Branch Strategies to Optimize Decision Trees for Wide-Issue Architectures

Patrick Carribault [123], Christophe Lemuet [1],
Jean-Thomas Acquaviva [1], Albert Cohen [2], and William Jalby [1]
Contact Author: William.Jalby@prism.uvsq.fr

[1] PR*i*SM, University of Versailles
[2] ALCHEMY group, INRIA Futurs, Orsay
[3] Bull Les Clayes-sous-Bois

Abstract. Branch predictors are associated with critical design issues for nowadays instruction greedy processors. We study two important domains where the optimization of decision trees — implemented through `switch-case` or nested `if-then-else` constructs — makes the precise modeling of these hardware mechanisms determining for performance: compute-intensive libraries with versioning and cloning, and high-performance interpreters. Against common belief, the complexity of recent microarchitectures does not necessarily hamper the design of accurate cost models, *in the special case of decision trees*. We build a simple model that illustrates the reasons for which decision tree performance is predictable.
Based on this model, we compare the most significant code generation strategies on the Itanium2 processor. We show that *no strategy dominates in all cases*, and although they used to be penalized by traditional superscalar processors, *indirect branches regain a lot of interest* in the context of predicated execution and delayed branches. We validate our study with an improvement from 15% to 40% over Intel ICC compiler for a Daxpy code focused on short vectors.

1 Introduction

Due to the increasing depth of pipelines and wider instruction fetch, branch prediction becomes more and more crucial. Consequently, large hardware structures supporting branch prediction and acceleration are ubiquitous on modern microarchitectures. To supply the pipeline with a continuous stream of instructions, branch predictors have to forecast the outcome of the branch — *taken* or *not taken* — and the target address. In control-intensive codes, multiple kinds of branches occur at a very high rate, hence wide issue processors need to issue and predict multiple branches every cycle.

On the compilation side, many optimization strategies have been proposed [1], to expose more instruction-level parallelism in branch-intensive codes, or to circumvent the limitations of the branch predictor and instruction fetch engine.

We study two important domains where decision trees are responsible for a significant part of the computation time: optimized libraries with versioning and cloning, and high-performance interpreters. Decision trees are implemented through `switch-case` or nested `if-then-else` constructs. Generic strategies in general-purpose compilers are often sub-optimal, because they are not well suited to the size of the decision trees (from tens up to hundreds of leaf nodes). In addition, optimizing decision trees is a problem due to the dependence on input data, but it should be emphasized that most of the complexity is brought by the large variety of code generation options.

.

Versioning and cloning. When optimizing high-performance libraries, decision trees are often produced by the specialization of loops with small iteration counts, e.g., vector operations (for the BLAS or FFT), memory copying routines (important for I/O routines), or sorting networks (QuickSort libraries, adaptive sort [5]): a standard strategy consists in generating specialized code for 1, 2,... up to *k* iterations, using a `switch` instruction to select the right version. Procedure specialization and cloning is also common practice when performing context-sensitive interprocedural optimizations or partial evaluation. The resulting decision trees are based on interval tests in general, implemented as nested `if`-`then`-`else` structures.

High-performance interpreters. Bytecode interpreters and processor emulators also rely on decision trees for their computation kernel [2]. These trees are generally implemented as `switch`-`case` structures spanning over a hundred cases. Such kernels consist of a loop iterating over a decision tree, and two orthogonal optimization directions exist: one may either minimize the latency of the decision tree itself (on average, in the worst case, based on frequency or cost of each leaf, etc.), or restructure the loop such that global correlations can be better exploited. The latter scheme is explored in great detail by Ertl and Gregg in [2]. However, this scheme performs poorly on large decision trees whose global history overflows the hardware prediction tables. Also, it may not fit the constraints (memory capacity, performance predictability) of embedded systems.

After introducing decision trees in Section 2, and our experimental platform, — the Itanium 2 —, in Section 3 we provide details on branch implementation. Section 4 is focused one *one-way* branches, while Section 5 depicts more advanced features of *multiway* branches. Section 6 is dedicated to indirect branch. Experimental results and in-depth comparisons are provided in Section 7, at last Section 8 concludes and gives several openings.

2 Domain-Specific Optimization for Decision Trees

2.1 Decision Tree Patterns at the Source Level

Two code patterns are dominant: the nested tree of `if`-`then`-`else` tree conditionals and the linear `switch`-`case` selector. Nested conditionals are often restructured at intermediate compilation steps, leading to linear sequences of `else if` branches, see Figure 1. These linear sequences are more general than `switch` selectors whose case-distinction is based on integer constants only. However the `switch` selector deserves a specific treatment because it enables additional optimizations and is pervasive in interpreters [2] and codes optimized through versioning or cloning.

