An XOR-based approach to merging entries for instruction register files

Naoki Fujieda and Shuichi Ichikawa
Department of Electrical and Electronic Information Engineering,
Toyohashi University of Technology
fujieda@ee.tut.ac.jp, ichikawa@tut.jp

***Abstract*—The instruction register file (IRF) is an attractive approach to reduce power consumption, which is essential to many embedded systems. However, the previously proposed IRF implementation is not efficient in merging similar instructions into a single entry in the IRF. In this paper, we propose an XOR-based merging approach that achieves higher efficiency in grouping instructions with simple hardware. Our evaluation shows that the proposed approach can convert 19.6% more dynamic instructions into references of the IRF than the previous techniques, and that it reduces the number of instruction fetches from the cache by 4.8% on average.**

I. Introduction

Reducing power consumption is essential for most embedded systems. In particular, it is very important to improve the efficiency of instruction fetch logic in embedded processors because it is the most power-hungry part in some processors [1].

The major reasons for high power consumption of instruction fetch are twofold. First, the information density of fetched instructions is very low, so processors retrieve much less information from them than their theoretical limit. The instruction set architectures (ISAs) of RISC processors are usually designed to simplify decode logic rather than to shorten program codes. An additional, high-density ISA can be a solution of this problem. For example, some MIPS embedded processors support MIPS16 ISA [2] where an instruction has 16 bits, a half-length of an original MIPS instruction. MIPS16 instruction set includes a fixed subset of original MIPS instructions. Shrivastava *et al.* [3] reported that the code size of MIPS32/16 program is reduced by 22% on average with a careful selection of instructions. However, the efficiency of such dual-ISA designs is heavily dependent on applications: if most of the executed instructions are not included in the subset, they must be written in the traditional MIPS instructions and thus the number of instruction fetches is almost unchanged. Second, even though the L1 instruction cache is much smaller than lower-level caches, it is still a large component in the context of embedded processors. A lot of proposals include a smaller storage than the L1 and help processors to supply instructions from it. A filter cache [4] and a loop cache [5] operate as L0 instruction caches, which try to store heavily-reused instructions in different strategies. A scratchpad memory (SPM) [6] has a physical address that is separated from the main memory. It is managed by software to keep the most frequent instructions (and/or data).

An instruction register file (IRF) [7] is a remedy for both of the problem. The IRF is a small memory storing the most common expressions of instructions specified by the compiler. Fig. 1 describes the concepts of the filter cache, the SPM, and the IRF. A remarkable difference of the IRF from the others is that the storage is accessed by an index written in fetched instructions rather than the program counter. By packing common instructions into a single instruction with these indexes, it improves the information density of instructions and reduces the number of fetches from the L1.

To compensate for lack of entries in the IRF, we can merge multiple instructions with similar expressions into a single group. An instruction that is not the most common in a group is referenced with the index corresponding to the group and a parameter that has the same length as the index.

The problem of the previously proposed IRF implementation is the inefficiency in grouping instructions. We propose an XOR-based approach to merging instructions more efficiently. It increases the number of continuous references to the IRF, that is, the chance of instruction packing. We evaluate the methods with traces of practical benchmarks and show that our approach reduces the number of instruction fetches.

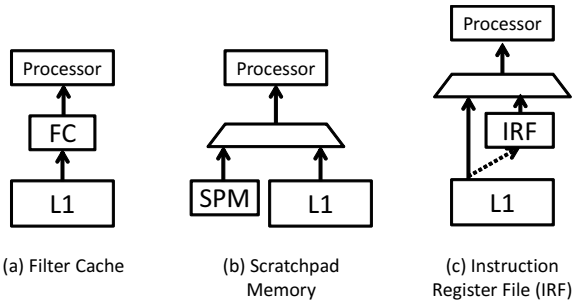

Fig. 1. Different approaches for reducing fetch power consumption with small RAMs.

