Improving Energy-Efficiency by Bypassing Trivial Computations

Ehsan Atoofian and Amirali Baniasadi
ECE Department, University of Victoria
{eatoofia, amirali}@ece.uvic.ca

ABSTRACT

We study the energy efficiency benefits of bypassing trivial computations in high-performance processors. Trivial computations are those computations whose output can be determined without performing the computation. We show that bypassing trivial instructions reduces energy consumption while improving performance. Our study shows that by bypassing trivial instructions and for the subset of SPEC'2K benchmarks studied here, on average, it is possible to improve energy and energy-delay by up to 4.5% and 11.8% over a conventional processor.

1. INTRODUCTION

In this work we improve energy-efficiency in high-performance processors by bypassing 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 improve energy-efficiency in two ways: First, it will result in faster instruction execution. This, consequently, could result in earlier execution of those instructions depending on the trivial instruction output. This results in shorter program runtime which in turn reduces energy consumption. Second, by bypassing trivial instructions we no longer spend energy on executing them. As such, we reduce total energy consumption.

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.

Previous study shows that a) an optimizing compiler is often unable to remove trivial operations since trivial values are not known at compile time and b) the amount of trivial computations does not heavily depend on program specific inputs [2].

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 does 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 *fully-trivial instructions*. We refer to those trivial instructions whose result could be computed only after knowing both operands as *semi-trivial instruction*s. Our study shows that semi-trivial instructions account for the majority of trivial instructions. However, bypassing a fully-trivial instruction can impact performance and energy more than bypassing a semi-trivial instruction. This is due to the fact that fully-trivial instructions can be bypassed earlier and save more energy as they make reading both operands unnecessary.

Table 1 reports the fully-trivial and semi-trivial computations studied in this work. We report both the operation and the particular source operand value that trivializes 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.

Generally, power-aware techniques save energy at the expense of performance. Bypassing trivial instructions, however, reduces energy consumption while improving performance. Note that computing trivial instruction results, while unnecessary, results in extra latency and additional energy consumption. Therefore, bypassing the computation and obtaining the result without performing the computation will improve both performance and energy simultaneously.

In this work we study the energy benefits achieved by dynamically identifying and bypassing both fully-trivial and semi-trivial computations. In particular, we make the following contributions:

Table 1: Full- and semi-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
OR: A | B	A=0xffffffff or B=0xffffffff
Logical Shift: A<<B,A>>B	A=0
Arithmetic Shift: A<<B, A>>B	A=0
Operation	*Semi 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
Logical Shift: A<<B,A>>B	B=0
Arithmetic Shift: A<<B, A>>B	B=0

- We show that, by bypassing trivial instructions, it is possible to reduce energy consumption and improve energy-delay, on average, by 4.5% (min.: 1.5%) and 11.8% (min.: 3.5%) respectively.
- We categorize trivial instructions based on the number of source operands needed to detect them and their source operands availability time. We also study how often trivial instructions belong to each category and how this may impact our energy and performance improvements.

The rest of the paper is organized as follows. In Section 2 we explain bypassing trivial instructions in more detail. In Section 3 we explain our implementation. In Section 4 we present our experimental evaluation. In Section 5 we review related work. Finally, in Section 6, we summarize our findings.

2. TRIVIAL INSTRUCTION BYPASSING

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 instruction bypassing. 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(a) we report trivial instruction frequency. In addition, and to provide better insight we also report both fully-trivial and semi-trivial instruction frequency. While the entire bar represents total trivial instructions, the lower part of each bar shows the frequency of semi-trivial instructions and the upper part represents fully-trivial instructions.

As represented by the entire bar, on average, trivial instructions account for about 12% of the total instructions. *Gcc* and *vpr* have higher number of trivial instructions compared to others. *Wlf* has the lowest number of trivial instructions.