In this generic form, the code fragments after each `else if` or `case` may contain further branches. We will only deal with *one-level* `else if` *structures* and *one-level* `switch`

```
 if (c0)
 | code0;
else if (c1)
 | code1;
 | ...
else if (ck-1)
 | codek-1;
else
 | codek;
```

```
switch (x) {
 | case const0:
 | | code0; break;
 | ...
 | case constk-1:
 | | codek-1; break;
 | default:
 | | codek;
}
```

Fig. 1. Code patterns

```
if (Y>=0 && Y<4)
| // code0
else if (Y>=4 && Y<=8)
| // code1
else if (Y>8 && Y<14)
| // code2
else
| // fall-through code
```

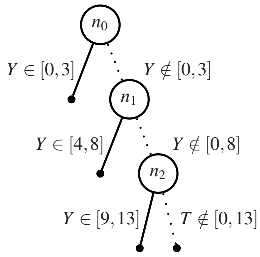

Fig. 2. Graphical representation of a decision tree (dotted lines correspond to fall-through)

structures, abstracting away the nested branches (i.e., handling them separately at each nesting depth). Notice complex nests of conditionals can be often linearized by control-flow restructuring passes.

For the sake of simplicity, expressions in `else if` conditions must be integer *interval checks* — strict or not. These checks neither have to be exclusive nor to involve the same variable throughout the one-level structure.

2.2 Representation of Decision Tree Implementations (Assembly Level)

For both `else if` and `switch` structures, we use a tree abstraction to study code generation strategies. Since we consider one-level structures, there is no a priori restriction on the shape of the tree: all leaves can be reorganized in any possible fashion — except when dependence constraints on non-exclusive conditions require the preservation of the original execution order. Notice that, in a more general setting, identical leaves may be reachable through separate branches; this is not a real problem since the following discussion could easily be extended to a directed acyclic graph.

- *Internal nodes* correspond to the evaluation of the condition associated with an `else if` or `case` construct. The latter pattern restrictions guarantee that `switch` conditions can be evaluated by a single operation (comparison to a constant) and that `else if` conditions require exactly two operations.
- *Leaves* represent the code that has to be executed after the correct branch decision is made; it may be arbitrarily complex, but we may have to make some assumptions about its execution time in several of the proposed optimization strategies.
- *Directed edges* are labeled with the condition of the branch or its negation. Edges have an additional attribute to tell whether they correspond to taken branches — solid lines — or fall-through — dotted lines. Notice dotted-lines correspond to the predicted case in general [10].

A one-level `else if` structure and its associated decision tree are shown in Figure 2. Such a figure is named a *one-way comb* in the remainder of the paper. In fact, a *comb* refers to any list of internal nodes, where the *one-way* or *multi-way* (introduced in next section) prefix is added to characterize the number of extra edges for every node in addition of the fall-through edge (in dotted line on the figure).

3 Description of the Target Architecture

Optimizing decision trees for the Itanium architecture is interesting because the platform provides the programmer or compiler with many opportunities to "drive the processor to the correct path". It is also representative of the modern mechanisms implemented in wide-issue architectures, including the IBM Power4 and Power5 (multiway branches), as well as embedded VLIW processors like the Philips TriMedia [9] and the HP Lx alias ST-Micro 2x0 series [3] (predication, delay slots). This section quickly surveys the Itanium2 branch mechanisms from a programmer perspective [4, 6, 8].

3.1 Itanium Instruction Set for Branches

First of all, traditional conditional branches are implemented as predicated branches. Typically, a compare instruction evaluates a relation and sets accordingly a pair of mutually exclusive 1-bit predicate registers. Any instruction guarded by a predicate register (called the qualifying predicate) is "nullified" if this predicate is false; thus, a predicated branch behaves exactly like a conditional branch.

Like in any pipelined processor, there is a delay between the issue cycle of the branch and the cycle when the qualifying predicate is known. If a misprediction occurs the associated penalty corresponds to the pipeline re-steer.

Branch instructions fields descriptions are given in [4]. One of these fields provides a guess on branch outcome (*taken* or *not taken*) either statically or dynamically. The others for example allow to determine how many instructions should be prefetched or to avoid the pollution of predictors tables with information relative to the branch. The ISA also supports indirect branch through the storage of target address within a set of dedicated branch registers.

In addition to this fairly complex branch instruction, the Itanium instruction set provides a specific instruction (`brp`) for controlling instruction prefetch.

The Itanium instructions are grouped by chunks of three instructions called *bundles*. A bundle can contain, one, two or three branch instructions. While up to two bundles can be issued every cycle, only three branches can be dispersed to branch functional units per cycle. Interestingly, branches do not have to be mutually exclusive: their relative ordering matters when resolving the correct path.

3.2 Itanium2 Branch Hardware

The branch prediction algorithm used in the Itanium2 is based on the two-level prediction scheme proposed by Yeh and Patt [10] combining local prediction (branch history tables) and global predictions (pattern history tables). On the Itanium2, a large amount of hardware resources have been dedicated to branch prediction mechanisms, for instance tables keeping track of several thousands of branches.

For minimizing the impact of instruction cache misses, the Itanium2 provides an instruction streaming engine which can be controlled by the compiler through branch hints. We will always assume a perfect locality in the L1 instruction cache in this paper (thanks to compact code or smart usage of `brp` instructions).

