Weighted Decision Trees

Saumya Debray Sampath Kannan Mukul Paithane

Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{debray, kannan, mukul}@cs.arizona.edu

Abstract: While decision tree compilation is a promising way to carry out guard tests efficiently, the methods given in the literature do not take into account either the execution characteristics of the program or the machine-level tradeoffs between different ways to implement branches. These methods therefore offer little or no guidance for the implementor with regard to how decision trees are to be realized on a particular machine. In this paper, we describe an approach that takes execution frequencies of different program branches, as well as the costs of alternative branch realizations, to generate decision trees. Experiments indicate that the performance of our approach is uniformly better than that of plausible alternative schemes.

1 Introduction

There has been a great deal of research, in recent years, on the design and implementation of concurrent logic and constraint programming languages (see, for example, [12, 13, 14, 15, 17]). Much of the implementation effort in this context has focussed on the so-called "flat" versions of these languages: here, a procedure definition consists of alternatives, each alternative preceded by a *guard* that consists of a set of *ask actions* or *primitive tests*. An alternative can be selected at runtime only if the corresponding guard tests can be satisfied. For such languages, a compilation technique called *decision tree compilation* seems quite promising [6, 7, 8]. The idea here is to improve program efficiency by structuring the collection of all guard tests for a procedure into a "decision tree", thereby reducing the number of redundant tests executed. Algorithms for decision tree compilation have been given by Kliger and Shapiro [6, 7] and Korsloot and Tick [8].

The algorithms given by the authors cited above are concerned primarily with generating a decision tree for a set of tests by choosing an order in which the tests should be executed. They use various heuristics to accomplish this, e.g., by first considering tests that are "cared about" by the largest number of clauses (the *max-care* heuristic), then choosing from such tests one that has the fewest different results (the *min-variability* heuristic). These algorithms generate "conceptually reasonable" decision trees. However, as far as we can see, these algorithms give little or no guidance towards the actual *machine realization* of a decision tree, i.e., the actual structure and nature of branch instructions that should be generated at the machine level for a particular program in a particular implementation, assuming that certain characteristics of the program and the machine are given. There are a number of reasons why this problem is not entirely trivial:

1. Procedures are typically defined by more than one clause, and not all clauses are executed with equal frequency. For example, since programs typically spend most of their execution time in loops, the recursive clauses for a procedure are likely to be executed much more often than the non-recursive clauses that terminate recursion. If the different clauses for a procedure have differ-

ent "weights", or execution frequencies, then the decision tree should be constructed in such a way that the "heavier" a clause, i.e., the more frequently it is executed, the shorter the path from the root of the decision tree to the node corresponding to a decision on that clause. (Note that this notion of the "weight" of a clause refers only to its frequency of execution, which determines its importance in the context of decision tree compilation—it has nothing to do with its "granularity", i.e., computational cost.) Moreover, a good decision tree compilation algorithm should be robust with respect to program transformations such as *loop unrolling* or *partial evaluation*, which can change control flow characteristics and affect the relative execution frequencies of different branches.

2. It is not enough to consider the weights of different clauses in isolation when generating decision trees. For example, one could imagine a compilation scheme where a partial decision tree is generated considering the tests for the heaviest clause first, after which any remaining tests for the next heaviest clause are "grafted" onto this tree, and so on down the other clauses. Such an approach can give surprisingly poor performance, because a set of clauses may have weights that are not individually very large, but are collectively much heavier than the clause with heaviest weight (we discovered this the hard way while experimenting with decision trees for the lexical analysis phase of a compiler).

3. Conditional jumps can be implemented in a variety of ways using different addressing modes, and the alternatives have different capabilities and different costs. For example, a multi-way jump can be implemented using a tree of 2-way conditional branches, or by an indirect jump through a "jump table". The former is cheaper—typically, one or two machine instructions—but is limited to two alternatives; the latter is more expensive—typically, six to ten machine instructions—but can address many different alternatives. Unless the realization choices are made intelligently, the machine-level overheads may reduce considerably, or even nullify, the benefits of using decision trees.

This entire discussion is predicated on being able to associate execution frequencies (or, when normalized, estimated execution probabilities) with the clauses defining a procedure. For a discussion of this issue in the context of compilers for traditional languages, see [2, 9, 10, 18, 19]; techniques for estimating execution frequencies of logic programs from their call graph structure are discussed in [3, 16]. A point to note is that the techniques described in [3, 16] involve a simple and efficient linear-time traversal of the call graph of the program (i.e., a graph describing the caller-callee relationships between predicates): there is no iterative fixpoint computation of the sort encountered in global dataflow analyses. Thus, the overhead of estimating execution frequencies using such techniques is small. An alternative is to profile the program on sample inputs to estimate execution frequencies: as the results of Gorlick and Kesselman [5] indicate, the overhead for this approach is also small.

