Improving Processor Performance by Simplifying and Bypassing Trivial Computations

Joshua J. Yi and David J. Lilja
Department of Electrical and Computer Engineering
University of Minnesota - Twin Cities
{jjyi, lilja}@ece.umn.edu

Abstract

During the course of a program's execution, a processor performs many trivial computations; that is, computations that can be simplified or where the result is zero, one, or equal to one of the input operands. This paper shows that, despite compiling a program with aggressive optimizations (-O3), approximately 30% of all arithmetic instructions, which account for 12% of all dynamic instructions, are trivial computations. The amount of trivial computation is not heavily dependent on the program's specific input values. Our results show that eliminating trivial computations dynamically at run-time yields an average speedup of 8% for a typical processor. Even for a very aggressive processor (i.e. one with no functional unit constraints), the average speedup is still 6%. It also is important to note that the area cost (i.e. hardware) required to dynamically detect and eliminate these trivial computations is very low, consisting of only a few comparators and multiplexers.

1 Introduction

Many programs have a significant amount of trivial computation due to the way they are written and compiled. A trivial computation is an instruction whose output can be determined without having to perform the specified computation by either converting the operation to a less complex one or by determining the result immediately based on the value of one or both of the inputs. An example of the former is a multiply operation where one of the input operands has a value of two. In this case, the multiply instruction can be converted to a shift-left instruction. An example of the latter type is an add instruction where one of the input operands is zero. In this case, the result is the value of the other input operand.

While it seems as though an optimizing compiler should be able to remove many of these trivial computations, it is unable to do so unless the value of the input operands is known at compile time. Furthermore, the compiler may use trivial computations, such as 0 + 0, for initialization purposes. This paper shows that, due to these two factors, trivial computations can be a significant part of the program's overall execution time. Therefore, dynamically eliminating these trivial computations could reduce the program's execution time.

This paper makes the following contributions:

1. It quantifies the amount of trivial computation that is present in programs from the SPEC 95, SPEC 2000, and MediaBench [5] benchmark suites and shows that the amount of trivial computation is independent of the specific input values.
2. It determines the speedups that can be obtained by dynamically eliminating trivial computations.

The remainder of this paper is organized as follows: Section 2 quantifies the amount of trivial computation that exists in typical programs, Section 3 presents the speedup results achieved by eliminating these trivial computations, Section 4 describes some related work, and Section 5 summarizes our results and conclusions.

2 Types and Amounts of Trivial Computation

In this paper we identify two classes of trivial computations, those that can be bypassed and those that can be simplified. Table 1 shows the types of computations that are defined as trivial in this paper. The first column shows the type of operation while the second column shows how the result is normally computed. The third and fourth columns show which trivial computations can be bypassed and simplified, respectively.

Most of these trivial computations are straightforward with the exception of square root. For a square root, if the value of X is an even power of 2 (e.g. 4, 16, 64), then the result can be computed by halving the value in the exponent field. As a result, the exponent needs only to be shifted to the right by one bit. For example, the exponent for 16 is 0100. By applying this simplification, 0100 is right-shifted by 1 to produce 0010. Using this new exponent, the square root of 16 is then $1 * 2^2$, or 4.

We classify the computations in the fourth column as trivial because their operation can be simplified. While those trivial computations cannot be fully bypassed, they can use less complex, lower latency hardware instead.

