# FUNCTIONAL ANALYSIS AND PERFORMANCE EVALUATION OF DECODER DECISION TREE GENERATION ALGORITHMS

Lillian Tadros
Robotics Research Institute
Technical University of Dortmund
Email: lillian.tadros@tu-dortmund.de

## KEYWORDS

Irregular instruction set architectures; Instruction decoders; Decision trees; Hardware modeling

## ABSTRACT

Instruction decoders are indispensable components of processor toolchains. The strenuous manual implementation of decoders can be greatly alleviated by decoder generation tools. These need to handle the rising complexity of modern instruction sets, notably irregularities such as non-uniform opcodes and logic propositions on bit fields. Furthermore, tools need to provide cost-optimized decoders, the efficiency of which can have a substantial effect on the overall performance. This paper analyzes five published algorithms for decoder generators from two perspectives: First, in terms of functionality, we systematically assess how each tool handles different properties of modern instruction sets, highlighting properties that are challenging, unhandled by current algorithms or even result in functionally erroneous decoders. Second, we challenge seemingly intuitive definitions of decoder optimality using a sophisticated model of decision tree cost. We experimentally validate this analytical model for generated decoders of the SPARC, MIPS32 and ARMv7 instruction sets. For our analysis, we implemented all five algorithms after correcting conceptual errors and extending the functionality to handle the above-mentioned ISAs. Our work reveals that state-of-the-art decoder generation tools are unable to fully and correctly handle complex ISAs and adopt an erroneous notion of optimality.

## INTRODUCTION

Instruction decoders are common components in SoC designs, where they span a wide range of application options: As part of ASIPs or COTS processor models, inside instruction set simulators (ISS) or binary tools such as assemblers or debuggers, for high-level system simulations or cycle-accurate verification.

Designing decoders, however, is arguably one of the most time-consuming, complex and error-prone modeling tasks and one that is still largely manual. There are few tools that generate decoders given a high-level description of the instruction set. Typically, those generate a decision tree, where each node contains a decision function over a set of instruction bits. The instruction is thus fully classified upon reaching the corresponding leaf.

As the trend is towards larger and more complex instruction sets, decoder generators face new challenges. To accommodate new instructions with minimal impact and maximum backward-compatibility, instruction set designers often resort to non-linearities such as multiple opcode fields, encoding sub-variants or logic propositions on groups of bits. The few tools that attempt automatic generation of instruction decoders are not fully equipped to address such irregularities.

Beside the functional requirements, the efficiency of the decoder model, i.e. the speed at which it can classify a given instruction, is crucial to the performance of the final processor model and of the simulation platform at large, and is still as relevant as ever even on modern hardware. This becomes apparent when one considers the extensive body of research that has been dedicated to devising static, dynamic or partial code-translation, relocation and caching techniques (e.g. [1, 2]), all of which try to avoid interpretive simulation altogether. While such techniques do improve speed, they are all inherently inflexible for simulation purposes, completely unfit for debugging, and therefore ultimately unable to replace interpretive decoding. Thus, it is incumbent that decoder generation tools should seek efficient solutions. Unfortunately, the available tools either make incorrect assumptions regarding cost modeling, or do not consider efficiency altogether.

This paper is organized as follows: We refrain from including an extra section on related work since it is the purpose of this paper to assess existing tools for automatic decoder generation, making the respective tools rather the main content of our work. We are not aware of any related work that surveys different decoder generation algorithms. We therefore proceed directly with the required definitions in the following section, which include the properties and challenges of modern instruction sets. Our main contribution then follows, consisting of a thorough analysis of five algorithms for generating decoder decision trees, first in terms of functionality, then in terms of cost. In terms of functionality, we identify certain properties of instruction sets that are especially challenging to generator tools. We show that some irregularities either cannot be handled at all or even lead to erroneous decoders. In terms of performance, we first discuss several popular definitions of optimality that prove misleading. We then carry out a theoretical cost analysis based on a sophisticated model for decision tree cost. We verify these analytical results by implementing, correcting and expanding the five algorithms and experimentally benchmarking the performance of the generated decoders for the SPARC [3], MIPS32 [4] and ARMv7 [5] platforms. We finally conclude with some open questions.

Communications of the ECMS, Volume 37, Issue 1,
Proceedings, ©ECMS Enrico Vicario, Romeo Bandinelli,
Virginia Fani, Michele Mastroianni (Editors) 2023
ISBN: 978-3-937436-80-7/978-3-937436-79-1 (CD) ISSN 2522-2414

TABLE I: Example of an Irregular Instruction Set

| | Encoding | Condition | Feature |
|---|---|---|---|
| A | `11-- 1---` | | Non-uniform opcodes (A,B,C,D) |
| B | `0-1- ---1` | | Non-identification bit $b0$ |
| C | `-00- 0---` | | |
| D | `10-- 1---` | $\neg(b2 \wedge b1 \wedge b0)$ | Boolean proposition |
| AA | `11-- 111-` | | Specialization |
| AB | `11-- 10-1` | | Multiple specialization |
| AAA | `11-- 1111` | | Nested specialization |

The instruction encodings are defined using a notation where the bits are listed in MSB order, set bits are written as '1', unset bits as '0' and don't-care bits as '-'. The formal mathematical notation for e.g. instruction $A$ in row 1 would be: $b7 \wedge b6 \wedge b3$.

## BACKGROUND AND DEFINITIONS

### Preliminary Definitions

We begin by defining an *instruction instance* or bit string $s$ as an n-length string over the binary alphabet. Formally, $s \in \{0,1\}^n$.

An *instruction encoding* or bit pattern $p$ is an n-length string over the ternary-logic alphabet, i.e. $p \in \{0,1,-\}^n$ where "–" designates an undefined or "don't-care" bit. Note that shorter encodings can be padded with don't-care bits to achieve a uniform encoding size.

