Dynamic Feature Selection for Hardware Prediction

Alan Fern, Robert Givan, Babak Falsafi, and T. N. Vijaykumar

School of Electrical & Computer Engineering
Purdue University
1285 EE Building
West Lafayette, IN 47907
{*afern, givan, babak, vijay*}*@ecn.purdue.edu*

Abstract

Most hardware predictors are table based (e.g. two-level branch predictors) and have exponential size growth in the number of input bits or features (e.g. previous branch outcomes). This growth severely limits the amount of predictive information that such predictors can use. To avoid exponential growth we introduce the idea of "dynamic feature selection" for building hardware predictors that can use a large amount of predictive information. Based on this idea, we design the dynamic decision tree (DDT) predictor, which exhibits only linear size growth in the number of features. Our initial evaluation, in branch prediction, shows that the general-purpose DDT, using only branch-history features, is comparable on average to conventional branch predictors, opening the door to practically using large numbers of additional features.

1 Introduction

Hardware prediction and speculation is a key technique to hide latency and improve performance in computer systems. Computer architects are exploiting predictive techniques in a variety of tasks such as branch prediction [28,29]; value prediction [19]; cache way prediction [7]; memory address [13,1], dependence [21], and sharing prediction [17,16]. In all cases, these hardware predictors capitalize on repetitive application behavior resulting in predictability in system event outcomes. By predicting such outcomes and thereby hiding the long latency of the corresponding system events, hardware predictors improve performance. For instance, value predictors rely on the repetitive occurrence of specific values in an application's execution to hide long read latencies in the memory system.

A hardware predictor is typically a finite state machine that, given its internal state and the values of some input bits, or *features*, outputs a prediction of the unknown value of a particular bit. For instance state-of-the-art branch predictors typically maintain some internal state such as a table of 2-bit saturating counters and use local history [29] (corresponding to prior outcomes of the same branch), global history [20,29] (corresponding to prior outcomes of any branch), branch program counters [22], or register values [25] as input features. Hybrid predictors may incorporate two or more input feature types [9].

The vast majority of previously proposed hardware predictors are table-based—for example, a gen-

eral mechanism that is a component of nearly all branch prediction schemes is the two-level table-based predictor introduced by Yeh & Patt [28]. Such table-based predictors store a counter for each possible combination of values of the input features. Each such counter generally tracks the most likely value for the bit being predicted given the particular combination of features that selects that counter. A fundamental and significant limitation of table-based prediction techniques is that *the size of a table-based predictor scales exponentially with the number of input features made available to it* (e.g., if a global history branch predictor uses n history bits as input features the size of the table used scales as 2^n). Due to the exponential scaling property of table-based predictors the number of input features and hence the predictive information that can be made available to such predictors is extremely limited. Table-based methods have been successful for predicting branches mainly because researchers have been able to identify small sets of features (e.g., recent global/local history) that lead to acceptable prediction accuracies.

In general, however, table-based predictors suffers significant drawbacks. While there may only be a few features relevant to any single prediction, the set of relevant features may change during the operation of the predictor—as a result, for some problems there may be no single small feature set that can be used to achieve high accuracy. For instance, in both value prediction and branch prediction, besides other branch outcomes (in the case of branches) and load addresses (in the case of values), outcomes may be correlated with instruction program counters, load addresses, register names, or other processor state [25] constituting a prohibitively large feature space for implementation in tables.[1]

Our work below enables hardware prediction to be applied to domains where no small set of relevant features can be determined in advance. In addition, our work opens up the possibility of extending current prediction domains like branch prediction to consider many more processor-state features. We believe that fully exploiting the potential for hardware prediction in support of new speculative techniques requires the development of effective alternatives to table-based predictors that do not scale exponentially with the number of input features considered during prediction. In Section 7 we discuss

1. Other processor state has been used by various table-based branch-prediction schemes but this is only possible with some form of information loss (such as using XOR to combine features) or hybrid scheme that requires committing in advance to particular feature groupings—the hybridized predictors use different features sets, where each set must be small. At any particular time one of the hybridized predictors is selected to make a prediction, with the selection based on its recent performance relative to the other predictors. This approach requires that the designer be able to identify multiple small subsets of features such that at least one of the subsets will lead to adequate performance most of the time. Such subsets need not exist and may be difficult to identify when they do exist.

the two recent alternatives [31,32] and contrast them with our approach.

In this paper, we propose the framework of *dynamic feature selection* to provide hardware prediction that scales linearly in size with the number of input features. To make a prediction, dynamic feature selection: (1) monitors a large number of features—large enough to include most or all information relevant for making predictions at any time, and (2) selects and stores information about *only the most relevant features from the larger feature set*. Such a predictor can monitor a very large set of features in step (1) without using the prohibitive amount of table space that traditional predictors would require, and only in step (2) be very selective in order to keep table sizes reasonable.

We then present and evaluate a particular predictor based on the dynamic feature selection framework. Our "dynamic decision tree" (DDT) predictor uses an on-line adaptation of *decision trees* from the machine learning research community to implement dynamic feature selection. By its design, our predictor has a storage requirement that grows reasonably as additional features are considered during prediction, allowing the prediction to be sensitive to many more features than is possible with table-based approaches. We argue that the lookup and update times for our predictor, as well as the logic overhead, is within the tight time and space constraints required for practical hardware prediction, and we thus expect that future hardware prediction applications can benefit from this technique.

As an initial evaluation of our predictor we perform experiments in the domain of branch prediction because of the easy availability of suitable benchmarks and high-quality alternative predictors for comparison. It is not our goal in this paper to show better performance than the best-known branch predictor but to introduce and evaluate a new framework of feature selection which enables predictors to consider large numbers of features in a systematic manner, adding to design options for future hardware prediction schemes. To evaluate our approach, we built a branch predictor which monitors 64 bits of history: the most recent 32 bits of both local and global history. We demonstrate that our approach can automatically select (without any built-in knowledge of which bits are local/global/recent/less recent) the most useful bits from these 64 features in order to compete successfully with table-based approaches. Interference-free simulation results on SPECint95 indicate that our predictor on average performs comparable to conventional interference-free GAp and PAp predictors with similar storage requirements and generally performs close to the better of GAp and PAp, while accuracies among GAp and PAp themselves vary significantly across applications. These results indicate that our approach is successfully selecting the relevant features from the large set provided. We note that similar gains and robustness rel-

ative to GAp and PAp can be achieved using a hybrid technique [9,20], but our technique does not require dividing the feature space by hand into global and local bits—in other prediction domains there may not be an obvious effective division of the feature space to exploit with a hybrid predictor. Also, our technique opens the door to branch prediction using additional processor state such as register values without any serious distraction from the more important history bit features, as the register bits would be automatically ignored when not relevant and automatically focussed upon when relevant.

Sections 2 and 3 introduce our terminology for prediction and motivate the dynamic feature selection approach in more detail. Section 4 contains the key technical contributions of the paper, describing previous machine learning work on decision trees and our novel on-line adaptation of that work to hardware prediction—here we define "dynamic decision tree". Section 5 argues that DDTs can be implemented under the necessary time and storage constraints. Section 6 gives our simulation results for DDTs in the branch prediction domain. Section 7 discusses related work.

2 Dynamic Prediction: Terminology and Problem Description

In a binary prediction problem the task is to predict the unknown value of a particular bit—this bit is called the *target bit*, and its eventual value is called the *target outcome*, or *outcome* for short. To aid in predicting the target outcome an *n*-bit vector is made available to the predictor—this vector is called a *feature vector* and generally conveys information about the target bit. For example, in the branch prediction domain the feature vector may consist of the last 8 bits of global branch-outcome history and the target bit corresponds to the future outcome of an unresolved branch. In this work, we are concerned with *dynamic* predictors—these are predictors that adjust their internal state over time to improve accuracy by better fitting the relationship between the feature vector and target outcome (which may be changing over time).

A dynamic hardware predictor is a finite state machine that has two modes of operation. In *prediction mode* the input is a feature vector and the output is a prediction of the target outcome. In *update mode* the input is a *feature vector/target outcome* pair and there is no output other than updated internal predictor state—intuitively, the new internal state should cause the predictor to be more likely to predict the given outcome in the future when encountering the given feature vector. For example in the domain of branch prediction, which we will frequently use as an example domain, prediction mode is used to speculatively resolve a conditional branch when it is encountered during prefetch, and update mode is

used to update the predictor tables when the branch is actually resolved.

3 Motivation for the Dynamic Feature Selection Approach

In this section we give an informal introduction to the idea of dynamic feature selection. Put simply, our goal is to enable hardware predictors to consider large numbers of features while making predictions. This is fundamentally impossible with current table-based hardware predictors because the storage requirement for these predictors grows exponentially with the number of features considered in making predictions.

For example, the GAp and PAp two-level branch predictors introduced in [29] assign a two-bit saturating counter to each possible branch feature vector. For example, the feature vectors used to describe branches in the GAp predictor contain n global history bits, requiring 2^n two-bit counters (one for each possible history). To the best of our knowledge, prior to the original report on our work [33], all branch predictors that perform as well as or better than the GAp and PAp schemes inherit this exponential growth property, which severely limits the number of features they can consider given fixed storage. This property is also inherited by other table-based hardware predictors (outside of branch prediction). We will say that predictors such as GAp and PAp that have counters for each possible feature vector are *table-based predictors*.

To understand the space savings we hope to derive over table-based predictors, consider an example prediction problem having feature vectors and target with the following properties:

- when features 1 and 2 are true, feature 3 equals the target outcome, and
- otherwise if feature 1 is true then feature 4 equals the target outcome, and
- otherwise if feature 1 is false then feature 5 equals the target outcome.

Note that for a table-based predictor to learn this pattern, it will have to allocate 32 counters based on all combinations of values for features 1 through 5. However, from the properties given, it is clear that only one feature out of features 3, 4, and 5 is relevant for any one prediction. We are trying to avoid considering all combinations of features 3, 4, and 5 for such prediction instances. Our approach is to consider one feature at a time, and select the most predictive single feature—we then use that feature to divide the prediction problem into two subproblems: one for the feature vectors with that feature false and one for those with that feature true. Each of these subproblems can then be addressed in the same

manner (i.e. selecting the next most predictive feature). In the branch prediction domain the results we present below show that for SPECint95 benchmarks, selecting just *one* history bit (one feature) dynamically and predicting with just that bit can often outperform GAp or PAp style predictors that use 2^{16} counters.

