.umn.edu

Abstract. Value reuse improves a processor's performance by dynamically caching the results of previous instructions and reusing those results to bypass the execution of future instructions that have the same opcode and input operands. However, continually replacing the least recently used entries could eventually fill the value reuse table with instructions that are not frequently executed. Furthermore, the complex hardware that replaces entries and updates the table may necessitate an increase in the clock period. We propose *instruction precomputation* to address these issues by profiling programs to determine the opcodes and input operands that have the highest frequencies of execution. These instructions then are loaded into the precomputation table before the program executes. During program execution, the precomputation table is used in the same way as the value reuse table is, with the exception that the precomputation table does not dynamically replace any entries. For a 2K-entry precomputation table implemented on a 4-way issue machine, this approach produced an average speedup of 11.0%. By comparison, a 2K-entry value reuse table produced an average speedup of 6.7%. Instruction precomputation outperforms value reuse, especially for smaller tables, with the same number of table entries while using less area and having a lower access time.

1 Introduction

A program may repeatedly perform the same computations during the course of its execution. For example, in a nested pair of FOR loops, an add instruction in the inner loop will repeatedly initialize and increment a loop induction variable. For each iteration of the outer loop, the computations performed by that add instruction are exactly identical. An optimizing compiler typically cannot remove these operations since the induction variable's initial value may change for each iteration.

Value reuse [3, 4] exploits this program characteristic by dynamically caching an instruction's opcode, input operands, and result into a value reuse table (VRT). For each instruction, the processor checks if its opcode and input operands match an entry in the VRT. If a match is found, then the processor can use the result stored in the VRT instead of re-executing the instruction.

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 481–485.

Since the processor constantly updates the VRT, a redundant computation could be stored in the VRT, evicted, re-executed, and re-stored. As a result, the VRT could hold redundant computations that have a very low frequency of execution, thus decreasing the effectiveness of this mechanism.

To address this frequency of execution issue, instruction precomputation uses profiling to determine the redundant computations with the highest frequencies of execution. The opcodes and input operands for these redundant computations are loaded into the precomputation table (PT) before the program executes. During program execution, the PT functions like a VRT, but with two key differences: 1) The PT stores only the highest frequency redundant computations, and 2) the PT does not replace or update any entries. As a result, this approach selectively targets those redundant computations that have the largest impact on the program's performance.

This paper makes the following contributions:

1. It shows that a large percentage of a program is spent repeatedly executing a handful of redundant computations.
2. It describes a novel approach of using profiling to improve the performance and decrease the cost (area, cycle time, and ports) of value reuse.

2 Instruction Precomputation

Instruction precomputation consists of two main steps: profiling and execution. The profiling step determines the redundant computations with the highest frequencies of execution. An instruction is a redundant computation if its opcode and input operands match a previously executed instruction's opcode and input operands.

After determining the highest frequency redundant computations, those redundant computations are loaded into the PT before the program executes. At run-time, the PT is checked to see if there is a match between a PT entry and the instruction's opcode and input operands. If a match is found, then the instruction's output is simply the value in the output field of the matching entry. As a result, that instruction can bypass the execute stage. If a match is not found, then the instruction continues through the pipeline as normal.

For instruction precomputation to be effective, the high frequency redundant computations have to account for a significant percentage of the program's instructions. To determine if this is the situation in typical programs, we profiled selected benchmarks from the SPEC 95 and SPEC 2000 benchmark suites using two different input sets ("A" and "B") [2]. For this paper, all benchmarks were compiled using the gcc compiler, version 2.6.3 at optimization level O3 and were run to completion.

To determine the amount of redundant computation, we stored each instruction's opcode and input operands (hereafter referred to as a "unique computation"). Any unique computation that has a frequency of execution greater than one is a redundant computation. After profiling each benchmark, the unique computations were sorted by their frequency of execution. Figure 1 shows the percentage of the total dynamic instructions that were accounted for by the top 2048 unique computations. (Only arithmetic instructions are shown here because they are the only instructions that we allowed into the PT.) As can be seen in Figure 1, the top 2048 arithmetic unique computations account for 14.7% to 44.5% (Input Set A) and 13.9% to 48.4% (B) of the total instructions executed by the program.

