On Conditional Branches in Optimal Search Trees

Michael B. Baer, *Member, IEEE*

Abstract

A commonly used type of search tree is the alphabetic binary tree, which uses (without loss of generality) "less than" versus "greater than or equal to" tests in order to determine the outcome event. The process of finding an optimal alphabetic binary tree for a known probability distribution on outcome events usually has the underlying assumption that the cost (time) per comparison is uniform and thus independent of the outcome of the comparison. This assumption, however, is incorrect in the case of software to be optimized for a given microprocessor, e.g., compiling switch statements or fine-tuning program bottlenecks. Branch prediction causes the cost for the more likely comparison outcome to be less — often far less — than for the less likely comparison outcome. Here we introduce a variety of novel dynamic programming algorithms to solve both such alphabetic binary tree problems and more general search tree problems, optimizing for the behavior of processors with predictive branch capabilities, both static and dynamic. Entropy-based performance bounds can be used to quickly estimate performance of optimal alphabetic binary trees. Solutions found using these methods are often faster in practice than "optimal" search trees as formulated in the literature.

Index Terms

Branch prediction, graph and tree search strategies, optimal alphabetic tree.

I. Introduction

Consider a problem of assigning grades to tests. These tests might be administered to humans or to objects, but in either case there are grades 1 through n — n being 5 in most academic systems — and the corresponding probabilities of each grade, $p(1)$ through $p(n)$, can be assumed to be known; if unknown, they are assumed to be identical. Each grade is determined by taking the actual score, a; dividing it by the maximum possible score, b; and seeing which of n distinct fixed intervals of the form $[v_{i-1}, v_i)$ the key (ratio) a/b lies in, where $v_0 = -\infty$ and $v_n = +\infty$. This process is repeated for different values of a and b enough times that it is worthwhile to consider the fastest manner in which to determine these grades (or, in general, "items").

A straightforward manner of assigning scores would be to multiply (or shift) a by a constant k ($\log_2 k$), divide this by b, and use lookup tables on the scaled ratio. However, division is a slow step in most CPUs — and not even a native operation in others — and a lookup table, if large, can take up valuable cache space. The latter problem can be solved by using a numerical comparisons to determine the score, resulting in a type of search tree known as a *binary decision tree* of an *alphabetic binary tree*. In fact, with this decision tree, we can eliminate division altogether; instead of comparing scaled ratio ka/b with grade cutoff value, v_i, we can equivalently compare ka with bv_i, replacing the slow division of variable integers with a fast multiplication of a variable and a fixed integer. Depending on the application, this can be useful even if $b = 1$ and no division is inherent in the problem. The only matter that remains is determining the structure of the decision tree.

Such trees have a large variety of applications, including nontechnical uses, such as the game of Twenty Questions [1, pp. 94–95] (also known as "Yes and No" [2] or "Bar-kochba" [3]). Technical uses includes the compilation of switch (case) statements [4], [5]. An optimized binary decision tree is known as an *optimal alphabetic binary tree*.

The author is with Ocarina Networks, Inc., 42 Airport Parkway, San Jose, CA 95110-1009 USA (e-mail: calbear@ieee.org).

```
if (V >= 34)      A. compare V, 34
                  B. branch to M if V<34
  if (V >= 42)    C. compare V, 42
                  D. branch to K if V<42
    if (V >= 65)  E. compare V, 65
                  F. branch to I if V<65
      P = 1;      G. P = 1
    else          H. go to N
      P = 2;      I. P = 2
  else            J. go to N
    P = 3;        K. P = 3
else              L. go to N
  P = 4;          M. P = 4
                  N. end
```

Fig. 1. Steps in a simple decision tree

Often times these decision trees are hard coded into software for the sake of efficiency, as in the high-speed low-memory ONE-SHIFT Huffman decoding technique introduced in [6] and illustrated using C code in Fig. 2 of the same paper. A shorter but similar decision tree is illustrated in Fig. 1 above by means of C and assembly-like pseudocode. We discuss this sample tree in Section II of this paper, where a pictorial representation of the tree is given as Fig. 2.

Algorithms used for finding such trees generally find trees with minimum expected path length, or, equivalently, minimum expected number of comparisons [7]–[9]. We, however, want a tree that results in minimum expected run time, which is generally expressed in terms of machine cycles, since these are usually constant time for a given machine in a given mode. The general assumption in finding an optimal decision tree is that these goals are identical, that is, that each decision tree (edge) takes the same amount of time (cost) as any other; this is noted in Section 6.2.2 of Knuth's *The Art of Computer Programming* [10, p. 429]. In exercise 33 of Section 6.2.2, however, it is conceded that this is not strictly true; in the first edition, the exercise asks for an algorithm for where there is an inequity in cost between a fixed cost for a left branch and a fixed cost for a right branch [11], and, in the second edition, a reference is given to such an algorithm [12]. Such an approach has been extended to cases where each node has a possibly different, but still fixed, asymmetry [13].

In practice the asymmetry of branches in a microprocessor is different in character from any of the aforementioned formulations. On complex CPUs, such as those in the Pentium family, branches are predicted as taken or untaken ahead of execution. If the branch is predicted correctly, operation continues smoothly and the branch itself takes only the equivalent of one or two other instructions, as instructions that would have been delayed by waiting for the branch outcome are instead speculatively executed. However, if the branch is improperly predicted, a penalty for misprediction is incurred, as the results of speculatively executed instructions must be discarded and the processor returned to the state it was at prior to the branch, ready to fetch the correct instruction stream [14]. In the case of the Pentium 4 processor, a mispredicted branch takes the equivalent of dozens of instructions [15]. This penalty has only increased with the deeper pipelines of more recent processors. While this time penalty pales in comparison to time taken by division — over a hundred adds and shifts can take place in the time it takes to do one 32-bit division on Pentium-family processors — it certainly reveals comparison tally as being a poor approximation to run time, the ideal minimization.

In this paper, we discuss the construction of alphabetic binary trees — and more general search trees — which are optimized with respect to the behavior of conditional branches in microprocessors. We introduce a general dynamic programming approach, one applicable to such architecture families as the Intel Pentium architectures, which use advanced dynamic branch prediction; ARM Limited's ARM architectures, most instances of which use static branch prediction; and Knuth's MMIX architecture, in which branch instructions explicitly "hint" whether or not the corresponding branches are assumed taken or untaken [16, p. 20]. The first two of these are not only representative of two styles of branch prediction; they are also by far the most popular processor architecture families for 32-bit personal computers and 32-bit embedded applications, respectively. Pentium designs and the XScale [17] — which is viewed as the successor to ARM architecture StrongARM — use dynamic prediction. ARM architectures such as those of the ARM7 and ARM9 families use static branch prediction [18]. Such processors are used for mobile devices such as cell phones and iPods. Some other ARM family processors use no prediction; that is, they speculatively execute instruction by always assuming that branches are untaken. Some more recent ARM processors — like the ARM1176JZF-S, the processor underlying the Apple iPhone — use dynamic branch prediction by default, but have modes for static branch prediction and for no branch prediction [19].

