Generating Decision Trees for Decoding Binaries *

Henrik Theiling
Universität des Saarlandes and
AbsInt Angewandte Informatik GmbH
Saarbrücken, Germany
theiling@cs.uni-sb.de

ABSTRACT

Tools reading binary code, like analysers, debuggers, disassemblers, etc., need to decode the target's machine code. A decision tree is often used to represent the decoding function.

Manually writing a decoder is a lengthy and error-prone task. It is desirable to be able to use the vendor's instruction code manual and to easily transform the documentation into a specification that a tool can use to generate a decoder.

This paper presents a novel algorithm that computes a decision tree from machine code bit patterns alone. Neither the bit fields of the machine code, nor the width of the machine command, nor the order in which the bits should be decoded need to be specified. The decoding algorithm accesses any significant bit exactly once during decoding.

1. INTRODUCTION

In our research project Transferbereich 14 'Runtime Guarantees for Real-Time Systems', we faced the problem that machine decoders for several architectures had to be written (*see* [1]). These are used to decode byte streams into instructions, which are then classified precisely for safe analysis.

Manual implementation for every target architecture, however, is an error-prone task. Instead, it is desirable to use the vendor's machine code documentation directly to write a specification and have the decoder generated automatically. This way, for every target, only the specification has to be written increasing the degree of safety.

This paper presents an algorithm whose input is a set of bit patterns, one for each machine instruction to be recognised. For each bit, it is specified whether its value is zero, one or insignificant. The algorithm then uses the set of bit patterns to recursively compute a decision tree for decoding. The decoding algorithm only accesses those bits that the user specified to be significant, and only tests each bit maximally once. Figure 1 shows a simple example.

A major advantage is that our algorithm needs no user defined order in which bits shall be tested and no specification of bit fields in the machine code. This facilitates writing a specification from vendor manuals that are organised as a long list of machine instructions as well as from bit field based manuals that are divided into instruction groups by e.g. a primary opcode. The framework has interfaces for both specification methods. Other styles are easily supported by simply listing all possible instructions. We even succeeded in converting a manual in PDF format into a specification automatically.

The generated decision tree consists of inner nodes that describe how to make a decision and of leaves that contain results. In our framework, the leaves contain machine instruction classifications suitable for reconstruction of a control flow graph (*see* [21]). Each decision node describes which bits have to be tested in the input bit string to select a child at this node.

The decision tree should be as shallow as possible in order to require the smallest number of tests. Our algorithm computes a partition at each inner node of the decision tree that tests the maximally possible number of bits at that node. This involves testing non-adjacent bits in one step. This is an advantage compared to other approaches.

Furthermore, in contrast to many other types of decision trees, our tree is required to always test all significant bits for all inputs even if the final selection decision can be drawn from fewer bits. This is because the algorithm operates in the safety critical environment of real-time systems analysis, where every part is required to be *safe*. In this safety critical environment, the putatively superfluous test serves to detect malformed input bit string.

Our algorithm is able to handle specialisations of machine instructions, i.e., patterns that are subsumed by others. This is done by a default child that is used if no special child was found during a decision. This child must always be a leaf node.

*The work of the USES group (**U**niversity of the **S**aarland **E**mbedded **S**ystems group) is partially supported by Transferbereich 14 of the Deutsche Forschungsgemeinschaft.

Instr.	Bit Pattern b_0	b_1	b_2	b_3
A	0	$*$	$*$	0
B	1	$*$	$*$	1
C	0	$*$	0	1
D	0	$*$	1	1
E	0	0	0	0

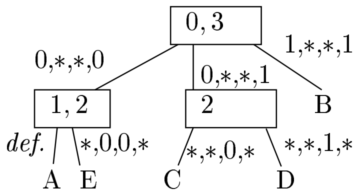

Figure 1: Example input for four bit commands and the computed decision tree. The nodes are labelled with the set of bit indices tested in that node. Specialised bit patterns are handled by a default edge labelled *def.* (E is a specialisation of A).

We have implemented a decoder for the Coldfire architecture and for the PowerPC using this technique.

This paper is structured as follows. Section 2 introduces decision trees for decoding machine instructions formally. Then, Section 3 introduces our algorithm in detail. After that, Section 4 presents an efficient way of implementing the algorithm and shows its run-time. Section 5 shows experimental results from our practical tests. Section 6 compares our work to that of others and, finally, Section 7 draws conclusions.