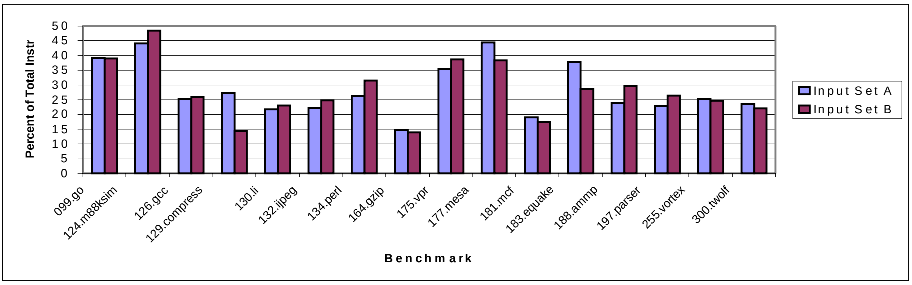

Fig 1. Percentage of the Total Dynamic Instructions Due to the Top 2048 Arithmetic Unique Computations

3 Results and Analysis

To determine the performance of instruction precomputation, we modified sim-outorder from the Simplescalar tool suite [1] to include a precomputation table. The PT can be accessed in both the dispatch and issue stages. In these two stages, the current instruction’s opcode and input operands are compared against the opcode and input operands that are stored in the PT. If a match is found in the dispatch stage, the instruction obtains its result from the PT and is removed from the pipeline (i.e. it waits only for in-order commit to complete its execution). If a match is found in the issue stage, the instruction obtains its result from the PT and is removed from the pipeline only if a free functional unit *cannot* be found. Otherwise, the instruction executes as normal.

The base machine was a 4-way issue processor with 2 integer and 2 floating-point ALUs; 1 integer and 1 floating-point multiply/divide unit; a 64 entry RUU; a 32 entry LSQ; and 2 memory ports. The L1 D and I caches were set to 32KB, 32B blocks, 2-way associativity, and a 1 hit cycle latency. The L2 cache was set to 256KB, 64B blocks, 4-way associativity, and a 12 cycle hit latency. The memory latency of the first block was 60 cycles while each following block took 5 cycles. The branch predictor was a combined predictor with 8K entries.

To reiterate one key point, the profiling step is used only to determine the highest frequency unique computations. Since it is extremely unlikely that the same input set that is used for profiling also will be used during execution, we simulate a combination of input sets, that is, we profile the benchmark using one input set, but run the benchmark with another input set (i.e. Profile A, Run B or Profile B, Run A).

Figure 2 shows the speedup of instruction precomputation as compared to the base machine for Profile B, Run A. We see that instruction precomputation improves the performance of all benchmarks by an average of 4.1%–11.0% (16 to 2048 entries). Similar results also occur for the Profile A, Run B combination. *These results show that the highest frequency unique computations are common across benchmarks and are not a function of the input set.*

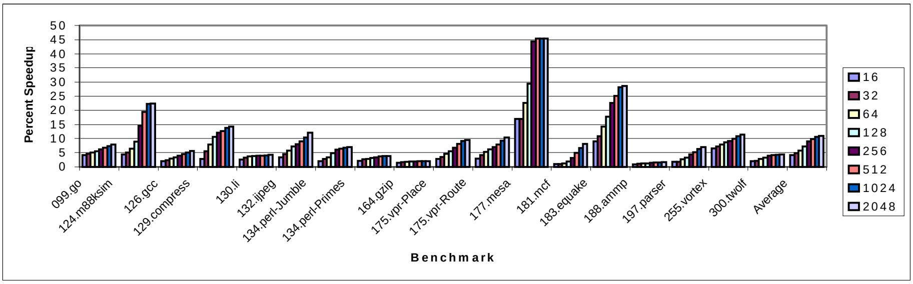

Fig 2. Percent Speedup Due To Instruction Precomputation for Various Table Sizes; Profile Input Set B, Run Input Set A

