# Forest Packing: Fast Parallel, Decision Forests*

James Browne† Disa Mhembere† Tyler M. Tomita† Joshua T. Vogelstein†
Randal Burns†

**Abstract**

Decision Forests are popular machine learning techniques that assist scientists to extract knowledge from massive data sets. This class of tool remains popular because of their interpretability and ease of use, unlike other modern machine learning methods, such as kernel machines and deep learning. Decision forests also scale well for use with large data because training and run time operations are trivially parallelizable allowing for high inference throughputs. A negative aspect of these forests, and an untenable property for many real time applications, is their high inference latency caused by the combination of large model sizes with random memory access patterns. We present memory packing techniques and a novel tree traversal method to overcome this deficiency. The result of our system is a grouping of trees into a hierarchical structure. At low levels, we pack the nodes of multiple trees into contiguous memory blocks so that each memory access fetches data for multiple trees. At higher levels, we use leaf cardinality to identify the most popular paths through a tree and collocate those paths in contiguous cache lines. We extend this layout with a re-ordering of the tree traversal algorithm to take advantage of the increased memory throughput provided by out-of-order execution and cache-line prefetching. Together, these optimizations increase the performance and parallel scalability of classification in ensembles by a factor of ten over an optimized C++ implementation and a popular R-language implementation.

## 1 Background and Motivation.

Decision forests are a broad class of machine learning algorithms that include many flavors of gradient boosted forests [6] and random forests [4]. Decision forests are *interpretable*—decision boundaries and features used for inference can be observed and visualized. This is important for scientists and analysts to explain results and make comprehensible conclusions. These systems are extremely flexible, tend not to overfit data, and can be used for classification, regression, density estimation, and manifold learning [9]. Recent advances have defined the first scalable implementation [6] integrated with High Performance Computing (HPC) schedulers and message-passing distribution [5]. Despite the extreme utility, decision forests have been relegated to applications where model interpretation is important or to competitions with low error rate objectives [6, 12].

For many real-time applications, such as computer vision and spam detection, decision forests are unusable due to high inference latency caused by the combination of large model size with random memory access patterns. As an ensemble classifier decision forests are composed of many weak classifiers, each of which may have a size in order with the training set, leading to a large overall memory footprint. When the model size is larger than a systems's cache, typically KBytes for fast cache and low MBytes for slow cache, the inference operation relies on the order of magnitude slower main memory. The paths through a decision forest are purposefully uncorrelated, which leads to random accesses throughout the model for each inference task, making common acceleration tools—such as GPUs—unusable for general forest inference.

Decision forest research typically focuses on model training systems without mention of run time operation. Application of these powerful tools is hindered by this oversight, which has lead to increased research into forest inference acceleration and development of several third-party post training optimizations. Current solutions either place structure limiting requirements on the forests, e.g. maximum node depth, or focus on increasing throughput at the cost of increased inference latency, e.g. batching. Forest packing extends previous research on this topic by introducing several tree storage memory optimizations and a reordering of the tree traversal process to take advantage of modern CPU enhancements.

## 2 Methods and Technical Solutions.

Attempts to increase the speed of decision forest inference falls into two categories: memory access opti-

*Supported by NSF grants IOS-1707298 and ACI-1649880; DARPA grants HR001117S0016-L2M-FP-054 and N66001-15-C-4041; and NIH/NINDS grants 1R01NS092474-01 and 1U01NS090449-01.

†Johns Hopkins University, Baltimore {james.browne, ttomita, disa, jovo, randal}@jhu.edu.

mizations and inference algorithm modifications. Our proposed improvement to forest based inference, Forest Packing, takes advantage of both improvement types. We use inference latency, specifically classification latency, to indicate model performance (training time is a popular complementary topic that we do not consider here). Other decision forest variants such as regression or distance learning forests are not specifically discussed here, but the techniques we describe should extend to these tools. The input to forest packing is a trained forest, $F$, that consists of decision trees, $t$. Each internal node of the trees describes a splitting condition on the data and has two children. Each leaf node contains a single class label; a criteria typical of random forests [4, 11], but not true for all variants. One of the novel optimizations that we present relies on the single class assumption.