The instruction set architecture defines the set of *encoding entries* as $E = \{(l,p,o)\}^m$, where $m$ is the total number of instruction encodings. Each tuple thus relates a pattern $p$ to a unique *instruction label* $l \in L$ and *occurrence* $o \in [0,1]$, which denotes the probability that a given bit string $s$ will match $p$.

Within a decoding decision tree, each leaf is tagged with an instruction label $l$, while every internal node contains a *decision function.* A decision tree implements the function $d : s \to E$: At every internal node, the decision function is applied to a bit string $s$ until the correct encoding is determined upon reaching a leaf. The classification to a given leaf is called *matching*: Formally, $s$ matches an encoding entry $e$ along with its corresponding pattern $p$ iff $\forall i, 0 \leq i \leq n-1$, either $s[i] = p[i]$ or $p[i] = -$. We denote a successful match by writing $s \in p$. Note that all strings that do not match any pattern can be trivially matched to an invalid label. The definition of $d$ as a function assumes that the set $E$ is *well-formed*, i.e. each string matches exactly one entry. We define well-formedness as a string distance greater than zero between every pair of encodings, where an undefined bit in one of the encodings contributes zero to the distance metric, whether Euclidean, Hamming or otherwise [6].

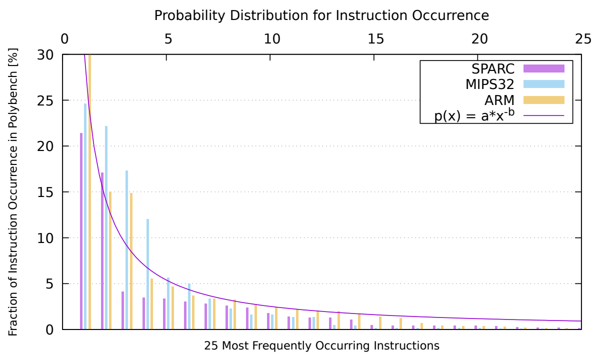
### Properties of Instruction Sets

We first identify some of the challenging features of modern instruction sets. To this purpose, we analyzed three ISAs: SPARC was chosen for historical reasons, as we already had a working decoder definition to build upon. MIPS32 was chosen as an exponent of classic RISC, while ARMv7 was chosen due to its market dominance in the embedded world and as a hybrid instruction set that has sparked a heated discussion over whether it is, in fact, a CISC-disguised-as-RISC. With the instructions numbering $\approx$ 350 and several instructions that are anything but modular (LDM$x$, PUSH/POP, etc), the question is, at any rate, not trivial. Interestingly, despite the RISC label, all three ISAs have non-trivial encoding complexity and exhibit almost all of the below-mentioned irregularities: The simplest, SPARC, got away without the most challenging attribute (propositions) and with uniform opcodes, while ARMv7 boasts all the below-mentioned features. Excepting SPARC, all ISA descriptions were written by the author.

***Non-identification bits.*** A reliable decoder needs to test bits not only for differentiating between two instructions, but also for ensuring the validity of a single instruction, even after it is fully classified. In the example instruction set in table I, bit $b0$ in instruction $B$ is not relevant for classification but still needs to be tested for validation.

***Specializations.*** Instruction specializations, also termed *sub-instructions*, share the same encoding with a parent instruction, but define further bits that the parent leaves undefined. Note that specializations are not allowed to either redefine or undefine any bit set by the parent – the first option would be considered a different instruction altogether and the second an ambiguity in the instruction set. Instructions $AA$ and $AB$ are both specializations of $A$, in that case an example of *multiple specialization.* Instruction $AAA$ is a *nested specialization*, as it is a descendant of $AA$. Another way to view specializations is as an exclusion proposition over the parent instruction (see Section "Propositions"). We choose not to adopt this view, however, because the instruction set architecture will typically not explicitly mention the sub-instruction property nor any exclusion proposition in the definition of the parent instruction, requiring the decoder to automatically deduce the relationship.

***Non-uniform opcodes.*** An opcode denotes the smallest set of bits that are sufficient to uniquely identify an instruction. Traditionally, opcodes comprised the same contiguous set of bits in all instructions, which implied that a simple masking of the opcode sufficed for unique identification. In modern instruction sets this is far from being the case. The constant addition of special opcodes, sub-opcodes and so forth have resulted in a plethora of instruction formats where there is hardly any bit that does not contribute to the opcode of some instruction. This can have two effects: First, the number of bits required to differentiate between instructions can, in the extreme case, encompass all $n$ instruction bits, instead of the lower limit of $\lceil log_2(m) \rceil$ that are strictly required to classify $m$ instructions. For example, differentiating instructions $A, B, C, D$ requires comparing bits $b3, b5, b6, b7$. This effect obviously enlarges the search space for decision functions. The second and more important effect is that there is no guarantee that there would always exist some bit that is defined (i.e. not set to don't-care) for all instructions in a given group. Formally, this means that the intersection of the defined bits in a group of instructions can be empty. This means that a decision function able to



The number of occurrences of each encoding in the PolyBench suite is summed over all individual benchmarks and displayed as a percentage for the first 25 most frequently occurring encodings. The curve is a regression of the SPARC histogram to a power-law function. Reprinted from [8].

divide a given set of instructions by testing some combination of bits is not guaranteed to exist; there might always be at least one instruction that returns ‘don’t-care’ upon applying the decision function. If this situation is not handled at all, the decoder generation algorithm may fail on perfectly valid instruction sets, such as instructions $A, B, C, D$. One solution is *duplication*: In this case, if the chosen decision function evaluates don’t-care bits on some instruction, the instruction is added to all matching branches.