Now consider the same example just described, but where there are 5 additional features involved in the problem (*i.e.*, useful perhaps when the target characteristics change) which are not useful in predicting the current target. Table-based predictors have no mechanism for dynamically detecting which features are useful directly, and would need to have 2^5 times as many counters due to the 5 irrelevant features. In contrast, our approach will dynamically select one bit at a time, and will essentially just ignore the irrelevant features (they are considered each time a single feature is selected as useful, but combinations of values of the irrelevant features are never considered—as a result the irrelevant features have a linear rather than exponential effect on the storage required).

Why do we expect that we can dynamically select a small number of features from a large set and still obtain good prediction accuracy? Apart from abstract examples like that just described, in practice this phenomenon has been observed in the branch prediction domain. In [10] a small number of features were *statically* selected from a larger feature set on a per-branch basis (off-line) and only the selected features were used by a predictor assigned to each branch. It was shown that for most benchmarks the accuracies of the predictors that only used the smaller but carefully selected feature sets were close to those of predictors that used the entire set of features. A predictor that forms rules based on the principle of utilizing only a small set of carefully selected features is said to perform *feature selection*. Previous research has given no method for performing feature selection dynamically. Through the use of *dynamic* feature selection, our predictor avoids exponential growth in size relative to the number of features considered.

In the next section we use the dynamic feature selection approach to design one specific type of predictor, based on a decision tree method. Decision trees are widely used for prediction tasks in the field of machine learning [34] and provide a natural way to incorporate feature selection into prediction.

4 The Dynamic Decision Tree Predictor

This section describes the dynamic decision tree (DDT) predictor. First, we discuss decision trees as a general means of representing predictors. Second, we discuss the main component of the dynamic deci-

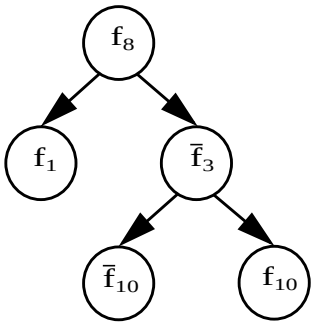
sion tree, the correlation predictor. Finally, we give the structure and function of the DDT predictor.

4.1 Decision Trees

Table-based predictors learn and store a prediction for each of the exponentially many possible feature vectors (i.e. each possible combination of feature values). Rather than store a separate prediction for each feature vector, we can avoid exponential growth by partitioning the set of feature vectors into a small number of subsets and then only store a single prediction for each subset. Given such a partition, the prediction for a feature vector is the prediction stored for the subset that the vector is in. Clearly, this method performs best when a high percentage of the vectors in each subset have the same most likely outcome—in this case, using that likely outcome as the prediction for that subset results in a high accuracy. Doing this we can avoid providing resources for learning/storing a separate outcome for each feature vector. Instead, resources must be provided for dynamically selecting a partition of possible feature vectors into "good" subsets, for determining which subset a feature vector falls into, and for learning an outcome for each subset. By providing space and time efficient methods for these tasks, we can define predictors that can consider a much larger number of features than is currently possible. For these reasons we consider a new class of hardware predictors, based on binary decision trees.

A *decision tree* is a binary tree where each node is labelled with a feature or its negation. The *depth* of a decision tree node is the number of arcs traversed on the path from the root to the node. The depth of a tree is the depth of its deepest node. An example decision tree is shown in Figure 1—this tree has a depth of 2.

Making Predictions with a Decision Tree. The label of a tree node is referred to as the *test* of the node. This test, evaluated with respect to a given feature vector, is equal to either 0 or 1, according to whether that feature is false or true in the feature vector. For example, the test of the root node in Figure 1 evaluates to the value of feature f_8, which might be global history bit 8 in a branch prediction domain. Likewise the test of the root's left child evaluates to the negation of the value of feature f_1. Given a binary decision tree and a feature vector, the prediction made by the tree is defined recursively as follows. If the root node is a leaf node (*i.e.*, the tree is a single node), then return as the predicted outcome the value of the test at the root node for the given feature vector. Otherwise, evaluate the test at the root node for the feature vector, and return the prediction made by either the left child tree or right child tree of the node, depending on whether the test evaluates to 0 or 1, respectively. For example, when the



f_1 - local history bit 4
f_8 - global history bit 3
f_3 - global history bit 3
f_{10} - local history bit 1

Corresponding prediction rules:

IF ($\overline{f_8}$) THEN Predict(f_1)
IF (f_8 AND $\overline{f_3}$) THEN Predict(f_{10})
IF (f_8 AND f_3) THEN Predict($\overline{f_{10}}$)

Figure 1: Example of a 5 node, depth 2 binary decision tree and its corresponding prediction rules.

tree in Figure 1 is presented with a feature vector such that $f_8 = 1$, $f_3 = 0$, and $f_{10} = 1$, a path is followed from the root through the node labeled $\overline{f_3}$ to the leaf node labelled f_{10}. The final prediction for this feature vector is 1 since that is the value of f_{10} (perhaps local history bit 1 for a branch prediction domain).

A decision tree can be viewed as representing a set of prediction rules such that there is one rule for each leaf. Any feature vector will activate exactly one rule from this set of rules. The prediction rule corresponding to each tree leaf can be written as an IF/THEN rule, as shown in Figure 1.

Advantages of Decision Trees. Rules formed by a decision tree are not required to utilize all the available features but instead can select a small number of features to use while ignoring features that are less predictive. Doing so, a decision tree in theory has the ability to form highly predictive rules while at the same time avoiding any exponential growth in predictor size as the number of history/state bits considered increases. The size of a decision tree, however, does grow exponentially with the maximum depth of the tree. This depth determines the maximum number of features that can occur in any single predictive rule (like those in Figure 1). Decision trees are advantageous in domains where high accuracy can be achieved by small rules built from a large feature space. We argued in Section 3 that branch prediction is one such a domain, it is reasonable to expect other speculative prediction domains to have this property.

It may appear that decision trees are similar to context trees, which have been previously used for branch prediction [11] and are general enough to be applied to other hardware prediction problems; however, a decision tree can capture a far greater variety of predictive rules than a context tree of the same depth, and context trees exhibit the same exponential growth in the number of features considered that table-based predictors show. See Section 7 for a more complete discussion of context trees.

Training Decision Trees. Most known methods of training decision trees are batch training methods rather than dynamic ones (for example ID3 [23], CARTE [2]). These methods use a training set of feature-vector/target-outcome pairs to construct the decision tree off-line. The resulting tree is then used to make predictions in its intended domain. Though it does not seem practical to perform batch learning in hardware prediction domains, due to time and space constraints, a basic understanding of the principles employed by batch learning methods is helpful for understanding our novel dynamic training method, described in the following sections.

Most batch training algorithms recursively grow a tree from the top down using greedy heuristics to select tests at each decision node. First, the entire training data set is analyzed and a heuristic is used to select a test that is judged to aid the prediction task most; this test is associated with the root decision node. Then, *based on this test*, the training data is partitioned into two smaller left and right data sets. These sets are used as training data to recursively grow left and right subtrees. The recursion continues until a stopping criterion is met. Methods differ in the test selection heuristic and the stopping criterion used. Intuitively a test selection heuristic should select a test that partitions a given data set into two more predictable sets. The worst possible test to select is one that divides a data set such that each partition contains an equal number of examples with 1 and 0 target outcomes. The best test divides a data set such that all of the examples belonging to each partition block have the same target outcome. Some theoretical and empirical investigations of test selection heuristics appear in [3], [4], [5].

For our initial work presented here we have focused on using immediate predictive accuracy as the metric for our test selection heuristic. This means the selected test at a node is the one that is estimated to have the highest accuracy if used to directly predict the outcomes of the data set examples. A main reason for selecting this metric is because of the ease with which fast on-line estimators of accuracy can be constructed.

To use decision trees in hardware prediction domains, we propose a novel method of dynamically training decision trees rather than use batch-training methods. The following two sections describe our proposed dynamic decision tree. We first describe the correlation feature selector that will be used as the accuracy-based test-selection mechanism at each decision tree node. Next, we describe the tree architecture and the computations involved in training and making predictions with the tree.

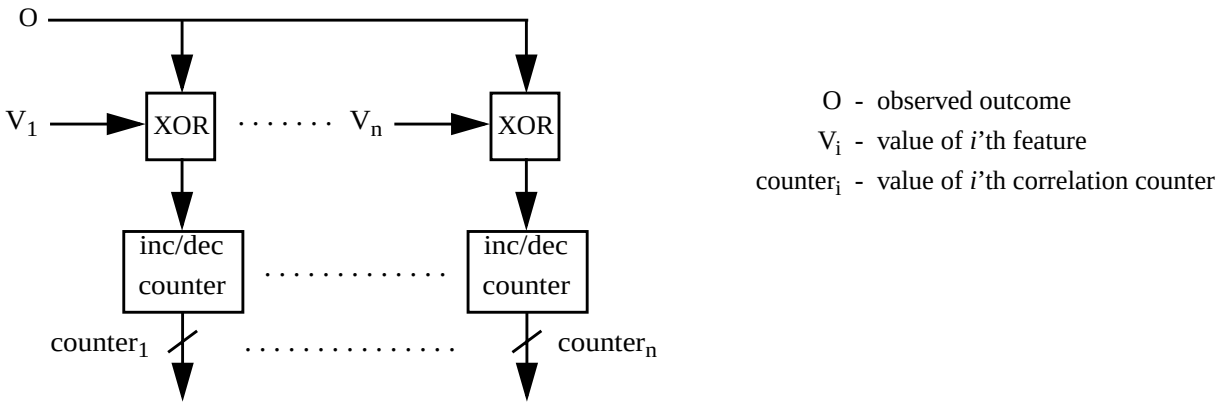


Figure 2: After observing outcome *O* the value of each feature is XOR'ed with *O* and the results are used to increment or decrement the counter value of each feature.

