Improving energy-efficiency in high-performance processors by bypassing trivial instructions

E. Atoofian and A. Baniasadi

Abstract: Energy-efficiency benefits of bypassing trivial computations in high-performance processors are studied. Trivial computations are those computations whose output can be determined without performing the computation. Bypassing trivial instructions reduces energy consumption while improving performance. The present study shows that by bypassing trivial instructions and for the subset of SPEC'2K and MiBench benchmarks studied here, it is possible to improve energy and energy-delay up to 15.6 and 30.6%, respectively, in an optimistic scenario and by 10.8 and 21.7% in a pessimistic scenario, over a conventional processor.

1 Introduction

In this work, we improve the energy-efficiency of 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 an 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 instruction set architecture, where each instruction may have up to two source operands. We refer to the operand that trivialises the operation as the trivialising operand (TO). Examples of TOs are the operand equal to zero in an add operation or the operand equal to one in a multiplication.

It has been shown that an optimising compiler is often unable to remove trivial operations, as trivial values are not known at compile time, and the amount of trivial computations does not heavily depend on program specific inputs [1].

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, for example, 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, for example, 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 instructions. 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 on both the operation and the particular source operand value that trivialises 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, although unnecessary, leads to extra latency and additional energy consumption. Therefore bypassing the computation and obtaining the result without performing the computation should 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.

- We show that, by bypassing trivial instructions, it is possible to reduce average energy consumption and improve energy-delay considerably. We study both optimistic and pessimistic timing/complexity scenarios and show that energy and energy-delay can be improved under both scenarios [energy: 15.6% (optimistic), 10.8% (pessimistic), energy-delay: 30.6% (optimistic), 21.7% (pessimistic)].
- We investigate trivial instructions in detail. We categorise 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

IEE Proceedings online no. 20050084

doi:10.1049/ip-cdt:20050084

Paper first received 22nd April 2005 and in final revised form 16th January 2006

The authors are with the Electrical & Computer Engineering Department, University of Victoria, 3800 Finnerty Rd., Victoria, British Columbia, Canada V8P 5C2

E-mail: amirali@ece.uvic.ca

Table 1: Fully- and semi-trivial instructions studied in this work

Operation	Fully 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: $A \ll B$, $A \gg B$	$A = 0$
Arithmetic shift: $A \ll B$, $A \gg B$	$A = 0$
Operation	**Semi triviality condition**
Addition: $A + B$	$A = 0$ or $B = 0$
Subtraction: $A - B$	$B = 0$ or $A = B$
Multiplication: $A * B$	$A = 1$ or $B = 1$
Division: A/B	$B = 1$
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	$A =$ 0x00000000 or $B =$ 0x00000000
Logical shift: $A \ll B$, $A \gg B$	$B = 0$
Arithmetic shift: $A \ll B$, $A \gg B$	$B = 0$

instructions belong to each category and how this may impact our energy and performance improvements.

2 Trivial instruction bypassing

2.1 Trivial instruction example

To provide better insight into how bypassing trivial instructions can result in better performance and energy consumption, in Fig. 1, we present an example picked from the 164.gzip benchmark. 164.gzip divides input streams into blocks and compresses each block separately by using a combination of the LZ77 algorithm [2] and Huffman coding [3]. LZ77 generates an output file, which is later used by Huffman coding to produce the final compressed file. The 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, 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. By using Huffman coding, we further compress the input file.

Fig. 1*a* shows the *deflate( )* function. This function generates compressed data by using LZ77 and Huffman coding. Inside of this function, the *ct_tally( )* function is called. *Ct_tally( )* saves the matched information and measures 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. 1*c*, we show a part of the assembly code corresponding to the *ct_tally( )* function.

The first instruction in Fig. 1*c* loads *length_code[lc]* into *R[2]* register. The Huffman code corresponding to length 3 is equal to 0. This value does not change within a block. Whenever a matching with length 3 is detected in the input stream, *lbu* loads the same value (zero) into register *R[2]*, and *addiu* instruction adds zero with 257. By using trivial bypassing, the outcome of *addiu* instruction is determined earlier, as we do not spend time on executing the trivial instruction. All the instructions following *addiu* depend directly or indirectly on the value of *R[3]*, which is the outcome of *addiu* instruction. Consequently, by bypassing *addiu* instruction, dependent instructions can execute earlier. Also, *addiu* no longer needs arithmetic logic unit (ALU) resources, leaving ALU resources to instructions, which may otherwise be stalled due to structural hazards. We also save energy by bypassing *addiu*.

2.2 Trivial instruction frequency and distribution

The result of a trivial operation could be either one of the source operands or zero (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 Fig. 2, 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 the frequency of fully-trivial instructions.

As represented by the entire bar, on average, trivial instructions account for about 23% of the total instructions. *Jpeg-decode* and *vpr* have higher number of trivial instructions compared to others. *Swim* has the least number of trivial instructions.

In general, semi-trivial instructions outnumber fully-trivial instructions. Fully-trivial instructions may account for as much as 23% of the total number of trivial instructions (e.g. *swim*). Meantime, they may account for as little as 2% of the total trivial instructions (e.g. *apsi*). On average, about 89% of the trivial instructions are semi-trivial, and the remaining 11% 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.

