Paper

An XOR-based Parameterization for Instruction Register Files

Naoki Fujieda[a], Non-member
Shuichi Ichikawa, Non-member

The instruction register file (IRF) shortens and obfuscates instruction sequences by compressing multiple instructions into a packed instruction. The IRF could improve its efficiency by parameterization, while the previously proposed parameterization techniques did not extract the similarity of instructions well. In this paper, we propose an XOR-based parameterization to utilize the limited capacity of the IRF more efficiently. According to our evaluation, with an improved algorithm of instruction selection, our approach makes 20.2% more dynamic instructions IRF-resident than the previous techniques. It also reduces the number of instruction fetches from the cache by 6.3% on average. We also confirmed that the hardware overhead of our parameterization was about a quarter of the previous one. © 2014 Institute of Electrical Engineers of Japan. Published by John Wiley & Sons, Inc.

Keywords: Computer Architecture, Embedded Systems, Instruction Fetch, Instruction Register Files

Received ...

1. Introduction

In embedded systems, the reduction of power consumption has been one of the greatest concerns. In particular, instruction fetch logic in embedded processors is often considered as a target of improvement because it is the most power-hungry part in some processors [1]. More recently, the protection of embedded software from reverse engineering has been another important issue because it usually includes trade secrets. We can encrypt [2, 3] or obfuscate [4, 5, 6, 7] instruction sequences to achieve it.

In terms of power consumption, there are two major approaches to improve instruction fetch. One approach is to increase the information density of fetched instructions, or to compress instruction sequences. The addition of a high-density ISA is a typical solution for RISC processors. For example, some MIPS embedded processors support a 16-bit ISA called MIPS16 [8]. MIPS16 instruction set includes a fixed subset of original MIPS instructions. Shrivastava *et al.* [9] reported that the code size of MIPS32/16 program was reduced by 22% on average with a careful selection of instructions. Similarly, some of recent ARM processors have an additional 16- and 32-bit ISA called Thumb-2 [10]. 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 executed by the traditional instructions and thus the number of instruction fetches is almost unchanged. The other approach is to supply most instructions from a smaller storage than the L1 instruction cache. A filter cache [11] and a loop cache [12] operate as L0 instruction caches, which try to store heavily-reused instructions in different strategies. A software-managed scratchpad memory [13] has a physical address that is separated from the main memory.

An instruction register file (IRF) [14], which we focus on in this paper, has the properties of the both approaches. The IRF is a small memory storing the most common expressions of instructions specified by the compiler. A remarkable difference from the L0 caches and the scratchpad memory is that the storage is accessed by an index or indices in a fetched instruction rather than the program counter. By packing frequently used instructions into a single instruction with these indices, it improves the information density of instructions and reduces the number of fetches from the L1.

We can apply parameterization techniques for the IRF to address a problem of tradeoff about the number of entries. Since the efficiency of instruction packing depends on how many consecutive IRF-referring instructions appear, the IRF should be large enough to cover most of the frequently used instructions. However, if the IRF is too large, it will become meaningless because the energy consumption by referring to the IRF and the width of the indices will be increased. With the parameterization, we can merge multiple instructions with similar expressions into a single group. An instruction that is not the most common in a group is parameterized: it is referenced not only with the corresponding index but also with a parameter.

In addition, the IRF gives resistance against reverse engineering by translating frequently used instructions into IRF indices, if the contents of the IRF (i.e. the original expressions of the instructions) are hidden [7, 15]. In this case, the original form of an IRF-referring instruction might be guessed from other instructions nearby. For example, when a hidden instruction is located between an increment of register 3 and an increment of register 5, it is likely to be an increment of register 4. In order to prevent such analyses, it is similarly important for the IRF to cover more instructions with the parameterization.

A problem of the previously proposed IRF implementation resides in the inefficiency of parameterization. This paper presents an improved parameterization technique called XOR-based parameterization, along with algorithms to select IRF entries with parameters. It is evaluated with traces of practical benchmarks to show that our approach increases the coverage of the IRF and reduces the number of instruction fetches.

We have presented a preliminary version of this work in CANDAR 2013 [16]. The differences from the preliminary work include:

- a new score-based algorithm for instruction selection that covers the shortcomings of the previous *frequency-based* algorithm (Section 4.2),
- an evaluation of hardware overhead with FPGA (Section 5.4),
- a comparative discussion with a simple large IRF without parameterization (Section 6.1),
- a discussion about the feasibility of the IRF on ARM architecture (Section 6.2), and
- a discussion about concerns of the parameterization for the purpose of the protection from reverse engineering (Section 6.3).

The rest of this paper is organized as follows: Section 2 provides an overview and shortcomings of the previous IRF proposal and its parameterization techniques. The hardware organization of our XOR-based parameterization is described in Section 3. In Section 4, we examine how to select groups of instructions as IRF entries. We evaluate the efficiency and the hardware overhead of our methods in Section 5, and then some discussions are made in Section 6. Finally, we conclude the paper in Section 7.

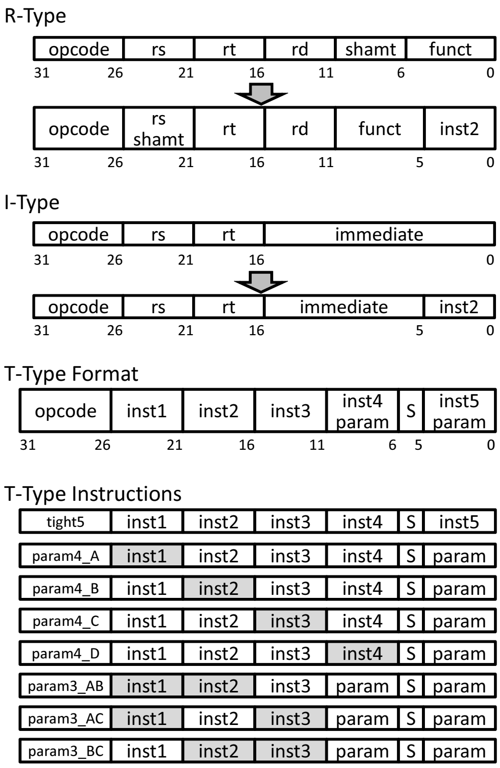

R-Type
opcode
rs
rt
rd
shamt
funct
31 26 21 16 11 6 0
opcode
rs shamt
rt
rd
funct
inst2
31 26 21 16 11 5 0
I-Type
opcode
rs
rt
immediate
31 26 21 16 0
opcode
rs
rt
immediate
inst2
31 26 21 16 5 0
T-Type Format
opcode
inst1
inst2
inst3
inst4 param
S
inst5 param
31 26 21 16 11 6 5 0
T-Type Instructions
tight5 inst1 inst2 inst3 inst4 S inst5
param4_A inst1 inst2 inst3 inst4 S param
param4_B inst1 inst2 inst3 inst4 S param
param4_C inst1 inst2 inst3 inst4 S param
param4_D inst1 inst2 inst3 inst4 S param
param3_AB inst1 inst2 inst3 param S param
param3_AC inst1 inst2 inst3 param S param
param3_BC inst1 inst2 inst3 param S param

