Decision Tree Generation for Decoding Irregular Instructions

Katsumi Okuda, Haruhiko Takeyama
Advanced Technology R&D Center, Mitsubishi Electric Corporation, Japan
Email: {Okuda.Katsumi@eb,Takeyama.Haruhiko@cw}.MitsubishiElectric.co.jp

***Abstract*—Instruction set simulators (ISS) are indispensable tools for the development of new architectures and embedded software. One essential part of any ISS is its instruction decoder. Since manual implementation of an instruction decoder for a complex instruction set is tedious and error-prone, automatic generation of an instruction decoder is required. However, as a result of the increasing irregularity of instruction encoding because of the incremental addition of instructions, generating efficient instruction decoders is complicated. In this paper, we propose a generation algorithm of a decision tree for decoding irregular instructions. Our algorithm can generate decision trees by using not only significant bits of opcode patterns but also exclusion conditions in decoding entries. Our results on ARMv7, Thumb-2, MIPS64, RH850, and TriCore show that our algorithm generates efficient instruction decoders in terms of both depth and memory consumption regardless of whether the target instruction set is irregular or not.**

I. Introduction

An instruction set simulator (ISS) is a tool that mimics the target architecture on a host machine and is used to develop new architectures and embedded software. One essential part of any ISS is its instruction decoder. Since manually writing an instruction decoder is a tedious and error-prone task, the automated generation of efficient instruction decoders is necessary. However, the increasing complexity of instruction encoding complicates generating efficient ones. Because of the requirement for upward binary compatibility while extending an instruction set, more opcodes and operands must be densely encoded within the restricted size of the instruction words. As a result, the instruction set becomes irregular. In an irregular instruction set, some instructions are encoded using specific values or value exclusion of bit fields; otherwise the fields are dedicated to parameters. For example, most ARM [1] instructions use the upper four bits of the instruction word as a condition parameter, and when their values all equal one, the other instructions are encoded irregularly.

Generally, ISS instruction decoders use a decision tree to decode the instruction words. Conventional decoder generators take the encoding entries of an instruction set as input and generate a decision tree. The generator splits the decoding entries and recursively computes a decision tree until a leaf node is reached at which the decoding entry can be unambiguously classified. Decision tree generation algorithms are classified into two groups by how they split decoding entries. The first group uses the significant bits of the opcode patterns to split the decoding entries. The second uses a cost model. The former algorithms can generate efficient trees in terms of depth and memory consumption, but it cannot generate them for irregular instruction sets. On the other hand, the latter can generate decision trees for irregular instruction sets, but the generated trees are not efficient in terms of memory consumption. In this paper, we propose a generation algorithm that creates efficient decision trees in terms of both depth and memory consumption for irregular instruction sets. Our algorithm splits the decoding entries using not only the significant bits of opcode patterns but also the exclusion conditions of the instructions. Our results on several instruction sets show that the generated trees are shallower than the trees generated by the current state-of-the-art algorithms. Moreover, the memory efficiency of the generated trees is comparable to binary trees regardless of whether the target instruction set is irregular or not.

The remainder of this paper is organized as follows. Section II discusses related work. Preliminaries are described in Section III, and Section IV introduces our decoder generation algorithm. The experimental results are shown in Section V. Finally, our conclusion is in Section VI.

II. Related Work

Processor description languages nML [2], ISDL [3], LISA [4], EXPRESSION [5], ASIP Meister [6], and HARMLESS [7] can generate ISS. However, these literatures don't refer to the details of decoder generation. We believe that our generation algorithm will be incorporated into generation frameworks with these or similar languages.

The generation of instruction decoders has also been described [8]–[13]. These algorithms can be applied to generate decoders for ISS, disassemblers, and debuggers, etc. In the remainder of this section, we emphasize the difference between our approach and these approaches.

Many simple binary decoder generation algorithms have been described [8], [11], [14]. The generated decoders sequentially match the input bit string with all possible instruction opcode patterns. The complexity of the decoder is in $O(n)$, which is acceptable for dissassemblers or debuggers, but not for ISS that decodes instructions at runtime. On the other hand, our proposed algorithm generates a decision tree with which a decoder can efficiently decode instructions.

