Online Ensemble Learning: An Empirical Study

Alan Fern AFERN@ECN.PURDUE.EDU
Robert Givan GIVAN@ECN.PURDUE.EDU
Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA

Abstract

We study resource-limited online learning, motivated by the problem of conditional-branch outcome prediction in computer architecture. In particular, we consider (parallel) time and space-efficient ensemble learners for online settings, empirically demonstrating benefits similar to those shown previously for offline ensembles. Our learning algorithms are inspired by the previously published "boosting by filtering" framework as well as the offline Arc-x4 boosting-style algorithm. We train ensembles of online decision trees using a novel variant of the ID4 online decision-tree algorithm as the base learner, and show empirical results for both boosting and bagging-style online ensemble methods. Our results evaluate these methods on both our branch prediction domain and online variants of three familiar machine-learning benchmarks. Our data justifies three key claims. First, we show empirically that our extensions to ID4 significantly improve performance for single trees and additionally are critical to achieving performance gains in tree ensembles. Second, our results indicate significant improvements in predictive accuracy with ensemble size for the boosting-style algorithm. The bagging algorithms we tried showed poor performance relative to the boosting-style algorithm (but still improve upon individual base learners). Third, we show that ensembles of small trees are often able to outperform large single trees with the same number of nodes (and similarly outperform smaller ensembles of larger trees that use the same total number of nodes). This makes online boosting particularly useful in domains such as branch prediction with tight space restrictions (i.e., the available real-estate on a microprocessor chip).

Keywords: online learning, ensemble learning, boosting, bagging, decision trees, branch prediction

1 Introduction

Ensemble methods such as boosting and bagging have been shown to provide significant advantages in offline learning settings—however, little work has been done exploring these methods in online settings. Here we consider an online setting, motivated by the problem of predicting conditional branch outcomes in microprocessors. Like many online learning problems, branch prediction places tight time and space constraints on a learning algorithm (i.e., the space is limited by the available microprocessor chip real-estate and the time is limited by the frequency the processor encounters conditional branches, typically every few nanoseconds). Thus, time and space efficiency are crucial factors in the design of our online ensemble methods. Our application does offer the benefit of cheap natural parallelism (at the silicon level) to assist in meeting the time constraint.

Ensemble learning algorithms provide methods for invoking a base learning algorithm multiple times and combining the results into an ensemble hypothesis. Many empirical investigations have shown that ensemble learning methods often lead to significant improvements across a wide range of learning problems (Breiman, 1996a; Freund & Schapire, 1996; Quinlan, 1996; Bauer & Kohavi, 1999; Dietterich, 2000). To our knowledge, however, all of these investigations have taken place in an offline learning setting. The main goal of this research is to demonstrate that similar performance gains can be obtained in online learning settings by using time and space efficient online ensemble algorithms. Secondary goals include designing and evaluating appropriate online base learners for use in ensembles, and measuring the cost/value of ensembles in reducing the space requirement needed to achieve a given classification accuracy.

We consider the simplified problem of online binary-concept learning with binary features—however, it is likely that the methods presented here can be extended to non-binary problems in familiar ways. For this work, we use decision-tree base learners, partly because our hardware-oriented application requires nanosecond prediction delays. Due to our resource constraints, we prefer an online decision-tree method that does not store a large number of training instances, and so as our base learner we use a novel variant of ID4 (Schlimmer & Fisher, 1986) (which is an online version of the ID3 (Quinlan, 1986) offline decision-tree algorithm). We present empirical evidence that our extensions to ID4 improve performance in single trees and are critical to good performance in tree ensembles—our results support the suggestion that the original ID4 warms up too erratically and slowly for use in online ensembles.

We note that our time and space constraints also rule out the direct application of offline ensemble algorithms by storing the training instances and invoking the offline algorithm when each new instance arrives. When a training instance arrives we update our ensemble immediately and the instance is then discarded.

Freund (1995) describes a version of the *boost-by-majority (BBM)* boosting algorithm for the "boosting by filtering" ensemble learning framework. In the "boosting by filtering" framework ensembles are generated online and without storing previous instances, as in our methods. The BBM algorithm implements a sequential ensemble gener-

ation approach where the ensemble members are generated one at a time. In practice, to use a sequential generation approach such as BBM for online problems we must address at least two challenging issues. First, we must select some method for the learner to determine when to stop generating one ensemble member and to begin generating the next. BBM provides a theoretical method in terms of parameters of the base learner that are generally not known in practice. Second, we must provide a means for the ensembles to adapt to drifting target concepts, since BBM itself does not update ensemble members once they are created. In light of these issues we consider a variation of the 'boosting by filtering' approach that generates ensemble members in parallel—that is when a training instance arrives, more than one (and potentially every) ensemble member may be updated rather than a single member as is done by sequential approaches such as BBM. Because these updates occur in parallel in our application, there is no additional time cost to our parallel approach. In addition to being simpler to specify (for the reasons above), we also expect parallel-generation approaches to yield learners that warm up more quickly (in parallel time) because each training instance is potentially used to update many ensemble members rather than just one (we discuss empirical results supporting this expectation in Section 7). Unlike BBM, however, the online boosting-style algorithm we present has not been proven to be a boosting algorithm in the theoretical sense (hence the term boosting-style rather than boosting)—the results we give are empirical in nature.

We describe two such "parallel-generation" online ensemble algorithms: one inspired by the offline ensemble method of bagging, and one inspired by the offline boosting-style algorithm Arc-x4. These methods have an efficient parallel hardware implementation where the time complexities of updating and making predictions with the ensemble grow only logarithmically with the number T of ensemble members. The space complexity of this implementation grows linearly with T and is typically dominated by the space occupied by the ensemble members. This efficient parallel implementation is extremely important in the branch prediction domain where the implementation platform (VLSI circuits) invites parallelism.

We note that ensemble learning methods generally lend themselves to parallel implementation, and thus are natural for use under tight time constraints. We also note that online learning domains are naturally likely to present such time constraints. These facts suggest that ensemble methods may be particularly useful in online settings. In addition, parallel-generation approaches incur no extra time cost in parallel implementations and so may also be particularly well-suited to online settings. However, our results also indicate that online ensembles improve classification error over single base learners in sequential implementations—in this case the ensemble will take much more time than the base learner, and other time-consuming approaches may be competitive.

Using our ID4-variant as the base learner, we empirically evaluate our online ensemble methods against instances of the branch prediction problem drawn from widely-used computer-architecture benchmarks, as well as against online variants of several familiar machine-learning benchmarks. Our results indicate that our boosting-style

algorithm online Arc-x4 consistently outperforms our online bagging methods. Online Arc-x4 is also shown to significantly improve the error rate compared to single base learners in most of our experiments. In addition, we show that ensembles of small trees often outperform large single trees that use the same total number of tree nodes—similarly, large ensembles of small trees often outperform smaller ensembles of larger trees that use the same number of nodes. These results are important to domains with tight space constraints such as branch prediction. Finally, we give results indicating that our base-learner extensions are critical to obtaining these effective ensembles.

The remainder of this paper is organized as follows. In Section 2 we briefly describe our motivating application of branch prediction. In Section 3 we briefly discuss the problem of online concept learning and then present our novel online boosting-style algorithm, online Arc-x4. Section 4 discusses the parallel time and space complexity of online Arc-x4. In Section 5 we give and discuss empirical results for online Arc-x4. In Sections 6 and 7, we describe our extensions to ID4 (used as the base learner) and give results evaluating their effect on single tree and ensemble performance. Finally, Appendix A describes our online ensemble algorithms based on bagging and gives empirical results showing poor performance for online bagging in our domains relative to online Arc-x4 (however, online bagging still improves upon individual base learners).

2 Branch Prediction

This research is motivated by the problem of dynamic conditional-branch outcome prediction in computer architecture. It is not our primary goal here to beat current state-of-the-art branch predictors but rather to open a promising new avenue of branch-predictor research, as well as to explore empirically an online setting for boosting (which is of interest independently of branch prediction)

Problem Description. Critical to the performance of nearly all modern out-of-order processors is their ability to predict the outcomes (taken or not-taken) of conditional branch instructions—this problem is known as branch prediction. During out-of-order execution if a branch instruction is encountered whose condition is unresolved (i.e., the condition depends on instructions that have not yet finished execution) the prediction of its outcome guides the processor in speculatively executing additional instructions (down the path the branch is predicted to take). Finding accurate branch prediction techniques is a central research goal in modern microprocessor architecture.

Typical programs contain conditional branches about every third instruction, and individual branches are encountered hundreds of thousands of times. For each encounter, the branch predictor predicts the outcome (i.e., taken or not-taken) using a feature vector composed of a subset of the processor state during prefetch. After the true branch outcome is known the feature vector and outcome are used by the branch predictor as a training example leading to an updated predictive model. Branch prediction is thus a two-class concept-learning problem with a binary feature space

in an online setting.

Machine learning ideas have previously been applied to the different but related problem of *static* branch prediction (Calder et al., 1997). Static branch prediction involves predicting the most likely outcomes of branches before program execution (i.e., at compile time) rather than predicting the outcome of actual branch instances as they are encountered during program execution which is the goal of *dynamic* branch prediction. To the best of our knowledge there has been no other work in the machine learning community focused on dynamic branch prediction.

Qualitative Domain Characteristics. Branch prediction is a bounded time/space problem—predictions must be made quickly, typically in a few nanoseconds. Additionally, a hardware implementation is required, so the resource constraints are much tighter and qualitatively different than those usually encountered in software machine-learning applications. Generally, giving a well-designed predictor more time/space results in a corresponding increase in prediction accuracy (e.g., allowing deeper trees). Using a larger predictor, however, implies less chip space for other beneficial microprocessor machinery. Thus when applying machine learning ideas to this problem it is important to carefully consider the time/space complexity of the approach—exploiting the VLSI parallel implementation platform to meet these time and space constraints.

Additional domain characteristics of interest from a machine learning perspective include: branch prediction requires an online rather than offline learning setting—conditional branches must be predicted as they are encountered; the number of encountered instances of a given branch is unknown ahead of time; context switching creates concept drift; branch prediction provides a fertile source for large automatically-labelled machine-learning problems; and finally, branch prediction is a domain where significant progress could have a large impact (reducing branch predictor error rates by even a few percent is thought to result in a significant processor speedup (Chang et al., 1995)).

Contribution to branch prediction. An electronic appendix (Fern & Givan, 2001) contains an overview of past and present branch prediction research. Virtually all proposed branch predictors are table-based (i.e., they maintaining predictive information for each possible combination of feature values) causing their sizes to grow exponentially with the number of features considered. Thus, state-of-the-art predictors can only use a small subset of the available processor state as features for prediction.[1] The methods we describe avoid exponential growth—our predictors (ensembles of depth-bounded decision trees) grow linearly with the number of features considered. This approach is able to flexibly incorporate large amounts of processor state into the feature space while remaining within architecturally-re-

1. One approach to easing the exponential growth problem is to use a fixed hash function that combines the feature vector bits into a smaller number of hashing bits used to access the prediction table (e.g., XOR'ing bit sets together). Such methods lose information but reduce the exponent of the space complexity. To avoid exponential dependence on the number of features the number of hashing bits must be logarithmic in the number of features—therefore exponentially many different branch instances are mapped to the same hash location. It seems unlikely that a single fixed hash function can be found giving logarithmic compression that avoids loss in accuracy for most branches (each branch represents a distinct prediction problem but encounters the same hash function) relative to the accuracy attained without compression. Current methods do not achieve logarithmic compression, rather they reduce the exponent by a linear factor and are still exponential in the number of features.

alistic space constraints. The ability of our predictors to incorporate substantial additional processor-state information should lead to substantial improvements in the prediction accuracy available for a fixed space usage. The most common features used by current branch predictors are global and local history bits. *Global history* bits store the outcomes of the most recently resolved branches. In contrast, *local history* bits store the most recent previous outcomes of the branch whose outcome is being predicted. Examples of additional processor state (beyond local and global history) that are known to contain predictive information include register bits and branch target address bits; however, current methods for utilizing this information are table-based, e.g., (Heil et al., 1999; Nair, 1995). Our intended contribution to branch prediction (and to the design of other architecture-enhancing predictors) is to open up the possibility of using much larger feature spaces in prediction.

3 Online Ensembles

This research addresses the problem of online concept learning using ensembles. For our purposes, a *concept* is a mapping from some domain to either zero or one. In concept learning problems we are provided with *training instances*: tuples comprised of a domain element and the class (zero or one) assigned to that element by the *target concept*. Based on the training instances we are asked to find a *hypothesis concept* that accurately models the target concept. *Offlin*e learning algorithms take as input a set of training instances and output a hypothesis. In contrast, *onlin*e learning algorithms take as input a single labelled training instance as well as a hypothesis and output an updated hypothesis. Thus, given a sequence of training instances an online algorithm will produce a sequence of hypotheses. Online learning algorithms are designed to reuse the previous hypothesis in various ways, allowing them to reduce update times to meet the constraints of online learning problems—these constraints are typically much tighter than for offline problems. The advantages of this hypothesis reuse are even more significant in an ensemble learning algorithm, since offline ensemble construction can be very expensive.

In recent years ensemble learning algorithms have been the topic of much theoretical and experimental research. These algorithms provide methods for invoking a learning algorithm (the base learning algorithm) multiple times and for combining the resulting hypotheses into an ensemble hypothesis (e.g., via a majority vote). The goal in using an ensemble of hypotheses is to be superior in some sense to the individual hypothesis generated by the base algorithm on the training instances. In this work we consider two popular ensemble methods, boosting and bagging.

To our knowledge, all previous empirical evaluations of ensemble methods have taken place in offline learning settings. In this research we investigate online variants of ensemble learning algorithms and demonstrate online performance gains similar to those seen in the previous offline evaluations. We also ensure that our online variants have efficient implementations that might be applied to online learning problems with significant resource constraints—without this restriction an offline algorithm can be used directly in the online setting at substantial resource cost[2].

In the remainder of this section we will first briefly describe (for completeness) the popular offline ensemble method, boosting, that inspired our most successful online algorithm. Next, we distinguish between sequential-generation and parallel-generation ensemble approaches, and give reasons to focus on parallel generation in this research. We then describe a generic online ensemble algorithm that allows for parallel generation. We show a boosting-style instantiation of this algorithm that we have implemented called online Arc-x4. Our results for an online bagging instantiation of the generic algorithm were not favorable when compared to online Arc-x4 (but still improve on individual base learners). Hence, we postpone our discussion of online bagging until Appendix A—noting that in domains where online bagging is competitive with online Arc-x4, bagging is preferable with respect to complexity.

