Available online at www.sciencedirect.com
ScienceDirect
Microprocessors and Microsystems 32 (2008) 183–196
MICROPROCESSORS AND MICROSYSTEMS
www.elsevier.com/locate/micpro

Early detection and bypassing of trivial operations to improve energy efficiency of processors ☆

Mafijul Md. Islam *, Magnus Själander, Per Stenström

Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Goteborg, Sweden

Available online 1 November 2007

Abstract

This paper addresses the issue of improving the energy efficiency of processors by eliminating trivial operations. The paper provides a new classification of trivial operations and quantifies their relative frequency in desktop and embedded applications. It then presents a hardware technique to remove trivial operations as early as at the decode stage of the pipeline to save energy. This paper shows that 13.6% and 8.6% of the instructions are identity-trivial in the selected applications in the SPEC CPU2000 and EEMBC1.1 benchmark suites, respectively. Early detection and elimination of trivial operations reduce the average energy consumption of the core pipeline by 9% and 6%, respectively.

Keywords: Energy efficiency; Identity-trivial; Register renaming; Trivial operation; Value locality

1. Introduction

A trivial operation is an instruction whose outcome can be inferred from the input values without performing the specified computation. For example, the result of the instruction R1 ← R2 * R3 is zero if either R2 or R3 equals zero. So the instruction can be performed trivially and no functional unit is required to do the actual computation. As a result, trivial operations may improve performance and energy efficiencies of processors. Nowadays, energy is one of the key concerns for both low power embedded processors and high-end processors. This has motivated researchers to propose various micro-architectural techniques to reduce energy consumption of the different components of processors [1,3,4,6,9,16,17,20,21,26,29]. In this work, we quantify the frequency of trivial operations and present architectural support to detect and bypass them at the decode stage of the pipeline to improve the energy efficiency of processors.

Studies have shown that many instructions in programs are often executed with the same input values and thus produce the same output [14,23,31,35]. Based on these studies, various hardware techniques such as 'instruction memoization' (IM) [11,18,28], 'value prediction' (VP) [15,24] and 'instruction reuse' (IR) [27,33,40] have been proposed to obtain the results of the instructions from their previous executions. In all these approaches, information related to the previous execution of the instructions is stored in a buffer. When a new instruction is encountered, a lookup into that same buffer is performed to predict the values in VP and to bypass the execution of the instruction in IM as well as in IR. On the contrary, detecting and bypassing trivial operations do not require that the instruction is already executed. As a result, it does not require a buffer to store the related information. Hence, detecting and bypassing trivial operations may provide better opportunity to improve the energy efficiency.

This study is inspired by the frequent-value locality phenomenon shown by Yang and Gupta [39]. They showed that a few values are more likely to be used than others in the program execution. They observed that the top

☆ A preliminary version of this paper appears in the Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2006).

* Corresponding author. Tel.: +46 31 772 1711; fax: +46 31 772 3663.
E-mail addresses: mafijul.islam@chalmers.se (M.Md. Islam), magnus.sjalander@chalmers.se (M. Själander), pers@chalmers.se (P. Stenström).

0141-9331/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2007.10.001

two common values are zero and one though many of the top-rated values may differ from application to application. However, trivial computation was coined in 1993 by Richardson [30]. He considered a restricted set of trivial operations which includes multiplications (by 0, 1, −1), divisions ($X \div Y$ with $X = \{0, Y, -Y\}$), and square roots of 0 and 1. Recently, several works [2,42] have extended the notion of trivial operations to other cases to show further performance and energy improvements. It has been shown in [42] that about 12% of the total executed instructions are trivial despite aggressive compiler optimizations of the program and the amount of trivial computations does not heavily depend on specific inputs of the programs.

None of the aforementioned works has studied the extent to which the removal of trivial operations can save energy of the different components of the pipeline. Moreover, previous works have neither provided the required micro-architectural structure to detect and bypass trivial operations nor have they quantified the energy consumption of the additional hardware. This paper is intended to fill these gaps. We show that an arithmetic/logical operation can be eliminated from the pipeline by quickly converting it into a move operation if one of the source operands is the identity element of the operation; for example, zero for addition and one for multiplication. We have proposed a modification of the register renaming unit of MIPS R10000 [41] to detect and bypass trivial operations as early as at the decode stage of the pipeline. The result of a trivial operation is mapped to one of its source operands instead of mapping the result to an available register. The execution of the operation is thus bypassed. For example, the execution of $R1 \leftarrow R2 + R3$ can be bypassed by remapping R1 to R3 if R2 contains zero.

This work makes the following contributions:

- We provide a new classification of trivial operations. We find that about 94% and 65% of the trivial operations are identity-trivial for the chosen applications of SPEC CPU2000 and EEMBC1.1 benchmark suites, respectively.
- We provide a micro-architectural technique to detect and bypass trivial operations as early as at the decode stage of the pipeline. We find that about 12% and 6.4% of the total executed instructions can be detected as trivial at the decode stage.
- We quantify the energy consumption of the additional hardware required to detect and bypass trivial operations. We also estimate the energy reduction of the different components of the pipeline because of early detection and bypassing of trivial operations. We show that the proposed method yields an overall energy reduction of the core pipeline by 9% in SPEC applications and by 6% in EEMBC applications.

The rest of the paper is organized as follows: Section 2 presents the set of trivial operations used in this study as well as the classification and frequency of trivial operations. The architectural model and the experimental methodology to detect and bypass trivial operations are described in Sections 3 and 4, respectively. The obtained results are presented and discussed in Section 5. The comparison of our work with the previous related works is done in Section 6. Finally, we conclude in Section 7.

2. Trivial operations

In this section, we first present the set of trivial operations and the classification of trivial operations. Then we quantify the relative frequency of trivial operations.

2.1. Condition and classification of trivial operations

We have first identified the particular input values of the source operands of an instruction which will make it trivial. This has yielded the set of trivial operations shown in Table 1. The set of trivial operations and their classification used in this study are adopted from our previous work presented in [25]. It is notable that the input to addition, subtraction, multiplication, and division may be either integer or floating-point (FP) values. On the other hand, the input to AND, OR, XOR, logical shift and arithmetic shift is always integer values.

An instruction is classified as *identity-trivial* if one of the source operands is the identity element of the corresponding operation and the result of the instruction is the value of the other source operand. For example, $Z = X * Y$ is identity-trivial if either X or Y equals 1 which is the identity element of the multiplication operation. Another class of trivial operations is *inverse-trivial* in which one of the source operands is the inverse element of the other and the result is the identity element of the operation. For example, $Z = X + Y$ is inverse-trivial if the values of X and Y are equal in magnitude but have opposite sign. In this case, the result is zero which is the identity element of the addition operation. Moreover, operations such as '$Z = X/X$', '$Z = X/1$', '$Z = X * 0$', etc. are trivial. These are defined as *other-trivial*.

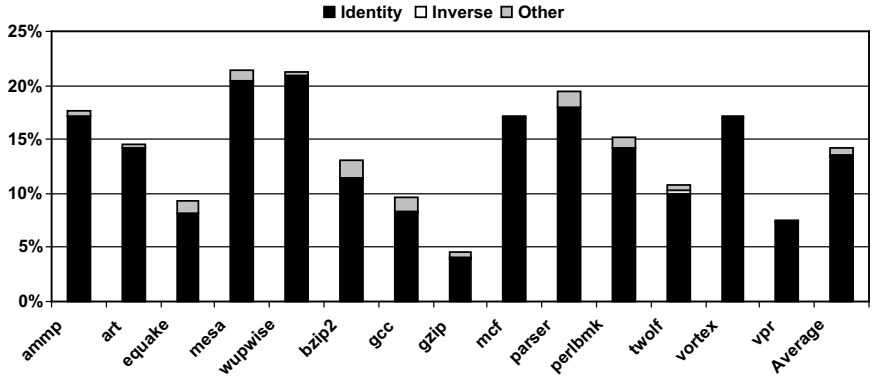
2.2. Frequency of trivial operations

We have measured the frequency of trivial operations as a fraction of the total executed instructions using the architectural model and simulation methodology described in Sections 3 and 4, respectively. In this study, we have considered integer and FP arithmetic and logical instructions as well as the effective address calculations for load and store operations.