Other algorithms generate decision trees for instruction decoding [9], [10], [12], [13]. Both [9], on which our research is based, and [12] use significant bits of opcode patterns to split decoding entries and can generate efficient decision trees in terms of depth and memory consumption. However, the algorithm cannot generate decision trees for irregular instruction sets. Our work extends a previous algorithm [9] and makes it possible to generate decision trees for irregular instruction sets. Our algorithm generates the same decision tree as the one generated by [9] when the target instruction set is not irregular. Our algorithm can also take irregular instruction sets and generate decision trees that are as efficient as the generated trees of irregular instruction sets.

Other algorithms, [10] and [13], use cost models to split the decoding entries into subsets. Since identical entries are

978-3-9815370-6-2/DATE16/

TABLE I
EXAMPLE OF INSTRUCTION SET

Instruction	7	6	5	4	3	2	1	0	Pattern
A	0	0	a		b		c		00------
B	0	1	a		b		c		01------
C	1	0	a		b		0	0	10----00
D	1	0	a		b		0	1	10----01

prone to exist in the multiple subsets, the resulting tree have multiple leaves that are labeled with the same instruction. As a result, the generated trees consume too much memory. In contrast, our algorithm prevents duplicate entries as much as possible. Another difference between the algorithms that use cost models and our algorithm is whether to use the instruction's occurrences. Cost models include the occurrence information. In contrast, our algorithm does not use the occurrence information. Since the occurrences are not available when we generate decision trees for new instruction sets, the algorithms that use cost models is not suitable for generating trees for new architectures.

[13] introduced the PART algorithm which can handle irregular instruction sets. PART preprocesses the decoding entries and converts the entries with irregular encoding to decoding entries without irregular encoding. Section V-A in [13] shows that 442 entries were converted to 4,362 entries for the ARM instruction set. As a result, the generated tree is deeper than the tree generated by our algorithm.

III. PRELIMINARIES

In this section, we formalize the input and the output of decision tree generation. First we introduce the decoding entries that are input to the generator and extend them for irregular instruction sets. Then we introduce a decision tree, which is the generator's output.

A. Decoding Entries

A bit pattern is defined as $p = \{0, 1, -\}^n$, where $-$ stands for a don't-care value and n is the length of the bit pattern. Bit string $s = B^n \quad (B = \{0, 1\})$ matches pattern p iff $\forall i \in \{0, 1, ..., n-1\} : \quad s[i] = p[i] \quad \vee \quad p[i] = -$. In this paper, we write $s \in p$ if s matches p. For example, bit string 0010 matches pattern $001-$, and we describe it as $0010 \in 001-$.

Each instruction encoded in the standard way is represented by a decoding entry in the form of pair (p_o, l), where bit pattern p_o is a opcode pattern and l is a label for this decoding entry. D is defined as a set of entire decoding entries. For example, the members of decoding entry set D for the instruction set shown in Table I are ($00------$, A), ($01------$, B), ($10----00$, C) and ($10----01$, D). The instruction decoder's task is to map bit string s to decoding entry (p_o, l) so that $s \in p_o$.

B. Decoding Entries for Irregular Instruction-set

To add new instructions to an existing instruction set or to encode many instructions in short instruction words, processor designers reuse opcode patterns. More specifically, when parameter field f of instruction I does not take specific bit string s, new instructions $J_1, J_2, ...$, whose field f has constant bit string s, are added using the same opcode pattern as for instruction I. For example, an irregular instruction set that has extended instructions based on the instruction set in Table I is presented in Table II. Table II's exclusion condition column

TABLE II
EXAMPLE OF IRREGULAR INSTRUCTION SET

Instruction	7	6	5	4	3	2	1	0	Pattern	Exclusion Condition
A	0	0	a		b		c		00------	a=00 $\wedge$ c$\neq$00 $\wedge$ c$\neq$11 a=11
B	0	1	a		b		c		01------	a=11
C	1	0	a		b		0	0	10----00	
D	1	0	a		b		0	1	10----01	
E	0	0	0	0	b		0	1	0000--01	
F	0	0	0	0	b		1	0	0000--10	
G	0	f	1	1	b		c		0-11----	

means that if one of the exclusion conditions is satisfied by an instruction word, that instruction word is not an instance of the entry's instruction. The instruction set in Table II includes instructions E, F, and G. Instructions E and F were added using specific bit strings 01 and 10 (Bit 0-1) that field-c of instruction A does not take when field-a equals 00. Similarly, instruction G is added using bit string 11 (Bit 4-5) that field-a of instructions A and B does not take.