Fig. 3 reports how often each instruction type is trivial. Similar to Fig. 2, for every instruction type, we also report semi- and fully-trivial frequencies. Note that, as presented in Table 1, not all instruction types can be fully-

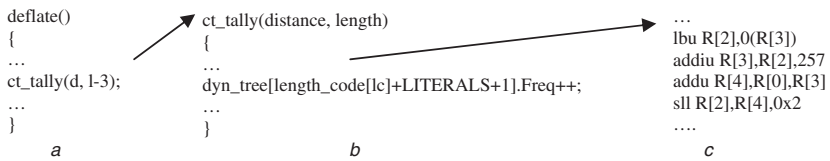


Fig. 1 *Part of 164.gzip benchmark*

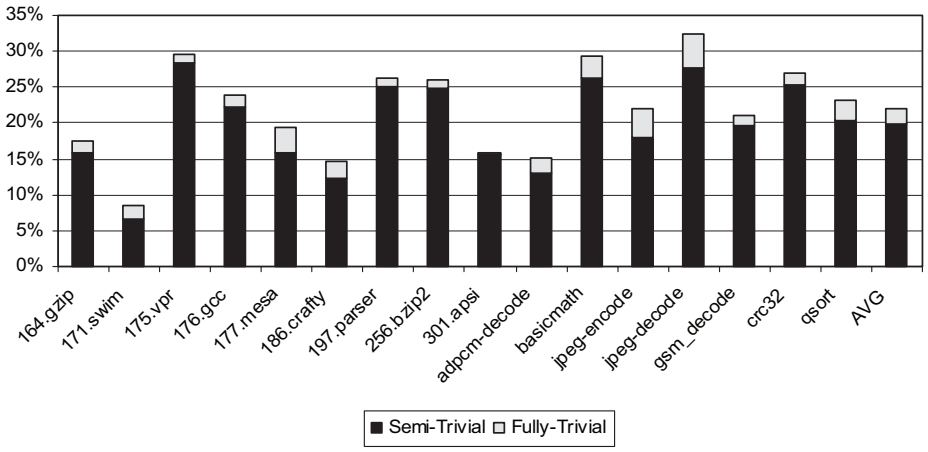


Fig. 2 *Trivial instruction frequency and distribution*

Entire bar represents trivial instruction frequency
Lower part shows semi-trivial instruction frequency and the upper part shows fully-trivial instruction frequency

trivial. In particular, *addition*, *subtraction* and *XOR* instructions cannot be fully-trivial. For these instructions both operands are required to decide the outcome. Therefore as presented in Fig. 3, all trivial instructions for these instruction types are semi-trivial.

On average, at least 20% of each instruction type is trivial. In cases such as *or* and *fadd*, trivial instructions account for more than half of the instructions. For instruction types that can be fully-trivial, fully-trivial instruction frequency changes from 3% in *fdiv* to 51% in *or*. 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 performance. For example, although 70% of *or* instructions 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 fully-trivial) are known. On the basis of the operand(s) availability time(s), we categorise trivial instructions into two groups.

The first group of instructions are those instructions 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 required source 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 Fig. 4, we report the percentage of decode-trivial and issue-trivial instructions. While the entire bar represents total trivial instructions (similar to Fig. 2), the lower part of each bar shows the frequency of decode-trivial instructions and the upper part represents the frequency of issue-trivial instructions.

As presented, issue-trivial instructions account for the majority of the trivial instructions for most benchmarks. However, for some benchmarks (e.g. *swim* and *adpcm-decode*), the number of decode-trivial instructions exceeds issue-trivial instructions.

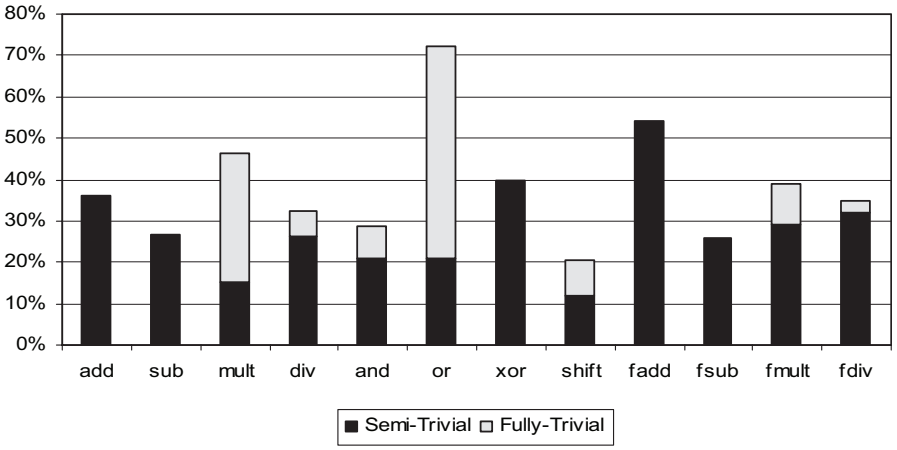


Fig. 3 *How often each instruction type is trivial?*

Lower part shows semi-trivial instruction frequency and the upper part shows fully-trivial instruction frequency