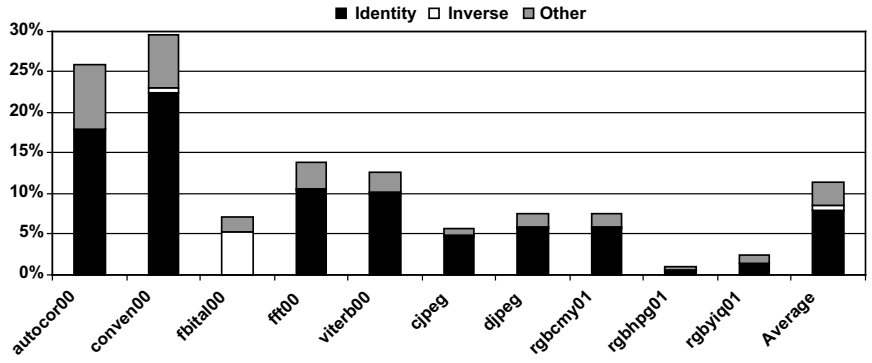
The results for SPEC CPU2000 and EEMBC1.1 benchmarks are shown in Fig. 1a and b, respectively. Each bar of Fig. 1 consists of three sections – the bottom section refers to identity-trivial, the middle section refers to inverse-trivial and the top section corresponds to other-trivial. It is clear from Fig. 1a that about 14% of the total executed

Table 1
The trivial operations and their classification used in this study

Operation	Identity-trivial	Inverse-trivial	Other-trivial
Addition: X + Y	X = 0 or Y = 0	X = −Y or −X = Y	
Subtraction: X − Y	Y = 0	X = Y	X = 0 and Y = 0
Multiplication: X * Y	X = 1 or Y = 1		X = 0 or Y = 0
Division: X/Y			Y = 1; X = 0; X = Y
AND: X & Y	X = 0xffffffff or Y = 0xffffffff		X = Y; X = 0x00000000 or Y = 0x00000000
OR: X | Y	X = 0x00000000 or Y = 0x00000000		X = Y; X = 0xffffffff or Y = 0xffffffff
XOR: X ⊕ Y	X = 0x00000000 or Y = 0x00000000	X = Y	
Logical shift: X ≪ Y, X ≫ Y	X = 0 or Y = 0		
Arithmetic shift: X ≪ Y, X ≫ Y	X = 0 or Y = 0		X = 0xffffffff or Y = 0xffffffff



(a) Frequency of identity-trivial, inverse-trivial and other-trivial in SPEC CPU2000.



(b) Frequency of identity-trivial, inverse-trivial and other-trivial in EEMBC1.1.

Fig. 1. Frequency of identity-trivial, inverse-trivial and other-trivial operations.

instructions are trivial in SPEC CPU2000 and on average 94% of all the trivial instructions are identity-trivial. It is noticeable that trivial operations are common in both integer (bzip2, gcc, gzip, mcf, parser, perlbmk, vortex, vpr and twolf) and FP (ammp, art, equake, mesa and wupwise) applications. Moreover, only a negligible fraction of the trivial operations are inverse-trivial (1%) and other-trivial (5%). Fig. 1b shows that on average 11.4% of the total executed instructions are trivial in EEMBC1.1 and about 65% of all trivial instructions are identity-trivial. Further, trivial operations are common in all the applications except rgbhpg01 and rgbyiq01. We can also see that identity-trivial is the dominant component in all the applications except fbital00 in which about 74% of the trivial operations are inverse-trivial. Moreover, other-trivial and inverse-trivial consist of 9% and 26% of the total trivial operations, respectively. Thus, the dominance of identity-trivial operations is a common phenomenon in both SPEC CPU2000 and EEMBC1.1.

However, knowing only the overall frequency of trivial operations is not sufficient to evaluate the energy efficiency of this technique. It is equally important to know whether

trivial operations are common in all types of arithmetic/logical instructions or limited to a certain class of instructions. The instructions which have single-cycle execution latency include AND, OR, XOR, logical and arithmetic shifts, and integer addition and subtraction. These are referred to as *short* instructions in this study. On the other hand, instructions such as integer and FP multiplication and division, and FP addition and subtraction require multiple cycles to complete their execution. These are referred to as *long* instructions. Thus, long instructions are expected to have more impact on the energy consumption than short instructions.

Fig. 2 shows the frequency of trivial operations as percentage of the total executed instructions of each type of arithmetic/logical instructions. In Fig. 2, the left bar represents the average frequency of trivial operation in SPEC CPU2000 benchmark suite and the right bar represents the average frequency of the corresponding operation in EEMBC1.1 benchmark suite. The obtained results confirm the findings of [2,25,42] and show that instructions of all categories contribute significantly to the overall frequency of trivial operations. We see from Fig. 2 that on average 17% of the short instructions are trivial in SPEC applications whereas about 23% of such instructions are trivial in EEMBC applications (denoted by AVG-S). On the other hand, about 15% and 26% of the long arithmetic instructions are trivial in SPEC and EEMBC applications, respectively (denoted by AVG-L).

The impact of an operation on energy and performance is determined by (a) the type of the operation along with the percentage of trivial operations of that type, (b) the frequency of that operation during the execution of the program and (c) the pipeline stage at which it is detected as trivial. As a result, we have measured the frequency of short and long arithmetic instructions. We have found that about 93% of the arithmetic/logical instructions are short and the remaining 7% are long arithmetic instructions in SPEC applications. In EEMBC applications, about 97% of the arithmetic instructions are short and the remaining only 3% are long. Thus even though the percentage of long trivial operation is higher than that of short instructions, long instructions are likely to have less impact on the energy reduction.

Finally, the results presented in this section show that a significant portion of the total executed instructions are trivial in almost every application and trivial operations are common in all categories of arithmetic/logical instructions. Moreover, it can be noted that the vast majority of the trivial operations are identity-trivial. Thus, the mechanism of detecting and bypassing trivial operations is expected to improve the energy efficiency of processors.

3. Architectural model and implementation

In this section, we first present the basic pipeline model. Then, we classify the trivial operations as decode-trivial and issue-trivial, and discuss the implications of this classification on the energy efficiency. Finally, we describe the register renaming unit which detects and bypasses trivial operations as early as at the decode stage of the pipeline.

3.1. Basic pipeline model

We have assumed a micro-architectural model which consists of six pipeline stages as shown in Fig. 3. Although this pipeline model is simple, it is attractive in many high-performance embedded processors because of its energy efficiency. In fact, multi-core processors such as Sun Microsystems' Niagara implement a simple pipeline with six stages (fetches, thread select, decode, execute, memory, and write back) [22]. In our assumed model, instructions are fetched and stored in the instruction queue during the instruction fetch (IF). Both instruction-decoding and register renaming are done at the decode pipeline stage (ID) as in the MIPS R10000 superscalar processor [41]. The entries are then allocated in the reorder buffer and the instructions advance to the issue stage. The instruction window stores the values until the instructions are ready to commit. As soon as the dependences are satisfied and the needed functional units are available, the instructions get executed. The execution outcome is then broadcast to wake up the dependent instructions at the write back stage (WB). Finally, the resources allocated for the instructions are freed at the commit stage (CT) [19].

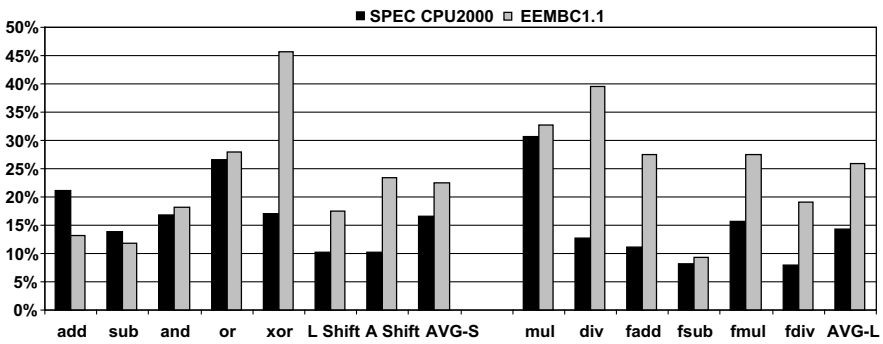


Fig. 2. Frequency of trivial operations by operation type.