Forest Packing, the system presented here, reorganizes the trees in a forest to minimize cache misses, allow for use of modern CPU capabilities, and improve parallel scalability. The system outputs $\lceil T/B \rceil$ bins, where $T$ is the number of trees in the forest and $B$ is the bin size—a user provided parameter defining the number of trees in a bin. A bin is a grouping of at most $B$ trees into a contiguous memory structure. Most bins will contain $B$ trees while one bin may contain between 1 and $B$-1 trees. Within each bin, we interleave low-level nodes of multiple trees to realize memory parallelism in each cache line access. We decrease cache misses by using split cardinality information to store popular paths contiguously in memory. During inference, each bin is assigned to an OpenMP thread (for shared memory). Forest packing includes a runtime system that prefetches data and evaluates tree nodes out-of-order as data is ready, leveraging the memory layout to maximize performance.

**2.1 Memory Layout Optimization.** We describe our memory layout as a progression of improvements over the breadth-first layout, BF, typically used in random forests, with each improvement reducing cache misses or encoding parallelism. Fig. 1 describes this evolution with each panel displaying a tree in which the nodes are numbered breadth-first and the resulting layout in memory is shown as an array at the bottom of the panel. Breadth-first layouts are typical in decision forests because they provide the sequential traversal through all nodes used by batched inference processing. The depth-first (DF) layout is preferred for single observation inference because it allows for the possible reuse of a memory load. We will use aspects of both breadth- and depth-first layouts in our ultimate design.

Our first optimization reduces the duplication of

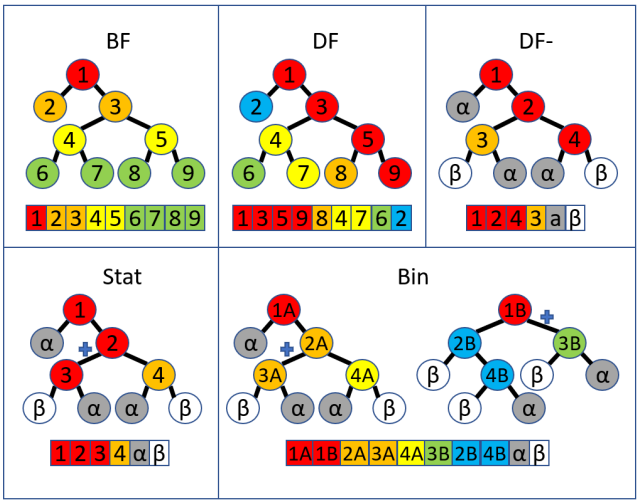



Figure 1: The representation of trees as arrays of nodes in memory for breadth-first (BF), depth-first (DF), depth-first with class nodes (DF-), statistically ordered depth-first with leaf nodes replaced by class nodes (Stat), and root nodes interleaved among multiple trees (Bin). Colors denote order of processing, where like colors are placed contiguously in memory.

information contained in leaf nodes thereby reducing the number of nodes in the forest almost in half. In many classification forest variants, each leaf node provides both a signal that the tree traversal is complete and a class label determined during training by the plurality observation class in a leaf node. Rather than pointing parent nodes to their own unique children, parent nodes instead point to communal leaf nodes which provide the functionality of a typical leaf node without the duplication. We call this encoding DF-. We are unaware of other literature recommending this encoding, which reduces the size of a tree from $n$ nodes to $n/2+C$ nodes, where C is the number of classes present in a dataset.

In most datasets, the number of distinct classes are many orders of magnitude smaller than the number of nodes in the tree and so we expect the DF- trees to be nearly half the size of DF trees. Removing these nodes has the dual benefit of allowing more useful nodes to be loaded by each memory fetch (a fact we exploit in the Stat layout) and also greatly reduces cache pollution. The class nodes that replace leaf nodes are placed at the end of each tree's block of contiguous memory. The DF- panel of Fig. 1 shows the resulting layout when building a decision tree for a two-class $(\alpha, \beta)$ problem. DF- indicates depth-first with leaf nodes removed.

The next improvement encodes paths through trees sequentially in memory based on the number of observations used to create the nodes during training. This

statistical redefinition of the BF- layout is called Stat. We use the leaf cardinalities collected during training to determine the likelihood that any given data point routes to a specific leaf. If cardinality information is unavailable, these statistics can be inferred after training using a suitably large set of observations. We then enumerate depth first paths based on their probability of access. The Stat panel of Fig. 1 illustrates this process with "+" indicating a more likely path, leading to the decision to enumerate the path to node 3 prior to node 4. The statistical ordering applies to nodes from the root to the leaf parents, because class nodes that replace leaf nodes are at the end of the block of memory and so are not considered for statistical ordering.