2. DECISION TREES FOR DECODING

Given a mapping of bit patterns to machine instruction classifications, the goal is to construct a decision tree for implementing that mapping. Formally, this can be described as follows:

DEFINITION 1. *Let* $\mathbb{B} = \{0, 1\}$ *be the set of bits. Let* $n \in \mathbb{N}_0$. *A* bit pattern b *is a string of bits together with a set of significant bit indices:* $b \in \mathbb{B}^n \times \mathfrak{P}(\mathbb{N}_0)$. *Thus,* n *is the width of the bit patterns.*

Let D *be a set of machine instruction classifications (whose precise structure is irrelevant for the algorithm). Then* $f_0 : \mathbb{B}^n \times \mathfrak{P}(\mathbb{N}_0) \to D$ *is a mapping from bit patterns to classifications.*

A pattern could have been defined as a tuple over $\{0, 1, *\}$, but this would have complicated the definition of the algorithms in the following.

Also note that defining that all bit patterns have the same length n does not mean that machine commands have the same length. Shorter patterns can simply be padded with insignificant bits.

In the following, we will regard the input for the algorithm as the set that represents f_0, so we will treat $f_0 \subseteq \mathbb{B}^n \times \mathfrak{P}(\mathbb{N}_0) \times D$. Let $F := \mathbb{B}^n \times \mathfrak{P}(\mathbb{N}_0) \times D$ to improve readability.

For a triple $(b, m, d) \in F$, we define

$$\begin{array}{rcll} bits(b, m, d) & = & \mathrm{b} & \textit{the bit values} \\ mask(b, m, d) & = & \mathrm{m} & \textit{the indices of} \\ & & & \textit{significant bits} \\ data(b, m, d) & = & \mathrm{d} & \textit{the classification} \end{array}$$

We will often show insignificant bits as $*$.

In the following definition, the instruction classifications are used as terminal nodes of decision trees for binary decoding.

DEFINITION 2. *A* decision tree *is a labelled tree* (V, E) *where* $V = D \cup N$, D *the set of terminal nodes,* N *the set of inner nodes and* $E \subseteq N \times (N \cup D)$ *the edges of the tree.*

Node labels are assigned by a function $node\_label(v) \subseteq \mathbb{N}_0, v \in N$ *which are the bit numbers of the bits to be tested at node* v.

For $e \in E$ *let* $edge\_label(e) \in \mathbb{B}^n \cup \{default\}$ *be the function that labels an edge* e.

A decision tree can be used to select a machine code classification from a bit string as follows. Let $(b_0, \ldots, b_{k-1}) \in \mathbb{B}^*$ be the input bit string of length k. In the decoder application, this is a block of bytes from the executable of which the first instruction is to be classified by a $d \in D$. d stores the bit width of the command, so that this many bits can be skipped and the decoding can advance to the next instruction in the bit string.

The following selection algorithm selects the first instruction from the bit string by using the decision tree.

During the algorithm, v is the current node in the decision tree and v_d will be the most recently encountered default node, i.e., the most specific one.

Step 1 Start the selection by letting v be the root node of the tree and v_d = undef.

Step 2 If at the current node $v \in N$, there is an edge $e = (v, v')$ such that $edge\_label(e)$ = default, then let $v_d = v'$.

Step 3 If v is a leaf, that node is the algorithm's result.

Step 4 At the current node $v \in N$, try to select an outgoing edge $e = (v, v')$ such that $\forall i \in node\_label(v)$:

$$\begin{array}{ll} & i < k \\ \wedge & edge\_label(e) \neq \text{default} \\ \wedge & edge\_label(e)_i = b_i \end{array}$$

If such an edge e exists, go to Step 2 with $v := v'$.

If e does not exist, and if $v_d \neq$ undef, go to Step 3 with $v := v_d$ and let v_d = undef.

Otherwise, let the selection fail, since no classification exists for the input bit pattern.

This selection algorithm keeps track of the most recent default node in order to be able to backtrack of if the selection algorithm fails on subsequent nodes. The backtracking is important for the algorithm only to fail if there is no matching pattern:

$$\begin{array}{ll} \mathrm{A} & * * * \\ \mathrm{B} & 0\,0 * \\ \mathrm{C} & 1 * 1 \end{array} \tag{1}$$

Without keeping track of old default nodes, no node would be found for the input $(0, 1, 0)$, although A matches. Note that we do not need a stack of default nodes for backtracking, since the default nodes are required to be leaves.

