Available online at www.sciencedirect.com

ScienceDirect

Journal of Systems Architecture 53 (2007) 587–601

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Speculative trivialization point advancing in high-performance processors

Ehsan Atoofian, Amirali Baniasadi *

ECE Department, University of Victoria, Victoria BC, Canada V8P5C2

Received 3 May 2006; received in revised form 31 October 2006; accepted 19 December 2006
Available online 17 January 2007

Abstract

Trivial instructions are those instructions whose output can be determined without performing the actual computation. This is due to the fact that for these instructions the output is often either one of the source operands or zero (e.g., addition with or multiplication by zero). In this work we study trivial instructions and use our findings to improve performance in high-performance processors.

In particular, we introduce speculative trivialization point advancing to detect and bypass trivial instructions as soon as possible and as early as the decode stage. Consequently, we improve performance over a conventional processor (up to 30%) and a processor that detects and bypasses trivial instructions at their conventional point of trivialization (up to 5%).
© 2007 Elsevier B.V. All rights reserved.

Keywords: High-performance processors; Trivial instructions; Value prediction

1. Introduction

A trivial instruction (TI) is an instruction whose output can be determined without performing the actual computation. Almost all arithmetic and logic computations could be trivialized. Examples are and or add instructions where one of the input operands is zero. It is important to note that (a) an optimizing compiler is often unable to remove TIs and (b) TIs do not heavily depend on program specific inputs [7].

TIs can account for as much as 20% of the executed instructions in some applications. Executing such instructions, unnecessarily, results in extra latency and power dissipation. To exploit this inefficiency previous study has suggested detecting and bypassing TIs to improve performance [7]. By skipping TIs we improve performance in two ways: First, we determine the TI outcome sooner and without performing the computation. As a result, instructions depending on the TI outcome can execute sooner. Second, we increase the number of free resources, reducing the number of structural hazards and instruction stalls.

Assuming a typical load/store ISA, each instruction may have up to two source operands. We refer to the operand which trivializes the operation as the *trivializing operand (TO)*. An example of a TO is the operand equal to zero in an add operation. We refer to the other operand, (e.g., the non-zero

* Corresponding author.
E-mail address: amirali@ECE.UVic.CA (A. Baniasadi).

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2006.12.009

operand in the add operation) as the *non-trivializing operand (NTO)*.

To detect a TI, the instruction opcode and source operands should be known. While the opcode is known as early as instruction decode, source operands may become available later. The moment source operands and the opcode are known a TI can be identified and bypassed. We refer to this moment in time as the *Trivialization Point* (TP). Performance improvement achieved by bypassing a TI depends on how early the TP occurs. In a conventional processor TP coincides with the source operands availability time or instruction decode, whichever comes later. This may occur at anytime from instruction decode time (for already available source operands) to instruction issue time (for instructions whose source operands are not available at decode time).

In this paper we show that for the majority of the TIs, TP occurs too late to be exploited efficiently. In other words, by the time we know about the instruction triviality, the instruction has already consumed many processor resources. We use value prediction to advance the trivialization point and to identify TIs at a point earlier than their original TP. We refer to our technique as *speculative TP advancing* (or simply *TP-advancing*). TP-advancing improves performance by breaking dependency chains earlier than when it is done in a conventional processor.

While identifying TIs is possible as soon as the TO and the instruction opcode are known, 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 as the result does not depend on the NTO. In other cases, e.g., addition to zero, both operands are needed. We refer to those TIs whose outputs can be obtained knowing only one of the operands as *fully-trivial instructions.* We refer to those TIs whose result can be computed only after knowing both operands as *semi-trivial instructions*. In Table 1 we report fully- and semi-trivial computations studied in this work. We report both the operation and the trivializing source operand value. We exclude all instructions with invalid operands (e.g., zero divide by zero) in this study. 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.It is important to note that the result of a TI is either zero or the NTO.

Table 1
Fully- and semi-trivial instructions and their trivializing operands

Operation	Fully triviality condition
Multiplication: A∗B	$A = 0$ or $B = 0$
Division: A/B	$A = 0$
AND: A & B	$A = 0 \times 00000000$ or $B = 0 \times 00000000$
Logical shift: $A \ll B$, $A \gg B$	$A = 0$
Arithmetic shift: $A \ll B$, $A \gg B$	$A = 0$
Operation	Semi triviality condition
Addition: $A + B$	$A = 0$ or $B = 0$
Subtraction: $A - B$	$B = 0$
Multiplication: $A * B$	$A = 1$ or $B = 1$
Division: A/B	$B = 1$
AND A&B	$A =$ 0xffffffff or $B =$ 0xffffffff
OR: $A \vert B$	$A = 0 \times 00000000$ or $B = 0 \times 00000000$
XOR: A XOR B	$A = 0 \times 00000000$ or $B = 0 \times 00000000$
Logical shift: $A \ll B$, $A \gg B$	$B = 0$
Arithmetic shift: $A \ll B$, $A \gg B$	$B = 0$

TP-advancing could be applied to both fully- and semi-trivial computations. Accordingly, fully-trivial instructions could be bypassed as soon as the TO is speculated (i.e., if the NTO is not available, there is no need to wait for its availability). Semi-trivial instructions could be executed as soon as the NTO is known (i.e., we wait for the NTO but speculate the TO).

TO speculation does not always improve performance. To be precise, there is a subset of trivial operands, i.e., non-critical trivial values in semi-trivial instructions, whose accurate and early prediction does not improve performance.[1] In this work we study TO criticality and show that the majority of TIs benefit from TP-advancing.

In summary, we make the following contributions:

(1) We study TIs from several aspects including locality, triviality degree, point of triviality, frequency and distribution.
(2) We show that using value prediction to advance instruction TP improves processor performance considerably over a conventional

[1] 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 semi-trivial instruction outcome before both operands are known, speculating the non-critical trivial operand does not result in early instruction execution.

processor. We also show that TP-advancing improves performance over a processor detecting and bypassing TIs at their original TP (referred to as the *TP-original processor*).
(3) We introduce a low cost hardware implementation of TP-advancing. We achieve this by using speculation selectively and for operands with higher data locality. We also suggest mechanisms to identify and bypass TIs.
(4) We show that speculating trivial source operands (rather than instruction results) can achieve higher prediction accuracy.

The rest of the paper is organized as follows. In Section 2 we discuss TP-advancing in more detail. In Section 3 we explain hardware implementation. In Section 4 we present the benchmarks and the evaluation methodology. We report our results in Section 5. We review related work in Section 6. Finally, in Section 7, we summarize our findings and offer conclusions.

2. TP-advancing

In this Section we discuss TP-advancing in more details. We provide better insight by providing a real example in Section 2.1. In Section 2.2 we discuss TI frequency and distribution. In Section 2.3 we use instruction TP and classify TIs to decode- and issue-trivial and study how bypassing each group impacts performance.