Overall branch penalties (assuming no instruction miss) are given below:

– 0 cycle for correctly predicted branch (both outcome and target);
– 1 cycle for branches for which the outcome has been correctly predicted but the predicted address is incorrect (making use of dedicated address calculation units);
– 6 cycles for branches whose outcome has been incorrectly predicted (the pipeline has to be drained).

Indirect (or delayed) branches behave slightly differently: they take 2 cycles to execute, and there is a 7-cycle delay corresponding to the length of the pipeline, between the assignment of the branch register (`b0` to `b7`) and the beginning of instruction fetch at this target address. During these 7 cycles, any code can be executed (provided the branch register is not overwritten) like in traditional delay slots of many in-order issue processors. Unlike VLIW processors, delay slots do not have to be filled with `nop`s, a pipeline flush will occur if the delay is not satisfied. In addition, the assignment to a branch register triggers a prefetch stream, and one may want to wait for a few more cycles to let this stream operate.

3.3 Experimental Settings

Throughout this paper, we use ICC version 8, choosing the best result from `-O2` and `-O3`. The target platform is a NovaScale 4020 server from Bull featuring two Itanium2 1.5GHz (Madison) processors.

We used both “regular” and “random” data input to study the effects of our optimizations on the prediction hardware. While for the sake of results stability we used a surrounding loop, predictor learning effectiveness is tested by using data input either completely random or exhibiting regular patterns.

Special care was taken to randomize branch histories without flushing the caches, given the large prediction tables and their integration in the L1 and L2 caches.

4 Generating a Tree of One-Way Conditional Branches

An intuitive algorithm to order a decision tree is a balanced binary tree (built by dichotomy), with a logarithmic distance between each leaf. In practice, a balanced tree has a very poor performance since most branches are locally unpredictable and lead to strong penalties.

Conversely, a comb shape has a better branch behavior due to its bias towards the fall-through path. [4] Over a few tenths of branches, some dichotomy is needed to achieve a logarithmic complexity. An optimization framework should consider this strategy as an option.

4.1 Code Motion and Hoisting

Partially redundant expression elimination (PRE) and invariant code motion can bring significant performance benefits for decision trees. The reason is not specific to the

[4] By default, branches are thus predicted *not taken*; only those with a recent *taken* history occupy hardware prediction structures.

Itanium: if an expression is common to several leaves, it can be moved up to a common branch, filling some unused issue slots into the tree of conditional branches itself.

Thanks to predicated execution, hoisting can be even more aggressive on the Itanium, allowing to fill more issue slots with speculative computations coming from the leaves. ICC and Open64/ORC [7] implement both PRE and this hoisting strategy.

Alternatively, code motion can be used to fill issue slots with instructions ahead of the decision tree (if the decision is not control-dependent on them). This requires less speculation than hoisting, but increases code size by duplicating code along several branches of the tree.

4.2 Node Reordering

The order in which internal nodes are visited can often be changed, permuting the `else if` or `case` leaves at the source level. There are several motivations for this.

- When optimizing for the average-case on a distribution of decisions, leaves with the highest probability of being executed can be moved closer to the root of the tree. This type of optimization is typically interesting for high-performance interpreters.
- When optimizing the performance of a versioned computation kernel, it is more important to take the *relative execution time of the leaves* into account. Indeed, the decision tree overhead is only significant when executing short leaves, i.e., when the versioned kernel is run on small data-sets. Typical short-vector or memory copying operations are of this kind.
- Code motion and hoisting can be more profitable when grouping together leaves with similar or partially redundant expressions.

4.3 Cost Evaluation of the One-Way Comb Approach: Definition

First, we define a performance metric, the *Number of Mispredictions per Decision (MpD)*, as the number of mispredicted branches per taken decision, i.e. per traversal of the decision tree. In addition to input data, *MpD* depends on the decision tree structure.

Cost evaluation for `else if` structures. We recall that the conditions of `else if` constructs are assumed to be interval tests requiring two integer comparisons per cycle, and that the cost of a branch misprediction is 6 cycles (length of the back-end of the pipeline). An extra one cycle overhead is needed to pipeline predicate computations (comparisons) ahead of their qualified branches. The cycle costs for a one-way comb implementation, for the last leaf (worst latency) and for the median one are displayed in Table 1. The *last leaf* corresponds to the case where the targeted leaf is the last one in the comb therefore yielding to the maximum number of branch predictions for this decision tree. Accordingly the *median leaf* corresponds to the case where the targeted leaf is the median one in the comb.

Per cycle, three instruction slots are consumed by the branch and the two compare instructions. This leaves three slots for hoisting instructions of each leaf, or for moving code that precedes the decision tree.

Cost evaluation for `switch` structures. Compared to the previous evaluation, the only difference is the number of integer comparisons per node, reduced to one. The cycle costs for a one-way comb implementation, are given in Table 1. Every cycle, only two slots are consumed, leaving four available slots for hoisting.

Cost model	Last Leaf	Median Leaf
One-Way `else if`	$c_{oneway,last} = 1 + L + 6MpD$	$c_{oneway,median} = 1 + \lceil \frac{L}{2} \rceil + 6MpD$
One-Way `switch`	$c_{oneway,last} = 1 + L + 6MpD$	$c_{oneway,median} = 1 + \lceil \frac{L}{2} \rceil + 6MpD$

Table 1. Cost model in cycle for one-way comb implementation in the case of `else if` and `switch` structures. *MpD* depends on the decision tree structure, furthermore it is not a fixed constant for each cost function.