The primary technical contribution of this paper is to give an algorithm to construct "weighted" decision trees. The idea is to reduce the expected machine-level cost of executing the decision tree by taking into account estimated execution probabilities of the different clauses of a procedure, together with the execution costs of alternative machine realizations for (conditional) branches. We argue that in general, it is not enough to consider only the probabilities given, but that a related information-theoretic notion of *entropy* can be used to advantage. One interesting—and apparently novel—aspect of our algorithm is that in the process of generating a "good" decision tree for a procedure, it may generate tests that do not appear in the original source program, but which can be used to improve the execution characteristics of the decision tree. Experiments indicate that in most cases, the decision trees so generated correspond very closely to what one would desire for the particular weight distributions and machine instruction costs.

2 Preliminaries

2.1 Normalized Programs

To simplify the discussion that follows, we assume that programs are in a normalized form satisfying the following two properties:

1. the guards for any procedure are *exhaustive*, i.e., for any possible values of actual parameters to that procedure, there is at least one guard that will not fail; and
2. all ask actions are of the form '$f(v_1, \ldots, v_n)$ **op** c', where **op** is a comparison operator, $f(\cdots)$ is an n-ary "evaluable function", and c is a constant over a totally ordered domain.

These requirements may appear to be very restrictive, but it turns out that programs can be transformed to satisfy these conditions fairly easily. First, consider a procedure p whose guards are $\{G_1, \ldots, G_n\}$: such a procedure can be made exhaustive simply by adding "default" clauses that catches any input that causes each of the guards $G_1, \ldots, G_n$ to fail: the guards for these default clauses is obtained by transforming the formula $\neg G_1 \wedge \cdots \wedge \neg G_n$ to disjunctive normal form. The details are fairly obvious, and not pursued here further. An important point to note is that there is no need to determine whether the guards $G_1, \ldots, G_n$ in the original definition are already exhaustive: the "default" clause(s) can be added blindly without affecting the behavior of the program in any way. If $G_1, \ldots, G_n$ are exhaustive, then the guard of each default clause so generated is unsatisfiable, so the transformation does not affect program semantics.

Next, consider transforming ask actions to normal form: we assume that the language under consideration allows (only) the following kinds of ask actions:

Relational Tests on Values : Given a test of the form '$expr_1$ **op** $expr_2$' where **op** is a comparison operator,[1] let $expr_1 \equiv expr'_1 + c_1$ and $expr_2 \equiv expr'_2 + c_2$, then the normal form test is '$(expr'_1 - expr'_2)$ **op** $(c_2 - c_1)$.' E.g., the test '$X > Y + 5$' becomes transformed to the normal form test '$X - Y > 5$'.

Tests on Types : We assume that the language has a finite set of base types $\tau_1, \ldots, \tau_n$, with corresponding type tests $\texttt{is\_}\tau_1, \ldots, \texttt{is\_}\tau_n$. In the transformation to normal form, the extraction and checking of type tags is made explicit via an operation $tag(e)$ that returns the tag bits of the value of the expression e. Assume that the tag bit patterns for the types $\tau_1, \ldots, \tau_n$ are $\kappa_1, \ldots, \kappa_n$ respectively. Then, a type test '$\texttt{is\_}\tau_i(expr)$' is transformed to a normal form test '$tag(expr) = \kappa_i$'.

[1] We assume that there is some reasonable set of operators can be allowed in the expressions $expr_1$ and $expr_2$, e.g., the usual arithmetic operators, selectors for extracting components of compound structures and aggregates, etc.

2.2 Definitions and Notation

The techniques in this paper for generating decision trees rely heavily on our ability to "decompose" the set of primitive tests in a procedure definition into subsets of tests where tests in any subset are "independent" of the other tests. To formalize this notion we need the following definitions. An *outcome* of a primitive test is the result of the test for a particular assignment of values to the variables in the test. The set of possible outcomes of a test t is denoted by $\mathsf{outcomes}(t)$. The idea can be extended to a set of tests S: assume an arbitrary (but fixed) ordering for the elements of S, then the set of possible outcomes for S is denoted by $\mathsf{outcomes}(S)$, where an element $\sigma \in \mathsf{outcomes}(S)$ is a tuple $\langle \sigma_1, \ldots \sigma_{|S|} \rangle$ where σ_i represents the outcome of the i^{th} test in S.

Definition 2.1 An outcome $\sigma = \langle \sigma_1, \ldots \sigma_{|S|} \rangle \in \mathsf{outcomes}(S)$ is *consistent* if there is some substitution of values for the variables of S that makes the outcome of the i^{th} test in S equal σ_i for every i, $1 \leq i \leq |S|$. The set of all consistent outcomes of a set S of primitive tests is denoted $\mathsf{outcomes}^*(S)$. ▮

Definition 2.2 Given a set of tests U, $S \subseteq U$ is an equivalence class if S is minimal with respect to the property that for every σ in $\mathsf{outcomes}^*(S)$ and τ in $\mathsf{outcomes}^*(U - S)$, there exists a valuation θ of the variables in U such that $\theta(S)$ has outcome σ and $\theta(U - S)$ has outcome τ. ▮

Although it is not immediately obvious, it can be shown that the classes defined above are indeed equivalence classes and induce a partition on the set of tests. A point to note is that this generalizes the intuitive notion of a pair of tests being (in)dependent: according to this notion, we can only talk of the dependence of a *set* of tests, which means that the outcome of one of them provides some information about the possible outcomes of the others (this is roughly analogous to the notion of a set of vectors being linearly (in)dependent).

