Improving Processor Performance by Simplifying and Bypassing Trivial Computations

Joshua J. Yi and David J. Lilja
Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota - Twin Cities
Minneapolis, MN 55455
{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 study shows that, despite compiling a program with aggressive optimizations (-O3), approximately 30% of all arithmetic instructions, which account for 12% of all instructions executed, are trivial computations. Furthermore, our results show that the amount of trivial computation is not heavily dependent on the program's specific input values. Since a significant percentage of all instructions are trivial computations, dynamically detecting and eliminating these trivial computations can greatly reduce the program's execution time. 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 first type of trivial computation 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. Therefore, the instruction has been converted from a complex, long latency multiply instruction to a simpler, shorter latency shift-left instruction. An example of the second type of trivial computation is an add instruction where one of the input operands is equal to zero. As a result, no computation actually needs to be performed for this instruction – the result is simply 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 often is unable to do so unless the value of the input operands is known at compile time. However, the only values that the compiler can determine at compile time are the values of constants. As a result, the compiler cannot remove trivial computations that are a function of the input set. Furthermore, the compiler may use trivial computations for initialization purposes. For example, to set the value of r1 to zero, the following instruction could be used: add r1, r0, r0, where r0 is a register whose value is always zero. This study shows that, due to these two factors, trivial computations can be a significant part of the program's overall execution time. Therefore, dynamically detecting and eliminating these trivial computations can reduce the program's execution time.

This paper makes the following contributions:

1. It identifies the different types of trivial computations that remain in programs after aggressive compiler optimization.
2. It quantifies the amount of trivial computation that is present in programs from the SPEC 95, SPEC 2000, and MediaBench [Lee97] benchmark suites and shows that the amount of trivial computation is not heavily dependent on the program's specific input values.
3. It determines the speedups that can be obtained by dynamically detecting and eliminating trivial computations in several different processor models.

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 speedups results achieved by detecting and 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 study we identify two classes of trivial computations, those that can be bypassed and those that can be simplified, but not bypassed. Table 1 shows the types of computations that are defined as trivial in this study. 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. Note that the add, subtract, multiply, and divide instructions include both integer and floating-point data types while the absolute value instruction operates on only floating-point values.

Operation	Representation	"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 Profiled and Bypassed or Simplified in this Study

Most of these trivial computations are straightforward with the possible exception of the computation for square root. For example, for an add instruction, if either input operand is equal to 0, then the result is equal to the value of the other input operand. Similarly, for an AND instruction, if both input operands have the same value, then the result is equal to the value of either input operand. 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 the 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 result is then $1 * 2^2$, which corresponds to the correct result value of 4.

We classify the computations in the fourth column as trivial because their operation can be simplified. While those trivial computations cannot be bypassed entirely, as the trivial computations in the third column can be, they can use less complex hardware, such as shifters, to compute the correct result. Note that integer and floating-point instructions of the same type (e.g. multiply, divide, etc.) are handled differently due to the format of the number. For example, in the integer version of X * 2, the result is simply a left shift of X. Alternatively, the floating-point version adds one to the exponent of X to simplify its computation.

Figure 1 shows the amount of trivial computation that is present in the benchmark programs from the SPEC 95, SPEC 2000, and MediaBench suites that we used in this study. Each pair of results shows the percentage of trivial computations that are present for that instruction type. For example, 34.73% of all Integer ADD instructions (ADD) are trivial in the SPEC benchmarks while only 13.18% are trivial in the MediaBench benchmark suite. The pair of bars labeled "Total" shows the percentage of the total dynamic instructions that are trivial computations, over all instruction types.