In more detail, Stat considers each parent node and its two children. The child that is accessed most often is placed adjacent to the parent node in memory while the less accessed child is placed later in the block of memory. If a parent node has a leaf node child and an internal node child, the internal child node is always placed adjacent to the parent while the leaf node is shared among all leaf nodes of the same class at the end of the memory block. Similar optimizations have been recommended in one form or another by other researchers [13].

Our final memory optimization, Bin, interleaves the nodes of multiple trees into a single block of memory, called a bin, to both reduce memory latency and to allow for the encoding of parallel memory accesses. This layout takes advantage of the fact that a parent node is accessed about twice as often as each of it's children and so nodes at lower levels of a tree are accessed far more often then nodes at higher levels. This is similar to the hot (often used nodes) and cold (rarely used nodes) memory model recommended by Chilimbi et al. [7]. By interleaving the lower level nodes from multiple trees in a bin, we are increasing the density of "likely to be used" nodes which allows a single cache line fetch to be more useful while also reducing cache pollution. The Bin panel of Fig. 1 shows the root (level 0) nodes interleaved in the layout and all higher level nodes are stored one tree at a time using the Stat method. A similar recommendation was proposed by Ren et al. where trees are interleaved by level for the entirety of the tree [14]. In our testing, interleaving trees past a certain depth results in an increase of inference latency (see Fig. 2).

There are two parameters used when interleaving nodes. (1) The bin size determines the number of trees interleaved in a bin. Larger bin sizes encode more parallelism for each memory fetch at low levels in the tree. Each bin is an independent array of trees that can be distributed for parallel evaluation across threads or

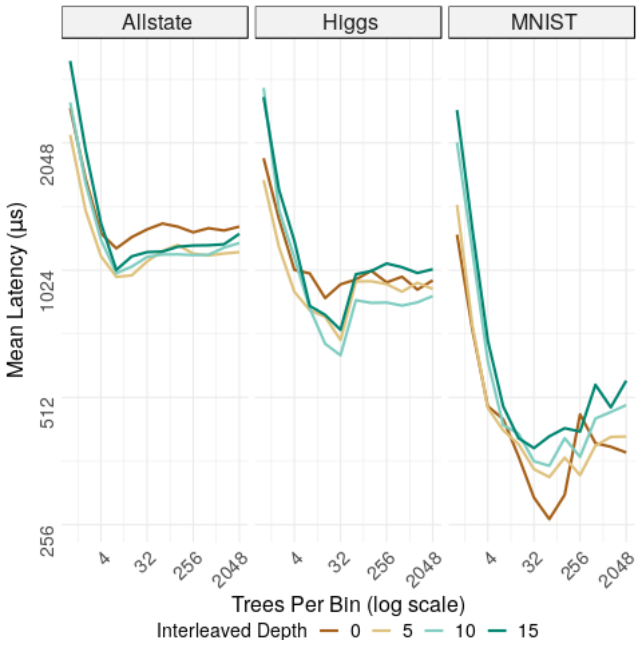



Figure 2: Prediction time as a function of the number of trees in each bin and their interleaved depths. Ideal bin parameters are forest dependent. Interleaving trees beyond a certain depth becomes detrimental to performance.

a cluster, so at least one bin is required per thread in order to occupy parallel resources. In other words, the number of bins will ideally be a multiple of the number of threads used to perform inference. (2) Interleaved depth determines how many lower levels of the trees in a bin are interleaved; the remaining levels of each tree are stored according to the Stat encoding. Interleaving too many levels results in breadth-first like behavior resulting in degraded performance.

The Bin layout combines binned interleaving for the top levels of a tree and statistical depth-first layout, Stat, for the bottom levels in a tree. A more in depth example of this layout can be found in Fig. 3, which demonstrates the layout for a forest of 4 trees and 3 classes with varying bin sizes and depths interleaved. A design experiment helps choose the bin size that makes sense in light of the properties of the processor memory hierarchy and forest characteristics. Fig. 2 explores the effects bin size and interleaved depth has on a trained forest. The figure displays average prediction time (lower is better) for an observation given varying values of bin size. The characteristics of the datasets and their resulting forests can be seen in Table 1.