4.2 The Correlation Feature Selector

The correlation feature selector is a mechanism for selecting a "most predictive" single feature from a large set of candidate features (*e.g.,* 32 local and 32 global history bits), given repeated observation of feature vector/target outcome pairs. After observing each pair, the correlation feature selector updates information stored about each feature in order to be able to select the best feature at any time.

Given a set of features, a correlation feature selector associates a signed counter to each feature in the set. We use both the sign and the magnitude of the counters in our predictor. Our intention is to update the values so that after encountering of a large number of feature/outcome pairs, a large counter magnitude for a feature will indicate that feature is strongly correlated with the outcome (either positively or negatively, according to the sign of the counter). Upon observing a feature vector/target outcome pair, the update method increments each counter for a feature which agrees with the target outcome, and decrements the remaining counters. Figure 2 shows a diagram depicting this computation. After observing a long sequence of feature vector/target outcome pairs, the magnitude of the counter value for a given feature is proportional to the number of correct predictions that would have been made by using the better of that feature or its complement to directly predict the outcome.[2]

At various points in the dynamic decision tree algorithm described below, we wish to select the most predictive feature amongst various subsets of the entire feature space. Given candidate features

2. Let $C(f)$ be the number of correct predictions made by f after observing T instances since initialization. We know that $C(f) + C(\bar{f}) = T$ and f's counter V_f is $C(f) - C(\bar{f})$ and from these we can get that $\text{Max}(C(f), C(\bar{f})) = (T + |V_f|)/2$.

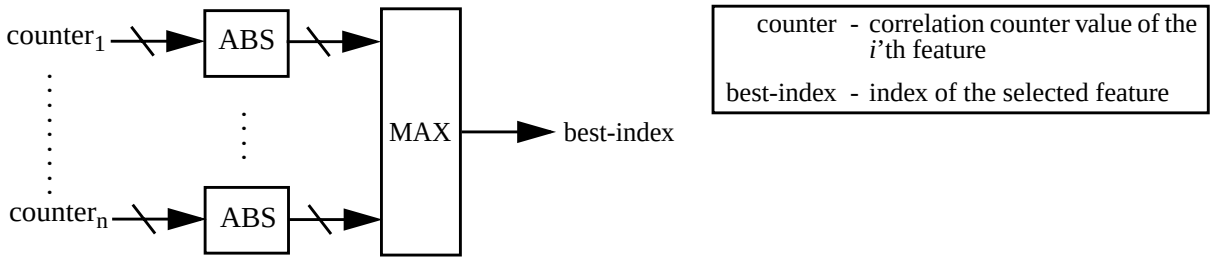


Figure 3: Computing the selected feature given the counter values of the features.

$f_1, \ldots, f_n$, we select as the *selected feature* the feature with the highest magnitude counter, breaking ties by selecting the feature of lower index. Figure 3 shows a diagram depicting this computation. We provide a complete formal definition of the correlation feature selector in the Appendix.

The counters in a correlation feature selector are, of course, finite precision, and therefore will eventually saturate. However, unlike the counters in table-based predictors, these counters are used by *comparing* them to each other to select a maximum value (table-based predictors make decisions based only on the sign of the counter rather than its magnitude relative to other counters). For this reason, allowing the counters to saturate may lose important relative information. Instead, whenever any counter in a correlation feature selector would saturate, we halve the magnitude of all its counters.

4.3 The Dynamic Decision Tree

We describe the data and functionality at each DDT node, and then how nodes are combined into trees. Please note that this description concentrates on conveying the key concepts and functionality rather than presenting the actual implementation. Approaches to efficient concrete implementation are discussed and analyzed in Section 5. Also note that our description here is somewhat informal—a complete formal definition of a DDT is presented in the Appendix for reference.

4.3.1 Nodes of a Dynamic Decision Tree

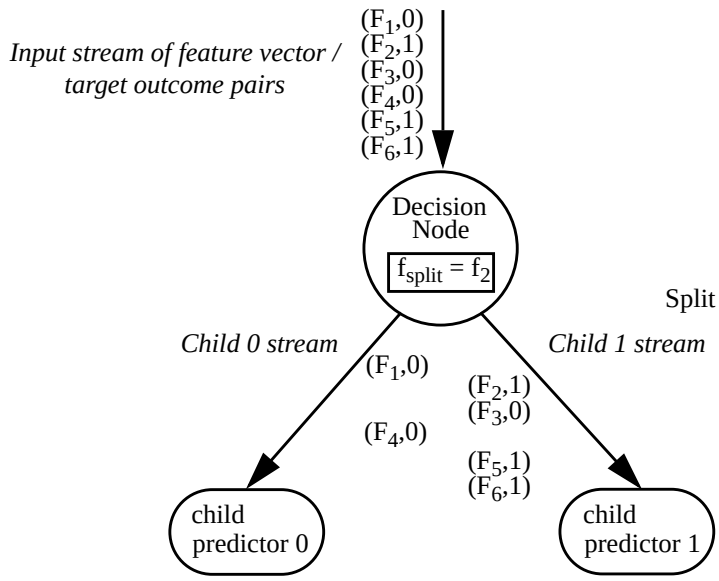
There are two types of DDT node: leaf nodes, which have no children, and internal nodes, which each have two children that are predictors themselves (typically other tree nodes but generally any form of predictor). The two node types have very similar functionality. We describe the internal nodes first, and then indicate which aspects are omitted for leaf nodes.

Each node has two modes of operation: a prediction mode and an update mode. In the prediction mode, the node receives a feature vector and must make an outcome prediction quickly. In the update

mode, the node receives a feature vector/outcome pair, and must update its internal state to improve future predictions based on the information in the pair. The essential internal state of each node is a correlation feature selector defined over the candidate features along with two extra features described below.

A critical part of the function of each internal node is in managing/combining the two child predictors in order to make a good prediction for the stream of feature vector/target outcome pairs seen by the node. The role of each node as "manager" involves splitting this stream of pairs into two substreams according to the feature currently measured to be most predictive. Each of these two substreams is then passed to one of the child predictors—the intention is that each substream will be more easily predicted (typically the target outcomes will be more uniform) than the original stream because the information in the most predictive feature has been used to split the stream. The "managing" node also tracks the performance of the child predictors and may choose to neglect their "recommended prediction" in favor of a prediction based on a single feature if that prediction is outperforming the child predictors. This performance tracking is efficiently implemented by adding a single feature to the correlation feature selector at the node—this feature represents the child predictor's recommendation, and the correlation selector automatically estimates its accuracy and compares it to the accuracies of the alternative single feature predictions. For each input feature vector, we call the most predictive feature (as measured by the correlation feature selector) the *split feature* f_{split}, and we call the child predictor selected by the most predictive feature the *selected child*. Figure 4 shows a node splitting a stream of feature vector/target outcome pairs between its children. Other predictors such as the Bi-Mode branch predictor [18] use the idea of splitting the original branch stream into substreams, however, the stream splitting is based on a fixed set of features (PC bits) and not dynamically tuned to the incoming stream.

Data at Each Node of the Dynamic Decision Tree. At each node, there is a correlation feature selector that includes counters for the candidate features as well as two "extra" features which play an important role in the predictor. The first extra feature is called the *constant feature* (denoted f_c), whose value is always 1. Each node's feature selector has a counter for the constant feature, which evaluates the option at that node to always return the same prediction (1 or 0 based on the sign of the correlation). This counter is important for recognizing when the input stream of feature vectors to a node has nearly uniform target outcomes. The second extra feature is the *subtree feature* (denoted f_{sub}), which is equal



features

Vectors	f_1	f_2	…	f_n	Outcome
F_1	—	0	…	—	0
F_2	—	1	…	—	1
F_3	—	1	…	—	0
F_4	—	0	…	—	0
F_5	—	1	…	—	1
F_6	—	1	…	—	1

Split feature (f_2) values and target outcomes for each pair.

NOTE: Substreams arriving at each child have more uniform target outcomes than the original input stream for the decision node. The child streams are not perfectly uniform because f_2 is not a perfect predictor of the target outcome.

Figure 4: The decision node receives a stream of feature vector/target outcome pairs and divides the pairs between the two child predictors based on the value of its split feature (in this case f_2). The table in the upper right shows the value of the split feature f_2 for each of the six feature vectors in the input stream along with the corresponding target outcomes. A feature vector is sent to the right child predictor if the value of the spit feature is 1 and sent to the left predictor otherwise. The goal of the decision node is to split the input stream into two more predictable streams for the child predictors. Thus, the split feature is selected to be the feature that is most correlated with the target outcome.

to the outcome predicted by the selected child predictor. The counter corresponding to the subtree feature evaluates how good it is to make predictions based on the output of the selected child predictor. This feature is not used for prediction, but is important during learning for determining whether it is better to predict locally at this node based on a single feature or to use the prediction of one of the children.[3]

Figure 5 shows the features used by the correlation feature selector of each internal node as well as the computations to select the child prediction that is used to update f_{sub}. A child is selected according to the output of the xor gate, which represents either f_{split} or $\bar{f}_{split}$ depending on the sign of the correlation between f_{split} and the target outcome. Thus, the sign of the correlation determines whether the tree node is labelled by f_{split} or $\bar{f}_{split}$. The effect of the xor gate is to select child 1 (child 0) when we would predict 1 (predict 0) based on only the value of the split feature and its correlation value. This has the

3. The correlation selector's counter for f_{sub} does not measure the accuracy of either child, but the accuracy of the weighted combination of the children over the entire stream of feature vector/target outcome pairs seen by the parent node. That is, for each feature vector/target output pair only the selected child effects the counter of f_{sub}.