Example 2.1 Consider the following clause:

```
p(X, Y) :-  X < 0, X > Y | ...
```

Previous authors have considered the notion of a clause "caring" about a test (e.g., see [7]): a clause C cares about a test g if there is a test g' in the guard of C such that exactly one of the tests $g' \wedge g$, $g' \wedge \neg g$ is satisfiable. By this definition, the clause given above does not care about the test $\mathtt{Y} > \mathtt{0}$. However it is clear that the guard tests '$\mathtt{X} < \mathtt{0},\ \mathtt{X} > \mathtt{Y}$' cannot be satisfied if $\mathtt{Y} > \mathtt{0}$ is true. In our notion, the three tests $\{\mathtt{X} < \mathtt{0},\ \mathtt{X} > \mathtt{Y},\ \mathtt{Y} > \mathtt{0}\}$ would be in the same equivalence class, since it is the minimal set of tests in this case that satisfies the definition of an equivalence class above (no proper subset of this set satisfies the definition). □

As this example illustrates, the notion of an equivalence class differs from the notion of "cares about" in that we consider all possible outcomes of the tests in an equivalence class, not just the outcomes where the guard tests are true. The intutive justification for this is that we get valuable information not only from finding out that certain guard tests hold, but also from finding out that certain guard tests do not hold.

The algorithmic problem of breaking up a set of primitive tests into equivalence classes is in general rather complex. A sophisticated algorithm would analyze the relations (if any) between the variables mentioned in the primitive tests and take these relations into account in deciding equivalence classes. It is not hard to prove that the general equivalence class finding problem is NP-Complete. In practice, however, a good heuristic is to put two tests in the same equivalence class if the tests both involve a common variable. Algorithms for finding equivalence classes are not the main focus of this paper, since in most examples that we have encountered, this is far easier than the other tasks involved in finding the optimal decision tree.

In the next section we describe the algorithm to find the optimal decision tree. At a very high level, the algorithm breaks up the problem of sequencing the primitive tests in a procedure definition into hierarchical problems of sequencing the various equivalence classes of queries and sequencing the queries within an equivalence class. We show that there is no loss of optimality in this hierarchical breakup and that the sequencing between equivalence classes is independent of the weights on the various possible actions. The weights only affect the sequencing of tests within each equivalence class.

In order to describe our heuristic for ordering the tests within an equivalence class we borrow the notion of *entropy* (also known as the *uncertainty function*) from information theory[1]:

Definition 2.3 Let X be a random variable that takes on a finite number of possible values $x_1, x_2, \ldots, x_m$ with probabilities $p_1, p_2, \ldots, p_m$, respectively, such that $p_i > 0, 1 \leq i \leq m$, and $\sum_{i=1}^{m} p_i = 1$. The *entropy* of X, denoted $H(X)$, is defined to be $\sum_{i=1}^{m} -p_i \log_2(p_i)$. ▮

At first glance the choice of this particular function seems somewhat arbitrary, but it can be shown that this is the *only* function that satisfies some very reasonable axioms on the behaviour of an uncertainty function (see [1] for details). The notion of entropy extends in a straightforward way to tests: if a test t has m possible outcomes, with probabilities $p_1, \ldots, p_m$ respectively, then the entropy of t is $H(t) = \sum_{i=1}^{m} -p_i \log_2(p_i)$. The underlying idea here is that execution frequencies for different clauses (i.e., guards) can be normalized to give us estimates of execution probabilities for the guard tests, whence we can use entropies to guide the generation of decision trees.

Intuitively, the way to think of entropies in our situation is that, when we enter a procedure definition, the average amount of uncertainty we have to dispel before choosing the clause to execute is represented by the entropy. Each test that we perform dispels a certain amount of uncertainty based on the probabilities of each of the outcomes of that test. The relevant property of entropy here is that, given an initial entropy e_1, if we perform a test with etropy e_2, then the average amount of uncertainty that remains given the outcome of the test is $e_1 - e_2$. Hence, tests that dispel a greater amount of uncertainty make greater progress towards our goal. Another feature that makes this approach very attractive is that the entropy function (and a normalized version of it) is especially useful for comparing tests that take differing numbers of instructions to perform (i.e., have different costs) and have different numbers of outcomes.[2] For instance we can use a normalized entropy function to compare a binary decision, as exemplified by an **if** statement,

[2] Initially, we looked for more elementary ways to solve what we hoped would be a simple compilation problem. However, we were unable, after considerable thought, to come up with a

with a multiway decision, as exemplified by a **case** or **switch** statement. To this end we define the normalized entropy of a test as follows:

Definition 2.4 The *normalized entropy* of a test t with entropy $H(t)$ and cost C is given by $\widehat{H}(t) = H(t)/C$. ▮

We defer a discussion of the use of this definition to the next section. In the next section we will see the application of entropy (or uncertainty) to ordering tests within an equivalence class.