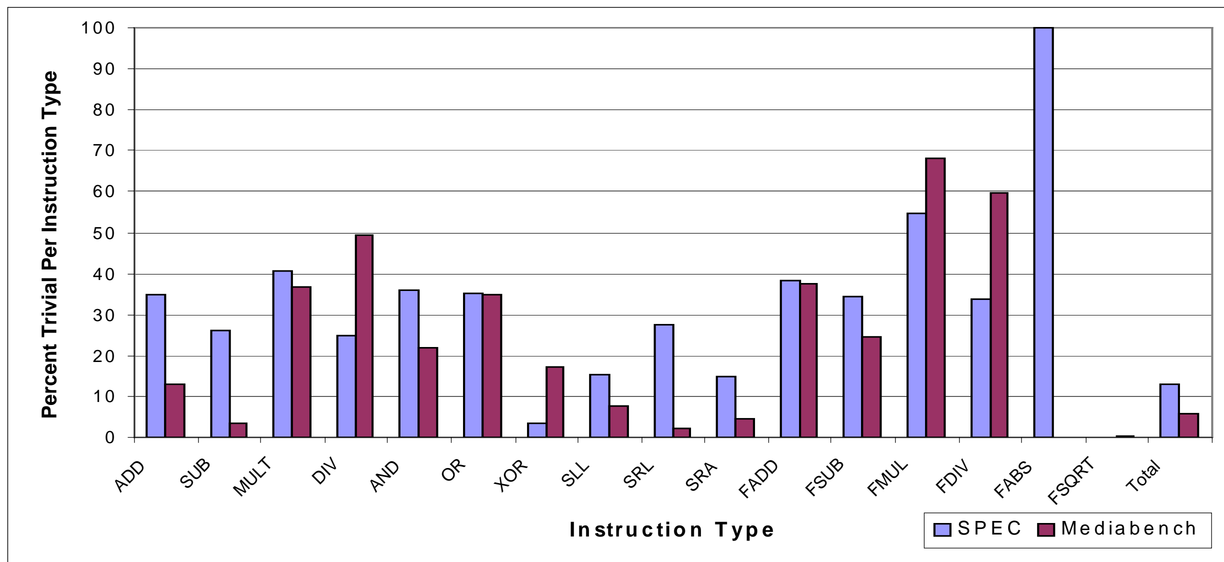

Figure 1: Percentage of Trivial Computations per Instruction Type and per Total Number of Dynamic Instructions for the SPEC and MediaBench Benchmarks

These results show that trivial computations account for 12.89% and 5.92% of the total dynamic instructions in the SPEC and MediaBench benchmarks (11.90% combined), respectively. The result for the MediaBench benchmarks surprised us. Due to the nature of the benchmarks, our initial expectation was that the MediaBench benchmarks would have a higher percentage of trivial computations than the SPEC benchmarks. However, as shown in Figure 1, this is not the case.

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 byproduct of the benchmark's input set or are a result of the benchmark itself, we profiled the same benchmarks with another input set. The results from the second input set, which are not shown here due to space limitations, were very similar to the results from the first. 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

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 [Burger97].

3.1 Methodology and Benchmarks

Table 2 lists the benchmark programs that were used in this study. All of these 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

Benchmark	Suite	Input Set
099.go	SPEC 95	Train
124.m88ksim	SPEC 95	Train
126.gcc	SPEC 95	Test
129.compress	SPEC 95	Train
130.li	SPEC 95	Train
132.ijpeg	SPEC 95	Test
134.perl	SPEC 95	Test
164.gzip	SPEC 2000	Reduced Small
175.vpr	SPEC 2000	Reduced Medium
177.mesa	SPEC 2000	Reduced Large
181.mcf	SPEC 2000	Reduced Medium
183.equake	SPEC 2000	Reduced Large
188.ammp	SPEC 2000	Reduced Medium
197.parser	SPEC 2000	Reduced Medium
255.vortex	SPEC 2000	Reduced Medium
300.twolf	SPEC 2000	Test
adpcm	MediaBench	clinton.pcm, clinton.adpcm
epic	MediaBench	test.image.pgm, test_image.pgm.E
g721	MediaBench	clinton.g721, clinton.pcm
mpeg2	MediaBench	mei16v2.m2v, rec

Table 2: Benchmarks Profiled and Tested in this Study

benchmarks. Benchmarks that use a reduced input set exhibit behavior similar to when the benchmark is executed using the reference input set [KleinOsowski00]. For 134.perl, 175.vpr, and the MediaBench benchmarks, each input set is composed of two separate inputs. To avoid losing any information about each benchmark's behavior with a different input, the results for the separate inputs are reported separately.

3.2 Processor Configuration and Models

Table 3 shows the base processor configuration used in this study while Table 4 shows the instruction execution latencies. The values shown in both tables are representative of the values that are used in commercially available processors such as the Alpha 21264 and the MIPS R10000 [Kessler98, Yeager96].