In general, semi-trivial instructions outnumber fully-trivial instructions. Fully-trivial instructions may account for as much as one third of the total number of trivial instructions (*e.g.*, *equ)*. Meantime they may account for as little as 2% of the total trivial instructions (*e.g., wlf*). On average, about 85% of the trivial instructions are semi-trivial while the remaining 15% are fully-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 1(b) reports how often each instruction type is trivial. As reported, at least 10% of each instruction type is trivial. In cases such as *mult, div*, and *or* trivial instructions account for more than half of the instructions. However, note that a high percentage of trivial instructions for a specific instruction type does not always mean that the particular instruction type will have a considerable impact on energy-efficiency. For example, while 90% of the divisions appear to be trivial, they only account for less than 1% of the total number of instructions executed.

Trivial instructions can only be bypassed when either both operands (for semi-trivial) or their TO (for

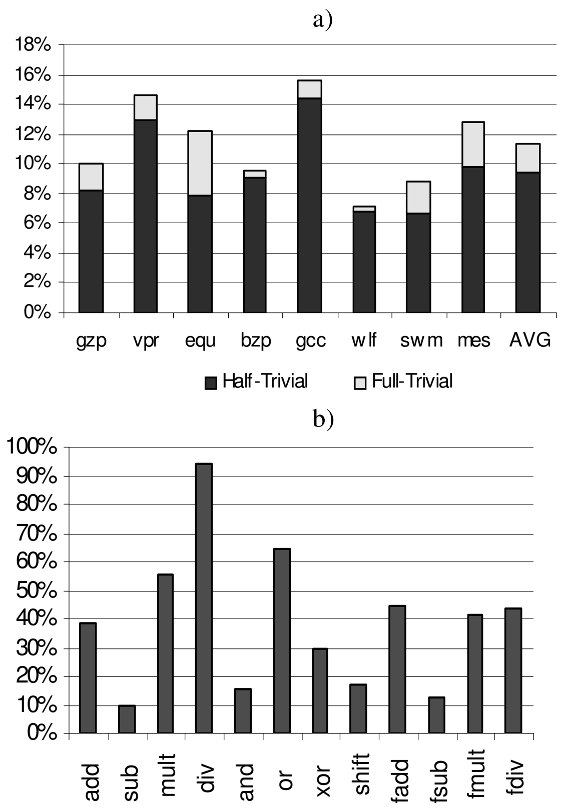

Figure 1: (a) Trivial instruction frequency and distribution: The entire bar represents trivial instruction frequency. The lower part shows semi-trivial instruction frequency while the upper part shows fully-trivial instruction frequency. (b) How often each instruction type is trivial. (See Table 2 for benchmark abbreviations)

fully-trivial) are known. Based on the source operand(s) availability time(s), we categorize trivial instructions to two groups:

The first group are those instruction whose source operand/operands (both operands for semi-trivial, the TO for fully-trivial) is/are known while they are at the decode stage. For this group, the required source operands have been produced early enough so the trivial instruction could be bypassed at decode stage.

The second group of trivial instructions are those instructions whose necessary operands are not available at instruction decode stage. Therefore, these trivial instructions could not be bypassed at the decode stage and are sent to the issue queue where they wait for their operands and the required resources to become available. This group of trivial instructions is identified at the issue stage and when the required source operands (again, both operands for semi-trivial, TO for fully-trivial) are known.

We refer to the trivial instructions identified at decode as *decode-trivial* and to those identified at issue as *issue-trivial.* In Figure 2 we report the percentage of decode-trivial and issue-trivial instructions. While the entire bar represents total trivial instructions (similar to Figure 1(a)), the lower part of each bar shows the frequency of decode-trivial instructions and the upper part represents issue-trivial instructions.

On average, issue-trivial instructions account for half of the trivial instructions. However, for some benchmarks (*e.g.*,*bzp* and *equ*) the number of decode-trivial instructions exceeds issue-trivial instructions. For other benchmarks (*e.g., gcc* and *vpr*) issue-trivial instructions are more than decode-trivial instructions.

Note that the earlier a trivial instruction is identified, the earlier it could be bypassed. As such, we expect higher energy savings and performance improvements achieved by decode-trivial instruction compared to issue-trivial instructions.