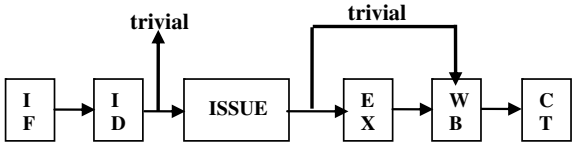


Fig. 3. Pipeline model supporting detection of trivial operations.

3.2. Trivial operation detection

The detection of trivial operations can be done either at the decode stage or at the issue stage of the pipeline shown in Fig. 3. The earlier in the pipeline an instruction is identified as a trivial operation; the better is the opportunity to achieve energy and performance benefits. A natural place of detecting and avoiding trivial operations is the decode stage at which register renaming is done. The logical register of an operation is mapped to an available physical register during register renaming. To detect and bypass trivial operations, this can be altered so that the result of the operation is mapped to the register hosting one of the source operands instead of mapping the result to an available register. Thus, trivial operations detected at the decode stage can eliminate write accesses to the register file. Moreover, register read accesses can be completely eliminated for such trivial operations if we assume dedicated registers for operand values zero and one as suggested in [5]. Apart from saving energy by eliminating the register accesses, the operation identified as trivial at the decode stage also saves energy by not having to issue and execute it. As a result, energy is saved in the result bus, the instruction window, and the functional units.

If the detection were done at the issue stage, more operations could have been detected as trivial. However, since the operation has already been dispatched, it is necessary to commit the operation to update the processor state and resolve data dependencies. Hence, trivial operations detected at the issue stage provide no energy benefits for the result bus. Moreover, issue-trivial operations provide much less energy benefits for the instruction window and register file as the register file has already been accessed at this point.

3.3. Decode-trivial and issue-trivial operations

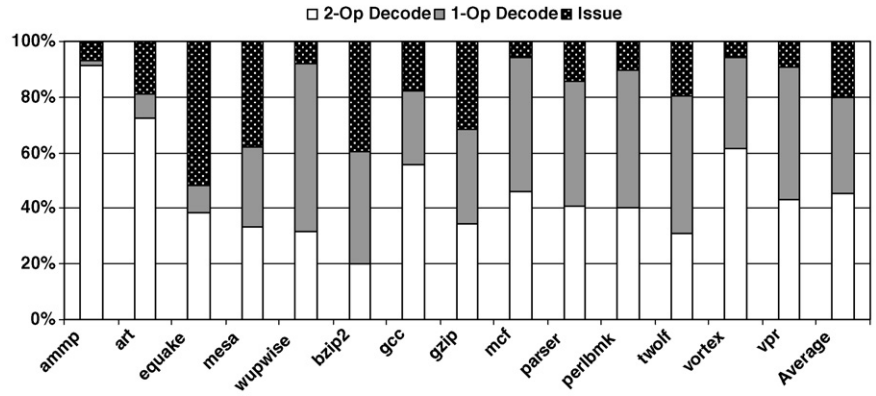
Trivial operations have been grouped into *decode-trivial* and *issue-trivial* based on the pipeline stage at which they can be detected [2,25]. However, there are two different cases of detecting trivial operations at the decode stage. Firstly, both operands are available and secondly, the operand dictating the result of the operation, for example, the identity element of the identity-trivial operation is available. In this study, the former corresponds to *2-Op* and the later corresponds to *1-Op* decode-trivial. For example, the instruction $R1 \leftarrow R2 + R3$ becomes 1-op decode-trivial if (a) the instruction is at the decode stage, (b) the value of R2 is known to be zero - the identity element of the operation, and (c) the value of R3 is still unknown. In this case, we can bypass the instruction by remapping R1 to R3 without waiting for the value of R3 to be known. This distinction of decode-trivial has not been done in the previous studies [2,25].

The distinction between 1-Op and 2-Op makes a lot of sense if we review the results of SPEC CPU2000 and EEMBC1.1 benchmarks shown in Fig. 4a and b, respectively. Each bar of Fig. 4 consists of three sections – the bottom section refers to 2-Op decode-trivial, the middle section refers to 1-Op decode-trivial, and the top section corresponds to issue-trivial. It is clear from Fig. 4a that on average 80% of the trivial operations are indeed decode-trivial and 20% are issue-trivial in SPEC applications. We see from Fig. 4b that about 54% of the trivial operations are decode-trivial whereas the remaining 46% are issue-trivial in EEMBC applications. It is noticeable that 1-Op decode-trivial constitutes a significant portion of the total trivial operations in both SPEC (35%) and EEMBC (33%) applications.

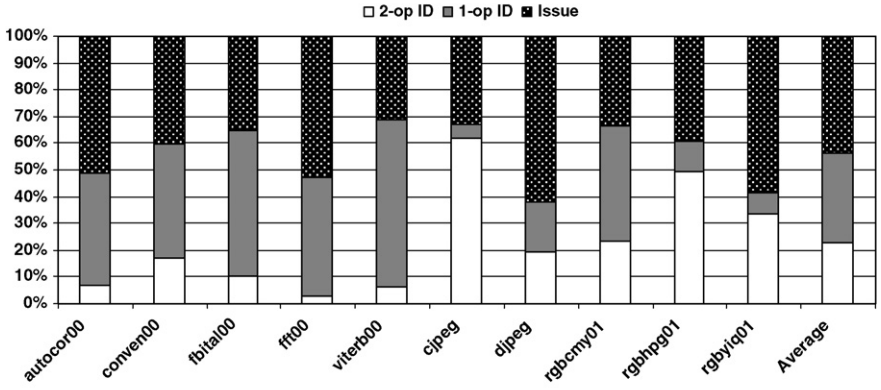
3.4. Discussion of energy reduction opportunities

We have shown in Section 2 that identity-trivial operations dominate among the different categories of trivial operations. This is an interesting finding because this class of trivial operations provides better opportunities to improve the energy efficiency. Firstly, it can enable simple detection and elimination of most of the trivial operations because we need to detect if one of the operands is zero or one. Secondly, the result of an identity-trivial operation is always the value of the other source operand and thus remapping can eliminate the update of the register file [38]. Finally, if one of the source operands of a particular arithmetic or logical instruction is already known to be the identity element, it becomes unnecessary to read and wait for the value of the other source operand.

Since both source operands must be known to detect an inverse-trivial operation, the inverse-trivial category offers almost no opportunity of saving energy. On the contrary, it might hurt the performance in some cases, for example, for integer addition. As a result, though we have measured the frequency of inverse-trivial operations, we have not considered them as a candidate for the energy efficiency improvement. The impact of other-trivial operations on the energy reduction may differ from operation to operation [25]. For example, the other-trivial '$Z = 0/X$' is as useful as an identity-trivial because one of the operands is zero. On the other hand, the other-trivial '$Z = X/Y$ assuming X equals Y' has similar problems as the inverse-trivial category because we need to know and compare the values of both the operands to detect it. Moreover, decode-trivial operations will save energy in the instruction window (IW), ALU, the result bus, and the register file (RF) as mentioned in Section 3.2. On the contrary, issue-trivial operations provide much less energy benefits and hence, we have not considered issue-trivial operations to evaluate their impact



(a) Frequency of 1-Op decode, 2-Op decode and issue trivial operations in SPEC CPU2000.



(b) Frequency of 1-Op decode, 2-Op decode and issue trivial operations in EEMBC1.1.

Fig. 4. Frequency of 1-Op decode, 2-Op decode and issue-trivial operations.

on the energy efficiency. Finally, we have a set of decode-trivial operations which consists of identity-trivial and other-trivial behaving similar to identity-trivial. This set contains about 99% of the decode-trivial operations which provides efficient detection (only need to detect if one of the operands is zero or one) and significant energy reduction. We have therefore opted for detecting and bypassing only such decode-trivial operations to improve the energy efficiency.

