Improving Performance by Speculating Trivializing Operands in Trivial Instructions

Ehsan Atoofian, Amirali Baniasadi and Nikitas Dimopoulos
Electrical and Computer Engineering Department, University of Victoria
3800 Finnerty Road, Victoria BC V8P 5C2, CANADA
{atoofian, amirali, nikitas}@ece.uvic.ca

ABSTRACT

We use value prediction to improve processor performance by speculating the trivializing operands in trivial instructions. Trivial instructions are those instructions whose output can be determined without performing the actual computation. We show that it is possible to predict the trivializing operand in a trivial instruction with high confidence. As such, we produce trivial instruction results earlier than when they become available in a conventional processor. Moreover, we use the speculated operands to bypass executing trivial instructions. Consequently, we further reduce program execution time.

Our study shows that by using our technique and for the representative subset of SPEC'2K benchmarks studied here, it is possible to improve performance by up to 8.6% over a conventional processor.

1. INTRODUCTION

In this work we use value prediction selectively to speculate trivializing values and to break true dependencies for trivial instructions. A trivial instruction is an instruction whose output can be determined without performing the actual computation. For such instructions, we can determine the results immediately based on the value of one or both of the source operands. Examples are multiply or add instructions where one of the input operands is zero.

Determining the trivial instruction result without performing the computation will result in faster instruction execution. This, consequently, could result in earlier execution of those instructions depending on the trivial instruction output. This is, of course, if the required resources (functional units, ports, etc.) are available for the depending instructions.

We assume a typical load/store ISA where each instruction may have up to two source operands. We refer to the operand which trivializes the operation as the *trivializing operand (TO)*. Examples of TOs are the operand equal to zero in an add operation or the operand equal to one in a multiplication. We refer to the other operand, (*e.g.*, the non-zero operand in the add operation, or the operand not equal to one in the multiplication) as the *non-trivializing operand (NTO)*.

Yi and Lilja [8] show that detecting and eliminating trivial instructions dynamically can reduce the program's execution time. They identify trivial computations dynamically and improve performance by bypassing or simplifying them. Our study shows that simplifying instructions (*e.g.*, replacing a multiplication with a shift operation if the multiplicand is a power of 2) does not impact performance considerably. This is due to the fact that simplifiable instructions are infrequent and therefore do not contribute to performance as much as bypassable instructions do. Therefore in this study we focus on bypassing trivial instructions.

Yi and Lilja [8] also show that an optimizing compiler is often unable to remove trivial operations. This is the result of the fact that trivial values are not known at compile time. They also show that the amount of trivial computations does not heavily depend on program specific inputs.

Identifying trivial instructions dynamically is possible as soon as the TO and the instruction opcode are known. However, computing the result may not always require knowledge of both source operands. In some cases, *e.g.*, multiplying by zero, we do not need both operands to compute the result. Under such circumstances, the result will not depend on the other operand value. In other cases, *e.g.*, addition to zero, both operands are needed. We refer to those trivial instructions whose output could be calculated knowing only one of the operands as *full-trivial instructions*. We refer to those trivial instructions whose result could be computed only after knowing both operands as *half-trivial instruction*s. Our study shows that half-trivial instructions account for the majority of trivial instructions.

Table 1 reports full- and half-trivial computations studied in this work. We report both the operation and the particular source operand value that will result in trivializing the operation. It is possible to extend our study further to include other instruction types (*e.g.*, ABS). However, this will not impact our results as such instructions are very infrequent.

Table 1: Full- and half-trivial instructions studied in this work

Operation	*Full Triviality Condition*
Multiplication: A*B	A=0 or B=0
Division: A/B	A=0
AND: A & B	A=0x00000000 or B=0x00000000
Logical Shift: A<<B,A>>B	A=0
Arithmetic Shift: A<<B, A>>B	A=0 or A=0xffffffff
OR: A | B	A=0xffffffff or B=0xffffffff
Operation	***Half Triviality Condition***
Addition: A+B	A=0 or B=0
Subtraction: A-B	B=0 or A=B
Multiplication: A*B	A=1 or B=1
Division: A/B	B=1 or A=B
AND A & B	A=0xffffffff or B=0xffffffff or A=B
OR: A | B	A =0x00000000 or B=0x00000000 or A=B
XOR: A XOR B	A or B =0x00000000 or 0xffffffff
Logical Shift: A<<B,A>>B	B=0
Arithmetic Shift: A<<B, A>>B	B=0

Generally, computing trivial instruction results, while unnecessary, consumes available resources and results in extra latency. Therefore, bypassing the computation and obtaining the result without performing the computation will improve performance. Additionally, and in the case of full-trivial instructions, both performing the computation and obtaining the NTO are unnecessary.