3 Generating Weighted Decision Trees

3.1 The Mutually Exclusive Case

Recall that a set of guards is exhaustive if any consistent outcome of the primitive tests comprising the guards turns on *at least* one of the guards. In practice most procedure definitions satisfy a further property which we call *mutual exclusion.*

Definition 3.1 A pair of tests t_1 and t_2 is *mutually exclusive* if and only if $\exists(t_1 \wedge t_2)$ is not satisfiable.

A set of tests is mutually exclusive if the tests are pairwise mutually exclusive. ▮

In this section we will focus on procedure definitions where the set of guards are exhaustive as well as mutually exclusive — i.e. every consistent outcome of the primitive tests turns on *exactly one* of the guards.

For such procedure definitions we can rigorously establish the 'form' of the (provably) optimal decision tree. Our results in this section will apply to arbitrary procedure definitions that are mutually exclusive and exhaustive (note that that the general problem of generating an optimal decision tree where the tests may not be mutually exclusive and exhaustive is NP-Complete [4]). We can show that in any procedure definition where the guards are mutually exclusive and exhaustive, there is a single equivalence class such that each guard "cares about" this class. In other words, the outcome of the tests in the equivalence class *must* be determined before we can decide which guard is turned on.

Definition 3.2 An equivalence class is said to be *dominant* if the outcome of the tests in the equivalence class must be determined before we can decide if any of the guards is true. ▮

Theorem 3.1 *In any procedure definition where the guards are exhaustive and mutually exclusive, there is a dominant equivalence class of tests.* ■

The proof is omitted due to space constraints. This result immediately suggests an optimal algorithm for generating a decision tree in the case where the procedure definition is mutually exclusive and exhaustive:

1. Find a dominant equivalence class.

reasonable approach using only execution weights that would be able to compare the relative costs of two-way branches using conditional branches and multi-way branches using a branch table.

2. Produce a decision tree for the equivalence class (along the lines of Figure 1), and recursively construct trees for the subproblems at each of the leaves of this tree.

The optimality of the algorithm follows from the fact that the outcome of tests in the dominant equivalence class must be determined by *any* scheme to evaluate the procedure definition.

The central part of the above algorithm is to produce an optimal decision tree for an equivalence class. This is the subject of our next subsection.

3.2 Generating the Decision Tree for an Equivalence Class

In this section we present a heuristic for finding a near-optimal decision tree for an equivalence class using the notion of normalized entropy defined in the previous section. This is the portion of our general algorithm for mutually exclusive and exhaustive procedure definitions that is not necessarily optimal. Recall that each test is assumed to be of the form '$f(\bar{x})$ **op** c', where f is an evaluable function and c is a constant over a totally ordered domain. Given an equivalence class of tests S for which to generate a decision tree, we first group the elements of S into partitions, called *families*, such that tests in the same family compute the same "left hand side" expression $f(\bar{x})$. For example, given the tests

{`I > 0, J > 0, I-J < 0, I-J = 0, I-J >= 1`}

we get three families: {`I > 0`}, {`J > 0`}, and {`I-J < 0, I-J = 0, I-J >= 1`}. To generate the decision tree for the original equivalence class, we have to construct the decision tree for each family so generated. As discussed at the end of the previous section, at each point we construct a decision tree for a dominant equivalence class of tests. There may be semantic dependencies that impose an ordering, e.g., it may be necessary to test that a variable is bound to a cons cell before attempting to access the head of that cell: if there are such dependencies, we assume that the different families are ordered in a way that respects these dependencies and yields a legal ordering. Our choice of an order for processing these families may also be guided by low-level considerations, e.g., we may choose an order that groups together different families that test the same variables, so as to improve our use of registers and better exploit the cache. For all these reasons, we do not focus on the ordering between families in this paper, although this ordering has a bearing on the average number of instructions executed.

According to earlier treatments of decision tree compilation, the next step, namely the construction of a decision tree for a family of tests, which is of the form $\{f(\bar{x})\ \mathbf{op}_1\ c_1, \ldots, f(\bar{x})\ \mathbf{op}_n\ c_n\}$, is trivial: we generate a multi-way jump based on the value of the expression $f(\bar{x})$. This does not address the crucial implementation decision of how this multi-way branch is to be realized. Depending on the addressing modes available on our target architecture, there may be a variety of options available, with different capabilities and costs: for example, we may use a tree of conditional branches (corresponding to `if-then-else` statements), or an indirect jump through a branch table (corresponding to a `case` or `switch` statement), or possibly a combination of both. In general, each option has different capabilities and different costs: for example, a conditional branch takes two or three machine instructions but is able to address only two alternatives, while an indirect jump through a branch table may take a total of six to ten machine instructions, but can address a large number of alternatives. Further, even if we decide to use a conditional test rather than jump through a branch table, we still have to make the choice of what that test should be. Typically, the best choice will be a test that tries to balance, as far as possible, the weights corresponding to each of its outcomes: this may produce a test that does not appear in the original source program. One of the novel features of our algorithm is that it (when appropriate) generates tests which do not occur in the source program, resulting in improved performance.