Because the approach introduced here is more general than extant alphabetical dynamic programming methods, using it to find optimal decision trees is somewhat slower, having $O(n^3)$-time $O(n^2)$-space performance. This generality allows for different costs (run times) for different comparisons due to such behaviors as dynamic branch prediction and the use of conditional instructions other than branches. In the simplest case of static branch prediction, entropy-based performance bounds are obtained based on known results from related unequal edge-cost problems. These results can be extended to cases of dynamic branch prediction. It should be emphasized that the one-time $O(n^3)$-time $O(n^2)$-space cost of optimization of these (usually small) problems is dwarfed by even the slightest gain in repeated run-time performance. The main contribution is thus a method by which decision trees can be coded on known hardware in order to have minimum expected execution time.

II. No prediction and static prediction

Consider Knuth's pedagogical MMIX architecture [16], which has a simple rule for branching: If a "hint" indicates that a branch should be taken, a taken branch will take fewer cycles; otherwise, an untaken branch will take fewer cycles. Thus, if we know ahead of time which branch is more likely and which less likely, we can hard code the more likely branch to take $1 + c$ clock cycles and the less likely branch to take $3 + c$ clock cycles, where c represents the time taken by instructions other than the branch itself, e.g., multiplications, additions, comparisons.

It is easy to code asymmetric branch biases for such hint-based architectures, for no branch prediction, and for static branch prediction. In static prediction, opcode or branch direction is used to determine whether or not a branch is presumed taken, the most common rule being that forward conditional branches are presumed taken and backward conditional branches are presumed not taken [14]. If the presumption is satisfied, the branch takes a fixed number of cycles, while, if it is not, it takes a greater fixed number of cycles. Assume, for example, that we want to use a forward branch, which is assumed not to be taken. We thus want the less likely outcome to be the costlier one, that the branch is taken: If it is less likely than not that the item is less than v_i, the branch instruction should correspond to "branch if less than v_i," as in all branches used in Fig. 1.

This *branching problem*, applicable to problems with either no true branch prediction or static branch prediction, considers positive weights c_0 and c_1 such that the cost of a binary path with predictability $b_1 b_2 \cdots b_k$ is $\sum_{j=1}^{k} c_{b_j}$, where $b_j = 0$ if the jth of k comparisons is mispredicted and $b_j = 1$ otherwise. Such tree paths are often pictorially illustrated via edge length on the corresponding tree, so that path depth corresponds to path cost, as in Fig. 2. This tree corresponds to the C and pseudocode of Fig. 1. The overall expected cost (time) to minimize is

$$T_{p,c}(b) \triangleq \sum_{i=1}^{n} p(i) \sum_{j=1}^{l(i)} c_{b_j(i)}$$

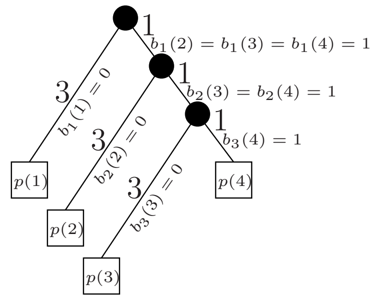


Fig. 2. An optimal branch tree with edge costs for $c = (c_0, c_1) = (3, 1)$

where $p(i)$ is the probability of the ith item, $l(i)$ is the number of comparisons needed, and $b_j(i)$ is 0 if the result of the jth branch for item i is contrary to the prediction and 1 otherwise.

More formally,

Given	$p = (p(1), p(2), \ldots, p(n)),\ p(i) > 0, \sum_{i=1}^{n} p(i) = 1$;
	$c_0, c_1 \in \mathbb{R}_+$ such that $c_0 \geq c_1$;
find	B, a full binary tree;
	b, an assignment of costs to edges of B such that each nonleaf is connected to its children by edges, one with cost c_0, and the other with cost c_1;
minimizing	$T_{p,c}(b) \triangleq \sum_{i=1}^{n} p(i) \sum_{j=1}^{l(i)} c_{b_j(i)}$;
where	the jth edge along the path from root to ith leaf is assigned cost $c_{b_j(i)}$;
	the number of edges on the path from root to ith leaf is $l(i)$.

Sample representations are shown in Fig. 2 and Fig. 4, the former being labeled with the values of $b_j(i)$. Again, to emphasize the total cost in this pictorial representation, edges are portrayed with depth proportional to their cost. The cost (and thus pictorial depth) of leaf 3 in Fig. 2 is, for example,

$$\begin{aligned}\sum_{j=1}^{l(3)} c_{b_j(3)} &= c_{b_1(3)} + c_{b_2(3)} + c_{b_3(3)} \\ &= c_1 + c_1 + c_0 = 1 + 1 + 3 = 5.\end{aligned}$$

Table I gives the context for this branching problem among other binary tree optimization problems. These other problems are referred to as in the survey paper [20]. In most problem formulations, edge cost is fixed, and, where it is not fixed, edges generally have costs according to their order, i.e., a left edge has cost c_0 and a right edge has cost c_1. Relaxing this edge-order constraint in the unequal-cost alphabetic problem results in the branching problem we are now considering. Relaxing the alphabetic constraint from either the original alphabetic problem or the branching problem leads to Karp's nonalphabetic (coding) problem; since output items in Karp's problem need not be in a given (e.g., alphabetical) order, the tree optimal for the ordered-edge nonalphabetic problem is also optimal for the unordered-edge nonalphabetic problem.