***Propositions.*** Propositions are restrictions on bit fields that are more complex than the simple conjunction of bits that we assumed so far. Propositions are often defined as a negation and should thus lead to rejecting some strings that would otherwise match an encoding. As with simple bit definitions, propositions can either be required for identification (if another instruction defines the rejected encoding) or for validation (where the encoding is not caught by another instruction yet should be rejected nonetheless). The proposition on instruction $D$ pertains to the latter case.

***Non-uniform probabilities.*** The distribution of instruction occurrence inside a given program, i.e. the fraction of instruction instances that match a certain encoding, is in general highly imbalanced. Figure 1 shows the probability of occurrence for some instructions inside the PolyBench benchmark suite [7] compiled for our three platforms. For all three instruction sets, well over half of the encodings are not present in PolyBench at all, while at most only 7% of the encodings account for approximately 95% of the instruction strings. While the details of the distribution and quality of the estimate are obviously dependent on, and limited by, the selected benchmark, the general form seems to comply well to a power-law (Pareto) distribution.

***Non-uniform instruction sizes.*** Variable instruction lengths, as e.g. partially the case with the *Thumb* extension to ARM ISAs, are only superficially relevant to decoder generators. As mentioned above, during the generation process, shorter encodings can be padded to the maximum instruction size. During decoding, any bits pertaining to the next instruction can easily be discarded after identifying the current one.

## Expanded Instruction Representation

The definitions provided so far sufficiently account for all the instruction set properties presented in Section except for propositions. To incorporate such propositions into our framework, we first need to choose an appropriate logic language. For the purpose of *representation*, the language should primarily satisfy succinctness. For *manipulating* the propositions, other language features, primarily satisfiability, need to be considered. We assume that the formulation of the propositions in the ISA, since written for readability, is already in the most succinct form – or if not the most, then at least not unnecessarily unwieldy. Table II shows examples of propositions from the ARMv7 and the MIPS32 instruction sets, where we took over the notation of the ISA with only slight syntactical adaptation. All propositions encountered in the three examined ISAs comply to first-order logic without quantification and with predicates in the binary domain. If binary numbers are converted to decimal, all propositions, with the exception of row 3, would also comply to the stricter set of Satisfiability Modulo Theories (SMT) in combination with integers. Furthermore, if predicates over bit fields are regarded as atoms, and negation pushed inside parenthesized expressions, the notation would reduce to conjunctive normal form (CNF), expressed by the following BNF:

```
term        ::=  clause ( '∧' clause )*
clause      ::=  predicate
              |  '(' predicate ( '∨' predicate )+ ')'
predicate   ::=  negliteral
              |  '(' literal op literal ')'
negliteral  ::=  [ '¬' ] literal
op          ::=  '=' | '≠' | '<' | '≤' | '≥' | '>'
literal     ::=  'b1' | 'b2' | ...
```

Row 3 applies an ISA-defined function *Ones*, which returns the number of bits that are set to one, and which would require expanding the SMT-domain. As this is the case only for the ARMv7 PUSH/POP instructions, we decided to convert the proposition to the following SMT-compliant form, which can then be expressed in decimals:

$$\neg(R = 0_2) \wedge \neg(R = 1_2) \wedge \neg(R = 10_2) \cdots \wedge \neg(R = 1000\ 0000\ 0000\ 0000_2)$$

## FUNCTIONAL ANALYSIS OF DECODER DECISION TREE ALGORITHMS

Since we are concerned with automatically generated decoders, we will not consider manual decoders [9, 10, 18] that are traditionally implemented as nested conditionals. Nor will we deal with decoder generators that require human intervention or data-mining in pre-grouping the instructions or providing extra semantic information [11–13, 17]. Such pre-processing is cumbersome, error-prone and distorts optimization efforts. Likewise, solutions that are automated but produce decoders that linearly traverse instructions are also considered inapplicable for instruction sets in the range of several hundred [1, 14–16]. We will therefore concentrate on the five published solutions which output

TABLE II: Examples of Propositions in the ARMv7 and MIPS32 ISAs

| ISA | Instruction | Encoding | Proposition |
|---|---|---|---|
| ARMv7 | ADD (sh-reg) | `CCCC 0000 100- ---- ---- ---- 0--1 ----` | $\neg(C = 1111_2)$ |
| | ADD (imm) | `CCCC 0010 100S NNNN DDDD ---- ---- ----` | $\neg(C = 1111_2) \wedge \neg(N = 1101_2) \wedge \neg(N = 1111_2 \wedge S = 0_2) \wedge \neg(D = 1111_2 \wedge S = 1_2)$ |
| | PUSH (block) | `CCCC 1001 0010 1101 RRRR RRRR RRRR RRRR` | $\neg(C = 1111_2) \wedge (Ones(R) > 1)$ |
| MIPS32 | BEQZALC | `001000 SSSSS TTTTT ---- ---- ---- ----` | $\neg(T = 00000_2) \wedge (S < T)$ |
| | BLTZC | `010111 SSSSS TTTTT ---- ---- ---- ----` | $\neg(S = 00000_2) \wedge \neg(T = 00000_2) \wedge (S = T)$ |
| | BLTC | `010111 SSSSS TTTTT ---- ---- ---- ----` | $\neg(S = 00000_2) \wedge \neg(T = 00000_2) \wedge (S <> T)$ |

The instruction encodings define named sets of bits using single capital letters that are repeated to indicate the size of the bit set. (The repetition thus does *not* mean that the bits are of equal value.) The formal mathematical notation for e.g. ADD (shifted register) in row 1 would be: $\neg(b31 \wedge b30 \wedge b29 \wedge b28) \wedge \neg b27 \wedge \neg b26 \wedge \neg b25 \wedge \neg b24 \wedge b23 \wedge \neg b22 \wedge \neg b21 \wedge \neg b7 \wedge b4$.