Note that this selection algorithm can be implemented very efficiently if $(b_0, \ldots, b_{k-1})$, $\mathit{edge\_label}(e)$ and $\mathit{node\_label}(v)$ are implemented as machine words (insignificant bits set to zero). We will write the conversion to bit tuples (thus, machine words) using parentheses, e.g.:

$$(\mathit{node\_label}(v)) = (m_i)_{i=0,\ldots,n-1} \text{ where} \tag{2}$$

$$m_i = \begin{cases} 1 & \text{if } i \in \mathit{node\_label}(v) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Children of $v \in N$ are stored in a hash table at node v that is indexed with the labels $\mathit{edge\_label}(e)$. Then, a child can be selected by indexing the hash table with $(b_0, \ldots, b_{k-1})$ `bit_and` $(\mathit{node\_label}(v))$. If the hash table lookup fails, a potential default child can be selected.

The goal of the algorithm presented in the following section is to compute N, E, *node_label* and *edge_label* in such a way that the number of edges and nodes is kept small. Insignificant bits shall never be tested by the above selection algorithm. Significant bits shall only be tested once.

We do not expect that a decision tree can be built for all input bit pattern sets. Consider the following patterns:

$$\begin{array}{ll} \mathrm{A} & 0 * * * \\ \mathrm{B} & * * * 0 \end{array} \tag{4}$$

It is unclear which pattern should be selected for e.g. $(0, 0, 0, 0)$.

One way of resolving this problem is by assigning priorities to ambiguous bit patterns. However, our bit patterns are descriptions of micro processors, so we expect them to be unambiguous since the processor can identify them uniquely as well. Therefore, we decided that prioritisation need not be included in our algorithm.

Furthermore, our algorithm will not handle pattern sets like the following, which provide a unique match for all inputs, but require that insignificant bits be tested.

$$\begin{array}{ll} \mathrm{A} & 1\,0 * \\ \mathrm{B} & 0 * 1 \\ \mathrm{C} & * 1\,0 \end{array} \tag{5}$$

Again, we assume that microprocessors will not have machine code bit patterns organised like that. Techniques to handle these patterns by testing some of the insignificant bits are described in [10] for functional languages with argument pattern matching.

3. ALGORITHM

When building the decision tree, we consider a decision tree where all possible bit combinations are checked in the root node to be infeasible due to the mere number of $O(2^n)$ edges.

The goal will be to have few nodes and few edges. We decided that the we will not check any insignificant bits, which will naturally bound the number of nodes and edges, because bit patterns not in the input will not be checked in the tree. With this prerequisite, the depth of the tree will be the measure of quality.

The principle of the construction will be to make inner nodes in such a way that they test maximally many significant bits at once, since they have to be tested anyway. On the other hand, prevent testing any insignificant bits in order to keep the out-degree of the nodes small.

The idea of our algorithm is recursive partitioning of the input set of bit patterns. First, a set of bits is computed that are significant for all patterns. Then, the input set is partitioned into subsets that have different values for these significant bits. For each set, the algorithm recurses. The recursive function of the algorithm returns a new node with the sub-tree underneath. Together with the subset of the input bit patterns, a mask of already tested bits will be passed down the recursion to prevent double testing of bits (this will be called *gmask* in the algorithm).

At the beginning of the algorithm, we assume all bits to be potentially significant, so the initial bit mask is $\{0, \ldots, n-1\}$. So in order to compute the decision tree, make_tree is invoked in the following way, where f_0 is the machine code bit patterns from the user.

$$\mathsf{make\_tree}\ (f_0,\ \{0, \ldots, n-1\})$$

The algorithm is depicted in Figure 2 and will be described in detail now.

Throughout the algorithm, we require $f \neq \{\}$. This is needed for well-definedness at some points.

Step 1 of the algorithm computes a bit pattern of bits that are significant in all patterns in f. The bit pattern is maximal, i.e., a bit is only found to be insignificant if there is a pattern where it is insignificant, due to the definition of the set intersection.

Deciding about termination in Step 2 works by checking that no significant bits remain and that f is a singleton. It is necessary to ensure that no significant bits are left in *mask*, since the selection algorithm must test all significant bits in the input bit string even if there is only one candidate left for selection. This must be done in order to detect malformed input bit strings.

In Step 3, we know that no leaf, but an inner node will be generated.