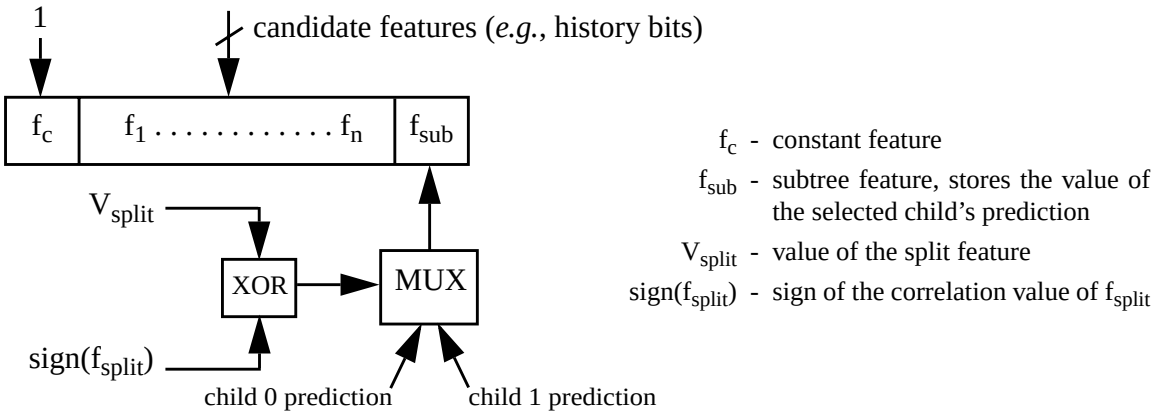


Figure 5: The features of the correlation predictor for internal nodes and computations for child selection. The inclusion of the xor gate allows a node to be labeled by either a feature or its negation. That is, if sign(f_{split}) is positive then the node's label is f_{split} and child 1 is selected when f_{split} is 1 (otherwise child 0 is selected). Similarly if sign(f_{split}) is negative then the node's label is $\bar{f}_{split}$ and child 1 is selected when f_{split} is 0 (otherwise child 0 is selected).

practically useful effect of letting predictor 1 (predictor 0) focus on predicting streams that are biased toward outcomes of 1 (outcomes of 0).

Additionally, each node stores *summary information* about the counters to enable quick predictions without accessing the counter values. The summary information includes *best-index* which is the index of f_{split}, the correlation counter signs of f_c and f_{split}, and bits *use-sub?* and *use-f_c?* indicating whether the prediction should be made using a feature or using the prediction of the selected child.

Operations at each Node of the Dynamic Decision Tree. In prediction mode, a node consults the summary information (generated by the previous update phase) to determine whether to predict directly with the currently most predictive feature or to instead use that feature to select a child predictor with which to make the prediction. Figure 6 shows the computations performed during the prediction phase. The top-most MUX uses the summary information *use-f_c?* and *use-sub?* to determine whether to make a prediction with the constant feature (*use-f_c?* is true), the selected child (*use-sub?* is true), or the split feature directly. Note that the purpose of the XOR operations is to select either the value or its negation based on whether the correlation counter indicates a positive or negative correlation. The bottom-most MUX corresponds to the MUX in Figure 5 and selects which of the two children to use based on the value of the split feature, V_{split}, and the sign of its correlation.

In update mode, a node updates its correlation feature selector based on the feature values and target outcome of the current feature vector/target outcome pair. Note that the feature values used here include

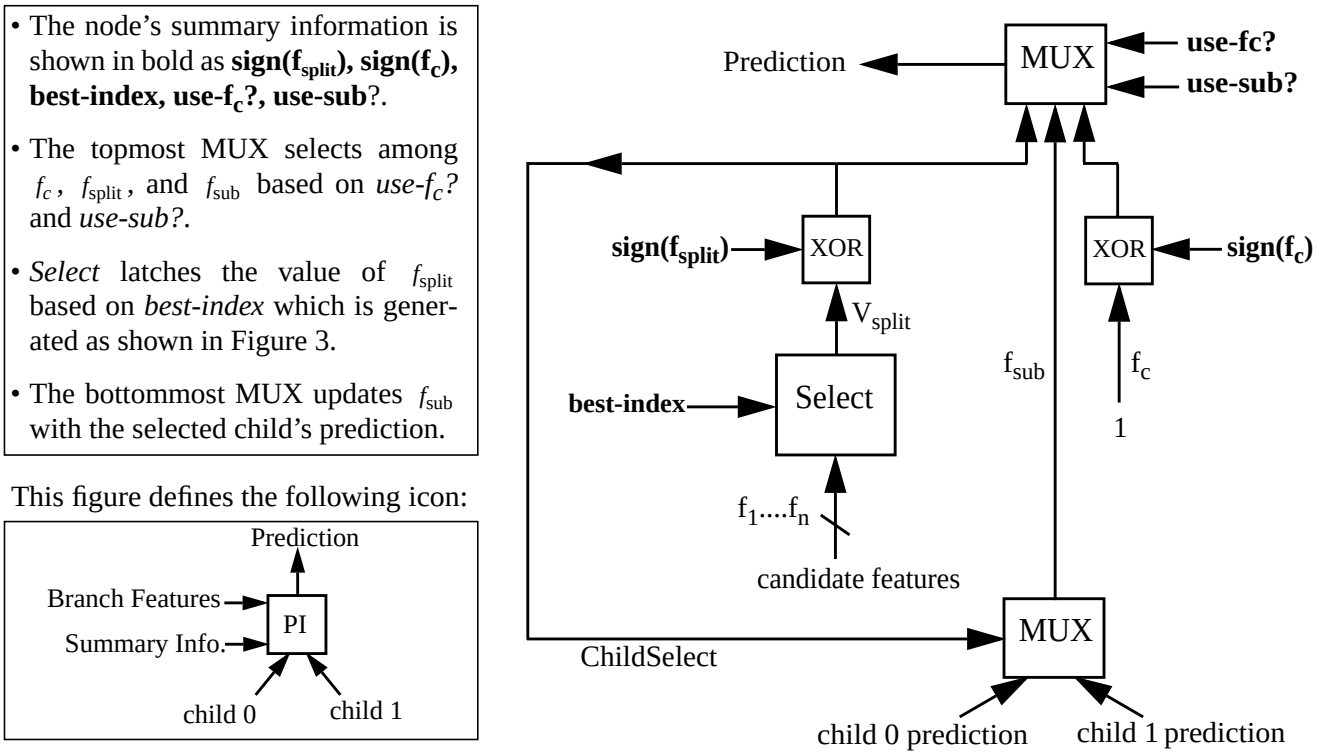


Figure 6: Prediction Phase of an Internal DDT Node (abbreviated PI).

the constant feature f_c and the child-based feature f_{sub} available from the preceding prediction phase. In addition to updating its own correlation selector, the node must activate the selected child predictor so that the child will also update its feature selector.[4] Note that only *one* child needs to update its selector (*i.e.*, to learn) because only one child is expected to be used in predicting each instance—we update the selector in the "selected child" since that is the child which would be considered during prediction. This update occurs even if the current prediction is being made by a single feature, neglecting the child's recommendation.[5] After the correlation feature selector of a node has been updated the summary information stored at the node is updated based on the new correlation values.[6] Finally, note that update can occur at all nodes in parallel *along the path formed by the sequence of selected children from the root down*, but must not occur at the other nodes.

We have now completed the description of internal nodes. Leaf nodes behave identically to internal nodes except they do not have child predictors and therefore do not use or maintain an f_{sub} feature. We

4. If the child predictor is not another tree node, but some other kind of predictor, this activation signals that predictor to update whatever state it maintains. In our implementation, the child predictor is always another tree node.
5. In this case, the update allows the child predictor to improve by learning so that eventually it may be better than the currently preferred single feature.

provide a complete formal definition of the dynamic decision tree in the Appendix for reference.

4.3.2 Combining the Nodes into a Dynamic Decision Tree.

Internal and leaf nodes of the above design are connected into a full binary tree to form the dynamic decision tree hardware predictor. The connection of these elements into a tree does not alter their basic operation, except in one respect: we restrict each node so that the "most predictive feature" f_{split} selected during update cannot be the same as that selected by any ancestor of that node in the tree. This restriction prevents the node from splitting on a feature that has already been used earlier in the tree (such a split would result in one of the child predictors at that node seeing an empty input stream).

5 Predictor Implementation Issues

In Section 4 we presented an abstract description of dynamic decision trees and how they work. In this section, we present a discussion of real on-chip implementation issues for a DDT predictor. The key to a realistic implementation of DDTs for the purpose of hardware prediction is to organize the information collected in the trees in the form of tables that occupy moderate on-chip real estate and can be accessed quickly. Tables offer the advantages of simplicity as well as the existing expertise in implementing and optimizing prediction structures as tables.

Many hardware prediction tasks involve numerous separate prediction problems using the same feature space; where each prediction problem ideally needs a separate hardware predictor. For example, branch prediction involves many prediction problems over the same feature space of history bits—each problem involves only dynamic occurrences of a single static branch. Efficient DDT implementation depends critically on sharing logic between many DDT predictors over the same feature space—in branch prediction, this means a DDT predictor will have only one logic array shared by predictors for all static branches (table-based predictors do similar but less critical sharing of the logic used to index into the table). Aliasing between separate problems (such as occurs in typical branch predictor imple-

6. One final practical point: we believe it is important to minimize the frequency with which the value of the split feature f_{split} changes, because such changes can dramatically alter the streams of data seen by the child predictors. For this reason, when updating f_{split} we implement a preference for keeping f_{split} unchanged in the case of tied correlation values, even if a feature with a smaller index is involved in the tie. Additionally, when comparing correlation values the 'goodness' of f_{split} is measured by the correlation value of f_{sub}. Thus, the split feature will only be changed if another feature is performing at least as well as the subtrees given the current f_{split}. In the long run, we believe it will be important to implement an even stronger preference to sticking with the current split feature to avoid oscillation/vacillation between features of similar predictive utility. We have not addressed this issue further in this work, however.

mentations) complicates this picture only slightly and we can still share the DDT logic between all the predictors.

Because typically only one tree (or in the case of wide-issue superscalar machines, a small number of trees) is used to predict or learn at a time, all the logic can be shared among many similar problems, amortizing the implementation cost of the logic. Such an organization lends itself well to being implemented using tables by grouping the information stored at a particular node of each tree into a table. Each table is then accessed by a *predictor hash function* (PHF), which for branch prediction is a lower-order prefix of PC bits from the branch being predicted. The PHF is used to select the tree information corresponding to the current problem from the tables.