TABLE III: Implemented Features in Selected Decoder Generation Tools

| Feature | Decoder | | | | |
|---|---|---|---|---|---|
| | Theiling | Qin | Fournel | | Okuda |
| | | | PART | EFF | |
| Non-identification bits | Yes | Maybe[1] | Yes | Maybe[1] | Maybe[1] |
| Specializations | Yes | Maybe[1] | Yes[2] | Yes[2] | Maybe[1] |
| Non-uniform opcodes | No | Yes | Unclear[3] | Yes | No[4] |
| Propositions | No | No | Partially[5] | Partially[6] | Partially[7] |
| Optimization | No | Partially[8] | No | Partially[8] | No |

[1]Description missing or unclear.
[2]Sub-instructions are treated as exclusion propositions over the parent.
[3]No algorithm for handling non-orthogonality described, but experimental results indicate some workaround may have been applied.
[4]Non-orthogonality erroneously handled by trying to split on propositions.
[5]BDD expansion results to $2^{16} - 17$ instructions for the ARMv7 PUSH/POP instructions.
[6]Definition of optimality leads algorithm to fail in finding propositions of size larger than two bits.
[7]Descriptive formalism not sufficiently expressive. No combining of propositions. Non-identification propositions unhandled.
[8]Optimization based on heuristics for memory consumption, decision function cost and tree height. Significant restriction of set of possible decision functions.

decision trees after autonomously processing an ISA description.

## Functional Evaluation of Greedy Algorithms

***Theiling [19].*** The first attempt at generating automatic decoders uses a greedy algorithm where the decision function is a mask over all bits that are defined for every instruction in the current subset. This means there is no support for non-uniform opcodes, causing the algorithm to fail if no comprehensively defined bit is found. Non-identification bits are tested for correctness. Specializations are handled by tagging an internal node as a fall-back for the common, non-specialized case. Propositions are not handled and no attempt at optimization is done. Table III gives an overview of implemented features. The algorithm set the stage for the study of fully automatic decoder generation and was later picked up by the four below-mentioned algorithms as well as others (e.g. [20]). In our implementation, we could apply Theiling's algorithm to the SPARC, but not to the MIPS32 or ARMv7 instruction sets, as the latter two contained a multitude of non-uniform opcodes and propositions.

As to optimization, the fact that undefined bits are per definition ignored rules out likely more efficient solutions that involve duplication. Furthermore, no attention is given to the frequency of instruction occurrence, running the risk of banishing more common instructions to the bottom of the tree. Lastly, the algorithm takes no heed of the size of the mask: In instruction sets where a large number of bits is defined, the resulting tree can metamorphose into a giant table with many memory-consuming branches. The algorithm clearly favors shallow trees with many branches, equates performance with tree depth and yet is not even guaranteed to generate the shallowest trees.

***Fournel/PART [21].*** The authors present two algorithms, the first of which, named PART, augments Theiling's solution to handle logic propositions. In a preliminary step, propositions are written as binary decision diagrams (BDD), solved, and the solutions inserted into the instruction set in place of the original instructions. Theiling's algorithm is then applied to the expanded, proposition-free instruction set. The authors recognize the notorious inefficiency of solving BDDs. In the case of the problematic PUSH/POP block instructions in ARMv7, the BDD expansion of the representation discussed above that uses 17 exclusion propositions leads to a whopping $2^{16} - 17$ expanded instructions, which our test platform could not handle. To circumvent this, we resorted to hardcoding 120 separate instructions thus:

$$
\begin{aligned}
POP_1 &= \cdots \wedge (R = \texttt{---- ---- ---- --11})\\
POP_2 &= \cdots \wedge (R = \texttt{---- ---- ---- -101})\\
&\cdots\\
POP_{15} &= \cdots \wedge (R = \texttt{1000 0000 0000 0001})\\
POP_{16} &= \cdots \wedge (R = \texttt{---- ---- ---- -110})\\
&\cdots\\
POP_{120} &= \cdots \wedge (R = \texttt{1100 0000 0000 0000})
\end{aligned}
$$

As for other features, sub-instructions are not handled as done by Theiling, but rather by defining a condition over the parent that excludes the child encoding. This is problematic for several reasons: First, sub-instructions would typically be added later to an ISA, meaning that, in most cases, the parent encoding would not be changed to explicitly exclude the child. This means that this relationship needs to be first deduced manually, which entails tremendous effort. Second, this quickly gets unhandy with multiple or nested encodings, which would have to be added to the encoding of *all* parents. Regarding non-uniform opcodes, the authors explicitly mention that they are handled by duplication, though it is unclear whether this applies only to their second (EFF) or to both algorithms. If it applies to PART as well, then it is ambiguous: It is unclear whether duplication is only restricted to non-uniform cases, and, more importantly, how a decision function is chosen if no comprehensively defined bit is

TABLE IV: Irregular Instruction Set Challenging Okuda's Algorithm

| | Encoding | Condition | Feature |
|---|---|---|---|
| A1 | `11-- --00` | | No proposition |
| A2 | `0-1- --00` | | |
| A3 | `-00- --00` | | |
| B1 | `11-- --01` | $\neg(b4 \wedge b3 \wedge b2)$ | Proposition not relevant for identification |
| B2 | `0-1- --01` | | |
| B3 | `-00- --01` | | |
| C1 | `00-- --10` | | Identification only by combination of proposition and set bits |
| C2 | `1--- --10` | $\neg(b6 \wedge b5)$ | |
| C3 | `-1-- --10` | $\neg(b7 \wedge \overline{b5})$ | |
| D1 | `---- --11` | $\neg(\overline{b7} \wedge \overline{b6}) \wedge \neg(b7 \wedge b6)$ | Identification only by comb. of propositions |
| D2 | `---- --11` | $\neg(\overline{b7} \wedge b6) \wedge \neg(b7 \wedge \overline{b6})$ | |

found. The authors seem to have implemented some workaround, since, in our implementation of PART, disregarding non-orthogonality leads the decoder generation to fail for both ARMv7 and MIPS32. We therefore decided to implement PART as follows: Duplication is used only when no other decision function is found. The decision function then consists of a single bit, which is the bit defined for the largest number of instructions. If the bit happens to be set to the same value in all instructions, the second best bit is chosen and so forth. It is clear that this approach makes no attempt at optimizing or balancing the tree, but, since the authors left this point vague, we chose the most straightforward implementation.