2.1. Trivial instruction example

To provide better insight on how TP-advancing improves performance, in Fig. 1, we present an example selected from the *gzip* benchmark. *Gzip* divides streams into blocks and compresses each block separately by using a combination of the LZ77 algorithm [16] and Huffman coding [17]. LZ77 generates a file which is later used by Huffman coding to produce the final compressed file. LZ77 algorithm finds repeating sequences in the input data stream. The first time a sequence appears LZ77 writes it to its output file. Future reappearances of the sequence are coded using two numbers: a distance showing how far back the first appearance of the sequence is located and a length representing how many characters build the sequence. Once LZ77 has processed all data blocks the generated file is passed to Huffman coding. Note that Huffman codes are constant within a block. By using Huffman coding *gzip* further compresses the input file.

The deflate() function is a part of *gzip* (Fig. 1a). This function generates compressed data by using LZ77 and Huffman coding. Inside of this function, the ct_tally() function is called (Fig. 1b). Ct_tally() saves the matched information and counts the frequency of different Huffman codes. The two arguments of ct_tally() are distance and length for a matched string. Length_code is an array that relates matched length to Huffman code. Dyn_tree[].Freq shows the number of times a Huffman code is used during compression. In Fig. 1c we show a part of the assembly code corresponding to the ct_tally() function. The first instruction in Fig. 1c loads length_code[lc] into the R [2] register. The Huffman code corresponding to length three is equal to zero. This value does not change within a block. Whenever a match with length three is detected in the input stream, lbu loads the same value (zero) into register R [2]. By using value prediction, we can predict the value of R [2] when ct_tally() is called in the future. All three instructions following lbu depend directly or indirectly on the value of R [2]. Consequently, by predicting the value of R [2] (which is the TO of addiu), dependent instructions can execute earlier. Also, addiu no longer needs ALU resources, leaving ALU resources to instructions which may otherwise be stalled due to structural hazards.

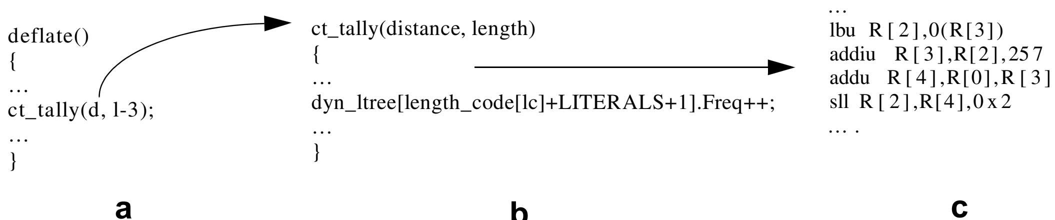

Fig. 1. (a) Deflate function, (b) ct_tally() function, (c) part of assembly code corresponding to ct_tally() function.

2.2. Trivial operand frequency and distribution

To decide if detecting and bypassing TIs is worthwhile, in Fig. 2 we report TI frequency. In addition, and to provide better insight we also report both fully- and semi-trivial instruction frequency for a subset of SPEC'2k and MiBench benchmarks studied here. While the entire bar represents total TIs, the lower part of each bar shows the frequency of semi-trivial instructions and the upper part represents fully-trivial instructions. TI frequency can be as low as 5% and as high as 20% for the applications studied here. *Basicmath, patrica* and *qsort* have higher number of TIs compared to others. *Adpcm-decode* has the lowest number of TIs.

In general semi-trivial instructions outnumber fully-trivial instruction. Fully-trivial instructions may account for as much as one third of the total number of TIs (e.g., *jpeg-encode*). Meantime they may account for less than 1% of the total number of TIs (e.g., *adpcm-decode*).

As reported in Table 1, different instruction types can be trivial depending on their source operand values. As reported in Fig. 3, at least 20% of each instruction type is trivial. In cases such as *mult* and *or* TIs account for more than half of the instructions. A high TI percentage for an instruction type does not always mean that the particular instruction type has a considerable impact on performance. For example, while 69% of the multiplications are trivial, they only account for less than 2% of the instructions executed.

2.3. Decode- and issue-trivial instructions

TI source availability time impacts performance improvement achieved by TP-advancing. Advancing TP is only possible if the instruction operands are not available at decode. If the TI source operands are available at decode, the TI can be identified and bypassed without speculation. Accordingly, based on the source operand(s) availability time(s), we categorize TIs into two groups:

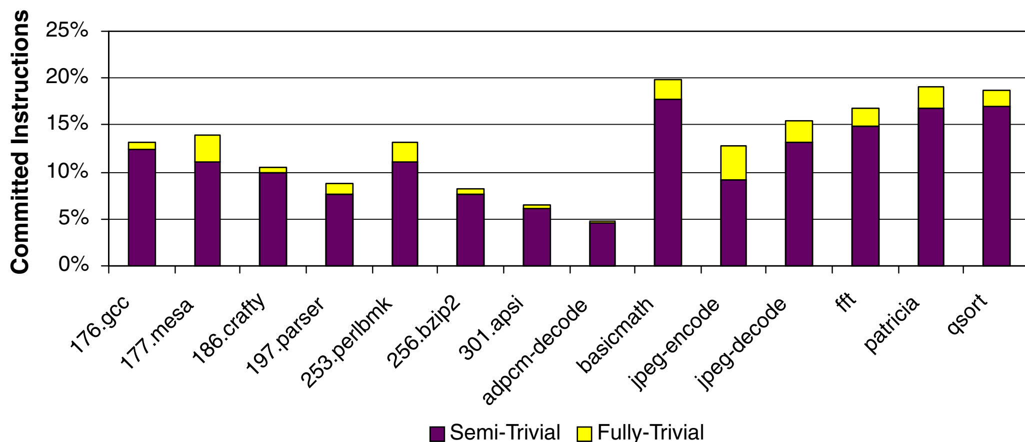

Fig. 2. TI frequency and distribution: the entire bar represents TI frequency. The lower part shows semi-trivial instruction frequency while the upper part shows fully-trivial instruction frequency.

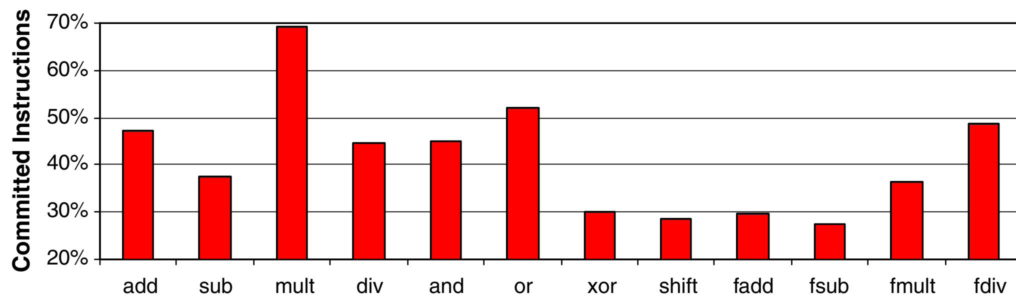

Fig. 3. Instruction type triviality frequency.

The first group includes instructions whose source operand(s) (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 TI could be bypassed at the decode stage.