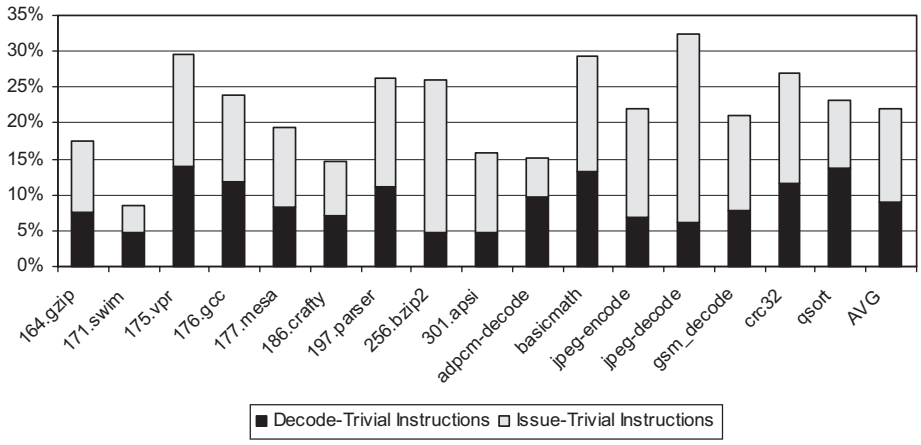


Fig. 4 *Trivial instruction frequency and distribution*

Entire bar represents trivial instruction frequency
Lower part shows decode-trivial instruction frequency and the upper part shows issue-trivial instruction frequency

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

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) [4]. 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 [5]. While trivial instruction bypassing could be used on top of both implementations, here we assume the former.

Fig. 5 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 Fig. 5*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 immediately (subject to resource

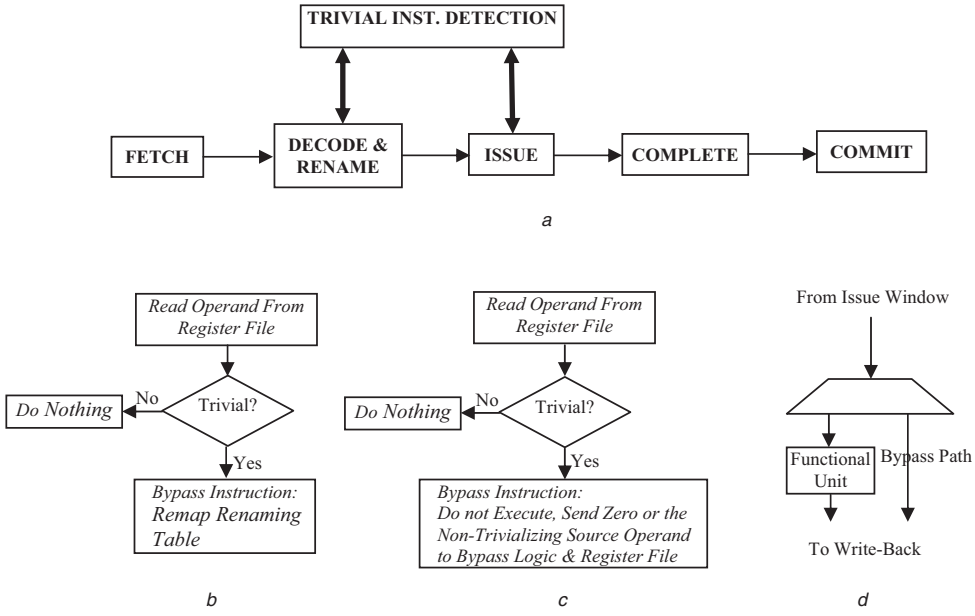


Fig. 5 *Trial instruction detection*

a Schematic for a pipelined processor bypassing trivial instructions
b Decode-trivial instruction detection procedure
c Issue-trivial instruction detection procedure
d Bypass structure for issue-trivial instructions

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 Fig. 5*c* and depicted in Fig. 5*d*. However, the destination register of an issue-trivial instruction should not be released, as there may still be instructions depending on the trivial instruction outcome which have not read their source operands yet.

Detecting and bypassing trivial instructions comes with little hardware overhead. This is due to the fact that many already available resources can be used to implement trivial instruction detection and bypassing. Detecting trivial instructions requires decoding the instruction type and checking the triviality condition based on the instruction type (as presented in Table 1). The former is already being done at the decode stage. Therefore we do not need additional hardware. The latter, that is, checking the trivial instruction triviality condition, requires checking if any of the source operands are 0, 1, 0xFFFFFFFF, or if the two source operands are equal. This can be done by using NOR, NOR with one inverted input, NAND, and XOR gates, respectively.

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 with decode-trivial instructions.

While this study focuses on superscalar processors, it is important to note that trivial instruction bypassing is applicable in the scalar space too. Scalar processors may benefit from our technique as detecting and bypassing trivial instructions will provide data for dependent instructions sooner. This leads to earlier execution of dependent instructions. Moreover, as the bypassed trivial instructions do not consume execution units, we would expect that ALU structural hazards occur less frequently in the scalar processor. On the other hand, and as a possible negative impact, structural hazards may increase in the register file as there is a possibility that the bypassed trivial instruction and other completing instructions may require to write to the register file simultaneously.

4 Methodology

In this section, we report our analysis framework. We used both SPEC CPU2000 [6] suite and MiBench [7] benchmarks compiled for the MIPS-like PISA architecture used by the Simplescalar v3.0 simulation tool set [8]. The benchmark set studied here includes different programs with different control flow regularities. We report branch prediction rate for each benchmark in Table 2.

We used Wattch [9] for energy estimation. Wattch is an architectural-level simulation tool and is used to estimate energy consumption. In Wattch, the energy consumed by major components is modelled on the basis of architectural parameters. For example, the energy consumed by accessing the register file is measured using architectural parameters such as the number of registers and the register width. Wattch counts the number of accesses to each component. The energy consumption of each component is calculated by multiplying the number of accesses by the energy consumed per access.