The goal of this work is to use TO value locality to speculate the TO and improve processor performance by effectively bypassing both full- and half-trivial computations.

In particular, by predicting the TO, we perform the following:

- We execute full-trivial instructions as soon as the TO is speculated. Consequently, when the NTO is still not available, we do not wait for its availability.
- We execute half-trivial instructions as soon as the NTO is known. In other words, we wait for the NTO but speculate the TO.

Based on the above, we use value prediction to speculate trivial operands and produce trivial instruction results earlier than when they would have been produced in a conventional superscalar processor.

Provided that a sufficient number of trivial operands are accurately identified, we can potentially improve processor performance. However, possible mispredictions can increase overall program runtime. We take this overhead into account in our study and report how it impacts performance.

In summary, we make the following contributions:

- We show that trivializing operands are good candidates for value prediction.
- We show that it is possible to improve processor performance by using value prediction to identify and bypass trivial operations effectively.

The rest of the paper is organized as follows. In Section 2 we explain TO prediction in more detail. In Section 3 we explain implementation. In Section 4 we present our experimental evaluation. In Section 5 we review related work. Finally, in Section 6, we summarize our findings and offer suggestions for future work.

2. TRIVIAL OPERAND PREDICTION

The result of a trivial operation could be either one of the source operands or zero or one (*e.g.*, operations reported in Table 1). Trivial instruction frequency impacts potential benefits of trivial operand speculation. Therefore, in order to decide if detecting and bypassing trivial operations is worthwhile we need to know how frequently they appear in the code stream. In Figure 1 we report trivial instruction frequency. In addition, and to provide better insight we also report both full- and half-trivial instruction frequency for the subset of SPEC'2k benchmarks studied here. While the entire bar represents total trivial instructions, the lower part of each bar shows the frequency of half-trivial instructions and the upper part represents full-trivial instructions.

As represented by the entire bar, on average, trivial instructions account for 13% of the total instructions. *Mcf*, *art* and *vpr* have higher number of trivial instructions compared to others. *Amm* has the lowest number of trivial instructions.

In general half-trivial instructions outnumber full-trivial instructions. Full-trivial instructions may account for as much as one third of the total number of trivial instructions (*e.g., art)*. Meantime they may account for as little as 1% of the total trivial instructions (*e.g., bzp*). On average, about 90% of the trivial instructions are half-trivial while the remaining 10% are full-trivial instructions.

As reported in Table 1, different instruction types can be trivial depending on their source operand values. However the trivial instruction frequency is different from one instruction type to another.

Figure 2 reports how often each instruction type is trivial. With the exception of *sub* and *and* at least 20% of each instruction type is trivial. In cases such as mult, div, or and fadd trivial instructions account for more than half of the instructions. However, note that a high

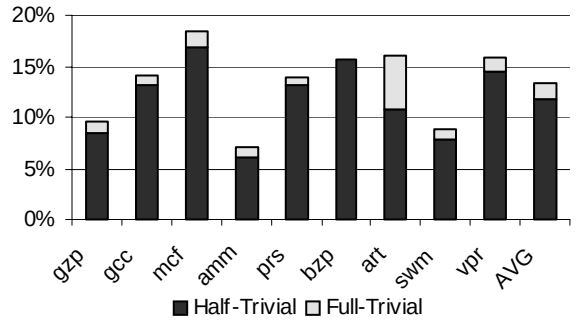

Figure 1: Trivial instruction frequency and distribution: The entire bar represents trivial instruction frequency. The lower part shows half-trivial instruction frequency while the upper part shows full-trivial instruction frequency. (See Table 2 for benchmark abbreviations)

percentage of trivial instructions for a specific instruction type does not always mean that the particular instruction type will have a considerable impact on performance. For example, while 90% of the multiplications appear to be trivial, they only account for less than 2% of the total number of instructions executed.

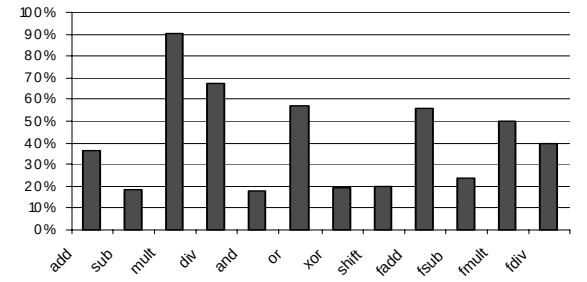

Figure 2: How often each instruction type is trivial.