The second group of TIs includes instructions whose necessary operands are not available at the decode stage. Therefore, these TIs 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. In a conventional processor this group of TIs can only be detected 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 TIs detectable at decode as *decode-trivial* and to those detectable at issue as *issue-trivial*. In Fig. 4 we report the percentage of decode-trivial and issue-trivial instructions. While the entire bar represents total TIs, the lower part of each bar shows the frequency of decode-trivial instructions and the upper part represents issue-trivial instructions. Issue-trivial instructions account for the majority of the TIs for most benchmarks. However, for others (e.g., *gcc* and *parser*), decode-trivial instructions outnumber issue-trivial instructions.

TP-advancing only applies to issue-trivial instructions as it identifies issue-trivial instructions earlier than when they could be identified in a conventional processor. While TP-advancing relies on accurate and early TO value prediction, accurate TO prediction is not always helpful. To be precise, there is a subset of TOs, i.e., non-critical TOs in semi-trivial instructions, whose accurate and early predications do not improve performance. Since it is impossible to know the semi-trivial instruction outcome before both operands are known, speculating the non-critical TO does not result in early instruction execution. In such cases, the processor

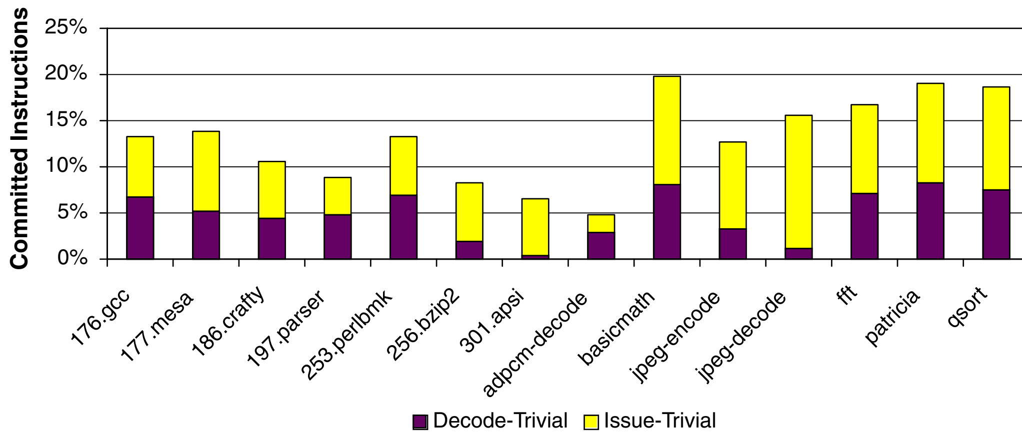

Fig. 4. TI frequency and distribution: the entire bar represents TI frequency, lower part shows decode-trivial instruction frequency while the upper part shows issue-trivial instruction frequency.

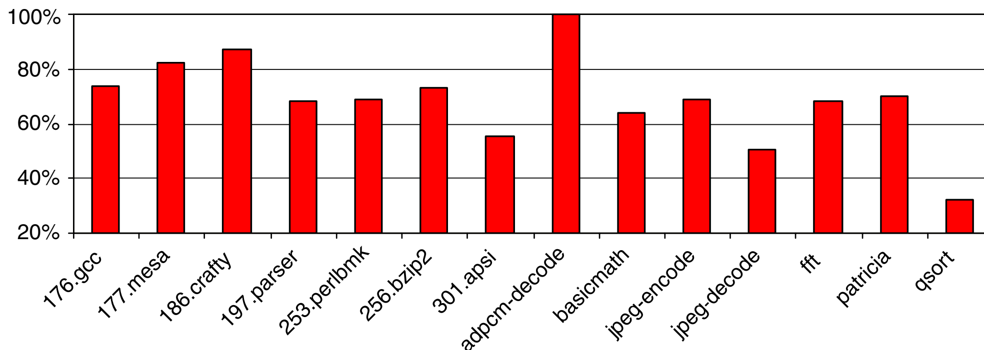

Fig. 5. Trivializing operand criticality frequency.

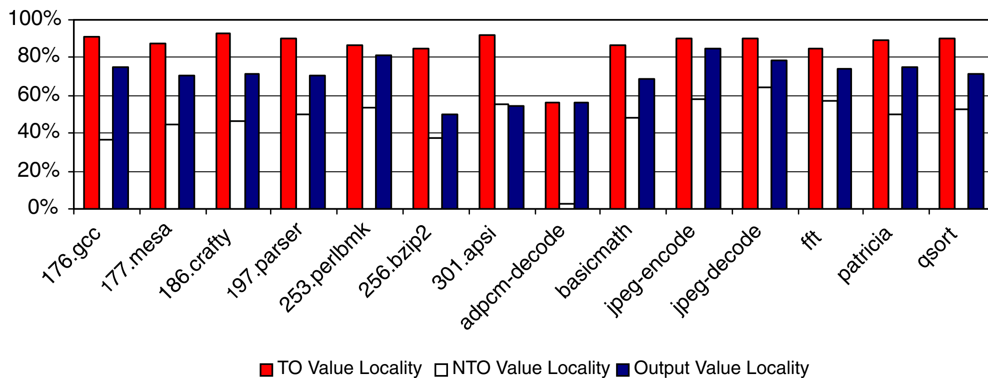

Fig. 6. Bars from left to right report value locality for TOs, NTOs and TI outcomes.

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. Fig. 5 reports how often TOs are critical in issue-trivial instructions. As reported, for many applications, TOs are indeed often critical.

One alternative to TO prediction is to speculate the TI result. This would make faster (speculative) instruction execution possible as by speculating the instruction result, we no longer have to wait for both source operands.[2] Predicting the TI outcome would have been an attractive alternative if the TI output was as predictable as the TO. Unfortunately, this is not the case. To explain this further, in Fig. 6 we report value locality for TOs, NTOs and the TI outcomes for the baseline processor (see Table 2). Value locality describes the likelihood of the recurrence of a previously seen value within a register. We report value locality for a history depth of one, i.e., how often the retrieved operand matches the most recently seen value. As reported, across all benchmarks, TO shows higher value locality compared to the NTO and TI outcome. Accordingly, to maintain accuracy, we do not speculate the output.

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) [18]. 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 [19]. While TI bypassing could be used on top of both implementations, here we assume the former.

3.1. Trivial bypassing hardware

Fig. 7 shows the schematic of a processor that bypasses TIs and the procedures followed. The *Trivial Instruction Detection Unit (TDU)* detects decode-trivial instructions at decode (Fig. 7b). Upon detection, 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. To avoid increasing front-end latency we assume an increase in the number of renaming ports. Once the renaming table is modified, we no longer execute the TI. As such, instructions depending on the TI result can start execution immediately. Decode-trivial instructions, once detected, do not consume execution unit resources and do not require an additional output register as the output is either zero or the NTO.

To guarantee accurate code execution under TP-advancing, after a destination register of a decode-trivial instruction is mapped to the source operand,

[2] Note that in speculating one of the source operands (i.e., the TO), sometimes (i.e., when the NTO is critical in semi-trivial instructions), we may still have to wait for the availability of the second operand (i.e., the NTO).