3.5. Conventional register renaming

The register renaming of the MIPS R10000 [41] is shown in Fig. 5 and consists of a mapping table, a list of free registers, a list of busy register, and an active list. The mapping table consists of as many entries as logical registers and each entry stores the physical register to which the logical register is mapped. The free list stores physical registers that are available for mapping to a logical register. The busy list keeps track of whether the value of the physical register is valid or whether the result has not yet been produced. The active list is responsible for retiring operations in the sequential order of the application.

When a new instruction is decoded, a physical register has to be mapped to the logical result register of the instruction. This is done by retrieving the address of a physical register

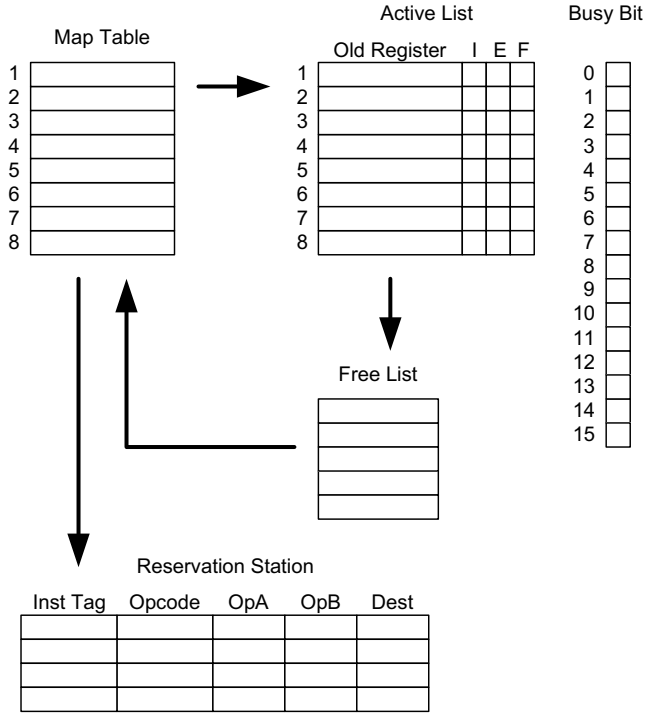


Fig. 5. Illustration of the MIPS R10000 register renaming.

from the free list and writing it to the location of the logical register in the map table. The logical register locations of the mapping table are read to know the physical registers of the operands. The instruction can then be written into the reservation station. To preserve the sequential order, the new instruction is given a tag corresponding to a location in the active list. There are three possible states of the instruction in the active list – ready to be issued (I), at the execute stage (E) or the result is available to be written to the physical register (F). The active list also stores the previous physical register to which the logical register of the instruction was mapped. When all the previous instructions in the active list have retired, it is known that no instructions depend on the previous mapping and the physical register can therefore be added to the free list. At this time, the instruction can also be retired as soon as the result is available. To respect data dependences, the physical register used for storing the result of the instruction is marked as busy. An instruction can not be issued from the reservation station unless all its operands are marked as not busy. Once the result of an instruction is available, the busy bit for its result register is cleared.

3.6. Register renaming and trivial operation detection

We have modified the register renaming unit to detect and bypass identity-trivial operations at the decode stage. In addition to the base model described in Section 3.5, the register renaming which supports trivial operation detection has a list consisting of two bits for each physical register to indicate if the value of the register is zero or one. Thus, it is expected that the energy consumption of the renaming unit will be increased. The modified register renaming unit is shown in Fig. 6.

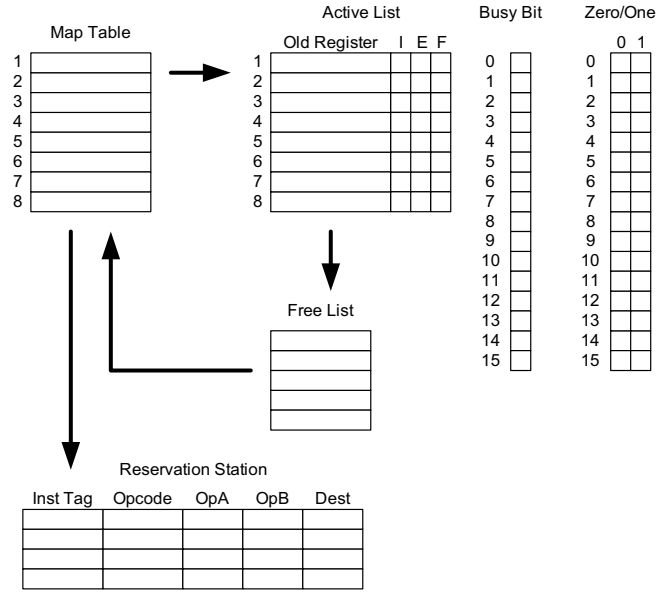


Fig. 6. Register renaming to support detecting and bypassing trivial operations.

We need to inspect the operands of an instruction to detect if it is an identity-trivial. The addresses of the physical registers of the operands which are read from the mapping table are therefore used to index the zero/one and busy lists. This knowledge of the operands along with the opcode determines if the operation is trivial or not, for example, the operation is an ADD and one of the operands is zero. If an instruction is not detected as an identity-trivial operation, the register renaming unit functions in a similar way as described in Section 3.5.

However, if the instruction is detected as an identity-trivial, the result of the instruction is one of its source operands and no computation is therefore needed. The logical result register is simply mapped to the physical register of the operand containing the result instead of using one of the physical registers from the free list. Since the instruction is pointing to the physical register with the result, there is no need to write the instruction to the reservation station and the instruction can be marked as finished (F) in the active list. There is no need to update the list with busy registers. To detect an identity-trivial operation, the zero/one list needs to be maintained. To avoid checking the actual operand each time it is accessed, the check is done for each value written to the register file and the result is stored in the zero/one list. The zero/one list is an entity by its own to avoid unnecessary accesses to the register file.

4. Simulation methodology

We have used the 3.0c distribution of the SimpleScalar toolset [8] to simulate a dynamically scheduled processor. The processor model supports speculative out-of-order execution. The detailed architectural parameters of the simulated processor are shown in Table 2. We have modified *sim-outorder*, a detailed performance simulator of the SimpleScalar toolset, to obtain the results. Wattch-1.04 [7] is used to estimate the energy usage and savings of the different units. VHDL implementations have been created for both the conventional and the modified register renaming units described in 3.5 and 3.6, respectively. Both VHDL implementations can rename four instructions in parallel. The two VHDL implementations have been taken through synthesis [12] for a commercially available 0.13 μm technology. This is used to capture the power, delay and area impacts of the additional logic introduced to detect and bypass trivial operations. Power consumption has been estimated for the two implementations at the same clock frequency.

We have used five applications (ammp, art, equake, mesa and wupwise) from the SPEC CFP2000 and nine applications (bzip2, gcc, gzip, mcf, parser, perlbmk, vortex, vpr and twolf) from the SPEC CINT2000 of the SPEC CPU2000 benchmark suite [36]. We have also used the five applications (autocor00, conven00, fbital00, fft00 and viterb00) of the telecom domain and the five applications (cjpeg, djpeg, rgbcmy01, rgbhpg01 and rgbyiq01) of the consumer domain of the industry-standard embedded

Table 2
Parameters of the simulated processor

Parameter	Value
Fetch, decode and commit width	4 instructions/cycle
Issue width	4 instructions/cycle (out-of-order)
Size of register update unit (RRU)	64 instructions
Size of load/store queue (LSQ)	32 instructions
Functional units (quantity, execution latency in cycles)	Integer ALU (2, 1), FP ALU (2, 2), integer MULT/DIV (1, 3/20), FP MULT/DIV (1, 4/12), Memory port (2, 1)
Architected registers	32 integer and 32 floating-point (FP)
L1 instruction cache	64 KB, block size 32 bytes, 4-way set-associative, number of sets 512, Replacement policy: least recently used (LRU), 1 cycle latency
L1 data-cache	32 KB, block size 32 bytes, 2-way set-associative, number of sets 512, replacement policy: LRU, 1 cycle latency
Unified L2 cache	256 KB, block size 64 bytes, 4-way set-associative, number of sets 1024, replacement policy: LRU, 12 cycles latency
Memory latency – first, following blocks	60 cycles, 5 cycles
Instruction TLB	16 sets, page size 4096, 4-way set-associative, 64 entries
Data TLB	32 sets, page size 4096, 4-way set-associative, 128 entries
Branch predictor	2048-entry Bimodal predictor; Return stack size (RAS): 8-entry; branch target buffer (BTB): 4-way, 512 sets

benchmark suite EEMBC1.1 [13]. All of the programs were compiled at the optimization level −O3 using the PISA GCC 2.7.2.3 cross-compiler.