Fig. 4 shows performance comparisons between the memory optimizations described above. The first three

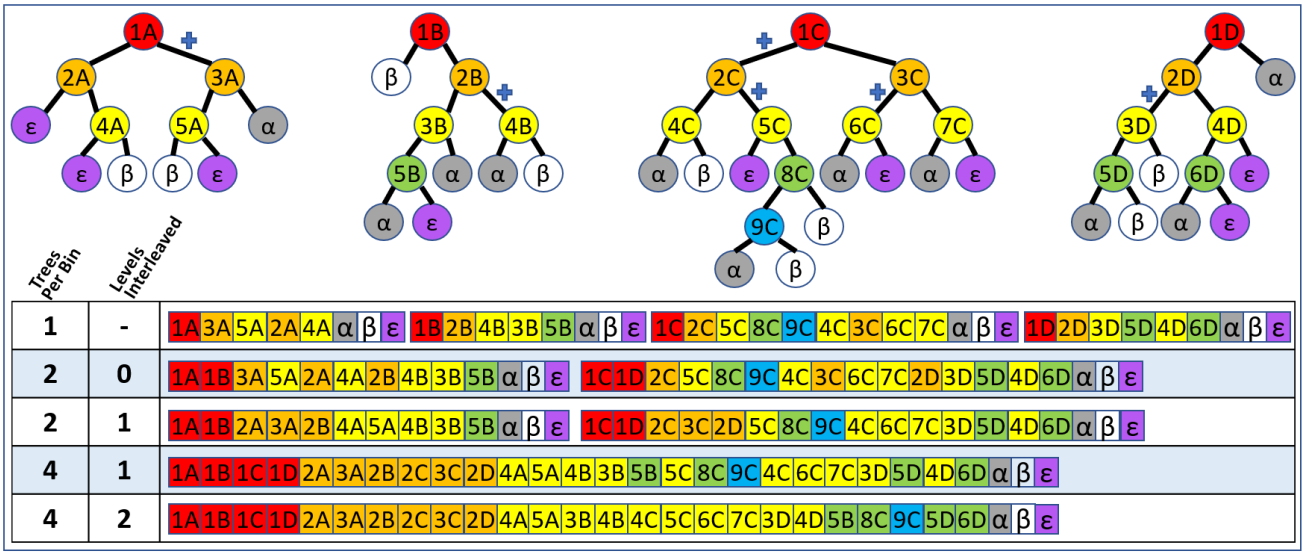



Figure 3: Forest packing with the Bin memory layout given four trees and varying setting. We show different parameterizations of number of trees per bin and depth of interleaving. Colors of internal nodes denote node level in the tree. Levels Interleaved denotes the highest level of each tree interleaved.

Table 1: Dataset and Resulting Forest Information

| | **Allstate** | **Higgs** | **MNIST** |
|---|---|---|---|
| Observations | 500000 | 250000 | 60000 |
| Test Observations | 50000 | 25000 | 10000 |
| Features in Dataset | 33 | 30 | 784 |
| Trees in Forest | 2048 | 2048 | 2048 |
| Internal Nodes (avg) | 90020 | 23964 | 5008 |
| Avg Leaf Depth | 25 | 21 | 17 |
| Deepest Leaf Depth | 74 | 65 | 50 |

performance improvements (DF, DF-, and Stat) are purely reorganizations of nodes within memory. In addition to reorganizing nodes, Bin groups trees into bins and interleaves the nodes of trees within a bin.

**2.2 Tree Traversal Modification.** The inference algorithm includes the traversal of trees from root to leaf using a path determined by a simple comparison of data stored in each node to the test observation, Algorithm 1. The most common modification of this traversal processes multiple test observations through a tree simultaneously in batches. This optimization results in much higher throughput of classification inference [8, 15]. Although groups of observations are processed faster, the latency of each inference task is increased. Other methods have been used to accelerate forest based inference using GPUs, SIMD operations, traversal unrolling, and branch prediction [3, 8, 14]. Many of these optimizations place restrictions on forest structure such as a maximum depth or require a full tree [10].

The binned forest memory optimization, Bin, described in 2.1 allows us to efficiently modify the processing order of trees (Algorithm 2) to simultaneously traverse multiple trees in a manner that takes advantage of modern CPU enhancements while also allowing us to use intra-observation threaded parallelism. Forest Packing, our acceleration technique, is the use of the culmination of memory improvements, Bin, with this efficient tree traversal method described below. For brevity, we will refer to Forest Packing, the combination of memory improvements and efficient tree traversal, as Bin+.