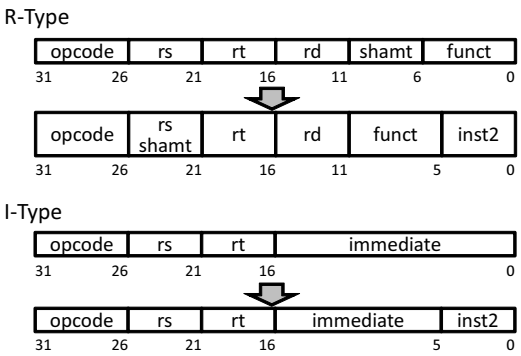

Fig. 2. Loosely packed instructions where a regular instruction and an IRF instruction are merged [7].

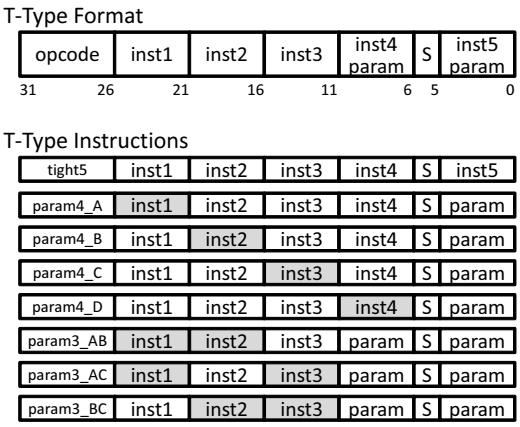

Fig. 3. Tightly packed instructions that can contain up to 5 IRF instructions [7].

II. Instruction Register File

A. Overview

The concept of the instruction register file (IRF) [7] is shown as (c) in Fig. 1. The IRF is placed between instruction fetch and instruction decode stages and accessed by an index written in fetched instructions. The target instruction set (MIPS in [7]) is modified so that some instructions can include a reference (or references) to the IRF. If a fetched instruction has a reference, the corresponding instruction in the IRF is read and sent to the decoder. In other words, frequently executed instructions are given different expressions, which are much shorter than usual ones, by the IRF. They are called IRF instructions. The modified instruction set has two kinds of instructions that include multiple instructions: loosely packed and tightly packed.

The loosely packed instructions modify the existing R-Type and I-Type instruction formats of MIPS. Fig. 2 shows the difference between the formats of normal MIPS instructions and those of loosely packed instructions. The *shamt* field (in the R-Type) or a part of the *imm* field (in the I-Type) is replaced by the 5-bit *inst2* field, which indicates an index for the IRF reference. When an R-Type or I-Type instruction is followed by an IRF instruction, they can be packed into a loosely packed instruction; however, if the latter instruction does not reside in the IRF, they cannot be packed and the *inst2* field of the former instruction is set to (the index of) a `nop`.

The tightly packed instructions add the T-Type format to the traditional MIPS instruction formats. Fig. 3 describes the T-Type format and the tightly packed instructions. It is composed of a 6-bit *opcode* field, five 5-bit *inst*/*param* fields, and an S bit: a supplement bit to *opcode*. Each *inst*/*param* field represents an index of the IRF or a parameter, which is described later, attached to an IRF instruction. Thus each tightly packed instruction contains up to five IRF instructions or three IRF instructions with two parameters. They consist of eight different instructions for specifying the number of parameterized IRF instructions and their positions. Parameterized IRF instructions are shown as shaded fields in Fig. 3.

By parameterizing instructions, we can merge multiple instructions with similar expressions into a single entry in the IRF. A 32-entry IRF that is referenced only by a 5-bit index contains 32 kinds of instructions, which is too few to cover most of the frequent instructions in some cases. On the other hand, if each entry represents 32 instructions differentiated by a 5-bit parameter, the IRF will have as many as 1024 kinds of IRF instructions. An instruction that is the most commonly used in a group is stored in the IRF as a default so that it can be referenced without a parameter. The other instructions in the group are reconstructed by an index and a parameter.

The previously proposed IRF implementation uses three techniques for merging instructions: compressing immediate values with an immediate table, merging short-distance branches, and using indirect register specifiers [7]. We introduce the first two techniques; we do not consider the last technique in this paper because it has little effect on efficiency of merging despite some shortcomings [7].

The first technique can be applied to I-Type instructions except branches. An immediate table (IMM) is a 32-entry table that stores the most common immediate values. If two instructions have the same expressions in the fields other than *immediate*, and both the immediate values exist in the table, they can be merged.