Thus the cost T^{Karp} for the optimal tree under Karp's formulation — also called the *lopsided tree problem* — is a lower bound on the cost of the optimal branch tree, whereas the cost T^{Itai} for the optimal tree under Itai's (alphabetic) formulation is an upper bound on the cost of the optimal branch tree. This enables the use of bounds in [21] — including the lower bound originally formulated in [22] — for the branching problem. Specifically, if b^{opt} is the optimal branching function and $T^{\text{opt}} = T_{p,c}(b^{\text{opt}})$ the associated cost for the optimal tree, then

$$\frac{H(p)}{d} \overset{[22]}{\leq} T^{\text{Karp}} \leq T^{\text{opt}} \leq T^{\text{Itai}} \overset{[21]}{\leq} \frac{H(p)+1}{d} + \max\{c_0, c_1\}$$

restriction on output order	*restriction on edge order and/or cost* **Constant edge cost**	**Fixed edge-cost order**	**Unrestricted edge-cost order**
Alphabetic	Hu-Tucker [8]–[10], [23]	Itai [12], [13]	branching problem
Nonalphabetic	Huffman [24]–[26]	Karp [27]–[29]	

TABLE I
TYPES OF DECISION TREE PROBLEMS

where H is the entropy function $H(p) = -\sum_i p(i) \log_2 p(i)$ and d satisfies $2^{-dc_0} + 2^{-dc_1} = 1$. If $\rho = c_0/c_1$ and x is the sole positive root of $x^\rho + x - 1 = 0$, then $d = -c_1^{-1} \log_2 x$. Thus, for example, when $c = (3,1)$,

$$x = \sqrt[3]{\frac{1}{2} + \sqrt{\frac{31}{108}}} - \sqrt[3]{-\frac{1}{2} + \sqrt{\frac{31}{108}}}$$

so $d = \log_2 x^{-1} \approx 0.5515$ and

$$T^{\text{opt}} \in [(1.813\ldots)H(p), (1.813\ldots)H(p) + 4.813\ldots].$$

When $c = (2,1)$, $x = 1/\phi$ so $d = \log_2 \phi$, where ϕ is the golden ratio, $\phi = (\sqrt{5}+1)/2$. These bounds can be used to estimate optimal performance and determine — in $O(n)$ time — whether or not to use a decision tree when it is one of multiple implementation choices.

The key to constructing an optimizing algorithm is to note that any optimal branching tree must have all its subtrees optimal; otherwise one could substitute an optimal subtree for a suboptimal subtree, resulting in a strict improvement in the result. The branching problem is thus, to use the terminology of [30], *subtree optimal*. Each tree (and subtree) can be defined by its *splitting points*. A splitting point being s for the root of the tree means that all items (grades) after s and including s are in the right subtree while all items before s are in the left subtree, as per the convention in [7], [10], [23] (and contrary to that in [12]). Since there are $n-1$ possible splitting points for the root, if we know all potential optimal subtrees (of sizes 1 through $n-1$) for all possible ranges, the splitting point can be found through sequential search of the possible combinations. The optimal tree is thus found through dynamic programming, and this approach has $O(n^3)$ time complexity and $O(n^2)$ space complexity, in a similar manner to [23].

The dynamic programming algorithm is relatively straightforward. Each possible optimal subtree for items i through j has an associated cost, $c(i,j)$ and an associated probability $p(i,j)$; at the end, $p(1,n)$ is 1, and $c(1,n)$ is the expected cost (run time) of the optimal tree.

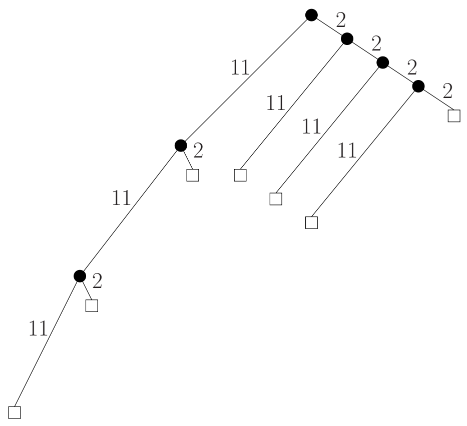
The base case and recurrence relation we use are similar to those of [12]. Given unequal branch costs c_0 and c_1 and probability mass function $p(\cdot)$ for 1 through n,

$$\begin{array}{rcl} c(i,i) & = & 0 \\ c'(i,j) & = & \min_{s\in(i,j]}\{c_0 p(i,s-1) + c_1 p(s,j) + c(i,s-1) + c(s,j)\} \\ c''(i,j) & = & \min_{s\in(i,j]}\{c_1 p(i,s-1) + c_0 p(s,j) + c(i,s-1) + c(s,j)\} \\ c(i,j) & = & \min\{c'(i,j), c''(i,j)\} \end{array} \qquad (1)$$

where $p(i,j) = \sum_{k=i}^{j} p(i)$ can be calculated on the fly along with $c(i,j)$, which is calculated in order of increasing $|j-i|$. The last minimization determines which branch condition to use (e.g., “assume taken” vs. “assume untaken”), while the minimizing value of s is the splitting point for that subtree. The branch condition to use — i.e., the bias of the branch — must be coded explicitly or implicitly in the software derived from the tree.

Knuth [7] and Itai [12] begin with similar algorithms, then reduce complexity by using the property that the splitting point of an optimal tree for their problems must be between the splitting points of the two (possible) optimal subtrees of size $n-1$. Note that [12] claims that this property can be extended to nonbinary decisions, a claim that was later disproved in [31]. The (binary) branching problem considered here also lacks this property. Consider $p = (0.3, 0.2, 0.2, 0.3)$ and $c = (3, 1)$, for which optimal trees split either at 2, as in Fig. 2, or at 4, the mirror image of this tree. In contrast, the two largest subtrees, as illustrated in the figure and its mirror image, both have optimal splitting points at 3. Similarly, applying the less complex Hu-Tucker approach [8], [32] to the branch problem fails for $p = (0.2, 0.15, 0.15, 0.2, 0.3)$ and $c = (3, 1)$.

The optimal tree of Fig. 2 is identical to the optimal tree returned by Itai's algorithm for order-restricted edges [12]. Consider a larger example in which this is not so, the binomial distribution $p = (1, 6, 15, 20, 15, 6, 1)/128$ with $c = (11, 2)$. If edge order is restricted as in [12], the tree at Fig. 3(a) is optimal, yielding an expected cost of 15.109375. If we relax the restriction, as in the problem under consideration here, the tree at Fig. 3(b) is optimal, with an expected cost of 12.984375, a 14% improvement.


(a) Edge order restricted

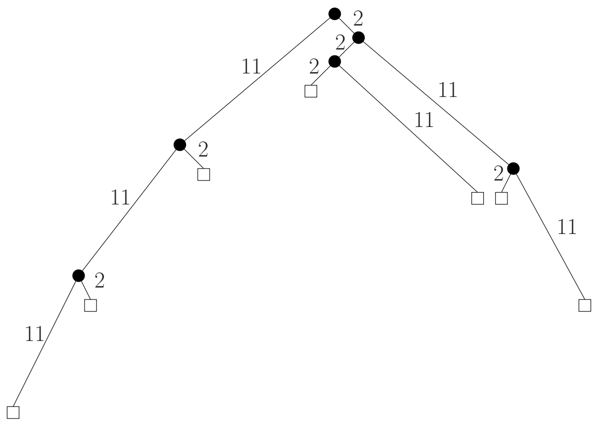

(b) Edge order unrestricted

Fig. 3. Optimal branch trees for two restriction types for $p = (1, 6, 15, 20, 15, 6, 1)/128$ with $c = (11, 2)$

A practical application of this formulation is encountered in implementation of the ONE-SHIFT Huffman decoding technique introduced in [6]. This implementation of optimal prefix coding is fastest for applications with little memory or small caches. (If enough memory is available, other approaches can be somewhat faster for certain prefix codes and architectures, especially when architecture details are taken into account in program design [33]). Where the ONE-SHIFT technique is the preferred technique, we can apply the methods of this section to optimize the method's decision tree. In the implementation illustrated in [6], the decision tree is used to determine codeword lengths based on 32-bit keys. The suggested "optimal search" strategy involves a hard-coded decision tree in which branches occur if "greater than or equal to" each splitting point; in most static branch schemes, this would result in "less than" taking fewer cycles than "greater than or equal to," but the tree used in [6] was found assuming fixed branch costs [34]. Here we show that we can improve upon this.