Our binning strategy adds an additional layer of hierarchy within the forest (i.e. the bins) which provides for multiple levels of parallelism. The first level of parallelism, inter-bin parallelism, takes advantage of the embarrassingly parallel nature of trees, allowing each bin to be processed by a thread independent of work done in other bins. This form of parallelism was always available within forests using individual trees rather than bins.

Intra-bin parallelism realizes performance improvements through fine-grained, low-level prefetching and scheduling provided inherently in modern CPUs. Bin+

takes advantage of this capability by evaluating all trees in a bin using a round-robin order, thereby giving a single thread additional non-blocking avenues of work. This has the dual benefit of using more of each thread's processing capability and gives us an opportunity to explicitly prefetch the upcoming node prior to it being required (line 17, Algorithm 2).

**Algorithm 1** Single Tree Traversal

```
1:  procedure TreePrediction(t, o[])
2:      ▷ Tree, t, an array of nodes
3:      cn ← 0                                  ▷ start at tree root
4:      while t[cn] is internal node do
5:          ▷ get split feature of node cn
6:          sf ← t[cn].splitFeature
7:          ▷ compare observation's feature to split value
8:          if o[sf] < t[cn].splitValue then
9:              cn ← t[cn].leftChild
10:         else
11:             cn ← t[cn].rightChild
12:         end if
13:     end while
14:     return t[cn].classNumber
15: end procedure
```

We use simple policies to encode execution overlap because more complex policies incur scheduling overheads that exceed gains. Overlap happens among tens of memory accesses and hundreds to thousands of instructions. Trying to control this fine-grained process in software is not practical. Instead, we assign a single thread to each interleaved bin which evaluates the bin's trees in round-robin order. For each node that we evaluate, we issue a prefetch instruction for the memory address of the resulting child node, with the idea that useful work from other trees can be performed while the resulting child node is being loaded into cache. We proceed navigating through the levels of the tree in the same order that we processed the root nodes. This round-robin scheduling submits independent instructions for each tree. In practice, the processor executes these independent instructions out-of-order as data becomes available.

## 3 Empirical Evaluation.

We evaluate Forest Packing in order to quantify the benefits from memory layout and scheduling optimizations. We start with a breakdown of the contributions of each optimization, applying them incrementally. An overview experiment using CPU performance counters demonstrates the improvement of Forest Packing versus other layouts. We explore the scalability of each of the optimizations across a shared-memory multithread system to minimize inference latency. We finish our evaluation by comparing the inference latency of Forest Packing to two commonly used forest systems.

**3.1 Experimental Setup.** We implement all of the optimizations described in Section 2 and apply them successively. We start with BF as our baseline implementation because it performs similarly to a popular decision forest implementation, XGBoost. Thus, comparative evaluations are against our re-implementation, BF. We focus on standard decision forests that create deep trees with a single class per leaf node. We infer the class of each test observation sequentially, only starting the inference of an observation at the completion of the previous observation in order to measure test latency.

All experiments were run on a 64-bit Ubuntu 16.04 platform with four Intel Xeon E7-4860V2 2.6 GHz processors, with 1TB RAM. gcc 5.4.1 compiled the project using -fopenmp, -O3, and -ffast-math compiler flags.

**Algorithm 2** Binned Tree Traversal

```
1:  procedure BinPrediction(b, o[])
2:      ▷ Bin, b, composed of intertwined trees
3:      ▷ binSize ← number of trees in a bin
4:      ▷ int np[binSize]                       ▷ node pointers
5:      ▷ int predictions[numClasses] ← 0
6:      np[] ← 0 : binSize − 1                  ▷ set np to root nodes
7:      do
8:          notInLeaf ← 0
9:          for all p in np do
10:             if b[p] is an internal node then
11:                 sf ← b[p].splitFeature
12:                 if o[sf] < b[p].splitValue then
13:                     p ← b[p].leftChild
14:                 else
15:                     p ← b[p].rightChild
16:                 end if
17:                 prefetch(b[p])
18:                 ++notInLeaf
19:             end if
20:         end for
21:     while notInLeaf ≠ 0
22:     for all p in np do
23:         ++predictions[b[p].classNumber]
24:     end for
25:     return predictions[]
26: end procedure
```