Table 2
Base processor configuration

Reorder buffer size	128	Unified L2	256 K, 4-way SA, 64-byte blocks, 16-cycle hit latency
Load/store queue size	64	Main memory	Infinite, 100 cycles
Scheduler	64 entries, RUU-like	Memory port #	2
Fetch unit	Up to 8 instructions/cycle 64-Entry fetch buffer	Branch predictor	16 K GShare + 16 K bi-modal w/16 K selector
OOO Core	8 instructions/cycle	Latency from branch predict to decode stage	8
L1 – instruction caches	64 K, 4-way SA, 32-byte blocks, 3 cycle hit latency	Decode & Renaming Latency	5
L1 – data caches	32 K, 2-way SA, 32-byte blocks, 3 cycle hit latency	Write-back to commit latency	6

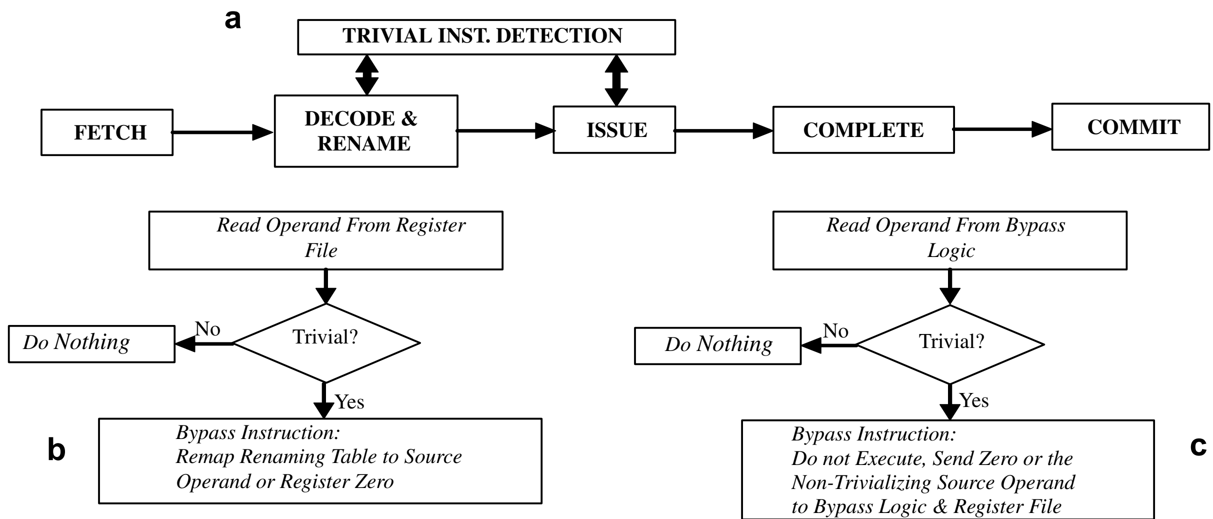

Fig. 7. (a) Schematic for a pipelined processor bypassing TIs. (b) Decode-trivial instruction detection procedure. (c) Issue-trivial instruction detection procedure.

the physical register is not released until a subsequent non-trivial instruction with the same destination register commits.

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. Therefore, we assume that issue-trivial instructions take issue slots but produce results without using the ALU.

To identify issue-trivial instructions, the TDU checks the produced operand as soon as the associated tag is received by the reservation station. Upon detecting an issue-trivial instruction, we bypass the instruction and send the result to the write-back unit (Fig. 7c).

3.2. Trivial value prediction

Previous work has introduced several value prediction techniques including last value predictor [2,3] stride predictor [4,20] context predictor [4,11,27] and hybrid predictor [13,21].

We use the context predictor [11] presented in Fig. 8. We picked this predictor as our study shows that, compared to other alternatives, this predictor provides good accuracy without imposing unnecessary overhead.

The context predictor has two tables: the Value History Table (VHT), and the Pattern History Table (PHT). Each VHT-entry records a tag, LRU, data, and value history pattern (VHP) [12]. The data field stores the four most recent unique values. LRU is an 8-bit (two bits per data value) field which determines which value in the data field

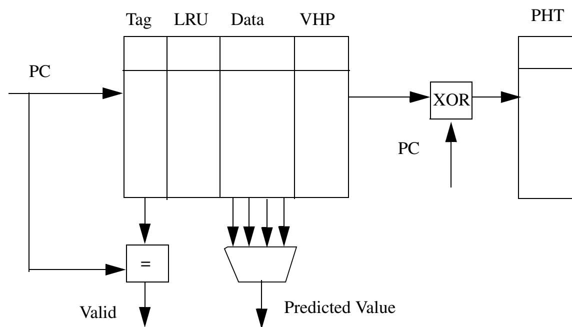

Fig. 8. Structure of a context predictor.

should be replaced if a new value arrives. VHP is also an 8-bit (two bits per data value) field that records the four most recent data values used. It is important to note that LRU and VHP fields do not store actual values. Rather they use binary encoding (00, 01, 10, and 11) to point to values in the data field. Every PHT-entry holds four 2-bit confidence counters. Every counter in PHT corresponds to one value in the VHT data filed. The confidence threshold is set to two. Among the four counters in the PHT-entry the one that has highest value is selected. If the selected entry has a value higher than the threshold the corresponding value in VHT is selected as the predicted value. To reduce the overhead we do not store data values in the VHT data field. Instead we use the 3-bit binary codes presented to represent TOs. Note that TOs are limited to 0, 1 and 0xffffffff (see Table 1). Since each instruction has two source operands we need 6 combinations (requiring 3-bits) to store both the TO and whether the operand is the first or second source operand in the instruction.

Fig. 9 shows how accurately the 128-entry context predictor with 2-bit confidence counters predicts TOs for the TP-advanced processor. As reported we are able to predict trivial values with accuracy up to 95%.

In Fig. 10 we show the architecture of a processor that predicts trivial operands. The processor fetches instructions from the instruction cache. In the next step, instructions are decoded based on their opcode and renamed to eliminate false dependency. Those instructions whose source operands are not ready wait in the issue window to receive the source operands from producer instructions. Upon functional unit availability, instructions execute. To predict the TO, we access the value predictor at the fetch stage. Since the predictor is accessed using instruction PC, we assume accessing the predictor can be done in parallel with instruction fetch/decode and would not result in a deeper pipeline front-end.

Skipping TIs requires knowing whether the output is zero or the NTO. This depends on the operation and may not be known at prediction time. To address this issue we store an extra bit per predictor entry indicating if the TI output is zero or the NTO.