Table 2: Branch prediction rate in studied benchmarks

Benchmarks	Branch prediction rate
164.gzip	91.17
171.swim	96.85
175.vpr	93.02
176.gcc	89.5
177.mesa	93.58
186.crafty	93.43
197.parser	93.22
256.bzip2	95.23
301.apsi	98.55
adpcm-decode	81.22
basicmath	93.01
jpeg-encode	95.21
jpeg-decode	95.36
gsm_decode	98.57
crc32	99.99
qsort	98.09

A complete timing/complexity analysis requires detailed simulations of the processor, which is beyond this work. However, to provide insight, we report results for two extreme scenarios. In the first (optimistic) scenario, we assume that the complexities associated with bypassing issue-trivial instructions are negligible. This provides an upper bound for energy and performance improvements achieved. In the second (pessimistic) scenario, we assume that such complexities disallow bypassing issue-trivial instructions. Under this scenario, we report results achieved only by bypassing decode-trivial instructions.

We use GNU's gcc v2.9 with –O3 optimisation level. We simulated 500M instructions after skipping 500M instructions. We simulated an aggressive 8-way superscalar processor. The processor is deeply pipelined to reflect modern processors. We detail the base processor model in Table 3.

5 Results

To evaluate how bypassing trivial instructions impacts performance and energy, we compare our processor with a conventional processor that does not bypass trivial instructions. In Section 5.1, we report performance improvement. We also report issue-delay reduction to provide better insight. In Section 5.2, we report energy, energy-delay and energy breakdown. We report sensitivity analysis in Section 5.3.

5.1 Performance improvement

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 Fig. 6, we report performance improvements compared with a conventional processor. For each application, the entire bar reports results under the optimistic scenario where both issue- and decode-trivial

Table 3: Base processor configuration

Reorder buffer size	128
Load/store queue size	64
Scheduler	64 entries, RUU-like
Fetch unit	Up to 8 instructions/cycle 64-entry fetch buffer
OOO core	8 instructions/cycle
L1 - instruction caches	64 K, 4-way SA, 32-byte blocks, 3 cycle hit latency
L1 - data caches	32 K, 2-way SA, 32-byte blocks, 3 cycle hit latency
Unified L2	256 K, 4-way SA, 64-byte blocks, 16-cycle hit latency
Main memory	Infinite, 80 cycles
Memory port number	2
Branch predictor	16 K GShare + 16 K bi-modal w/16 K selector
Latency from branch predict to decode stage	8
Latency of decode and renaming	5
Latency of write-back to commit	6

instructions are bypassed. The lower part of each bar reports for the pessimistic scenario, where only decode-trivial instructions are bypassed. On average, performance improvements are 22.7% under the optimistic and 18% under the pessimistic scenario. *Vpr* and *qsort* show higher performance improvements compared with other benchmarks. *Swim* has the lowest performance improvement among all benchmarks.

Performance improvements achieved can be the result of faster instruction execution (by breaking the dependency chains) or improving resource utilisation (by avoiding using functional units for trivial instructions). Our study shows that, for the configuration and applications used in this study, more than 90% of the performance improvement achieved is due to faster execution of trivial instructions.

To provide better insight, in Fig. 7, we report instruction issue-delay reduction for a processor that bypasses trivial instructions for both complexity scenarios. Note that by bypassing trivial instructions, trivial instruction outcomes become available sooner. Consequently, dependent instructions may not need to wait in the issue window as long as they have to wait in a conventional processor. As reported, by skipping trivial instructions, on average, instructions spend 20.2 and 13.6% less time in the issue window for optimistic and pessimistic scenarios, respectively.

5.2 Energy, energy-delay and energy breakdown

In Figs. 8*a* and *b*, we report energy and energy-delay measurements as reported by Wattch [9], respectively. Similar to Figs. 6 and 7, the entire bar reports results for the optimistic scenario and the lower part of each bar reports for the pessimistic scenario. On average, energy savings are 15.6% under the optimistic and 10.8% under the pessimistic scenario. As reported in Fig. 8*a*, *vpr* and *qsort* have higher energy reduction compared with the rest of the benchmarks.

In Fig. 8*b*, we report energy-delay improvements achieved by bypassing trivial instructions. On average, energy-delay improvements are 30.6% under the optimistic and 21.7% under the pessimistic scenario. Again, *vpr* has the highest energy-delay improvement among all benchmarks.

Note that *jpeg-decode* has the highest percentage of trivial instructions (Fig. 2), but does not show the highest energy improvement. The low number of decode-trivial instructions in *jpeg-decode* may explain this. *Qsort* has considerable energy improvement despite its moderate trivial instruction percentage. This may be the result of high number of decode-trivial instructions.

In Fig. 9, we show how bypassing trivial instructions can potentially impact energy for each component by reporting energy reduction breakdown over different components for the optimistic scenario. Components benefit differently from trivial bypassing. The highest energy saving is achieved in ALU. This is expected as ALU is not used for trivial instructions. Register file comes second. Decode-trivial instructions do not write their result to the register file. They use remapping to eliminate writing to the register file. Decode-trivial instructions also reduce the energy consumed by the issue window and the result bus. This is due to the fact that decode-trivial instructions are identified at the dispatch stage and skip later stages. As such they do not use the issue window and the result bus.