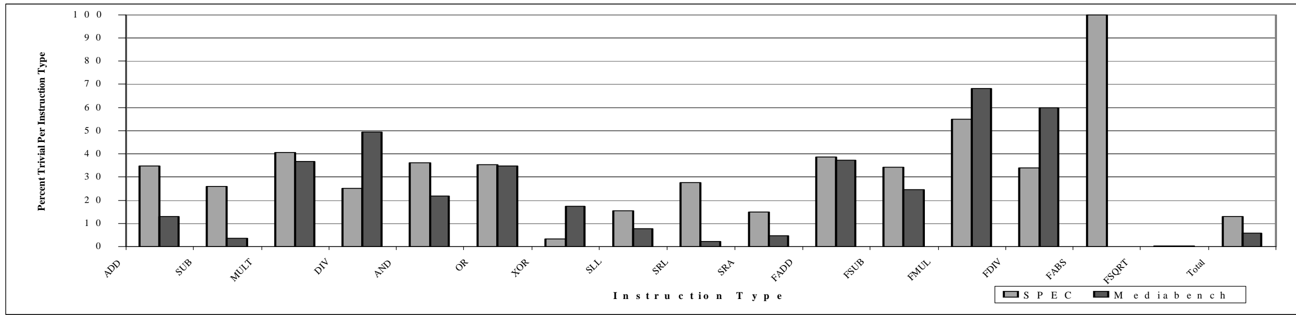

Figure 1. Percentage of trivial computations per instruction type and per total number of dynamic instructions for the SPEC and MediaBench benchmarks

Operation	Normal	Bypassable	Simplifiable
Add	X+Y	X,Y=0	
Subtract	X–Y	Y=0; X=Y	
Multiply	X*Y	X,Y=0	X,Y= Power of 2
Divide	X÷Y	X=0; X=Y	Y=Power of 2
AND, OR, XOR	X&Y, X|Y, X⊕Y	X,Y= {0,0xffffffff}; X=Y	
Logical Shift	X<<Y, X>>Y	X,Y = 0	
Arithmetic Shift	X<<Y, X>>Y	X,Y= {0,0xffffffff}	
Absolute Value	|X|	X= {0, Positive}	
Square Root	$\sqrt{X}$	X=0	X=Even Power of 2

Table 1. Trivial computations

Figure 1 shows the amount of trivial computation that is present in the benchmark programs from the SPEC 95, SPEC 2000, and MediaBench benchmark suites that we used in this study. Each pair of results shows the percentage of trivial computations that are present for that instruction type. The “Total” bars show the percentage of the total instructions that are trivial computations, over all instruction types.

These results show that trivial computations account for 12.89% and 5.92% of the total instructions in the SPEC and MediaBench benchmarks, respectively.

Figure 1 shows that almost all instruction types have a significant percentage of trivial computations. However, a high percentage does not necessarily mean that those instructions will have a significant impact on the program’s overall execution time since they could account for a very small percentage of the total executed instructions. For example, nearly 100% of the absolute value instructions (FABS) are trivial, but they account for only 0.04% of the total instructions executed.

To determine whether the trivial computations are a result of the benchmark itself, or of the benchmark’s input set, we profiled the same benchmarks with another input set. The results from the second input set were very similar to the results from the first [11]. This result indicates that trivial computations are primarily due to the benchmark programs themselves and not due to the specific values of their inputs.

3 Simulation Results

3.1 Benchmarks and Processor Configuration

The results in this section show the speedups that can be obtained by bypassing or simplifying trivial computations. These results are based on simulations performed by using a modified version of the sim-outorder superscalar processor simulator from the SimpleScalar tool suite [1].

All of the benchmarks were compiled at optimization level -O3 using the SimpleScalar version of gcc. To control the execution time, reduced input sets were used for some of the SPEC 2000 benchmarks. Benchmarks that use a reduced input set exhibit behavior similar to when the benchmark is executed using the reference input [4].

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. These parameter values are similar to those found in the Alpha 21264 [3] and the MIPS R10000 [10].

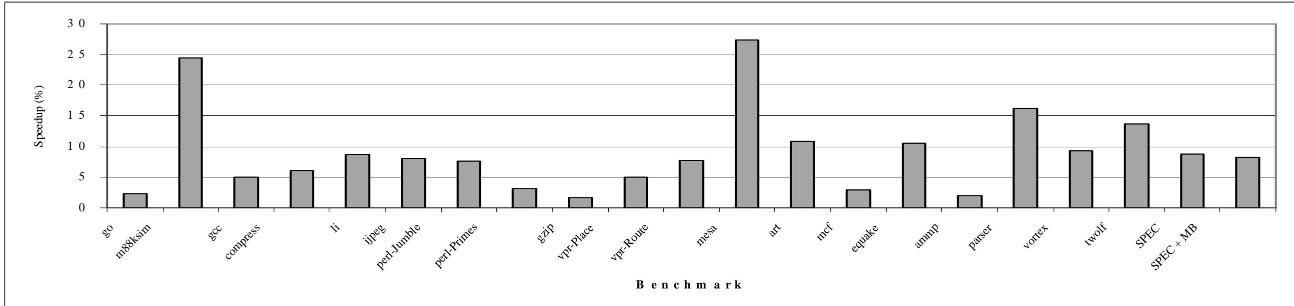

Figure 2. Speedup due to trivial computation bypass/simplification for the SPEC benchmarks