3.1 Offline Ensemble Generation via Boosting

Boosting is an ensemble method that has received much attention and has been shown in several studies to outperform another popular ensemble method, bagging, in a number of offline domains[3] (Freund & Schapire, 1996; Quinlan, 1996; Bauer & Kohavi, 1999; Dietterich, 2000). We assume here that the base learning algorithms take into account a weight associated with each training instance, and attempts to return a learned hypothesis that minimizes the weighted classification error. Some of the most commonly used boosting algorithms for offline problems generate hypotheses sequentially as follows. The first hypothesis is the result of presenting the set of training instances, all with weights of one, to the base learning algorithm. Now assume the algorithm has already generated t–1 hypotheses. Weights are then assigned to the training instances such that larger weights are associated with instances that the previous hypotheses performed poorly on (the "hard" instances). These weighted instances are then given to the base learning algorithm which outputs the t'th hypothesis. Boosting algorithms differ mainly in the ways weights are assigned to instances and the ways hypotheses are combined. The AdaBoost algorithm (Freund & Schapire, 1997) and the boost by majority algorithm (Freund, 1995) have been proven to be boosting algorithm in the theoretical sense[4]. Arc-x4 (Breiman, 1996b) is another ensemble method inspired by boosting and is the basis for our online boosting-style method. AdaBoost and Arc-x4 have been empirically compared and exhibit similar performance (Bauer & Kohavi, 1999; Breiman, 1996b).

3.2 Online Approaches

There are several avenues that could be explored when designing an online ensemble algorithm. A naive approach is

2. An offline algorithm can be used in an online setting by simply storing the training examples as they arrive and invoking the offline algorithm on the stored example set whenever a new example arrives. This naive method can have a substantial cost in terms of space and update time.
3. The advantages of boosting have been shown to degrade as noise levels increase and boosting may actually hurt performance in some of these domains (Dietterich, 2000).
4. Technically a boosting algorithm is one that 'transforms' a weak learning algorithm into a strong learning algorithm (Schapire, 1990).

to maintain a dataset of all observed instances and to invoke an offline algorithm to produce an ensemble from scratch when a new instance arrives. This approach is often impractical both in terms of space and update time for online settings with resource constraints. To help alleviate the space problem we could limit the size of the dataset by only storing and utilizing the most recent or most important instances. However, the resulting update time is still often impractical, particularly for boosting methods—when the training set used by a boosting algorithm is altered we potentially need to recalculate weights and invoke the base learning algorithm for each of the T hypotheses from scratch. It is unclear whether there is a time-efficient online boosting variant that stores the set of previous instances in order to duplicate the offline algorithm performance. In part because of the tight resource constraints in our application domain of branch outcome prediction, for this research we chose to consider only methods that do not store previous instances—other approaches may also be feasible. Below we discuss two possible online ensemble approaches that do not require previous instances to be stored; we call these the sequential-generation and parallel-generation approaches—we then argue for and focus on the parallel-generation approach.

We say that an online ensemble algorithm takes a *sequential-generation* approach if it generates the ensemble members one at a time, ceasing to update each member once the next one is started (otherwise, we say the approach is *parallel-generation*). We say the algorithm takes a *single-update* approach if it updates only one ensemble member for each training instance encountered (otherwise, we say *multiple-update*). We note that any algorithm taking a sequential-generation approach will generally also take a single-update approach.[5] The boosting by filtering framework described by Freund (1995) can be viewed as taking the sequential-generation single-update approach—two algorithms for the filtering framework are the *boost-by-majority (BBM)* algorithm (Freund, 1995) and MadaBoost (Domingo & Watanabe, 2000). Note that these algorithms are boosting algorithms in the theoretical sense[4] while the parallel-generation boosting-style algorithm we investigate here has not been shown to possess this property.

We wish to avoid the single-update approach (and thus the sequential approach). One reason for this is that the offline methods of boosting and bagging both have the property that a single training instance can contribute to the training of many ensemble members—we believe that achieving this property in the online setting is essential to obtaining rapid convergence to the desired target concept; this is particularly important in the presence of concept drift. Our empirical results described on page 35 provide evidence that our parallel-generation multiple-update ensembles converge more quickly than a sequential approach would. Sequential-generation algorithms also suffer additionally in the presence of concept drift because at any time most ensemble members are never going to be updated again—this patently requires adapting such algorithms with some kind of restart mechanism. Sequential methods also require a

5. The reader is advised that we use the terms “parallel generation” and “parallel implementation” for very different meanings herein (likewise for “sequential...”). “Parallel generation” refers to a method for training ensembles which can be implemented either serially or in parallel. “Parallel implementation” refers to an implementation technique in which more than one computation is carried out simultaneously.

difficult-to-design method for determining when to stop updating an ensemble member in favor of starting on another member.

To address these problems, we considered in this work only algorithms taking the parallel-generation multiple-update approach. We note that this approach interacts well with our motivating application in that multiple updates can easily be carried out simultaneously on a highly parallel implementation platform such as VLSI.

Generic multiple-update algorithm. Here we formally present a generic online ensemble algorithm that allows for multiple updates. Two instances of this algorithm are described (one here and one in Appendix A) and will be used in our experiments. An ensemble is a 2-tuple consisting of a sequence of T hypotheses $h_1,...,h_T$ and a corresponding set of T scalar voting weights $v_1,...,v_T$. A hypothesis h_i is a mapping from the target concept domain to zero or one (i.e., $h_i(x) \in \{0,1\}$ for each domain element x). Given a domain element x the prediction returned by an ensemble $H = \langle (h_1,...,h_T), (v_1,...,v_T) \rangle$ is simply a weighted vote of the hypotheses, i.e., one if $\left(v_1[2h_1(x)-1] + \ldots + v_T[2h_T(x)-1]\right) > 0$ and zero otherwise. A training instance is a tuple $\langle x,c \rangle$ where x is a domain element and c is the classification in $\{0,1\}$ assigned to x by the target concept (assuming no noise). We assume Learn is our base learning algorithm: an online learning algorithm that takes as input a hypothesis, a training instance, and a weight; the output of Learn is an updated hypothesis.

Figure 1 shows the generic multiple-update algorithm we will use. The algorithm outputs an updated ensemble, taking as input an ensemble, a training instance, an online learning algorithm, and two functions Update-Vote() and Weight(). The function Update-Vote() is used to update the $(v_1,...,v_T)$ vector of ensemble member voting weights (typically based on how each member performs on the new instance). The function Weight() is used for each ensemble member h_t to assign a weight w_t to the new instance for use in updating h_t.

For each hypothesis h_t the algorithm performs the following steps. First, in line 2 a new scalar voting weight v_t is computed by the function Update-Vote(). For example, if Update-Vote() always returns the number one, the ensemble prediction will simply be the majority vote. Next, in line 3 a scalar instance weight w_t is computed by Weight(). For example, in boosting Weight() would typically be a function of the number of mistakes made by previous hypotheses on the current instance, whereas in bagging Weight() would not depend on the ensemble members. Finally, in line 4, h_t is updated by Learn() using the training instance with the computed weight w_t. After each hypothesis and voting weight in the ensemble is updated in this manner (possibly in parallel), the resulting ensemble is returned.

The immediate research goal is to find (parallel) time and space efficient functions Update-Vote() and Weight() that produce ensembles that outperform single hypotheses in classification accuracy. In this paper we consider two very simple memoryless instances of this algorithm that are inspired by bagging and Arc-x4.

Input:
ensemble $H = \langle (h_{1,\ldots}, h_T), (v_1, \ldots, v_T) \rangle$
new training instance $I = \langle x, c \rangle$
base online learning algorithm Learn (*instance, weight, hypothesis*)
voting weight update function Update-Vote (*ensemble, instance, hypothesis-number*)
instance weight function Weight (*ensemble, instance, hypothesis-number*)

1. for each $t \in \{1, 2, \ldots, T\}$, ;; *possibly executed in parallel*
2. do $\hat{v}_t$ = Update-Vote (H, I, t) ;; *the new voting weight of hypothesis t*
3. w_t = Weight (H, I, t) ;; *the weight of this instance for updating* h_t
4. $\hat{h}_t$ = Learn (I, w_t, h_t)

Output: new ensemble $\hat{H} = \langle (\hat{h}_{1\ldots}, \hat{h}_T), (\hat{v}_1, \ldots, \hat{v}_T) \rangle$

Figure 1: Generic multiple-update online ensemble learning algorithm.

3.3 Online Arc-x4

Online Arc-x4 uses the same instance weight function Weight() as that used by the offline ensemble algorithm Arc-x4 (Breiman, 1996b). The instance weight function is computed in two steps:

$$\text{Weight}\,(H, I, t) = 1 + m_t^4, \qquad m_t = \sum_{i=1}^{t-1} \left| h_i(x) - c \right| \tag{1}$$

The weight for the t'th hypothesis w_t is calculated by first counting the number m_t of previous hypotheses that incorrectly classify the new instance. The weight used is then one more than m_t to the fourth power, resulting in a boosting-style weighting that emphasize instances that many previous hypotheses get wrong. This function was arrived at (partly) empirically in the design of offline Arc-x4. Nevertheless it has performed well in practice and its simplicity (compared to AdaBoost for example where we would need to consider ways to avoid the floating-point computations) made it an attractive choice for this application. For online Arc-x4 the function Update-Vote() computes the new hypothesis voting-weight simply by counting the number of correct predictions made by the hypothesis on the training instances seen so far,

$$\text{Update-Vote}\,(H, I, t) = v_t + 1 - \left| h_t(x) - c \right|, \tag{2}$$

where v_t is the previous voting weight (in this case the previous count of correct predictions) as shown in Figure 1. Thus, hypotheses that are more accurate will tend to have larger voting weights. Note that the offline version of Arc-x4 uses a majority rather than a weighted vote. We found, however, an empirical advantage to using weighted voting for small ensemble sizes. For large ensemble sizes the two methods gave nearly identical results.

We now briefly compare our parallel-generation approach to sequential-generation methods in cases where the two variations encounter the same stationary target concept and stationary distribution of instances. We assume that the online base learner is convergent in the following sense; for sequences of training instances drawn from a given stationary distribution and target concept the learner produces hypothesis sequences that all converge to the same hypothesis regardless of the initial hypothesis in the sequence. We also assume that both methods use the same instance-weighting and vote-weighting functions. We consider only sequential-generation methods that allow each base learner to converge before moving on to the next.[6] In parallel-generation methods, this corresponds roughly to using a weighting scheme where the instance weights for each hypothesis depend only on the results from previous hypotheses in the ensemble — we call such weighting schemes "ordered", and note that Arc-x4 uses a ordered weighting scheme. Under these assumptions, parallel-generation methods based on the generic algorithm in Figure 1 using a ordered weighting scheme converge to the same learned hypothesis that sequential methods converge to. This can be seen by noting that in the parallel method, the ordered weighting scheme implies that the first ensemble member converges independently of what is happening with later members (thus exactly as quickly as the first member in a sequential method); once the first member converges, the second member then converges independently of what is happening to other members since its weighted distribution depends only on the (now converged) first member. Extending this reasoning by induction, each ensemble member will converge to the same hypothesis under either sequential or parallel ensemble generation.

4 Complexity and Implementation of Online Arc-x4

An efficient parallel implementation is particularly significant to our target domain of conditional-branch outcome prediction where the implementation platform of VLSI invites and requires parallelism (see Section 5.2 for details of our target domain). In this section we show that the time complexity of a parallel implementation of online Arc-x4 is better than $O(\log^2 T)$ plus the prediction time used by an individual base learner; and that the space complexity of the same implementation is $O(T \cdot \log T)$ plus the space used by the T individual base learners. The detailed bounds are slightly tighter, and are derived below. All the results in this section are estimates in that they ignore specialized VLSI issues such as the space cost of high fan-out.

We note that our focus here is on a direct, specialized VLSI implementation like that needed for branch prediction. However, these calculations also shed some light on the complexity of a shared-memory multi-processor ensemble implementation, such as might be used for other online learning applications. A detailed discussion concerning the VLSI issues involved in a hardware implementation of online Arc-x4 is beyond the intended scope of this paper.

6. Of course practical sequential methods need a way of moving on to the next hypothesis should the current hypothesis take a long time or even fail to converge. Similar practical adaptations can be added to a parallel-generation method, preserving our claim in practice.

Our estimates here, however, suggest that Arc-x4 has a space/time efficient hardware implementation provided that the base learners have a space/time efficient hardware implementation. Our work targeted to the computer architecture community (Fern et al., 2000) proposes a parallel hardware design of the decision tree base learner we use in this paper (the base learner is described in Section 6.1).

The space and time complexity results are based mainly on the fact that the sum of T n-bit integers can be calculated in $O(\log\ T \cdot \log(n + \log T))$ time and $O(T \cdot (n + \log T))$ space, using a tree of 2-addend additions of at most $n + \log T$ bits. First we show that the prediction and update time complexities in terms of ensemble size T are both $O(\log T \cdot \log\log T)$. Next, we show that the space complexity in terms of T for both the prediction and update mechanisms is $O(T \cdot \log T)$. Below t_p is the worst-case time complexity for making a prediction using any base-learner hypothesis generated by Learn(), t_u is the worst-case time complexity for updating a base-learner hypothesis, and S_h is the worst-case space complexity of a base-learner hypothesis. The voting weights v_1,..., v_T as seen in Figure 1 are taken to have a precision of n bits (this defines n, which we treat here as a constant[7]).

Prediction Time. A prediction can be made by having all of the individual hypotheses make predictions in parallel and then summing the T voting weights v_1 through v_T (where the prediction of hypothesis h_t determines the sign associated with v_t). The sign of this sum determines the ensemble prediction. Therefore, the worst case time complexity of returning an ensemble prediction is the time to get the ensemble member predictions in parallel plus the time to calculate the sum of the vote weights which is $O(t_p + \log T \cdot \log(n + \log T))$, which is $O(\log T \cdot \log(n + \log T))$ if we take t_p to be constant.

Update Time. To update the predictor we first obtain the predictions of all hypotheses in parallel and update the voting weights. Next, for each hypothesis the number of mistakes made by previous hypotheses is stored and the update weights are calculated in parallel (see Equation 1). Finally the hypotheses are updated in parallel by Learn(). The worst-case time complexity[8] of calculating w_t in Equation 1 given the number of previous mistakes m_t is $O(\log \log T)$.

We now consider the worst-case time complexity of calculating m_t. To calculate m_t we first calculate the sum s_t of all $h_j(x)$ for j ranging from 1 to t–1. Given this sum we know the value of m_t is one of two values depending on the class of x; if the class is one then m_t is $t - s_t$ otherwise m_t is just s_t. The worst case time complexity for calculating s_t occurs when $t = T$ and is the time to sum T–1 bits which has complexity $O(\log T \cdot \log \log T)$. Notice that we can calculate s_t and hence the two possible values for m_i without knowing the class of x (this is the reason we introduce s_t)—

7. We note that n must be bounded in a practical hardware implementation. A common way of dealing with counter saturation is to divide all the counters by 2 whenever one of the counters saturate—this operation maintains the ordering among the counters approximately.