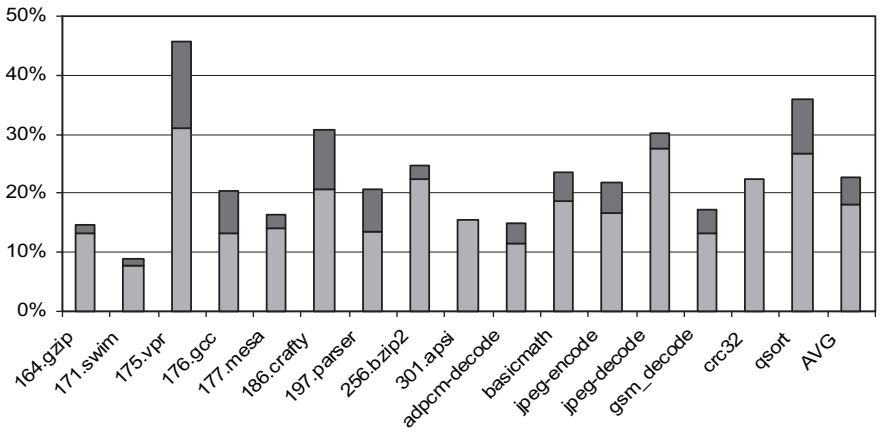


Fig. 6 *Performance improvement achieved by bypassing trivial instructions over a conventional processor*

The entire bar reports performance improvement achieved under the optimistic scenario (i.e. both issue- and decode-trivial instructions are bypassed) where the lower bar reports for a pessimistic scenario (i.e. only decode-trivial instructions are bypassed)

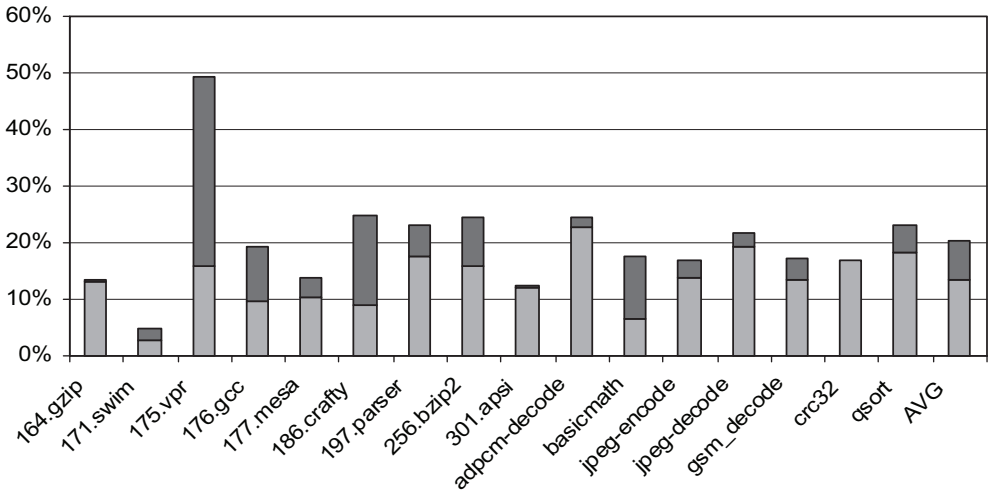
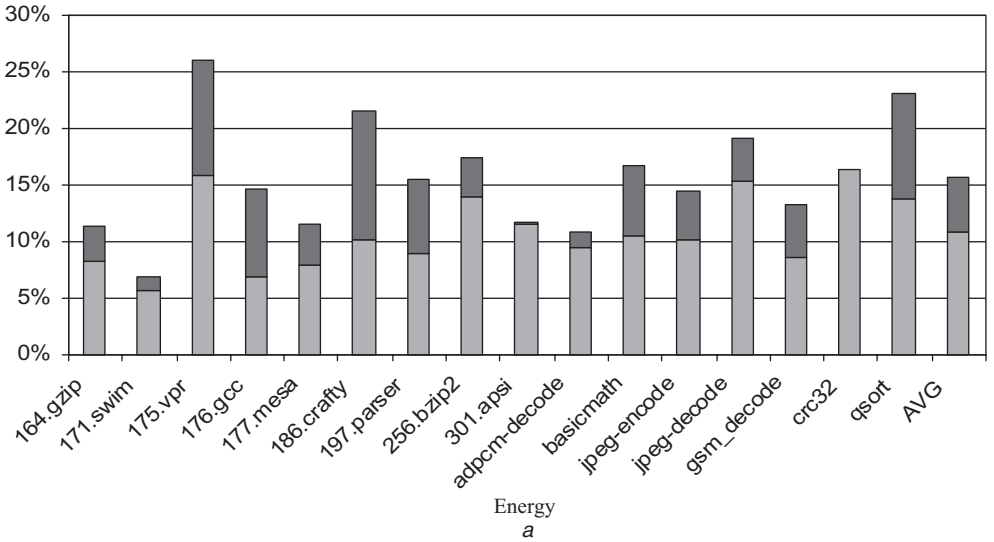
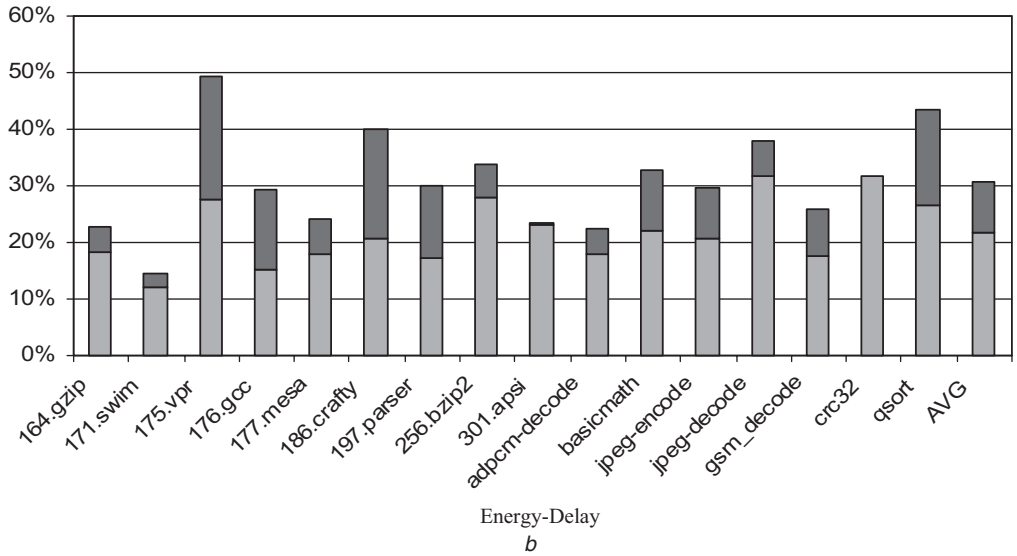