For the sake of simplicity, explaining Step 4 will be postponed. For now, we assume that if a default node was selected, it is the correct one, that its pattern has been excluded from f and that $mask \neq \{\}$ after Step 4.

In Step 5, the selection mask for the new node is known and assigned as a node label.

Partitioning in Step 6 groups bit patterns that have the same bit values for the significant bits defined by *mask*. The function returns the set of equivalence classes for each element of f. An efficient implementation will be given later.

fun partition $(f \subseteq F,\ mask \subseteq \mathbb{N}_0)$
    return $\{$ equ_class$(p)\ :\ p \in f \}$
    where equ_class $(p) =$
        $\{p'\ :\ p' \in f$ **such that** $\forall i \in mask :$
                $bits(p')_i = bits(p)_i\}$

Note that each equivalence class equ_class(p) contains at least one element, namely p, so no empty sets will be used during the recursive calls in Step 7, thus the new sets all fulfil the requirement made in Step 1.

Finally in Step 7, the function make_tree invokes itself recursively for all subsets found in Step 6. The bits that have been tested at node v are excluded from the new *gmask* to prevent double testing the same bits. Computation of an appropriate edge label remains to be defined.

fun get_label $(f_i \subseteq F,\ mask \subseteq \mathbb{N}_0)$ **returns** $\in \mathbb{B}^n$
    – *Extract significant bits from some element of* f_i
    return value_bits(p) **for some** $p \in f_i$
    where value_bits $(p)_j = \begin{cases} bits(p)_j & \text{if } j \in mask \\ 0 & \text{otherwise} \end{cases}$

This function is well-defined since for every pair of elements of f_i, the bit values selected by *mask* are equal due to the construction of the equivalence classes in the partitioning step.

3.1 Default Nodes

In the previous section, the handling of default nodes was postponed. To understand when these are needed, assume the following input to the algorithm:

$$\begin{aligned} f \ = \ \{ & ((0,0), \{\}, A), \\ & ((0,0), \{1\}, B)\} \end{aligned}$$

Here, A subsumes B and the computation of *mask* in Step 1 will yield $mask = \{\}$ with f not being a singleton.

At this point, the default node should select A and the decision node should use the second bit to check whether B should rather be selected. If we made A the default node and repeated the computation of *mask* in this example, the effect would be as desired.

Up to now, it is not clear whether the default node must be a single bit pattern. The subset of f that will be used for the default node will be called M in the following.

For the construction of the default node, we need to think about correctness. To construct a decision tree that selects the correct bit patterns, it is required that all elements in M, which are selected at the current node by default, must subsume those in $f \setminus M$, which are tested first. This means that the bit mask of any element in M must be a proper subset of the bit mask of any element in $f \setminus M$, i.e., the default node must test strictly less bits.

Example

	b_0	b_1	b_2	b_3
A	1	0	$*$	$*$
B	1	0	0	$*$
C	1	0	0	0

(6)

In this example, assume b_0 and b_1 have been processed already, so we have $gmask = \{2,3\}$. If it is clear that A is the default node since the bit patterns of B and C are both subsumed.

When the algorithm arrives at a node requiring a default node, significant bits common to all masks will always have been processed already, because otherwise the intersection of the masks is non-empty, therefore, the algorithm would not have arrived in get_default. So finding the default node is very easy: its set of remaining significant bits must be empty. There must be at most one node with this property, otherwise, the input set is undecidable as A and B in the following example:

	b_0	b_1	b_2	b_3
A	1	0	$*$	$*$
B	1	0	$*$	$*$
C	1	0	0	$*$
D	1	0	0	0

(7)

So we have seen that a) finding the default node is trivial by searching for an empty remaining bit mask and b) the algorithm need not recurse in the default node, because it must be a single input bit pattern: $|M| = 1$. We can simply use the data of that pattern as a leaf node.

Figure 3 shows the function get_default.