Our aim is to generate a decision tree that reduces, as far as possible, the expected length (in machine instructions) over all paths. To do this, we use normalized entropies (see Definition 2.4) to compare the "merit" of alternative realizations, and pick the best.[3]. The justification for using normalized entropy is as follows: What we would like to do is to minimize the average path length which is the weighted average of the number of instructions it takes to get to each leaf of the tree. On the average, we need to dispel an amount of uncertainty equal to the entropy of the probability distribution induced by the weights on the leaves before we can get to the leaves. In order to find the way that takes the fewest number of instructions to dispel this uncertainty, we use the greedy heuristic and pick the test that dispels the greatest amount of entropy per instruction. Of course, this is only a heuristic and we can construct somewhat pathological examples where it is not optimal. Our algorithm is described below and the decision trees produced by our algorithm for some examples are described in the next section. To simplify the discussion that follows, we assume that there are only two alternative realizations possible: conditional jumps, with cost C_{branch}, and indirect jumps through a branch table, with cost C_{switch}: the algorithm can be extended to deal with other realizations (e.g., where a set of tests is realized using a `switch` after "lopping off" the boundaries using two `if-then-else` statements) without much trouble. We use the following notation:

- the probability (i.e., normalized weight) of a test t is denoted by $prob(t)$;
- because not every set of tests can be realized using a branch table (for example, if there are tests of the form `x > 0`), we assume that there is a predicate $switchable(S)$ that is true if and only if the set of tests S can be implemented using a branch table; and
- given a family of tests S, we use the notation '$\langle S_1, c, S_2\rangle = split(S)$' to indicate that
  - (i) S is partitioned into two pieces S_1 and S_2 such that the total weight of the tests in S_1 is as close as possible to the total weight of tests in S_2; and
  - (ii) c is the "dividing line" between the tests S_1 and S_2, i.e., tests in S_1 imply that the expression being evaluated has a value less than c, while tests in S_2 imply that this value is greater than (or equal to) c.

The algorithm, which is given in Figure 1, can be extended in a straightforward way to consider more than two alternative realizations. In the function *gen_tree*, it is

[3] If one were only interested in finding the optimal binary decision tree for the example above, techniques for generating optimal binary search trees using dynamic programming would apply, but these techniques do not permit an easy comparison of this tree with a decision tree using multiway branches.

Input : A set of tests S forming an equivalence class.

Output : A decision tree T realizing the tests S.

Method : **return** $T := \mathit{gen\_tree}(S)$;

function $\mathit{gen\_tree}(S)$ **: decision_tree**
begin
  normalize the weights of tests in S;
  partition S into families;
  arrange these families in some order $\{S_1, \ldots, S_n\}$;
  for $i := 1$ **to** n **do**   /* construct decision trees for each family */
    if $\mathit{switchable}(S_i)$ **and** $\mathit{entropy\_switch}(S_i) > \mathit{entropy\_cond}(S_i)$ **then**
      `gen_switch`(S_i);
    else
      `gen_cond`(S_i);
    fi;
  od
end;

procedure $\mathit{gen\_switch}(S)$
begin
  implement the tests in S at n as an indirect jump through a jump table;
end

procedure $\mathit{gen\_cond}(S)$
begin
  let S be a family of tests $\{\mathcal{E}(\bar{x})\ \mathbf{op}_1\ c_1, \ldots, \mathcal{E}(\bar{x})\ \mathbf{op}_n\ c_n\}$;
  let $\langle S_1, c, S_2 \rangle = \mathit{split}(S)$;
  let $p_1 = \sum\{\mathit{prob}(t) \mid t \in S_1\}$ and $p_2 = \sum\{\mathit{prob}(t) \mid t \in S_2\}$;
  generate the decision tree "**if** $\mathcal{E}(\bar{x}) < c$ **goto** U_1 **else goto** U_2;"
      where $U_1 = \mathit{gen\_tree}(S_1)$ **and** $U_2 = \mathit{gen\_tree}(S_2)$;
end

function $\mathit{entropy\_switch}(S)$: **real**
begin
  return $(\sum\{-\mathit{prob}(t)\log_2(\mathit{prob}(t)) \mid t \in S\})/C_{switch}$;
end

function $\mathit{entropy\_cond}(S)$: **real**
begin
  let $\langle S_1, c, S_2 \rangle = \mathit{split}(S)$;
  let $p_1 = \sum\{\mathit{prob}(t) \mid t \in S_1\}$ and $p_2 = \sum\{\mathit{prob}(t) \mid t \in S_2\}$;
  return $-(p_1 \log_2(p_1) + p_2 \log_2(p_2))/C_{branch}$;
end

Figure 1: An Algorithm for Ordering Tests Within an Equivalence Class

important that the weights be normalized before proceeding with the construction: otherwise, in subsequent invocations of *gen_tree* from within *gen_cond*, the computations of weighted entropies may become distorted. Note that the procedure *gen_cond* can introduce tests into the decision tree that are not present in the original source program. Given the treatment of type tests such as `integer`/1, `atom`/1, etc., described in Section 2.1, an esthetically pleasant consequence of this is that a set of type tests on a variable may compile into decision tree tests with non-equality comparisons on type tags, e.g., something like '`if` *tag*(`X`) < `LIST` ...'