An instruction set is irregular when Equation 1 is satisfied:

$$\exists (p_d, l_d) \in D,\ (p_e, l_e) \in D\ (l_d \neq l_e)\ \ match(p_d, p_e) \quad (1)$$

Equation 1 means that there is more than one decoding entry whose opcode pattern is matched by bit string $b \in B$. For example, in the instruction set presented in Table II, bit string 01110000 matches the opcode patterns of instructions B and G.

We extend a decoding entry to triplet (p_o, l, C_e) so that each entry consists not only of pattern p_o and label l but also exclusion condition set C_e. If bit string s satisfies one of the exclusion conditions, even if the bit string matches the opcode pattern, the bit string does not match the decoding entry. An element of the exclusion condition set is pair (p_m, P_u), where p_m is a matching pattern and P_u is a set of unmatching patterns. As an example, the exclusion condition set of instruction A in Table II is $\{(--00----, \{(------00), (------01)\}), (--11----, \emptyset)\}$. In this exclusion condition set, matching condition a=00 is represented by matching pattern $--00----$, and other conditions are represented in similar ways. Note that one or more matching conditions is represented by a single matching pattern. For example, matching conditions a=11 $\wedge$ c=00 would be represented by matching pattern $--11--00$.

The satisfaction of the exclusion conditions of a decoding entry is checked by:

$$\bigvee_{(p_m, P_u) \in C_e} (s \in p_m \wedge \bigwedge_{p_u \in P_u} s \notin p_u) \quad (2)$$

For example, bit string 00000001 matches both the A and E patterns, but it satisfies one of the exclusion conditions of instruction A: $00000001 \in --00---- \ \wedge \ 00000001 \notin ------00 \ \wedge \ 00000001 \notin ------11$; bit string 00000001 is not an instance of instruction A but of instruction E.

C. Decision Trees

Instruction decoders are implemented as search procedures that find matching entries. The search procedure can be represented by a decision tree. The problem of decoder construction is equivalent to the construction of a decision tree.

We define decision tree (V, E), where V is the set of nodes and E is the set of edges. V is defined as $V = N \cup L$, where N

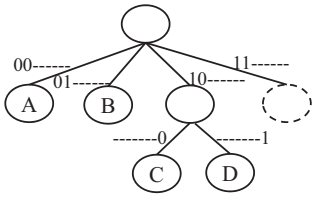

Fig. 1. Example of a decision tree

is the inner node and L is the leaf. An inner node is labeled with decision function $f_v : B^n \rightarrow V$, and a leaf is labeled with decoding entry $d \in D$.

The instruction decoder takes a bit string as input and applies the decision function of the root to get the next node to visit. When the next node to visit is a leaf, the decoder outputs the leaf's label, which is a decoding entry. When the node to visit is not a leaf, the decoder repeatedly applies the decision function until it reaches a leaf.

As an example of a decision tree, the decision tree for the instruction set in Table I is shown in Figure 1. Decoding entries A, B, C, and D are represented as labels of the leaves. The decision function's value of each node is determined by the labels of the edges connected to its children. If a bit string matches one of the labels of the edges, the decision function's value becomes the node to which the edge is connected. For example, when a decoder takes bit string 10000000 as input, it applies the root's decision function to get the next node to visit. Because $10000000 \in 10------$, the next node to visit becomes the label of the third child node from the left. Subsequently, the decoder applies the decision function of the child node. Because $10000000 \in -------0$, the next node to visit becomes the leaf labeled with C and the decoded instruction is reached.

Since the decision functions of each node can be implemented as lookup tables, the complexity of the decision functions is $O(1)$. As a result, the decoder's efficiency depends on the decision tree's depth, which should be as shallow as possible to require the smallest number of tests.

IV. Methodology

In this section, we show our generation algorithm for irregular instruction sets and how to deal with irregular encodings. First, to introduce a generation problem for irregular instruction sets, we show the base algorithm and extend it to solve the problem.

A. Base Algorithm

The base algorithm, which closely resembles a previously introduced algorithm [9], consists of mutually recursive procedures *MakeTree* and *MakeNode*, as shown in Algorithm 1.

Procedure $MakeTree$ takes decoding entry set D described in Section III-A as input and outputs a constructed decision tree. If input set D contains only one entry, procedure *MakeTree* creates a leaf labeled with the entry and returns it. When D contains more than one entry, procedure *MakeTree* makes an inner node by a mutually recursive procedure *MakeNode*.