Fig. 1. A loosely packed instruction (R-Type and I-Type) includes a regular instruction and an IRF instruction. Tightly packed instructions (T-Type) contain up to 5 IRF instructions. [14]

2. Instruction Register File

2.1. Overview

The instruction register file (IRF) [14] has a set of frequently executed instructions, which are called IRF instructions. It is placed between instruction fetch and instruction decode stages and accessed by an index written in fetched instructions. The target instruction set (MIPS in [14]) is modified so that instructions can include an index or indices. If fetched instructions refer to the IRF, the corresponding IRF instructions are 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. The modified instruction set has two kinds of instructions that include multiple instructions: loosely packed and tightly packed.

The upper part of Figure 1 shows how the loosely packed instructions modify the existing R-Type and I-Type instruction formats of MIPS. 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, described in the lower part of Figure 1, to the traditional MIPS instruction formats. 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, four IRF instruction with one parameter, 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 Figure 1.

In addition to the compression of instruction sequences for which the IRF has originally been proposed, it provides resistance against reverse engineering by hiding the original instruction sequences with indices [7, 15]. This kind of obfuscation of the instruction sequence is called instruction set randomization (ISR) [4, 5, 6]. Comparing with encryption of instruction memory such as the Execute Only Memory (XOM) [2] and the AEGIS architecture [3], most ISR approaches are not much robust in a cryptographic sense. However, they are favorable for embedded systems because they can be implemented with a minimal overhead.

In terms of obfuscation of instructions, one of the differences of the IRF with the other ISR methods is that it does not hide all of the instructions: it only hides IRF instructions. Thus it is also important to cover more instructions with the IRF.

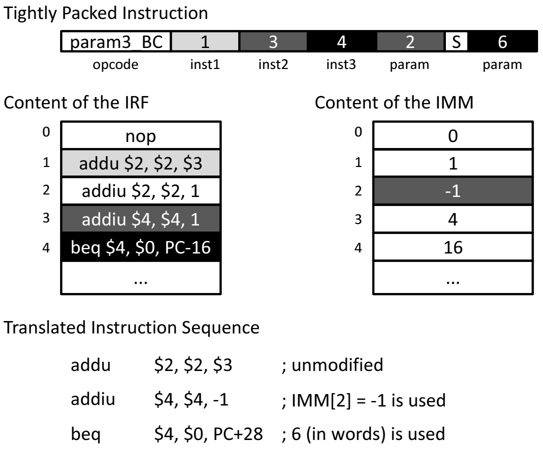
2.2. Parameterized IRF entries

By parameterizing instructions, we can merge multiple instructions with similar expressions into a single entry in the IRF. An IRF that is referenced by a 5-bit index contains 32 kinds of instructions, which is too few to cover most of the frequent instructions. 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 similar instructions: compressing immediate values with an immediate table, merging short-distance branches, and using indirect register specifiers [14]. We introduce the first two techniques; we do not consider the last technique in this paper because it has little effect despite some shortcomings [14].

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.

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. Though the offset will be recalculated in consequence of instruction packing, it is always shortened and 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 bits, the branch is then treated as an IRF instruction regardless of the subsequent recalculation of the distance.



```
addu    $2, $2, $3     ; unmodified
addiu   $4, $4, -1     ; IMM[2] = -1 is used
beq     $4, $0, PC+28  ; 6 (in words) is used
```

Fig. 2. How to extract the actual instructions from a tightly-packed instruction in the original method[14].

Figure 2 illustrates an example of the extraction of the traditional instructions from a parameterized packed instruction. From the *opcode* field and the S bit, the packed instruction is identified as `param3_BC`. It packs three instructions, where the second and the third are parameterized. Since the first instruction does not have a parameter, `addu $2, $2, $3` is extracted from the IRF without modification. The second instruction is a non-branch instruction with a parameter; its immediate value is replaced with `-1`, the corresponding entry of the IMM. The last instruction is a parameterized branch instruction; the sign-extended parameter, `6`, is used as its immediate value. In this example, the packing efficiency is increased by the parameterization because these instructions would have to be written separately without it. Note that the efficiency may also decrease; if the second and the third instructions also resided in the IRF before parameterization and the subsequent instruction was (instructions were) IRF-resident, it could packed four or five instructions into one.

Figure 3 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 are read from the IRF and used

[*] More precisely, the branch offset is relative to the PC of its delay slot (i.e. the PC plus 4) [17].

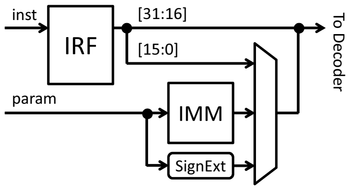


Fig. 3. Extraction of instructions from the IRF in the original method [14].

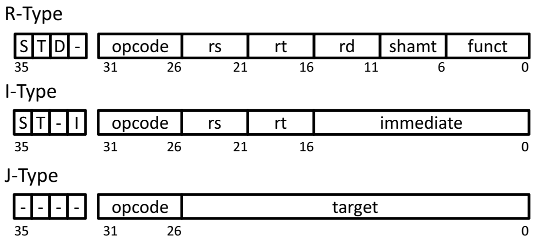


Fig. 4. The format of IRF entry for each type of MIPS instruction [16].

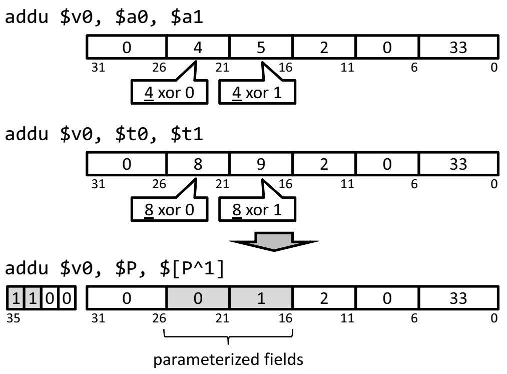


Fig. 5. Example: merging two instructions into a single IRF entry [16].

without modification; the lower 16 bits depend on which format the instruction is, whether the corresponding entry is parameterized, and whether the instruction has a parameter.