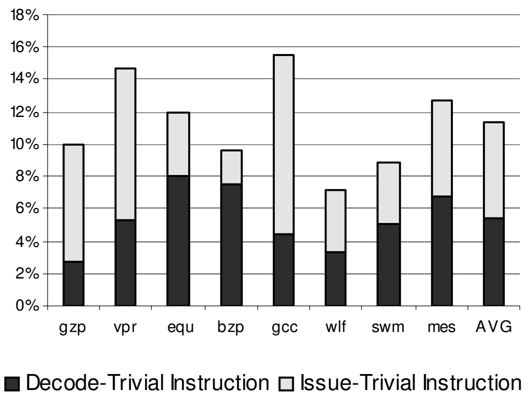

Figure 2: Trivial instruction frequency and distribution: The entire bar represents trivial instruction frequency. The lower part shows decode-trivial instruction frequency while the upper part shows issue-trivial instruction frequency. (See Table 2 for benchmark abbreviations)

3. IMPLEMENTATION

In this work we assume that all reservation stations monitor their source operands for data availability simultaneously. We also assume that at dispatch, already-available operand values are read from the register file and stored in the reservation station. The reservation station logic compares the operand tags of unavailable data with the result tags of completing instructions. Once a match is detected, the operand is read from the bypass logic. As soon as all operands become available in the reservation station, the instruction may issue (subject to resource availability) [9]. An alternative implementation is storing pointers to where the operand can be found (*e.g.* in the register file) rather than storing the data in the reservation station [10]. While trivial instruction bypassing could be used on top of both implementations, here we assume the former.

Figure 3 shows the schematic of a processor that bypasses trivial instructions and the procedures followed. We first discuss decode-trivial instructions. At decode, the *Trivial Instruction Detection* unit examines source operands. If the instruction is trivial, 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 as presented in 3(b). Once the renaming table is modified, we no longer execute the trivial instruction. As such, instructions depending on the trivial instruction result can start execution immidiately (subject to resource availability). Note that decode-trivial instructions, once detected, do not consume execution unit resources.

To identify trivial instructions while they are in the issue queue the trivial instruction detection unit examines the produced data as soon as the associated tag is received by the reservation station. Once we detect an issue-trivial instruction we bypass executing the instruction and send the result to the write-back unit as presented in 3(c). However, the destination register of an issue-trivial instruction should not be released since there may still be instructions depending on the trivial instruction outcome which have not read their source operands yet.

Note that, in order to improve performance, modern processors wakeup consumer instructions in advance and before the data is actually available. This makes executing producer-consumer pairs in consecutive cycles possible. As a result, issue-trivial instructions would have to be issued first and then read operands to test triviality. Consequently, in this study we assume that issue-trivial instructions take issue slots but will not be executed in the ALU and will write their results as soon as possible. Therefore, issue-trivial instructions benefit less from trivial instruction bypassing compared to decode-trivial instructions.