All of the embedded applications have been simulated by running them until their completion. To reduce the simulation time of the SPEC applications, we have used SimpleScalar BBV and SimPoint3.2 to find the simulation points to quickly simulate parts of a program's execution to represent the entire execution [32]. SimpleScalar BBV uses a modified version of *sim-fast*, a fast functional simulator of the SimpleScalar toolset, to generate *Basic Block Vectors (BBV)* by running each benchmark and reference-input pair to completion. Then, we have used SimPoint3.2 to generate the simulation points and their corresponding weights from BBV using an interval length of 100 million. A *simulation point* is a starting simulation place (in number of instructions executed from the start of execution) in a program's execution. The weight represents the percent of overall executed instructions each simulation point represents. The weight for a simulation point is the total instructions executed by all of the intervals in that simulation point's cluster divided by the total number of instructions executed for the program/input pair [32]. The single simulation point typically captures the behavior of two dominant phases. Since the programs may have many phases, e.g. gcc, gzip, bzip2, the single simulation point may have a higher error rate than if multiple points are used [32]. This has motivated us to simulate the SPEC applications using multiple simulation points.

We have used three simulation points for each of the chosen SPEC CPU2000 applications except for art. In case of art, the tool generated only two simulation points to capture the entire program behavior. Table 3 shows the SPEC benchmark applications, the simulation points along with their corresponding weights and the input set used in this study. The multiple simulation points are finally simulated by using *sim-outorder* and following the methodology described in [32]. For example, if the simulation point is 20 and the interval size is 100 million, we have fast-forwarded the 20*100 million instructions (functional simulation),

Table 3
The SPEC CPU2000 applications used in this study

Benchmark	(Simpoint, Weight)	Input
ammp	(561, 0.14), (846, 0.48), (3194, 0.38)	ammp-ref
art	(75, 0.15), (257, 0.85)	art-470
equake	(197, 0.17), (985, 0.41), (1376, 0.42)	equake-ref
mesa	(937, 0.04), (3909, 0.78), (4889, 0.18)	mesa-ref
wupwise	(11450, 0.47), (12308, 0.48), (13322, 0.05)	wupwise-ref
bzip2	(248, 0.26), (348, 0.46), (483, 0.28)	bzip2-program
gcc	(61, 0.31), (116, 0.47), (341, 0.22)	gcc-166
gzip	(113, 0.33), (412, 0.16), (567, 0.51)	gzip-source
mcf	(112, 0.11), (134, 0.42), (321, 0.47)	mcf-ref
parser	(24, 0.50), (2036, 0.31), (2869, 0.19)	parser-ref
perlbmk	(4, 0.03), (64, 0.65), (166, 0.32)	perlbmk-diffmail
twolf	(400, 0.26), (1402, 0.55), (2754, 0.19)	twolf-ref
vortex	(518, 0.16), (615, 0.24), (696, 0.59)	vortex-one
vpr	(368, 0.46), (458, 0.40), (507, 0.14)	vpr-route

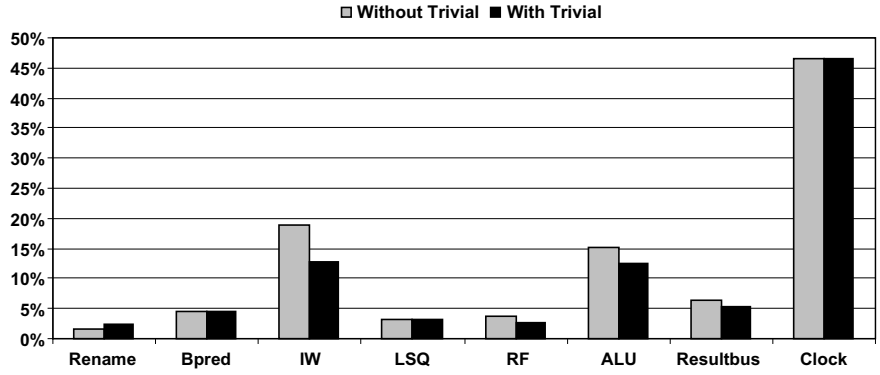
started the detailed performance simulation at instruction 20*100 million (2 billion) and stopped simulating at instruction 2.1 billion. The simulation points are used with the weights to compute the weighted average for a given metric. For example, if we collected performance statistics in terms of CPI for three simulation points (CPIsp1, CPIsp2, CPIsp3) with the weights (0.25, 0.25, 0.5), then the combined average of these points is: $CPI = 0.25 \times CPIsp1 + 0.25 \times CPIsp2 + 0.5 \times CPIsp3$. The computed CPI will be the estimate for the full execution [32].

5. Results and discussion

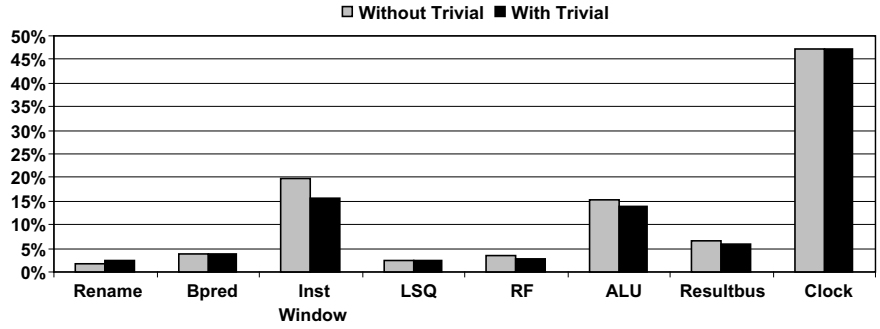
This section provides the energy reduction obtained by detecting and bypassing trivial operations at the decode stage of the pipeline. The dynamic energy usages of the different units of the processor model presented in Table 2 are shown in Fig. 7a and b for SPEC and EEMBC applications, respectively. While the cache is often one of the most power consuming parts of the entire system, we will focus on the micro-architectural (or pipeline) units instead. This is because trivial instruction elimination can only reduce the energy in these units.

In Fig. 7, the left bar shows the energy consumption of a unit of the base processor model which does not support the detection of trivial operations. The right bar in Fig. 7 shows the energy consumption of a particular unit of the processor model in which the register renaming unit is modified to detect and bypass trivial operations.

Fig. 7a shows that the instruction window (IW), ALU, result bus, and register file (RF) are among the most energy consuming units of the pipeline and these units collectively consume about 44% of the total energy. However, in accordance with the expectation of Section 3.2, detecting trivial operations at the decode stage can reduce the energy consumption of all of these units which is noticeable from the right bar of Fig. 7a. We can see from the right bar of Fig. 7a that the energy consumption of the instruction window, ALU, and the result bus has been reduced on average by 31%, 16%, and 14%, respectively. We can notice an average 28% energy reduction of the RF as well. This is due to the elimination of register reads and writes for the detected trivial operations. The new register renaming unit that is capable of detecting identity-trivial operation consumes 47% more energy than a conventional register renaming unit. However, the total energy consumption is reduced by 9.3%, on average, in the core pipeline units for the SPEC applications. Similarly, we can see from Fig. 7b that the overall energy reduction of the processor is on average 6% for the embedded applications and the IW, ALU, RF, and result bus are among the most energy consuming units of the processor. The following subsections present and discuss the energy consumption of the different units of the processor pipeline in detail.



(a) Energy usage of the simulated processor model using SPEC CPU2000.



(b) Energy usage of the simulated processor model using EEMBC1.1.

Fig. 7. Energy usage of the simulated processor model.