Parameters	Final Values
# of Integer ALUs	2
# of FP ALUs	2
# of Integer Multipliers/Dividers	1
# of Floating-Point Multipliers/Dividers	1
# of Instruction Fetch Queue Entries	32
Decode, Issue, Commit Width	4-Way
# of Reorder Buffer Entries	64
# of Load-Store Queue Entries	32
# of Memory Ports	2
L1 D-Cache Size, Associativity, Block Size, Replacement Policy, Latency	32KB, 2-Way, 32 Bytes, Least Recently Used, 1 cycle
L1 I-Cache Size, Associativity, Block Size, Replacement Policy, Latency	32KB, 4-Way, 32 Bytes, Least Recently Used, 1 cycle
L2 Unified Cache Size, Associativity, Block Size, Replacement Policy, Latency	256KB, 4-Way, 64 Bytes, Least Recently Used, 12 cycles
Memory Latency (First, Following Blocks)	60 Cycles, 5 Cycles
Branch Predictor Type, Configuration	Combined Predictor, 8K Entries
Branch Misprediction Penalty	3 Cycles

Table 3: Base Processor Configuration

Functional Unit	Latency (Cycles)
Integer ALU	1
Integer Multiply	3
Integer Divide	19
Floating-Point ALU	2
Floating-Point Multiply	4
Floating-Point Divide	12
Floating-Point Square Root	24

Table 4: Function Unit Latencies in Cycles

While we were able to find very little published work on this subject, we assumed that some commercial processors already eliminate trivial computations to some degree. (See Section 4 for a detailed description of previous work.) Therefore, to account for this possibility, we defined the following three processor models.

In the "aggressive" processor model, trivial computations that can be bypassed are eliminated in either the issue stage or the execute stage, depending on when the input operand(s) are available. Trivial computations that can be simplified continue through the pipeline to execute on a different (i.e. lower latency) functional unit. The "realistic" machine is the same as the aggressive machine with the exception that trivial multiplies and divides can be eliminated only in the issue stage (i.e. they cannot be eliminated in the execute stage). In the "conservative" machine, all trivial computations can be eliminated only in the issue stage, with the exception of floating-point ALU instructions. These instructions can be eliminated in either the issue stage or the execute stage. Table 5 summarizes the pipeline stages in which bypassable and simplifiable trivial computations can be eliminated for each processor model. Of these three models, the realistic machine is probably the closest to what is currently implemented in existing commercial processors in terms of trivial computation elimination.

Trivial Computation	Issue Stage	Execute Stage
Bypassable Integer ALU	Aggressive Realistic Conservative	Aggressive Realistic
Bypassable Floating-Point ALU	Aggressive Realistic Conservative	Aggressive Realistic Conservative
Bypassable Multiply and Divide	Aggressive Realistic Conservative	Aggressive
Simplifiable Multiply and Divide	N/A	Aggressive
Bypassable Square Root	Aggressive Realistic Conservative	Aggressive Realistic Conservative
Simplifiable Square Root	N/A	Aggressive Realistic Conservative

Table 5: Summary of the Processor Models

The key implementation point, and the point that separates this technique from other microarchitectural mechanisms, such as value reuse (see Section 4), 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 the 0 value 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 in the pipeline

(i.e. the single input operand factor), decreasing the number of resource conflicts, and reducing the execution latency of trivial computations.

3.3 Discussion and Analysis

Figures 2 and 3 show the speedups for all three processor models for the SPEC and MediaBench benchmarks, respectively. The rightmost two sets of bars show the average