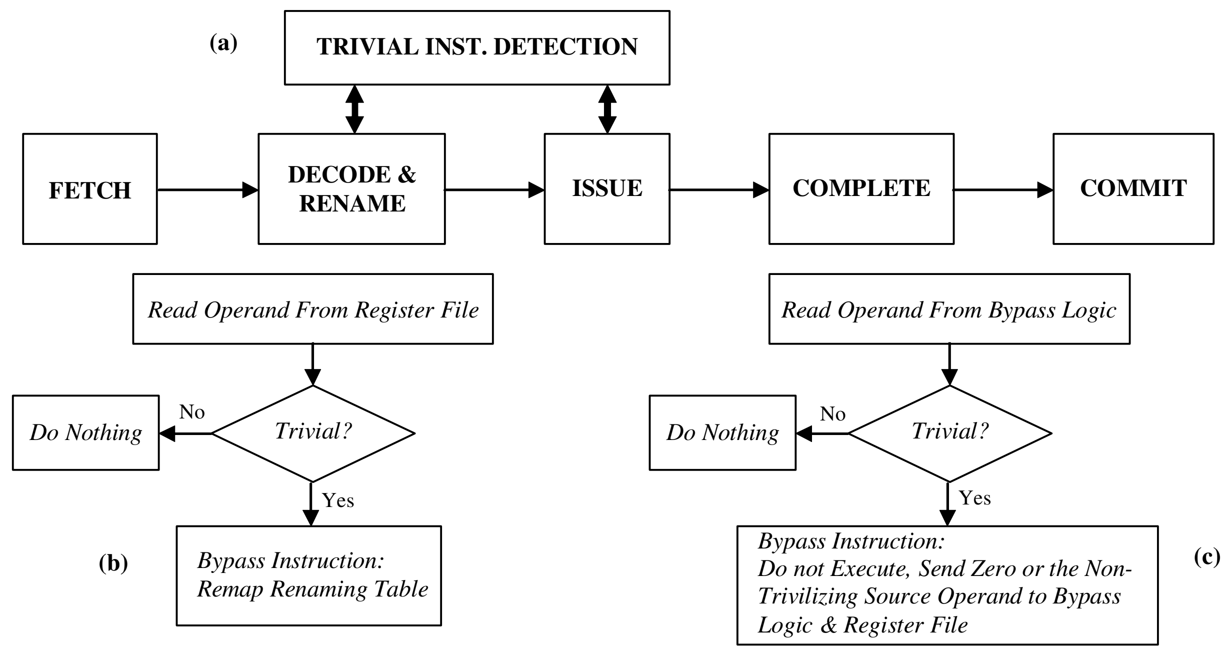
(a)
TRIVIAL INST. DETECTION
FETCH
DECODE & RENAME
ISSUE
COMPLETE
COMMIT
Read Operand From Register File
Trivial?
No
Do Nothing
Yes
(b)
Bypass Instruction: Remap Renaming Table
Read Operand From Bypass Logic
Trivial?
No
Do Nothing
Yes
Bypass Instruction: Do not Execute, Send Zero or the Non-Trivilizing Source Operand to Bypass Logic & Register File
(c)

Figure 3: a) Schematic for a pipelined processor bypassing trivial instructions b) Decode-Trivial instruction detection procedure c) Issue-Trivial instruction detection procedure.

4. METHODOLOGY AND RESULTS

In this Section, we report our analysis framework. To evaluate how bypassing trivial instructions impacts performance and energy, we compare our processor with a conventional processor that does not bypass trivial instructions. We report performance, energy and energy-delay.

We used both floating point (*equ*, *mes* and *swm*) and integer (*gzp, vpr, gcc, bzp* and *wlf*) programs from the SPEC CPU2000 suite compiled for the MIPS-like PISA architecture used by the Simplescalar v3.0 simulation tool set [1]. We used WATTCH [4] for energy estimation. The benchmark set studied here includes different programs including high and low IPC and those limited by memory, branch misprediction, etc.

Note that detecting and bypassing trivial instructions requires additional hardware. Consequently, and depending on how the technique is implemented, this will result in power overhead. Through this study we assume that this power overhead is negligible compared to our savings.

We used GNU's gcc compiler. In the interest of space, we use the abbreviations shown under the "Ab." column in Table 2. We simulated 500M instructions after skipping 500M instructions. We detail the base processor model in Table 3.

Table 2: Benchmark abbreviations used here

Program	Ab.	Program	Ab.
164.gzip	gzp	***177.mesa***	mes
171.swim	swm	***183.equake***	equ
175.vpr	vpr	***256.bzip2***	bzp
176.gcc	gcc	***300.twolf***	wlf

Table 3: Base processor configuration.

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

4.1. Performance

Bypassing trivial instructions will improve performance only if the bypassed instructions are on the critical path. To investigate how bypassing trivial instructions impacts performance, in Figure 4, we report performance improvements compared to a conventional processor. *Vpr* and *mes* show higher performance improvements compared to other benchmarks. *Wlf* has the lowest performance improvement among all benchmarks.

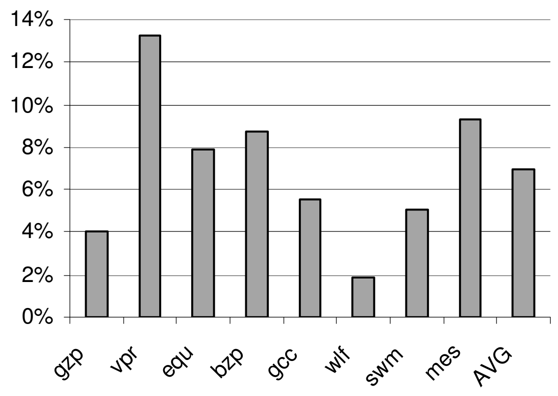

Figure 4: Performance improvement achieved by bypassing trivial instructions over a conventional processor.