Fig. 7 *Issue delay reduction achieved by bypassing trivial instructions over a conventional processor*

The entire bar reports issue-delay reduction achieved under the optimistic scenario (i.e. both issue- and decode-trivial instructions are bypassed) where the lower bar report issue-delay reduction for a pessimistic scenario (i.e. only decode-trivial instructions are bypassed)



a



b

Fig. 8 *Bypassing trivial instructions over a conventional processor*

a Energy improvement
b Energy-delay improvement
The entire bar reports results assuming the optimistic scenario (i.e. both issue- and decode-trivial instructions are bypassed)
The lower bar reports results assuming a pessimistic scenario (i.e. only decode-trivial instructions are bypassed)

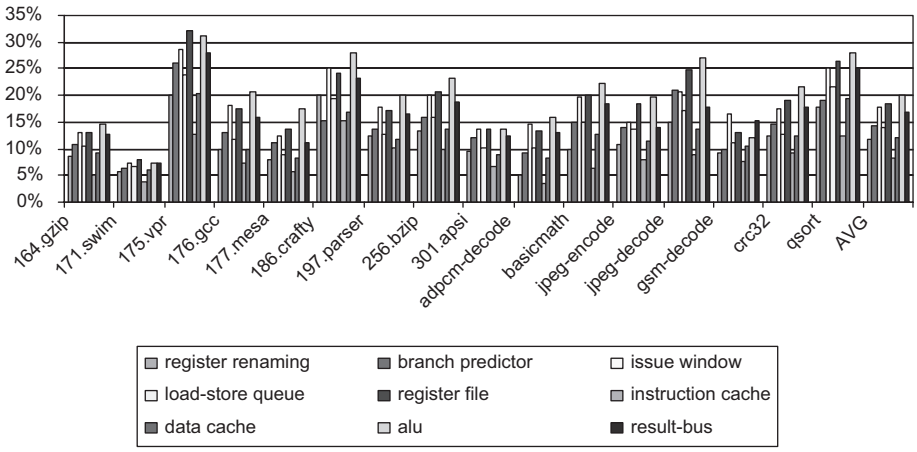


Fig. 9 *Energy breakdown: the amount of energy reduction in different components*

Other components mostly benefit from trivial bypassing as the result of shorter execution time.

Energy savings could be the result of shorter runtimes (by breaking the dependency chains) or avoiding using functional units to execute trivial instructions. Our study shows that, for the configuration and applications used in this study, more than 93% of the energy savings achieved is due to shorter runtimes.

5.3 Sensitivity analysis

In this section, we investigate the sensitivity of our method to different architectural parameters. Although we have observed similar trends under both complexity scenarios, in this section, we only report for the optimistic scenario. In Section 5.3.1, we report how different compiler optimisation levels impact energy. In Section 5.3.2, we report how bypassing trivial instructions impacts energy in wider processors.

5.3.1 Compiler optimisations: Fig. 10 reports energy improvements achieved for different compiler optimisations for the applications studied in this work. Bars from left to right report energy improvement for −O1, −O2 and −O3 compiler optimisations, respectively. As reported, changing the compiler optimisation flag does not impact energy considerably.

5.3.2 Issue bandwidth: Fig. 11 reports energy reduction for processors with wider execution bandwidths. Bars from left to right show energy improvement in 8-, 16- and 32-way processors, respectively. Generally, trivial bypassing results in higher energy improvements in wider processors. On average, energy is improved by 15.6, 18.2, 21.3% in 8-, 16- and 32-way processors, respectively.

Bypassing trivial instructions not only leads to their faster execution, but also results in faster execution of those instructions depending on the trivial instructions. A processor with a higher issue bandwidth can execute more instructions every cycle. Therefore in a wider processor a larger number of depending instructions can potentially benefit from early computation of a trivial instruction. This, ultimately, results in a shorter program runtime, better resource utilisation and consequently higher energy reduction for wider processors.