We perform all experiments against three common machine-learning datasets that are widely used in competitions and benchmarks. Training size is reported

here as this determines the size and structure of the resulting forests. Categorical data is one-hot encoded so that all datasets contain numeric data represented as doubles in our data structures. MNIST is a popular numeric dataset of 60,000 handwritten digit images. The Higgs dataset was used in a popular Kaggle competition and contains 250,000 observations of numeric features [2]. We use a subset of the Kaggle competition Allstate dataset which consists of categorical and numeric features [1]. Categorical features are converted to numeric features by the forest growing system (RerF or XGBoost). The principles shown are independent of the system used to grow the forest, but for benchmarking purposes RerF is used to create our forests which are then post-processed into packed forests. Further information about these datasets and resulting trained forests can be found in Table 1.

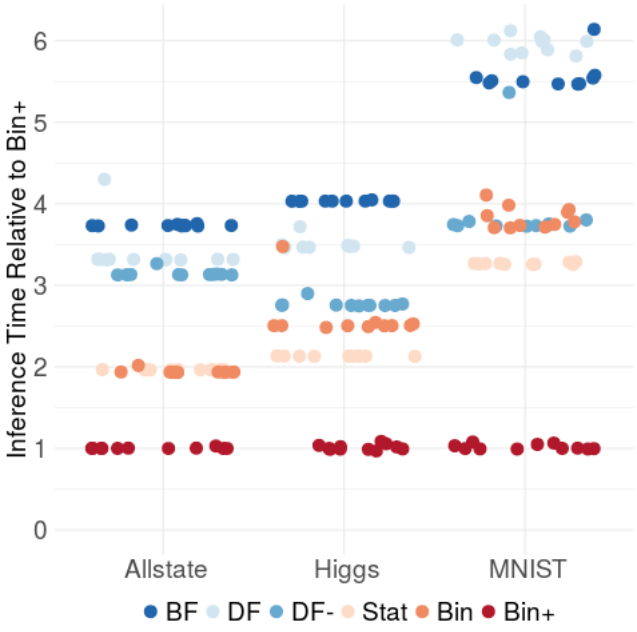



Figure 4: Our optimized breadth first (BF) layout serves as a baseline for all performance gains. Bin shows performance gains based on all memory layout optimizations. Bin+ is the combination of Bin with efficient tree traversal methods. This experiment uses a single thread.

Each of the datasets contain both training and testing observations. The training observations were used to train the forests and provide node traversal statistics. The testing observations were only used for measuring prediction speeds and were not used to determine traversal statistics.

**3.2 Layout Contributions.** We now turn to a detailed study of the effects of layout optimizations on runtime performance. Tab. 2 shows the incremental improvement of runtime statistics for the optimizations. The high ratio of wasted cycles of each of the encodings shows that runtimes are dominated by CPU stalls which are typically caused by slow memory accesses or high numbers of branch mispredictions. Because the runtime is nearly cut in half between the BF and Bin with little change in the numbers of instructions executed, branches, and branch mispredictions, we conclude that stalls occur mainly because of slow memory accesses. This is corroborated by greater than one Cycles Per Instruction (CPI), which indicates an application is memory bound.

We attribute the improvement of run times for DF, DF-, and Stat to reductions in the level at which cache misses are resolved. Average stall duration is reduced by finding required memory lower in the cache hierarchy, which is shown in "Avg Stall (cycles)" column of Tab. 2. The Bin optimization sees a slight increase in stall duration which we attribute to interleaving trees. This happens because "likely to be used" nodes, those near the roots, of several trees are stored together. This makes the access of a needed node in one tree likely to also load a soon to be useful node from another tree. Whereas other encodings quickly recover from stalls due to cache misses for often used nodes, the Bin encoding avoids these stalls altogether. In other words, the Bin optimization avoids quickly resolved stalls which increases the overall average stall duration.

**3.3 Algorithm Contributions.** Fig. 4 shows the inference time of the encodings relative to Bin+, the combination of memory and algorithm improvements. Bin+, improves single thread performance by a factor of five on the MNIST forest and by a factor of three on the Allstate forest. The Bin and Bin+ layouts use a bin size of 32 trees per bin and interleaved level of 3.