The critical information stored at each node in a DDT is the correlation feature selector, which consists of one saturating counter for each feature. In table form, this information is stored in a *correlation table*, a two-dimensional table of correlation feature selectors indexed on one dimension by PHF value and on the other dimension by tree node. Thus, the row corresponding to a given PHF value consists of the correlation feature selectors for all tree nodes used for the corresponding predictor—given the PHF value, the table can then produce all the relevant correlation feature selectors for processing by the DDT logic.

The values stored in the correlation table are updated in the update phase of the predictor. However, to expedite the time-critical prediction phase, we can additionally store summary information in a separate table (as described in Section 4, and including for example the identity of the “splitting” feature for each DDT node) for each problem—this summary information amounts to a small number of bits per tree node, and is kept in a separate much smaller table, called the *prediction table*, for the entire DDT, again indexed by PHF and by tree node. Figure 7 shows the two tables and their contents.

In prediction mode, the PHF and a vector of the feature values are sent to the DDT. The prediction table is accessed with the PHF, and a row containing the prediction summary information for each node is read out. The summary information at every node is used in conjunction with the feature vector to select the decision to be made at that node: either to predict based on a single feature or to pass on the prediction of the appropriate child, as shown earlier in Figure 6. Although this selection can be done in parallel at all the nodes, the overall prediction comes from combining the decisions at all nodes to select a path through the tree, a process sequential in the depth of the tree. Because the selection logic is

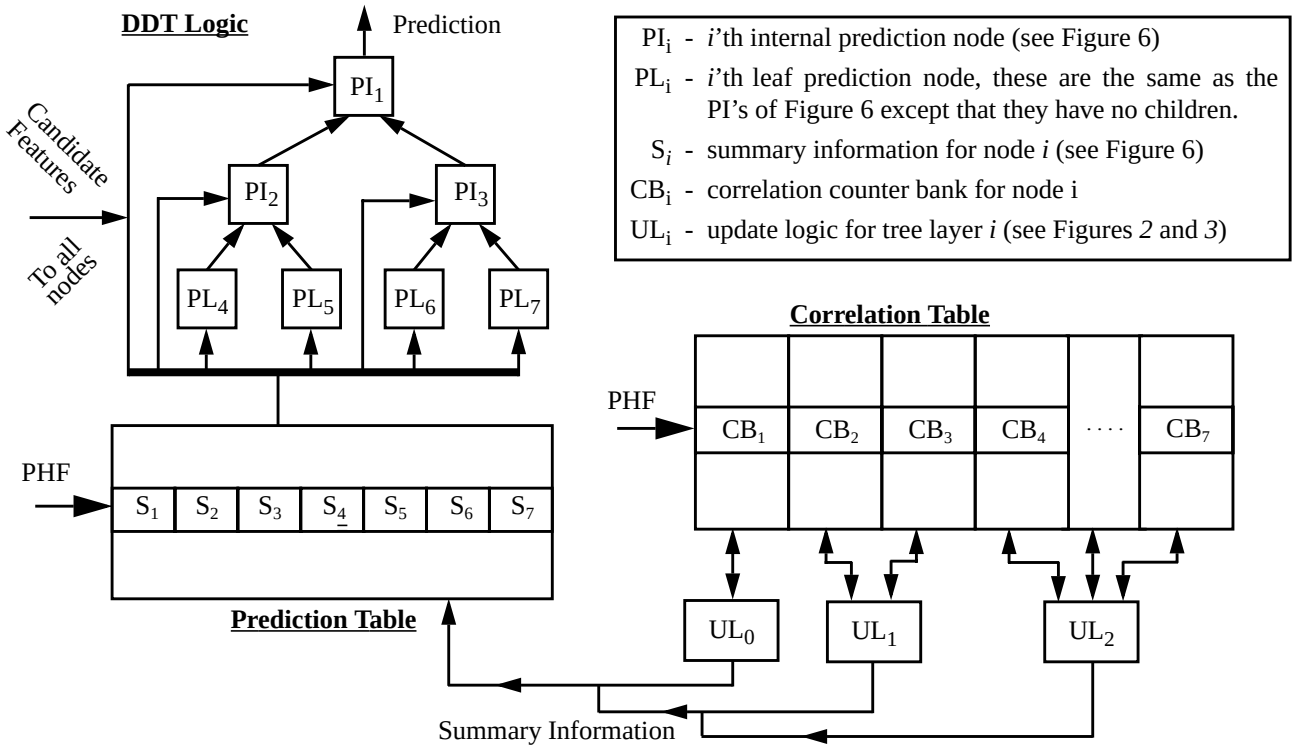


Figure 7: Overview of the DDT hardware predictor showing the connections between the DDT logic, the correlation and prediction tables, and the update logic. Each table is accessed by a predictor hash function (PHF) that selects one predictor from the table.

essentially a multiplexor that uses the summary information to choose one feature, its implementation may be moderately space expensive. To alleviate this problem, the implementation cost is amortized over all the predictors using the same feature space as this logic is reused repeatedly.

The prediction process can be viewed as reading parameters for the DDT logic from the prediction table followed by a single parallel decision operation at all nodes based on the parameters and features, finished off by returning a prediction from a tree leaf along the selected sequential path through the tree. Once the parallel decision operations have occurred, the signals coming from the prediction table to the DDT logic define a single path from the root of the tree to one leaf—it is along this path of *activated nodes* (or some prefix of it) that the prediction flows back to the root. As shown in Figure 7 above the prediction table, the DDT logic, which inherits the topology of the DDT, performs the parallel decision operations at all nodes and returns the prediction. The only operation that uses time proportional to the depth of the tree is the flow of the prediction up the tree, after the nodes have selected their respective decision in parallel. The depth of the tree is at most 7 in our work to date, so this process can be completed within a small number of gate delays. The prediction process identifies a path of activated nodes

from a leaf to the root along which the prediction is returned. This path is used below during update.

When a target outcome is resolved, the decision tree corresponding to the prediction instance needs to be updated by executing an update phase. To that effect, the correlation table is accessed with the PHF, returning the row of correlation feature selectors constituting the DDT predictor state for that PHF value. The resulting row is updated using the feature vector of feature values for the current prediction instance, and then stored back in the correlation table. In particular, during this update, the split feature (and other summary information, similarly) at every node of the tree in parallel is computed by identifying the feature with the maximum counter magnitude, as shown earlier in Figure 3. Because the 'split feature' at one or more nodes may have changed due to the updating of the correlation table, the prediction table now has to be updated with new summary information as shown by the update paths at the bottom of Figure 7—this update ensures that future prediction will adapt by incorporating the changed split feature.

The logic required to compute the maximum among the counters of all the features is perhaps the most expensive overhead of our technique. Much as the selection logic in prediction mode is shared among all the predictors using the same feature space, the logic for the maximum computation is also shared and need only be replicated as many times as the depth of the tree. Even that replication can be avoided by performing the maximum-computation sequentially one level of the tree at a time if desired. While we believe this can typically be computed within a problems space and time constraints either in parallel or in sequence along the activated path, we also believe that approximations to the correlating feature selector along the lines described in [9] can reduce the costs of this computation with little loss in accuracy. In that work, the task was to dynamically select the best of a set of predictors—this task is closely related to the task of dynamically selecting the highest correlated feature.

5.1 Table Size Estimates

We estimate the size of a correlation table entry and a prediction table entry, as a function of the tree depth (denoted as d), the number of features monitored in the nodes (denoted as n), and the number of bits in the feature counters (denoted as b). The size of each correlation table entry, therefore, is:

$$(\#\text{ leaves})[\text{bits per leaf}] + (\#\text{ internal nodes})[\text{bits per internal node}]$$
$$= 2^d[(\#\text{counters/leaf})\,b] + (2^d - 1)[(\#\text{counters/internal})\,b] = 2^d[(n+1)\,b] + (2^d - 1)[(n+2)\,b]$$

In the leaf nodes, there are only n+1 counters: f_c, f_1,..., f_n. In the internal nodes, there is one more

counter, f_{sub}, that estimates the selected child's outcome correlation.

The size of each prediction table entry is given by:

$$(\#\text{ leaves})[\text{bits per leaf}] + (\#\text{ internal nodes})[\text{bits per internal node}]$$
$$= 2^d[\#\text{bits summary/leaf}] + (2^d - 1)[\ \#\text{bits summary/internal}] = 2^d[n + 2] + (2^d - 1)[n + 4]$$

Each leaf contains n bits of summary information to keep track of the decisive feature, one bit for whether we are using the split feature for prediction or f_c, and one more bit for the sign of the feature being used for prediction. Note that we use n bits, instead of $\log n$ bits, to track the split feature to avoid encoding/decoding delays in selecting the decision suggested by the summary information at every node during the prediction phase. The prediction table entry is much smaller than the correlation table entry (by about a factor of b) enabling fast access during the time-critical prediction phase.

6 Experiments

To demonstrate the DDT's effectiveness in selecting features dynamically, we use the DDT for branch prediction and compare to table-based predictors. This paper is a first step towards evaluating *dynamic feature selection* using the DDT. As such, we compare an interference-free version of our DDT predictor against interference-free versions of GAp and PAp.[7] We note that it is not our immediate goal here to beat the state-of-the-art in branch prediction, but rather to demonstrate that our general-purpose DDT predictor can effectively select the relevant feature bits from a large set in order to compete with highly-specialized branch predictors. We intend future research to determine how to best use this selection capability both in branch prediction and in other prediction domains.

We varied the key parameter controlling the predictor size (tree depth for DDT, and number of history bits used for GAp and PAp), and compared the accuracy obtained by the three predictors for each fixed size (here "size" is taken to be the number of state bits stored in tables for the predictor — we argued in Section 5 that the logic required could be shared across many branch instances). In these experiments our predictor has available 64 history bits (32 global and 32 local bits) without any knowledge of which bits are more recent history bits or which bits are global/local. For instance, before seeing dynamic branch instances, our DDT is just as likely to select the 32nd history bit as the first for use in prediction. In spite of this lack of built-in knowledge about the feature space, our results show that

7. By interference-free we mean that each static branch is assigned its own DDT/GAp/PAp predictor.

the DDT predictor is able to automatically select useful bits from the 64 bits. These results substantially support our claim that DDT predictors can automatically select useful state bits to make predictions, even in domains where the architect does not know ahead of time which bits these will be (or even where the useful bits vary over time so that no fixed set will provide good prediction).