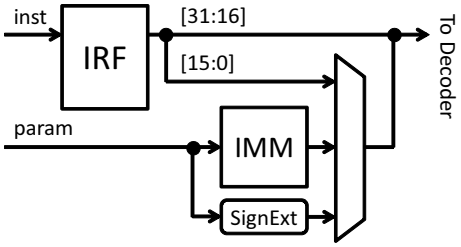

Fig. 4. Extraction of instructions from the IRF in the original method.

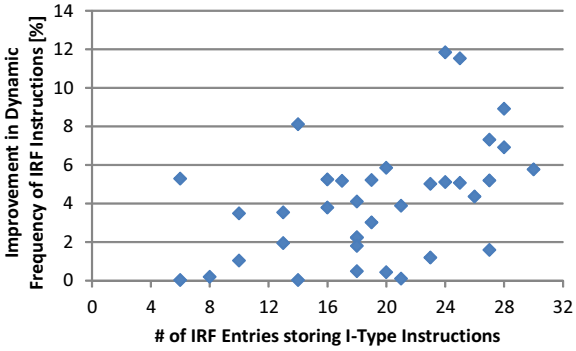

Fig. 5. The number of I-Type IRF instructions vs. coverage improvement (R = 0.459)

The second technique is for I-Type branch instructions. In these instructions, immediate values are parameterized with a 5-bit signed integer, which is sign extended to 16-bit, rather than an index of the immediate table. An immediate value in such an instruction means the PC-relative branch offset, or the distance to the branch target. While it will be shortened in consequence of instruction packing, it cannot get longer. Therefore, when the instruction group corresponding to a branch is listed in the IRF, once the immediate value can be expressed in 5-bit, the branch is then treated as an IRF instruction regardless of the subsequent recalculation of the distance.

Fig. 4 shows how parameterized IRF instructions are reconstructed in the original implementation with these two techniques. The two inputs, *inst* and *param*, are retrieved from specific fields of a packed instruction. The IRF and the immediate table are 32-bit and 16-bit RAMs, respectively. Both of them have 32 entries. SignExt means a sign extension unit where the 5-bit parameter is sign extended to a 16-bit immediate value. The upper 16 bits of an instruction read from the IRF are used without modification; however, the lower 16 bits depend on which format the instruction is, whether the corresponding entry is parameterized, and whether the instruction has a parameter.

B. Shortcomings

The techniques for merging entries in the previously proposed implementation mainly have two problems on efficiency, which we discuss in this section.

The first problem is that these techniques can be applied to only I-Type instructions. It is true that about a half of executed MIPS instructions are I-Type, while it is not trivial to decide the instructions to be listed in the IRF. The number of IRF entries storing I-Type instructions is likely to affect the efficiency of instruction merging.

Fig. 5 depicts the relation between the number of entries in the IRF that contain I-Type instructions and the increase of the ratio of executed (dynamic) IRF instructions. The X-axis is the number of entries for I-Type instructions. The Y-axis shows how many instructions become IRF-referable through the parameterization. Points in the graph represent the results of individual applications. The conditions for the evaluation are the same as what we will describe in Section IV. The correlation coefficient (R) of the two axes calculated from these results was 0.459. Through the preliminary evaluation, we confirmed that the number of IRF entries for I-Type varied widely with applications and that it was well related to the efficiency of merging. Consequently, we think that merging techniques should be able to be applied to both I-Type and R-Type instructions.

The second problem is the imbalance of the immediate table between the efficiency and the hardware overhead. By consulting IRF instructions parameterized with the immediate table, we notice that most of the variation of immediate values resides in the few lowest bits. In other words, there is little chance that the table provides significant benefit over just parameterizing with the 5 lowest bits. As a result, the immediate table may not be worth its hardware cost, which is about a half as large as the IRF.

III. XOR-based Merging Method

In this section, we propose an XOR-based merging method to improve the efficiency of instruction merging and therefore the efficiency of the IRF. From the discussion of the shortcomings of the previous implementation, we think that an efficient merging technique should be capable of being applied to most of the instructions and be implementable with simple hardware. Our method is designed so that it can meet both of the requirements.