4.4 Cost Evaluation of the One-Way Comb Approach: Analytical

Considering a tree representation, we call L the number of leaves and $\overline{MpD}$ the average MpD. Of course, $\overline{MpD}$ depends on the input data; yet we show that the *strongly biased predictions in the comb-shaped implementation makes it almost impossible to find a sequence of decisions leading to bad* average *branch prediction performance*.

Indeed considering a dumb branch predictor always predicting each branch as *not taken*, $\overline{MpD}$ will be equal to 1 on a one-way comb because all branches will always be correctly predicted except the last one. Notice this result is independent of input data.

Consider a more representative branch predictor based on the *2-bit saturating up-down counter* [10] shown in Figure 3; computing $\overline{MpD}$ is more difficult. For a comb, whatever is its initial state, we are going to demonstrate that the average number of mispredictions per decision (i.e bad leaves improperly called due to branches predicted as *taken* or the good leaf not called due to the branch in *not taken* predicted state) is bounded by 1.

We will denote $MpD(i)$ the number of mispredictions for the i^{th} decision, $tk(i)$ the number of 2-bit counters in a *taken* state and $TK(i)$ the number of potential mispredictions considering that the *strongly taken* state allows two possible consecutive mispredictions:[5]

$$0 \leq MpD(i) \leq tk(i) + 1 \qquad TK(i+1) \leq TK(i) - MpD(i) + 1 \qquad MpD(i) \leq TK(i)$$

The maximal number of mispredictions per decision is equal to number of branch in *taken* state in the comb plus 1 corresponding to the called leaf potentially in *not taken* state. The counter associated to this called leaf is incremented so the maximal number of possible consecutive mispredictions is decremented by the previous mispredictions and incremented by 1. By induction, one may deduce the following inequality:

$$TK(n+1) \leq n - \sum_{i=1}^{n} MpD(i) \Rightarrow \sum_{i=1}^{n} MpD(i) \leq n - TK(n+1) \leq n$$

For n decisions, the total number of mispredictions is less or equal to n so $\overline{MpD}$ is at most 1 whatever the input data.

[5] $TK(i)$ counts predictors in a *weakly taken* state plus twice those in a *strongly taken* state.

4.5 Cost Evaluation of the One-Way Comb Approach: Experimental

However, the previous reasoning does not easily extend to more precise models of the predictor, due to state-space explosion. In some unfortunate cases, it is expected that false sharing and history patterns may actually degrade the prediction rate beyond the 1 misprediction per decision limit. For example, Table 2 shows experimental results on multiple decision tree sizes and on practical distributions (for a one-way comb implementation): the $\overline{MpD}$ varies between 0 and 1.1.

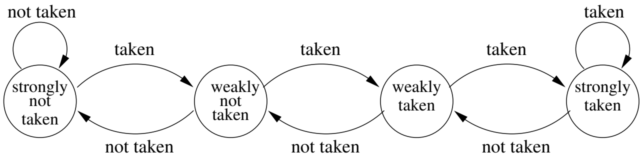

Fig. 3. 2-bit Saturating Up-down Counter

Although the mispredict ratio can be significant — up to 26% — on small decision trees, these experiments confirm that the comb shape keeps the number of mispredictions per decision under control. This result is a good indication that it is possible to design an accurate cost function which does not suffer from input data variability.

Distribution (sequence of leaf selections)	Correct predictions 4 leaves	8 leaves	16 leaves	Mispredictions per Decision 4 leaves	8 leaves	16 leaves
Random	74 %	82 %	88 %	0.9	1.0	1.1
Upwards $(0,1,\ldots,n)$	99 %	83 %	89 %	0.0	0.9	1.1
Downwards $(n,\ldots,1,0)$	99.9 %	88 %	89 %	0.0	0.6	1.1
Up-and-Down $(0,1,\ldots,n,\ldots,1,0)$	96 %	82 %	92 %	0.1	0.8	0.8
Custom (3 *identical then up*)	82 %	85 %	90 %	0.6	0.8	0.9

Table 2. Branch predictor experimental performance on a one-way comb implementation.

Notice the highest prediction rates (96 % to 99.9 %) come from the global predictor, which only succeeds in capturing history patterns on small decision trees. The local predictor can act too on the deeper nodes predictions. Because these nodes are called a few times compared to the total number of decisions, this predictor could recognize a period and, therefore, make correct predictions.

5 Generating a Tree of Multiway Conditional Branches

We now detail the additional strategies offered by *multiway conditional branches*. Of course, hoisting, code motion, node reordering and rebalancing can be generalized from the one-way strategies.

5.1 Dealing With Multiway Branches

Starting from a binary decision tree, using multiway branches is equivalent to merge internal nodes. Node merging can be combined with hoisting, code motion and modifications of node ordering, see Figure 4; Figure 5 shows a three-way comb.