8. The computation requires two multiplications. The time complexity of multiplying two n-bit numbers is $O(\log n)$ (Cormen et al., 1997). Since the maximum number of bits needed to represent m_t is $\log T$ the time complexity to perform the multiplications is $O(\log\log T)$.

this feature of Arc-x4's weight function is useful in domains (such as branch prediction) where a prediction is made for a domain element before its class is known. In such domains we can calculate all the s_t in parallel (typically during the prediction voting calculation) and use these values later when the class is known to generate the instance weights used in updating the hypotheses.

Adding the base-learner prediction and update times, the time to update the voting weights, the time to calculate s_t, and the time to calculate w_t together, the worst-case complexity for the update time of online Arc-x4 is given by $O(t_p + \log n + \log T \cdot \log \log T + t_u)$.

Space Complexity. Since making a prediction amounts to calculating the sum of T n-bit integers the space complexity of the prediction circuit is $O(T \cdot (n + \log T))$. The space needed to update all voting weights in parallel is $O(T \cdot n)$. It can also be shown that a circuit for calculating all of the s_t values has space complexity $O(T \cdot \log T)$, by combining redundant computations. Also, the space required to compute the two multiplications[9] needed to compute w_t given m_t is $O(\log^2 T)$ and since we will perform the T computations of w_t for different t in parallel the total space complexity of the w_t computations will be $O(T \cdot (\log T)^2)$. The total space required by online Arc-x4 is the sum of the space for the update and prediction circuits as well as the T hypotheses, which is $O(T \cdot (S_h + n + \log^2 T))$.

5 Detailed Empirical Results for Arc-x4

From here on we refer to our online variant of Arc-x4 as simply "Arc-x4". In this section we present empirical results using Arc-x4 to generate decision-tree ensembles for several online problems, using our online decision-tree base learner (described later in Section 6). We focus on Arc-x4 because our preliminary results (presented in Appendix A) indicate that our online bagging variants perform poorly in comparison to online Arc-x4. First, we perform online learning experiments that use ML benchmarks as the data source. Second, we describe the problem of branch prediction and show results of our experiments in that domain. Arc-x4 is shown to significantly improve prediction accuracy over single trees in most of our experiments. In addition, we show that boosting produces ensembles of small trees that are often able to outperform large single trees with the same number of nodes (and similarly outperform smaller ensembles of larger trees that use the same total number of nodes). This is particularly useful in the branch prediction domain where space is a central concern.

5.1 Machine Learning Datasets

One of our goals in this research is to evaluate online ensemble methods—this motivates our inclusion of results on familiar ML data sets.

9. The space complexity of multiplying two n-bit numbers is $O(n^2)$ (Cormen et al., 1997).

5.1.1 EXPERIMENTAL PROCEDURE

The ML benchmark datasets[10] we studied are usually used for offline learning experiments, but we convert them to an online setting as follows. First, we randomly divide a dataset into testing and training sets of equal size, twenty different times. This gives twenty different training/testing set pairs. We repeat the following procedure for each of these pairs and average the results of the twenty trials. Given a particular training/testing-set pair, for $N_{\max}$ iterations an example from the training set is randomly selected with replacement and used to update the online ensemble. After every S (sampling interval) updates of the ensemble, the testing and training error rates are calculated; the training/testing error is calculated by using the current ensemble to predict the class of each example in the training/testing set and recording the error rate. Thus, for a particular training/testing set pair we obtain $N_{\max}/S$ time sampled measurements of both training and testing error. After obtaining results from all twenty training/testing set pairs we calculate the mean and standard deviation of these error measurements. This results in training and testing plots of an ensembles mean error vs. time and standard deviation of error vs. time. Most of our plots use only the average testing error at the final sampling, in order to compare across varying ensemble sizes or space allocations.

5.1.2 DESCRIPTION OF DATASETS

Eleven-bit Multiplexor (multiplexor). The eleven-bit multiplexor concept uses eleven binary features. The feature vector bits are divided into three address bits and eight data bits for a total of 2048 distinct examples. The class assigned to a feature vector is simply the binary value of the data bit specified by the address bits. ID3 has been shown to perform poorly on this concept with respect to induced tree size (Quinlan, 1988). Utgoff (1989) showed that the online methods ID4 and ID5R can learn this concept (again with large trees).

Tic-Tac-Toe (TTT1 and TTT2). This problem is taken from the Tic-Tac-Toe endgame database at the UCI ML repository (Merz & Murphy, 1996). There are 958 instances each described by nine three-valued features that indicate the content of each square (x, o, or blank). The class of an instance is positive if the position is a 'win for x' and negative otherwise. The two versions of this problem we use correspond to different binary encodings of the features. TTT1 uses two binary features to represent the contents of each square for a total of 18 features. TTT2 is included as a more difficult version of this problem and uses eight bits to (ASCII) encode the contents of each square for a total of 72 binary features. This is a highly disjunctive concept and has been used as a testbed for offline constructive induction of decision trees in (Matheus & Rendell, 1989).

Vowel (vowel). This problem is taken from the letter recognition database at the UCI ML repository. There are 19976 instances each labelled by the letter it represents and described by sixteen four-bit integer features. We use a

10. These four benchmarks were selected to offer the relatively large quantities of labelled data needed for online learning as well as a natural encoding as binary feature space two-class learning problems. We have not run these algorithms on any other machine learning benchmarks.

naive binary feature representation that simply views each of the four relevant bits from the integers as a feature for a total of 64 features. We define the class of an instance to be positive if it is a vowel and negative otherwise.

5.1.3 Results For ML Benchmarks

We vary the number of trees T in an ensemble from 1 to 100, and the maximum allowable depth d of any tree[11] from one to ten. By varying d we change the representational strength of the trees. Figures 2 and 3 show the mean ensem-

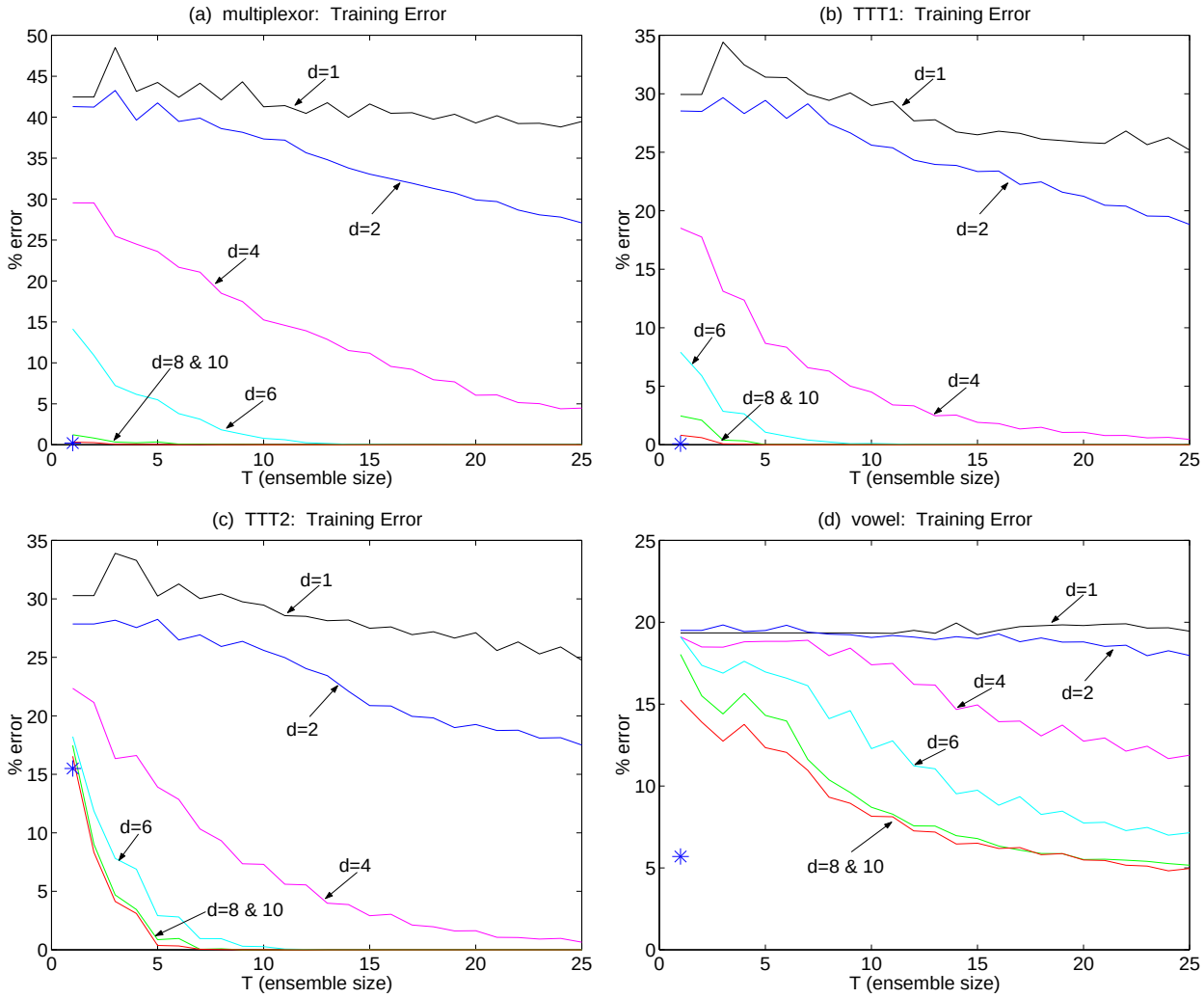

Figure 2: Final Training Error vs. Ensemble Size for the machine-learning benchmarks. (*Note that the x-axis is not time).* Results after ensembles encounter 50,000 training instances for multiplexor, TTT1, and TTT2 and 100,000 for vowel. The ensembles corresponding to a particular curve all have the same depth limit, as indicated on the graph. The stars indicate T=1 performance for unbounded depth (i.e., single trees).

11. The depth of a tree-node is the number of arcs on the path from the tree root to the node. The depth of a tree is the depth of its deepest node.

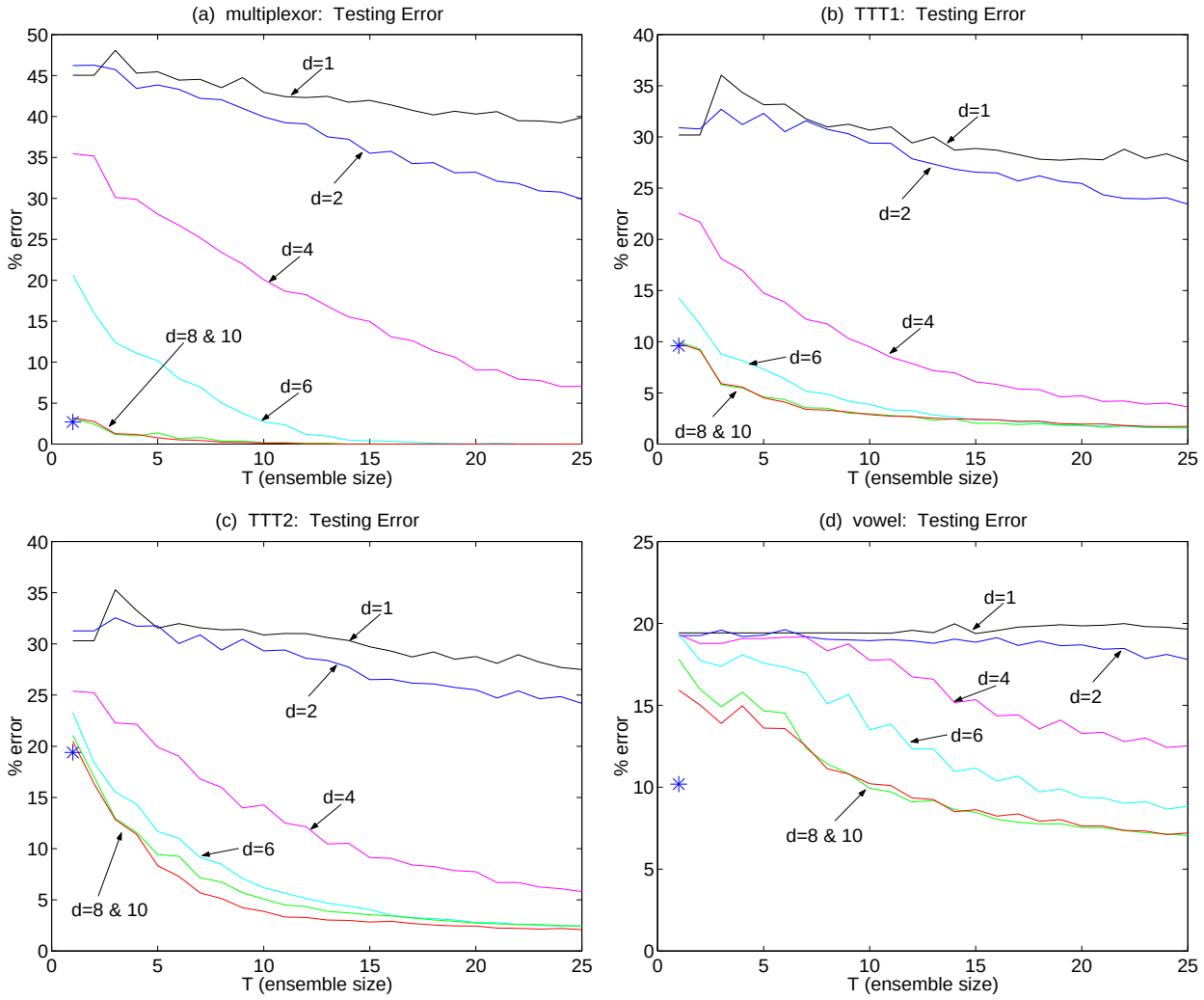

Figure 3: Final Testing Error vs. Ensemble Size for the machine-learning benchmarks. (*Note that the x-axis is not time).* Results after ensembles encounter 50,000 training instances for multiplexor, TTT1, and TTT2 and 100,000 for vowel. The ensembles corresponding to a particular curve all have the same depth limit, as indicated on the graph. The stars indicate T=1 performance for unbounded depth (i.e., single trees).

ble training and testing errors (respectively) versus T for the four benchmarks (averaged over different test/training divisions and different random online presentation sequences, as described above). We show results for T ranging from 1 to 25; the errors do not change significantly as T increases further. We used 100,000 training instances ($N_{\max}$ = 100,000) for the vowel dataset and 50,000 instances for the other datasets. The standard deviations are not shown, but were small relative to the differences in means. Each figure has six curves with each curve corresponding to ensembles that use a particular value of d.