2.3. Shortcomings of the existing parameterization

The parameterization techniques in the previously proposed implementation mainly have two problems on efficiency.

One problem is that these techniques can be applied to only I-Type instructions. As a result of our preliminary experiment [16], there is a certain amount of correlation between the number of IRF entries corresponding to I-Type instructions and the increase of IRF instructions executed with the previous techniques (i.e. their availability). Parameterization methods should be able to be applied to both I-Type and R-Type instructions.

Another problem is the imbalance of the immediate table between the packing 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.

3. XOR-based Parameterization

In this section, an XOR-based parameterization is proposed to achieve higher packing efficiency and lower hardware overhead than the previous techniques. The discussion of the shortcomings of the previous implementation can be summarized as that a new technique should be capable of being applied to most instructions and being implemented with simple hardware. Our method is designed so that it can meet both of the requirements.

3.1. Coding of IRF Instructions

Figure 4 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 [17], respectively) and therefore some of them may be included in the IRF instructions. The unused bits are set to zero to keep the corresponding fields unchanged.

Figure 5 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 ˆ 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 where the first parameterized field is zero. For example, the code

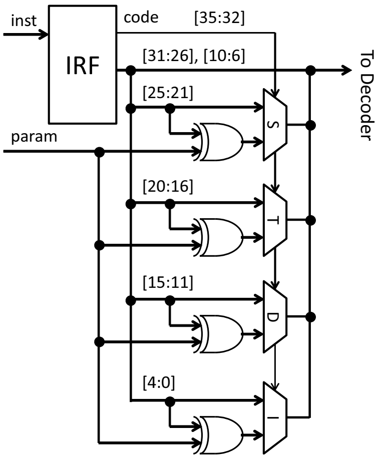


Fig. 6. Extraction of instructions from the IRF in the proposed method [16].

for `addu $v0, $P, $[P^1]` shown in Figure 5 is a normalized form because the first parameterized field, or *rs*, is zero.

3.2. Difference in Hardware Organization

Figure 6 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 Figure 3 and Figure 6. 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 they are implemented in an FPGA, our implementation may enjoy additional benefit from an optimization of putting an XOR and a selector into a single 3-input look-up table. Therefore, it is expected that our method can be implemented with much smaller hardware than the previous implementation. A quantitative evaluation will be shown in Section 5.4.

4. Selection of IRF Instructions

4.1. Frequency-based Algorithm

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 practical time. So we use a heuristic, which is basically similar to a greedy algorithm that is used in the original IRF proposal when indirect register specifiers are applied [14].

Figure 7 outlines the algorithm to prepare a list of codes and the instructions corresponding to each code. An instruction profile ($PROFILE$) may be gathered by static analysis or dynamic profiling. Each instruction in the $PROFILE$ has its occurrence rate. The function coding_candidates (in line 4) gives a set of all normalized codes for the corresponding instruction. After the preparation, $INSTS[code]$ stores all instructions that belong to *code* with their occurrence rates.

```
 1: $PROFILE \leftarrow$ instruction profile
 2: $INSTS \leftarrow$ 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:             $INSTS[code] \leftarrow$ empty array
 7:         end if
 8:         append $inst$ to $INSTS[code]$
 9:     end for
10: end for
```

Fig. 7. Pseudocode of the preparation of code list.

```
 1: $IRF[0] \leftarrow$ code for nop
 2: for $i$ in [1..31] do
 3:     find $code$ that has the highest frequency of $INSTS[code]$
 4:     $flags \leftarrow code[35..32]$
 5:     $def\_inst \leftarrow$ most frequent instruction in $INSTS[code]$
 6:     $IRF[i] \leftarrow$ bit_concat($flags, def\_inst$)
 7:     for $inst$ in $INSTS[code]$ do
 8:         for $candidate$ in coding_candidates($inst$) do
 9:             remove $inst$ from $INSTS[candidate]$
10:         end for
11:     end for
12: end for
```

Fig. 8. Pseudocode of the frequency-based instruction selection [16].

The algorithm of the ‘frequency-based’ selection of IRF entries, which has been proposed in [16], is shown in Figure 8. The function bit_concat (in line 6) returns the bit concatenation of the inputs. The process of generating IRF contents is as follows. Since the entry 0 is reserved for `nop` (line 1), 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 3). Then it sets the most frequent instruction in the code as a default one, which is stored in the IRF (lines 4 through 6). As a result, default instructions, which are referenced without a parameter, come to be extracted by setting *param* in Figure 6 to zero. Lastly the instructions in the selected code are excluded from the corresponding lists (lines 7 through 11). This exclusion may be skipped in the last iteration.

4.2. Score-based Algorithm

The frequency-based algorithm, presented in Section 4.1, aims at maximizing the sum of the occurrence rates of IRF-resident instructions. However, it does not care whether the listed instructions require parameters or not; a frequently used instruction may be accessed with a parameter as a result of being merged with another instruction that appears more frequently.

Figure 9 depicts an example where the frequency-based algorithm performs poorly. Assume that we are going to select

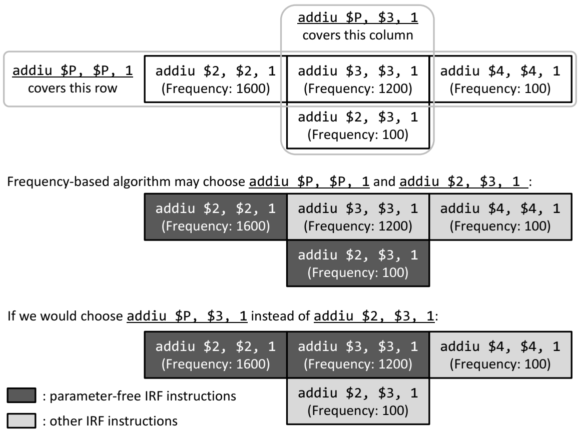


Fig. 9. An example instruction profile with which the frequency-based algorithm will not work efficiently.

```
$IRF[0] \leftarrow$ code for nop
$SELECTED \leftarrow$ empty array
for $i$ in [1..31] do
    find $code$ that has the highest $score(INSTS[code], SELECTED)$
    add $INSTS[code]$ to $SELECTED$
    $flags \leftarrow code[35..32]$
    $def\_inst \leftarrow$ most frequent instruction in $INSTS[code]$

    $IRF[i] \leftarrow$ bit_concat($flags, def\_inst$)
    for $candidate$ in coding_candidates($def\_inst$) do
        remove $inst$ from $INSTS[candidate]$
    end for
end for
```