While we allow speculated TIs to advance in the pipeline, we make sure they do not commit speculatively. To assure this, for speculated instructions, an extra bit (per-entry) is used to mark both the TO and the output register in the register alias table (RAT). We also mark the output register for instructions depending on a TI directly or indirectly. During register renaming, if one of the source operands of an instruction is marked as speculated we mark the output as speculated too. Marking speculated registers is necessary as future consumers should know if they are using speculated values. Upon validating a speculated operand, the associated TI and the depending instructions are allowed to commit using conventional mechanisms [9]. On a

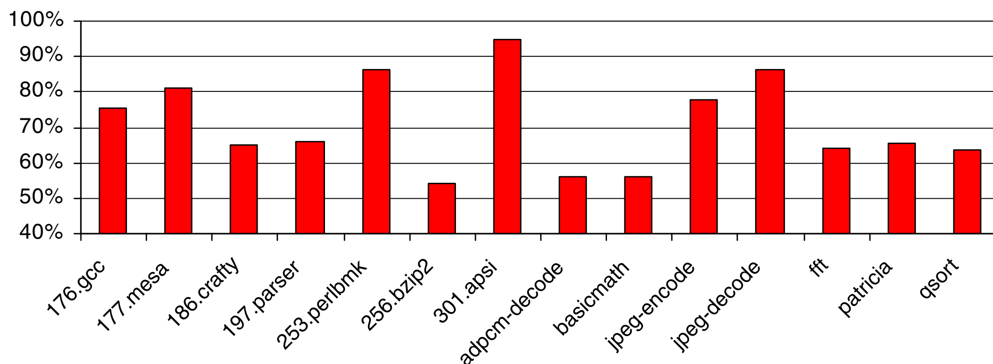

Fig. 9. Prediction accuracy for the 128-entry predictor used in this work.

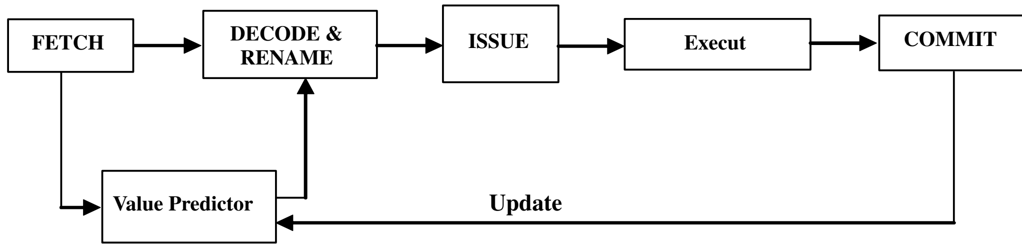

Fig. 10. TP-advanced processor architecture.

misprediction, we use selective flush [9] so that only the affected instructions are flushed. At the commit stage, the predictor is updated to reflect the accuracy of the latest predictions.

To simulate a TP-advanced processor we assume the modified reservation station presented in Fig. 11. Every entry has three tag values corresponding to source0, source1 and output. Every source operand has four fields and can be either speculated or non-speculated. To issue an instruction, the ready bit (R) of both source operands should be set. The predict bit (P) is set when the source operand is a speculated value.

We categorize instructions into three groups:

(1) Non-Speculated Instructions (N-SPEC): Instructions that do not use predicted values.
(2) Directly Speculated Instructions (D-SPEC): Instructions that have at least one speculated TO directly predicted by the value predictor.
(3) Indirectly Speculated Instructions (I-SPEC): Instructions that do not depend on a speculated TO directly but are using a speculated outcome produced by a D-SPEC or another I-SPEC instruction.

N-SPEC instructions are easy to identify as their P bits are zero and are executed in a conventional manner. D-SPEC instructions enter the issue window and wait until their predicted TO is known. These instructions are not executed as they are predicted as trivial. Later, if the predicted value is validated, D-SPEC instructions commit (as presented by the correct prediction path in Fig. 12). A validated TI informs depending instructions so they can commit. This requires forwarding the result tag with a validation signal. In the case of a misprediction, the instruction clears the P bit and sets the corresponding R bit. The instruction stays in the issue window until ALU becomes available. After execution, the instruction forwards the result tag, the result data and an invalidation signal to the instructions waiting in issue window. Depending instructions re-execute using the correct data value. Depending instructions inform I-SPEC instructions so they can re-execute too.

I-SPEC instructions may use values generated by either other D-SPEC or I-SPEC instructions. In both cases the P-bit of the speculated (directly or indirectly) source operands is set. When both source

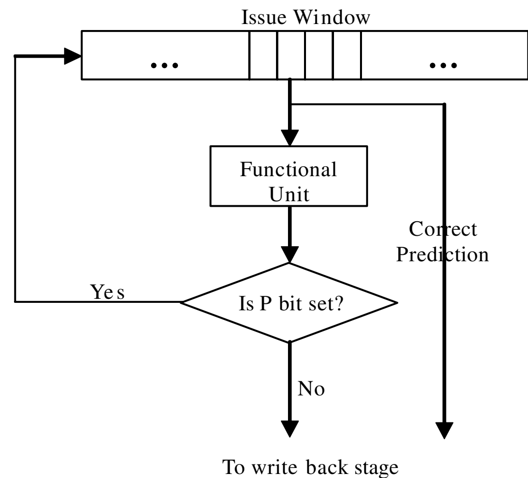

Fig. 12. D-SPEC instruction execution procedure.

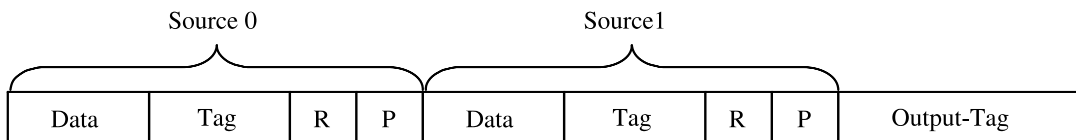

Fig. 11. Reservation station in a TP-advanced processor.

operands are ready (speculatively or no-speculatively) the instruction executes subject to resource availability. Executed I-SPEC instructions forward their result and the data tag to dependent instructions residing in the issue window. These instructions, as Fig. 12 shows, do not leave their issue window entry so long their P bit is set. They wait in the issue window until they receive the correct data. The next steps are similar to steps taken by D-SPEC instructions.

We use selective execution to resolve mispredictions [9]. We re-execute only those instructions whose outcome is affected by the misprediction values. One alternative is flushing all instructions after the misspeculated instruction. We do not use this to avoid re-executing the correctly executed instructions.

4. Methodology

We used both the SPEC CPU2000 suite and MiBench [22] benchmarks compiled for the MIPS-like PISA architecture [26] used by the Simplescalar v3.0 simulation tool set [5]. We used GNU's gcc compiler (flags: -O3-funroll-loops -finline-functions). We simulated 500 M instructions after skipping fast forward values generated by the SimPoint toolkit [29]. We use an aggressive 8-way superscalar processor. The processor is deeply pipelined to reflect modern processors. We detail the base processor model in Table 2.

5. Results

To evaluate TP-advancing, we compare our processor to a conventional processor. To show that advancing TP improves performance, we also compare to a processor which bypasses TIs detected at their original TP. We refer to this processor as the *TP-original processor*. Also, to provide better understanding, we report how TP-advancing impacts average instruction issue delay. We also investigate the overhead associated with TP-advancing and compare performance improvement for TP-advancing to a processor using conventional value prediction. In addition and to provide better insight, we evaluate trivial instructions frequency in a processor with higher execution bandwidth.