Procedure $MakeNode$, which is defined in lines 8–18, is composed of two separate parts; the first creates the patterns to split the decoding entries and the second recursively builds subtrees.

Algorithm 1 Base algorithm

```
1:  procedure MakeTree(D)
2:      if |D| = 1 then
3:          return CreateLeaf(D)
4:      else
5:          (result, node) ← MakeNode(D)
6:          if result ≠ FAILED then
7:              return node
8:  procedure MakeNode(D)
9:      P ← MakePatterns(D)
10:     if P ≠ ∅ then
11:         return (FAILED, nil)
12:     node ← CreateNode()
13:     for all p ∈ P do
14:         D_m ← MakeMatchingEntries(D, p)
15:         if D_m ≠ ∅ then
16:             child ← MakeTree(D_m)
17:             node.AddChild(p, child)
18:     return (OK, node)
```

Procedure *MakePatterns* in line 9 makes pattern set P, which is defined as follows:

$$P = \{p \mid p \notin \{-\}^n, p[i] \in \begin{cases} \{0,1\} & significant_D(i) \\ \{-\} & (otherwise) \end{cases}\} \quad (3)$$

where the Boolean value of $significant_D(i)$ is given by:

$$\begin{gathered} \forall (p_o, l) \in D : p_o[i] \neq - \\ \wedge \\ (\exists (p_o, l) \in D : \ p_o[i] = 0) \ \wedge \ (\exists (p_o, l) \in D : \ p_o[i] = 1) \end{gathered} \quad (4)$$

If function $significant_D(i)$ is $true$, the i-th bit can be used to split the decoding entry set. Set P has all the possible bit patterns of the significant bits. For example, when $D = \{(0100, X), (01{-}1, Y), (11{-}0, Z)\}$ is given, the value of function $significant_D$ is $true$ when $i = 0$ or $i = 3$. Therefore set P consists of all possible bit patterns $0{-}{-}0$, $0{-}{-}1$, $1{-}{-}0$ and $1{-}{-}1$. Note that when no bit satisfies Equation 4, set P becomes empty. In such cases, procedure *MakeNode* returns failed status (lines 10–11).

After the creation of bit patterns P, procedure *MakeNode* creates a node to which it adds children by recursively applying procedure $MakeTree$. $MakeMatchingEntries$ in line 14 takes the decoding entry set and the bit pattern as input and makes subset $D_m = \{(p_o, l) \in D \mid match(p_o, p)\}$, where function $match$ is defined as follows:

$$\begin{gathered} match(p_x, p_y) = \\ \forall i \in \{0, 1, ..., n-1\} : \\ p_x[i] = p_y[i] \ \vee \ p_x[i] = - \ \vee \ p_y[i] = - \end{gathered} \quad (5)$$

The tree generated by Algorithm 1 from the instruction set in Table I is presented in Figure 1.

When Algorithm 1 is applied to an irregular instruction set, there is a case where no significant bit is found. As a result, the tree generation algorithm cannot continue correctly. For example, consider tree generation for the instruction set in Table II. The tree generation's first application to all of the decoding entries splits the set into subsets {A, B, E, F, G} and {C, D}. When applying tree generation to subset {A, B, E, F, G}, no significant bit is found. Since pattern set P becomes empty, the algorithm cannot continue.

Algorithm 2 Tree generation algorithm

```
 1: procedure MAKETREE'(D)
 2:     if |D| = 1 then
 3:         return CreateLeaf(D)
 4:     else
 5:         (result, node) ← MakeNode'(D)
 6:         if result ≠ FAILED then
 7:             return node
 8:         else
 9:             return MakeConditionNode(D)
10: procedure MAKECONDITIONNODE(D)
11:     p_m ← SelectPattern(D)
12:     D_m ← MakeMatchingEntries'(D, p_m)
13:     D_o ← MakeOtherEntries(D, p_m)
14:     t_m ← MakeTree'(D_m)
15:     t_o ← MakeTree'(D_o)
16:     return CreateConditionNode(p_m, t_m, t_o)
```

B. Extended Algorithm

We extend Algorithm 1 to split the entries into D_m and D_o using exclusion conditions so that the significant bits are available in D_m and D_o again. We also introduce a condition node labeled with a decision function that selects the next node depending on whether the condition is satisfied or not.