Fig. 10. Pseudocode of the score-based instruction selection.

two entries from four instructions: `addiu $2, $2, 1`, `addiu $3, $3, 1`, `addiu $4, $4, 1`, and `addiu $2, $3, 1`. Their respective occurrence rates are 1600, 1200, 100, and 100. When no parameterization is applied, two frequent instructions are selected and thus the sum of the occurrence rates is 2800. If we select with the frequency-based algorithm, `addiu $P, $P, 1` is selected as the first entry with a default `P` of 2. Since the instructions corresponding to `addiu $P, $P, 1` are removed from consideration, the remaining `addiu $2, $3, 1` is selected as the second entry. Thus the selection result is shown in the middle of Figure 9. The sum of the occurrence rates increases to 3000. However, there is obviously a better selection for the second entry of `addiu $P, $3, 1` with a default `P` of 3, shown in the bottom of Figure 9. While the sum of the occurrence rates of all IRF instructions does not differ, that of parameter-free IRF instructions significantly increases from 1700 to 2800.

The problem of the frequency-based algorithm resides in that an instruction selected as a parameterized IRF instruction is not reconsidered any more. It is also important that a parameterized IRF instruction becomes parameter-free one, though the benefit of such a promotion is smaller than the advantage to make a non-IRF instruction IRF-resident.

```
$score \leftarrow 0$
for $inst$ in $INSTS[code]$ do
    $freq \leftarrow$ frequency of $inst$
    if $inst$ is the most frequent in $INSTS[code]$ then
        $score \leftarrow score + freq \times 22$
    end if
    if $inst$ is not included in $SELECTED$ then
        $score \leftarrow score + freq \times 5$
    end if
end for
```

Fig. 11. Pseudocode of the calculation of score.

With consideration of the promotion of IRF instructions, we propose a modified, 'score-based' algorithm shown in Figure 10. The differences of the score-based algorithm from the frequency-based one are threefold:

- it keeps a set of instructions that have been already selected as IRF instructions to an array $SELECTED$,
- it selects a *code* based on the score calculated from the occurrence rates and whether the corresponding instructions exist in $SELECTED$, rather than the sum of the occurrence rates, and
- the target of the exclusion is reduced to parameter-free IRF instructions.

Figure 13 shows the algorithm to calculate the score. It is calculated as the sum of the product of the occurrence rate and the following weighting factor:

- 22 points when a non-IRF instruction becomes a parameterized IRF instruction (because a 32-bit expression is reduced to 10 bits),
- 5 points when a parameterized IRF instruction promotes to parameter-free (because a 10-bit expression is reduced to 5 bits),
- 27 points when a non-IRF instruction becomes a parameter-free IRF instruction (the sum of above), or
- 0 points otherwise.

The above mentioned weights are purely based on heuristics: they correspond to the potential reduction of instruction length to be fetched and it is not considered how instructions are likely to be packed. It is left for future studies to find the optimal set of weights.

In the example of Figure 9, the score-based algorithm has four major candidates for the selection of the second entry: choosing each of the instructions separately other than `addiu $2, $2, 1` or choosing `addiu $P, $3, 1`. The score for each selection is as follows:

- `addiu $3, $3, 1`: Since it has been already a parameterized IRF instruction, the score is $1200 \times 5 = 6000$.
- `addiu $4, $4, 1`: The score is $100 \times 5 = 500$, calculated as the same way as above.
- `addiu $2, $3, 1`: It is not an IRF instruction yet, thus the score is $100 \times 27 = 2700$.

- `addiu $P, $3, 1`: It gets $1200 \times 5 = 6000$ points for the promotion of `addiu $3, $3, 1` and $100 \times 22 = 2200$ points for the selection of `addiu $2, $3, 1`. Therefore the total score is $6000 + 2200 = 8200$.

Finally, `addiu $P, $3, 1` is selected as the second entry and the result is correspond to the bottom of Figure 9.

5. Evaluation

In this section, we compare the parameterization methods through evaluations. In respect of the efficiency, two scales 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 the instruction packing and the latter means how efficient the packing actually is. In respect of the hardware overhead, we evaluate the hardware amount and the maximum frequency of circuits when we implement the methods on an FPGA.

5.1. Methodology

We use all of the 36 benchmarks of MiBench [18] 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 [19]. The benchmarks are compiled with gcc 4.7.3, uClibc 0.9.33.2, and binutils 2.21. The instruction set examined is MIPS32 Release 1 with floating point instructions. The compile options are the same as the defaults of MiBench except asserting a flag for static linking (`-static`). Some programs are slightly modified to remove the compile errors due to the difference of compiler versions.

In this section, we define four settings as follows:

- **No Param** stands for the IRF without any parameterization. It is referred to for a baseline of efficiency.
- **Conventional** applies the existing techniques that have been described in Section 2.
- **Frequency** utilizes the XOR-based parameterization along with the frequency-based instruction selection. This is the same as what we have proposed in [16].
- **Score** combines the XOR-based parameterization with the score-based instruction selection.

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 [17]. Second, some I-Type instructions are unable to express their immediate values in 11 bits (See Figure 1). It will cause the addition and modification of instructions and thus decreases the efficiency of instruction packing. However, we do not consider it because it is equally 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 may be included by the IRF instructions. Though it slightly improves the efficiency of merging techniques, we think that the chance of improvement is small because it equally applies to all of them.

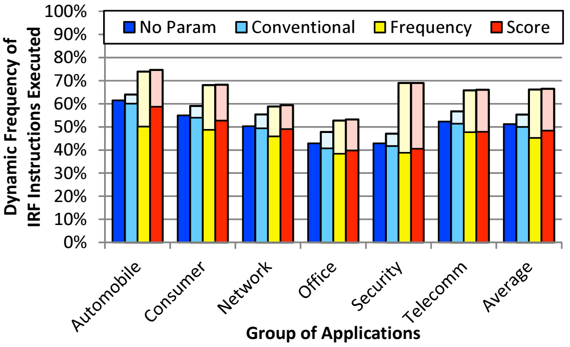


Fig. 12. Improvement in dynamic frequency of IRF instructions. Light bars represent IRF instructions with non-default parameters.