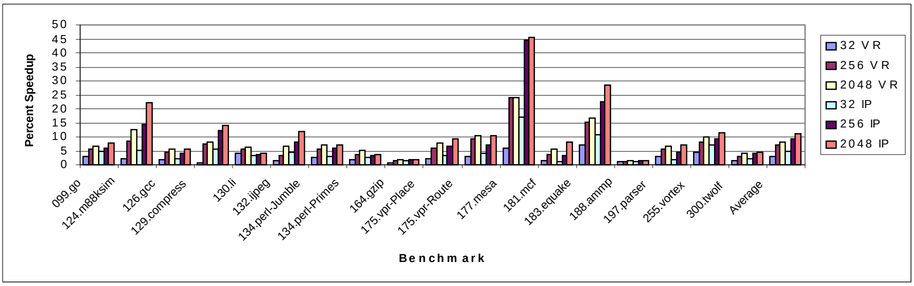

Fig 3: Speedup Comparison Between Value Reuse (VR) and Instruction Precomputation (IP) for Various Table Sizes; Profile Input Set A, Run Input Set B

In addition to having a lower area and access time, instruction precomputation also outperforms value reuse for tables of similar size. Figure 3 shows the speedup of instruction precomputation and value reuse, as compared to the base machine, for three different table sizes. For almost all table sizes and benchmarks, instruction precomputation yields a higher speedup than value reuse does. A more detailed comparison of instruction precomputation and value reuse can be found in [5].

4 Related Work

Sodani and Sohi [4] found speedups of 6% to 43% for a 1024 entry dynamic value reuse mechanism. While their speedups are comparable to those presented here, our approach has a smaller area footprint and a lower access time.

Molina *et. al.* [3] implemented a dynamic value reuse mechanism that exploited value reuse at the both the global (PC-independent) and local levels (PC-dependent). However, their approach is very area-intensive and their speedups are tied to the area used. For instance, for a realistic 36KB table size, the average speedup was 7%.

5 Conclusion

This paper presents a novel approach to value reuse that we call instruction precomputation. This approach uses profiling to determine the unique computations with the highest frequencies of execution. These unique computations are preloaded into the PT before the program begins execution. During execution, for each instruction, the opcode and input operands are compared to the opcodes and input operands in the PT. If there is a match, then the instruction is removed from the pipeline. For a 2048 entry PT, this approach produced an average speedup of 11.0%. Furthermore, the speedup for instruction precomputation is greater than the speedup for value reuse for almost all benchmarks and table sizes. Instruction precomputation also consumes less area and has a lower table access time as compared to value reuse.

Acknowledgements

This work was supported in by National Science Foundation grants EIA-9971666 and CCR-9900605, by IBM, and by the Minnesota Supercomputing Institute.

References

1. D. Burger and T. Austin; “The Simplescalar Tool Set, Version 2.0”; University of Wisconsin Computer Sciences Department Technical Report 1342.
2. A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja; "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research"; Workload Characterization of Emerging Computer Applications, L. Kurian John and A. M. Grizzaffi Maynard (eds.),Kluwer Academic Publishers, (2001) 83-100
3. C. Molina, A. Gonzalez, and J. Tubella; "Dynamic Removal of Redundant Computations"; International Conference on Supercomputing, (1999)
4. A. Sodani and G. Sohi; "Dynamic Instruction Reuse"; International Symposium on Computer Architecture, (1997)
5. J. Yi, R. Sendag, and D. Lilja; " Increasing Instruction-Level Parallelism with Instruction Precomputation "; University of Minnesota Technical Report: ARCTiC 02-01