Advantages with larger ensemble size. In all four problems, increased ensemble size generally reduces the training

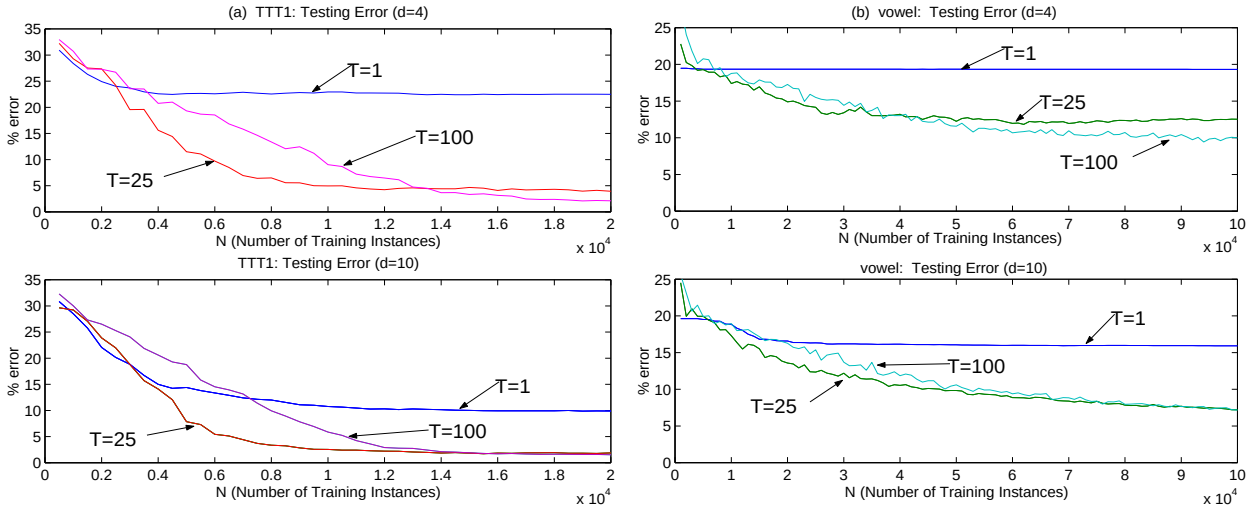

Figure 4: Performance versus time. Testing error vs. Number of instances (time) for two benchmarks. Each curve corresponds to a single ensemble and shows how its performance changes as more training instances are observed.

error effectively. It is particularly interesting that increasing ensemble size leads to significant improvements for TTT2, since we have hit an apparent performance limit for single trees with increasing depth as shown by the star on the graph (showing unbounded-depth single base-learner error)—for this problem the ensemble approach is of crucial value. The difficult problem encoding in TTT2 is completely overcome, but only with the use of ensembles.

It is also interesting to note that the weaker learners (low d values) are generally less able to exploit ensemble size. For instance, large ensembles of depth one trees perform very similarly to single trees on the vowel problem. This phenomenon is also observable in less extreme instances by comparing the slopes of the plots for varying depth bounds—we find a steeper error reduction with ensemble size for deeper trees. This observation indicates that ensemble learning is exploiting different leverage on the problem than increased depth. However, the ability of learners to exploit ensembles must also eventually fall off as we consider "stronger" base learners—once the individual learner approaches the Bayes optimal error it clearly will not benefit from ensembles.

Testing error graphs. As expected Figure 3 shows that the testing error is generally larger than the corresponding training error (as seen in Figure 2), but with the same trends. This indicates that Arc-x4 is producing ensembles that generalize the four concepts well. Also note that larger ensembles can yield improved testing error even when a small ensemble achieves zero training error—consider the d=10 curve for the TTT1 problem. This observation has been made in many empirical studies of offline boosting methods and is a a counterexample to the Occam's razor principle. The reason(s) for this phenomenon are still not fully understood (Grove & Schuurmans, 1998).

Warmup behavior. Figures 4a and 4b show how the error rate relates to the number of instances N used to update the ensembles (our measure of time) for the TTT1 and vowel problems. We show graphs for $d \in \{4,10\}$ and $T \in \{1,$

25, 100}. We observe that larger ensembles warm up more slowly than small ensembles—confirming the intuition that ensembles are particularly useful when many training instances are available and we are willing to wait for a slower warmup. However, we note that the smaller 25-member ensemble achieves nearly identical warmup performance to the single base learner (T=1) if we assume a parallel implementation (so that number of instances encountered is a fair measure of parallel time), but the 25-member ensemble achieves a strikingly lower error after warmup. These comments rely heavily on the assumption of efficient parallel implementation—the 25-member ensemble is nearly 25 times cheaper to train when implemented in parallel due to the highly parallel nature of ensembles. These results suggest that a 25-member ensemble is equal or superior to a single tree for any but the very smallest anticipated number N of training examples as long as the implementation is parallel. This comparison is useful in domains (such as branch prediction) where the number of training instances is unknown.

The warmup comparison between T=25 and T=100 indicates that there is a warmup price to pay for large ensembles—this price may be worth paying when large numbers of training instances are expected. It may be possible to exploit the warmup properties of smaller ensembles along with the increased accuracy of larger ensembles by providing a dynamic mechanism for selecting how many ensemble members to use for each prediction.

5.2 Branch Prediction Results

Our empirical work in this section has the immediate goal of exploring the utility of online ensemble methods to improve performance under time and space constraints when compared to single-learner methods. With this goal in mind, we limit our current exploration of the branch prediction problem to the same feature space used by most current state-of-the-art predictors (rather than exploit our linear growth in feature space dimension to consider additional processor state). Future work will explore the use of additional processor state to exceed the state of the art in branch prediction—this work will require an expensive empirical comparison to current techniques[12] over large benchmarks containing thousands of different branches (with millions of dynamic instances) to be considered convincing in the architecture community, and will be targeted to the architecture rather than machine learning community.

5.2.1 Experimental Procedure & Description of Datasets

Our branch prediction experiments were carried out by interfacing our tree-ensemble predictor to the trace-driven microprocessor simulator provided by the SimpleScalar 2.0 suite (Burger & Austin, 1997). We focused our study on eight branches from three different benchmark programs in the SPEC95 benchmark suite (Reilly, 1995)—the branches were selected because previous experiments indicated that single tree learners were performing significantly

12. Since simulations are carried out on serial machines the cost of running large simulations is proportional to the ensemble size and will take months on current high-performance multiprocessors.

TABLE 1. Information about branches used in our experiments.

Branch Name	# of Instances	% Taken	ideal-hybrid %error	Benchmark Program
go-A	413,908	35%	5.3%	
go-B	370,719	47%	20.3%	**go:** This program plays the game of go.
go-C	407,380	33%	14.4%	
go-D	451,042	57%	14.1%	
li-A	2,653,159	20%	5.4%	**li:** This program is a lisp interpreter
li-B	1,238,803	71%	1.6%	
com-A	253,031	56%	5.9%	**com:** This is the unix compress program
com-B	17,104	75%	3.1%	

worse than Bayes optimally on these branches. Table 1 provides information about the benchmark programs used and the branches selected from these benchmarks. The ideal-hybrid accuracy column presents for reference results for the highly specialized "hybrid" predictor from computer architecture (McFarling, 1993)—it is not our immediate goal to exceed these results with our general-purpose approach. Because the "hybrid" predictor scales exponentially in space with number of features considered, we expect that our general-purpose approach will open new avenues of leverage on the problem in the long term. The experiments we present are "interference-free", meaning that each predictor is only updated by and used to predict the outcomes of a single branch. In the "interference-free" setting the hybrid predictor is arguably among the best proposed predictors that restricts itself to using only local and global history bits as features—this predictor is often used in the branch prediction literature to evaluate new designs and a variant has been implemented in the Alpha 21264 microprocessor (Gwennap, 1996), making it arguably the state-of-the-art implemented branch predictor. Many newer branch prediction architectures have been proposed, but almost all of these aim at extending table-based methods to better deal with branch aliasing[13] (e.g., the YAGS predictor (Eden & Mudge, 1998) and references therein) which is orthogonal to the focus of our current work.

We assigned each selected branch its own tree ensemble predictor. We then simulated the benchmark programs to completion, arranging that when a selected branch was encountered (during prefetch) its ensemble was queried for a prediction (using the current state to determine the branch features), and then later updated with the true branch outcome when the branch is resolved.[14] We counted the number of errors made by each ensemble and computed a final percent error by dividing by the total number of occurrences of the branch. The percent errors given for the ideal-hybrid predictor were arrived at as follows: first, for a particular branch we simulated the hybrid predictor for all sizes (measured in number of global and local history bits used to index the table) ranging from 2 to 50 (which was the

13. Branch aliasing is when a predictor is used to predict the outcomes and learn from branch instances from two different branch instructions.

14. In an out-of-order processor an instruction is referred to as resolved (committed) when the processor state has been updated with the result of its execution. Otherwise the instruction is unresolved.

maximum size we could simulate due to memory limitations). We then recorded the error rate for each of these predictors and reported the smallest of these rates as the ideal-hybrid error rate. This results in an optimistic estimate of how well a particular hybrid predictor would perform when encountering these branches.

5.2.2 RESULTS FOR ONLINE ARC-X4

We varied ensemble size (T) and tree depth (d) from 1 to 100 and from one to twelve, respectively. The features used by the predictors included the most recent 32 global and 32 local history bits for a total of 64 binary features.

Basic Arc-x4 Performance. Figures 5a-5f give the percent error versus ensemble size for six branches, with curves plotted for six different depth bounds (d). Each graph also shows (with a star) the percent error achieved by a single online tree of unbounded depth and the error achieved by the ideal-hybrid predictor (with a dotted line). The curves in these graphs exhibit some of the same trends as the results above for the machine learning benchmarks. In general as T increases the errors tend to significantly decrease with respect to the error of single trees.

Particularly interesting are branches such as go-A and go-B where there is very little improvement for single trees when d is increased from 12 to ∞. By increasing the ensemble size, however, we are able to further decrease the error. As was the case for the vowel and TTT2 benchmarks, the extra dimension of ensemble size appears to be crucial to improving the error rate by exploiting additional space when using an online decision tree algorithm.

Small ensemble effects. When compared to the curves from the ML benchmarks the branch prediction error curves exhibit more erratic behavior particularly for small d and T values. One conjectured explanation for this erratic behavior is that trees that are close to each other will see similar weighted input sequences and hence may form similar hypotheses. Some of these weighted sequences may be particularly hard for the trees and therefore we may get several trees that are similarly bad. When T is small these trees are more likely to be able to incorrectly determine the prediction. We note that when unweighted voting is used (not shown) the erratic behavior for small T becomes worse. As T increases, however, the performance of unweighted voting and weighted voting are nearly identical.

Large ensemble effects. We note that in many cases there is a slight increase in error rate as we approach large ensemble sizes. We attribute this increase to the fact that later ensemble members generally take a longer time to warm-up than earlier members. For a fixed number of training instances it is possible that the later members of large ensembles do not adequately warm-up. Despite this lack of warm-up, our scheme still uses the later members to make predictions—degrading the error rate for ensembles too large for the training set size. This hypothesis is based on experience with the ML-dataset experiments which showed the error increased for the larger ensemble sizes when the number of training instances was small, but the increase disappeared for a large enough training set (this information can be inferred from Figure 4). These observations suggest future study of dynamic ensemble size selection.

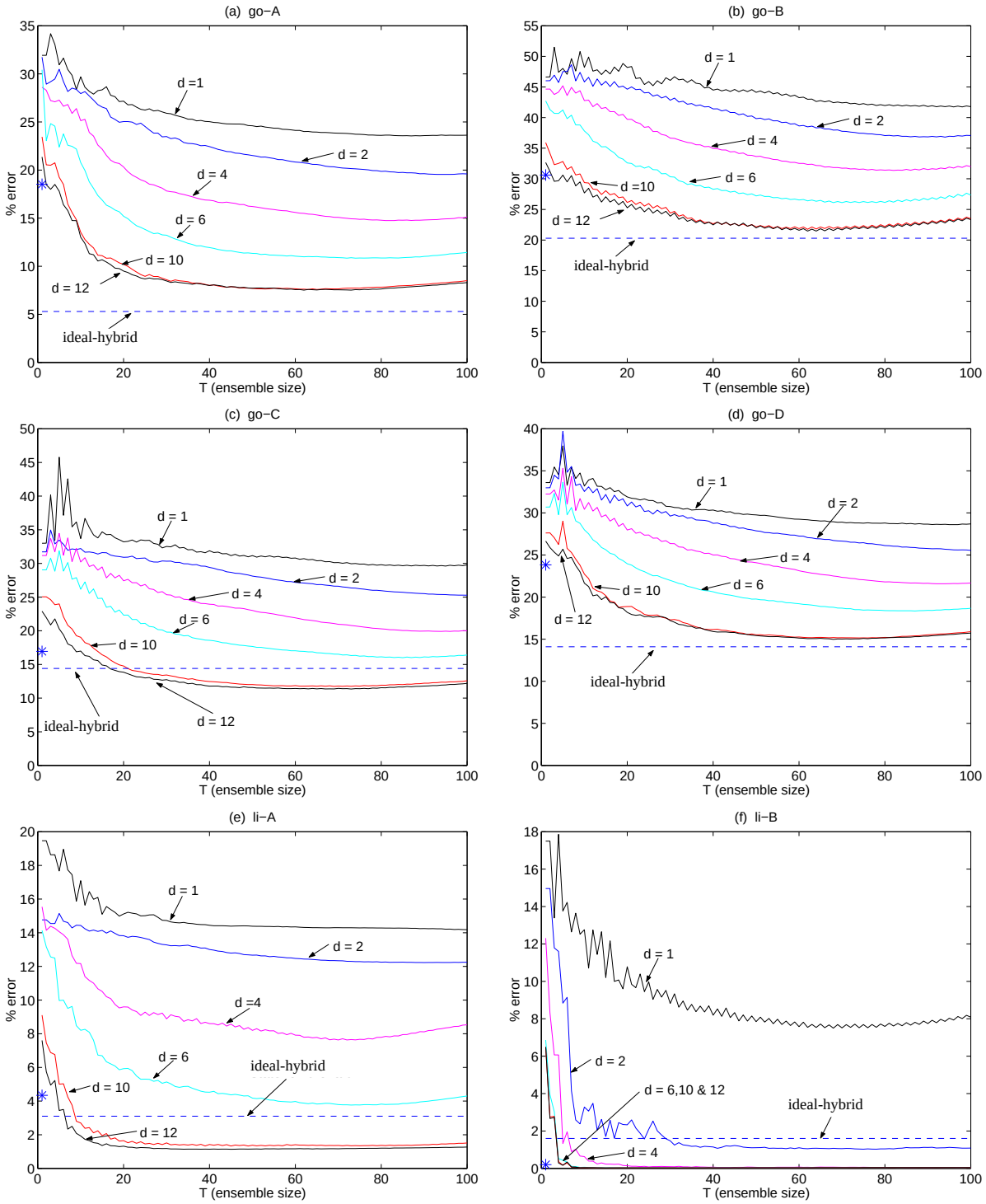

Figure 5: Percent Error vs. Ensemble Size for six <u>hard</u> branches. Each curve corresponds to ensembles of varying size with the same depth limit. The stars on each graph show the error achieved by a single tree with unbounded depth.