The extended generation algorithm is presented in Algorithm 2. Procedure $MakeTree'$ takes decoding entry set D described in Section III-B as input and outputs the decision tree. The main difference between Algorithm 1 and Algorithm 2 is that Algorithm 2 has an extended part in lines 8–9 to handle the cases where Algorithm 1 fails to make a node.

Procedure $MakeNode'$ is the same as procedure $MakeNode$ in Algorithm 1, except that it uses procedure $MakeMatchingEntries'$ instead of procedure $MakeMatchingEntries$. Procedure $MakeMatchingEntries'$ selects entries depending not only on the opcode patterns but also the exclusion conditions. The details of $MakeMatchingEntries'$ are described below in Section IV-D.

The pseudocode in lines 10–16 defines procedure $MakeConditionNode$ that makes a condition node. In line 11, procedure $MakeCondtionNode$ selects one matching pattern in the exclusion condition set by procedure $SelectPattern$, which will be described below. Procedure $MakeMatchingEntries'$ in line 12 makes a set of entries that matches selected pattern P_m. Procedure $MakeOtherEntries$ makes a set of entries that matches the patterns other than the selected pattern. The pseudocode in lines 14–16 recursively applies procedure $MakeTree'$ to create subtrees and returns a node whose children are the subtrees.

A decision tree generated by Algorithm 2 is shown in Figure 2. Its shaded nodes are condition nodes. A decision function's value of a condition node is determined by whether an input bit string matches the left edge's label which is a matching pattern selected in line 11. If the bit string matches the label, the next node to visit is the left child; otherwise the next node to visit is the child on the right. The decision functions can be implemented as lookup tables indexed by the Boolean result of the pattern matching. Hence the complexity of the decision function is $O(1)$.

The rest of this section describes the details of procedures $SelectPattern$, $MakeMatchingEntries'$, and $MakeOtherEntries$.

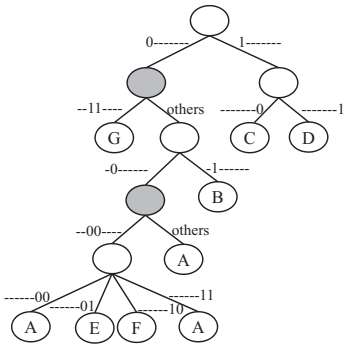

Fig. 2. Decision tree for irregular instruction set

C. Pattern Selection (SelectPattern)

Algorithm 2 uses a matching pattern in the exclusion conditions to split the decoding entries when no significant bit is found. The absence of significant bits is due to instructions $J_1, J_2, ...$, which are added using matching pattern p_m in the exclusion conditions of base instructions $I_1, I_2, ...$ The aim of using matching pattern p_m in the exclusion conditions is to separate base instructions $I_1, I_2, ...$ from instructions $J_1, J_2, ...$ so that the significant bit of the opcode patterns is found again.

Since there is more than one matching pattern in the decoding entries, the problem is the proper selection of matching pattern p_m. If an invalid matching pattern is selected, no significant bit is found. Moreover, the invalid matching pattern cannot split the decoding entries effectively. In such cases, identical decoding entries exist in the matching entry set D_m and the other entry set D_o. As a result, $|D_m| + |D_o|$ becomes larger than when a matching pattern is properly selected. In other words, the properly selected matching pattern splits the decoding entries into D_m and D_o that minimize $|D_m| + |D_o|$. Procedure $SelectPattern$ selects the pattern that can minimize $|D_m| + |D_o|$, after once applying procedures $MakeMatchingEntries'$ and $MakeOtherEntries$ to all the possible matching patterns.

D. Creation of Matching Entries (MakeMatchingEntries')

Procedure $MakeMatchingEntries'$ takes decoding entry set D and pattern p as input and outputs a newly created set that contains the entries that match the input pattern. Note that not every entry in the newly created set is the same entry in the input entries because the exclusion conditions were modified. The exclusion conditions that are invalidated by the input pattern are removed from the exclusion condition sets of output entries. In addition, when the matching pattern in an exclusion condition are invalidated, the unmatching patterns in the condition are expanded to their opcode pattern.

Procedure $MakeMatchingEntries'$ makes the output entries in the following four steps.