We carried out all simulations using trace-driven inputs generated by SimpleScalar 2.0 [6] run on eight integer benchmarks from the SPECint95 suite. This was the same suite used in [10], which allows us to draw some conclusions below based on their results. All benchmarks were simulated until completion and Table 1 shows the inputs and the number of static and dynamic branches for each of the benchmarks used. All predictor counters were updated directly after each prediction based on the true branch outcome. It was demonstrated in [30] that this counter update method as opposed to speculative updates or updating only after a branch has been resolved does not significantly impact the resulting accuracies.

Benchmark	Input	Static Branches	Dynamic Branches	Benchmark	Input	Static Branches	Dynamic Branches
compress	test.in	486	3889933	**li**	text.lsp	709	148707928
gcc	cccp.i	19555	195124643	**m88ksim**	ctl.raw	1400	92366311
go	2stone9.in	5215	106288644	**perl**	jumble.in	2087	332004699
ijpeg	vigo.ppm	1725	98664499	**vortex**	vortex.raw	6822	299718894

Table 1. Inputs and number of dynamic and static branches for SPECint95 benchmarks.

The GAp and PAp predictors utilized two-bit saturating counters and were simulated using history lengths ranging from 8 to 16 bits yielding predictor sizes ranging from 0.5 to 128 Kbits.[8] The dynamic decision tree was given the feature set LG that consists of 32 local and 32 global history bits for a total of 64 binary features.[9] Eight-bit signed correlation counters were used[10] and the tree depths were varied from 0 to 7 yielding predictor sizes ranging from 0.57 to 147.17 Kbits (see Section 5.1).

6.1 Results and Analysis

We conducted experiments for interference-free PAp and GAp predictors using history lengths of 8, 10,

8. Note that by “predictor size” we mean the size of a predictor for a single static branch.
9. Results not shown here indicate that DDT with 32 bits each of L&G history outperforms DDT with 16 bits of each, showing that dynamic feature selection enables exploitation of *useful* history unavailable to traditional techniques.
10. Preliminary study indicated that 8 bit counters perform reasonably for the DDT predictor on these benchmarks.

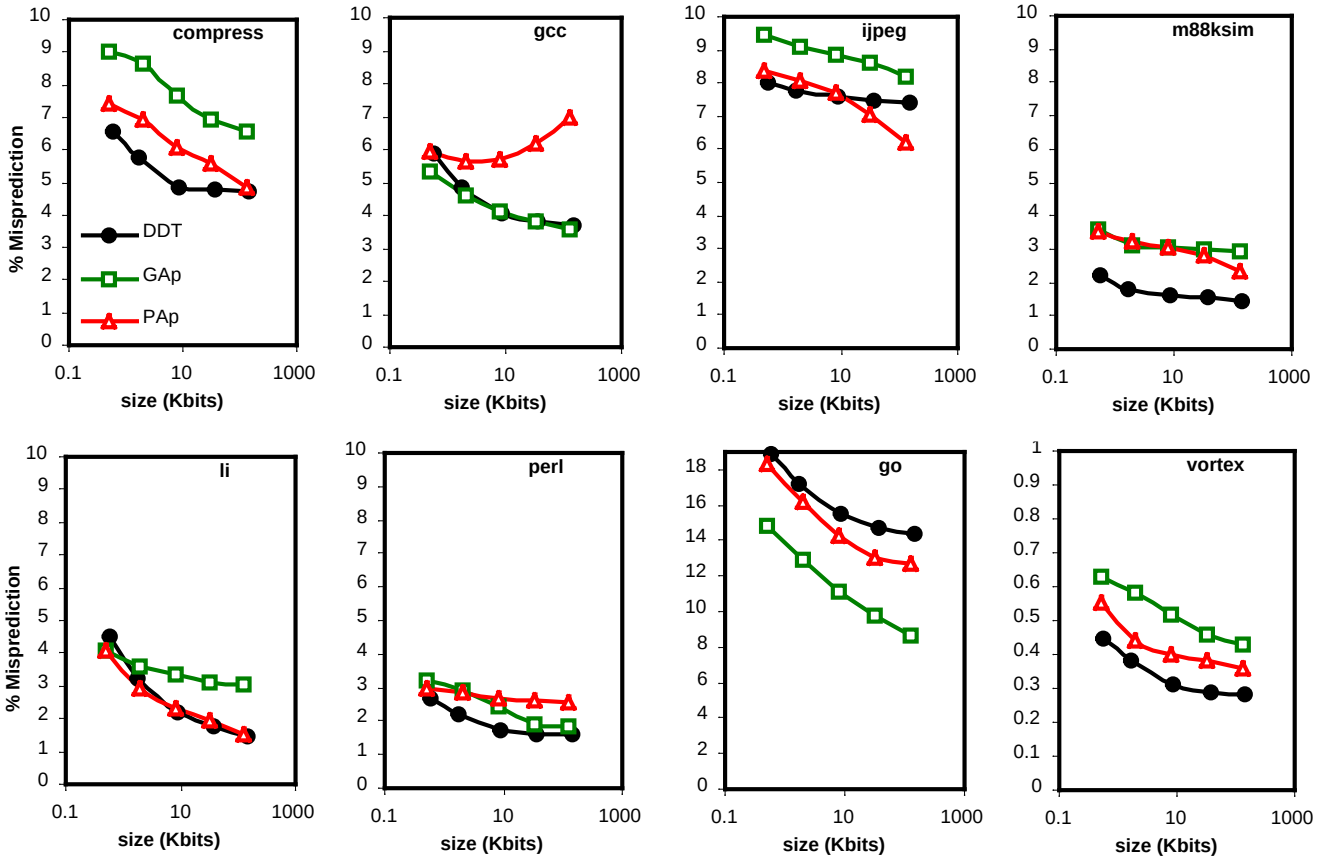


Figure 8: The above graphs show the percent misprediction versus size curves for each benchmark using the dynamic decision tree, GAp and PAp two-level predictors. Note for all graphs the percent misprediction scales are the same, except for *go* and *vortex*. The data points on the GAp and PAp curves from left to right correspond to history lengths of 8, 10, 12, 14, and 16. The data points on the dynamic decision tree curve correspond to depths of 0, 1, 3, 5, and 7. All nodes of the dynamic decision trees used the LG feature set (64 features) and 8-bit counters to store correlation values.

12, 14, and 16 bits and for the dynamic decision tree predictor using the LG feature set for depths of 0, 1, 3, 5, and 7. We selected these numbers to yield predictors of comparable sizes. The *misprediction rate* for a predictor on a benchmark is defined to be the number of mispredictions made by the predictor on the benchmark divided by the number of dynamic branches in the benchmark. We give misprediction rate versus predictor size curves for the PAp, GAp, and DDT predictors for each benchmark in Figure 8.

As expected and as previously known, when GAp and PAp are compared to each other neither predictor is superior across all benchmarks, and in fact each is significantly better than the other for some benchmark. The differences in their relative performance across benchmarks can be attributed to the different information they utilize. The graphs also show that for all benchmarks except *go* and *ijpeg* the dynamic decision tree curve is as good or better than the best of the GAp and PAp curves. This indicates

that DDT is able to robustly select important features on a per-branch basis, and to form prediction rules using these features. The feature selection ability of DDT coupled with its larger feature set (including both local and global history) make it a more *robust* predictor across benchmarks than GAp or PAp.

It is interesting to observe the performance of the depth-zero decision trees (the left-most point on each decision-tree curve) compared to the performance of GAp and PAp across the benchmarks. Note that a depth-zero tree is equivalent to a single correlation predictor so that predictions are made by simply selecting a *single feature* from the 64 LG features and using its value as the predicted outcome. Surprisingly this predictor is able to achieve nearly the same or better performance (losing by at most 0.03 percent) than the *worse* of the 16-history-bit versions of GAp or PAp for the benchmarks *compress*, *m88ksim*, *ijpeg*, and *vortex*. In addition, the depth-zero tree beats *both* the 16-history-bit GAp and PAp predictors on the *m88ksim* benchmark. Despite the fact that the GAp and PAp predictors form rules involving 16 features (using 2^{16} counters) and the depth-zero tree forms rules involving only a single dynamically selected feature, the depth-zero trees achieve nearly the same or better performance on these benchmarks. This result demonstrates the utility of feature selection mechanisms in hardware predictors.

The curves also show that for the four benchmarks *compress*, *m88ksim*, *perl*, and *vortex* the DDT curve is better than both the GAp and PAp curves. There are several potential explanations for the superior performance of the dynamic decision tree on these benchmarks. First, the ability of DDT to select the “best” features on a per-branch basis may be responsible for its ability to beat both GAp and PAp on these four benchmarks. It is possible that some branches within a particular benchmark are predicted best with global history while others are predicted best with local history and still others may be predicted best with a combination of the two. Second, experiments not presented here have shown that DDT gains predictive value from exploiting history bits beyond 16 bits. The GAp and PAp predictors we implemented consider at most history bits 0 through 15 (considering more bits would have an unreasonable and exponential cost in predictor size) whereas the DDT considers global and local history bits 0 through 31. Thirdly, we expect DDT to have a less severe warm-up penalty for some static branches since DDT has the ability to ignore irrelevant features—adding an irrelevant feature to GAp, for instance, doubles the number of counters that must be warmed up for good performance; no such warm-up penalty applies to DDT. This phenomenon should be most significant for branches that can be predicted well by rules involving only a small number of features. We have not yet determined which of

the three factors just mentioned or possibly others are primarily responsible for the superior performance of DDT on these four benchmarks.

The graphs show that for *li* and *gcc* the dynamic decision tree curves are nearly the same as the GAp and PAp curves respectively. Note that DDT robustness is again apparent: for *li* it is GAp that performs better than PAp, and for *gcc* it is the reverse; in each case DDT is near the better of GAp and PAp. It is interesting to note that for these two benchmarks DDT and the better of GAp and PAp give nearly identical misprediction versus size curves in spite of the drastic differences in implementation technique.