A. Coding of IRF Instructions

Fig. 6 shows the formats of IRF entries. We call these 36-bit formats "codes" in this paper. Each code has additional 4 bits, S, T, D, and I bits, as flags of parameterization. They determine whether *rs*, *rt*, *rd*, and (the 5 lowest bits of) *immediate* fields in the entry will be XORed with a parameter, respectively. R-Type instructions use S, T, and D bits, and I-Type instruction use S, T and I bits. J-Type instructions, `j` (jump) and `jal` (jump and link), does not utilize any additional bits: they do not benefit from parameterization. However, many of them can be translated into I-Type unconditional branches (`b` and `bal` pseudoinstructions, which actually are `beq` and `bgezal` with tautologies [8], respectively) and therefore some of them may

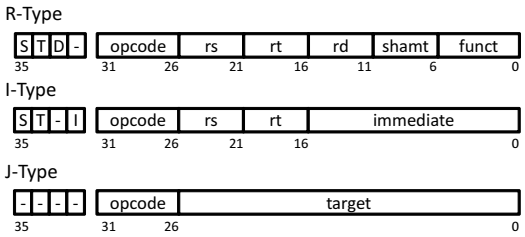

Fig. 6. The format of IRF entry for each type of MIPS instruction.

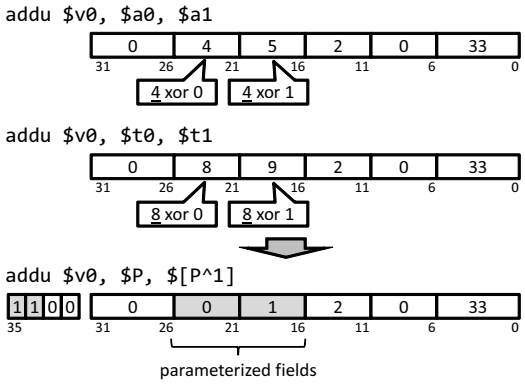

Fig. 7. Example: merging two instructions into a single IRF entry.

be included in the IRF instructions. The unused bits are set to zero to keep the corresponding fields unchanged.

Fig. 7 illustrates an example of grouping two similar instructions, `addu $v0, $a0, $a1` and `addu $v0, $t0, $t1`. They differ in two source registers *rs* and *rt*. The actual register numbers of *rs* and *rt* in the former instruction are 4 and 5, which can be expressed as 4 XOR 0 and 4 XOR 1, respectively. Similarly, those of the latter instruction, 8 and 9, can be described as 8 XOR 0 and 8 XOR 1, respectively. We now introduce a pseudoinstruction of `addu $v0, $P, $[P^1]` using a parameter `P`, where `^` is an XOR operator. It is encoded in an IRF entry as described in the figure, asserting S and T bits that correspond to *rs* and *rt* fields, respectively. The former instruction can be retrieved from the pseudoinstruction with a parameter 4, while the latter can be retrieved with 8.

Along with the pseudoinstruction shown above, `addu $v0, $[P^4], $[P^5]` is also a possible pseudoinstruction that coordinates the two `addu` instructions. In this case, the parameters become 0 and 12. In general, there are 32 possible pseudoinstructions that differ in the corresponding parameters from each other. We define the normalized form of a code as the code the highest parameterized field of which is zero. For example, the code for `addu $v0, $P, $[P^1]` shown in Fig. 7 is a normalized form because the highest parameterized field, or *rs*, is zero.

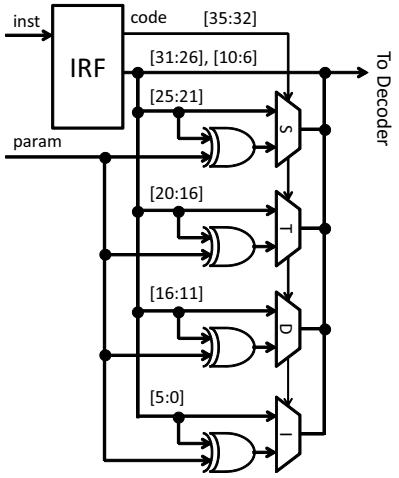

Fig. 8. Extraction of Instructions from the IRF in the proposed method.