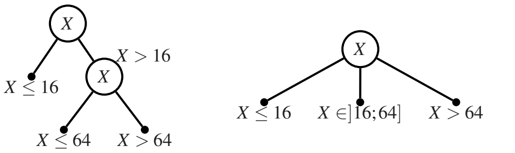

Fig. 4. Merging Nodes in a Decision Tree. (left: not merged, right: merged)

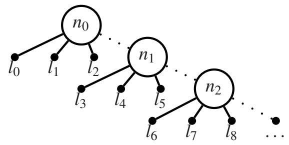

Fig. 5. Comb of multiway branches

5.2 Cost Evaluation of the Multiway Comb Approach

Using multi-way branches makes the combs shorter, the MpD is experimentally kept below 1.1, reinforcing the ability to build an accurate cost function. *MpD* depends on the decision tree structure, furthermore it is not a fixed constant for each cost function.

Cost evaluation for `else if` structures. We recall that the conditions of `else if` constructs are assumed to be interval tests requiring two integer comparisons per cycle.

The cycle costs for a two-way comb implementation, for the last leaf and for the median one are displayed in Table 3. Hoisting is not possible because the branches and the four comparison instructions are already using all of the available slots. For the same reason, using three-way branches will yield to the same cost function (but requires two thirds of the predictions of the two-way implementation).

Cost evaluation for `switch` structures. Compared to the previous evaluation, the only difference is the number of integer comparisons per node, which is reduced to one. For a two-way comb implementation, two slots are available for hoisting. In the case of a three-way comb no hoisting is possible. Costs models are detailed in Table 3.

Cost model	Last Leaf	Median Leaf
Two-Way `else if`	$c_{twoway,last} = 1 + \lceil \frac{L}{2} \rceil + 6MpD$	$c_{twoway,median} = 1 + \lceil \frac{L}{4} \rceil + 6MpD$
Two-Way `switch`	$c_{twoway,last} = 1 + \lceil \frac{L}{2} \rceil + 6MpD$	$c_{twoway,median} = 1 + \lceil \frac{L}{4} \rceil + 6MpD$
Three Way `switch`	$c_{threeway,last} = 1 + \lceil \frac{L}{3} \rceil + 6MpD$	$c_{threeway,median} = 1 + \lceil \frac{L}{6} \rceil + 6MpD$
Three Way `else if`	$c_{threeway,last} = 1 + \lceil \frac{L}{3} \rceil + 6MpD$	$c_{threeway,median} = 1 + \lceil \frac{L}{6} \rceil + 6MpD$

Table 3. Cost model in cycle for two-way comb implementation in the case of `else if` and `switch` structures and three-way comb in case of a `switch` structure. MpD value depends on the cost model.

6 Generating Indirect Branches

Indirect branches have not been popular in recent compilers due to the overhead and high unpredictability of such instructions on superscalar processors. Exploring alternative implementations, we show that the delayed branch and predicated execution capabilities of the Itanium make indirect branches very attractive.

6.1 The Basics on a Simple Example

On the Itanium2, an indirect branch implies computing the target address and storing it in one of the branch registers. The execution of this indirect branch is an unconditional jump to the target address. This is a convenient way to implement Fortran `goto`s.

For sake of simplicity, let us start with the simple one-level `switch` structure with four leaves in Figure 6. Assume that the code size of each leaf is less than $64B$ and starts at base address a. First the code for each leaf is allocated to a very specific address: namely $\texttt{code}_k$ is allocated to address $a + k \times 64B$. Such alignment can be performed by `.skip` assembly directives to line-up instructions on a specific boundary.

Now by multiplying the value of `X` by 64, the value of a relative displacement can be obtained, then added to the instruction pointer and stored in a branch register. A single branch using that computed address will jump directly to the correct leaf.

```
switch(X) {
  case 0:
    code0; break;
  case 1:
    code1; break;
  case 2:
    code2; break;
}
```

```
X*64: compute displacement
Computation of the target address
... delay slots
... (7+ cycles)
Branch to the computed address
a:  code0;
a+64: code1;
a+128: code2;
```

Fig. 6. Target code structures

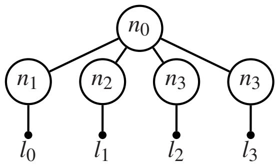

Fig. 7. Indirect linear method

6.2 General One-Level Switch Structures: the Indirect Linear Method

To extend the previous method, it is sufficient to compute the maximum code size over all leaves and to round it to the next power of two. The corresponding code generation strategy is named *indirect linear*. The key advantage of such method is that there is a single branch instruction (always taken) instead of running down a chain of branches. The main drawback is that code layout may be fairly sparse when leaves are disproportionate or when the dispersion of selection constants is too wide.

This problem can often be worked around by using intermediate nodes consisting of a single unconditional branch. This layer of intermediate nodes will reduce the impact on memory. There is a high risk of target address misprediction for these branches when the decision tree is large, because the address prediction tables are far smaller than the outcome prediction ones. Fortunately, the address computation only takes one cycle on the Itanium2 (cf. Section 3). Figure 7 presents an example with a one-level `switch` and intermediate node layer. An alternative followed by ICC consists in storing target addresses in a table, indexed by the value of the `switch` argument. This increases the L1 cache pressure and may lose the benefit from instruction prefetching streams of the previous implementation.