Step 1: Procedure $MakeMatchingEntries'$ selects all the entries $(p_o, l, C_e) \in D$ that satisfy $match(p_o, p)$ in Equation 5 and that do not satisfy the exclusion conditions shown below:

$$\exists (p_m, P_u) \in C_e : contain(p, p_m) \ \wedge \bigwedge_{p_u \in P_u} \neg match(p, p_u) \tag{6}$$

where function $contain(p_x, p_y)$ is defined as follows:

$$contain(p_x, p_y) = \\ \forall i \in \{0, 1, ..., n-1\} : \ p_x[i] = p_y[i] \ \vee \ p_y[i] = - \quad (7)$$

The Boolean value of function $contain$ is $true$ when bit pattern p_x contains all fixed bits of pattern p_y as part of it. Hence, Equation 6 is satisfied when the input pattern contains the matching pattern and does not match all of the unmatching patterns.

Step 2: Procedure $MakeMatchingEntries'$ updates each exclusion condition set C_e in the output entries selected in *Step 1* with C'_e as follows:

$$C'_e \leftarrow \{(p_m, P_u) \in C_e | \ match(p, p_m) \bigwedge_{p_u \in P_u} \neg contain(p, p_u)\} \quad (8)$$

In addition, procedure $MakeMatchingEntries'$ updates unmatching pattern P_u of $(p_m, P_u) \in C'_e$ with P'_u as follows:

$$P'_u \leftarrow \{p_u \in P_u | match(p, p_u)\} \quad (9)$$

Step 3: All of the bits in the patterns of the exclusion condition fixed by input pattern p become useless and must be invalidated. Procedure $MakeMatchingEntries'$ updates each bit $p_c[i]$ of all the matching and unmatching patterns in the output decoding entries with $p'_c[i]$ as follows:

$$p'_c[i] \leftarrow \begin{cases} p_c[i] & (p[i] = -) \\ - & (otherwise) \end{cases} \quad (10)$$

Step 4: As a result of *Step 3*, there is a case where all the bits of the matching pattern become $-$: when the matching condition has been satisfied and the unmatching conditions have not been satisfied yet. In this step, procedure $MakeMatchingEntries'$ selects the matching conditions given by $\{(p_m, P_u) \in C'_e \mid p_m = \{-\}^n\}$ for all the decoding entries, and expands each unmatching pattern $p_u \in P_u$ to their opcode pattern. Each bit in the opcode pattern p_o in the expanded entry has value $p'_o[i]$ as follows:

$$p'_o[i] \leftarrow \begin{cases} p_o[i] & (p_u[i] = -) \\ p_u[i] & (otherwise) \end{cases} \quad (11)$$

Note that a new decoding entry is created and added to the output set for each expansion. The new decoding entry inherits the exclusion conditions except for the one that includes the unmatching pattern expanded to the opcode pattern.

E. Creation of Other Entries (MakeOtherEntries)

Procedure $MakeOtherEntries$ takes decoding entry set D and pattern p as input and outputs a decoding entry set that matches the patterns other than input pattern p. Output entries are made in the following two steps.

Step 1: Procedure $MakeOtherEntries$ selects entries D_o given by:

$$D_o = \{(p_o, l, C_e) \in D | \neg contain(p_o, p)\} \quad (12)$$

Step 2: Procedure $MakeOtherEntries$ updates each exclusion condition set C_e of $(p_o, l, C_e) \in D_o$ with C'_e as follows:

$$C'_e \leftarrow \{(p_m, P_u) \in C_e | \neg contain(p_m, p)\} \quad (13)$$

This is equivalent to removing the exclusion condition that is determined to be unsatisfied. In addition, procedure $MakeOtherEntries$ updates each unmatching pattern set P_u of $(p_m, P_u) \in C'_e$ in the decoding entries with P'_u as follows:

$$P'_u \leftarrow \{p_u \in P_u | \neg contain(p_u, p)\} \quad (14)$$

V. EXPERIMENTAL RESULTS

To evaluate the decision trees generated by our algorithm, we implemented the tree generation algorithm and applied it for several instruction sets: ARMv7, MIPS64, Thumb-2, RH850, and TriCore. ARMv7, Thumb-2, and RH850 are irregular instruction sets, and MIPS64 and TriCore are not irregular instruction sets. We selected ARM and MIPS because the results of the state-of-the-art PART [13] and EFF [13] for these instruction set are available. We used the exclusion conditions specified in vendor's user manuals for the decoding entries. We also added $cond \neq 1111$ to the exclusion conditions of all of the ARMv7's decoding entries whose instructions have a field named $cond$.