The algorithm fails if the set of bits is still irresolvable after exclusion of the default node. An example input for this situation would be the following:

```
fun make_tree (f ⊆ F, gmask ⊆ ℕ₀)
returns ∈ N ∪ D
    – Step 1: compute a bit mask of bits that are
      significant for all patterns
    mask := gmask ∩ ⋂_{p∈f} mask(p)

    – Step 2: possibly terminate: f must be singleton
    if mask = { } and |f| = 1
        return data(p) where f = {p}

    – Step 3: construct a new node
    v= new InnerNode

    – Step 4: decide about default node and edge
    if mask = { }
        (v^def, f, mask):= get_default (f, gmask)
        e^def := new Edge (v, v^def)
            with edge_label(e^def) := default

    – Step 5: label the current node
    node_label(v) := mask

    – Step 6: make partition of f using mask
    {f_1, ..., f_k} := partition (f, mask)

    – Step 7: recurse on subsets and add edges
    for i in {1, ..., k}
        v':= make_tree (f_i, gmask \ mask)
        e':= new Edge (v, v')
            with edge_label(e') := get_label(f_i, mask)

    – Step 8: return the new node
    return v
```

Figure 2: Decision tree generation algorithm

```
fun get_default (f ⊆ F, gmask ⊆ ℕ₀)
    – Compute the set of bit patterns that have
      empty remaining bit masks
    M := {p : p ∈ f and mask(p) ∩ gmask = { }}
    if |M| ≠ 1
        fail

    – Similar to Step 1, get a mask. Fail if empty.
    mask := gmask ∩ ⋂_{p∈f\M} mask(p)
    if mask = { }
        fail

    – Return the result, M is a singleton
    return (data(p), f \ M, mask) where M = {p}
```

Figure 3: The function that computes the default node and the new bit mask

$$\begin{array}{c|cccc} & b_0 & b_1 & b_2 & b_3 \\ \hline \mathrm{A} & 1 & 0 & * & * \\ \mathrm{B} & 1 & 0 & 0 & * \\ \mathrm{C} & 1 & 0 & * & 0 \end{array} \tag{8}$$

This function fulfils the constraint that after Step 4: $mask \neq \{\}$.

3.2 Unresolved Bit Patterns

get_default can be extended to handle bit patterns like (5) but care must be taken: obviously, a bit has to be tested that is insignificant in some pattern. In (5), any bit could be chosen for disambiguation purposes. However, because machine instructions may have different lengths, decoding might fail although a valid instruction is in the input. As an example, consider (5) and an input bit string of $(1, 0)$. Clearly, pattern A should be selected. But if the algorithm had selected bit index 2 to resolve the pattern set, a node checking for a bit outside the input bit string is encountered before the correct decision can be drawn. Decoding would fail because the input bit string has fewer bits.

This is a similar problem as that of pattern sets in some lazy functional languages, where accessing insignificant arguments of a function in order to select a pattern might lead to non-termination if that operand does not terminate (*see* [10]).

Furthermore, in practice, we may have byte-swapped input, so in an implementation, the problem of test bits being outside the input bit string occurs at both sides of the bit patterns if the bit string is not known to be byte-swapped or not at pattern compilation time. In (5), the second bit should be selected, because this bit is either significant, or there are significant bits to both sides (and holes are impossible). So it can be concluded that the input bit string will contain the middle bit, if it contains a valid instruction.

To optimise the disambiguation wrt. the number of nodes and edges that are required, the number of patterns for which insignificant bits must be tested should be minimised.

This problem is non-trivial, but we do not expect it to occur with machine specifications anyway, so we did not try to implement a way of disambiguation.

3.3 Termination

Termination happens by either failing, in get_default, or succeeding, in which case the recursion comes to an end normally.

It can be seen immediately that in each new incarnation of a recursion, *gmask* contains fewer bit indices, as some are deleted in Step 7 by the non-empty *mask*, so eventually, *gmask* becomes empty and the algorithm terminates.

3.4 Proof of Correctness

Correctness of the selection and make_tree algorithms is defined in the following way. Assume that a decision tree can be computed. Using that tree, the selection algorithm

1. always selects an input pattern that is matching,
2. selects the most specific pattern if more than one pattern matches,
3. never fails if a pattern matches unambiguously,
4. always tests all significant bits to have the desired value.

We will prove this claim by induction on the maximal number n of remaining significant bits in the input pattern set, taking into account the value of *gmask*.

n = 0: Step 1 finds $mask = \{\}$. There are two possibilities:

Case 1: $|f| = 1$. This means that a node with the data of that pattern can be constructed. The recursion terminates.

During the selection, all of the above correctness prerequisites are fulfilled: the pattern matches (no more bits are significant), the most specific one is selected (there is only one), the algorithm does not fail (so it does not fail even of matching patterns exist), and all remaining significant bits have been tested (there is none left to be tested).