***Okuda [22].*** This work also specifically adapts Theiling's algorithm to instruction sets that include propositions. The authors state that the generated decoders are identical to Theiling's if the instruction set does not contain propositions, leading us to assume that non-identification bits and specializations are handled accordingly. Unlike Fournel, however, propositions are neither converted to an alternative representation nor expanded, but rather treated as first-class citizens, i.e. as possible atomic criteria for decision functions. The formalism chosen for representation is similar to ours – an SMT in CNF over bit field predicates – but allows only equalities between a bit field and a constant. While the approach is promising, there are several flaws: First, the formalism obviously lacks sufficient expressiveness. It is unclear how the authors used it to represent inequalities between a bit field and a constant or relations between two bit fields, as in rows 4-6 in table II. The function *Ones* in row 3 is likewise not expressible using this restricted formalism.

Second, the algorithm resorts to classification using propositions *iff* Theiling's original algorithm fails. This presupposes the following: 1) That a proposition will always be defined in case no common masking bit is found. 2) That this proposition, if such indeed exists, can be used for classification, as opposed to a non-identification proposition. 3) That a proposition will be successful in splitting the set on its own, not necessarily in combination with other propositions in the same instruction group. 4) That propositions need only be used if the original algorithm fails, thus generally omitting to check non-identification propositions. Table IV illustrates a well-defined instruction set where none of these assumptions hold. In the first iteration, the instruction set will be split into three groups containing the instructions *Ai*, *Bi*, and *Ci*, respectively. All further splitting will then fail on all three groups. Group A is valid but non-orthogonal. The algorithm can handle this case only by resorting to propositions. Since none are defined, the algorithm will fail. Group B proceeds similarly and then attempts to split using the defined proposition. Since the proposition is irrelevant for identification, the algorithm will likewise fail. Groups C and D are not a case of non-orthogonality but would, in fact, have been ambiguous without the identification-relevant propositions. However, the identification can only be accomplished by *combining* either multiple propositions or propositions and set bits – a case that is unhandled by the algorithm, which will fail trying to use each proposition separately. Lastly, assumption 4 means the algorithm will refrain from checking the non-identifying proposition of Group B.

In order to successfully run their algorithm on our ISAs, we applied the following modifications: First, we resorted to the SMT representation discussed in Section  to be able to define such propositions as would have otherwise not been possible (e.g. the MIPS32 BEQC, BNEC, BOVC and BVNC instructions).

Second, we expanded the decision function population step as follows: If the single propositions foreseen by the algorithm fail in splitting the set, we successively try combinations of bits and propositions until the encoding set can eventually be split, possibly by testing all propositions of a given instruction in one go. Our expansion involving multiple propositions was required for 27 out of 267 decision nodes in the ARMv7 instruction set, without which the algorithm would have failed (see table V for statistics on generated trees).

Third, we replaced the complex matching steps described in their section IV.D, which are very specific to the restricted representation chosen, with the following generic matching steps: When a single proposition is chosen to split the set, the unmatching branch is modified by removing the proposition from the containing clause (false predicate inside a clause). On the matching side, the whole clause is removed (true predicate inside a clause). This concurs with the quintessence of their algorithm and is applicable to our more generic representation.

## Functional Evaluation of Optimizing Algorithms

***Qin [23].*** Qin et al. pursue an altogether different approach: Instead of using all defined bits as a mask, they attempt to find the decision function that results in an optimized search tree. From the functional viewpoint, the search space for optimal decision functions is reduced by allowing only two function types: The first, termed *table decoding*, is similar to Theiling's masks, except that the bits are required to be contiguous. The second, termed *pattern decoding*, allows non-adjacent bits but restricts the branches to a *matches/does-not-match* pair. The search space is thus reduced from $2^n - 1$ to $n(n+1)/2$. It is obvious that, even should their notion of optimality hold, the restriction on decision functions is significant and likely to exclude more efficient solutions.

As for other features, Theiling's main drawback of not handling non-orthogonality is remedied by duplication on undefined bits: If, after applying a decision tree, an instruction complies with multiple branches, it is replicated in each conforming branch. Apart from non-orthogonality, duplication is also used voluntarily if it entails lower cost. Non-identification bits and specializations are not specifically mentioned, but we assume the authors treat them as in Theiling. We added them to our implementation of Qin in order to be able to process the SPARC ISA. Propositions are not handled.

To address optimization, we note that the authors define optimality in terms of execution speed, which we also find plausible. During generation, however, this cost is obviously not yet available, so the authors choose the average path cost as an approximation. This is defined as the product of path length and probability of occurrence of the leaf. Nodes are considered of equal cost independent of their complexity. This definition is less convincing, since the *complexity* of the node (one vs. multiple tests) *size* of a node (binary vs. multiple branches) and the *location* of the edge (first branch vs. rightmost sibling) might well have a significant effect on the execution speed. To unsettle matters further, this cost value is not available at generation either, since the number of candidate subtrees is too large to analyze at each node being generated. In order to, nevertheless, exhibit some discrimination in choosing a candidate, the authors approximate the average path cost by the cost of a fictitious, best-case Huffman tree. To avoid overly broad and shallow trees, they expand their cost-formula by a "memory consumption" factor in relation to tree breath. To summarize, the restrictions on decision functions exclude possibly more efficient solutions, the algorithm remains very compute-intensive despite limiting the search space, and the cost definition and estimation heuristics are questionable at best.