Comparing to ensemble size one. The graphs in Figure 5 all exhibit a similar trend with respect to increasing *d*. For small *d*, large ensembles are needed to see much benefit, but the eventual benefits are larger (than when *d* is large). However, every curve shows peak performance at an ensemble size larger than one. The stars on the graphs showing the best single tree performance indicate that bounded-depth ensembles can be used to outperform single trees of any depth for every branch tried except li-B.

Moreover, even if there were no such advantage, our examination below of the performance obtained for a fixed space usage indicates that ensembles of smaller trees outperform single deep trees with the same space requirement.

Achieving same error using less space. Note that in the branch prediction domain we are assuming a hardware implementation. For such implementations, it is not easy to dynamically allocate tree nodes[15]—thus, we must provide space for full bounded-depth decision trees. Therefore, our predictors necessarily grow exponentially with the tree-depth bound. This implies that doubling the number of trees in an ensemble is roughly size-equivalent to increasing the depth bound of the trees in the ensemble by one. Using ensembles allows us to more effectively use the limited space by trading off depth for more trees.

The graphs in Figures 6a-6f show the percent error versus the logarithm of the ensemble tree-node count, giving a basis for selecting d and T to optimize accuracy when facing space constraints. Each curve corresponds to ensembles with a particular depth-bound d. An ensemble of T base learners of bounded-depth d has $T \cdot (2^{d+1} - 1)$ non-leaf nodes. Here we assume the number of nodes is linearly related to the physical size of an ensemble when implemented in hardware (e.g., on a silicon microchip).

Note that as *d* increases the ensemble curves shift to the right. Generally for a fixed number of nodes the best percent error is achieved by an ensemble with *T* greater than one. This observation is strongest for the go branches and weakest for the li-A branch. For example on each graph consider ensembles with approximately 1000 nodes, and determine which depth-bound *d* gives the lowest error—the smaller *d* values correspond to larger ensembles (to get to 1000 nodes). It is clear that for five of the six branches, ensembles with depth four (and thus *T*=66) achieve the best percent error when we are constrained to 1000 nodes. The graphs indicate that in many cases when facing size constraints, ensembles are preferable to single trees—given any space constraint larger than 400 nodes (and often much less), all of our branches show the best performance for an ensemble size larger than one.

Now suppose that instead of a size constraint we are given a maximum percent error constraint. From these graphs we can see that using ensembles often allows us to achieve a particular percent error using a much smaller

15. Note that a software implementation need only allocate space for nodes that are actually encountered during learning. This technique would be very difficult to implement in hardware. A similar space-equivalence study could be conducted for the software implementation, but we have not done so here.

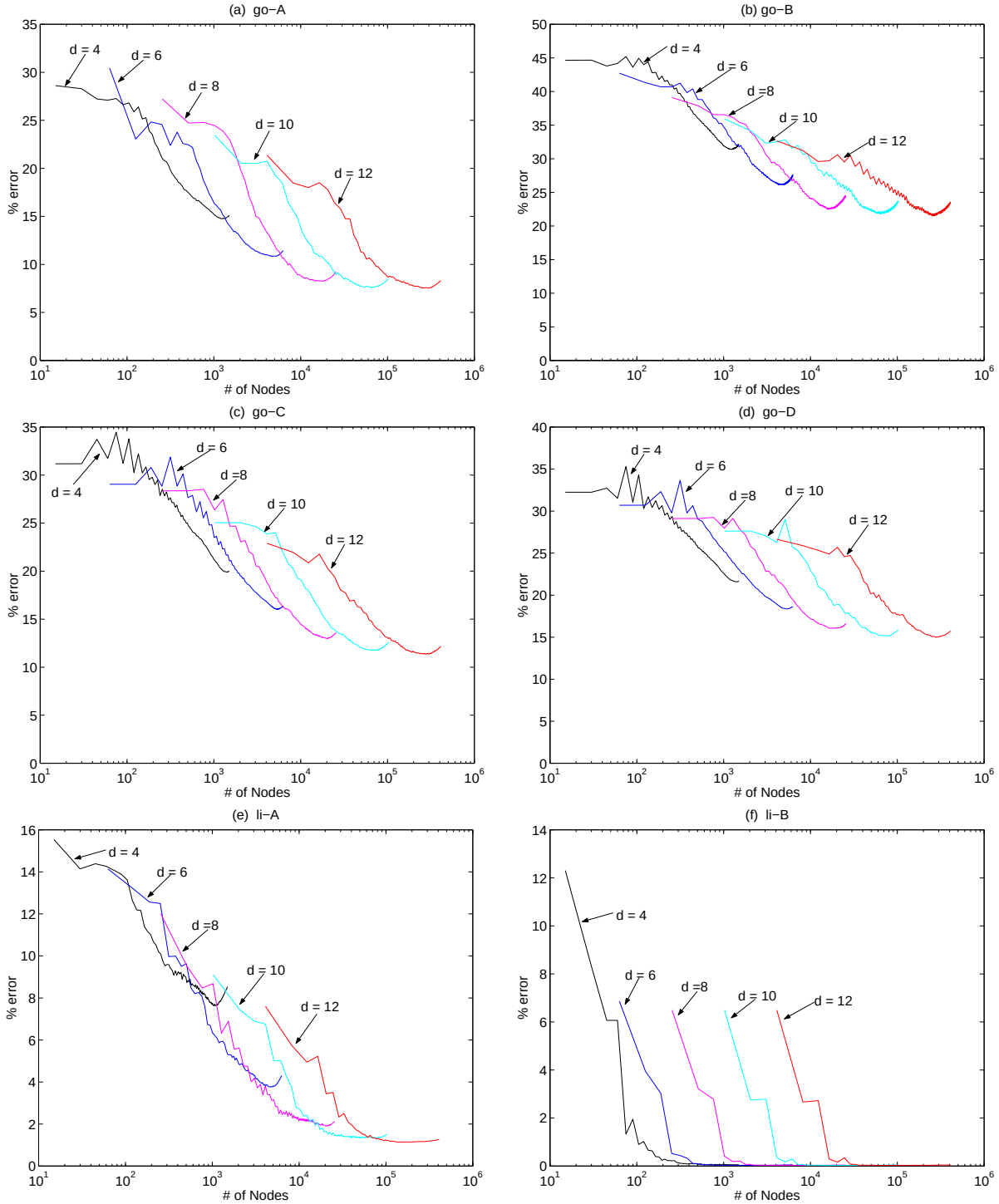

Figure 6: Percent Error vs. Tree Nodes Allocated for six <u>hard</u> branches. Each curve corresponds to ensembles of varying size made up of trees with the same depth on every path (our hardware implementation requires static tree node allocation).

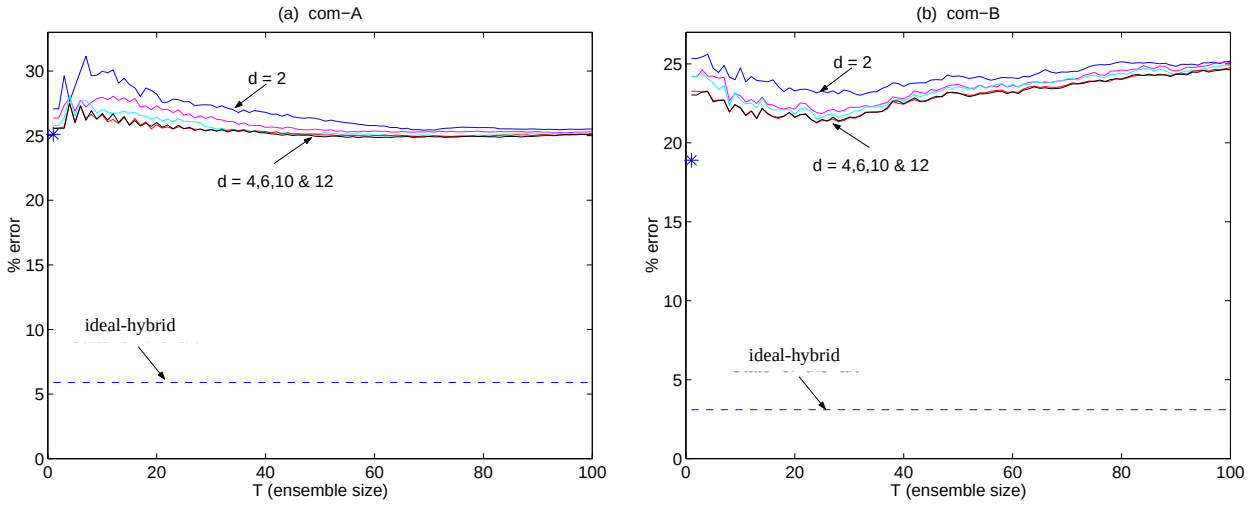

Figure 7: % Error vs. Ensemble Size for two branches where performance was poor. Each line on a curve corresponds to ensembles that all use the same depth limit, as labelled.

number of nodes by using large ensembles of small trees rather than smaller ensembles of larger trees (in many cases, a single tree cannot achieve the desired error at any depth). All of these observations suggest that online boosting may be particularly advantageous in online domains that place tight space constraints on the predictors.

Poor performance on com benchmark branches. Two of the eight branches we studied showed relatively poor performance for Arc-x4 ensembles. Figures 7a and 7b show the error rate versus T plots for the com-A and com-B branches. The error rates for com-A tend to increase for the smaller values of T and then begin to decrease as T becomes larger. The error reduction achieved for these large values of T is relatively small compared with the other six branches. These results are particularly interesting because we know a much better error rate is possible since the table-driven "hybrid" branch predictor used in computer architecture today (see (Fern & Givan, 2001) for a description of the hybrid predictor) achieves a much better error rate as indicated on the graph. Currently we do not have an adequate explanation for the poor performance on the com-A branch, however the poor performance for the unbounded-depth single base learner (shown by the star on the graph) suggests that this concept is not well captured by the online decision trees used as base learners here (the same explanation applies as well for com-B). The com-B branch shows the opposite trend: the error rate exhibits a bowl shaped trend and for large values of T is actually larger than the error rates of single trees. Again we know a much better error rate is possible as demonstrated by the ideal-hybrid predictor results. The com-B error curve suggests that the larger ensembles have not had enough training data to warm up (see Figure 4). Table 1 shows that the number of occurrences of com-B is an order of magnitude less than the other branches suggesting the ensembles have not observed enough examples to stabilize. Further training data on the concept would likely improve the larger-ensemble performance; however, it appears that the ideal-hybrid performance is

again out of reach of this approach.

Comparison to ideal-hybrid branch predictor. We wish to emphasize that the branches used in these experiments were selected because of the poor performance of single base learners in our previous experiments (not shown here), as compared to the "hybrid" branch predictor. On many branches not shown, single decision-tree base learners can equal or exceed the performance of the hybrid predictor. Considering branch predictor performance averaged over entire benchmark programs with millions of different branches (one target concept for each), the single base-learner is only slightly inferior to the hybrid predictor. It is our expectation that incorporation of ensemble methods into branch prediction can lead to better performance than the hybrid predictor. It is not our goal here to demonstrate this improved performance, but rather to explicate the methods and effects of ensembles in online settings. Clearly evaluating branch predictors across large benchmarks is extremely compute-intensive.[16]

We also note again that the hybrid predictor scales exponentially in space usage with the number of branch features considered. As a result, practical instances of this predictor typically use a very small number of features (less than sixteen bits out of thousands of bits of processor state), and are highly specialized to using those features well. Because our ensemble/decision-tree approaches avoid exponential scaling in space usage, we hope to dramatically improve branch prediction performance in future studies by cheaply incorporating a larger portion of processor state in learning and making predictions (our approach scales linearly in the number of features considered).

6 Online Decision-tree Induction

In this section we describe the novel online decision-tree learning algorithm (a variant of ID4) that we used as the base learner in our ensemble experiments—our extensions to ID4 prove important to single-tree accuracy and more importantly critical to ensemble accuracy. Empirical results shown in Section 7 suggest that smooth and rapid warmup behavior is important for the base learners in our ensembles, and that ID4 without our extensions often lacks smooth rapid warmup.

Most decision-tree algorithms are designed for offline settings, as in the well-known ID3 algorithm (Quinlan, 1986); however, there has also been previous research into online decision tree algorithms, with two key proposed methods being ID5R (Utgoff, 1989) and ID4 (Schlimmer & Fisher, 1986). ID5R is an online algorithm that is guaranteed to produce the same tree as offline ID3 when applied to the same training instances. To guarantee ID3-equivalence ID5R must store all of the previous examples, and restructure the tree as new examples arrive. Empirical work has demonstrated that the ID3-equivalent tree produced by ID5R often outperforms the ID4-generated trees that avoid

16. Since simulations are carried out on serial machines the cost of running these simulations is proportional to the ensemble size and will take months on current high-performance multiprocessors.

storing instances, given a limited amount of training data (Utgoff, 1989). In this sense, ID5R online learners "warm up" more quickly than ID4 learners. In online domains where it is not practical to store all of the training instances, some mechanism must be used to select what instances to store for ID5R, and ID3-equivalence is lost—in such domains it is not clear if the advantages relative to ID4 persist.

While ID5R avoids the full cost of running the offline ID3 algorithm as every instance arrives, the tree restructuring operations that must be supported can be expensive and somewhat complex for use in resource-bounded online settings. In addition, although the recursive restructuring operation is straightforward to implement in software, our motivating domain requires a hardware implementation that appears quite difficult for ID5R—the same can be said for the related and more recent ITI algorithm (Utgoff et al., 1997). For this reason we chose to use a variant of the simpler online decision tree algorithm ID4, described below. ID4 does not store training instances and does not perform any complicated tree restructuring. Below we describe ID4 and our extensions to it.