B. Difference in Hardware Organization

Fig. 8 shows how IRF instructions are extracted in our method in hardware. The bit length of the IRF is increased from 32 bits to 36 bits to store the most frequent codes, rather than instructions, in individual applications. The *opcode* and *funct* fields are sent to the decoder without modification. The other fields are selected by the corresponding flag bits from either the value in the entry or the XOR of it and *param*.

We compare our merging method with the previous techniques using Fig. 4 and Fig. 8. Even though our method requires 4 extra bits per entry in the IRF, it only spends a quarter of the 16-bit immediate table. In addition, though selection logic is quite simple in both methods, when we implement methods in an FPGA, our implementation may enjoy additional benefit from an optimization of merging an XOR and a selector into a single 3-input look-up table. Therefore, we think that our method can be implemented with smaller hardware than the previous implementation.

C. Selection of IRF Instruction Groups

There are eight normalized codes corresponding to each IRF instruction (except J-Type) because the number of ways to choose the flag bits is $2^3 = 8$. Finding the optimal combination of codes that maximizes the number of IRF instructions executed or minimizes the size of the modified program is too complex to complete in a realistic time. So we use a heuristic, which is similar to a greedy algorithm that is used in the original IRF proposal when indirect register specifiers are applied [7].

Fig. 9 shows the algorithm for selecting a set of codes in the IRF from an instruction profile that is gathered by static analysis or dynamic profiling. Each instruction in the profile has its occurrence rate. The function coding_candidates (in lines 4 and 19) gives a set of all normalized codes for the corresponding instruction. The function bit_concat (in line 17) returns the bit concatenation of the inputs. The process of generating IRF contents is as follows: Before the main loop,

```
 1: PROFILE ← instruction profile
 2: INSTS ← empty associative array
 3: for inst in PROFILE except nop do
 4:     for code in coding_candidates(inst) do
 5:         if INSTS[code] does not exist then
 6:             append code to INSTS
 7:         end if
 8:         append inst to INSTS[code]
 9:     end for
10: end for
11: IRF[0] ← code for nop
12: for i in [1..31] do
13:     find code that has the highest frequency of INSTS[code]
14:     flags ← code[35..32]
15:     def_inst ← most frequent instruction in INSTS[code]
16:     IRF[i] ← bit_concat(flags, def_inst)
17:     for inst in INSTS[code] do
18:         for candidate in coding_candidates(inst) do
19:             remove inst from INSTS[candidate]
20:         end for
21:     end for
22: end for
```

Fig. 9. Pseudocode for selecting IRF entries.

it calculates the sets of instructions that belong to each of the possible codes and stores to *INSTS* (lines 2 through 10). Since the entry 0 is reserved for `nop` (line 11), the main loop determines the 31 most common codes, rather than 32. In the main loop, it first finds the code whose sum of frequency is the highest in *INSTS*, that is, the most frequent code (line 13). Then it sets the most frequent instruction in the code as a default one, which is stored in the IRF (line 14 through 16). As a result, default instructions, which are referenced without a parameter, come to be extracted by setting *param* in Fig. 8 to zero. Lastly the instructions in the selected code are excluded from the corresponding lists (line 17 through 21). This exclusion can be skipped in the last iteration.

IV. Evaluation

In this section, we compare the methods for merging instructions through evaluations. Two scales of efficiency are measured: the dynamic frequency of IRF instructions being executed and the decrease in the number of instruction fetches from the cache. The former scale corresponds to the chance of instruction packing. The latter is strongly connected to reduction of fetch power.

A. Methodology

We use all of the 36 benchmarks of MiBench [9] and make the instruction profiles from the traces of them. To get the traces, we use a modified version of SimMips version 0.7.5 [10]. The benchmarks are compiled with gcc 4.7.3, uClibc 0.9.33.2, and binutils 2.21. The instruction set is MIPS32 Release 1 with floating point instructions. The compile options are same as the defaults of MiBench except asserting a flag for static compilation (`-static`). Some programs is slightly modified so that compile errors due to the difference of compiler versions may be removed.

We define three settings as follows:

- **No Grouping** stands for the IRF without any of the instruction merging techniques. It is referred to for a baseline of efficiency.
- **Conventional** applies two existing techniques that have been described in Section II.
- **Proposed** utilizes the XOR-based merging that we have introduced in Section III.