5.1. Energy reduction of the functional unit

It has already been mentioned that a trivial operation, regardless of its type, does not require a functional unit (integer ALU or FP ALU) and thus, reduces the energy consumption of the ALU. However, we have estimated the energy savings of the ALU because of the decode-trivial operations in this study. The results are shown in Fig. 8a and b for SPEC and EEMBC applications, respectively. As expected, we can see that the energy savings of the ALU is directly proportional to the amount of trivial operations present in the execution of the program. We get from Fig. 8a that on average 16% of the ALU energy can be reduced by bypassing trivial operations and this contributes an average reduction of 2.4% to the overall energy of the core pipeline units. We get from Fig. 8b that on average 9.5% reduction of the ALU energy is achievable for the selected embedded applications and this leads to an average reduction of 1.4% of the total energy of the pipeline.

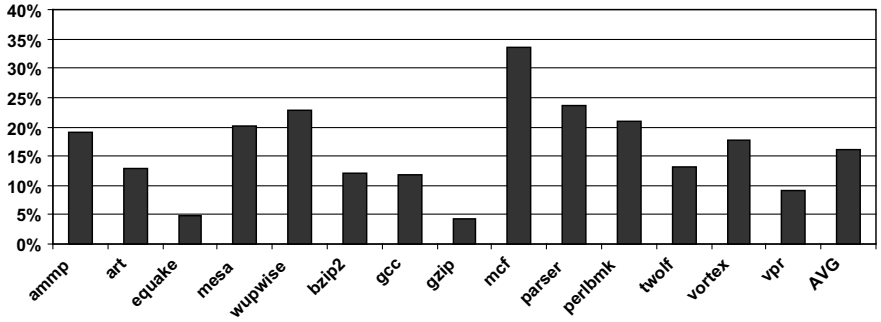
5.2. Energy reduction of the result bus and instruction window

Recall that a dependency between a decode-trivial operation and a subsequent instruction can be resolved by register remapping without broadcasting the result [25]. As a result, it is possible to save energy by eliminating the accesses to the result bus and in the wakeup-logic, part of the IW in this study [25]. Moreover, even the allocation of an entry in the IW is not required for a decode-trivial operation, thanks to the register renaming approach adopted in this study. Thus decode-trivial operations are expected to significantly reduce the energy consumption of the IW which is one of the most energy consuming units of processors. The results are shown in Fig. 9a and b for SPEC and EEMBC applications, respectively. In the figures, the left bar represents the energy saving in the IW whereas the right bar represents the savings in the result bus.

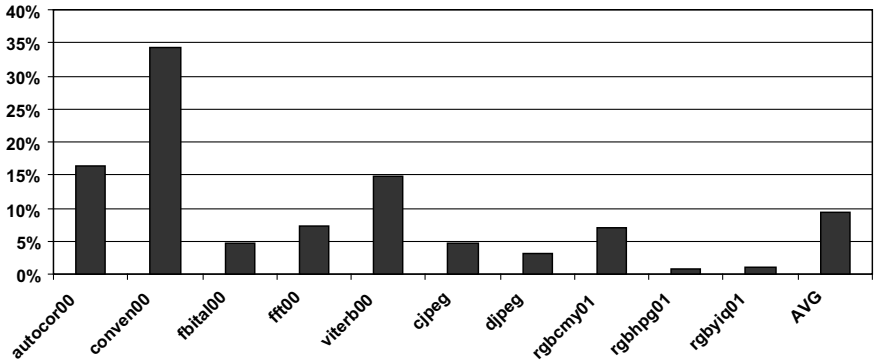
It is shown in Fig. 9a that the average energy consumption of the IW and the result bus can be reduced by 31% and 14%, respectively. The savings in the IW and the result bus correspond to the overall energy reduction of the processor by 5.9% and 0.9%, respectively. It is shown in Fig. 9b that the energy consumption of the IW and the result bus can be reduced by 20% and 9%, respectively for the embedded applications. The reductions in the IW and in the result bus contribute 4% and 0.6%, respectively to the overall energy reduction of the processor core.

5.3. Energy reduction of the register file

The energy efficiency improvement of the RF because of trivial operations is estimated in two ways. Firstly, when one of the operands dictates the results of the trivial operation, e.g. identity-trivial, the register read accesses are eliminated. Secondly, the register remapping method eliminates the write access by remapping the register id of the destination operand to the physical register containing

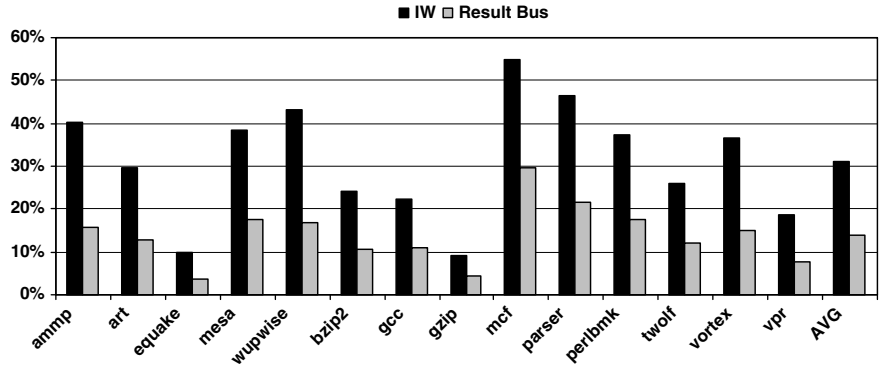


(a) Energy reduction of the ALU in SPEC CPU2000.

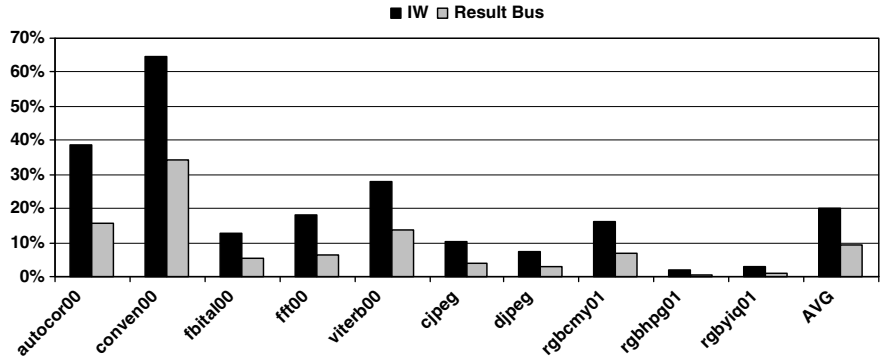


(b) Energy reduction of the ALU in EEMBC1.1.

Fig. 8. Energy reduction of the ALU.



(a) Energy reduction of the result bus and IW in SPEC CPU2000.



(b) Energy reduction of the result bus and IW in EEMBC1.1.

Fig. 9. Energy reduction of the result bus and instruction window (IW).

the result [25]. The obtained results for SPEC and EEMBC applications are shown in Fig. 10a and b, respectively. We find from Fig. 10a that an average 28% of the energy usage of the RF is reduced and this leads to an overall energy efficiency improvement of 1%. Similarly, Fig. 10b shows that the achievable energy reduction of the RF is 16% on average which is equivalent to an overall energy reduction of the processor core by 0.6%.

5.4. Impact on the register renaming unit

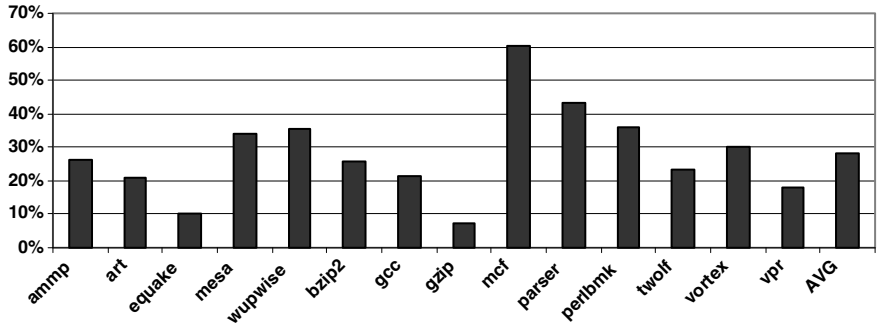
The power, delay and area impacts of introducing detection and bypassing of trivial operations in the register renaming unit have been evaluated as well. Table 4 shows the estimated values of power, area, and delay for the two implementations. The power presented in the table is for a clock period of 1.82 ns for both of the implementations. The table shows that the power consumption of the renaming unit increases by 47% and the area increases by 18.5%. The delay for the register renaming is increased by 28%.