***Fournel/EFF [21].*** The authors' second algorithm modifies Qin's work to handle logic propositions. Instead of expanding propositions, the algorithm is applied directly to the BDD representation. Their algorithm, however, is not guaranteed to find a decision function for well-defined ISAs: The functions, whether "pattern" or "table", are grown one-bit at a time. They thus start with an initial bit and estimate the cost of the resulting Huffman tree, as suggested by Qin. The next bit is then added. If the cost is larger, the bit is discarded and not used in any further combinations. Practically all propositions of size larger than two will therefore never be regarded as potential decision functions, leading the algorithm to fail. The authors vaguely mention "helping" the decoder with some "special pattern[s]", which presumably means hardcoding the propositions which the algorithm would not otherwise find. This means that, as with Okuda, we had to augment the algorithm in order to prevent it from failing on our ISAs: First, we used the same expanded

TABLE V: Decoder Generation Statistics

| Platform | Decoder | Generation Resources | | | | Decoder Properties | | |
|---|---|---|---|---|---|---|---|---|
| | | Time[s] | | Mem[MB] | | Num. Instr. | Leaves | Total Nodes |
| | | P1 | P2 | P1 | P2 | | | |
| SPARC | Theiling | 0.38 | 0.33 | 54 | 59 | 214 | 214 | 323 |
| | Fournel/Part | 0.37 | 0.32 | 54 | 59 | 214 | 214 | 323 |
| | Okuda | 0.38 | 0.32 | 54 | 59 | 214 | 214 | 323 |
| | Qin | 17 | 16 | 97 | 101 | 214 | 319 | 701 |
| | Fournel/Eff | 184 | 173 | 378 | 365 | 214 | 319 | 701 |
| MIPS32 | Fournel/Part | 289 | 271 | 307 | 280 | 213 | 6345 | 6555 |
| | Okuda | 233 | 195 | 192 | 197 | 213 | 213 | 336 |
| | Fournel/Eff | 313 | 287 | 443 | 396 | 213 | 350 | 705 |
| ARMv7 | Fournel/Part | 75 | 68 | 618 | 606 | 564 | 12026 | 17288 |
| | Okuda | 33 | 32 | 375 | 372 | 326 | 356 | 623 |
| | Fournel/Eff | 3109 | 3645 | 1785 | 1816 | 326 | 6543 | 13981 |

P1: Linux octa-core Intel i7-2600 CPU running (single-thread) at 3.40 GHz with 8 GB of RAM.
P2: Linux quad-core Intel i7-5600U CPU running (single-thread) at 2.60 GHz with 12 GB RAM.
Columns 5 and 6 denote the peak memory consumption during generation. Column 9 is the total number of leaves and internal nodes.

SMT representation for defining the instructions, from which the BDD representation is derived. This step was anything but intuitive for instructions that contain an inequality predicate of the type $fieldA \leq fieldB$ such as MIPS32 BEQC. Second, we expanded the decision function population step: When no pattern or table decision function is found, we fallback to the SMT representation and extract the propositions. Single propositions are tried first. Propositions involving inequality w.r.t. immediates are passed as "pattern" decision functions. Inequality predicates, as well as predicates involving two fields, as required e.g. to differentiate between the MIPS32 BGEZALC vs. BGEUC/BLEUC instructions, can neither be expressed as a "pattern" nor a "table" decision function. We therefore had to pass them as-is, creating a binary node. Should single predicates not prove sufficient, we successively and exhaustively try combinations of predicates, as we did with Okuda, until the instructions can finally be split. This was required 26 times for MIPS32 and a stunning 2600 times for ARMv7. This number, however, can be influenced by the $\gamma$ factor defined by Qin, which impacts the amount of duplication done. Obviously, our extension is quite an upgrade to the algorithm and might have well distorted the subsequent performance measurements.

## COST OF DECODER DECISION TREE ALGORITHMS

### Generation of Decoder Decision Trees

In order to implement the five algorithms, we largely rewrote the open-source processor generation tool TRAP-Gen (TRansactional Automatic Processor GENerator) [24–26] for our purposes. The tool could initially generate SystemC processor models from a common, high-level ISA description in Python 2.7 using Qin's algorithm. Neither MIPS32 nor ARMv7 could be generated using this setup. We thus implemented all five algorithms, together with our extensions discussed above, and used them to generate all decoder combinations. Theiling's and Qin's algorithms could not be applied to MIPS32 or ARMv7 because of their inability to handle propositions. We used pysmt [27] for

TABLE VI: Decoder Execution Statistics

| Platform | Decoder | Cost Models | | | | Experimental Cost | | |
|---|---|---|---|---|---|---|---|---|
| | | Av. Path | Qin | Tadros | | Total Instr. | Runtime[s] | |
| | | | | P1 | P2 | | P1 | P2 |
| | Theiling | 2.925 | 3.241 | 0.1603 | 0.1371 | | 28.40 | 28.93 |
| | Fournel/Part | 2.925 | 3.241 | 0.1603 | 0.1371 | | 28.42 | 29.68 |
| SPARC | Okuda | 2.925 | 3.241 | 0.1603 | 0.1371 | 559 | 28.45 | 29.01 |
| | Qin | 3.900 | 3.618 | 0.1515 | 0.1318 | $\cdot 10^6$ | 29.47 | 30.21 |
| | Fournel/Eff | 3.900 | 3.618 | 0.1515 | 0.1318 | | 29.40 | 30.28 |
| | Fournel/Part | 2.361 | 3.169 | 0.1641 | 0.1373 | | 522.7 | 524.8 |
| MIPS32 | Okuda | 3.188 | 3.589 | 0.1639 | 0.1380 | 463 | 428.8 | 430.0 |
| | Fournel/Eff | 3.120 | 4.356 | 0.2127 | 0.1755 | $\cdot 10^9$ | 581.0 | 607.6 |
| | Fournel/Part | 22.83 | 5.172 | 0.263 | 0.2273 | | 327.1 | 319.2 |
| ARMv7 | Okuda | 6.764 | 5.905 | 0.1917 | 0.1803 | 480 | 274.3 | 267.4 |
| | Fournel/Eff | 4.319 | 32.06 | 1.025 | 0.9712 | $\cdot 10^9$ | 2296 | 2590 |