6.1 The ID4 Online Decision-tree Induction Method

We assume familiarity with the offline top-down-tree-induction approach of ID3 (Quinlan, 1986). A key difference between ID3 and ID4 is that ID4 incrementally maintains an estimate of the split criterion[17] of each feature at each tree node as it observes new training instances. ID3 on the other hand calculates the split criterion for each feature at each node once, based on a fixed offline set of training examples. Of the commonly used split criteria, accuracy is by far the simplest to compute incrementally (particularly from a hardware perspective). Because our motivating application requires a hardware implementation, we use accuracy as the split criterion in this research.

In order to dynamically monitor the usefulness of each candidate split feature at each tree node and dynamically select the best feature over time, ID4 maintains certain data structures at each tree node (in addition to child-node pointers). For each node N, these data structures include the observed split criterion values for each feature i, denoted $N.SC[i]$, the currently selected split feature index $N.SplitFeature$ (an element of {1, ..., m} where m is the number of features), and a leaf indicator $N.leaf?$ which is true if the node is a tree leaf. Given values for these data structures and a particular unlabeled query, the prediction returned is the majority class of the leaf reached by starting at the root node and following a path down the tree determined by the currently selected split features at each node encountered in combination with the features of the query.

Figure 8 shows pseudocode for a weighted version of the original ID4 online decision-tree update algorithm. A decision tree is updated by invoking this procedure with the root node of the tree, a training instance I, a weight for the training instance w, and a depth limit d. Unbounded-depth trees can be induced by setting the depth limit equal to

17. The split criterion of a feature at a tree node assigns a numerical value to the feature based on some estimate of the desirability of splitting based on that feature at that tree node—e.g. information gain, gini index, or prediction accuracy.

the number of features. The procedure recurses through the tree, updating the split criterion information stored at each node and selecting new split features and/or pruning when indicated. The algorithm proceeds as follows.

- (lines 1–2) First, the split criterion information stored at the root node is updated based on the new training instance and the weight. The function UpdateSplitCriterion(*N*, *I*, *w*, *i*) incrementally maintains the value *N.SC*[*i*] based on instance *I* and weight *w*—for accuracy, this amounts to updating a weighted count of the number of correct predictions made by feature *i* at node *N* over time (as well as the total weight of instances encountered at *N*).

```
Procedure: ID4Update (root, I, w, d)
Input:
   subtree root node  root
   new training instance  I = ⟨x, c⟩
   training instance weight  w
   depth limit  d

1.   for each i ∈ {1, 2, …, m} ,                          ;; possibly executed in parallel
2.        do  root.SC [i] = UpdateSplitCriterion (root, I, w, i)   ;; update split criterion for feature i
3.   root.leaf? = LeafTest (root, d)                      ;; determine if root is a leaf
4.   if (NOT root.leaf?)                                  ;; stop if root is a leaf
5.        then  newSplit = indexOfMax(root.SC[1], ..., root.SC[m]) ;; find the best split feature
6.              if (newSplit ≠ root.splitFeature )        ;; if the split feature changed
7.                    then { root.splitFeature = newSplit
8.                           PruneSubtrees (root) }      ;; discard the subtrees when changing split feature
9.              ID4Update (SelectChild (root, I), I, w, d)   ;; update the appropriate child
```

Figure 8: Weighted ID4 online decision tree update procedure (without extensions).

```
Procedure: ID4Update2 (root, I, w, d)                    ;; comments highlight differences from Figure 8
Input:
   subtree root node  root
   new training instance  I = ⟨x, c⟩
   training instance weight  w
   depth limit  d

1.   for each i ∈ {1, 2, …, m, m + 1} ,                  ;; feature m+1 is the subtree monitoring feature
2.        do  root.SC[i] = UpdateSplitCriterion (root, I, w, i)
3.   root.leaf? = LeafTest2 (root, d)                    ;; LeafTest2 must implement post-pruning by
                                                         ;;    subtree monitoring (see text).
4.   ;; Note: we continue updating even if root is a leaf for
     ;; predictions in order to implement advanced warmup
5.   newSplit = indexOfMax (root.SC [1], ..., root.SC [m+1])
6.   if (newSplit ≠ root.splitFeature ) and (newSplit ≠ m + 1 )   ;; don't change split feature if subtree is winning
7.        then { root.splitFeature = newSplit
8.               PruneSubtrees (root) }
9.   if (depth(root) < d)                                ;; root.leaf? indicates leaves for prediction, so
          then  ID4Update2 (SelectChild (root, I), I, w, d)   ;;    termination here is based directly on depth
```

Figure 9: Extended version of the weighted ID4 online decision tree update procedure.

- (line 3) Next, the function LeafTest determines whether the node should be treated as a leaf or a decision node. In the original version of ID4 (Schlimmer & Fisher, 1986) LeafTest returns true if the feature with the best split criterion passes an χ^2-test of independence and false otherwise—this is the pre-pruning mechanism used by ID3. In Utgoff's (1989) version of ID4 there is no pruning and LeafTest is true when all of the instances observed at the node have the same class and false otherwise. In comparing the original ID4 to our extended ID4 (for the results shown in Section 7, we used Utgoff's version of ID4[18]). In bounded-depth settings, a node is also determined to be a leaf if its depth is equal to the specified depth limit.
- (line 4) Next, if the *root* node was determined to be a leaf then the update procedure is finished.
- (line 5) Otherwise, the procedure finds the feature (*newSplit*) with the best split criterion value *root*.*SC*[*newSplit*]. The function indexOfMax() accepts a sequence of split criterion values and returns the index of the best one.
- (lines 6–8) If the best feature is different from the current split feature then the split feature is changed and the subtrees are pruned. The function PruneSubtrees(*root*) deletes both subtrees (if any) of node *root*.
- (line 9) Finally, if the node is not a leaf then the update routine is recursively called to update either the left or right subtree of the root based on the value of the split feature in the training instance. The function SelectChild(*root, I*) selects the appropriate child, creating and initializing that child if it does not already exist.

A critical feature of the ID4 algorithm is its pruning of subtrees whenever a split feature changes. The reason for this pruning is that the split criterion information maintained in the subtree nodes is conditioned on the value of the current root split feature. When the root split feature changes, the subtree split criterion data is no longer conditioned on the correct feature. The ID4 algorithm prunes the subtrees rather than use the incorrect split criterion data. In domains where two or more features at a particular node have similar split criterion values, the split feature is likely to frequently switch between the competing features—as a result the subtrees will be repeatedly pruned. Generally we expect such split feature switching to occur most frequently when a node has observed a "small" number of examples. This is due to the high variance in the split criterion estimates at that point. As more examples are observed the variance decreases and if there is a unique best feature it will eventually prevail over the others. But until that point the subtrees of a node may be pruned at anytime by ID4 and the information from all previous training examples will be lost. This is one reason ID4 often takes a longer time to "warmup" than ID5R

6.2 Extensions to ID4

Here we describe three extensions to ID4 that improved its performance significantly in both single tree and ensemble experiments (see Section 7). The pseudocode for our extended ID4 is shown in Figure 9 and described below.

Advanced warmup (AW). In the ID4 algorithm when a node is a leaf, the node has no children and hence no data is sent below it during a tree update. This means that after a node selects a split feature (i.e., becomes a non-leaf) its

18. Preliminary experiments did indicate that the χ^2-test pruning mechanism performed similarly to the pruning at class purity in Utgoff's version; however, a full comparative evaluation of χ^2 pruning was beyond our means because it requires careful study of the ideal confidence parameter setting as well as storing a larger amount of data at each tree node and additional floating point calculations (this data storage and calculation is repeated millions of times over in our experiments—essentially once per feature per tree-node per ensemble-member per parameter setting per experiment).

children must begin learning from scratch. One way we extend ID4 is to allow for *advanced warmup*—we allow nodes that behave as leaves during prediction to have descendents that are continually learning from examples (even though they are not used for predictions). Thus, when a leaf node becomes a non-leaf node its subtrees have already had the opportunity to "warmup".

To implement advanced warmup we make a couple of changes to ID4. First, leaf nodes will have split features—simply the feature with the best split criterion value. These split features are used only during tree updates and not when making predictions. Additionally, we assume bounded-depth trees, and will always store a full tree of the maximum specified depth.[19] To make a prediction with the tree a path from the root is followed based on the split features at each node until a node marked as a leaf is reached—at this point a prediction is made. To update the tree a training instance is sent down a single path (based on the split feature at each node) from the root to a node at the maximum tree depth. The pseudocode in Figure 9 reflects this change in the absence of the conditional statement in line 4 (compare Figure 8), so that the extended ID4 is called recursively until the maximum depth is reached. For each node along the path the split criterion data is updated (line 2), the leaf bit indicator is set as appropriate (line 3), and the split feature is selected (line 5). Note that when the selected split feature at a node changes, the subtrees of the node are pruned—the function SelectChild will automatically create those subtrees again as needed.

Post-pruning by subtree monitoring (PP). Recall that the decision to make a node a leaf in the original ID4 is determined by a χ^2-test on a potential split feature. This form of pruning is known as *pre-pruning* since it makes a pruning decision before subtrees are constructed. When using advanced warmup, however, it is possible to monitor the performance of subtrees and to use that information to decide whether to prune or not. In the decision tree literature this is known as *post-pruning*. To implement post-pruning, we maintain at each node an estimate of the split criterion value that would be obtained making predictions with the subtrees. For notation uniformity, we introduce the notion of a *subtree monitoring "feature"*—this is an additional instance "feature" (with index m+1) given by the prediction of the subtrees when selected according to the current split feature.The split criterion value of this feature is then denoted by $root.SC[m+1]$, and is set in line 2 by treating the prediction returned by the selected subtree as feature m+1.

Unlike the other instance features, the value of the subtree monitoring "feature" is dependent on the tree node (and the tree itself)—hence the scare quotes on "feature". Specifically the value of a node's subtree monitoring feature is found by selecting the appropriate subtree of the node based on the current split feature value (even nodes marked as leaves have split features) and then querying that subtree for a prediction—i.e., the subtree monitoring feature is set equal to the prediction of the first leaf node encountered in the selected subtree. The function UpdateSplitCriterion must compute the subtree monitoring feature when called with index m+1. The function LeafTest2(*node, d*)

19. Actually, we only create a given node when there is some training instance that will need to update that node—this has allowed us to conduct experiments up to a depth bounds of 64. However, this approach is difficult to implement in silicon for our branch prediction application.

implements post-pruning by subtree monitoring—this function returns true if and only if the subtree-monitored criterion value *node*.*SC*[*m*+1] is worse than the criterion value obtained by treating the node as a leaf that uses the majority class to make a prediction[20], otherwise it returns false and the node is a decision node. This pruning mechanism is considerably less expensive to implement than the χ^2-test when accuracy is used as the criterion. Post-pruning by subtree monitoring is similar in spirit to the virtual pruning mechanism used in the ITI incremental decision tree induction algorithm (Utgoff et al., 1997). In ITI, previous training instances are stored and used to compute a pruning heuristic based on minimum description length—when a node is made a leaf its subtrees are not destroyed and hence it is referred to as a virtual leaf, subsequent pruning decisions may take these virtually pruned subtrees into account.

Feature-switch suppression by subtree monitoring (FS). Our third extension attempts to reduce the detrimental effect that subtree pruning has in the original ID4 algorithm. Recall that ID4 prunes the subtrees of a node whenever the current split feature at the node changes. It is possible that the pruned subtrees are performing well and that the leaf node resulting from pruning will have much lower accuracy—nevertheless these subtrees are discarded because the current split feature is slightly worse than the newly selected feature (using the split criterion). If a node has several features with similar split criterion statistics, this pruning can occur repeatedly, preventing the use of deeper and more accurate subtrees. We address this problem by refusing to change the split feature of a node unless the candidate new split feature *i* is better than the subtree-monitoring feature, comparing split criterion values (i.e., comparing *root*.*SC*[*i*] and *root*.*SC*[*m*+1]). In other words, we refuse to adopt a new split feature if the predictions given by the current subtrees are preferable to the predictions of the candidate feature as a leaf—this extension suppresses some feature switching that would otherwise occur. This extension to ID4 is shown in the more restrictive conditional test in line 5 of Figure 9 (compare Figure 8). With the stronger condition the split feature is changed and the subtrees are pruned only when the new split feature has a higher split criterion value than that of the subtree-monitoring feature.[21]

7 Empirical Results Evaluating Our ID4 Extensions

Here we show empirical results comparing the performance of different ID4-based base learners when training online ensembles using Arc-x4—these results show a significant improvement in accuracy for all ensemble sizes (including single trees) when ID4 is modified with our suggested extensions, and indicate that our extensions are particularly important for ensemble performance. We include results that suggest that Arc-x4-ensemble performance critically relies

20. This requires tracking the criterion value of the majority class prediction at each node. We omit these straightforward details for clarity.

21. There may be cost to adopting the FS extension in that it is possible to select a feature that does not optimize the split criterion before the split criteria statistics being collected have converged, and then refuse to later switch to a locally better feature because the subtrees have developed enough to add enough accuracy that the locally optimal feature cannot beat them. We expect that this variant of ID4 is significantly less likely to converge to the tree produced by ID3 on the same data in the infinite limit. Our empirical data indicate that the advantages of FS outweigh this possible disadvantage in our domains.

on smooth and rapid warmup behavior in the base learners. The ID4 variants we consider include different combinations of the extensions described above in Section 6. The experimental protocols used here, as well as many details of the experimental benchmarks and methods, are all the same as those in Section 5.

In the following we will refer to the ID4 algorithm described by Utgoff (1989) as simply ID4 (i.e., this algorithm differs from the original version of ID4 (Schlimmer & Fisher, 1986) by replacing the forward pruning χ^2-test with a test for class purity). The algorithm is described in Section 6.1, and accuracy is used as the split criterion. We will abbreviate the three ID4 extensions by: *AW for advanced warmup, PP for post-pruning by subtree monitoring, and FS for feature-switch suppression by subtree monitoring*. In our experiments we will distinguish between different combinations of these extensions by showing the abbreviations for the included extensions (e.g., the version of ID4 that includes all three extensions as given by the pseudocode in Section 6.2, Figure 9 will be referred to as ID4(AW,PP,FS)). The experimental results we show are for pure ID4, ID4(AW,PP), ID4(AW,FS), and ID4(AW,PP,FS). We found that the performance of ID4 and ID4(AW) were not significantly different and likewise for ID4(FS) and ID4(AW,FS); also note that no ID4(PP) experiments were run since the PP extension requires AW.

Figures 10a-10i show the percent error versus ensemble size (T) curves for nine of our benchmarks. The go-D benchmark results are not included but are similar to the go-C results. Each graph has four curves that correspond to ensembles trained with the online Arc-x4 algorithm using ID4 as the base learner with four different combinations of the extensions as labelled on the graph. A depth limit of twelve was used for all of these results.