Consider the optimal prefix code for random variable X drawn from the Zipf distribution with $n = 2^{16}$, that is, $\mathbb{P}[X = i] = 1/(i \sum_{j=1}^{n} j^{-1})$ which is approximately equal to the distribution of the n most common words in the English language [35, p. 89]. Using Huffman coding, one can find that this code has codeword lengths $\ell(X)$ between 4 to 20, with the number of codewords of each size and the probability that the codeword will be a certain size given by Table II.

Now consider a decision tree to find codeword lengths with an architecture in which comparisons that result in untaken branches take 3 cycles (for both compare and branch), while comparisons that result in taken branches take 5 cycles. This asymmetry, similar to that of many ARM architectures, is small, but taking advantage of it results in an improved tree. This optimal tree, shown in Fig. 4, takes an average of 15.93 cycles, while the "optimal search" (Hu-Tucker) approach takes an average of 16.44 cycles. This 3.1% improvement, although not as large as that of the binomial example, is still significant due to the

length (ℓ)	# of codewords		$p(i)$		
4	1	(2^0)	$\mathbb{P}[\ell(X)=4]$	=	0.08570759
5	2	(2^1)	$\mathbb{P}[\ell(X)=5]$	=	0.07142299
6	4	(2^2)	$\mathbb{P}[\ell(X)=6]$	=	0.06509695
7	8	(2^3)	$\mathbb{P}[\ell(X)=7]$	=	0.06216987
8	16	(2^4)	$\mathbb{P}[\ell(X)=8]$	=	0.06076807
9	32	(2^5)	$\mathbb{P}[\ell(X)=9]$	=	0.06008280
10	64	(2^6)	$\mathbb{P}[\ell(X)=10]$	=	0.05974408
11	128	(2^7)	$\mathbb{P}[\ell(X)=11]$	=	0.05957570
12	256	(2^8)	$\mathbb{P}[\ell(X)=12]$	=	0.05949175
13	512	(2^9)	$\mathbb{P}[\ell(X)=13]$	=	0.05944984
14	1024	(2^{10})	$\mathbb{P}[\ell(X)=14]$	=	0.05942890
15	2048	(2^{11})	$\mathbb{P}[\ell(X)=15]$	=	0.05941844
16	4096	(2^{12})	$\mathbb{P}[\ell(X)=16]$	=	0.05941321
17	8192	(2^{13})	$\mathbb{P}[\ell(X)=17]$	=	0.05941059
18	16384	(2^{14})	$\mathbb{P}[\ell(X)=18]$	=	0.05940928
19	32747	$(2^{15}-1)$	$\mathbb{P}[\ell(X)=19]$	=	0.05940732
20	2		$\mathbb{P}[\ell(X)=20]$	=	0.00000262

TABLE II
DISTRIBUTION OF HUFFMAN CODEWORD LENGTHS FOR ZIPF'S LAW

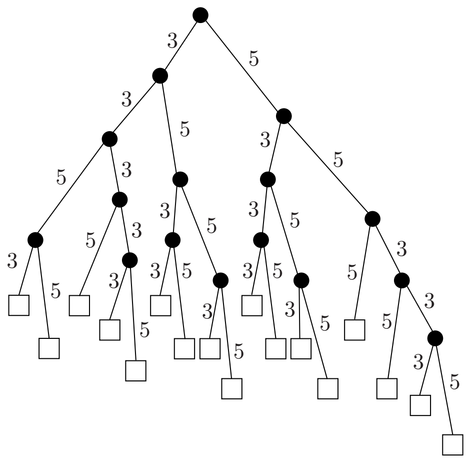


Fig. 4. Optimal branch tree for codeword lengths in optimal prefix coding of Zipf's law

impact of the decision tree on overall algorithm speed.

The $O(n^3)$ complexity used to get these better results is generally not limiting. Although this restricts the range of problems solvable within a given time, most hard-coded decision tree problems are small enough to be solved quickly. Larger problems do not generally follow this model due to issues such as instruction caching. In addition, some problems, such as the aforementioned grade-assignment problem, are actually better solved via alternative means when n is large; a division and lookup table might be more suitable, for example, if there are dozens of different grades. Note that if larger decision trees are required, the $O(n^2)$ space requirement for finding them would likely become an issue before time complexity.

If $p(\cdot)$ is unknown or unspecified, it is common to assume that $p(i) = 1/n$. This is justified by noting that, if $p(\cdot)$ is considered a random vector drawn from the probability simplex according to density g — as in [36] — the expected cost for a given coding scheme is

$$\mathbb{E}_p[T_{p,c}(b)] = \mathbb{E}_p\left[\sum_{i=1}^{n} p(i) \sum_{j=1}^{l(i)} c_{b_j(i)}\right] = \sum_{i=1}^{n} \mathbb{E}_p[p(i)] \sum_{j=1}^{l(i)} c_{b_j(i)}$$

and, if g is a symmetric function over its arguments, $\mathbb{E}[p(i)] = 1/n$ for all i. Since edge costs are not fixed, optimal trees for this uniform distribution need not be *complete trees*, that is, full trees for which all leaves have either depth $\lfloor \log_2 n \rfloor$ or $\lceil \log_2 n \rceil$. For example, the tree in Fig. 2 is optimal for this uniform distribution with an average cost of 3.75 (again, with $c = (3, 1)$), whereas the complete tree for $n = 4$ results in an average cost of 4. This is thus a better approach to use for compilers that code switch (case) statements partially [5] or entirely [4] as decision trees.

In most instances of static prediction, the disparity in performance between correctly anticipated branches and incorrectly anticipated branches is not as great as that for recent versions of the Pentium architectures, in which properly modeling the asymmetry of decision tree performance is even more important.