Platforms P1 and P2 are as described in table V. Cost(Qin) is calculated according to their paper and using $\gamma = 0.5$. Cost(Tadros) is likewise calculated according to the paper [8] using $ET_{if}(i) = ET_{conjunction}(i) = ET_{disjunction}(i) = k * (a * i + b)[ms]$ with $k = \{0.018; 0.021\}$; $a = \{0.21; 0.16\}$; $b = \{0.79; 0.84\}$ and $ET_{switch}(n) = m * (a * log(n) + 1)[ms]$ with $m = \{0.74; 0.45\}$ for platforms P1 and P2, respectively. Column 7 denotes the total number of instructions for ten runs of 30 PolyBench benchmark traces.

handling BDDs in Fournel/Eff. Instruction probabilities, required for training Qin and Fournel/Eff, were obtained from PolyBench traces, as described in the next section. As expected, the distribution was highly imbalanced, much as in figure 1.

The statistics on generating the decoders are summarized in table V. With respect to resources, SPARC is the least challenging due to the absence of propositions. As to algorithms, Fournel/Eff is clearly the most demanding, requiring up to almost an hour and nearly 2 GB of memory.

Regarding the properties of the generated decoders (columns 7-9), the different number of ARMv7 instructions in Fournel/Part is due to the hardcoded PUSH/POP instructions described above. The different number of terminal nodes versus number of instructions is either caused by duplication on non-orthogonality (Qin, Fournel/Eff), by using propositions as decision functions (Okuda, Fournel/Eff), or by the preprocessing step in Fournel/Part, where propositions are converted to a set of satisfying instructions.

## Cost Models of Decision Trees

Since decoder models are typically used in simulation systems, the most prominent efficiency criteria is arguably the execution speed. Since this can hardly be tested for all decoder candidates during generation, we need a cost model that can be applied statically during generation. All the algorithms discussed rely on two assumptions in this respect: First, that decision function complexity can be regarded as $O(1)$ regardless of content, and second, that tree cost is generally a direct function of tree depth. The second assumption comes in two variants, either as simply equating the cost with the average path length (Okuda), or in modulating the latter by the probability of occurrence of the leaf (Qin). To somehow penalize obviously inefficient trees of $depth = 1$, Qin introduces a notion of memory footprint, which basically assigns a cost factor to tree breadth.

A completely different approach to modeling the cost is found in [8]: Assuming that the decoder is a C-like implementation running on general purpose-hardware, the model assigns varying complexities to each decision function depending on its *type* (if-statement, switch-statement, lookup-table, etc.), its *size* (number of conjunctions or disjunctions), and the *location* of the first negative resp. positive term inside a conjunction or disjunction. The *probability of occurrence* of the different encodings is also considered, meaning that more frequent branches contribute a cost weighted accordingly. We performed separate measurements, as described in the paper, to determine the correct coefficients of the model on our platforms.

These different cost models are calculated in table VI for each decoder. Since SPARC defines no propositions, Fournel/Part and Okuda fall back to Theiling, and Fournel/Eff falls back to Qin, generating identical decoders, respectively. The predicative power of each model can be determined by comparing to the decoding runtime in the last two columns: The average path length exhibits no measurable correlation. The cost according to Qin does well on ARMv7 (correlation coefficient 0.999 P1 and P2) but fails on MIPS32 (P1: 0.5; P2: 0.6). The cost according to Tadros does quite a decent job predicting decoder speed on the last two platforms (ARMv7 0.999 and MIPS32 0.8 both platforms). We excluded SPARC from the correlation calculation due to the similar runtimes of the algorithms and the resulting distortion due to noise.

## Experimental Cost of Decision Trees

We proceeded to determine the decoding runtimes as follows: First, we cross-compiled the *mini* version of PolyBench [7] using the GNU toolchains [28–30] with no optimization. All 30 benchmarks were then run on QEMU (user mode) [10] in combination with the toolchain's GDB and the instruction machine code dumped to a text file. Next, a decoder test mode was coded in TRAP-Gen, where, instead of executing binaries, a text file is read and the hex code loaded into the memory model, which, in the functional abstraction level used in test mode, is a simple array. The instruction behavior is ignored in test mode, meaning that the instruction is only read from memory, decoded, then discarded. This avoids distorting the results by the overhead of executing the instruction behavior. Code for monitoring the time was also inserted in the decoder C++ implementation just before and immediately after decoding each instruction and summed up for all instructions of a benchmark.

The last three columns of table VI report the total number of instructions for ten iterations of all 30 PolyBench benchmarks and their corresponding decoding time on two platforms. The results are surprising, inasmuch as the optimization effort done by Qin and Fournel/Eff does not pay off in comparison to greedy algorithms. In fact, they produce consistently slower decoders. This is, in part, almost certainly due to the wrong notion of cost used in the algorithm. It would be interesting to find out to what extent a better cost model would improve decoder speed, e.g. by adapting the algorithm to use the more sophisticated cost model in [8].

## CONCLUSION

We have presented a comprehensive analysis of state-of-the-art tools for generating instruction decoders. We implemented the five published algorithms and used them to generate decoders for three platforms. We then ran a benchmark suite on the generated decoders to assess the execution speed. From a functional point of view, the available tools are either incapable of handling irregular instruction sets, or generate partially wrong results. From a performance perspective, they either do not fully consider the cost of the resulting decoder, or rely on cost models that are largely unusable. Our future work is aimed at developing an algorithm for generating optimized decoders in combination with an accurate cost model.