Single-tree performance. The leftmost point on each curve shows that the single-tree performance with our extensions is moderately but significantly better than the unextended algorithm, and indicates that the feature-switch suppression (FS) extension is particularly important in single tree learners. In all of the single-tree experiments ID4 is outperformed by ID4(AW,PP,FS)—typically the extensions reduce the single tree percent-error by a fifth. Also, in most cases the single tree performance of ID4(AW,FS) is very near to that of ID4(AW,PP,FS) and is always better than ID4(AW,PP). This supports our claim that feature-switch suppression (FS) is crucial to improving the single-tree performance—much more so than the post-pruning (PP) extension. Apparently discarding subtrees due to changing split features (the issue addressed by the FS extension) significantly degrades single-tree performance.

Performance in online Arc-x4 ensembles. When considering ensembles of ID4 base learners, the curves in Figure 10 show a substantial benefit for the combination of all three extensions, revealing that the post-pruning PP extension yields benefits in ensembles that are not seen in single base learners. Compare the ID4 and ID4(AW,PP,FS) curves—in many cases when ID4 is used as the base learning algorithm very little benefit is observed by increasing the ensemble size and in some cases the performance degrades with ensemble size even for small sizes. In contrast, ID4(AW,PP,FS) nearly always yields substantial improvements in accuracy with ensemble size. Also observe that in most cases for small ensembles feature-switch suppression (FS) is more important than post-pruning (PP), but as the

ensemble size is increased ID4(AW,PP) usually outperforms ID4(AW,FS). Not only is the fully extended ID4 usually the best curve shown, we note that often a single ID4(AW,FS,PP) tree outperforms substantial ensembles of the other variants. In other cases (e.g. TTT1, TTT2), the extensions are of relatively little value to single trees, but prove critical to achieving low percent error with ensembles—often these benefits are seen only with post-pruning (PP).

Why do the extensions help? We hypothesize that one way the PP and FS extensions improve ensemble performance is that they combine to produce trees that warmup more quickly and more smoothly. Warmup performance is particularly important in ensembles because later ensemble members already pay a warmup-rate penalty because they

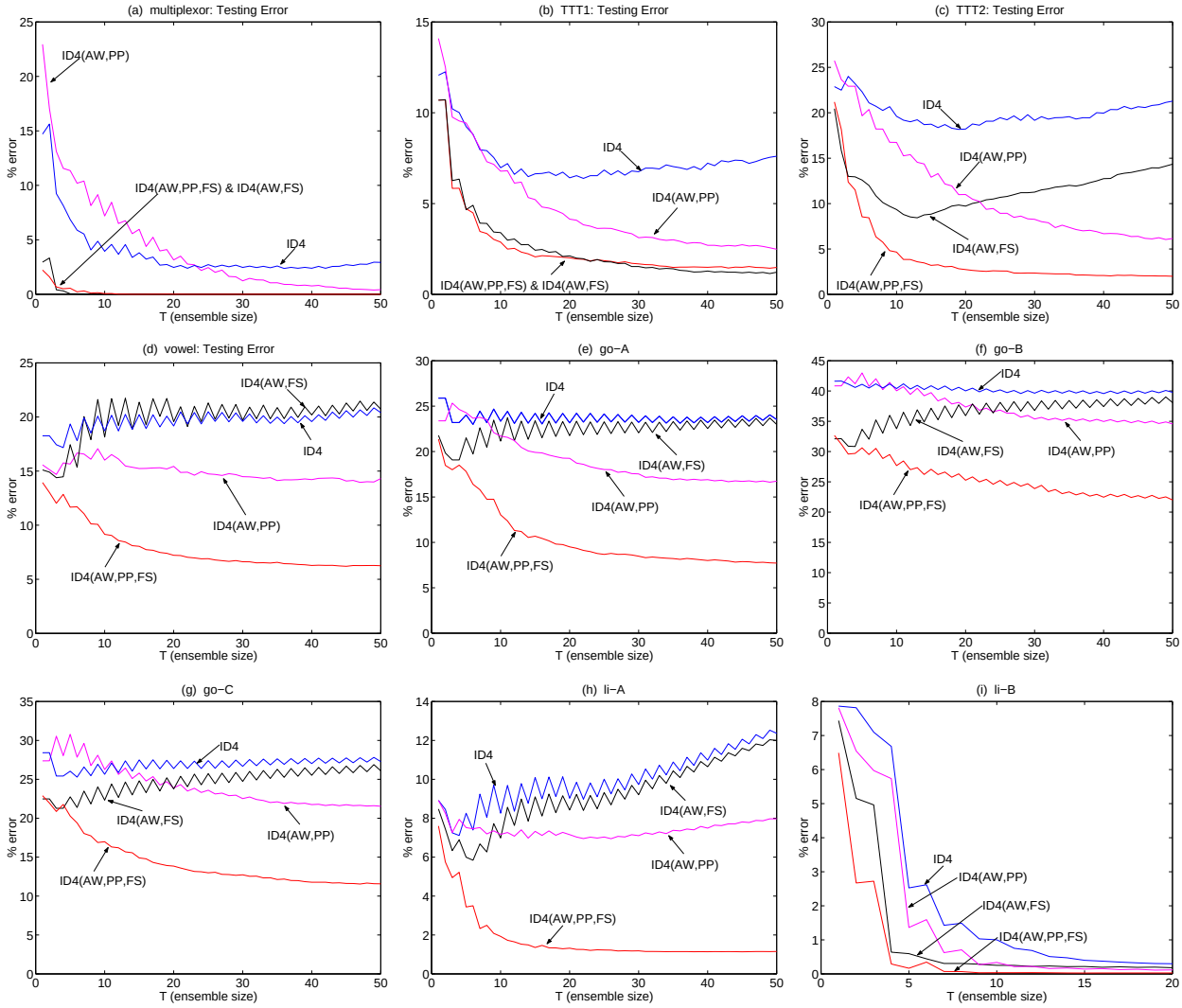

Figure 10: Final testing percent error vs. ensemble size for Arc-x4 using different ID4-variants as base learners. Each graph is labelled by the particular benchmark problem generating the curves. Each curve corresponds to ensembles ranging in size from 1 to 50 trained using Arc-x4 and the indicated ID4-variant. Abbreviations (AW, PP, and FS) are defined in the text.

do not see a stable weighted distribution of training instances until the earlier members converge. We note that online settings often naturally include concept drift, since a dynamic process is typically generating the training data—warmup is also naturally important in online settings, as queries are typically interleaved with training data so that poor warmup performance directly translates to poor performance on those interleaved queries.

Clearly we expect that if individual ensemble members warmup faster, then the ensemble as a whole will also warmup faster (particularly for ordered-weighting schemes such as Arc-x4). The smoothness of ensemble member warmup is also likely to impact the ensemble performance. Note that when subtrees are discarded by ID4 the tree performance may abruptly jump. Such jumps in accuracy directly translate to "jumps" in the weighted distribution of examples seen by subsequent ensemble members (whose instance weights depend on the performance of previous members), and hence may prevent or delay convergence of these subsequent members. Our extensions are specifically designed to reduce the frequency of such subtree pruning. Next, we provide evidence to support these ideas.

To provide such evidence, we show in Figure 11 the unweighted error rate of selected individual ensemble members over time. Note that we are not concerned here with the absolute error of these trees (a tree may have a large absolute error even though the tree contributes well to the ensemble) but rather with the rates and qualitative smoothness of their convergence to stable hypotheses—a fluctuating absolute error rate suggests failure to converge either due to a fluctuating local concept or to a poorly converging base learner, thus the absolute error rate gives some indication of ensemble member convergence.

The results in Figure 11 show, first, that ID4 without all of our extensions gives a percent error that often varies sharply as instances are encountered. More significantly, the results show a strong correlation between erratic (high variance) error rate and poor rate of convergence (relative to other variants). In other words, on different benchmarks, different base learner variants show this erratic error rate and poor convergence (generally together). All variants show slower convergence in later ensemble members—in erratic variants this results in error rates that do not appear to converge, or converge so slowly that the error at the end of the experiment is similar to that at the beginning. The variant with our extensions is generally much smoother than the other variants and never much more erratic, leading to the expectation that it will perform better in ensembles than the other variants. In particular, we note that for TTT1, the variants including FS both perform well in ensembles (see Figure 9) and give individual members that perform smoothly and warmup quickly—in contrast, the variants without FS perform poorly in ensembles and give individual members that perform erratically and converge poorly, particularly as position in the ensemble increases. Similar trends appear for the other two benchmarks, except that in those cases both the PP and FS extensions appear necessary for smooth warmup and consequent convergence in later ensemble members. The vowel benchmark is the least clear, and further study will be needed to fully understand the trends manifested for vowel—nevertheless, the extensions appear to increase the smoothness of the error rate curve and the rate of convergence for ensemble members in

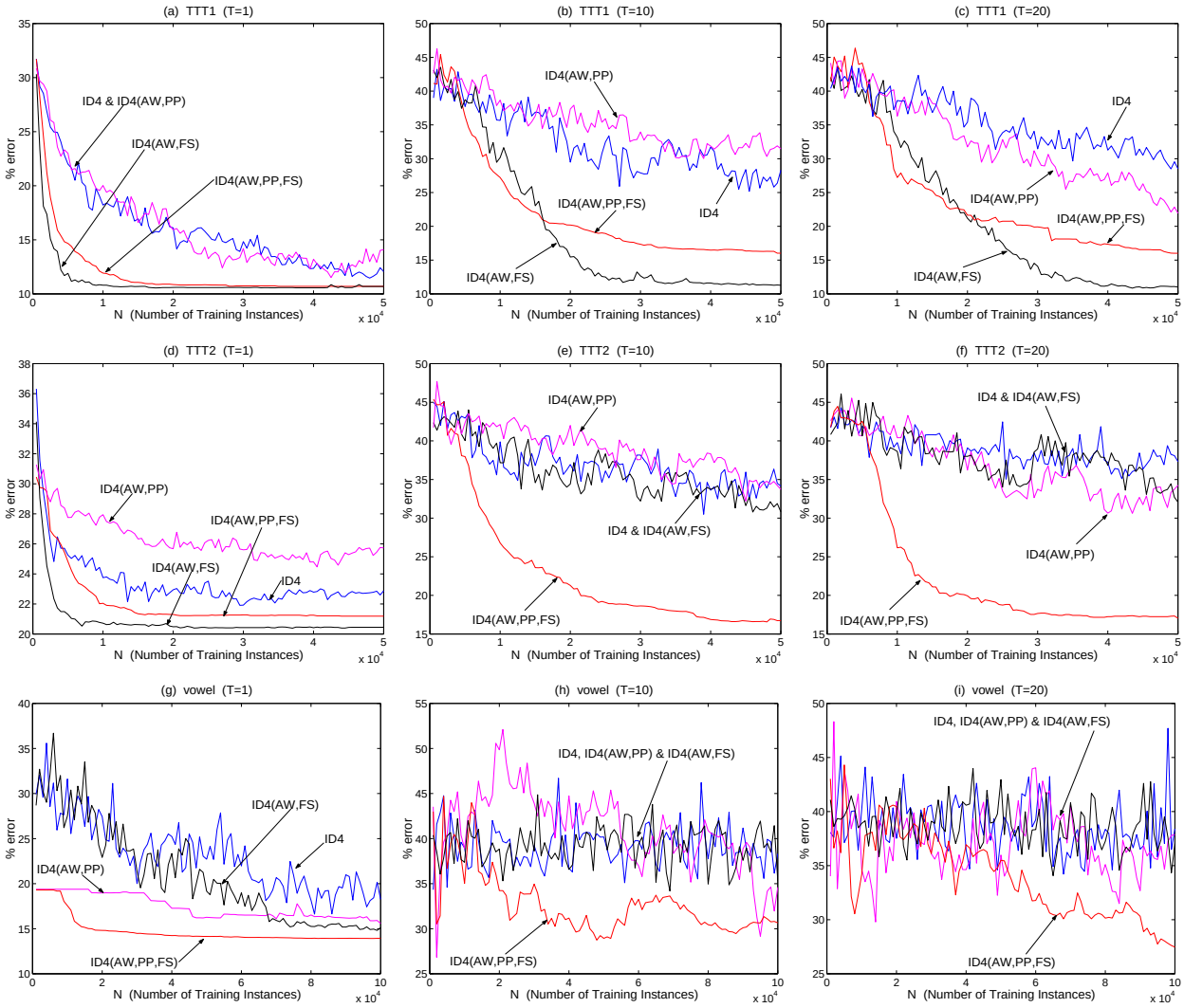

Figure 11: Comparing base learners—% test error for individual ensemble members versus time. Graphs are labelled at the top with benchmark and the position in the ensemble of the member being measured.The curves show the testing percent error for the indicated ensemble tree trained using Arc-x4 with the indicated ID4-variant as the base learner versus the number of training instances. Abbreviations (AW, PP, and FS) are defined in the text. The graphs for multiplexor (not shown) resemble those for TTT1. Note: our interest here is in the qualitative behavior—to best show this we have varied the axis scales across the graphs to optimally display the qualitative behavior (each row uses a common x-axis scale, however).

this benchmark too. The plots (not shown) for multiplexor strongly resemble those for TTT1.

From the above observations we see there is a correlation between the qualitative warmup characteristics of the individual ensemble members and the overall ensemble performance, and that our extensions appear to provide the desired qualitative characteristics (smooth, rapid convergence). It is likely that the inclusion of the feature-switch suppression (FS) extension improves smoothness over time by reducing feature switching and hence subtree pruning. We

also suspect that the inclusion of the post-pruning by subtree monitoring (PP) extension improves the effectiveness of the FS extension in suppressing feature switching as follows. Recall that the FS extension monitors the performance of the subtrees at a node and only changes the split feature if the accuracy of some other feature is better than that of making predictions with the subtrees. When the PP extension is not included it is likely that during early stages of learning the subtree performance will look bad (since the subtrees are being used before they are fully warmed up) resulting in a feature switch in spite of the FS extension. The PP extension can correct this problem by only using the portions of the subtrees that have warmed up enough to give a predictive advantage—dynamically extending these portions as the subtrees warm up. It is unclear why in many cases only the FS extension is needed for single trees (see the discussion of Figure 10), while the PP and FS extensions are both crucial to improving ensemble performance—we hypothesize that the benefits of PP are emphasized in the presence of drifting distributions over instances. Because we are generating our ensembles in parallel, later ensemble members see a drifting weighted instance distribution as the earlier members converge, and so can be said to be facing concept drift even when the domain itself has none.