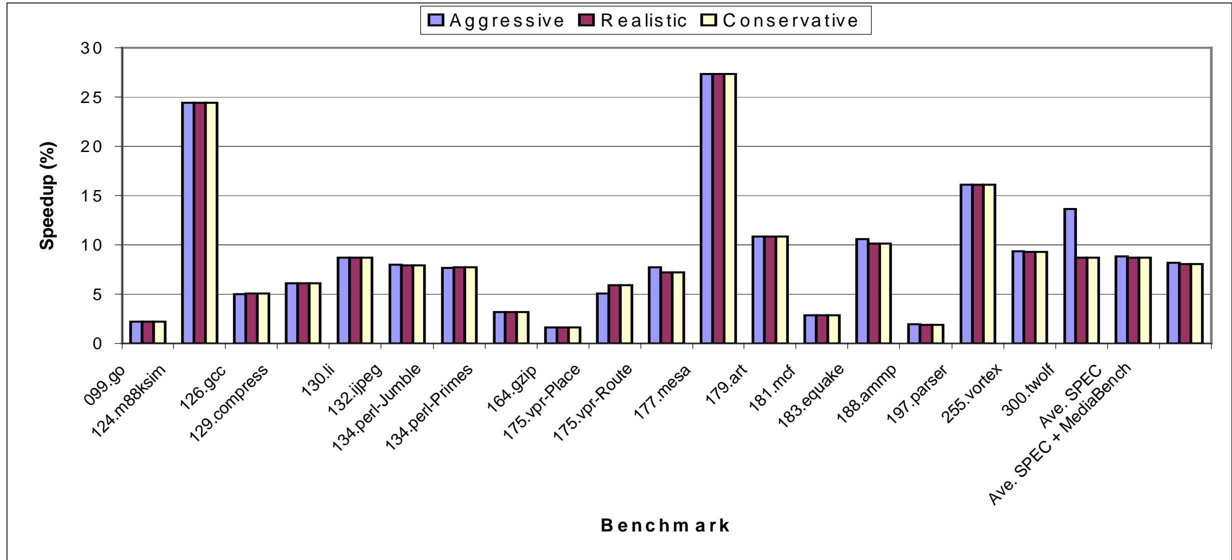

Figure 2: Speedup Due to Trivial Computation Bypass and Simplification for the SPEC Benchmarks

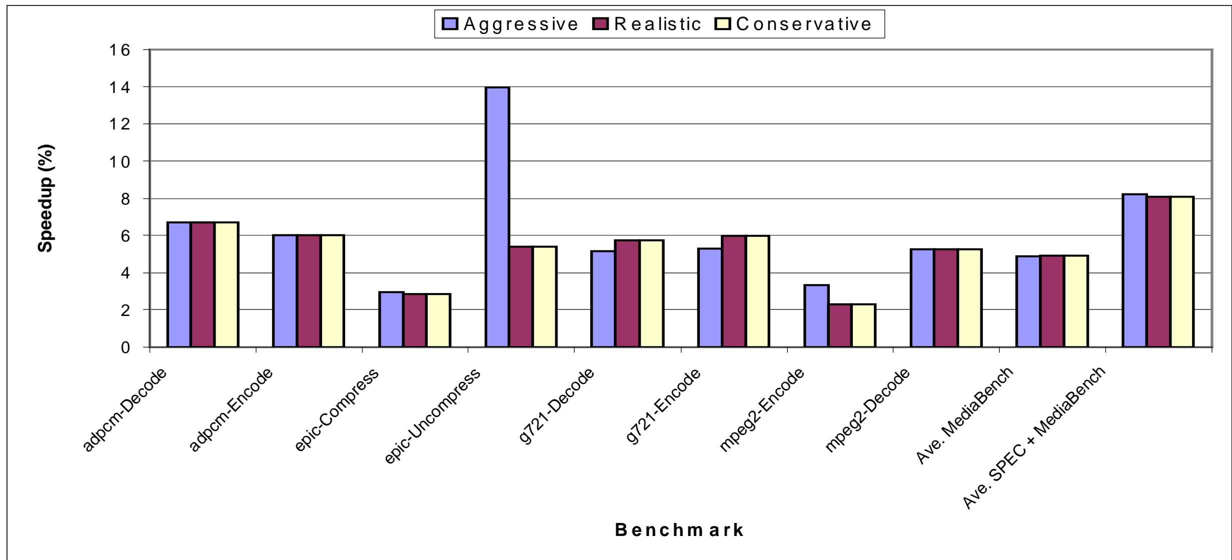

Figure 3: Speedup Due to Trivial Computation Bypass and Simplification for the MediaBench Benchmarks

speedups for each benchmark suite and the overall speedups (i.e. for both benchmark suites together).