Cost evaluation of the indirect linear method. We implemented a generic version of the indirect linear method in assembly (not shown for lack of space). From this implementation, and assuming a perfect instruction cache locality, the cost of the indirect linear selection of an arbitrary leaf is: $c_{indirect,linear} = 11$.

As explained in Section 3, instruction prefetching is initiated by the assignment to a branch register, therefore, in practice it could be beneficial to extend the delay beyond the minimal time required to update the branch register (7 cycles). Conversely, this code can be further optimized through code motion and hoisting. In the best case, this 7 delay cycles can be concealed, but it means that a large number of instructions must be displaced and possibly speculated (7 cycles x 6 instructions per cycle = 42 instructions).

Inserting an intermediate layer of unconditional branches adds only one or two cycles: the branch and, possibly the address calculation, likely to be mispredicted: $c_{indirect,linear+intermediate} = 13$.

6.3 General One-Level Else If Structures: the Indirect Compare Method

The difficulty in extending the previous technique to `else if` structures is the evaluation of the condition. This is not as simple as with `switch` structures.

First of all, comparisons corresponding to all `else if` conditions have to be performed. From these comparisons, the fastest way to derive a target address is to set a predicate register guarding an `add` instruction to compute the target address. If the comparison is true, the correct target address will be computed, otherwise no operation is performed. Eventually, after all comparisons have been made, an indirect branch always taken will jump to the correct address.

This approach is called the *indirect compare* strategy. In this case, we do not need to make any assumption about the size or layout of the leaves of the tree.

Cost evaluation of the indirect compare method. The cost of the compare indirect can be split in three parts: the cost of all comparisons is $L/3$ (2 comparisons per node, 6 per cycle); the cost of target address updates is $L/6$ (6 additions per cycle); the cost of the branch register assignment, the delay for the indirect branch and its intrinsic latency is $1+6+2=9$. Therefore the cost for the Itanium2 is: $c_{indirect,compare} = \lceil \frac{L}{2} \rceil + 11$.

This formula counts the number of slots used, but the remaining free slots might be filled through (speculative) code motion and hoisting.

Clearly, this strategy is less effective than the indirect linear method in general, because its cost is linear in the size of the decision tree. However, it may be useful for `switch` structures whose dispersion of selection values is too large.

7 Comparison of the Code Generation Strategies

We will now compare the various implementation strategies, taking the impact of hoisting into account. We will provide a worst case evaluation — called *no hoisting* — for which none of the free slots are filled, as well as a best case evaluation — called *hoisting* — assuming all empty slots are filled with useful instructions. For these cost models, we set *MpD* to 1 corresponding to the theoretical worst case when using the decision tree with random data.

7.1 Comparison for Else If Structures

We consider a size-L one-level `else if` structure. The *model-based* cost evaluation for the comb and indirect compare strategies are shown in Figures 8 and 9, respectively for the last and median leaf, and respectively with or without hoisting. The indirect compare method is interesting — especially on small decision trees — when code motion or hoisting can be applied very aggressively (30 to 40 instructions must be moved).