In general, improving performance by predicting the trivial operands requires accurate and early value prediction. Nonetheless, accurate prediction of trivial values does not always result in better performance. To be precise, there is a group of trivial operands, *i.e.*, non-critical trivial values in half-trivial instructions, whose accurate and early predications do not improve performance. Between the two source operands of each instruction the non-critical operand is the one which becomes available sooner than the other. Since it is impossible to know the half-trivial instruction outcome before both operands are known, speculating non-critical trivial operands does not improve performance. In such cases, the processor has to wait for both operands before it can start instruction execution. As such, no matter how early and accurately we predict the non-critical operand, we will not reduce the program execution time.

Quite often, the sooner the critical operand is available the shorter the program execution time will be. However, executing an instruction not only requires available operands but also it is subject to resource availability.

One alternative to TO prediction is to speculate both the TO and the NTO in a trivial instruction to improve performance possibly even more. This would have been an attractive alternative if speculating the non-trivial operand as accurately as the trivial operand was easily possible. Unfortunately, this is not the case. Figure 3 reports value locality for both the TOs and the NTOs for the benchmarks studied here. Note that value locality describes the likelihood of the recurrence of a previously seen value within a register. Here we report value locality for a history depth of one, *i.e.*, we report how often the retrieved operand matches the most recently seen value. As reported, across all benchmarks, the TOs show higher value locality compared to the NTOs. On average, TOs and NTOs show 93% and 60% value locality respectively. Accordingly, we do not find speculating the NTOs as reliable since there is much higher confidence in the TOs as they show much stronger value-locality. As such, speculating both source operands will result in too many misspeculations which will ultimately harm performance. Therefore, in general, trivial operands are better candidates for value prediction.

Value speculation can potentially result in misspeculation. In this work, we flush all instructions after the misspeculated instruction and refetch instructions from the cache starting from the instruction immediately after the misspeculated instruction.

3. IMPLEMENTATION

To implement our technique, at decode, the inputs of potentially trivial instructions are predicted. If the instruction is predicted to be a trivial one, the rename table is modified so it maps the destination register to the physical register assigned to the input source operand or to the zero register. Consequently instructions depending on the destination operand of a trivial instruction do not need to wait.

Since we are modifying the renaming table speculatively we also need to change the checkpointing scheme so we can recover from possible value mispredictions properly. To recover form value mispredictions we checkpoint both the RAT and the read pointer of the free list upon speculating a trivializing operand. When the instruction producing the trivializing operand completes we check the speculated source operand. In case

a value misprediction is detected, we restore the associated checkpoint.

4. METHODOLOGY AND RESULTS

In this Section, we report our analysis framework. To evaluate our technique, we compare our suggested design‘s performance with a conventional processor.

- We report performance for a 512-entry last-value [3,4] predictor. We picked a 512-entry predictor after testing different predictor sizes. We have observed that further increases in the predictor size does not result in improving our technique.
- We also report performance for a processor which bypasses trivial instructions but, unlike our suggested design, does not speculate the trivial source operands.

As we need to know the instruction type to perform trivial value prediction, unlike many studies [1,3,9,10] we do not perform value prediction at fetch. Rather, we perform value prediction after the instruction is decoded.

We used both floating point (*amm*, *art* and *swm*) and integer (*gzp*, *vpr*, *gcc*, *mcf*, *prs* and *bzp*) programs from the SPEC CPU2000 suite compiled for the MIPS-like PISA architecture used by the Simplescalar v3.0 simulation tool set [6]. The benchmark set studied here incudes different programs including high and low IPC and those limited by memory, branch misprediction, etc.

We used GNU’s gcc compiler (flags: -O2 –funroll-loops –finline-functions). In the interest of space, we use the abbreviations shown under the “Ab.” column in Table 2. We simulated 200M instructions after skipping 200M instructions. We detail the base processor model in Table 3.

Table 2: Benchmark abbreviations used here

Program	Ab.
164.gzip	gzp
171.swim	swm
175.vpr	vpr
176.gcc	gcc
179.art	art
181.mcf	mcf
188.ammp	amm
197.parser	prs
256.bzip2	bzp

4.1. Performance

In Figure 4 we report performance improvements achieved by trivial value prediction over a conventional processor. Our processor uses value prediction on top of trivial value detection to bypass trivial instructions whose trivializing source operand is not available at decode time.