Example 3.1 The following example illustrates the working of the algorithm of Figure 1. Let the cost of an indirect branch through a jump table be 10 instructions, while that of a test/conditional-branch combination is 2 instructions (these are the assembly instruction counts for `switch` and `if` statements in C on Sparcstations). Consider the predicate `p`/1 defined by 100 clauses:

```
p(X) :- X = 1   | true.
   ...
p(X) :- X = 100 | true.
```

Suppose that the weights of the clauses, for different values of the argument `X`, are given by the following table (the distribution is somewhat artificial, but it illustrates the algorithm in a simple way and produces a pretty decision tree):

X	*weight*	*normalized wt.*
1	520	0.5200
2–49	3	0.0030
50	236	0.2360
51–100	2	0.0020

We first consider the root node of the decision tree. The weighted entropy $\widehat{H}_{jt}$ for a jump table implementation of this node is given by

$$\begin{aligned}\widehat{H}_{jt} &= \tfrac{1}{10}((-0.52\log_2 0.52) + \textstyle\sum_{i=2}^{49} -0.003\log_2 0.003 + (-0.236\log_2 0.236) + \\ &\qquad \textstyle\sum_{i=51}^{100} -0.002\log_2 0.002) \\ &= 0.308.\end{aligned}$$

For a conditional branch implementation, the "split point" that balances the normalized weights best, given the distribution given above, is 2 (i.e., the test generated will be '`X < 2`'). The weighted entropy $\widehat{H}_{cb}$ for a conditional branch implementation is given by

$$\widehat{H}_{cb} = \tfrac{1}{2}((-0.52\log_2 0.52) + (-0.48\log_2 0.48)) = 0.499.$$

Since $\widehat{H}_{cb} > \widehat{H}_{jt}$, a conditional branch '`if (X < 2) ...`' is used to implement this node.

One of the children of this node is the node `1`, which is a leaf node that does not need a decision tree. The other child requires a decision tree for the cases `2`–`100`. For this, the recursive call to the function *gen_tree* results in a renormalization of the relevant weights, which produces the following:

X	*weight*	*normalized wt.*
2–49	3	0.0063
50	236	0.4917
51-100	2	0.0042

Computing weighted entropies as above, with the split point for the conditional branch case being at `50`, we get $\widehat{H}_{jt} = 0.437$, $\widehat{H}_{cb} = 0.442$. Since $\widehat{H}_{cb} > \widehat{H}_{jt}$, a conditional branch '`if (X < 50) ...`' is used to implement this node.

One of the children of this node is for the cases `2–49`. Each of these cases has a normalized weight of 0.0208. With the split point for the conditional branch at `25`, the weighted entropies are computed as $\widehat{H}_{jt} = 0.558$, $\widehat{H}_{cb} = 0.500$. Since $\widehat{H}_{jt} > \widehat{H}_{cb}$, this subtree is implemented using a jump table.

The other child is for the cases `50–100`. On normalization, we have

X	*weight*	*normalized wt.*
50	236	0.7024
51-100	2	0.0059

In this case, with the split point for the conditional branch at `51`, the weighted entropies are computed as $\widehat{H}_{jt} = 0.254$, $\widehat{H}_{cb} = 0.439$. Since $\widehat{H}_{cb} > \widehat{H}_{jt}$, a conditional branch '`if (X < 51) ...`' is used to implement this node.

One child of this node is the leaf node `50`, which does not need a decision tree. The other child is for the cases `51–100`, for which each test has a normalized weight of 0.02. In this case, with the split point for the conditional branch at `25`, the weighted entropies are computed as $\widehat{H}_{jt} = 0.564$, $\widehat{H}_{cb} = 0.500$. Since $\widehat{H}_{jt} > \widehat{H}_{cb}$, this subtree is implemented using a jump table.

The overall decision tree that is produced for this example is shown in Figure 2. The average number of instructions executed for this tree, given the weight distribution and implementation costs assumed above, is 5.78. By comparison, the average cost is 10 instructions if the decision tree is implemented as a single `switch` statement, and between 12 and 14 instructions (depending on the exact structure of the tree) if it is implemented as a binary tree without taking weights into account. (A cursory examination of the tree in Figure 2 suggests that it may be better, given the weight of the leaf labelled `50`, to test for this case earlier, e.g., using the test '`X = 50`' immediately after the test '`X` $<$ `2`'. However, a careful examination indicates that the average number of instructions executed for such a tree would be 5.94, which is slightly higher than that of the tree obtained using our algorithm.)