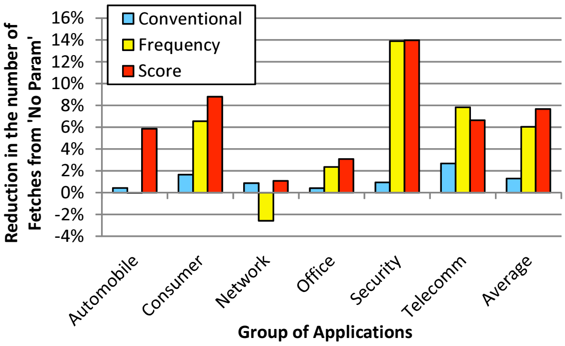


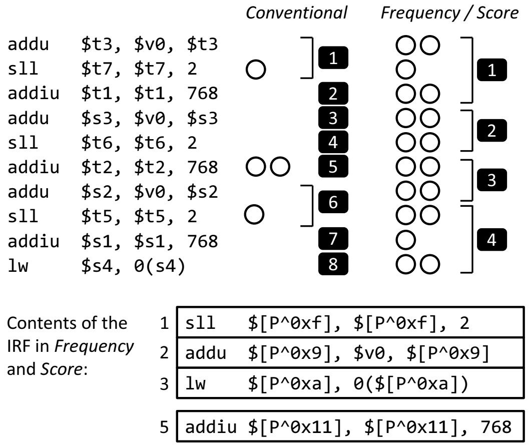
Fig. 13. Reduction in the number of instruction fetches. All values are relative to No Param.

5.2. Dynamic Frequency

Figure 12 shows the dynamic frequency of IRF instructions being executed. The X-axis is the name of a group of MiBench. The Y-axis is the average of the 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. The XOR-based parameterization with the score-based selection (*Score*) improved the average frequency by 20.2% (or 11.2 percentage points) over the techniques in the original IRF proposal (*Conventional*). 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 there are two reasons why the Security benchmarks fit in our method. 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 large 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.

Comparing the selection algorithms, we observed that *Score* improved the rate of parameter-free IRF instructions by 7.0% or



```
addu   $t3, $v0, $t3
sll    $t7, $t7, 2
addiu  $t1, $t1, 768
addu   $s3, $v0, $s3
sll    $t6, $t6, 2
addiu  $t2, $t2, 768
addu   $s2, $v0, $s2
sll    $t5, $t5, 2
addiu  $s1, $s1, 768
lw     $s4, 0(s4)
```

Contents of the IRF in *Frequency* and *Score*:

```
1  sll   $[P^0xf], $[P^0xf], 2
2  addu  $[P^0x9], $v0, $[P^0x9]
3  lw    $[P^0xa], 0($[P^0xa])

5  addiu $[P^0x11], $[P^0x11], 768
```

Fig. 14. The difference in instruction packing in the *rijndael* benchmark.

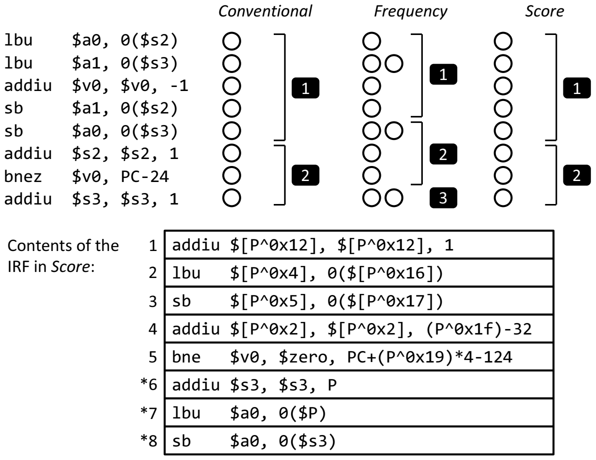
3.2 points over *Frequency*. The rate of all IRF instructions was also improved by 0.5% or 0.3 points over *Frequency*. It is regarded that this benefit comes from keeping parameterized instructions under consideration.

5.3. The number of instruction fetches

Figure 13 shows the reduction in the number of instruction fetches over *No Param*. The X-axis is the same as that of Figure 12. The Y-axis is the average decrease of the number of instruction fetches from the L1 cache. *Frequency* and *Score* reduced the number of fetches by 4.8% and 6.3% over *Conventional*, respectively.

In the Security benchmarks, our method showed remarkable decrease of fetches (12.8% and 12.9% in *Frequency* and *Score*, respectively), just as the increase of the dynamic frequency of IRF instructions being executed. The number of continuous IRF instructions in major loops was drastically increased there. For example, Figure 14 shows the difference in instruction packing of a part of the AES encryption routine in the *rijndael* benchmark, along with the contents of the IRF in both *Frequency* and *Score*. The conventional method did not utilize the similarities of instructions, though some instructions were loosely packed. With our methods, most of the instructions in the routine were retrieved from the IRF and were compressed into tightly packed instructions. In this example, triplets of `sll`, `addu`, and `addiu` were merged into their respective IRF entries and the remaining `lw` was also included in the IRF. As a result, these 10 MIPS instructions were compressed into 4 tightly-packed instructions with our methods, while they were only shortened into 8 instructions in the conventional method.

In Automobile and Network benchmark groups, while *Frequency* negatively affected the number of fetches (increased by 0.4% and 3.4% over *Conventional*, respectively), *Score* successfully reduced it (by 5.4% and 0.2%, respectively). This negative effect with *Frequency* comes from the decrease of the packing efficiency by parameterized instructions. Figure 15 shows the difference among *Conventional*, *Frequency* and *Score* in instruction packing of the most frequently executed loop in the *qsort* benchmark. Also, a part of the contents of the IRF in *Score* is shown in the bottom of Figure 15. 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 *Frequency*, however, some instructions got to require parameters as a result of merging a pair of `addiu`, `lbu`, and `sb`, where the number of required fields became 11. Consequently, *Frequency* decreased the efficiency of packing in this case. In *Score*, the pairs of instructions that had been merged in *Frequency* were separated again by the addition of entries. All the instructions became parameter-free again. As a result, the negative effect of *Frequency* was cancelled and the number of required fields returned to 8. Since the original IRF proposal relies on a greedy algorithm [14], such an unnecessary merge may also occur in *Conventional*; however, in this evaluation, it avoided the problem simply because it failed to find the similarities of instructions.



```
lbu    $a0, 0($s2)
lbu    $a1, 0($s3)
addiu  $v0, $v0, -1
sb     $a1, 0($s2)
sb     $a0, 0($s3)
addiu  $s2, $s2, 1
bnez   $v0, PC-24
addiu  $s3, $s3, 1
```

Contents of the IRF in *Score*:

```
 1  addiu $[P^0x12], $[P^0x12], 1
 2  lbu   $[P^0x4], 0($[P^0x16])
 3  sb    $[P^0x5], 0($[P^0x17])
 4  addiu $[P^0x2], $[P^0x2], (P^0x1f)-32
 5  bne   $v0, $zero, PC+(P^0x19)*4-124
*6  addiu $s3, $s3, P
*7  lbu   $a0, 0($P)
*8  sb    $a0, 0($s3)
```

* Entries 6 through 8 are not included in the IRF in *Frequency*

Fig. 15. The difference in instruction packing in the *qsort* benchmark.