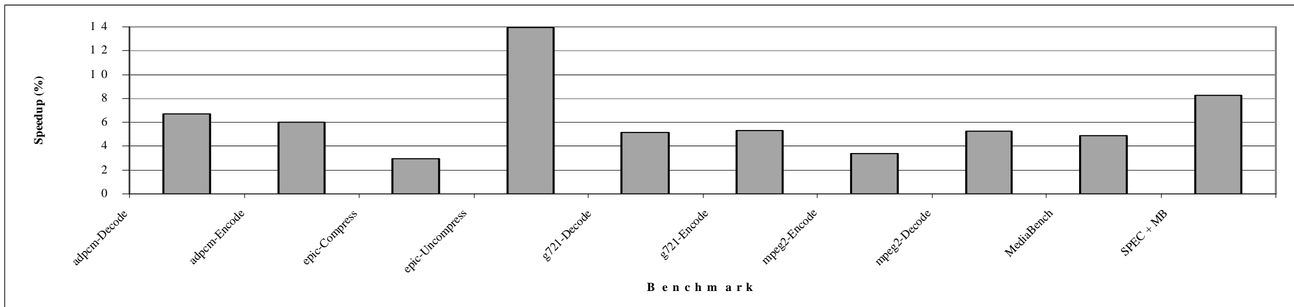

Figure 3. Speedup due to trivial computation bypass/simplification for the MediaBench benchmarks

The latencies for the integer functional units were 1, 3, and 19 cycles for the integer ALU, multiplier, and divider, respectively, while the latencies for the floating-point units were 2, 4, 12, and 24 cycles for the floating-point ALU, multiplier, divider, and square root unit, respectively.

3.2 Discussion and Analysis

The key point that separates this technique from other microarchitectural mechanisms is that in this technique only a single input operand (the trivial one) needs to be available for a *non-speculative* result to be generated. For example, in X * 0, the result can be computed non-speculatively as soon as 0 is available. This key point obviously has an important performance impact in that it allows the instruction to "execute" sooner than would normally be possible. Therefore, the speedups obtained from this technique are due to earlier scheduling of instructions, decreasing the number of resource conflicts, and reducing the latency of trivial computations.

Figures 2 and 3 show the speedups for the SPEC and MediaBench benchmarks, respectively. The speedup ranges from 1.64% (gzip) to 27.36% (mesa), with an execution time-weighted average of 8.84% for the SPEC benchmarks. For the MediaBench benchmarks, the speedup ranges from 2.97% (epic-Compress) to 13.97% (epic-Uncompress), with an average of 4.86%. The average speedup across all benchmarks is 8.22%. These results show that bypassing and simplifying trivial computations can produce significant speedups.

To determine the effect of the functional unit availability on the speedup, we varied the number of functional units. Due to space limitations, these results are not presented. However, even in the most unrealistic case in which the base processor has 4 of each type of functional unit, the speedup results are still quite good, with an average of 6.5% speedup for the SPEC benchmarks, 4.5% for the MediaBench benchmarks, and 6.2% overall [11]. This result demonstrates that the speedups shown in Figures 2 and 3 are not due primarily to the trivial computation elimination hardware acting like a pseudo-functional unit, but rather are due to the latency reduction and early instruction scheduling allowed by simplifying and bypassing the trivial computations.

4 Related Work

After extensive searches through several indexes, digital libraries, and the web, we found only a single publication directly on trivial computation [8]. In this paper, Richardson restricted the definition of trivial computations to certain multiplications (by 0, 1, and –1), divisions (X ÷ Y with X = {0, Y, -Y}), and square roots of 0 and 1. The two key differences between this previous work and our current study are the types of benchmarks that were used, and the scope of the definition of trivial computations. The first difference is that Richardson

studied only floating-point benchmarks (SPEC 92 and Perfect Club) while we studied a mix of integer, floating-point, and multimedia benchmarks. The second difference is that Richardson restricted the definition of trivial computations to the above 8 types while we defined the 26 types shown in Table 1. Not surprisingly, as a result of both differences, the average speedup of 2% that he reported was much lower than our 8% when comparing similar processor configurations. Richardson asserted that the lack of previous work on trivial computation was not due to its novelty, but due to a lack of knowledge as to how often trivial computations would occur.