Evidence favoring parallel-generation multiple-update ensemble creation. Our central purpose for showing the results in this section is to demonstrate the advantages of using our three extensions to the original ID4 online decision-tree algorithm. However, the results in Figure 11 also provide evidence to support our expectation that parallel-generation multiple-update ensembles warm up with fewer training examples than sequential-generation ensembles. There are many ways to design sequential generation approaches, so the approach we discuss here could be in some sense a "straw man"; however, it is perhaps the most obvious method. We consider a sequential method that trains each tree until the error reduction with new training instances is negligible— the graphs in Figure 11 then allow us to estimate how many training instances would be required to train the first ensemble member (in three different benchmarks). For example, in Figure 11 (a) we might conclude that a simple sequential-generation approach would require at least 5000 examples to train the first ensemble member to convergence. If we assume later ensemble members take this same number of examples (each) to train, we can estimate that 100,000 examples will be required to train 20 ensemble members to convergence during sequential generation. In contrast, the parallel-generation approach shown in Figure 11(c) achieves convergence in ensemble member 20 after about 40,000 examples.

8 Future Work

Parallel versus sequential generation. Above we distinguished between parallel-generation and sequential-generation ensemble approaches—we suggested that by taking a parallel approach it may be possible to improve ensemble warmup times, which is a practical concern in many online problems. In this work, however, we evaluated only the parallel-generation approach, and thus did not attempt to compare the approaches empirically. A related issue we

would like to investigate in the future is the use of parallel update methods designed to take into account the fact that the ensemble members are continually learning—in parallel-generation methods the early ensemble members are still changing while the later members are being generated. The methods explored here all use ordered weighting schemes (where the instance weights used for each ensemble member depend only on the previous ensemble members). We would like also to study unordered weighting schemes for parallel update (sequential update methods cannot use such weighting schemes) and to compare the asymptotic and warmup properties to ordered weighting schemes.

Dynamic ensemble size selection. Recall that our results show that the ideal ensemble size varies during warmup (see Figure 4). This fact suggests that we investigate methods for dynamically selecting the best ensemble size to use when making a prediction—such methods are likely to be particularly beneficial during warmup and/or in the presence of concept drift. This approach could provide some of the robustness of sequential generation during warmup while retaining the (possibly) more rapid warmup of parallel generation (much like our "advanced warmup" extension to ID4 in that some ensemble members might be learning without being used for prediction, during warmup).

Varying tree-depth bound across the ensemble. In this work we used the same depth bound for every tree in an ensemble. Is there a better way of distributing the space used by an ensemble? Methods for distributing the depth dynamically may allow better utilization of available memory. To explore this issue, it is possible to collect data on the depths at which predictions are actually being made—this may reveal that parts of the ensemble are simply not using the full depth available (note that the space for the full depth is in use due to our advanced-warmup extension to ID4).

Base learners. Our results showed that the extensions to ID4 are crucial to ensemble performance for most of the problems considered here. This suggests that we try to gain a better understanding of what base-learner properties interact well with our online ensemble methods. In addition, it will be useful to consider online base-learning algorithms other than ID4—for instance, in our branch prediction domain we could consider base learners resembling state-of-the-art traditional-style branch predictors such as the "hybrid" predictor described in (Fern & Givan, 2001). An additional base-learner issue that must be explored is the extension to features that are not binary or even finite domain, such as numerical features (e.g. exploring incremental methods for selecting numeric threshold values).

Other online domains. We would like to find other naturally occurring online learning domains. Some possibilities are: problems like branch prediction in which a system (perhaps a network? a car?) can improve its performance via prediction, data compression (e.g., online ensembles can be used to give probability estimates that are fed into an arithmetic coder), and problems where training data are distributed across a network where the network bandwidth limits the ability to consider all the data at once.

9 Conclusions

In this work we empirically investigated two online ensemble learning algorithms. The algorithms take bagging and boosting-style ensemble approaches and do not require the storage of online training instances. The algorithms have efficient parallel hardware implementations which is crucial to solving online problems with tight time and space constraints such as our target problem of conditional branch outcome prediction. The online boosting-style algorithm (online Arc-x4) is based on previous online boosting research in the "boosting by filtering" framework and on the offline boosting-style algorithm Arc-x4. In this work we generated decision-tree ensembles using these new algorithms with an online decision-tree base learning algorithm that extends ID4. Our extensions to ID4 led to significant performance gains for both single trees and ensemble learners in most of our experiments. Empirical results were given for instances of the conditional branch outcome prediction problem from computer architecture and online variants of several familiar machine-learning benchmarks. The results indicate that online Arc-x4 significantly outperforms our online bagging method in most all of the experiments. In addition, the ensembles produced by online Arc-x4 are shown to achieve significantly higher accuracies than single trees in most of our experiments. Finally, it was shown that ensembles of small trees were often able to outperform single large trees that use the same number of total nodes. Similarly large ensembles of small trees were often able to outperform small ensembles of large trees that use the same number of total nodes. This observations suggests that ensembles may be particularly useful in domains such as branch prediction that have tight space constraints. We hope this research motivates future theoretical and empirical investigating into online ensembles.

10 Acknowledgements

This work was supported by a National Science Foundation Graduate Fellowship and NSF Award No. 9977981-IIS.

11 References

Bauer, E., & Kohavi, R. (1999). An empirical comparison of voting classification algorithms: bagging, boosting and variants. *Machine Learning, 36*, 105–142.

Breiman, L. (1996a). Bagging predictors. *Machine Learning, 24*, 123–140.

Breiman, L. (1996b). *Arcing classifiers* (Technical Report). Dept. of Statistics, Univ. of California, Berkeley, CA.

Burger, D., & Austin, T. (1997). The SimpleScalar tool set, version 2.0 (Technical Report 1342). Computer Science Department, University of Wisconsin-Madison.

Calder, B., Grunwald, D., Lindsay, D., Jones, M., Martin, J., Mozer, M., & Zorn, B. (1997). Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19, no. 1, 188-222.

Chang, P.-Y., Hao, E., & Patt, Y. (1995). Alternative implementations of hybrid branch predictors. *Proceedings of the 28th ACM/IEEE International Symposium on Microarchitecture.*

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1997). Arithmetic circuits. In *Introduction to Algorithms.* Cambridge, Mass. MIT Press.

Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting, and randomization. *Machine Learning*, (to appear).

Domingo, C., and Watanabe, O. (2000). MadaBoost: a modification of AdaBoost. *Proceedings of the Thirteenth Annual Conference on Computational Learning Theory.*

Eden, A., and Mudge, T. (1998). The YAGS branch predictor. *31th Annual IEEE/ACM Symposium on Microarchitecture (*pp. 69-77).

Fern, A., and Givan, R. (2001). State-of-the-art Branch Predictors. (Electronic Appendix).

Fern, A., Givan, R., Falsafi, B., & Vijaykumar, T. N. (2000) *Dynamic feature selection for hardware prediction* (Technical Report TR-ECE 00-12). School of Electrical & Computer Engineering, Purdue University.

Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Information and Computation*, *121(2)*.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Proceedings of the Thirteenth International Conference on Machine Learning.*

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, *55(1),* 119–139.

Grove, A., & Schuurmans, D. (1998). Boosting in the limit: Maximizing the margin of learned ensembles. *Proceedings of the Fifteenth National Conference on Artificial Intelligence.*

Gwennap, L. (1996). Digital 21264 sets new standard. *Microprocessor Report*, Oct. (pp. 9–15).

Heil, T., Smith, Z., & Smith, J. (1999). Improving branch predictors by correlation on data values. *Proceedings of the 32nd Annual International Symposium on Microarchitecutre* (pp. 28–37).

Matheus, C. J., & Rendell, L. A. (1989). Constructive induction on decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 645–650). Detroit, MI: Morgan Kaufmann.

McFarling, S. (1993). Combining branch predictors (*WRL Technical Note TN-36*).

Merz, C. J., & Murphy, P. M. (1996). UCI repository of machine learning databases. http://www.ics.uci.edu/~mlearn/MLRepository.html.

Nair, R. (1995). Dynamic path-based branch correlation. *28th International Symposium on Microarchitecture* (pp. 15–23).

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

Quinlan, J. R. (1988). An empirical comparison of genetic and decision-tree classifiers. *Proceedings of the Fifth International Conference on Machine Learning.* Ann Arbor: Morgan Kaufmann.

Quinlan, J. R. (1996). Bagging, boosting and C4.5. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 725–730). AAAI Press.

Reilly, J. (1995). SPEC describes SPEC95 products and benchmarks. *Standard Performance Evaluation Corporation* August 1995 newsletter, http://www.spec.org.

Schlimmer, J. C., & Fisher, D. (1986). A case study of incremental concept induction. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 496–501). Philadelphia, PA: Morgan Kaufmann.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, *5*, 197–227.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning*, *4*, 161–186.

Utgoff, P. E., Berkman, N. C., & Clouse, J. A. (1997) Decision Tree Induction Based on Efficient Tree Restructuring.

Machine Learning, 29, 5-44.

Appendix A. Results for Online Bagging

In this work we also considered two online ensemble algorithms inspired by bagging—but our empirical results showed these methods to be inferior to online Arc-x4 in our domains, although online bagging still showed improvement over individual base learners. In this appendix we briefly describe bagging and our two online variants as well as show empirical results comparing them to Arc-x4. In application domains where online bagging is competitive with online Arc-x4 one might choose to implement online bagging since it is somewhat less complicated.

Bagging. Bootstrap aggregrating (abbreviated "bagging") is an ensemble learning algorithm introduced by Breiman (1996a). In an offline setting bagging first generates T bootstrap datasets where each set is created by randomly drawing with replacement N instances from the size N training set. Then the base learner is used to produce a hypothesis for each dataset giving us an ensemble with T members. The ensemble prediction is the majority vote of the members.

Online Bagging. The bagging ensemble method has a natural parallel implementation since it does not require any interaction among the T hypotheses. This was the reason we first investigated bagging for creating online ensembles. In this work we considered two online bagging variants—both variants simply ensure that each of the T hypotheses are the result of applying the base learner Learn() to different sequences of weighted training instances.

Our first bagging variant is called BagB ("B" stands for "Bernoulli" because the training instance weights are determined by a coin flip). BagB follows the generic algorithm in Figure 1 with the instance weight function given by

$$\underline{\text{Weight}}\,(H, I, t) = \underline{\text{flip}}\,(P_u), \quad 0 < P_u < 1 \tag{3}$$

where $\underline{\text{flip}}(P_u)$ returns one with probability P_u and zero with probability $1-P_u$. The update probability P_u is a user specified parameter. The parameter P_u dictates a trade-off between the diversity of the training sequences used to update different hypotheses and the fraction of the training data ignored by each hypothesis.

Our second online bagging variant is called BagP ("P" because the training instance weights have a Poisson distribution). The BagP method is designed to select instance weights from approximately the same distribution as the weights produced by offline bagging[22] for large datasets. Given a training data set of size N, offline bagging selects any instance exactly w times with a probability given by the binomial distribution. For moderately large values of N the binomial distribution is closely approximated by the Poisson distribution with a parameter of one. For this reason, the BagP algorithm draws weights from the Poisson distribution, taking $\underline{\text{Weight}}\,(H, I, t)$ to be w with probability $\frac{e^{-1}}{w!}$ for each $w \in \aleph$. Both BagB and BagP used the voting weight update function of online Arc-x4 (i.e., accuracy).[23]

22. The weight distribution in the offline setting is a distribution over the number times a training instance is included in a bootstrap dataset.

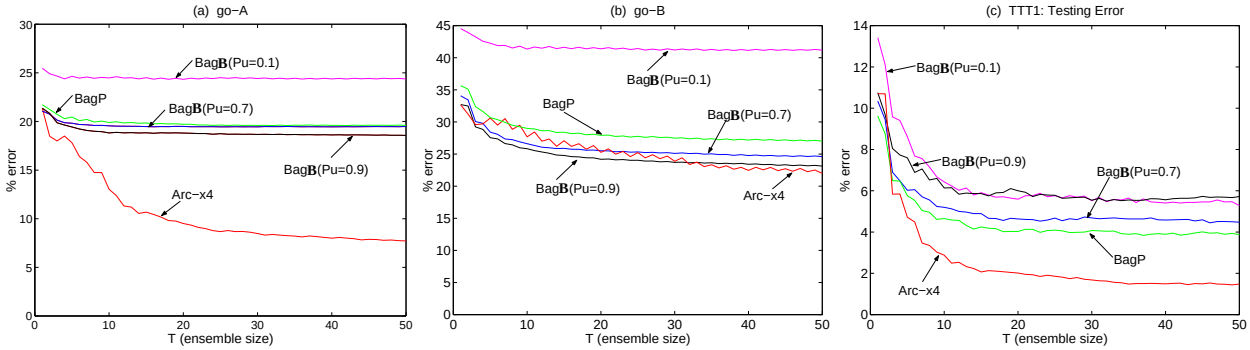

Figure 12: Bagging versus Arc-x4. (*note that the x-axis is not time).* Percent error versus the ensemble size T for three problems (as indicated above each graph). Three curves per graph show the performance of BagB for P_u = 0.1, 0.7, 0.9, one curve shows the performance of BagP, and one curve shows the performance of Arc-x4. The trees in all of the ensembles have a depth bound of twelve. Graph (a) strongly resembles the same plot for our other branch prediction problems as well as the ML benchmarks TT2 and vowel. For the multiplexor benchmark bagging was able to reduce the testing error to zero percent as was the case for Arc-x4.

Empirical Results. Our BagB, BagP, and Arc-x4 methods were used to train size T ensembles of online decision trees bounded to depth twelve. Figures 12a-12c compare the performance of bagging and Arc-x4 on three online learning problems—in each figure we plot percent error versus ensemble size for each method. The experimental protocols used in each benchmark are the same as those described for evaluating Arc-x4 in Section 5, except that the plots for branch prediction are now averaged over ten runs due to the random choices made in the bagging algorithm—results for the other branch prediction problems and the ML benchmarks TTT2 and vowel exhibit the same trends as Figure 12a, and results for the multiplexor ML benchmark resemble those for Arc-x4 depth ten in Figure 3(a) (except that BagB(P_u=0.1) performed poorly as expected, no better than an accuracy of 6.3%).

Neither BagP nor BagB seems to dominate the other. We show in Figure 12b the most significant improvement over Arc-x4 achieved by bagging in any of our bagging experiments. In the problems not shown (except for multiplexor) the bagging methods led to relatively little improvement in accuracy (compared to online Arc-x4) as the ensemble size grew (Figure 12a is a typical example). Generally the bagging curves flatten out as T increases much sooner than the Arc-x4 curves do, indicating that bagging is not as successfully exploiting ensemble size to reduce error. Overall, Arc-x4 significantly outperforms bagging for small ensembles (usually) and large ensembles (all experiments except multiplexor where bagging and Arc-x4 tie at zero error). We note however that online bagging is still generating significant error reductions compared to single base learners for some of the domains.

23. We have also implemented online bagging using straight majority vote and the empirical results are not substantially different.