An important difference of the score-based algorithm with the others is that it only succeeded in reducing the fetches in all applications. This result implies that an advantage of the score-based algorithm resides in its conservative selection for the applications such as *qsort* where the benefit of the parameterization is small.

5.4. Hardware Overhead

This section examines the overhead of hardware amount and operating frequency of the parameterization methods by implementing a soft processor with an IRF on a Xilinx FPGA. We chose the latest snapshot of Plasma [20] (as of September 2013) as the target processor. When an instruction is fetched, it is translated with the IRF before it is saved in pipeline registers. To focus on the overhead that comes from the IRF and its parameterization, we do not consider instruction packing. Normal MIPS instructions are transformed to specialized instructions (with

Table I. The results of FPGA implementations of Plasma with the IRF.

	Original	No Param	Conventional	Proposed
Slices	1,981	2,054	2,093	2,064
- Registers	612	587	587	587
- 4-input LUTs	3,765	3,889	3,968	3,912
Block RAMs	5	5	5	5
Maximum Frequency [MHz]	34.988	36.117	35.469	34.381

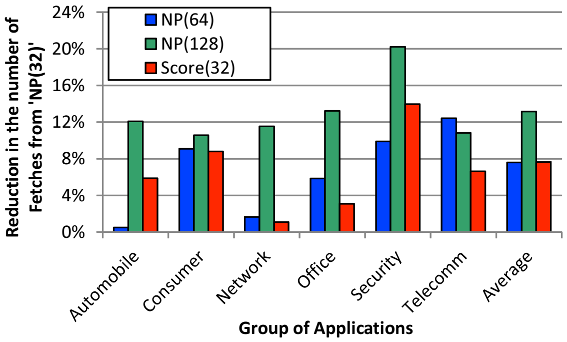


Fig. 16. Comparison of reduction in the number of instruction fetches with large IRFs. All values are relative to NP(32).

a 5-bit index of the IRF and, if needed, a 5-bit parameter) on a one-to-one basis.

We evaluate four implementations: *Original* (the original Plasma), *No Param*, *Conventional*, and *Proposed* (XOR-based parameterization regardless of selection algorithm). All these were synthesized and implemented with Xilinx ISE 14.6. Logic synthesis is optimized for speed. The implementation options are ParHighEffort (ISE 14) preset with ignoring timing constraints (−x). Plasma is configured to have '2-stage' pipeline (the actual pipeline depth is 4), DRAM controller, 4 kiB instruction/data cache, and 4 kiB internal memory. The system operation with a modified processor was verified on the Xilinx Spartan-3E Starter Board.

The evaluation results are summarized in Table I. While the addition of the IRF itself consumed 73 slices (basic units of hardware amount in Xilinx FPGAs), *Conventional* required 39 more slices, though it would be reasonable given that the immediate table is about a half as large as the IRF. On the other hand, *Proposed* required only 10 slices, which was much smaller than *Conventional* as we have expected in Section 3.2. In respect of the maximum frequency, it somehow improved with the addition of the IRF (from *Original* to *No Param*) and it declined by a few percent in *Proposed*. We think that this result was within the margin of error in implementation because there was little difference among the implementations in the estimated frequency reported after logic synthesis.

6. Discussion

6.1. Comparison with a Large IRF

This section examines whether our parameterization scheme has a greater merit than simply enlarging the IRF. A discussion about the number of entries in the IRF has already been made in [14]. The use of a large IRF increases the number of instructions covered by it but decreases the potential packing efficiency (the maximum number of instructions that a tightly packed instruction contains). Thus the number of fetches with a large IRF may be decreased, or may be increased. On the other hand, when we apply parameterization, the coverage will be unchanged or increased and the potential packing efficiency will remain the same. Therefore the number of fetches with parameterization will be decreased or, at worst, unchanged.

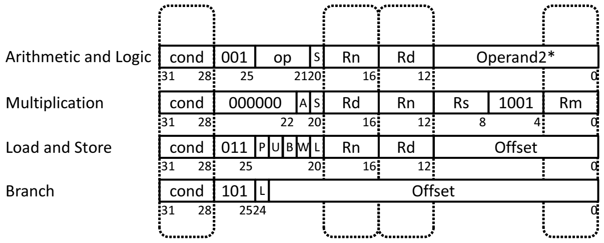
We describe quantitative evaluation results of the fetch reduction with large IRFs in this section. We evaluate the four architectural settings, abbreviating *No Param* to *NP*.

- **NP (32)**: is the same as *No Param* in Section 5.
- **NP (64)**: uses a 64-entry IRF without parameterization. The most frequent 32 entries can be referenced from loosely packed instructions. Since 6 bits are required for an index, a tightly packed instruction contains up to four MIPS instructions.
- **NP (128)**: uses a 64-entry IRF without parameterization. The same restriction as *NP (64)* is applied to loose packing. Since an index needs 7 bits, a tightly packed instruction includes up to four instructions only when the first instruction to be packed is in the most frequent 32 instructions; otherwise, it only packs up to three instructions.
- **Score (32)**: is the same as *Score* or *Proposed* in Section 5.

The restrictions of tight packing in *NP (64)* and *NP (128)* are based on the fact that the *opcode* field of MIPS is 6 bits wide.

Figure 16 shows the results of the fetch reduction. All values are relative to *NP (32)*. By increasing the number of IRF entries, the number of fetches reduced in most cases owning to the increase of the frequency of IRF instructions executed. The only exception was Telecomm from *NP (64)* to *NP (128)*, where the effect of decline in the potential packing efficiency was large. Though the enlargement of the IRF affects the fetches differently from parameterization, on average, *Score (32)* exhibited slightly more reduction in the number of fetches (7.7%) than *NP (64)* (7.6%)).