While there has been a definite lack of published material on trivial computation, several papers have described the related technique of value reuse [2, 6, 7, 9]. With value reuse, an on-chip table dynamically caches the opcode, input operands, and result of previously executed instructions. For each instruction, the processor checks if the current instruction's opcode and input operands match a cached entry. If there is a match, the processor reuses the result that is stored in the table instead of re-executing the instruction, thus bypassing the execution of the current instruction.

There are several differences between value reuse and our approach of bypassing trivial computations. The first and biggest difference is that value reuse requires the use of an on-chip table. For example, Molina *et al.* [6] used a 221KB table to achieve an average speedup of 10%. In contrast, the trivial computation approach that we propose uses only a small amount of area (a few comparators and multiplexors). The second difference is that each instruction that is bypassed using value reuse had to have been previously executed at least once. With trivial computation, in contrast, the instruction can be bypassed the first time it is encountered. Finally, the last difference is that for value reuse, both input operands must be available since they are both needed to access the value reuse table. Trivial computations, on the other hand, can be bypassed when only a single input operand is available. For example, if X * 0 were a frequently occurring computation, value reuse would need to have both input operands available before the instruction can be bypassed while trivial computation would need only the second input operand (0) to be available.

5 Conclusion

This paper presents a dynamic method of detecting and eliminating trivial computations to improve processor performance. A trivial computation is a computation that can be converted into a faster and less complex one or can be bypassed completely by setting the output value to zero, one, or to the value of one of the input operands. This paper shows that for a set of benchmarks from the SPEC 95, SPEC 2000, and MediaBench benchmark suites, a significant percentage of the computations for each instruction type are trivial and that nearly 12% of the total dynamic instructions are trivial. The compiler, due to a lack of run-time information or for initialization reasons, cannot remove these trivial computations. Furthermore, this paper demonstrated that the trivial computations are mainly a function of the benchmark and not of the benchmark's input values. Finally, dynamically eliminating trivial computations, through simplification or bypass, produced an average speedup of 8.2% for a typical processor and an average speedup of 6.2% for a processor without any functional unit constraints.

Acknowledgements

The authors would like to thank Chris Hescott, Baris Kazar, and Keith Osowski for their helpful comments on previous drafts of this work.

This work was supported in part by National Science Foundation grants EIA-9971666 and CCR-9900605, by IBM, by Compaq's Alpha Development Group, 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] J. Huang and D. Lilja, "Exploiting Basic Block Locality with Block Reuse", *International Symposium on High Performance Computer Architecture*, 1999.

[3] R. Kessler, E. McLellan, and D. Webb, "The Alpha 21264 Microprocessor Architecture", *International Conference on Computer Design*, 1998.

[4] A. KleinOsowski and D. Lilja, " MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research", *Computer Architecture Letters*, Volume 1, June 2002.

[5] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *International Symposium on Microarchitecture*, 1997.

[6] C. Molina, A. Gonzalez, and J. Tubella, "Dynamic Removal of Redundant Computations", *International Conference on Supercomputing*, 1999.

[7] S. Oberman and M. Flynn, "On Division and Reciprocal Caches", Stanford University Technical Report CSL-TR-95-666, 1995.

[8] S. Richardson, "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation", *International Symposium on Computer Arithmetic*, 1993.

[9] A. Sodani and G. Sohi, "Dynamic Instruction Reuse", *International Symposium on Computer Architecture*, 1997.

[10] K. Yeager, "The MIPS R10000 Superscalar Microprocessor", *IEEE Micro*, Vol. 16, No. 2, March-April 1996, Pages 28-40.

[11] J. Yi and D. Lilja, "Improving Processor Performance by Simplifying and Bypassing Trivial Computations", University of Minnesota Technical Report: ARCTiC 02-06, 2002.