4.2. Energy and Energy-Delay

In Figure 5 we report energy and energy-delay measurements. In 5(a) we report energy savings achieved by bypassing trivial instructions. *Wlf* has the lowest energy savings compared to other benchmarks. *Vpr* and *equ* have higher energy reduction compared to the rest of the benchmarks.

In 5(b) we report energy-delay improvements achieved by bypassing trivial instructions. Again, *wlf* has the lowest energy-delay improvement compared to other benchmarks. *Vpr* has the highest energy-delay improvement among all benchmarks

4.3. Discussion

In this Section we review the results. A detailed analysis of the results would require studying many issues including instruction type distribution for the bypassed instructions and how often critical path instructions are bypassed for each benchmark. Our discussion here, however, only focuses on the data presented earlier. We discuss integer and floating point benchmarks separately.

1) The integer benchmarks studied here include *gzp, vpr, bzp, gcc* and *wlf*. Among integer benchmarks, *vpr* and *gcc* have higher number of trivial instructions. The high number of trivial instructions for *vpr* explains why this benchmark benefits more than other benchmarks from bypassing trivial instructions. As for *gcc*, how-

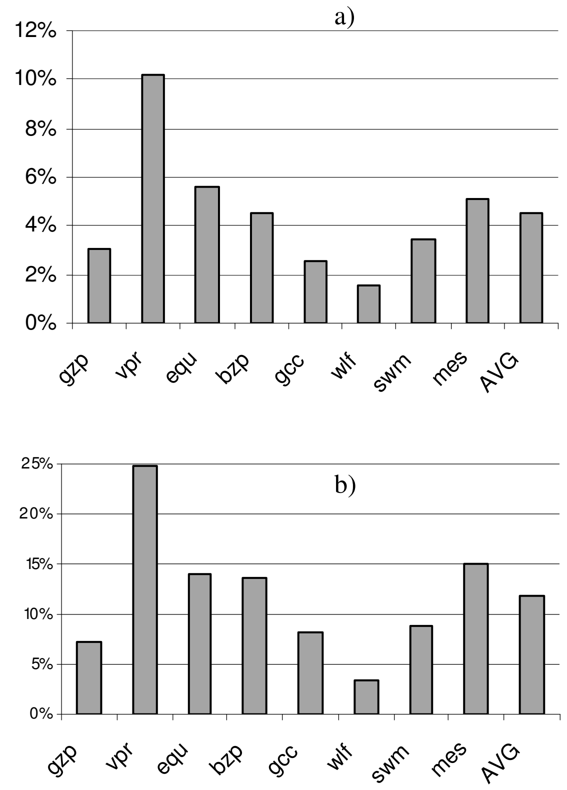

Figure 5: (a) Energy improvement achieved by bypassing trivial instructions over a conventional processor. (b) Energy-delay improvement achieved by bypassing trivial instructions over a conventional processor.

ever, the number of decode-trivial instructions is lower than *vpr* and *bzp* and only higher than *gzp* and *wlf.* Also note that *gcc* has a relatively low number of fully-trivial instructions compared to other integer benchmarks (see Figure 1(a)). These two factors may explain why gcc falls behind in performance and energy measurements despite having a high number of trivial instructions.

Among the integer benchmarks *wlf* has the lowest number of trivial instructions. This may explain why it has the lowest performance, energy and energy-delay improvement among integer benchmarks.

2) The floating point benchmarks include *equ, swm* and *mes*. Among floating point benchmarks *mes* has the highest number of trivial instructions. This may explain why *mes* has the highest performance improvement among the floating point benchmarks. *Equ,* while having less number of trivial instructions compared to *mes*, has a higher number of decode-trivial instructions. This may explain why *equ* shows high energy savings. Finally, *swm* falls behind *mes* and *equ* in both performance and energy measurements. This is consistent with the fact that *swm* has less number of trivial, decode-trivial and fully-trivial instructions when compared to *mes* and *equ.*

5. RELATED WORK

Previous study has introduced many dynamic optimization techniques to reduce the complexity or latency associated with producing operands.