Why is *go* hard? The performance of the dynamic decision tree on the *go* benchmark is significantly inferior to that of both the PAp and particularly the GAp predictor (losing by 5.78 percent). Although the reason for the inferior performance is not yet certain, we have formed one hypothesis that is supported by the work published in [10]. Recall from Figure 3, that decision trees are suited for domains where accurate predictions can be made by rules involving a small number of features selected from a larger set of features. [10] gives evidence for the hypothesis that *go* requires rules involving a large number of features. In that work experiments were conducted where the three most predictive features (based on global history) were selected off-line for each static branch and only those features were used to train a GAp-like predictor. On all of the benchmarks shown above except *go* the interference-free versions of this offline three-feature predictor achieved accuracies very close to those of the interference-free GAp predictor using 16 global history bits. This result suggests that the poor performance of DDT on *go* is directly attributable to the need for rules depending on many features to accurately predict the branches in *go*. Note that DDT for a depth d makes predictions based on rules using at most $d + 1$ features for any one prediction, whereas the corresponding sized GAp or PAp predictor may use considerably more features at once for a given prediction (GAp and PAp use all available features for *every* prediction). For example, for DDT using LG at depth-seven, the corresponding sized GAp predictor uses 16 history bits for every prediction where DDT is using at most eight.

We also note that whereas GAp significantly outperforms DDT on the *go* benchmark, PAp with history length 14 significantly outperforms the comparable depth DDT on the *ijpeg* benchmark (by 1.23 percent for a history of 16). The reason for this is also unclear at this point. We note one anomaly in the *ijpeg* results: all three predictors seem to benefit similarly from increasing size until 10 Kbits, where PAp derives much greater benefit from further size increase. The effect is striking, and is responsible for the advantage PAp has over DDT at large size, but we have not found an explanation at this point.

7 Related Work

Here we consider only the work most closely related to our proposed approach but not covered above.

The work in [11] introduced the use of context trees for branch prediction—however, this predictor is not tied to the branch prediction domain and could be applied to other domains with no major changes. The rules that can be expressed by a context tree are a strict subset of the rules that can be expressed by a DDT of the same depth. This can be seen by considering that a context tree is simply a decision tree with the following three restrictions: First, all nodes at the same depth of a context tree must use the same test. In contrast a DDT has the freedom to select tests on a per-node basis, making it a more flexible decision structure. A DDT of maximum depth d, can use as many as $2^{d+1} - 1$ different features[11] to form prediction rules, one at each node, whereas a context tree of the same depth can only use d features. Second, the tests that appear in a context tree must be in the same order (according to depth) as they appear in the feature vector. Thus, even if the first feature of the feature vector (history bit 0) is not important it is still used to divide the incoming data stream whereas the DDT would ignore this feature. Third, the depth of a context tree is the same as the number of features (all features must be used) causing its size to blow-up exponentially with the dimension. A decision tree on the other hand has the freedom to use a depth that is much smaller than the number of features, thus avoiding the blow-up in size. Context trees are similar to the *prediction by partial matching (PPM)* algorithm used for branch prediction in [8]. In that work it was pointed out that the GAp/PAp two-level predictors are approximations to the PPM algorithm and that PPM a has slight advantage resulting mainly from reduced warm-up error.

One method, used in branch prediction, for using more features is to combine predictors that use different sets of features. In [20] a method was described for dynamically selecting one of two predictors to be used for a branch based on an estimate of which predictor achieved a higher prediction accuracy for that branch. [9] extended this work and provided a mechanism to select between N predictors on a per-branch basis. Thus, it is possible for these methods to select a predictor for a branch that uses the most appropriate feature set. Note, however, that the potential feature sets that may be selected are fixed off-line and are not dynamically adjusted to meet the needs of a particular branch. In this respect the DDT is a more flexible prediction method than that of combining predictors. For example, suppose

11. This is the number of nodes in a depth d full binary decision tree.

a predictor is formed by combining a GAp and PAp predictor. It is only possible to make predictions based either on all the local history bits or all the global history bits. Predictions can not be made based on a combination of local and global history whereas the DDT has that flexibility (while this may or may not be critical in branch prediction, in wider hardware prediction this flexibility can be expected to be useful). Furthermore, the individual predictors that are currently combined still suffer from exponential blow-up. We also note that DDTs can also be hybridized. For example, we could form a hybrid predictor using two DDTs—one with a large amount of local history information and one with a large amount of global history information. We have shown elsewhere that hybrids of multiple DDT predictors can often make better predictions than a single large DDT (with equivalent space usage) using a machine-learning technique called “boosting” [12].

Work in [14] provides a limited form of feature selection in the branch prediction domain, providing a method for dynamically searching for the ‘best’ history length to use during a prediction. Using this method a predictor can ignore higher order history bits dynamically. The sizes of the predictors presented, however, are still exponential in the maximum history length. Feature sets under this method are much less general than those under DDT. For example, suppose that only history bit 15 is needed for prediction. The DDT is likely to select just this feature, however, dynamic history length must use all of bits 0 through 15. It is also not clear how to generalize this method to other prediction domains where the features may have no natural ordering from more useful to less useful (like recentness in branch prediction).

There has been some recent work that adapts machine learning techniques to dynamic hardware prediction. To the best of our knowledge, prior to our original DDT report [33] there was only one such proposal [35] (also see [32] for a more recent presentation), which explored the use of multi-layer neural networks for dynamic branch prediction. Later work explored the use of single-layer neural networks, known as perceptrons, again for branch prediction. We are not aware of any other work that considers the use of decision-tree learning for dynamic hardware prediction.

Neural-network predictors are qualitatively much different than decision trees—however, both types of predictors grow only linearly in the number of features, allowing the consideration of much more predictive information than possible with table-based methods. It is well known in the field of machine learning that there is no single prediction architecture that dominates all others across all domains. Thus, it is desirable to develop a library of hardware-prediction architectures and to under-

stand their relative strengths and weaknesses in order to make the best design choices. Below, we briefly compare the primary alternatives to table-based architectures in the existing “hardware-prediction library”.

The multi-layer neural networks in [32] were shown to have promising accuracy in the branch prediction domain. However, there remain serious, unresolved hardware implementation issues. That work used the training method of error backpropagation, which is somewhat complicated (e.g. involving floating point numbers and multiplication) compared to decision-tree and perceptron training. More work needs to be done to evaluate the feasibility of multi-layer neural networks as a hardware prediction architecture. However, both the perceptron predictor from [31] and the DDT appear to have reasonably efficient hardware implementations. Below, we compare the prediction times, training times, and storage requirements of these architectures.

The perceptron makes predictions by taking a weighted sum of the feature values (a weight is stored for each feature) and comparing the result to a threshold. This process can be carried out in time that is logarithmic in the number of features. The prediction time for the DDT depends only on tree depth and not on the number of features. Thus, in terms of prediction time the DDT has an advantage over perceptrons in domains with many features.

Regarding training, it is interesting that the perceptron training method presented in [31] is identical to training a depth-zero DDT (i.e. a single DDT leaf node).[12] That is, training a perceptron amounts to updating a single correlation feature selector (each counter corresponds to a perceptron weight). This suggests that the training time and storage requirements of the perceptron are similar to that of a depth-zero DDT. As we increase the depth of a DDT (to improve accuracy) the space increases—however, by using parallelism the training time can remain similar to a depth-zero tree (and thus perceptron).

More specifically, the perceptron is somewhat simpler to train than the DDT presented here, as the perceptron does not require the computation of the summary information that is used by the DDT prediction circuitry (see Section 5). Computing this information takes time logarithmic in the number of features. Effectively the DDT performs an extra logarithmic amount of work during training to avoid that work during prediction. This transfer of work is not required by the DDT, but is desirable since pre-

12. For those familiar with perceptron training, this equivalence is only true when the learning rate parameter is equal to one. This was the choice used in [31], which has the advantage of avoiding multiplication and floating point numbers.

diction time is often a primary bottlekneck. For the perceptron it does not seem possible to avoid the logarithmic work during prediction, i.e. the perceptron must compute a sum over all features to make a prediction. Note that it is the DDT's explicit use of feature selection that allows it to transfer the logarithmic work from prediction time to training time.

Finally we want to compare our DDT with related work outside of hardware prediction. Machine learning research offers several methods of dynamically training decision trees, such as [26, 15, 27]. These methods, however, all require that the observed feature vector/outcome pairs be stored in memory and therefore are not practical for most hardware-prediction domains, which will produce huge data sets. The dynamic training method used by the ID4 decision tree system [24] does not require the storage of past feature vector/outcome pairs and to the best of our knowledge is the closest previous work to our DDT. There are several differences between our DDT and ID4. First, ID4 uses an information rather than accuracy based test selection heuristic. Building on-line estimators of accuracy is relatively simple compared to estimating information. Second, if ID4 determines that the test at an internal node should change then the child subtrees of the node are discarded and must be retrained from scratch. Our DDT does not discard subtrees after a test has been changed, but instead uses the trees if they are found to be beneficial and otherwise allows them to adapt to the characteristics of the new data streams. Third, ID4 does not consider the performance of the subtrees when it decides to change the split feature at a node. Our DDT changes the split feature only if another feature is judged to be more predictive than all other features (including the split feature) and the subtrees. Our empirical investigations suggest that at least for the branch prediction domain it is beneficial to take the subtree performance into account.

8 Conclusion

In this paper we presented a novel framework, dynamic feature selection, for hardware prediction. This framework enables a predictor to dynamically select the most useful features from a large set of candidate features—using storage which grows only linearly in the number of candidate features considered. In comparison to table-based hardware prediction schemes, where the storage grows exponentially with the number of features, using dynamic feature selection makes it possible for a predictor to consider much larger amounts of information in making predictions.