Our TP-advanced processor relies on a 128-entry context value predictor. We selected this predictor after testing alternative configurations. Our study shows that further increases in the predictor size do not impact performance considerably. Every VHT-entry in the context predictor has 54 bits (25 bits tag, 4×2 bits LRU, 4×3 bits data, 4×2 bits value history pattern, and 1 bit output is zero or NTO). Every PHT-entry has 4×2 bits. Therefore the total predictor size of a 128-entry predictor is less than 8 K ($128 \times (54 + 8)$) bits.

5.1. Performance

In Fig. 13 we report performance improvements achieved by TP-advancing over a conventional processor. We also report the performance improvement achieved by the TP-original processor. As reported, TP-advancing improves performance across all benchmarks over the base (between 11% and 30%) and the TP-original processor (between 1% and 5%).

In Fig. 14 we report average instruction issue delay reduction for the TP-original processor and

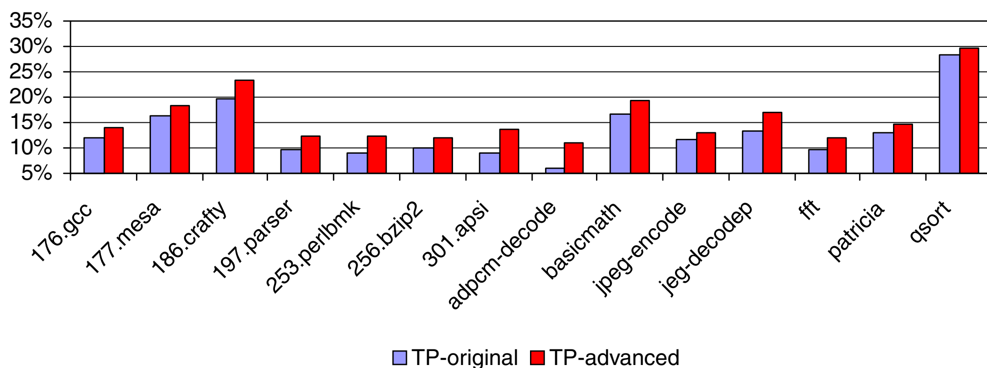

Fig. 13. Bars from left to right report performance improvement achieved by TP-original and TP-advancing over a conventional processor.

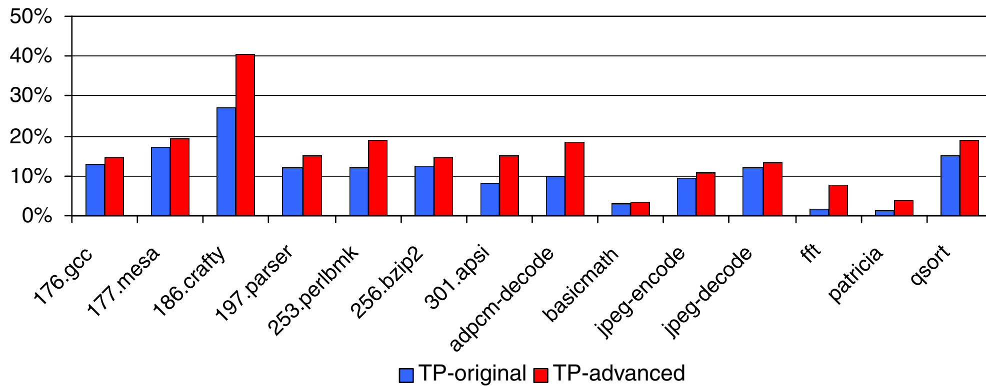

Fig. 14. Issue delay reduction by TP-original and TP-advanced compared to a conventional processor.

TP-advancing. Issue delay is the time instructions wait in the issue window before their source operands and the required functional units become available. By bypassing TIs, TI outcomes become available sooner. Consequently, dependent instructions may not need to wait in the issue window as much as they wait in a conventional processor. TP-advancing impacts issue delay further as it makes detecting a higher number of TIs possible.

5.2. Discussion

In this section we discuss how different applications react to TP-advancing. We consider TI frequency, issue- and decode-trivial distribution, the percentage of critical trivial operands and prediction accuracy.

Applications could be categorized to the following two groups based on their performance improvement under TP-advancing compared to a conventional processor.

1. The first group includes those showing higher performance improvements (i.e., more than 15%). This group includes *mesa, crafty, jepg-decode, qsort, basicmath* and *patricia*. *Qsort, patrica* and *basicmath* have high number of TIs which explains why they belong to this group (see Fig. 2). *Mesa, crafty* and *jpeg-decode*, however, have a lower number of TIs when compared to the other three. To explain why *mesa, crafty* and *jpeg-decode* perform so well under TP-advancing we take into account the percentage of issue-trivial and critical TOs presented in Figs. 4 and 5. A high percentage of TOs in *mesa* and *crafty* are critical. This could explain the high-performance improvement. As for *jpeg-decode,* a high percentage of TIs are issue-trivial which may explain why this application has high performance improvement.

A closer look at Fig. 13 reveals that among this group *qsort* shows a relatively small gap between TP-advancing and the TP-original processor. The fact that *qsort* has the lowest number of critical TOs among this group explains this. Note that if a TO is not critical, TP-advancing hardly results in faster TI execution.

2. The second group includes benchmarks that still offer acceptable performance but fall behind the first group. We include *gcc, parser, perlbmk, bzip2, apsi, adpcm-decode, jpeg-encode,* and *fft* in this group as their performance improvements are less than 15% compared to a conventional processor.

Parser, bzip2, apsi and *adpcm-decode* have fewer TIs compared to the first group. *Fft* has fewer TIs compared to all applications in the first group but *jpeg-decode, crafty* and *mesa*. *Fft*, however, has fewer critical TOs compared to *mesa* and *crafty* (see Fig. 5) and fewer issue-trivials (see Fig. 6) and lower accuracy (see Fig. 9) compared to *jpeg-decode*. The fact that critical TOs are less frequent in *gcc, perlbmk* and *jpeg-encode* compared to *crafty* could explain why the three fall behind *crafty* in performance while having higher number of TIs.

Note that *adpcm-decode* has a low TI frequency compared to the other applications included in this group. Nonetheless, it exhibits high performance improvement. This is consistent with the fact that *adpcm-decode* has a very high number of critical TOs. As presented in Fig. 5 almost all TOs are critical for *adpcm-decode*. As a result, when compared to the TP-original processor, *adpcm-decode* sees a performance improvement better than other benchmarks (see Fig. 13).

5.3. Overhead evaluation

In this section we investigate whether processor resources could be better used for other purposes. To investigate this, we study performance improvements achieved, if, instead of using TP-advancing, a bigger cache or a larger branch predictor is used. Our base processor configuration is presented in Table 2.

In Fig. 15 we report performance improvements for TP-advancing, a processor using a twice as big gshare branch predictor (BR-processor) and a processor using a cache blocks twice as big (CA-processor). This processor uses 64-entry cache blocks instead of the 32 entries used by the base processor. Note that both the BR and CA processor bypass TIs but do not use value prediction to advance TP.