Our parameterization achieved almost the same efficiency as a 64-entry IRF with smaller hardware overhead (about a quarter of a 32-entry IRF, as we have shown in Section 5.4). At least, our



* an 8-bit immediate value with a 4-bit rotate amount (which will be doubled)

Fig. 17. Formats of primary ARM instructions [23]. Parameterizable fields are enclosed by dotted lines.

proposal is more beneficial than simply increasing the number of entries to 64. When we use the IRF for the reduction of fetch energy, we can use the parameterization along with a large IRF to balance the packing efficiency with the energy per reference, considering the fact that a large IRF consumes more energy. By doing this, we may enjoy another benefit of the parameterization that a parameter is always 5 bits wide regardless of the number of entries.

6.2. Feasibility Study of the IRF on Other ISA

As a method to improve the information density, the IRF can shorten instructions that are *actually* frequently used. The length of machine code for a specific program is different for each ISA. For example, an ARM code is usually shorter than the corresponding MIPS code [21]. In addition, as we have seen in Section 1, MIPS16 [8] and Thumb-2 [10] assign shorter codings to the instructions that are *likely to* be frequently used. However, they cannot consider the *actual* frequency distribution of programs, which the IRF can do.

Important properties for the IRF to be applied to other ISAs are the bias of frequency of each instruction and the similarity of instruction formats. RISC ISAs such as MIPS and ARM may meet both of them, while CISC ISAs like IA-32 do not meet the latter property.

We evaluate the feasibility of the IRF on ARM using binaries compiled for ARM. Benchmarks and evaluation environments are the same as we shown in Section 5.1, except that the target processor is Cortex-A9 (which has ARMv7-A ISA) and that the simulator to obtain traces of the benchmarks is a modified version of QEMU version 1.7.0 [22].

The changes to adapt our parameterization to ARM are as follows. Each parameter has 4 bits. The parameterizable fields are condition (Bits 31–28), register numbers (Bits 19–16 and Bits 15–12), and an additional register number or a part of immediate value (Bits 3–0). Figure 17 depicts some ARM instruction formats [23] and the parameterizable fields. An instruction with a reserved code in the condition field is regarded as a tightly packed instruction. Since it requires 3 bits to determine the type of tightly packing (see Figure 1), the use of the remaining 25 bits is the same as we do in MIPS.

To adapt the conventional parameterization, we select most common thirty-two 12-bit operands or offsets in arithmetic, logic, load, and store instructions for the immediate table. Branch offsets are also parameterized using sign extension.

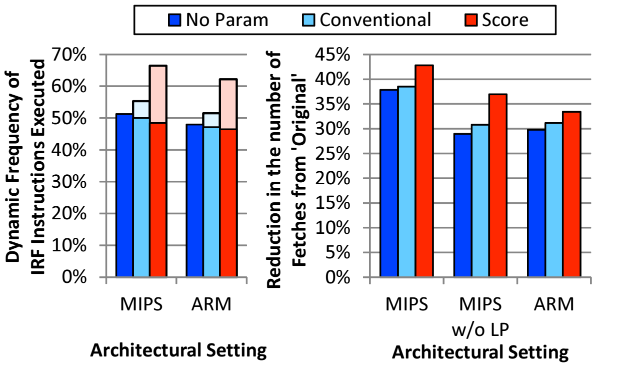


Fig. 18. Comparison of the coverage of the IRF and the fetch reduction from the original ISAs between MIPS and ARM.

Architectural settings that we examine here are as follows.

- **MIPS** applies both loose and tight packing in MIPS.
- **MIPS w/o LP** applies only tight packing in MIPS. It only affects the instruction packing (i.e. the number of instruction fetches). It is evaluated for comparison.
- **ARM** applies only tight packing in ARM. The adaptation of loose packing is left for future work.

Figure 18 shows the dynamic frequency of IRF instructions being executed (in the left graph) and the reduction of the number of instruction fetches over the original ISAs where the IRF is not applied (in the right graph). Each bar corresponds to the setting for the parameterization which we have shown in Section 5.1. Like Figure 12, dark and light bars in the left graph mean parameter-free and parameterized instructions, respectively. In terms of the frequency of IRF instructions executed, about 4 percentage points of the decrease was observed in the all three parameterization settings from MIPS to ARM, while the improvement by the parameterization was almost the same between them. In terms of the fetch reduction, it was declined by about 8 points in *No Param* and *Conventional*, from *MIPS* to *MIPS w/o LP* or *ARM*, because of the absence of loose packing. However, the improvement by our score-based parameterization (*Score*) greatly differed between *MIPS w/o LP* and *ARM*: it was 8.0 points in *MIPS w/o LP* and 3.6 points in *ARM*. We think that the difference in the fetch reduction comes from the difference in the average length of basic blocks. According to static analysis of the benchmarks, *MIPS* has 8.33 instructions per basic block on average, while *ARM* has only 4.80 instructions. Instruction packing may have been restricted by boundaries of basic blocks.

We leave optimization of instruction packing as future work, including the adaptation of loose packing and a branch into a packed instruction. Nevertheless, the results of the evaluation in this section imply sufficient possibility about the portability of the IRF.

6.3. Concerns for the Purpose of Obfuscation

As we have mentioned in Section 2.1, the IRF can also be used for obfuscation of instruction sequences [7, 15]. From the cryptographic point of view, as the obfuscation with the IRF relies on substitution ciphers, we should try to prevent the original instruction sequences from being guessed by frequency analysis [24]. Specifically, the distribution of indices of the IRF should be flat so that as little information as possible can be obtained from the statistical properties of the obfuscated sequences. Therefore not only the coverage of the IRF but also the flatness of the distribution of IRF indices should be considered [7].

Similarly, with the parameterization, the flatness of the distribution of parameters is an important property. In our XOR-based parameterization, two techniques can be applied to flatten the distribution. First, if no parameterization flags are asserted in an IRF entry (i.e. the entry corresponds to a single instruction), an arbitrary parameter can be added to a reference to it. Second, even though parameterization flag is (flags are) asserted in an entry, the set of parameters can be shuffled to some extent by choosing another one of 32 equivalent pseudoinstructions (as we have mentioned in Section 3.1). As a result, little information will be obtained from the flattened distribution of parameters, though there remains a certain correlation between indices and parameters.