We programmed the generator in Python and ran it on a Core-i7 2.8 GHz, Linux-based workstation with 16-GB main memory. The time for tree generation was negligible for all test inputs. The worst case is for the Thumb-2 instructions, which took 0.9 seconds.

All of the generated decision trees were implemented as nested switch case statements written in C. A switch case statement compiled with jump table optimization is one form of lookup table. The decoders that consists of the nested switch case statements were tested until C1 code coverage was achieved to check the correctness of the generated trees.

The following experimental results show that our algorithm generated shallow and small footprint trees.

A. Tree Depth

The generation results with our work and the quoted results of EFF and PART are presented in Table III. The result of EFF_OCC [13] is not quoted, because it was trained for specific benchmarks. The EFF and PART results are available in ARMv7 and MIPS 4k. MIPS64, for which our algorithm was applied, contains all of the MIPS 4k instructions.

The depth is the number of edges from a root to leaves and an index of the decoding efficiency. The average, minimum, and maximum depths of the trees generated by our algorithm are equal or shallower than the trees generated by EFF for ARM and MIPS . This result reflects that our algorithm uses non-adjacent bits to split the decoding entries, but EFF uses only adjacent bits to split them.

The average, minimum, and maximum depths of the trees generated by our algorithm are also equal or shallower than the trees generated by PART. PART preprocesses the input decoding entries and converts them to satisfying paths. That increases the size of the input to the generator. As a result, the generated tree becomes deeper than the tree generated by our algorithm that does not need the preprocessing.

The tree's depth depends on its instruction set. The average depth of the tree for MIPS64, which is the shallowest, is 2.21. In contrast, the tree for Thumb-2 is 6.88, which is the deepest among the trees generated by our algorithm. The depths of the trees generated by EFF and PART have different tendencies. EFF produced the shallower tree than PART for MIPS. On the other hand, PART produced the shallower tree than EFF for ARM. Our algorithm generated the shallower trees than both PART and EFF for both MIPS and ARM. Hence, our algorithm is suitable for a wide range of instruction sets.

TABLE III
GENERATION RESULTS

Algorithm	Instruction-set	Instructions (#)	Average Depth	Min. Depth	Max. Depth
Proposed Algorithm	ARM	442	6.15	2	10
EFF [13]	ARM	442	7.89	2	18
PART [13]	ARM	442	7.07	2	14
Proposed Algorithm	MIPS64	265	2.21	1	4
EFF [13]	MIPS 4k	163	3.66	1	8
PART [13]	MIPS 4k	163	7.84	1	26
Proposed Algorithm	Thumb-2	504	6.88	2	12
Proposed Algorithm	RH850	237	5.41	1	8
Proposed Algorithm	TriCore	827	2.39	1	3

B. Memory Efficiency

Memory efficiency is critical for decision trees because the larger a lookup table is, the higher the cache miss ratio is. Cache misses slow down instruction decoding.

We define memory efficiency m as follows:

$$m = |D| \ / \ \sum_{n \in N} s(n) \tag{15}$$

where D is a set of decoding entries, N is a set of inner nodes, and $s(n)$ is the number of slots in the lookup table that implements the decision function of node n. The lookup table of a node includes not only child slots but also empty slots. The value of $s(n)$ is always 2 if n is a condition node. The memory efficiencies of the generated decision trees are shown in Figure 3. The memory efficiencies, which range from 0.36 (Tricore) to 0.57 (MIPS64), are comparable to the memory efficiency of a binary tree regardless of whether the encoding is irregular or not. Note that efficiency of a binary tree is given by $\frac{|D|}{2(|D|-1)}$.

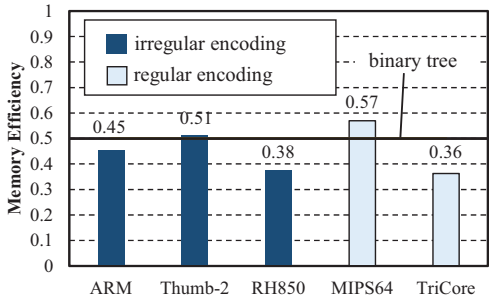

Fig. 3. Memory efficiencies of decision trees

C. Impact of Pattern Selection

To evaluate the effectiveness of the pattern selection method described in Section IV-C, we compared the proposed method and a random selection method that randomly selects a matching pattern from the exclusion conditions. We generated decision trees for ARMv7 by Algorithm 2 with our proposed and random selection methods. Because the results of the random selection method are different every time, we generated trees ten times with it. Table IV shows the experiment results. Our proposed method generated the best tree in terms of tree depth and memory efficiency. The worst average value of tree depth was 9.42 for the tree generated by random #8, which is 53% deeper than the tree generated by the proposed method. The worst value of memory efficiency was 0.08 from the tree generated with random #4 that consumes roughly six times as much memory as the tree generated with the proposed method.