The size of 128-entry context predictor is less than 8 K bits. Doubling the 16 k-entries gshare branch predictor and using a twice as big cache block, comes with 32 K bit and 32 K byte space overhead, respectively. Therefore the real-estate used by the BR- and CA-processor is 4 times and 32 times that used by TP-advancing. Also, our simulation with CACTI [32] shows that the power overhead associated with TP-advancing is 65% and 23% of the power overhead associated with BR- and CA-processors, respectively. Despite this, for most applications, TP-advancing provides better performance improvement compared to both the BR- and the CA-processor.

5.4. TP-advanced vs. conventional value prediction

In Fig. 16, we report performance improvements achieved by TP-advancing and conventional value prediction over the baseline processor. In the conventional method, source operands of all instructions are speculated. We use the context predictor with the same size for both TP-advanced and the

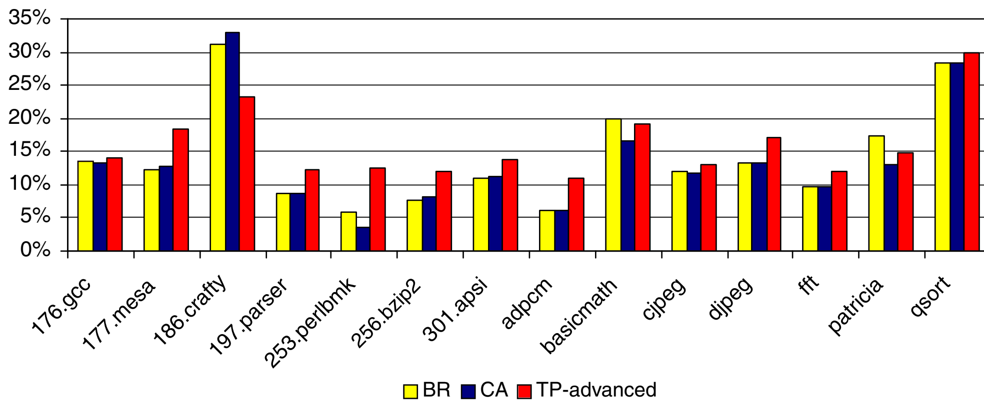

Fig. 15. Performance improvement for BR, CA, and the TP-advanced processor.

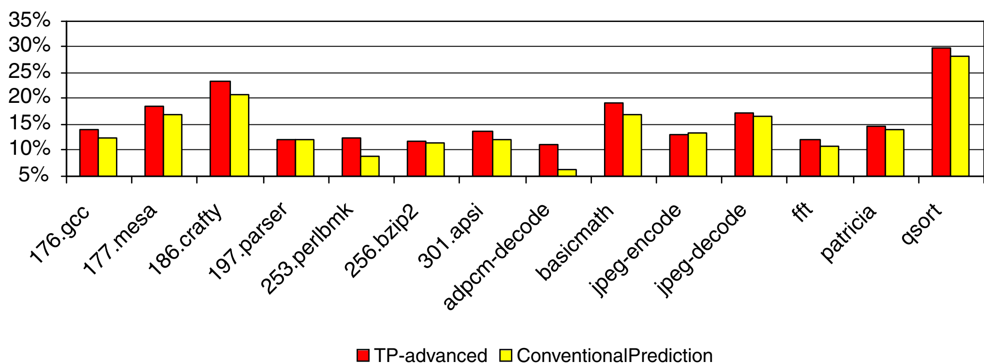

Fig. 16. Bars from left to right report performance improvements achieved by TP-advancing and conventional value prediction over the baseline processor.

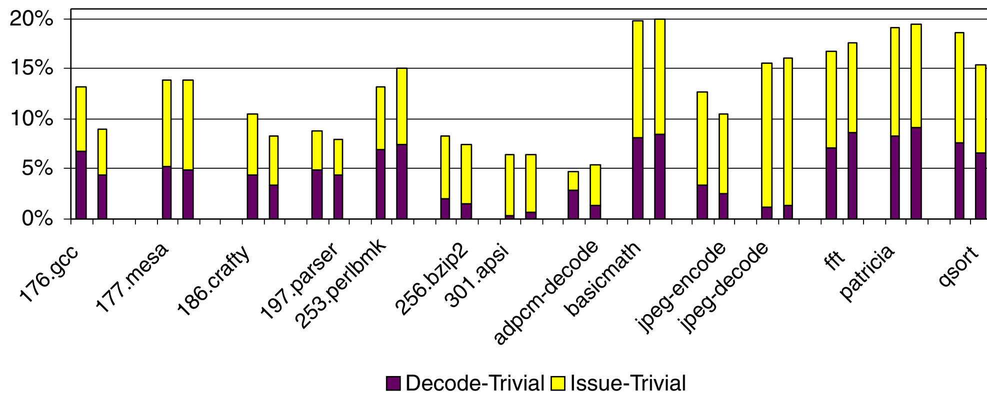

Fig. 17. Sensitivity of trivial instructions to processor execution bandwidth. For each benchmark, the left and right bar report trivial instruction frequency for 8-way and 16-way processors, respectively.

conventional prediction. Generally, Conventional method falls behind TP-advancing. This is mainly due to the fact that TP-advancing uses value prediction selectively and only for those operands that have high locality. Conventional value prediction, on the other hand, speculates source operands of all instructions. This results in less accurate and therefore less reliable predictions.

5.5. Impact of issue width

In this section we discuss how trivial instruction distribution changes as we increase processor execution bandwidth. Fig. 17 reports trivial instruction frequency for 8-way and 16-way processors. The lower and upper part of each bar report decode- and issue-trivial instruction frequency, respectively. For both processors, the majority of trivial instructions are issue-trivial. We conclude from Fig. 17 that decode- and issue-trivial instruction's frequency does not heavily depend on processor execution bandwidth.

6. Related work

Molina et al. [23] and Sodani and Sohi [24] used value reuse to eliminate repetitive computations. Our work is different as it does not rely on value reuse. Instead we take into account triviality to skip unnecessary computations.

Richardson [31] shows that detecting and eliminating trivial instructions dynamically can reduce the program's execution time. Yi and Lilja [7] extend their work by extending the scope of trivial instructions. Tran et al. [14] evaluated dynamic methods to reduce pressure on the register file. They explored the impact of bypassing trivial instructions on the register file pressure.

In our earlier work [25], we studied the effect of trivial bypassing on power dissipation in high-performance processors. We investigated power saving in different units and showed that ALU, register file and issue window benefit most from trivial bypassing.

Our work is different from the above studies. We use value prediction to advance trivialization point and to identify and execute TIs as early as possible. By using speculation, we improve performance by identifying a larger number of TIs and at an earlier stage. We also extend trivial bypassing to issue-trivial instructions. We provide a detailed study of TIs and explain how TIs can be categorized based on their triviality kind, trivialization point, and TO criticality. Moreover, we discussed implementation details which were not studied in any of the above studies.