As well as what we have done in Section 4, we have proposed an algorithm to find a sub-optimal set of instructions for the IRF-based obfuscation [7]. A challenge in adapting the algorithm to the parameterization is the consideration of the case that the merge of instructions reduces the flatness of the distribution of parameters. When multiple instructions are merged into a single pseudoinstruction, the coverage of the IRF becomes large as other (pseudo)instructions will come to be stored in the IRF. However, arbitrary parameters will not be selected any more after the instructions are merged: the parameters of the instructions will come to be mutually dependent. This may increase the correlation between indices and parameters and thus make frequency analysis easier. We should decide whether or not to merge instructions based on the balance between the coverage of the IRF and the flatness of the distribution of indices and parameters. We leave the design of an actual algorithm as future work.

7. Conclusion

This work proposed an XOR-based parameterization of instructions which efficiently utilizes the limited capacity of the IRF to reduce the power consumption of instruction fetch logic. According to our evaluation results, if we carefully select IRF entries with the score-based algorithm, 20.2% more instructions were fetched from the IRF, the number of instruction fetches was decreased by 6.3% compared to the previously proposed parameterization.

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 further evaluate the efficiency and the cost of our scheme on other ISAs such as ARM. To apply the parameterization to the obfuscation of instruction sequences, new algorithms to construct the contents of the IRF are required.

References

(1) Montanaro J, *et al.*. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits* 1996; **31**(11):1703–1714.

(2) Thekkath DLC, Mitchell M, Lincoln P, Boneh D, Mitchell J, Horowitz M. Architectural support for copy and tamper resistant software. *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*, 2000; 168–177, doi:10.1145/378993.379237.

(3) Suh GE, Clarke D, Gassend B, van Dijk M, Devadas S. AEGIS: architecture for tamper-evident and tamper-resistant processing. *Proceedings of the 17th annual international conference on Supercomputing*, 2003; 160–171, doi:10.1145/782814.782838.

(4) Barrantes EG, Ackley DH, Forrest S, Stefanović D. Randomized instruction set emulation. *ACM Transaction on Information and System Security* 2005; **8**(1):3–40, doi:10.1145/1053283.1053286.

(5) Kc GS, Keromytis AD, Prevelakis V. Countering code-injection attacks with instruction-set randomization. *Proceedings of the 10th ACM conference on Computer and communications security*, 2003; 272–280, doi:10.1145/948109.948146.

(6) Ichikawa S, Sawada T, Hata H. Diversification of processors based on redundancy in instruction set. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 2008; **E91-A**(1):211–220, doi:10.1093/ietfec/e91-a.1.211.

(7) Fujieda N, Ichikawa S. Enhanced Instruction Register Files for Embedded Software Obfuscation. *Proceedings of the 29th International Conference on Computers and Their Applications*, 2014; 153–158.

(8) Kissell KD. MIPS16: High-density MIPS for the Embedded Market. *Technical Report*, Silicon Graphics MIPS Group 1997.

(9) Shrivastava A, Biswas P, Halambi A, Dutt N, Nicolau A. Compilation framework for code size reduction using reduced bit-width ISAs (rISAs). *ACM Transaction of Design Automation Electronic System* 2006; **11**(1):123–146, doi:10.1145/1124713.1124722.

(10) Phelan R. Improving ARM Code Density and Performance. *Technical Report*, ARM Ltd. 2003.

(11) Kin J, Gupta M, Mangione-Smith WH. The filter cache: an energy efficient memory structure. *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997; 184–193.

(12) Lee LH, Moyer B, Arends J. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. *Proceedings of the 1999 international symposium on Low power electronics and design*, 1999; 267–269, doi:10.1145/313817.313944.

(13) Banakar R, Steinke S, Lee BS, Balakrishnan M, Marwedel P. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *Proceedings of the tenth international symposium on Hardware/software codesign*, 2002; 73–78, doi:10.1145/774789.774805.

(14) Hines S, Green J, Tyson G, Whalley D. Improving Program Efficiency by Packing Instructions into Registers. *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005; 260–271, doi:10.1109/ISCA.2005.32.

(15) Chang D, Hines S, West P, Tyson G, Whalley D. Program differentiation. *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*, 9, 2010, doi:10.1145/1739025.1739038.

(16) Fujieda N, Ichikawa S. An XOR-based approach to merging entries for instruction register files. *Proceedings of the 1st International Symposium on Computing and Networking – Across Practical Development and Theoretical Research –*, 2013; 332–337, doi:10.1109/CANDAR.2013.60.

(17) Sweetman D. *See MIPS Run Linux Second Edition*. Morgan Kaufmann, 2006.

(18) Guthaus M, Ringenberg J, Ernst D, Austin T, Mudge T, Brown R. MiBench: A free, commercially representative embedded benchmark suite. *2001 IEEE International Workshop on Workload Characterization*, 2001; 3–14, doi:10.1109/WWC.2001.990739.

(19) Fujieda N, Watanabe S, Kise K. A MIPS System Simulator SimMips for Education and Research of Computer Science. *IPSJ Journal* 2009; **50**(11):2665–2676.

(20) Rhoads S. Plasma – most MIPS I(TM) opcodes, http://opencores.org/project,plasma.

(21) Weaver V, McKee S. Code density concerns for new architectures. *2009 IEEE International Conference on Computer Design*, 2009; 459–464, doi:10.1109/ICCD.2009.5413117.

(22) Bellard F. QEMU, a Fast and Portable Dynamic Translator. *Proceedings of 2005 USENIX Annual Technical Conference*, 2005; 41–46.

(23) ARM Limited. *ARM7TDMI Technical Reference Manual Revision r4p1* 2004.

(24) Bauer FL. *Decrypted Secrets: Methods and Maxims of Cryptology*. 4th edn., Springer, 2006.

Naoki Fujieda (Non-member) received his D.E. degree in 2013 from the Department of Computer Science of Tokyo Institute of Technology. Since 2013, he is an assistant professor of the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. His research interests include processor architecture, applied FPGA systems, embedded systems, and secure processors. He is a member of IPSJ, IEICE, and IEEE.

Shuichi Ichikawa (Non-member) received his D.S. degree in Information Science from the University of Tokyo in 1991. He has been affiliated with Mitsubishi Electric Corporation (1991-1994), Nagoya University (1994-1996), Toyohashi University of Technology (1997-2011), and Numazu College of Technology (2011-2012). Since 2012, he is a professor of the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. His research interests include parallel processing, high-performance computing, custom computing machinery, and information security. He is a member of IEEE, ACM, IEICE, and IPSJ.