In the evaluation, there are three major limitations in instruction packing because of the selection of ISA or the ease of the calculation. First, some R-Type instructions that require both *rs* and *shamt* fields cannot be transformed into loosely packed instructions. Most of them are floating point instructions [8]. Second, some I-Type instructions fail to express their immediate values in 11 bits (See Fig. 2). It will cause addition and modification of instructions and thus decreases the efficiency of instruction packing. However, we do not consider it because it is applied to all of the settings. Lastly, we do not attempt to perform an iterative instruction packing. Packing may shorten the distance to the target in a branch and then some of the modified branch can be included by the IRF instructions. Though it slightly improves the efficiency of merging techniques, we think that the chance of improvement is rarely different between Conventional and Proposed because it is applicable to both of them.

B. Dynamic Frequency

We show the dynamic frequency of IRF instructions being executed in Fig. 10. The X-axis is the name of a group of MiBench. The Y-axis is the average of dynamic frequency in the group. The rightmost *average* bars stand for the average frequency of all the applications. The dark bars represent coverage of the default (parameter-free) IRF instructions. Our method improved the average frequency by 19.6% (or 10.5 percentage points) over the previous methods. In particular, in the 7 traces of the Security benchmark group, the improvement was as much as 46.7% (or 22.0 points) on average.

We think that the reasons why the Security benchmarks fit in our method are twofold: One is their high proportions of bit operations. Most of them have the R-Type format, which cannot be parameterized in the previous methods. The other is their high number of registers used in loops. Our method can absorb the difference in register numbers. Therefore it succeeded in merging many instructions into a few IRF entries.

C. The number of instruction fetches

Fig. 11 shows the reduction in the number of instruction fetches over No Grouping. The X-axis is the same as that of Fig. 10. The Y-axis is the average decrease of the number of instruction fetches from the L1 cache. Our method reduced the number of fetches by 4.8% over Conventional.

In the Automobile and the Network benchmark groups, our method negatively affected the number of fetches (increased by 0.5% and 3.4%, respectively). We think that the negative effect comes from the decrease of the packing efficiency by parameterized instructions. While a default IRF instruction can be expressed in a single field in tightly packed instructions, a

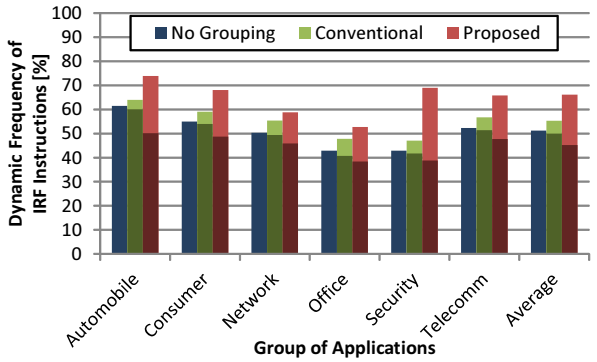

Fig. 10. Improvement in dynamic frequency of IRF instructions. Light bars represent IRF instructions with non-default parameters.

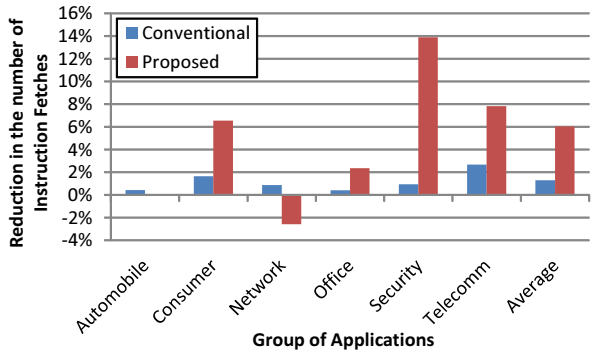

Fig. 11. Reduction in the number of instruction fetches. All values are relative to No Grouping.