Many studies have suggested different value prediction techniques. These techniques include last value prediction ([2,3]) stride prediction ([8,10]) context prediction ([4,11]), and hybrid predictors ([12,13,28]). In this work, we used the context predictor, as our study showed that other predictors are either less accurate or are not worth the additional complexity.

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 ([11–13]) there are a few which have addressed the cost, i.e., access latency and energy,

associated with value prediction (e.g., [6]). 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 TOs as we have observed that TOs make good candidates for selective value prediction since they show strong value locality and appear frequently.

Bhargava and John [6] 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.

Sato and Arita [15] 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). Their low-cost 0/1 predictors could potentially be used as attractive TO predictors as TOs mostly consist of 0 and 1.

7. Conclusions

In this work we introduced speculative trivialization point advancing to break dependencies and increase the number of available resources in high-performance processors. We used value prediction to speculate the trivializing operand of TIs and improved performance over a conventional processor and a processor that bypasses TIs without speculating the operands.

Furthermore, we studied TI locality, behavior, frequency and distribution. We showed that trivial operands are attractive candidates for value prediction as they show higher data locality.

We also suggested an efficient hardware implementation to detect, predict and bypass TIs. We reviewed how bypassing predicted TIs improves performance for applications from the SPEC2k and MiBench. We also discussed how applications react to our technique and compared our method to a conventional value prediction scheme that uses the same hardware budget to speculate source operands for all instructions.

We believe that TP-advancing and trivial computation bypassing can be exploited in other design spaces and at alternative levels. One possible future research avenue is investigating trivial functions in programs. By detecting and bypassing such functions the processor can run programs faster and with spending less energy. We also believe that due to its low overhead, bypassing trivial computations can also be used in the embedded space and for processors similar to intel's XScale [30].

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program and Canada Foundation for Innovation, New Opportunities Fund.

References

[1] B. Calder, G. Reinman, D.M. Tullsen, Selective value prediction, in: 25th International Symposium on Computer Architecture, May 1999, pp. 64–74.
[2] M.H. Lipasti, J.P. Shen, Exceeding the data flow limit via value prediction, in: 29th International Symposium on Microarchitectures, December 1996, pp. 226–237.
[3] M.H. Lipasti, C.B. Wilkerson, J.P. Shen, Value locality and load value prediction, in: 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996, pp. 138–147.
[4] Y. Sazeides, J.E. Smith, The predictability of data values, in: 30th International Symposium on Microarchitecture, December 1997, pp. 248–258.
[5] D. Burger, T.M. Austin, S. Bennett, Evaluating Future Microprocessors: The SimpleScalar Tool Set, Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.
[6] R. Bhargava, L. John, Latency and energy aware value prediction for high-frequency processors, in: Proceedings of 16th ACM International Conference on Supercomputing, June 2002, pp. 45–56.
[7] J.J. Yi, D.J. Lilja, Improving Processor Performance by Simplifying and Bypassing Trivial Computations, in: IEEE International Conference on Computer Design: VLSI in Computers and Processors, Germany, August 2002, pp. 462–465.
[8] F. Gabbay, A. Mendelson, The effect of instruction fetch bandwidth on value prediction, in: 25th International Symposium on Computer Architecture, June 1998, pp. 272–281.
[9] B. Rychlik, J. Faistl, B. Krug, J.P.Shen, Efficacy and performance impact of value prediction, in: International

Conference on Parallel Architectures and Compilation Techniques, October 1998, pp. 148–154.
[10] F. Gabbay, A. Mendelson, Speculative execution based on value prediction, Technical Report 1080, Technion – Israel Institute of Technology, November 1996.
[11] K. Wang, M. Franklin, Highly accurate data value prediction using hybrid predictors, in: 30th International Symposium on Microarchitecture, December 1997, pp. 281–290.
[12] S. Lee, Y. Wang, P. Yew, Decoupled value prediction on trace processors, in: 6th International Symposium on High Performance Computer Architecture, January 2000, pp. 231–240.
[13] B. Rychlik, J.W. Faistl, B.P. Krug, A.Y. Kurland, J.J. Sung, M.N. Velev, J.P. Shen, Efficient and accurate value prediction using dynamic classification, Technical report, Carnegie Mellon University, 1998.
[14] L. Tran, N. Nelson, F. Ngai, S. Dropsho, 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.
[15] T. Sato, 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.
[16] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1997) 337–343.
[17] D.A. Huffman, A method for the construction of minimum redundancy codes, in: Proceedings of the Institute of Radio Engineers, vol. 40, Number 9, September 1952, pp. 1098–1101.
[18] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kauffman, San Francisco, CA, 2003.
[19] G. Sohi, Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers, IEEE Transactions on Computers 39 (1990) 349–359.
[20] R. Eickemeyer, S. Vassiliadis, A load instruction unit for pipelined processors, IBM Journal of Research and Development 37 (1993) 547–564.
[21] G. Reinman, B. Calder, Predictive techniques for aggressive load speculation, in: 31st International Symposium on Microarchitecture, Dallas, Texas, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 127–137.
[22] M. Guthuas, J. Ringenberg, D. Ernst, T. Austin, T. Mudget, R. Brown, MiBench: a free, commercially representative embedded benchmark suite, in: IEEE 4th Annual Workshop on Workload Characterization (WWC-4), December 2001.
[23] C. Molina, A. González, J. Tubella, Dynamic removal of redundant computations, in: Proceedings of the 13th international conference on Supercomputing, NY, USA, 1999, pp. 474–481.
[24] A. Sodani, G. Sohi, Dynamic Instruction Reuse, in: International Symposium on Computer Architecture, ACM Press, New York, NY, 1997, pp. 474–481.
[25] E. Atoofian, A. Baniasadi, Improving energy-efficiency in high-performance processors by bypassing trivial instructions, in: the IEE Proceedings Computer and Digital Techniques, vol. 153, Issue 5, September 2006, pp. 313–322.
[26] Kenneth C. Yeager, The MIPS R10000 Superscalar Microprocessor, IEEE Micro 16 (2) (1996) 28–40.
[27] B. Goeman, H. Vandierendonck, K. de Bosschere, Differential FCM: increasing value prediction accuracy by improving table usage efficiency, in: 7th International Symposium on High Performance Computer Architecture, January 2001, pp. 207–216.
[28] Martin Burtscher, Benjamin G. Zorn, Hybrid load-value predictors, IEEE Transactions on Computers 51 (7) (2002) 759–774.
[29] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in: 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.
[30] L.T. Clark, E.J. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K.E. Velarde, M.A. Yarch, An embedded 32-bit microprocessor core for low-power and high-performance applications, IEEE Journal of Solid-State Circuits 36 (11) (2001) 1599–1608.
[31] S. Richardson, Caching function results: faster arithmetic by avoiding unnecessary computation, in: International Symposium on Computer Arithmetic, 1993.
[32] P. Shivakumar, N. Jouppi, CACTI 3.0: An Integrated Cache Timing, Power and Area Model, Technical Report 2001/2, Compaq Computer Corporation, August 2001.