The overall energy efficiency improvement of the processor supporting the detection and bypassing of trivial operations is shown in Fig. 11. Each bar in Fig. 11 consists of four sections which represent the energy savings of the ALU, RF, IW and result bus, respectively from bottom to top.

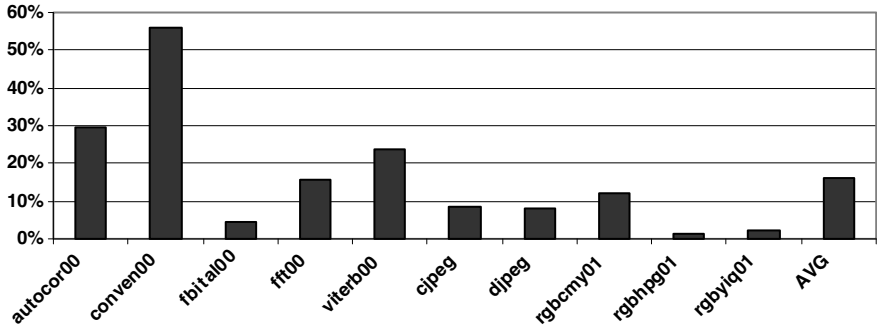
We can see from Fig. 11a that an average energy reduction of 10.1% is possible for the SPEC CPU2000 applications without considering the impact of the modified register renaming unit on energy. It is evident from Fig. 11a that the IW unit is the largest contributor (about 5.8% on average) and the ALU is the second largest contributor (about 2.4% on average) to the overall energy reduction. It is clear from Fig. 11a that the RF and the result bus have almost similar impacts on the overall energy reduction (1% for RF and 0.9% for result bus). However, the additional logic incorporated into the register renaming unit to detect and bypass trivial operations increases the overall energy usage of the processor by 0.8%, on average. Thus, the net energy efficiency improvement of the processor core is on average 9.3% for the selected SPEC CPU2000 applications. We can observe a similar trend for the embedded applications which is shown in Fig. 11b. We get that the average achievable energy saving of the processor for the embedded applications is 5.6%. It is noticeable from Fig. 11b that every application except rgbhpg01 and rgbyiq01 contributes significantly to the overall energy reduction of the processor core.

6. Related work

Studies have already shown that there exists significant result redundancy in programs [14,23,31,35]. Moreover, a



(a) Energy reduction of the register file in SPEC CPU2000.



(b) Energy reduction of the register file in EEMBC1.1.

Fig. 10. Energy reduction of the register file.

Table 4
The results of the register renaming unit

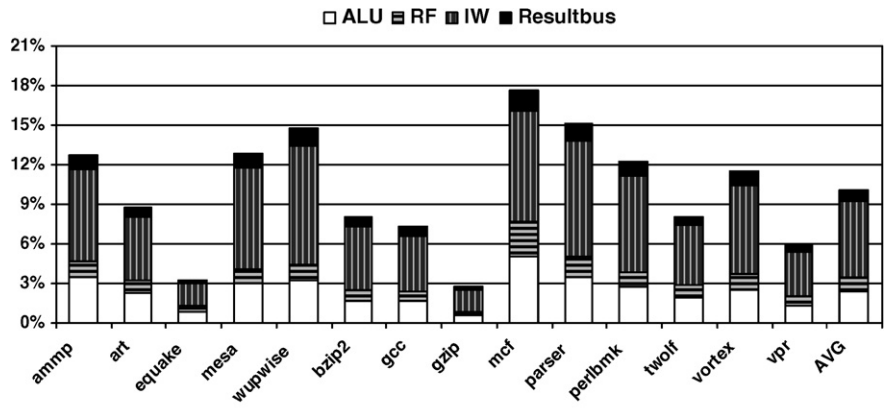
Renaming unit	Power at 550MHz (mW)	Area (μm^2)	Delay (ns)
Without trivial	30.5	211,000	1.42
With trivial	44.9	250,000	1.82

few values are more likely to be used than others in programs [39]. Based on these studies, various hardware techniques such as 'instruction memoization' (IM) [11,18,28,30], 'value prediction' (VP) [15,23,24] and 'instruction reuse' (IR) [27,33,40] have been proposed to exploit data value locality in programs. VP is speculative which does not bypass execution and may incur misprediction penalty. VP may increase both the branch misprediction penalty by either causing more mispredictions or delaying branch resolution and the demand for resources since instructions executed with wrong inputs need to be re-executed [34]. On the contrary, our proposed technique is non-speculative which not only reduces the execution latency of operations but also decreases structural hazard by bypassing them.

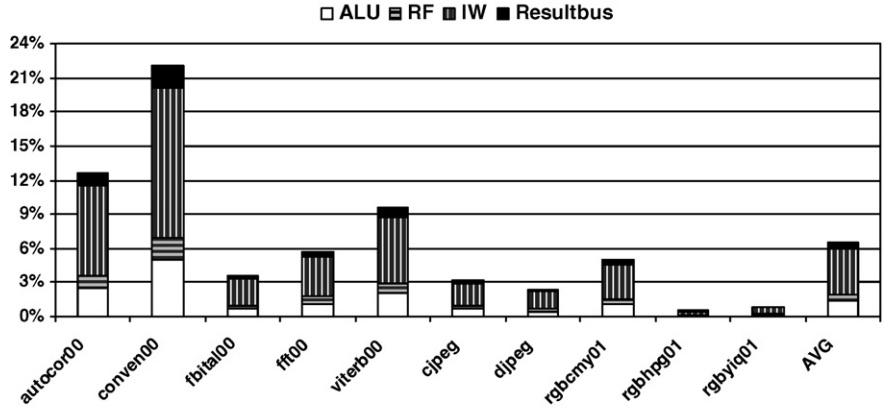
VP is speculative whereas IM and IR are non-speculative. However, each of the three techniques exploits redundancy in programs by obtaining results of instructions from their previous executions. They store the previous results in a data-cache like special hardware table named differently in different techniques – result cache in [30], memo-table in [11], reuse buffer (RB) in [33,37], redundant computation buffer in [27] and so on. In addition, when a new instruction is encountered, a lookup into the table is performed to bypass the execution of the instruction in IM and IR. IR as a mechanism of reducing power consumption has been evaluated by several proposals [3,29,37]. It has been shown in [37] that a 1024-entry two-way set-associative ALU RB increases the power consumption of a processor by 3–4%, on average, over a conventional processor. However, detecting and bypassing trivial operations do not require such extra hardware table. Thus our technique is more efficient in terms of area and energy with respect to the other techniques.

The detailed study of IR presented in [10] has shown that speedups of more than 3% are hard to obtain for most applications and this is due to the reduced opportunity of IR brought about by unavailability of instruction operands early in the pipeline. On the contrary, our technique has shown that many of the arithmetic instructions can be detected to be trivial at the decode stage of the pipeline and hence, bypassed even if only one of the operands is available, for example, 1-op decode-trivial. However, this is not possible in IR and IM.

The first work on trivial computations was published by Richardson [30]. He restricted the definition of trivial computations to certain multiplications (by 0, 1, −1), divisions ($X \div Y$ with $X = \{0, Y, -Y\}$), and square roots of 0 and 1 and achieved an average speedup of 2% [30]. The works [2,42] have significantly extended the set of trivial operations. This study is based on our previous work presented in [25] and different from [2,42] in several ways. Our work presents a new classification of trivial operation with respect to identity and inverse elements. This is motivated from the need to detect and bypass trivial operations as



(a) Overall energy reduction of the processor in SPEC CPU2000.



(b) Overall energy reduction of the processor in EEMBC1.1.

Fig. 11. Overall energy reduction of the simulated processor.

early as possible. Moreover, [42] has focused on the performance impact of trivial operations whereas we have focused on the reduction of energy consumption.