6 Related work

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

Lipasti and Shen [10] introduced value prediction and showed that data values exhibit ‘locality’. 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 [11] introduced the concept of dynamic instruction reuse. Their work relied on the observation that many instructions, having the same inputs, are executed dynamically. As such, many instructions do not have to

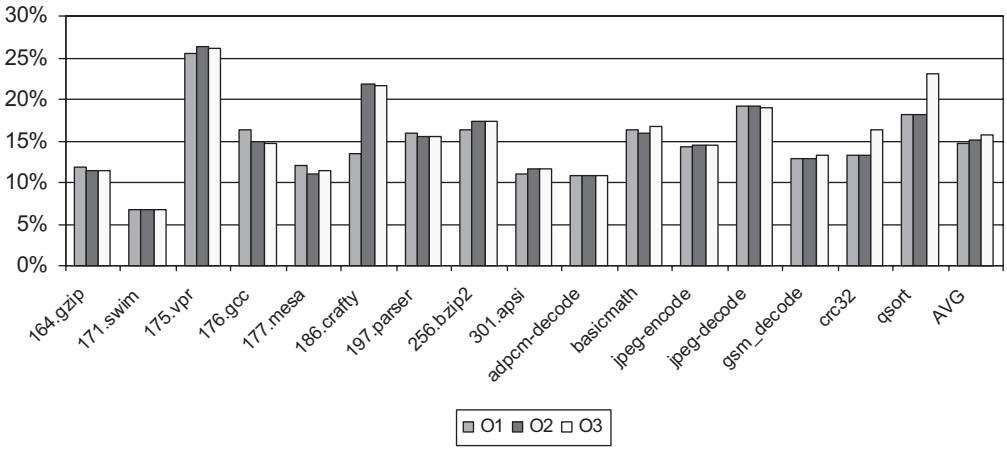


Fig. 10 *Energy improvement for different compiler optimisations*

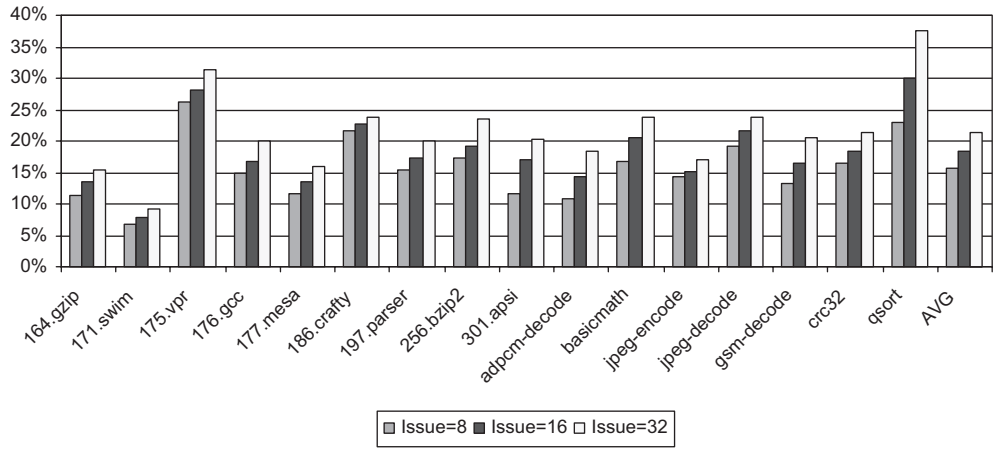


Fig. 11 *Energy improvement for processors with wider execution bandwidths*

be executed repeatedly, as their results can be obtained from a buffer, where they were saved previously.

Our work is different from the two studies discussed earlier, as 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 behaviour, we do not require additional storage to store the associated information. Consequently, we are able to improve both energy and performance simultaneously.

It should be noted that using speculation to predict trivialising operands can result in performance improvement [12]. However, speculating trivialising operands does not improve energy as it comes with considerable energy overhead required for speculating trivialising operands.

Our technique is orthogonal to and can be used on top of other novel architectural techniques. For example, it can be used in a system using trace cache, as using a trace cache (instead of a regular conventional cache) does not impact trivial instruction distribution or frequency. The same is true for using speculative data fetch techniques such as data pre-fetching or way prediction.

Richardson [13] suggested a restricted form of bypassing trivial instructions. His definition of trivial computations only included certain multiplications (by 0, 1 and −1), divisions ($X \div Y$ with $X = \{0, Y, -Y\}$) and square roots of 0 and 1. Our simulations show that the energy and performance improvements achieved by Richardson's method are lower compared with our method. The low improvements are due to restricting trivial definition to only three instructions. We also extended trivial bypassing to different instruction types and studied its energy savings benefits. In Table 4, we report energy, performance and energy-delay

Table 4: Trivial instruction bypassing method compared to Richardson's method

Benchmarks	Trivial bypass			Richardson		
	Performance improvement (optimistic/ pessimistic), %	Energy savings (optimistic/ pessimistic), %	Energy-delay improvement (optimistic/ pessimistic), %	Performance improvement, %	Energy savings, %	Energy-delay improvement, %
164.gzip	14.7/13.1	11.4/8.3	22.8/18.1	0.00	0.07	0.07
171.swim	8.8/7.7	6.8/5.6	14.3/12.1	0.00	0.08	0.08
175.vpr	45.6/30.9	26.0/15.8	49.2/27.5	0.03	0.10	0.13
176.gcc	20.3/13.3	14.7/6.9	29.1/15.2	0.12	0.19	0.31
177.mesa	16.4/14.0	11.5/7.8	24.0/18.0	1.13	0.97	2.08
186.crafty	30.7/20.7	21.6/10.2	40.0/20.6	2.00	1.12	3.06
197.parser	20.6/13.6	15.4/8.9	29.9/17.2	0.22	0.30	0.52
256.bzip2	24.7/22.4	17.3/13.9	33.7/28.0	0.00	0.08	0.08
301.apsi	15.5/15.4	11.6/11.4	23.5/23.2	0.68	0.45	1.13
adpcm-decode	14.8/11.6	10.7/9.4	22.3/17.8	0.00	0.10	0.10
basicmath	23.6/18.6	16.8/10.5	32.7/22.0	1.60	1.36	2.92
jpeg-encode	21.8/16.6	14.4/10.1	29.7/20.7	0.26	0.36	0.62
jpeg-decode	30.1/27.6	19.0/15.3	37.8/31.6	0.17	0.40	0.57
gsm_decode	17.2/13.2	13.2/8.6	26.0/17.7	0.24	11.8	12.02
crc32	22.4/22.4	16.3/16.3	31.7/31.7	0.16	50.5	50.66
qsort	35.9/26.7	23.0/13.7	43.3/26.5	1.22	0.81	2.00
AVG	22.7/18.0	15.6/10.8	30.6/21.7	0.49	4.30	4.77