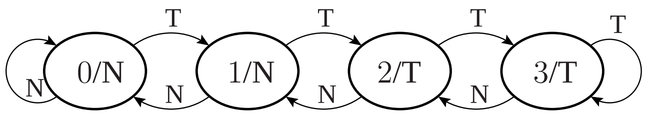
III. More advanced models

With dynamic branch prediction [14], which in more advanced forms includes branch correlation, branches are predicted based on the results of prior instances of the same and different branch instructions. This results in improved branch predictability for most software implementations, especially those in which branch profiling does not enter into software design. Where branch profiling does take place, however, the gains are often only marginal [14, pp. 245–248]. Thus, although large processors and general-purpose processors generally include dynamic prediction, many small processors and low-power processors forgo dynamic prediction, as this feature's sophistication requires the usage of significant additional semiconductor area and power for the associated logic. Where dynamic prediction is used, several predictors will often be used for the same branch instruction instance; the predictor in a given iteration will be based on the history of that branch instruction instance and/or other branches. In the problem we are concerned with, however, this does not result in as many complications as one might expect; the probability of a given branch outcome conditional on the branches that precede it is identical to the probability of the branch outcome overall. In the case of previous branch outcomes for the same search instance — i.e., those of ancestors in the tree — any given outcome is conditioned on the same events — i.e., the events that lead to the branch being considered. In the case of branches used to find previous items, if items are independent, so are these branches. In the case of branches outside of the algorithm, these can also be assumed to be either fixed given or independent of the current branch.

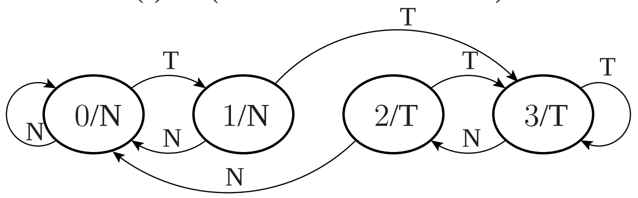
Thus, as long as each branch predictor is assigned at most one of the decision tree branches, prediction can be modeled as a random process. Although initial predictions are made using static criteria (often including optimal branch hints, as in the Intel Pentium 4 [37, p. 2-2] and the MIPS R4000 [38, p. 21]), the dynamic process results in each predictor converging to a stationary distribution, which can be analyzed and optimized for. Such a random process necessarily performs worse than optimized static prediction, although, in most instances, the difference is not too great. The cost of each branch result can be determined by the expected time taken by the branch, based on the costs involved and the probability that the branch is correctly predicted. Simple analysis of the stationary distribution of a branch prediction Markov chain, e.g., [39], can yield the expected time for a given branch direction as a function of the probability of the branch.

Consider the example of using a saturating up-down counter — the two-bit Markov chain of [40] shown as a Moore state diagram in Fig. 5(a) — for branch prediction. If the predictor is in states 0 or 1, it predicts that a branch will not be taken (N), whereas if it is in states 2 or 3, it predicts that a branch will be taken (T). If the actual branch is untaken, the state will decrement (with a lower bound of 0); otherwise, it will increment (with an upper bound of 3). The probability of misprediction is thus the joint probability of being in states 0 or 1 and the branch being taken plus the joint probability of being in 2 or 3 and the branch being untaken. This probability, found using standard algebraic methods for finding the stationary distribution, is

$$f_{\mathrm{A2}}(p_1) \triangleq \mathbb{P}[\text{mispredict on A2}] = \frac{p_1 - p_1^2}{1 - 2p_1 + 2p_1^2}$$



(a) A2 (recent Pentium architectures)



(b) A3 (*Computer Architecture – A Quantitative Approach*)

Fig. 5. Moore state diagrams for branch prediction

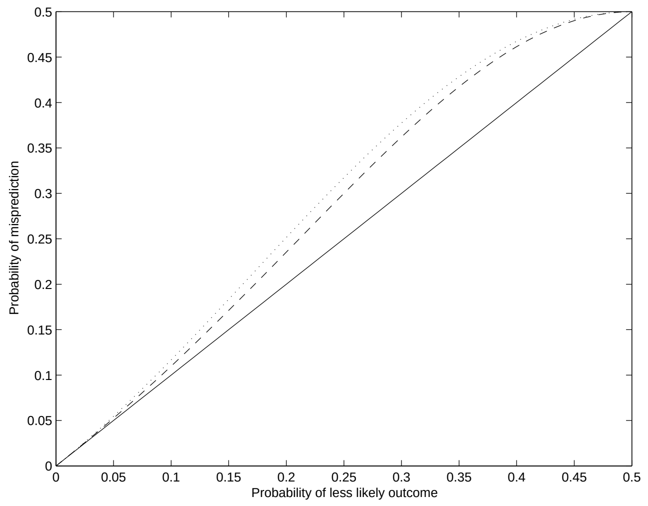


Fig. 6. Static and dynamic branch misprediction rates (top-to-bottom: A3, A2, static prediction)

where p_1 is the probability the less likely event will occur given the branch being considered. This Markov chain is used by the more recent Pentium architectures [15] and is referred to by Yeh and Patt as Automation A2 [41]. If branch prediction instead uses the two-bit Markov chain of [42], as in the MIPS-influenced pedagogical architecture in [14] and in Fig. 5(b), then the probability of a misprediction is given by

$$f_{\mathrm{A3}}(p_1) \triangleq \mathbb{P}[\text{mispredict on A3}] = \frac{p_1 + p_1^2 + 4p_1^3 + 2p_1^4}{1 - p_1 + p_1^2}.$$

This chain is referred to by Yeh and Patt as Automation A3.

Other state diagrams were considered by Yeh and Patt, but are not in as wide use, either being too simple or lacking symmetry between taken and untaken branches. We should also note that these automations existed prior to Yeh and Patt, e.g., in [43], where it is noted that, for each of the tested programs and architectures, there is little to no difference in performance between them. For the decision tree application, asymptotic performance is determined by the corresponding stationary misprediction rate (Fig. 6), which is worst for Automation A3 (dotted plot), having a rate up to 26.92% worse than that for static prediction (solid plot). Automatic A2 (dashed plot) does better, being at most 20.71% worse than static prediction. These statistics can be used as performance bounds in combination with those derived in Section II to approximate performance in linear time prior to application of the algorithm.