Independently of our work, Atoofian and Baniasadi [2] recently presented a study with similar goals as ours. However, unlike the observations in our paper, they found that only 50% of the trivial operations could be detected at the decode stage and showed much smaller reductions in energy consumption. Moreover, their study did not reveal in detail the individual energy reduction in each pipeline unit and only presented the overall reduction of the energy of the entire pipeline. By contrast, our study shows the dynamic energy usage of the different units of a processor such as the register file, the functional units, the result bus and the instruction window infrastructure. Unlike [2,25,42], we have evaluated the applications from EEMBC1.1 benchmark suite as well. Most importantly, none of the previous works [2,25,42] on trivial computations has estimated the power consumption of the trivial detection logic which has been done in this work.

7. Conclusion

In this work, we have studied the energy savings of processors by detecting and bypassing trivial operations as early as at the decode stage of the pipeline. We have found that 14% and 11.4% of the total executed instructions are trivial for the selected applications of SPEC CPU2000 and EEMBC1.1 benchmark suites, respectively. We have identified that a vast majority of them are identity-trivial −94% for the SPEC applications and 65% for the embedded applications. We have proposed an architectural technique which modifies the conventional register renaming to detect such trivial operations as early as at the decode stage of the pipeline. We have shown that the proposed technique is capable of detecting about 12% of the total executed instructions as trivial at the decode stage of the pipeline for SPEC applications while the average is 6.4% for embedded applications. This early detection and bypassing of trivial operations improves the average energy efficiency of the processor core by 9% in SPEC CPU2000 and by 6% in EEMBC1.1.

Acknowledgements

This research has been sponsored by the EU funded SARC Integrated Project and the FlexSoC research program funded by the Swedish Strategic Research Foundation. All the authors are members of the EU FP6 funded HiPEAC Network of Excellence.

References

[1] A. Aggarwal, M. Franklin, Energy efficient asymmetrically ported register files, in: Proc. ICCD, 2003, pp. 2–7.

[2] E. Atoofian, A. Baniasadi, Improving energy efficiency by bypassing trivial computations, in: Proc. of the First Workshop on High-Performance Power-Aware Computing, 2005.

[3] M. Azam, P. Franzon, W. Liu, T. Conte, Low power data processing by elimination of redundant computations, in: Proc. ISLPED, 1997, pp. 259–264.

[4] R.I. Bahar, G. Albera, S. Manne, Power and performance tradeoffs using various caching strategies, in: Proc. ISLPED, 1998, pp. 64–69.

[5] S. Balakrishnan, G.S. Sohi, Exploiting value locality in physical register files, in: Proc. MICRO-36, 2003, pp. 265–276.

[6] D. Brooks, M. Martonosi, Dynamically exploiting narrow width operands to improve processor power and performance, in: Proc. HPCA, 1999, pp. 13–22.

[7] D. Brooks, V. Tiwari, M. Martonosi, Wattch: A framework for architectural-level power analysis and optimizations, in: Proc. ISCA-27, 2000, pp. 83–94.

[8] D. Burger, T. Austin, The SimpleScalar Tool Set Version 2.0, University of WM, Computer Sciences Department, Technical Report 1342, 1997.

[9] R. Canal, A. Gonzalez, J.E. Smith, Very low power pipelines using significance compression, in: Proc. MICRO-33, 2000, pp. 181–190.

[10] D. Citron, D.G. Feitelson, Revisiting instruction level reuse, in: Proc. Workshop on Duplicating, Deconstructing and Debunking (WDDD), 2002.

[11] D. Citron, D. Feitelson, L. Rudolph, Accelerating multi-media processing by implementing memoing in multiplication and division units, in: Proc. ASPLOS, 1998, pp. 252–261.

[12] Synopsys, Design Compiler User Guide Version W-2004.12, December, 2004.

[13] EEMBC1.1 Benchmark Suite, <http://www.eembc.org/>.

[14] F. Gabbay, A. Mendelson, Speculative execution based on value prediction, Technical Report EE Department TR 1080, Technion – Israel Institute of Technology, 1996.

[15] F. Gabbay, A. Mendelson, Using value prediction to increase the power of speculative execution hardware, ACM Trans. Comp. Sys. (TOCS) 16 (3) (1998) 234–270.

[16] R. Gonzalez, A. Cristal, D. Ortega, A. Veidenbaum, M. Valero, A content aware integer register file organization, in: Proc. ISCA-31, 2004, pp. 314–324.

[17] R. Gonzalez, M. Horowitz, Energy dissipation in general purpose microprocessors, IEEE J Solid-State Circ 31 (9) (1996) 1277–1284.

[18] S.H. Harbison, An architectural alternative to optimizing compilers, in: Proc. ASPLOS, 1982, pp. 57–65.

[19] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, third ed., Morgan Kaufman Publishers, San Fransisco, CA, 2003.

[20] I. Kim, M.H. Lipasti, Half-price architecture, in: Proc. ISCA-30, 2003, pp. 28–38.

[21] M.G.J. Kin, W.H.M. Smith, The Filter Cache: An energy efficient memory structure, in: Proc. MICRO-30, 1997, pp. 184–193.

[22] P. Kongetira, K. Aingaran, K. Olukotun, Niagara: A 32-Way Multithreaded SPARC Processor, in: IEEE Micro, 2005, pp. 21–29.

[23] M.H. Lipasti, J.P. Shen, Exceeding the dataflow limit via value prediction, in: Proc. MICRO-29, 1996, pp. 226–237.

[24] M.H. Lipasti, C.B. Wilkerson, J.P. Shen, Value locality and load value prediction, in: Proc. ASPLOS, 1996, pp. 138–147.

[25] I.M. Mafijul, P. Stenstrom, Reduction of energy consumption in processors by early detection and bypassing of trivial operations, in: Proc. IC-SAMOS, 2006, pp. 28–34.

[26] S. Manne, A. Klauser, D. Grunwald, Pipeline gating: speculation control for energy reduction, in: Proc. ISCA-25, 1998, pp. 132–141.

[27] C. Molina, A. Gonzalez, J. Tubella, Dynamic removal of redundant computations, in: Proc. ICS, 1999, pp. 474–481.

[28] S.F. Oberman, M.J. Flynn, On division and reciprocal caches, Technical Report CSL-TR-95-666, Stanford University, 1995.

[29] D.V. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, P.M. Kogge, Energy-efficient issue queue design, IEEE Trans. VLSI Sys. 11 (5) (2003) 789–800.

[30] S. Richardson, Exploiting trivial and redundant computation. in: Proc. ARITH18, 1993, pp. 220–227.

[31] Y. Sazeides, J.E. Smith, The predictability of data values, in: Proc. MICRO-30, 1997, pp. 248–258.

[32] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, Automatically characterizing large scale program behavior, in: Proc. ASPLOS, 2002, pp. 45–57.

[33] A. Sodani, G.S. Sohi, Dynamic instruction reuse, in: Proc. ISCA-24, 1997, pp. 194–205.

[34] A. Sodani, G.S. Sohi, Understanding the differences between the value prediction and instruction reuse, in: Proc. MICRO-31, 1998, pp. 205–215.

[35] A. Sodani, G.S. Sohi, An empirical analysis of instruction repetition, in: Proc. ASPLOS, 1998, pp. 35–45.

[36] SPEC Benchmark Suite. <http://www.spec.org/>.

[37] G. Surendra, S. Banerjee, S.K. Nandy, Instruction reuse in SPEC, media and packet processing benchmarks: a comparative study of power, performance and related microarchitectural optimizations, J. Embed. Comp. 2 (1) (2006) 15–34.

[38] L. Tran, N. Nelson, F. Nagi, S. Dropsho, M. Huang, Dynamically reducing pressure on the physical register file through simple register sharing, in: Proc. ISPASS, 2004, pp. 78–87.

[39] J. Yang, Y. Zhang, R. Gupta, Frequent value locality and value centric data cache design, in: Proc. ASPLOS, 2000, pp. 150–159.

[40] J. Yang, R. Gupta, Load redundancy removal through instruction reuse, in: Proc. ICPP, 2000, pp. 61–68.

[41] K.C. Yeager. The MIPS R10000 superscalar microprocessor, in: IEEE Micro, 2004, pp. 28–40.

[42] J.J. Yi, D.J. Lilja, Improving processor performance by simplifying and bypassing trivial computations, in: Proc. ICCD, 2002, pp. 462 – 465.