improvements for both our technique (under both complexity scenarios) and that of Richardson [13].

Yi and Lilja [1] 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.

This paper is an extension of our previous work on trivial instruction bypassing [14]. We improve and extend our previous work by providing a more detailed analysis of trivial instructions, providing a real example of trivial instruction from SPEC'2K, and extending the number and variety of the benchmarks. In addition, we use a more appropriate baseline architecture that better reflects modern processors. Moreover, we analyse the effect of trivial bypassing on issue delay and energy breakdown over different components and report sensitivity analysis. We also report results for different timing/complexity scenarios and compare our work with a previous study.

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

Chung *et al.* [16] suggested optimisation of embedded software. 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 specialising frequently executed procedures for the most common values of their parameters. Their work included introducing software solutions to replace complex functions that have trivialising operands with more simple functions. Our work is different, as it introduces dynamic hardware-based optimisations. Dynamic optimisations can be integrated with the micro-architecture and result in better architectural transparency. As such, our technique can optimise legacy code without the need for recompilation.

7 Conclusion

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

We categorised trivial instructions to fully-trivial and semi-trivial instructions on the basis of whether both source operands are needed to decide the result. We also categorised trivial instructions to decode-trivial and issue-trivial instructions on the basis of the pipeline stage that they could be identified at. We showed that semi-trivial instructions account for the majority of trivial instructions, whereas, on average, decode-trivial and issue-trivial instructions account for an almost equal share.

We showed that trivial bypassing reduces energy consumption and energy-delay considerably over a conventional processor. Among different components, ALU, register file and issue window benefit most from trivial bypassing. We also showed that wider processors benefit more from bypassing trivial instructions.

8 Acknowledgements

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.

9 References

1 Yi, J.J., and Lilja, D.J.: 'Improving processor performance by simplifying and bypassing trivial computations'. Proc. 2002 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, September 2002, pp. 462–465
2 Ziv, J., and Lempel, A.: 'A universal algorithm for sequential data compression', *IEEE Trans. Inf. Theory*, 1977, **23**, (3), pp. 337–343
3 Huffman, D.A.: 'A method for the construction of minimum redundancy codes', *Proc. IRE*, 1952, **40**, (9), pp. 1098–1101
4 Hennessy, J., and Patterson, D.: 'Computer architecture: a quantitative approach' (Morgan Kauffman, San Francisco, CA, 1990, 1996, 2003)
5 Sohi, G.S.: 'Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers', *IEEE Trans. Comput.*, 1990, **39**, pp. 349–359
6 SPEC Benchmark Suite. Available at http://www.spec.org
7 Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B.: 'MiBench: a free, commercially representative embedded benchmark suite'. Available at http://www.eecs.umich.edu/mibench/
8 Burger, D., Austin, T.M., and Bennett, S.: 'Evaluating future microprocessors: the simple scalar tool set'. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996
9 Brooks, D., Tiwari, V., and Martonosi, M.: 'Wattch: a framework for architectural-level power analysis and optimizations'. Proc. Int. Symp. on Computer Architecture, 2000, pp. 83–94
10 Lipasti, M.H., and Shen, J.P.: 'Exceeding the dataflow limit via value prediction'. Proc. 29th Annual ACM/IEEE Int. Symp. and Workshop on Microarchitecture, December 1996, pp. 226–237
11 Sodani, A., and Sohi, G.S.: 'Dynamic instruction reuse'. Proc. 24th Annual Int. Symp. on Computer Architecture, July 1997, pp. 194–205
12 Atoofian, E., Baniasadi, A., and Dimopoulos, N.: 'Improving performance by speculating trivializing operands in trivial instructions'. 2nd Value-Prediction and Value-Based Optimization Workshop, Boston, Massachusetts, 10 October 2004, pp. 26–31
13 Richardson, S.: 'Caching function results: faster arithmetic by avoiding unnecessary computation'. Int. Symp. on Computer Arithmetic, 1993
14 Atoofian, E., Baniasadi, A., and Dimopoulos, N.: 'Improving energy-efficiency by bypassing trivial computations'. 1st Workshop on High-Performance, Power-Aware Computing, Denver, Colorado, 4 April 2005
15 Tran, L., Nelson, N., Ngai, F., Dropsho, S., and Huang, M.: 'Dynamically reducing pressure on the physical register file through simple register sharing'. Int. Symp. on Performance Analysis of Systems and Software, March 2004, pp. 78–88
16 Chung, E., Benini, L., DeMicheli, G., Luculli, G., and Carilli, M.: 'Value-sensitive automatic code specialization for embedded software', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2002, **21**, (9), pp. 1051–1067