parameterized instruction requires two fields. If the number of required fields in continuous IRF instructions increases, additional packed instructions may be inserted. Fig. 12 shows the difference between Conventional and Proposed in instruction packing of the most frequently executed loop in the *qsort* benchmark. Instructions with a circle and two circles stand for parameter-free and paramterized IRF instructions, respectively. In Conventional, all the instructions were referenced without parameters and the number of required fields in the loop was 8. The loop became only two tightly packed instructions. In Proposed, on the other hand, some instructions got to require parameters as a result of merging and the number of required fields became 11. Therefore, our method decreased the efficiency of packing in this case because the loop could not be expressed by two packed instructions. The result in Fig. 10 surely showed that our method decreased the frequency of default IRF instructions being executed. We might improve the algorithm for selecting codes by carefully checking the negative effect of merging multiple frequent instructions.

Nevertheless, our method showed remarkable decrease of fetches (12.8% on average) in the Security benchmarks. The number of continuous IRF instructions in major loops was drastically increased there. For example, Fig. 13 shows the difference in instruction packing of a part of the AES encryption routine in the *rijndael* benchmark. The previous method did not leverage the similarities of instructions, though some instructions were loosely packed. With our method, most of the instructions in the routine were able to be retrieved from the IRF and compressed into tightly packed instructions. Although we do not evaluate the fetch power itself, judging from these results, we think that our method make a significant contribution to fetch power savings.

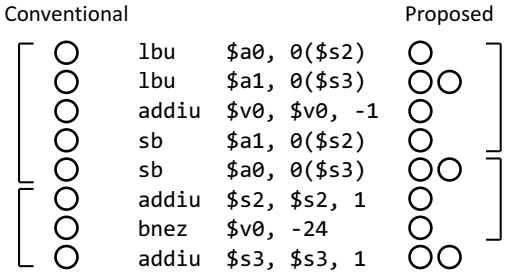

Fig. 12. The difference in instruction packing in the *qsort* benchmark.

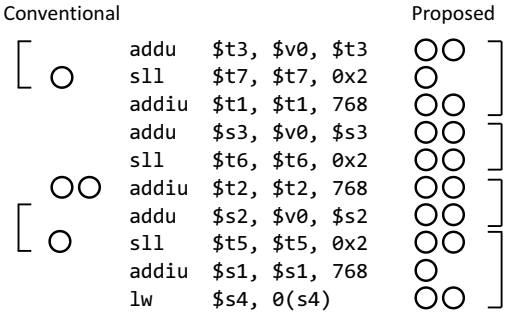

Fig. 13. The difference in instruction packing in the *rijndael* benchmark.

V. Conclusion

This work proposed an XOR-based instruction merging scheme, which efficiently utilizes the limited capacity of the IRF to reduce the power consumption of instruction fetch logic. According to our evaluation results, 19.6% more instructions were fetched from the IRF, the number of instruction fetches was decreased by 4.8% compared to the previous approach.

The following items are left for future studies. First of all, the power consumption of instruction fetch should be investigated more precisely. It is also important to apply our scheme, which is heavily dependent on MIPS ISA, to other ISAs.

References

[1] J. Montanaro *et al.*, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1703–1714, 1996.

[2] K. D. Kissell, "MIPS16: High-density MIPS for the Embedded Market," Silicon Graphics MIPS Group, Tech. Rep., 1997.

[3] A. Shrivastava, P. Biswas, A. Halambi, N. Dutt, and A. Nicolau, "Compilation framework for code size reduction using reduced bit-width ISAs (rISAs)," *ACM Transaction of Design Automation Electronic System*, vol. 11, no. 1, pp. 123–146, 2006.

[4] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997, pp. 184–193.

[5] L. H. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," in *Proceedings of the 1999 international symposium on Low power electronics and design*, 1999, pp. 267–269.

[6] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002, pp. 73–78.

[7] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving Program Efficiency by Packing Instructions into Registers," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005, pp. 260–271.

[8] D. Sweetman, *See MIPS Run Linux Second Edition*. Morgan Kaufmann, 2006.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *2001 IEEE International Workshop on Workload Characterization*, 2001, pp. 3–14.

[10] N. Fujieda, S. Watanabe, and K. Kise, "A MIPS System Simulator SimMips for Education and Research of Computer Science," *IPSJ Journal*, vol. 50, no. 11, pp. 2665–2676, 2009.