Case 2: $|f| \geq 1$: The pattern set is irresolvable and the algorithm fails. (3) holds, too, since the set is ambiguous.

n + 1: Assume the claim to be true for all numbers of remaining significant bits $\leq n$. We now prove this is is true for $n + 1$. After Step 1, there are two major cases:

Case 1: $mask \neq \{\}$: This means that no default node is needed.

The claim holds for all recursion steps, because some bits are removed from *gmask*, so $|gmask| \leq n$ in all the recursive steps.

1. Partitioning makes clusters of patterns that are equal at the bits in *mask*. So when testing these bits, the selection algorithm makes the only correct choice and selects a pattern that is matched by the bits in *mask*. Because the claim holds for the recursion, subsequent selection steps also select the correct patterns for the remaining bits. So in total, the correct pattern is selected.
2. Because the default node is selected after all other patterns have been tested, the most specific subset of patterns is selected, when it is not selected, since these patterns have strictly more bits set. This holds in the recursion steps, too, so the most specific pattern is selected.
3. If no pattern matches, the selection algorithm either chooses the most specific default node (which matches) or fails, which means that no default node was available, so no pattern matches.
4. The bits in *mask*, which are significant in all patterns, will all be tested. The recursive steps make sure that all other significant bits will also be tested, so the claim is fulfilled for $n + 1$, too.

Case 2: $mask = \{\}$. If there is only one pattern left, the argument is the same as for $n = 0$.

Assume that $|f| > 1$. A default node will be selected. If this succeeds, we have shown in Section 3.1 above that the default node is the least specific node and that all its significant bits have been processed. Furthermore, the number of significant bits of the default node is $\leq n$ since the other patterns all have strictly more significant bits.

1. If it is selected, it is the correct choice as the number of bits is $\leq n$.
2. Because the default node is selected after all other patterns do not match, the most specific one is selected (the other patterns have strictly more significant bits).
3. Analogously to the previous case, the selection algorithm only fails if no default node is available.
4. All significant bits of the default node have been tested.

For the other patterns, a new mask is computed in the same way as before, but excluding the default node. So the argument is the same as for $mask \neq \{\}$ (note that it has been shown that the default node is the least specific one, so the argument about specificity is valid, too).

Because we start the recursion with *gmask* containing all potentially significant bits, and the claim was proven for all remaining significant bits, the claim holds for all significant bits at the beginning.

4. EFFICIENT IMPLEMENTATION

The algorithm can be implemented very efficiently by using bit masks if we require that a machine word has at least n bits, thus enough to store all values $\in \mathbb{B}^n$ directly. It can be assumed that operations on machine words, like bitwise 'or', 'and' or 'not' work in $O(1)$.

Bit masks can be stored in machine words by setting bits to 1 if the bit number of that bit is in the masking set or to 0 otherwise.

The input set f_0 can be stored efficiently in an array. The current subset can be marked using two integers as parameters of make_tree marking the first and last index of the subset in this array. In the recursive step, this works as follows: When partitioning, the sub-array is sorted locally considering only the bits in the bit mask. This way, the new partitions are adjacent in the sub-array and can be passed down in the same way.

4.1 Complexity

With the help of the previous section a run-time for the algorithm can be computed. Let $m = |f_0|$ be the size of the input set and n the maximal width of the input patterns.

In each step of the algorithm, almost all steps work in $O(|f|)$ but partitioning takes $O(|f| \log |f|)$ due to the sorting that is done.

In each step, at least one bit is removed from *gmask*, so recursion depth is maximally n. In the worst case, only two partitions are made in each step, one consisting of 1 element, the other of all but this element. Then the run-time is

$T(n,m) = O(n+\sum_{i=m,\ldots,m-n+1} i \log i) = O(n+n\cdot m \log m)$. So if $n = m$, it becomes $O(m^2 \log m)$.

This worst case run-time looks slow. However, we expect m much larger than n, because the input is machine code patterns, where there are much more commands than bits in a machine word. We also expect that the recursion is much more shallow than n, since usually only few groups of bits have to be looked at to select a command. In total, we expect the recursion depths to be around $\log(m)$, since with $\log(m)$ bits, maximally m commands can be coded. Run-time then becomes quasi-linear.

The experiments have shown that the trees are even more shallow than $\log(m)$, so practice has justified the assumption.

4.2 Generalisation

The bit patterns that were used above can be viewed as sets of boolean attributes. This means that the algorithm is directly usable in applications where property tables with boolean attributes are the input. The benefit is parallel testing of attributes in each decision step and the possibility to have insignificant attributes.