In either dynamic case, we can compute optimal expected cost from the probabilities of each branch. Assume that taken and untaken branches are symmetric, that is, a branch rightly predicted as untaken has the same cost as a branch rightly predicted as taken, and mispredicted branches similarly have identical costs. Let the type of dynamic prediction be A, let the probability of the more likely subtree be $p_{\max}$, and let the probability of the less likely subtree be $p_{\min}$, so that $p_{\min} + p_{\max} \leq 1$ and $p_1 = p_{\min}/(p_{\min} + p_{\max})$ is the probability of the less likely outcome conditional on the branch being decided upon. Then, instead of the static cost of $c_0 p_{\min} + c_1 p_{\max}$, the expected cost of a given branch is

$$C(p_{\min}, p_{\max}) \triangleq c_0(p_{\min} + p_{\max}) f_A\left(\frac{p_{\min}}{p_{\min} + p_{\max}}\right) + c_1(p_{\min} + p_{\max})\left(1 - f_A\left(\frac{p_{\min}}{p_{\min} + p_{\max}}\right)\right).$$

Thus this plus the inductively computed costs of the subtrees is the overall tree cost.

Additional performance factors might include an additional asymmetry between taken and untaken branches, the performance of branch target buffers (which are discussed in [43] and [14]), and differences among different comparison types. For example, if a $(<, \geq)$ comparison with a certain value has a smaller cost than a comparison with another value — say a comparison with a power of two times a variable is faster due to reduced calculation time — then this can also be taken into account. Similarly, conditional instructions, often preferable to conditional branches, can often be used, but only to eliminate a branch to leaves in the decision tree. Thus branches deciding between only two items might be accounted differently than other branches.

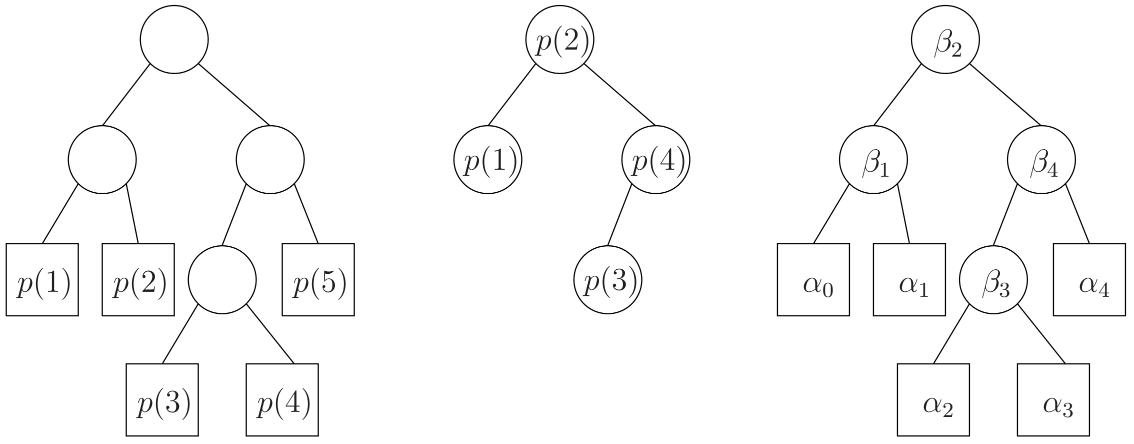
With such a variety of coding options, there could be multiple possible costs for any particular decision. A general cost function taking all this into account represents as $C_k(p', p'', i, j, s)$ the cost of choosing the kth of m splitting methods for the step necessary to split a subtree for items $[i, j]$ at splitting point s, with splitting outcome probabilities p' and p''. (The most common value for m is 2, the two choices being to assume a taken branch versus to assume an untaken branch.) The corresponding generalization of (1) is:

$$\begin{array}{rcl} c(i,i) & = & 0 \\ c_k(i,j) & = & \min_{s \in (i,j]} \{C_k(p(i,s-1), p(s,j), i, j, s) + c(i,s-1) + c(s,j)\} \quad \forall k \\ c(i,j) & = & \min_{k \in [1,m]} \{c_k(i,j)\}. \end{array} \tag{2}$$

Once again, this is a simple matter of dynamic programming, and, assuming all C_k are calculable in constant time, this can be done in $O(mn^3)$ time and $O(n^2 + n \log m)$ space, the $\log m$ term accounting for recalculation and storage of the type of cost function (decision method) used for each branch. An even more general version of this could take into account properties of subtrees other than those already mentioned, but we do not consider this here.

Because dynamic prediction is adaptive to dynamic branch performance, we need not explicitly code branch bias; the more and less likely branch outcomes will automatically be detected. However, most dynamic predictors begin with state that depends on the type of branch in the same manner as static prediction. That is, the first time a branch is encountered, it is usually statically predicted. Thus it is often worthwhile to modify the algorithm so as to have initial iterations of the decision tree behave as well as possible given the tree optimal for asymptotic behavior. Such a modification increases neither time nor space complexity for the above algorithm.

If the software is predetermined but the tree is not — that is, if the software is not hard coded for the specific tree — then matters change entirely; no prediction and static prediction result in this problem being equivalent to that of ordered edges, the problem proposed by Knuth and considered by Itai. Dynamic prediction with correlations, on the other hand, can result in a number of outcomes, depending on implementation. The software on a given processor could have near-perfect distinguishing of outcomes, in which the above dynamic analysis persists. More likely, without sufficient unrolling [14] of the tree data structure, there would be confusion of outcomes, as the software would not know whether all previous branches untaken indicates being at the root, at its right child, at its right child's right child, etc. Optimizing for the complex dependencies involved with such a system is no longer within the above framework, and the overall averaging effect means that one might want to just use the tree optimal for a corresponding static problem. Therefore the aforementioned methods are usually best suited for when the user has the option



(a) Alphabetic tree ($\beta_i = 0$) (b) Search tree ($\alpha_i = 0$) (c) Full search tree

Fig. 7. Sample search trees

of designing software specifically for a given decision tree, or designing hardware and/or programmable logic to allow the methods to work in fixed software.

IV. Search trees and equality comparisons

Knuth showed how a dynamic programming approach can be used for general search trees [7], in which the decision is no longer binary, but is instead, "Is the output greater than, less than, or equal to x?" This allows items to be implicitly or explicitly stored within the internal nodes (nonleaves) of the decision tree and allows us to consider items that might not be in a search tree. This model generalizes the concept of an alphabetic decision tree and can be used for applications in which there is an inherent "dictionary" known of items, such as token parsing and spell checking. Probabilities for both present and missing items are then needed.

Before formalizing this, we should note a few things about the applicability of the search tree model. Clearly the grade-finding problem at the beginning of this paper does not fall into this model, as strict equality cannot be tested for apart from directional inequality. Even where this model is applicable, it can be too restrictive. For example, this model is often inferior to the alphabetic model for the simple reason that, on most hardware, including all hardware considered here, three-way branches are not native operations. They must thus be simulated by two two-way branches in a manner that actually results in greater run time. Experimental analysis of this phenomenon can be found in [44] and numerical analysis can be found in [45] and [46, pp. 344–345]. These all find that an alphabetic tree is usually preferable in practice. Thus we only briefly discuss issues of this search tree model.