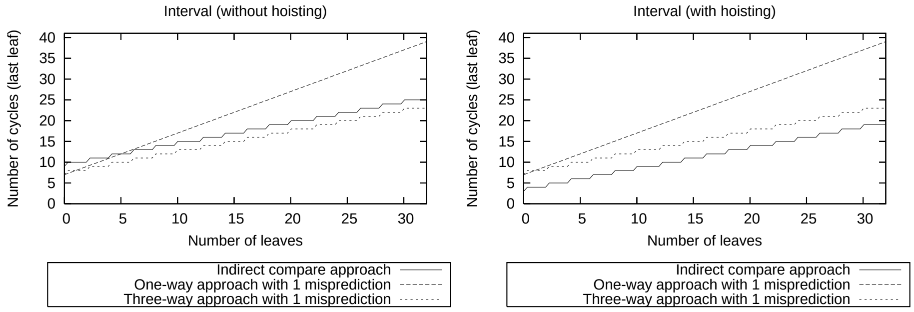

Fig. 8. Cost models comparison for `else if` structures, *last leaf*

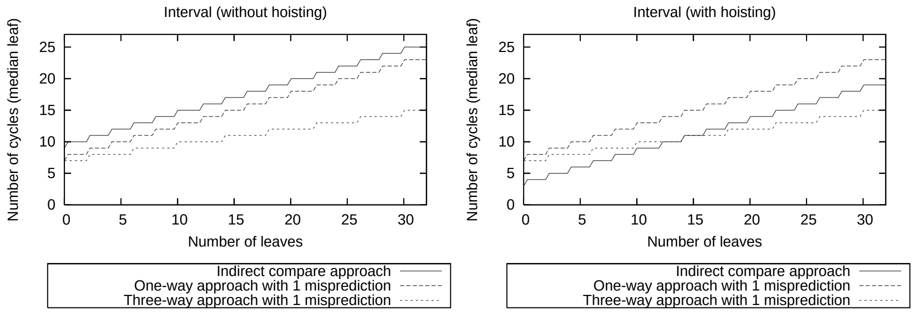

Fig. 9. Cost models comparison for `else if` structures, *median leaf*

7.2 Comparison for Switch Structures

We consider a size-L one-level `switch` structure. The *model-based* cost evaluation for the comb and indirect linear strategies are shown in Figures 10 and 11, respectively for the last and median leaf, and with or without hoisting, respectively. Without hoisting, the indirect linear strategy leads asymptotically to the most effective implementation, but the merits of the different comb approaches are hard to distinguish on smaller decision trees. This complex interplay advocates for an iterative optimization approach to empirically select the best implementation, which is what the compiler does with Profile Guided Optimizations (PGO). The indirect linear approach benefits much more from code motion and hoisting, when performed aggressively the delay before the indirect branch can be completely concealed. This situation is optimistic but may occur in practice, e.g., when leaves have a many instructions in common, or when the code ahead of the decision tree can be moved into the delay slots.

7.3 First Experimental Analysis

We compare here some decision tree implementations. For such study, we considered `switch 16`, `switch 32`, `switch 64` and `else if 9` structures, where all leaves are reduced to a single arithmetic instruction limiting as much as possible the impact of hoisting. Look-

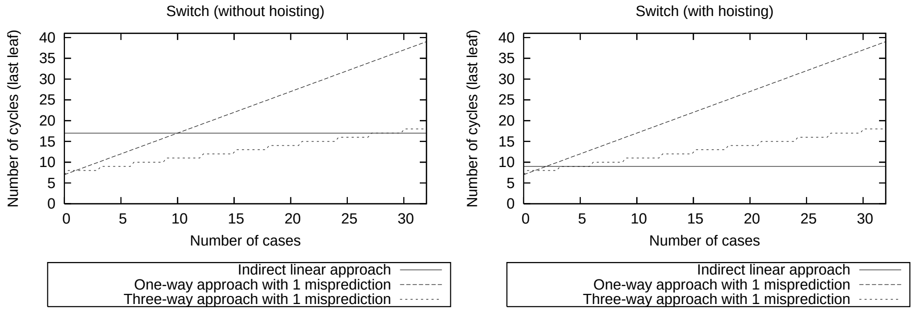

Fig. 10. Cost models comparison for `switch` structures, last leaf

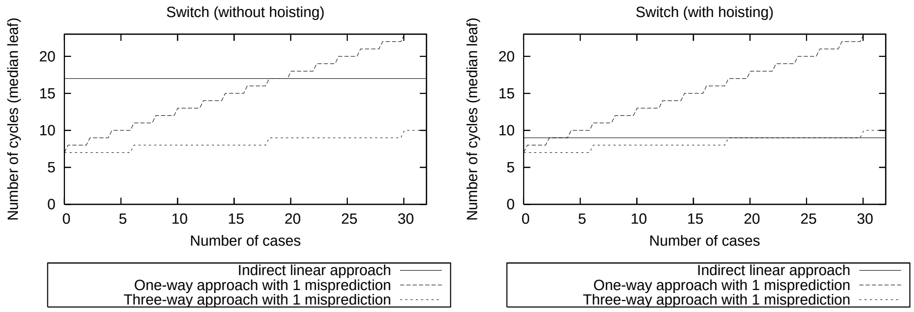

Fig. 11. Cost models comparison for `switch` structures, median leaf

ing at generated codes, rather than a constant degree comb, ICC favored a "mixed-way" approach trading multiway branches for speculative hoisting.

		ICC	One-way	Two-way	Three-Way	Indirect compare	Indirect linear
Switch-16	MpD	0.91	1.10	1.05	0.98	n.a.	0
	Cycle count	10.5	16.5	12.3	10.3		11.2
Switch-32	MpD	0.94	1.08	1.06	0.98	n.a.	0
	Cycle count	12.3	24.3	16.5	13.3		11.1
Switch-64	MpD	0.94	1.08	1.05	0.98	n.a.	0
	Cycle count	12.3	40.4	24.4	18.4		11.2
Else-if-9	MpD	0.96	1.05	1.02	0.94	0	n.a.
(intervals)	Cycle count	12.4	13.3	12.3	11.3	16.2	n.a.

Table 4. Experimental Results: Misprediction per Decision and cycle count (random sequences) for a decision tree

Table 4 compares the codes generated by ICC with different code variants : one-way, two-way, three-way comb and indirect linear variants. Empty cells correspond to irrelevant evaluations. These results clearly show that one-way and two-way combs are sub-optimal, the real confrontation being between ICC, three-way combs and indirect branches code variants. While ICC uses somehow complex heuristics involving combs and even indirect branches — for a `switch` larger than 17, it can be outperformed by simpler approaches, and in general, the efficiency of hoisting is over-estimated by ICC. In addition, the implementation of indirect branches by ICC uses an intermediate look-up table; this only adds a one-cycle delay on small kernels, but may incur higher penalties on real benchmarks (compared to the linear approach with intermediate unconditional branches). Eventually, the indirect compare approach is less effective than combs, but its code motion potential is much higher.