Of course, the alphabet could also be extended to be non-boolean (only an equality operator is required). However, the algorithm's major efficiency results from working with machine words, so that parallel testing of several attributes works in $O(1)$. But if the attribute values can be distributed to several bits (e.g. a four-value attribute uses two bits instead of one), the algorithm can still be applied.

5. EXPERIMENTAL RESULTS

The algorithm was implemented in our analysis framework for real-time systems (*see* [22, 1, 9, 21]). It consists of only 900 lines of C++ code, which include the selection algorithm, the decision tree algorithm and two interfaces to the algorithms for different styles of input data.

We have written machine descriptions for the IBM PowerPC (*see* [7]) and the Motorola Coldfire (*see* [15]) architecture.

Moreover, with the help of a small script (only 194 lines of Perl) we could automatically convert a PDF manual (*see* [8]) for the Infineon C166 architecture into a specification with all the processor's bit patterns. The result is a template file where the command classifications can be filled in. We included this file in our tests of the decision tree building algorithm. Writing this script only took a few hours.

The time for tree generation was negligible for all test inputs. For the Coldfire specification it took less than 0.2 seconds on a Pentium III with 650 MHz. The specification had 908 instruction patterns, because the Coldfire is quite a complex processor due to its CISC history.

Self-evident by the construction of the selecting mask, but still not uninteresting, is the fact that our algorithm generated a root node that tested exactly for the primary opcode for both the PowerPC and the Coldfire processor.

The generated decision tree for the Coldfire architecture had a maximal leaf node depth of 5 (including the root node). The average depth was 2.76, i.e., this is the average number of decisions needed to decode one instruction at the front of a bit string, which is very few.

To compare the algorithm to one that is limited to testing adjacent bit groups instead of arbitrary ones, we explicitly forced it to select only adjacent bits for testing. The generated tree then had a maximal depth of 7 and an average depth of 3.13.

The PowerPC 403 specification has 210 instruction patterns and the time to generate the decision tree was below the precision limits of measurement. The maximal leaf node depth was 3 including the root node. The average depth of the PowerPC decision tree was 2.2. The PowerPC is a RISC architecture, so these 3 levels are explained very easily: the root node tests opcode 1, the next node tests opcode 2 and some commands are distinguished to either set the condition code bits or not, thus another bit was tested for those commands in the third node. And because of the layout of the bits of the opcode 1 and 2, the decision tree always tested adjacent bit groups in this tree already.

The C166 specification has 230 instruction patterns and again, the time to generate the decision tree was below the precision limits of measurement. The maximal leaf node depth was 4, the average depth was 2.41. The tree with bit group tests of adjacent bits had a maximal depth of 6, and the average depth increased marginally to 2.48, because only very few commands had longer decoding paths.

The feature of default nodes for instructions that are subsumed by others was needed several times in the machine code specification of the Coldfire architecture. Some examples are two sorts of branch commands which have different sizes if the displacement constant of the shorter variant has a value of 0. Another example are the divide and remainder instructions which share the same opcodes but are distinguished by whether two of the three operands are identical.

The experiments showed that writing a decoder for machine code of a new processor has become much easier and less time-consuming, less error-prone and, therefore, safer.

6. RELATED WORK

There is a lot of work on decision trees in many different areas of Computer Science (*see* [14, 20]). A survey of decision tree building methods is given by Sreerama K. Murthy in [16]. The basic principles of recursive partitioning and finding a splitting method are introduced there. Methods of compiling lazy pattern matching can be found in [10].

[4] presents an algorithm for disassembling that searches linearly for the given bit patterns, thus not using a decision tree and involving $O(n)$ decoding runtime where n is the number of instruction patterns.

The contribution of this novel algorithm is the splitting function for bit patterns given as machine words. They form a special class of multiple boolean attributes, which this algorithm handles in parallel. To the best of our knowledge, no solution for this problem has been published before.

Other approaches to real-time systems analysis often do not read machine code, so the problem of decoding bit strings does not occur there. Either assembly (*see* [12, 13]) or source code (*see* [17, 3]) are used as input. Because for modern processors, the low-level effects of caches and pipelines need precise location information which is only available with machine code for some machines (not even on assembly level, because linking is not performed), our framework reads machine code from linked executables.