Lipasti and Shen introduced value prediction and showed that data values exhibit “locality” where values computed by some instructions tend to repeat [7].They suggested using this locality to exceed the dataflow limit and to effectively and speculatively produce operands earlier than when they normally become available.

Sodani and Sohi introduced the concept of dynamic instruction reuse[8]. Their work relied on the observation that many instructions, having the same inputs, are executed dynamically. As such, many instructions do not have to be executed repeatedly since their results can be obtained from a buffer where they were saved previously.

Trivial instruction bypassing is not speculative. Moreover, we detect trivial instructions no matter how infrequent they are. Also, as we do not rely on instruction past behavior, we do not require additional storage to store the associated information. Consequently we are able to improve both power and performance simultaneously.

Richardson suggested a restricted form of bypassing trivial instructions [5]. His definition of trivial computations only included certain multiplications (by 0, 1, and –1), divisions (X ÷ Y with X = {0, Y, -Y}), and square roots of 0 and 1.

Yi and Lilja [2] showed that detecting and eliminating trivial instructions dynamically can reduce the program execution time. They identified trivial computations dynamically and improved 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 overall energy-efficiency considerably. This is due to the fact that simplifiable instructions are infrequent and therefore do not contribute to energy or performance as much as bypassable instructions do. Therefore, in this study we focus only on bypassing trivial instructions. Moreover, in this work we studied how bypassing trivial instructions results in higher energy-efficiency. We also extended their study by providing a deeper and more detailed analysis of trivial instructions.

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 [3]. We used their implementation of register remapping to bypass decode-trivial instructions in this study.

Chung *et al.*, suggested optimization of embedded software [6].They presented software-based techniques to reduce the computational effort of programs, using value profiling and partial evaluation. Their tool reduced the computational effort by specializing frequently executed procedures for the most common values of their parameters. Their work included introducing software solutions to replace complex functions which have trivializing operands with more simple functions. Our work is different as it introduces dynamic hardware-based optimizations. Dynamic optimizations can be integrated with the microarchitecture and result in better architectural transparency. As such, our technique can optimize legacy code without the need for recompilation.

6. CONCLUSION

In this work we showed that it is possible to improve energy consumption and energy-delay by bypassing trivial instructions.

We categorized trivial instructions to fully-trivial and semi-trivial instructions based on whether both source operands are needed to decide the result. We also categorized trivial instructions to decode-trivial and issue-trivial instruction based on the pipeline stage that they could be identified at. We showed that semi-trivial instructions account for the majority of trivial instructions while, on average, decode-trivial and issue-trivial instructions each account for almost half of the total trivial instructions.

Our study showed that by bypassing trivial instructions, on average, we can improve energy consumption and energy-delay by 4.5% and 11.8% over a conventional processor.

REFERENCES

[1] 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.

[2] 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

[3] 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.

[4] D. Brooks, V. Tiwari M. Martonosi, “Wattch: A Framework for Architectural-Level Power Analysis and Optimizations”*, In Proceedings of International Symposium on Computer Architecture*, 2000.

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

[6] E. Chung, L. Benini, G. DeMicheli, G. Luculli, and M. Carilli, “Value-Sensitive Automatic Code Specialization for Embedded Software”, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* VOL. 21, NO. 9, September 2002

[7] M. H. Lipasti and J. P. Shen, “Exceeding the dataflow limit via value prediction”, in *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 226–237, December 1996.

[8] A. Sodani and G. S. Sohi. “Dynamic Instruction Reuse”. In *Proc. of 24th Annual International Symposium on Computer Architecture*, pages 194–205, July 1997.

[9] Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, San Francisco, CA, 1990, 1996, 2003.

[10] G. S. Sohi, “Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers”, *IEEE Trans. on Computers*, vol. 39, pp. 349-359, March 1990.