7.4 Application to a Real Code

To validate cost functions on real decision trees, we compared ICC's Daxpy implementations with our indirect linear implementations for short vectors. Daxpy is a BLAS1 function which computes the following scalar-vector product and sum: $y = \alpha \times x + y$ where α is a scalar, and x and y are vectors.

For small kernels performing a Daxpy on vectors whose sizes do not exceed 16 elements, ICC generated a comb structure, mixing one-way and two-way branches. Empty slots are filled with hoisted instructions (mainly addresses computations). Over 16 elements, ICC generated an indirect branch version with hoisting where target leaf address is stored in a look-up table. But it failed use the delay cycles between the branch register update and the branch instruction for hoisting.

We implement three versions of indirect linear Daxpy with an increasing exploitation of delay slots to perform hoisting. **`Hoist_adr`** uses some of the delay slots to compute array addresses necessary for computations in each leaf. **`Hoist_adr_load`** derives from Hoist_adr, all the 16 loads on y and x are now hoisted. **`Hoist_adr_load_fma`** derives from `Hoist_adr_load`, some of the floating point multiply adds required to compute a vector element are now hoisted.

With such aggressive hoisting, leaves only contain floating point arithmetics and stores instructions. Note that in `Hoist_adr_load` and `Hoist_adr_load_fma` versions, special care was required to avoid potential load exception (which can be done using speculative load instructions of the Itanium2).

Indirect Linear	ICC `switch 16`		ICC `switch 17`	
	Cycles (vs ICC)	Gain	Cycles (vs ICC)	Gain
`Hoist_adr`	45.1 (53.5)	15.6 %	45.1 (53.9)	16.3 %
`Hoist_adr_load`	35.4 (53.5)	33.8 %	35.4 (53.9)	34.3 %
`Hoist_adr_load_fma`	31.8 (53.5)	40.6 %	31.8 (53.9)	41 %

Table 5. Performance Gain (in %) of indirect versions over ICC with random vector size. For `switch 16`, ICC generates a comb shape decision tree while for `switch 17` it relies on indirect branch.

Table 5 summarizes performance improvements of our indirect linear versions over ICC. All these indirect linear versions outperform the ICC version from 15.6% to 40%. The performance gain comes from two combined factors :

- Filling up the delay slots allows to execute up to 32 loads and to avoid a misprediction due to wrong branch target address.
- Instructions in leaves were scheduled in a better way than what the compiler did.

It is important to note that in `Hoist_adr`, performance improvement mainly comes from the instructions rescheduling within the leaves whereas for `Hoist_adr_load` and `Hoist_adr_load_fma` versions, performance gain really comes from their hoisting capabilities.

Additionally, our indirect linear versions also surpassed ICC indirect version (switch 17) which uses look-up table because the compiler failed to exploit delay slots for hoisting to compute the target address or to hoist other kind of instruction. We obtained similar results (with a slightly narrower gap with ICC) on a Copy kernel.

8 Conclusion

Studying the Itanium2 architecture, we show that the complexity of the branch mechanism does not hamper the design of accurate cost models, in the special case of decision trees. We build simple models that are both accurate and relatively independent of the input data. Based on these models, we compare the most significant code generation strategies for the Itanium processor family: tree of multiway conditional branches, indirect branch with predicated comparisons, and indirect branch with linear-computation of the target address. We show that no strategy dominates in all cases. Relying on this study we develop a simple kernel outperforming from 15% to 40% a state-of-the-art compiler such as the Intel ICC compiler.

Future works include the design of finer cost models, defining more precisely the impact of hoisting and code motion. Robustness toward either instruction cache locality or statistical distribution are also hot topic of interest. On the long run we also wish to extend our work to multiple levels of nested decision trees, and to integrate our cost models and implementation strategies in an adaptive automatic optimization tool.

Acknowledgments. This work is supported by research grants from CEA DAM, Bruyères-le-Châtel, Bull Les Clayes-sous-Bois and the French Ministry of Research.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *ACM Symp. on Programming Language Design and Implementation (PLDI'03)*, San Diego, California, June 2003.
3. P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *ACM and IEEE Intl. Symp. on Computer Architecture (ISCA'00)*, Vancouver, BC, June 2000.
4. *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*, revision 2.1 edition. http://developer.intel.com/design/itanium/family.
5. X. Li, M.-J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *ACM Conference on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.
6. C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, pages 44–55, Mar./Apr. 2003.
7. Open research compiler. http://ipf-orc.sourceforge.net.
8. H. Packard. Inside the intel itanium 2 processor: an itanium processor family member for balanced performance over a wide range of applications. White paper, Hewlett Packard, July 2002.
9. Philips Semiconductors, Sunnyvale, CA. *TriMedia Compilation System, v.2.1User and Reference Manual*, 1999.
10. T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th International Symposium on Computer Architecture*, pages 124–134, Gold Coast, Australia, 1992. ACM and IEEE CS.