To simplify the discussion in this example, we have ignored the possibility of suspension due to underinstantiated inputs. To deal with suspension, it suffices to add a clause that specifies when suspension should occur:

```
p(X) :- tag(X) = VARIABLE | suspend(...).
```

The weight of such a "suspension clause" will depend on the execution characteristics of the program. For example, if `p`/1 is almost always called with a non-variable argument, and therefore rarely suspends, then the suspension clause will have a very small weight, and the corresponding node in the decision tree generated using our approach will be fairly deep, i.e., it will be considered towards the end. On the

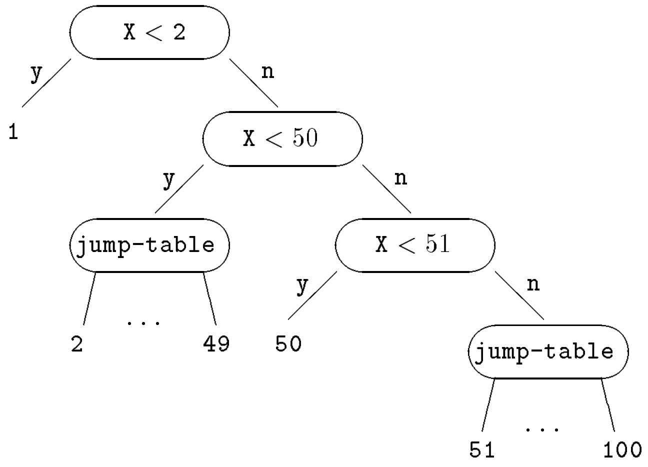

Figure 2: The decision tree produced for Example 3.1

other hand, if p/1 is usually called with a variable argument and has to suspend (as might happen in programs written in an "object-oriented" style), then the suspension clause will have a high weight and its node in the decision tree will be close to the root, i.e., it will be considered early in the execution of the predicate. As far as we can tell, earlier approaches, e.g., those of Kliger and Shapiro [6, 7], generate decision trees that consider suspension in *otherwise* branches, which appear to be considered at the end if none of the non-*otherwise* branches is taken, and therefore do not offer this flexibility. □

3.3 The Non-Mutually Exclusive Case

In this case, at any point there is a *set* of equivalence classes of tests, each of which is "cared about" by some subset of the set of clauses under consideration, rather than a single dominant equivalence class that every clause cares about. Theoretically, the notion of entropies seems less obviously applicable here, because the tests are not mutually exclusive. However, it turns out that we get intuitively reasonable results if we use weights or normalized entropies to order the different equivalence classes, then apply the previous algorithm to the equivalence classes in this order.

4 Performance

In this section, we compare the performance of the entropy based technique described earlier with those of a number of other plausible ways of implementing decision tree for an equivalence class. Our experiments considered an equivalence class consisting of a single multiway branch, which corresponds to several tests on the same group of variables. Our decision tree compiler takes a (switchable) set of tests with weights, together with machine cost parameters, and emits C code for these tests. The results reported are execution times for the code so generated, compiled using gcc on a Sparcstation-2: this allows us to examine the relative machine-level costs of different realizations of decision trees without obscuring the results by including time spent in non-decision-tree computations. The results are given in Table 1. The different approaches that we compare with our entropy-based approach are as follows:

If-Then-Else : Here, the n-way branch is implemented as a series of *if-then-else* statements. As a result, the last branch is executed only after $n-1$ tests have been performed. (This is not quite the same as not compiling a decision tree at all, since it is possible, in such a scheme, that tests from different guards are shared.)

Weighted If-Then-Else : Similar to the above, except that the tests are ordered in decreasing order of weight, with the branch with the highest weight tested for first.

Weighted Binary Tree : When the underlying set of values is totally ordered, it is possible to organize a set of tests so that we effectively use a binary search tree. The tests at the leaves of the tree are the tests that appeared originally in the program and the tests on the internal nodes are the ones that are inserted such that the probability of execution of either branch is as equal as can be made, depending on the probability values of the original program branches. Unlike the decision tree compilation schemes suggested in the literature, this scheme can generate (internal node) tests that do not appear in the original program.

Jump Table : The most obvious way of coding an n-way branch is using an indirect jump through a jump table. However, this approach is not suitable for non-equality tests, e.g., `x > 2`.

The benchmarks tested were the following:

1. Lexical Analyzer: In a compiler front-end, a lexical analyzer must examine each character of the input program to determine the lexical structure of the program. This requires a decision tree with an n-way branch, where n is the size of the alphabet. We restricted our alphabet to digits and lower case letters, so that the decision tree had 36 leaves. Letters were given heavier weight than digits (each letter had a weight of 10, and each digit a weight of 1). We used a 2 Mbyte text file as test input for our experiments. The decision tree produced by the entropy-based scheme in this case was a binary tree.

2. Final Code Generator: After all final code generation decisions have been made in the back end of a compiler, it is necessary to actually emit the instructions to create an object file. For this, the compiler must examine the opcode of each instruction (which is typically in some internal representation) to determine the exact bit patterns to emit. Thus, it is necessary to create a decision tree based on the relative (static) frequencies of different opcodes. We used `gcc` to compile itself on a Sparcstation and generate an assembler file, then used the static instruction counts obtained from this to estimate the relative frequency of different opcodes. The decision tree in this case had 53 leaves. The decision tree produced by our entropy-based approach was an indirect jump through a branch table. The time reported is the time taken to process the `gcc` opcodes.