For items 1 through n', β_i is defined as the probability that a search yields item i and α_i as the probability that a search fails and the item not in the search tree would be lexicographically between items i and $i+1$; α_0 is the probability it is before β_1 and α_n is the probability it is after β_n. Thus

$$\sum_{i=1}^{n'} \beta_i + \sum_{i=0}^{n'} \alpha_i = 1.$$

The alphabetic tree scenario is a special case, with $n' = n-1$, $\beta_i = 0$, and $\alpha_i = p(i+1)$, as in Fig. 7(a) for $n = 5$. Fig. 7(b) is a similar search tree configured for a three-way comparison; this time, there are only four items, and it is assumed that all items searched for will be in the tree. Fig. 7(c) is the same four-item search tree allowing one to search for both items in the tree (with probabilities $\{\beta_i\}$) and ranges of missing items (with probabilities $\{\alpha_i\}$).

In such a model, there are now three costs associated with a given node; the cost of the two branches c_0 and c_1, and the cost of an equality, e. The addition of this cost to (1) in the case of static prediction or no prediction yields:

$$\begin{aligned}
c(i,i) &= 0 \\
p(i,i) &= \alpha_i \\
c'(i,j) &= \min_{s\in(i,j]}\{c_0 p(i,s-1) + c_1 p(s,j) + e\beta_s + c(i,s-1) + c(s,j)\} \\
c''(i,j) &= \min_{s\in(i,j]}\{c_1 p(i,s-1) + c_0 p(s,j) + e\beta_s + c(i,s-1) + c(s,j)\} \\
c(i,j) &= \min\{c'(i,j), c''(i,j)\} \\
p(i,j) &= p(i,s-1) + \beta_s + p(s,j) \quad \forall s
\end{aligned}$$

where the root case is $c(0,n')$ and $p(i,j)$ is usually calculated using the optimizing s for the overall subtree in question.

Again, one can generalize this as in (2); for example, the cost of an equality comparison need not be fixed. A further generalization in which the equality comparison key value is different than the inequality comparison value has been considered for the constant edge costs, e.g., [47], [48]. Approaches for solving this have led to the more germane problem in which, rather than allowing an inequality *and* an equality comparison in each step, one allows an inequality *or* an equality comparison in each step. This is known as the *two-way key comparison* problem. If data are highly irregular such that the most probable item is much more probable than any other and is alphabetically neither first nor last, then an initial equality comparison to the most probable item — if feasible — would likely improve on the "optimal" $(<,\geq)$ decision tree. For fixed edge costs, the algorithm for solving this is a $O(n^5)$-time $O(n^3)$-space dynamic programming algorithm [49]. In this algorithm, instead of just i and j, a third variable d represents the number of items missing from the subtree due to equality comparisons above this subtree; this accounts for the increased complexity. This algorithm uses the conjecture that equality comparisons should always be with the most likely (remaining) item. This was not proved for equal edge costs, and, even given its veracity for equal edge costs, it is not clear whether this would also be true for unequal edge costs. Nevertheless, no counterexample has been presented, so it is a safe assumption to make, especially since such trees would necessarily perform at least as well as the optimal binary decision tree.

The two-way comparison algorithm has been extended to a large variety of problems, including a problem with nine different branch costs: unequal (ordered) costs for $(=,\neq)$ testing, unequal (ordered) costs for $(\leq,>)$ testing, unequal (ordered) costs for $(<,\geq)$ testing, and unequal (ordered) costs for three-way testing [50, Chapter 9]. This algorithm can be easily modified for unordered costs by adding tests for $(\neq,=)$, $(>,\leq)$, $(\geq,<)$, and other three-way tests. Other modifications can be made in a similar manner to those discussed in this paper. Note that some variants of this problems have complexity reduced from $O(n^5)$ to $O(n^4)$ [49], [51], although this has not been shown to be true of the more general cases which most accurately represent the behavior of hard-coded search trees.

V. Conclusion

In this paper, we presented methods for finding optimal decision and search trees given the real-world behavior of microprocessors, in which not all queries and decision outcomes have identical temporal costs. This approach most often assumes we can hard code the decision tree based on a known probability distribution and known processor behavior. The simplest method, that of Section II, must be generalized for more complex processor prediction techniques, as well as for other subtler performance considerations and for cases in which equality comparisons are allowed. Due to the large asymmetry of branch performance in complex processors, this often results in strictly better hard-coded search trees than the "optimal" trees produced using traditional methods.

References

[1] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 1st ed. New York, NY: Wiley-Interscience, 1991.

[2] C. Dickens, *A Christmas Carol*. London, UK: Chapman and Hall, 1843, available from http://www.gutenberg.org/etext/46.

[3] A. Rényi, *A Diary on Information Theory*. New York, NY: John Wiley & Sons Inc., 1987, original publication: *Naplò az információelméletről*, Gondolat, Budapest, Hungary, 1976.