One other algorithm that tries to solve the same problem as ours is found in the New Jersey Machine-Code Toolkit (*see* [19, 18]). The decision tree building algorithm used in that framework, however, it is not described in publications. Looking at the documentation in the source code, it can be seen that it uses opcodes defined by the user to compute the decision tree. For that framework, this information is available from the specification.

Another widely used way of decoding machine code is the GNU Binutils package (*see* [6]). One tool that uses Binutils is the Wisconsin Architectural Research Tool Set (*see* [5, 11]). In Binutils, decoders are written manually for each processor. E.g. for the Motorola 68xxx processor, the decoding is done by linearly searching the bit patterns and limiting the search algorithm to partitions by sorting the patterns by their 4-bit primary opcode. This partitioning was done automatically and in a generic way by our algorithm.

7. CONCLUSION & FUTURE WORK

In this paper, we have presented a novel algorithm for decision tree generation from bit patterns alone. The algorithm does not need any user-provided selection of bit fields, so no classification of opcodes and sub-opcodes is needed. Our algorithm can handle specialised instructions (those that are subsumed by others) by default nodes. It is generic and can be applied to various types of input data.

By handling non-adjacent groups of bits in one step, the decision tree is kept more shallow than with opcode or single bit-oriented approaches, improving decoding speed.

By its degree of automatism, the algorithm make porting to new architectures more easy.

The implementation is very fast by using $O(1)$ machine word operations and has proven in practice to be integrable into an existing framework.

We have stressed here that the safety properties of the algorithm make it suitable for safety-critical applications like real-time systems analysis.

Testing techniques of machine specifications like [2] would be nice to have in our framework. This is a future extension for bug detection.

Further work will include porting our decoder for the Infineon Tricore architecture to the new decoding technique in order to get a simpler executable for control flow construction, which is easier to maintain.

8. ACKNOWLEDGEMENTS

I would like to thank Reinhard Wilhelm for his support and for fruitful discussions about this work. Thanks are also due to Nicola Geismann, Reinhold Heckmann, Björn Huke and Markus Löckelt for proof-reading and giving many valuable hints. Additionally, thanks to the anonymous reviewer #2 who helped to fix a flaw in the selection algorithm.

9. REFERENCES

[1] C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems — The USES Approach. In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, Paderborn, 1999.

[2] M. Fernández and N. Ramsey. Automatic Checking of Instruction Specifications. In *Proceedings of the 19th International Conference on Software Engineering.* ACM Press, 1997.

[3] J. Gustafsson. Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD Thesis, Uppsala University, Mälardalens Högskola, 2000.

[4] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *36th Design Automation Conference (DAC'99)*, pages 927–932, 1999.

[5] http://www.ee.princeton.edu/~yauli/cinderella-3.0/. *Cinderella 3.0 Home Page*.

[6] http://www.gnu.org. *GNU Binutils*.

[7] IBM. *PPC403GCX Embedded Controller, User's Manual*, 1997.

[8] Infineon. *Instruction Set Manual for the C16x Family of Siemens 16-Bit CMOS Single-Chip Microcontrollers*, 1997.

[9] D. Kästner. TDL - A Hardware and Assembly Description Language. Technical report, Universität d. Saarlandes, 2000.

[10] A. Laville. Comparison of Priority Rules in Pattern Matching and Term Rewriting. *Journal of Symbolic Computations*, 11(4):321–348, April 1991.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*, 1996.

[12] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), 1995.

[13] S.-S. Lim, J. Hee Han, J. Kim, and S. Lyul Min. A Worst Case Timing Analysis Technique for Multiple Issue Machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, 1998.

[14] B. M. E. Moret. Decision Trees and Diagrams. *Computing Surveys*, 14(4), 1982.

[15] Motorola. *Coldfire Microprocessor Family Programmer's Reference Manual*, 1997.

[16] S. K. Murthy. Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. *Data Mining and Knowledge Discovery*, 2(4), January 1998.

[17] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1, 1989.

[18] N. Ramsey and M. Fernandez. The New Jersey Machine-Code Toolkit. In *Usenix Technical Conference*, New Orleans, USA, 1995.

[19] N. Ramsey and M. Fernandez. *The New Jersey Machine-Code Toolkit, Reference Manual*, 1996.

[20] S. Russell and P. Norvig. *Artifitial Intelligence, A Modern Approach*. Prentice Hall, 1995.

[21] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Cheju Island, South Korea, 2000.

[22] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.