Approach	*Lexical Analyser*	*Code Generator*	*Byte-Code Interpreter*
entropy-based	1.000	1.000	1.000
jump table	1.114	1.000	1.000
binary tree	1.114	1.347	1.470
if-then-else	2.770	1.732	> 5
weighted-if-else	1.033	1.732	2.625

Table 1: Normalized Performance Figures

3. Byte-Code Interpreter: Many programming language implementations use byte code interpreters, where programs are compiled to (a byte-code encoding of) a virtual machine instruction set, which is then interpreted by a machine-level program. Many well-known Prolog implementations follow this approach. Such an interpreter requires a decision tree on byte-code instruction opcodes. While the inner loop of such interpreters is typically implemented as an indirect branch through a jump table, it is not obvious that this is necessarily the best implementation, since this fails to take into account the relative (dynamic) frequencies of different opcodes. For our experiments, we instrumented SB-Prolog to obtain dynamic opcode traces for a number of medium-sized Prolog programs (e.g., `boyer`, the SB-Prolog compiler, the Berkeley PLM compiler, a dataflow analyser for Prolog, etc.), then used the opcode frequencies so obtained to measure the time taken by different decision tree realizations to process the traces so obtained. In this case, the decision tree had 91 leaves, and the particular byte-code encodings used, the dynamic opcode distribution, and the relative machine level costs assumed caused the entropy-based method to generate an indirect branch through a jump table.

5 Conclusions

While decision tree compilation is a promising way to carry out guard tests efficiently, the methods given in the literature do not take into account either the execution characteristics of the program or the machine-level tradeoffs between different ways to implement branches. These methods therefore offer little or no guidance for the implementor with regard to how decision trees are to be realized on a particular machine. In this paper, we describe an approach that takes execution frequencies of different program branches, as well as the costs of alternative branch realizations, to generate decision trees. Experiments indicate that the performance of our approach is uniformly better than that of other plausible alternatives.

Acknowledgements: Comments by Evan Tick and the anonymous referees helped improve the presentation of the paper. The work of the first and third authors was supported in part by the National Science Foundation under grant number CCR-8901283; that of the second author was supported by the National Science Foundation under grant number CCR-9108969.

References

[1] R. B. Ash, *Information Theory*, Dover Publications, NY, 1965.

[2] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs", *Proc. 19th. ACM Symp. on Principles of Programming Languages*, Albuquerque, NM, Jan.

1992, pp. 59–70.

[3] S. K. Debray, "A Remark on Tick's Algorithm for Compile-Time Granularity Analysis", *Logic Programming Newsletter* vol. 3 no. 1, July 1989.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[5] M. M. Gorlick and C. F. Kesselman, "Timing Prolog Programs Without Clocks", *Proc. Fourth IEEE Symp. Logic Programming*, San Francisco, CA, Sept. 1987, pp. 426-432. IEEE Press.

[6] S. Kliger and E. Shapiro, "A Decision Tree Compilation Algorithm for FCP(|, : , ?)", *Proc. Fifth Int. Conf. on Logic Programming*, Seattle, Aug. 1988, pp. 1315–1336. MIT Press.

[7] S. Kliger and E. Shapiro, "From Decision Trees to Decision Graphs", *Proc. NACLP-90*, Austin, Oct. 1990, pp. 97–116. MIT Press.

[8] M. Korsloot and E. Tick, "Compilation Techniques for Nondeterminate Flat Concurrent Logic Programming Languages", *Proc. Eighth Int. Conf. on Logic Programming*, Paris, June 1991, pp. 457–471. MIT Press.

[9] S. McFarling, "Program Optimization for Instruction Caches", *Proc. Third Int. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 183–191.

[10] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning", *Proc. SIGPLAN-90 Conf. on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 16–27.

[11] M. L. Powell, "A Portable Optimizing Compiler for Modula-2", *Proc. SIGPLAN-84 Symp. on Compiler Construction*, Montreal, June 1984, pp. 310–318. SIGPLAN Notices vol. 19 no. 6.

[12] V. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conf. on Logic Programming*, Austin, TX, Oct. 1990, pp. 431–446. MIT Press.

[13] E. Shapiro, "The Family of Concurrent Logic Programming Languages", *Computing Surveys*, vol. 21 no. 3, Sept. 1989, pp. 412-510.

[14] E. Shapiro (ed.), *Concurrent Prolog: Collected Papers*, MIT Press, 1987.

[15] S. Taylor, *Parallel Logic Programming Techniques*, Prentice Hall, 1989.

[16] E. Tick, "Compile-time Granularity Analysis for Parallel Logic Programming Languages", *Proc. Int. Conf. on Fifth Generation Computer Systems*, Tokyo, Japan, Nov. 1988, pp. 994-1000.

[17] K. Ueda, "Guarded Horn Clauses", in *Concurrent Prolog: Collected Papers*, vol. 1, ed. E. Shapiro, pp. 140-156, 1987. MIT Press.

[18] D. W. Wall, "Global Register Allocation at Link-time", *Proc. SIGPLAN-86 Conf. on Compiler Construction*, June 1986, pp. 264–275.

[19] D. W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles", *Proc. SIGPLAN-91 Conf. on Programming Language Design and Implementation*, June 1991, pp. 59–70.