[4] A. Sale, “The implementation of case statements in Pascal,” *Softw., Pract. Exper.*, vol. 11, no. 9, pp. 929–942, Sept. 1981.
[5] J. L. Hennessy and N. Mendelsohn, “Compilation of the Pascal case statement,” *Softw., Pract. Exper.*, vol. 12, no. 9, pp. 879–882, Sept. 1982.
[6] A. Moffat and A. Turpin, “On the implementation of minimum redundancy prefix codes,” *IEEE Trans. Commun.*, vol. 45, no. 10, pp. 1200–1207, Oct. 1997.
[7] D. E. Knuth, “Optimum binary search trees,” *Acta Informatica*, vol. 1, pp. 14–25, 1971.
[8] T. C. Hu and A. C. Tucker, “Optimal computer search trees and variable-length alphabetic codes,” *SIAM J. Appl. Math.*, vol. 21, no. 4, pp. 514–532, Dec. 1971.
[9] A. M. Garsia and M. L. Wachs, “A new algorithm for minimum cost binary trees,” *SIAM J. Comput.*, vol. 6, no. 4, pp. 622–642, Dec. 1977.
[10] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Reading, MA: Addison-Wesley, 1998.
[11] ——, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 1st ed. Reading, MA: Addison-Wesley, 1973.
[12] A. Itai, “Optimal alphabetic trees,” *SIAM J. Comput.*, vol. 5, no. 1, pp. 9–18, Mar. 1976.
[13] M. T. Shing, “Optimum ordered bi-weighted binary trees,” *Inf. Processing Letters*, vol. 17, pp. 67–70, Aug. 1983.
[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 3rd ed. San Francisco, CA: Morgan Kaufmann Publishers, 2003.
[15] A. Fog, “The microarchitecture of Intel and AMD CPUs: An optimization guide for assembly programmers and compiler makers,” 2007, available from http://www.agner.org/optimize/.
[16] D. E. Knuth, *The Art of Computer Programming, Vol. 1, Fascicle 1 : MMIX – A RISC Computer for the New Millennium*. Addison-Wesley, 2005.
[17] “Intel® XScale™ microarchitecture technical summary,” Intel Corporation, available from http://www.intel.com/design/intelxscale/.
[18] “Performance of the ARM9TDMI™ and ARM9E-S™ cores compared to the ARM7TDMI™ core,” ARM Limited, available from http://www.arm.com/pdfs/comparison-arm7-arm9-v1.pdf.
[19] “ARM1176JZF-S™ technical reference manual,” ARM Limited, 2007, available from http://www.arm.com/pdfs/DDI0301F_arm1176jzfs_r0p6_trm.pdf.
[20] J. Abrahams, “Code and parse trees for lossless source encoding,” *Communications in Information and Systems*, vol. 1, no. 2, pp. 113–146, Apr. 2001.
[21] D. Altenkamp and K. Mehlhorn, “Codes: Unequal probabilities, unequal letter costs,” *J. ACM*, vol. 27, no. 3, pp. 412–427, July 1980.
[22] R. M. Krause, “Channels which transmit letters of unequal duration,” *Inf. Contr.*, vol. 5, no. 1, pp. 13–24, Mar. 1962.
[23] E. N. Gilbert and E. F. Moore, “Variable-length binary encodings,” *Bell Syst. Tech. J.*, vol. 38, pp. 933–967, July 1959.
[24] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sept. 1952.
[25] J. van Leeuwen, “On the construction of Huffman trees,” in *Proc. 3rd Int. Colloquium on Automata, Languages, and Programming*, July 1976, pp. 382–410.
[26] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997.
[27] R. M. Karp, “Minimum-redundancy coding for the discrete noiseless channel,” *IRE Trans. Inf. Theory*, vol. 7, no. 1, pp. 27–38, Jan. 1961.
[28] M. J. Golin and G. Rote, “A dynamic programming algorithm for constructing optimal prefix-free codes for unequal letter costs,” *IEEE Trans. Inf. Theory*, vol. IT-44, no. 5, pp. 1770–1781, Sept. 1998.
[29] P. G. Bradford, M. J. Golin, L. L. Larmore, and W. Rytter, “Optimal prefix-free codes for unequal letter costs: Dynamic programming with the Monge property,” *J. Algorithms*, vol. 42, no. 2, pp. 219–223, Feb. 2002.
[30] H. Vaishnav and M. Pedram, “Alphabetic trees—theory and applications in layout-driven logic synthesis,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 1, pp. 58–69, Jan. 2001.
[31] L. Gotlieb and D. Wood, “The construction of optimal multiway search trees and the monotonicity principle,” *Intern. J. Computer Maths, Section A*, vol. 9, no. 1, pp. 17–24, 1981.
[32] T. C. Hu, D. J. Kleitman, and J. K. Tamaki, “Binary trees optimum under various criteria,” *SIAM J. Appl. Math.*, vol. 37, no. 2, pp. 246–256, Apr. 1979.
[33] G. Cheung and S. McCanne, “An attribute grammar based framework for machine-dependent computational optimization of media processing algorithms,” in *Proc. 1999 International Conf. on Image Processing*, vol. 2, July 24–28, 1999, pp. 797–801.
[34] A. Turpin, Private communication, Nov. 2006.
[35] G. K. Zipf, “Relative frequency as a determinant of phonetic change,” *Harvard Studies in Classical Philology*, vol. 40, pp. 1–95, 1929.
[36] T. M. Cover, “Admissibility properties of Gilbert’s encoding for unknown source probabilities,” *IEEE Trans. Inf. Theory*, vol. IT-18, no. 1, pp. 216–217, Jan. 1972.
[37] “IA-32 Intel® architecture software developer’s manual volume 2A: Instruction set reference, A-M,” Intel Corporation, available from http://www.intel.com/design/pentium4/manuals/253666.htm.
[38] J. Heinrich, “MIPS R4000 microprocessor user’s manual,” MIPS Technologies, Inc., 1994, available from http://techpubs.sgi.com/library/manuals/2000/007-2489-001/pdf/.
[39] P. G. Hoel, S. C. Port, and C. J. Stone, *Introduction to Stochastic Processes*. Boston, MA: Houghton Mifflin Company, 1972.
[40] S. T. Pan, K. So, and J. T. Rahmeh, “Improving the accuracy of dynamic branch prediction using branch correlation,” in *Proc., Tenth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Oct. 1992, pp. 76–84.
[41] T.-Y. Yeh and Y. N. Patt, “Alternative implementations of two-level branch prediction,” in *Proc., 19th Annual Int. Symposium of Computer Architecture*, May 1992, pp. 124–134.
[42] S. McFarling and J. Hennessy, “Reducing the cost of branches,” in *Proc., 13th Annual Int. Symposium of Computer Architecture*, June 1986, pp. 396–403.
[43] J. Lee and A. Smith, “Branch prediction strategies and branch target buffer design,” *Computer*, vol. 17, no. 1, pp. 6–22, Jan. 1984.
[44] A. Andersson, “A note on searching in a binary search tree,” *Softw., Pract. Exper.*, vol. 21, no. 10, pp. 1125–1128, Oct. 1991.

[45] T. C. Hu and P. A. Tucker, “Optimal alphabetic trees for binary search,” *Inf. Processing Letters*, vol. 67, no. 3, pp. 137–140, Aug. 1998.
[46] T. C. Hu and M. T. Shing, *Combinatorial Algorithms*, 2nd ed. Mineola, NY: Dover Publications, 2002.
[47] S.-H. S. Huang and C. K. Wong, “Generalized binary split trees,” *Acta Informatica*, vol. 21, pp. 113–123, 1984.
[48] J. H. Hester, D. S. Hirschberg, S.-H. H. Huang, and C. K. Wong, “Faster construction of optimal binary split trees,” *J. Algorithms*, vol. 7, no. 3, pp. 412–424, Sept. 1986.
[49] D. A. Spuler, “Optimal search trees using two-way key comparisons,” *Acta Informatica*, vol. 31, no. 9, pp. 729–740, Nov. 1994.
[50] ——, “Optimal search trees using two-way key comparisons,” Ph.D. dissertation, James Cook University, 1994.
[51] R. Anderson, S. Kannan, H. Karloff, and R. E. Ladner, “Thresholds and optimal binary comparison search trees,” *J. Algorithms*, vol. 44, no. 2, pp. 338–358, Aug. 2002.