The second key result is that in almost all of the benchmarks, the speedups for all three processor models are very similar. The only two exceptions are 300.twolf and epic-Uncompress. This is an extremely important result in that it proves that most of the performance improvement is *not* due to the trivial computations that commercial processors may already be eliminating. Since the differences in the speedups are the result of which trivial computations are eliminated in each processor model, not eliminating the trivial computations that commercial processors may be eliminating shows the impact of those trivial computations on the overall performance. Therefore, since the differences in the speedups between the processor models are small, and since the realistic processor model best represents commercial processors, the trivial computations that commercial processors may be eliminating have very little impact on the program's execution time. On the other hand, these results show that the trivial computations that account for most of the performance improvement are present in all processor models. Therefore, if commercial processors currently are eliminating various types of trivial computation, they probably are not targeting the trivial computations with the most impact.

For several benchmarks, the realistic processor model produces a slightly higher speedup than the aggressive processor model. While this result is counter-intuitive, the lower speedup for the aggressive processor model is due to increased Integer ALU contention. For simplifiable trivial computations, the shifter in an Integer ALU is used to reduce the complexity of the operation. Therefore, since the aggressive processor model targets the largest number of simplifiable trivial computations, this increased use of the Integer ALU's shifter results in greater contention and, thus, a lower speedup for the aggressive processor model compared to the realistic processor model.

Finally, we simulated several other machine configurations to determine the effect of the functional unit availability on the speedup. Due to space limitations, these results are not presented. However, even in the most unrealistic case in which we have 4 of each type of functional unit, the speedup results are still quite good, giving approximately 6.5% speedup for the SPEC benchmarks, 4.5% for the MediaBench benchmarks, and 6.2% overall. 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 [Richardson92]. In this paper, Richardson restricted the definition of trivial computations to certain multiplications (by 0, 1, and –1), divisions ($X \div Y$ with $X = \{0, Y, -Y\}$), and square roots of 0 and 1. The two key differences between this work and our current study are: 1) The types of benchmarks that were used and 2) 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 key 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 occur.

While there has been a definite lack of published material on trivial computations, several papers have described the related technique of value reuse [Citron98, Gonzalez98, Huang99, Molina99, Oberman95, and Sodani97]. With value reuse, an on-chip table dynamically caches the opcode, input operands, and results 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 the reuse technique 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.* [Molina99] 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. 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. The third difference between these approaches 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 study 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 tozero, one, or to the value of one of the input operands. This study 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, cannot remove these trivial computations. Furthermore, this study 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.

Finally, as items of future work, we are planning to examine the power consumption aspect of eliminating trivial computations, as well as eliminating trivial computation to improve the performance and table utilization of value reuse.

References

[Burger97] D. Burger and T. Austin; "The Simplescalar Tool Set, Version 2.0"; University of Wisconsin Computer Sciences Department Technical Report 1342.

[Citron98] D. Citron and D. Feitelson; "Accelerating Multi-Media processing by Implementing Memoing in Multiplication and Division Units"; International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.

[Gonzalez98] A. Gonzalez, J. Tubella, and C. Molina; "The Performance Potential of Data Value Reuse"; University of Politecenica of Catalunya Technical Report: UPC-DAC-1998-23.

[Huang99] J. Huang and D. Lilja; "Exploiting Basic Block Locality with Block Reuse"; International Symposium on High Performance Computer Architecture, 1999.

[Kessler98] R. Kessler, E. McLellan, and D. Webb; "The Alpha 21264 Microprocessor Architecture"; International Conference on Computer Design, 1998.

[KleinOsowski00] 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, pp. 83-100.

[Lee97] C. Lee, M. Potkonjak, and W. Mangione-Smith; “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”; International Symposium on Microarchitecture, 1997.

[Molina99] C. Molina, A. Gonzalez, and J. Tubella; "Dynamic Removal of Redundant Computations"; International Conference on Supercomputing, 1999.

[Oberman95] S. Oberman and M. Flynn; "On Division and Reciprocal Caches"; Stanford University Technical Report CSL-TR-95-666, 1995.

[Richardson92] S. Richardson; "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation"; International Symposium on Computer Arithmetic, 1993.

[Sodani97] A. Sodani and G. Sohi; "Dynamic Instruction Reuse"; International Symposium on Computer Architecture, 1997.

[Yeager96] K. Yeager; "The MIPS R10000 Superscalar Microprocessor"; IEEE Micro, Vol. 16, No. 2, March-April 1996; Pages 28-40.