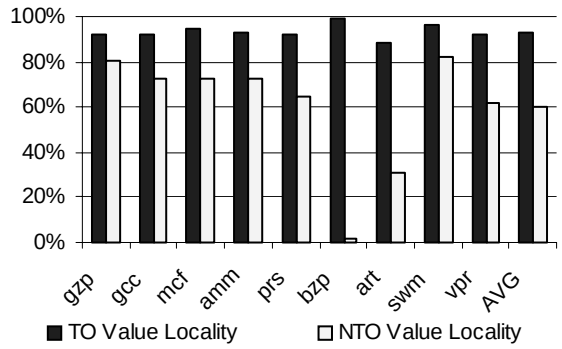

Figure 3: Value locality for the NTOs (left) and the TOs (right) in trivial instructions.

Table 3: Base processor configuration.

Integer ALU #	2
FP ALU #	2
Integer Multipliers/Dividers #	1
FP Multipliers/Dividers #	1
Instruction Fetch Queue #	32
Reorder Buffer Size	64
Load/Store Queue Size	32
Branch Predictor	8K GShare+8K bi-modal w/ 8K selector
Scheduler	64 entries, RUU-like
Fetch Unit	Up to 4 instr./cycle. 64-Entry Fetch Buffer
OOO Core	any 4 instructions / cycle
L1 - Instruction Caches	64K, 4-way SA, 32-byte blocks, 3 cycle hit latency
L1 - Data Caches	32K, 2-way SA, 32-byte blocks, 3 cycle hit latency
Unified L2	256K, 4-way SA, 64-byte blocks, 16-cycle hit latency
Main Memory	Infinite, 80 cycles
Memory Port #	2

To investigate if speculative trivial operand bypassing is worthwhile, we also report the performance improvement achieved by a processor which bypasses trivial instructions but does not use value prediction to identify trivial operands. Bars from left to right report performance improvement for a processor which bypasses trivial instructions without speculating TOs (referred to as non-speculative bypass) and a processor which uses a 512-entry last-value predictor over a conventional processor respectively.

As reported, processors using non-speculative bypass and last-value TO prediction improve performance by 6.8% and 8.6% over a conventional processor respectively.

Performance improvement is higher for *art* and *vpr* compared to other benchmarks when trivial instructions are bypassed. Meantime minimum performance improvement achieved by bypassing trivial instructions is observed for *amm.* This may be explained by the data presented in Figure 1 where *vpr* and *art* have high number of trivial instructions while *amm* has the lowest number of trivial instructions.

Note that there is an average performance gap of about 1.8% between the processor identifying trivial instructions speculatively and the processor that does not use speculation to identify trivial instructions. Speculative trivial instruction identification seems to be more effective for *vpr* and *art* where the performance improvement is higher over the non-speculative processor. On the other hand, speculating TOs seems to have little impact on performance for *amm, mcf, prs* and *bzp*. Improving the performance of the speculative processor may be possible by using more complex value predictors such as those suggested in [5,9,11,12,13,14].

The insignificant performance improvement achieved over the non-speculative bypassing processor for *amm*, *mcf*, *prs* and *bzp* could be explained as follows. While low predictor accuracy explains the small improvement achieved for *prs* a second factor needs to be considered to explain the results for *amm*, *mcf* and *bzp*. Our study shows that the second factor contributing to low performance for these benchmarks is the fact that for these three benchmarks TOs are mostly ready at the time of instruction decode and very rarely need to be speculated. As such, exploiting even more accurate predictors will not impact performance considerably. One possible way to improve performance for these benchmarks is to change our prediction scheme so value prediction could be done at instruction fetch. This would require storing information regarding instruction opcode in our value predictor.

Currently and as part of our ongoing research we are investigating how exploiting more accurate value predictors and different timing schemes could impact performance for our technique

.

5. RELATED WORK

Yi and Lilja [8]dynamically identified and bypassed trivial instructions. In this work we used value prediction on top of their technique to identify and execute trivial instructions that cannot be detected by their technique as the trivializing value is not available at decode time.

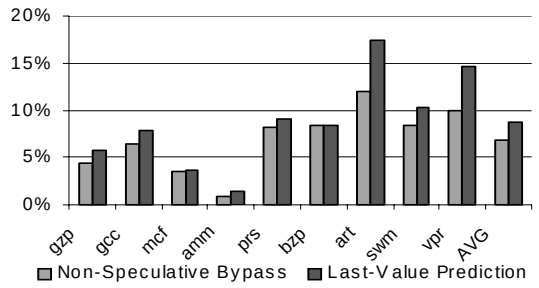

Figure 4: Bars from left to right report performance improvement achieved by using non-speculative bypass and last-value TO prediction over a conventional processor.

Many studies have suggested different approaches for data value prediction. These approaches include last value prediction [3,4], stride prediction [9,11], context prediction [5,12], and hybrid predictors [13,14]. In this work we use the last value predictor.