This paper also presents and evaluates a hardware prediction scheme implementing dynamic feature selection. Our Dynamic Decision Tree (DDT) predictor is derived from an on-line adaptation of "deci-

sion trees", widely-used tools in machine-learning research. We presented simulation results that indicate that this general-purpose predictor on average performs comparably to conventional table-based interference-free predictors designed specifically for branch prediction. In addition, the DDT predictor is generally more robust than familiar branch predictors, performing close to the best of them across applications that favor various familiar predictors — this robustness is similar to that achievable by hybridizing familiar predictors, but again it is achieved here without such special-purpose branch-related technique (our predictor's robustness derives from general-purpose feature selection and can be expected to generalize across prediction domains). We showed that dynamic feature selection can approximate conventional table-based predictors' performance by selecting useful features from a much larger set of features (without any specialized knowledge about which of the features are useful). Our results indicate that the DDT successfully implements dynamic feature selection, adding to design options for future hardware prediction schemes. We expect the DDT to be the beginning of a line of innovations in hardware prediction problems that benefit from the use of dynamic feature selection.

9 Acknowledgements

This material is based upon work supported under a National Science Foundation Graduate Fellowship and Award No. 9977981-IIS.

10 References

[1] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser. Correlated load-address predictors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 54–63, May 1999.

[2] L. Breiman, J. H. Friedman, R. A. Olsen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[3] Leo Breiman. Some properties of splitting criteria. *Machine Learning*, 24(1):41–47, July 1996.

[4] Carla Brodley. Automatic selection of split criterion during tree growing based on node location. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 73–80, 1995.

[5] Wray Buntine and Tim Niblett. A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 8:75–85, 1992.

[6] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[7] Brad Calder, Dirk Grunwald, and Joel Emer. Predictive sequential associative cache. In *HPCA96*, February 1996.

[8] I-Cheng K. Chen, J. Coffey, and Trevor Mudge. Compression and branch prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 128–137, October 1996.

[9] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[10] Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt. An analysis of correlation and predictability: What

makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, June 1998.

[11] Etian Federovsky, Meir Feder, and Shlomo Weiss. Branch prediction based on universal data compression algorithms. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 62–72, June 1998.

[12] Alan Fern and Robert Givan. Online ensemble learning: An empirical study. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[13] Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[14] Toni Juan, Sanji Sanjieevan, and Juan J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 155–166, June 1998.

[15] D. Kalles and T. Morris. Efficient incremental induction of decision trees. *Machine Learning*, 24, 1996.

[16] An-Chow Lai and Babak Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[17] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[18] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, pages 4–13, December 1997.

[19] Mikko H. Lipasti, Christopher B. Wilkerson, and John P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[20] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Laboratory, June 1993.

[21] Andreas Moshovos, Scott E. Breach, and T. N. Vijaykumar. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[22] Ravi Nair. Dynamic path-based branch prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, pages 142–152, December 1996.

[23] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[24] J. Schlimmer and D. Fisher. A case study of incremental concept induction. In *Proceedings of AAAI-86 Fifth National Conference on Artificial Intelligence*, pages 496–501, August 1986.

[25] Zak Smith, Timothy H. Heil and J. E. Smith. Improving branch predictors by correlating on data values. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, pages 28–37, December 1999.

[26] Paul E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4, 1998.

[27] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29, 1997.

[28] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 24)*, pages 51–61, December 1991.

[29] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[30] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 25)*, pages 129–139, November 1992.

[31] Daniel Jimenez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369-397, 2002.

[32] Colin Egan, Gordon Steven, Patrick Quick, Ruben Anguera, Fleur Steven, and Lucian Vintan. Two-level branch prediction using neural networks. *Journal of Systems Architecture*, to appear.

[33] Alan Fern, Robert Givan, Babak Falsafi, and T. N. Vijaykumar. Dynamic feature selection for hardware prediction. Technical Report (TR-00-12), School of Electrical and Computer Engineering, Purdue University, 2000.

[34] Tom Mitchell. *Machine Learning*. MIT Press and McGraw-Hill, 1997.

[35] Lucian Vintan and M. Iridon. Toward a high performance neural branch predictor. *International Joint Conference on Neural Networks*, 1999.

Appendix

In this Appendix, we provide for reference formal definitions and descriptions for the correlation feature selector and the dynamic decision tree.

Correlation Feature Selector. We define $\mathrm{CS}(F)$ for any set of features F to be a bank of $|F|$ signed counters, one for each feature in F. We write $\mathrm{CS}_f(F)$ for the value of the counter associated with feature f, and when the feature set F is implicitly clear, we write CS_f as an abbreviation. For a given bank of counters $\mathrm{CS}(F)$, we also define the *utility* of each feature f, written $U_f(\mathrm{CS}(F))$, to be the absolute value of the counter value associated with feature f in $\mathrm{CS}(F)$, *i.e.*, $U_f(\mathrm{CS}(F)) = \left|\mathrm{CS}_f(F)\right|$. Again, when the bank of counters is implicitly clear, we will abbreviate the utility as U_f.

We may also wish to know whether the feature selected is positively or negatively correlated with the outcome. For this purpose, we define a function $\mathrm{Sign}(\mathrm{CS}(F), f)$ that returns 1 if the signed counter $\mathrm{CS}_f(F)$ is positive, and 0 otherwise. Again, where $\mathrm{CS}(F)$ is implicitly clear, we will write $\mathrm{Sign}(f)$.

Updating a Correlation Predictor. We define $\mathrm{Update}(\mathrm{CS}(F), V, T)$, where V is a mapping from F to $\{1, 0\}$ representing the observed feature vector (so that $V(f)$ is the observed value of feature f) and T is either 1 or 0 representing the corresponding target outcome. $\mathrm{Update}(\mathrm{CS}(F), V, T)$ changes the counter values in $\mathrm{CS}(F)$ as follows: for each feature f we execute the "if" statement **if** $(V(f) = T)$ **then** $\mathrm{CS}_f{}^{++}$ **else** $\mathrm{CS}_f{}^{--}$**.** Figure 2 shows the functional diagram of this computation.

Selecting Features with a Correlation Predictor. Given a sequence of feature vector/target outcome pairs represented by $V_1, \ldots, V_m$ and $T_1, \ldots, T_m$, we can show that the result of sequentially updating an initially zero bank of counters $\mathrm{CS}(F)$ by executing the updates $\mathrm{Update}(\mathrm{CS}(F), V_1, T_1)$, …, $\mathrm{Update}(\mathrm{CS}(F), V_m, T_m)$ in order will result in counter values such that for each feature f, U_f is proportional to the number of correct predictions that would have been made by using the better of feature f or its complement to directly predict the outcome. Figure 3 shows the functional diagram of this computation.

Nodes of the Dynamic Decision Tree. Each tree node takes as input the values of the branch features, $f_1, \ldots, f_n$, and in addition each internal node also has two child predictors. Each node has a correlation feature selector: leaf nodes' selectors are $\mathrm{CS}(f_c, f_1, \ldots, f_n)$, whereas internal nodes use $\mathrm{CS}(f_c, f_1, \ldots, f_n, f_{\mathrm{sub}})$. Note that the order of the features is important in that we prefer to break ties by using earlier features, and thus prefer to make predictions using f_c and disprefer to make predictions using the child predictors via f_{sub}, given ties in the correlation values. Figure 4 shows the functional diagram of this computation.

In addition, each node contains summary information based on the correlation selector values, as follows: f_{split} caches the value of $\mathrm{Best}(f_1, \ldots, f_n)$, $x_{\text{use-fc?}}$ caches the value of $f_c = \mathrm{Best}(f_c, f_1, \ldots, f_n, f_{\mathrm{sub}})$, and at internal nodes only $x_{\text{use-sub?}}$ caches the value of $f_{\mathrm{sub}} = \mathrm{Best}(f_c, f_1, \ldots, f_n, f_{\mathrm{sub}})$. The equality comparison returns 1 if it is true and 0 otherwise. This summary information is used to speed up the critical prediction path. It would also be reasonable to cache $\mathrm{Sign}(f_c)$ and $\mathrm{Sign}(f_{\mathrm{split}})$, as they are also used in the prediction phase. Figure 5 shows the functional diagram of this computation.

We now describe the computation performed by each node during the prediction phase. The input to the prediction phase is the feature vector V mapping the features $f_1, \ldots, f_n$ to $\{0, 1\}$, describing the branch. Our prediction is calculated as follows. If $x_{\text{use-fc?}}$ is true we return the value of $\mathrm{Sign}(f_c)$ as our prediction. Otherwise, if $x_{\text{use-sub?}}$ is false we return $V(f_{\mathrm{split}}) \oplus \mathrm{Sign}(f_{\mathrm{split}})$ as our prediction. Otherwise, we use the child prediction indicated by the split feature: we use the prediction returned by the left child if $V(f_{\mathrm{split}}) \oplus \mathrm{Sign}(f_{\mathrm{split}})$ is false, and the prediction returned by the right child otherwise. In each case, we are comparing (with xor) the feature value to the sign of the correlation to handle both positive and negative correlation appropriately. So, for example, if f_1 is highly negatively correlated with the outcome, we want to use $\overline{f_1}$ as our prediction. Figure 6 shows the functional diagram of the prediction phase of an internal node.

Finally, we describe the update phase, where the input is a feature vector/target outcome pair (V, T). First, the value of f_{sub} is updated by selecting a child predictor based on the value of $V(f_{\mathrm{split}}) \oplus \mathrm{Sign}(f_{\mathrm{split}})$ as was done above and setting $V(f_{\mathrm{sub}})$ equal to the prediction returned by the selected child. Note that the child selection and resulting prediction may have already been done during the prediction phase (and can then be reused/shared here), but if it was not done in prediction it must

still be done for update. Second, the correlation selection counters CS(F) at the node (described above) are updated by performing the function Update(CS(F), V, T) where we assume V maps f_c to 1. Finally, the summary information is updated as described above based on the new counter values.

One final practical point: we believe it is important to minimize the frequency with which the value of the split feature f_{split} changes, because such changes can dramatically alter the streams of data seen by the child predictors. For this reason, when updating f_{split} we implement a preference to keeping f_{split} unchanged in the case of tied correlation values, even if an earlier feature in the sequence is involved in the tie. In the long run, we believe it will be important to implement an even stronger preference to sticking with the current split feature to avoid oscillation/vacillation between features of similar predictive utility. We have not addressed this issue further in this work, however.