## REFERENCES

[1] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, “A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1338–1354, 12 2001.

[2] M. Reshadi, P. Mishra, and N. D. Dutt, “Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation,” *ACM Transactions on Embedded Computing Systems*, vol. 8, pp. 20:1–20:27, 2009.

[3] *The SPARC Architecture Manual*, SPARC International Inc. Std. Version 8, Revision SAV080SI9308, 1992.

[4] *MIPS Architecture for Programmers. Volume II-A: The MIPS32 Instruction Set Manual*, Imagination Technologies Ltd. Std. MD00 086, Revision 6.05, may 2016.

[5] *ARM Architecture Reference Manual*, ARM Ltd. Std. ARMv7-A and ARMv7-R edition (C.b), jul 2012.

[6] L. Bortolussi and A. Sgarro, “Hamming-like distances for ill-defined strings in linguistic classification,” *Rendiconti dell’Istituto di Matematica dell’Università di Trieste. An International Journal of Mathematics*, vol. 39, pp. 105–118, 2007.

[7] L.-N. Pouchet, “PolyBench/C,” http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, version 4.2.1.

[8] L. Tadros, “A cost model for decoder decision trees,” in *Proceedings of the 2020 European Symposium on Software Engineering (ESSE)*. New York, NY, USA: Association for Computing Machinery, nov 2020, pp. 142–147.

[9] Free Software Foundation, Inc., “GNU binutils,” http://www.gnu.org/software/binutils.

[10] F. Bellard, “QEMU. Open Source Processor Emulator,” wiki.qemu.org/Main_Page, version 6.2.0.

[11] N. Ramsey and M. F. Fernandez, “The new jersey machine-code toolkit,” in *Proceedings of the USENIX Technical Conference*. Berkeley, CA: USENIX Association, jan 1995, pp. 289–302.

[12] G. Hadjiyiannis, P. Russo, and S. Devadas, “A methodology for accurate performance evaluation in architecture exploration,” in *Proceedings of the Design Automation Conference (DAC)*, jun 1999, pp. 927–932.

[13] A. Baldassin, P. Centoducatte, S. Rigo, D. Casarotto, L. C. V. Santos, M. Schultz, and O. Furtado, “An open-source binary utility generator,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 2, pp. 27:1–27:17, apr 2008.

[14] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, “EXPRESSION: A language for architecture exploration through compiler/simulator retargetability,” in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. New York, NY, USA: Association for Computing Machinery, 1999, p. 100–es.

[15] T. E. Jeremiassen, “Sleipnir - an instruction-level simulator generator,” in *Proceedings of the International Conference on Computer Design (ICCD)*, sep 2000, pp. 23–31.

[16] M. Abbaspour and J. Zhu, “Retargetable binary utilities,” in *Proceedings of the Design Automation Conference (DAC)*, jun 2002, pp. 331–336.

[17] N. Y. Fokina and M. A. Solovev, “Automated generation of machine instruction decoders,” vol. 45, pp. 390–397, 2019.

[18] Y. Klimiankou, “Rapid instruction decoding for IA-32,” in *Perspectives of System Informatics*, N. Bjørner, I. Virbitskaite, and A. Voronkov, Eds. Cham: Springer International Publishing, 2019, pp. 1–9.

[19] H. Theiling, “Generating decision trees for decoding binaries,” in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, jun 2001, pp. 112–120.

[20] T. Ratsiambahotra, H. Cassé, and P. Sainrat, “A versatile generator of instruction set simulators and disassemblers,” in *Proceedings of the International Symposium on Performance Evaluation of Computer Telecommunication Systems (SPECTS)*, jul 2009, pp. 65–72.

[21] N. Fournel, L. Michel, and F. Pétrot, “Automated generation of efficient instruction decoders for instruction set simulators,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, nov 2013, pp. 739–746.

[22] K. Okuda and H. Takeyama, “Decision tree generation for decoding irregular instructions,” in *Proceedings of the Conference on Design, Automation Test in Europe Conference (DATE)*, mar 2016, pp. 1592–1597.

[23] W. Qin and S. Malik, “Automated synthesis of efficient binary decoders for retargetable software toolkits,” in *Proceedings of the Design Automation Conference (DAC)*, jun 2003, pp. 764–769.

[24] L. Fossati, “trap-gen,” https://code.google.com/archive/p/trap-gen/.

[25] ——, “Development of the systemc model of the LEON2/3 processor,” Politecnico di Milano, Tech. Rep., 2012.

[26] L. Tadros, “A SystemC register model for multiple levels of abstraction using advanced object-oriented design patterns,” *International Journal of Computer Theory and Engineering (IJCTE)*, vol. 9, no. 3, jun 2017.

[27] M. Gario, A. Micheli, and F. B. Kessler, “PySMT: a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms,” https://github.com/pysmt/pysmt/, version 0.9.0.

[28] I. Free Software Foundation, “crosstool-NG,” http://crosstool-ng.org/download/crosstool-ng/, version 4.9.4.

[29] ——, “GCC Linux GNU Toolchain,” http://codescape.mips.com/components/toolchain/2017.10-07/index.html, version 2017.10-05.

[30] ——, “GCC Linux GNU Toolchain,” https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads, version 11.3.0.

**LILLIAN TADROS** obtained her Dipl.-Ing. in electrical engineering and information technology from the Ruhr-Universität Bochum, Germany. She joined the Institute for Robotic Studies of the Technische Universität Dortmund, Germany, where she is currently a research assistant. Her research focuses on simulation models of multi- and many-core platforms for embedded and cyber-physical systems.