Value prediction has proven to be efficient if performed with high accuracy. However, achieving high prediction accuracies could be costly. While many studies have introduced highly accurate value predictors [12,13,14], there are a few which have addressed the cost, *i.e.*, access latency and energy, associated with value prediction (e.g., [7]). One way to reduce the cost and complexity associated with value prediction is to use it selectively and only for those values that there is high confidence in their behavior.

Calder *et al.* [1] examined selective techniques for using value prediction in the presence of predictor size restrictions and misprediction penalties. They used filtering to exploit value prediction only for instructions on the longest data dependence path. Consequently they minimized capacity conflicts. We selectively use value prediction for trivial instructions as we have observed that trivial values make excellent candidates for selective value prediction since they are highly predictable and show strong value locality and appear frequently.

Bhargava and John [7] introduced latency and energy aware value prediction. Their study showed that the latency of a high-performance value predictor cannot be completely hidden by the early stages of the instruction pipeline as many studies have assumed and can result in noticeable performance degradation. To address this problem they studied a value prediction approach that combined the latency-friendly approach

of decoupled value prediction with a more energy-efficient implementation.

Tran *et al.* evaluated dynamic methods to reduce pressure on the register file. They explored the impact of bypassing trivial instructions on the register file pressure [15]. We used their implementation of register remapping to bypass trivial instructions in this study.

Sato and Arita proposed techniques that exploit frequent value locality, resulting in budget reduction. They evaluated two value predictors(*i.e.*, the zero-value predictor and the 0/1-value predictor) [16]. Their low-cost 0/1 predictors could be used for trivial value prediction.

6. CONCLUSION

In this work we showed that it is possible to improve performance by using value prediction to speculate the trivializing operand of trivial instructions.

We showed that by using our technique it is possible to improve performance over a conventional processor. We also showed that it is possible to improve performance over a processor that bypasses trivial instructions without speculating the operands for some of the benchmarks studied here. We also provided insight as to why performance does not always improve over a processor using non-speculative trivial instruction bypass.

Our study shows that by using a last-value predictor, on average, we can improve performance over a conventional processor by 8.6%.

Possible future extensions to our work include examining how exploiting different and possibly more accurate value predictors impact our technique and using value prediction to speculate trivial instruction destination operand (instead of source operands studied here).

ACKNOWLEDGEMENT

We would like to thank the anonymous referees for their insightful comments. This work was supported by the National Sciences and Engineering Research Council of Canada.

REFERENCES

[1] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *25th International Symposium on Computer Architecture*, pages 64–74, May 1999. ‘

[2] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ILP. In *International Conference on Supercomputing*, pages 21–28, Jul 1998.

[3] M. H. Lipasti and J. P. Shen. Exceeding the data flow limit via value prediction. In *29th International Symposium on Microarchitectures*, pages 226–237, Dec. 1996.

[4] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

[5] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, Dec. 1997.

[6] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. *Technical Report CS-TR-96-1308, University of Wisconsin-Madison*, July 1996.

[7] R. Bhargava and L. John, Latency and Energy Aware Value Prediction for High-Frequency Processors, In *Proceedings of 16th ACM International Conference on Supercomputing*, pp. 45-56, June, 2002

[8] J. J. Yi and D. J. Lilja, Improving Processor Performance by Simplifying and Bypassing Trivial Computations, *Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 02-06*, June, 2002

[9] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th International Symposium on Computer Architecture*, pages 272–281, June 1998.

[10] B. Rychlik, J. Faistl, B. Krug, and J. P.Shen. Efficacy and performance impact of value prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 148–154, Oct. 1998.

[11] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. *Technical Report 1080, Technion - Israel Institute of Technology*, Nov. 1996

[12] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th International Symposium on Microarchitecture*, pages281–290, Dec. 1997.

[13] S. Lee, Y. Wang, and P. Yew. Decoupled value prediction on trace processors. In *6th International Symposium on High Performance Computer Architecture*, pages 231–240, Jan. 2000.

[14] B. Rychlik, J. W. Faistl, B. P. Krug, A. Y. Kurland, J. J. Sung, M. N. Velev, and J. P. Shen. Efficient and accurate value prediction using dynamic classification. *Technical report, Carnegie Mellon University*, 1998.

[15] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang. Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing. In *International Symposium on Performance Analysis of Systems and Software*. March 2004.

[16] T. Sato and I. Arita. Low-Cost Value Predictors Using Frequent Value Locality. *In High Performance Computing: 4th International Symposium, ISHPC 2002, Kansai Science City, Japan*, May, 2002.