Our experiment shows that the selection of patterns for the creation of condition nodes significantly affected the quality of the generated trees and the selection method described in Section IV-C can generate high quality trees.

TABLE IV
COMPARISON OF PATTERN SELECTIONS (ARMV7)

Pattern Selection	Average Depth	Min. Depth	Max. Depth	Slots (#)	Memory Efficiency
min $\lvert D_m\rvert + \lvert D_o\rvert$	6.15	2	10	968	0.46
random #0	8.48	2	21	4,518	0.10
random #1	8.12	2	17	3,794	0.12
random #2	9.07	2	19	5,152	0.09
random #3	7.82	2	15	2,612	0.17
random #4	8.63	2	20	5,634	0.08
random #5	7.89	2	14	2,326	0.19
random #6	8.66	2	19	5,006	0.09
random #7	8.98	2	17	4,200	0.11
random #8	9.42	2	19	5,322	0.08
random #9	8.64	2	17	2,962	0.15

VI. CONCLUSION

This paper addresses the problem of efficient tree generation for irregular instruction sets. Our proposed algorithm uses not only significant bits but also exclusion conditions to split the decoding entries. By carefully choosing patterns from exclusion conditions, it can generate shallow and small footprint trees for irregular instruction sets without using the instruction's occurrences. The depth of the generated trees is shallower than the existing state-of-the-art algorithms. The memory efficiency of the generated trees is also comparable to binary trees regardless of whether the target instruction set is irregular or not.

REFERENCES

[1] J. Goodacre and A. N. Sloss, "Parallelism and the arm instruction set architecture," *Computer*, vol. 38, no. 7, pp. 42–50, Jul. 2005.

[2] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar, "Generation of software tools from processor descriptions for hardware/software codesign," in DAC '97, pp. 303–306.

[3] G. Hadjiyiannis, S. Hanono, and S. Devadas, "Isdl: An instruction set description language for retargetability," in DAC '97, pp. 299–302.

[4] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "Lisa machine description language for cycle-accurate models of programmable dsp architectures," in DAC '99, pp. 933–938.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "Expression: a language for architecture exploration through compiler/simulator retargetability," in DATE '99, pp. 485–490.

[6] K. Okuda, S. Kobayashi, Y. Takeuchi, , and M. Imai, "A simulator generator based on configurable vliw model considering synthesizable hw description and sw tools generation," *Proc. Workshop on Synthesis and System Integration of Mixed information Technologies, Apr. 2003*, pp. 152–159, 2003.

[7] R. Kassem, M. Briday, J.-L. BéChennec, G. Savaton, and Y. Trinquet, "Harmless, a hardware architecture description language dedicated to real-time embedded system simulation," *J. Syst. Archit.*, vol. 58, no. 8, pp. 318–337, Sep. 2012.

[8] T. Jeremiassen, "Sleipnir-an instruction-level simulator generator," in ICCD '00, pp. 23–31.

[9] H. Theiling, "Generating decision trees for decoding binaries," in LCTES '01, pp. 112–120.

[10] W. Qin and S. Malik, "Automated synthesis of efficient binary decoders for retargetable software toolkits," in DAC '03, pp. 764–769.

[11] M. Reshadi, N. Dutt, and P. Mishra, "A retargetable framework for instruction-set architecture simulation," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 431–452, May 2006.

[12] T. Ratsiambahotra, H. Cassé, and P. Sainrat, "A versatile generator of instruction set simulators and disassemblers," in SPECTS'09, pp. 65–72.

[13] N. Fournel, L. Michel, and F. Pétrot, "Automated generation of efficient instruction decoders for instruction set simulators," in ICCAD '13, pp. 739–746.

[14] G. Hadjiyiannis, P. Russo, and S. Devadas, "A methodology for accurate performance evaluation in architecture exploration," in DAC '99, pp. 927–932.