Forest Packing, Bin+ reduces the affects of being memory bound by re-ordering tree traversals (Algorithm 2). Our improvement paradoxically doubles the number of CPU instructions and branches required during runtime—leading to an increase in branch mispredictions. Despite these negative changes, Forest Packing halves the ratio of wasted cycles and stall duration, leading to the demonstrated performance improvements.

The Bin+ optimization is less likely to stall because of the overlapped, out-of-order, execution allowed by our traversal modification. When using Algorithm 1 a thread recovers from a stall when the sole execution buffer slot in use becomes ready to execute its instructions. The updated traversal method allows the thread

Table 2: Execution Statistics for 25,000 Higgs Observations, Single Thread

| | CPI | Instructions | Total Cycles | Useful Cycles | Wasted Cycles | Avg Stall (cycles) | Branches | Branch Misses |
|---|---|---|---|---|---|---|---|---|
| **BF** | 7.78 | 2.53e10 | 1.97e11 | 3.84e10 | 80.48% | 12.67 | 4.24e9 | 6.33e8 |
| **DF** | 6.30 | 2.53e10 | 1.59e11 | 3.59e10 | 77.47% | 11.74 | 4.24e9 | 6.33e8 |
| **DF-** | 5.70 | 2.53e10 | 1.44e11 | 3.59e10 | 75.05% | 10.58 | 4.24e9 | 6.32e8 |
| **Stat** | 5.09 | 2.53e10 | 1.28e11 | 3.41e10 | 73.38% | 10.05 | 4.24e9 | 6.32e8 |
| **Bin** | 4.33 | 2.68e10 | 1.17e11 | 3.30e10 | 73.30% | 10.43 | 4.66e9 | 6.35e8 |
| **Bin+** | 1.28 | 5.13e10 | 6.69e10 | 4.18e10 | 37.45% | 4.64 | 7.91e9 | 7.76e8 |

to use all execution buffers. The loading of any of the multiple execution buffers in use allows the thread to continue execution, thereby reducing both the liklihood a stall occurs and the duration of a stall when it does occur.

The benefits of Bin+ can only be realized with many concurrent tree traversals. As the inference process progresses through a bin, the trees arrive at their leaf nodes at different depths. The overlap of memory requests is reduced for each tree in a bin that reaches it's leaf node. This potential for skew is shown in Table 1, where the average leaf depth and deepest leaf depth of each of the test forests is shown, the latter being triple the former. As an example, when all but one tree finishes processing, the remaining tree will no longer benefit from out-of-order execution because there are no other trees with which to overlap memory accesses. When this occurs the remainder of the tree traversal reverts back to the Stat traversal pattern.

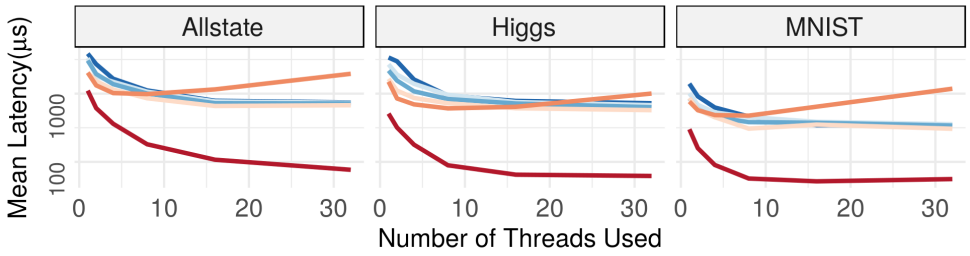
**3.4 Parallel Evaluation to Reduce Latency.** We study the effect on latency of parallelizing classification in a shared memory multithreaded system. This is a strong scaling experiment that runs a single classification on one to 32 threads. For the Bin and Bin+ encodings, the number of trees per bin is set to 16 and nodes are interleaved to depth of 3. To focus on reducing latency, we only begin processing an observation at the completion of the previous observation.

The bin data structure and improved algorithm allows Bin+ to scale more efficiently than the other encodings. Binning allows us to statically pin an execution thread to a subset of bins which allows a thread to process a subset of trees using out-of-order execution, maintain usefulness of all levels of cache, and reduce skew. Without the optimizations provided by binning, we can pin trees to threads and experience degraded parallelism due to skewed execution, or, we can allow threads to dynamically process trees and experience degraded parallelism due to poor use of cache. In neither case can we take advantage of out-of-order processing.

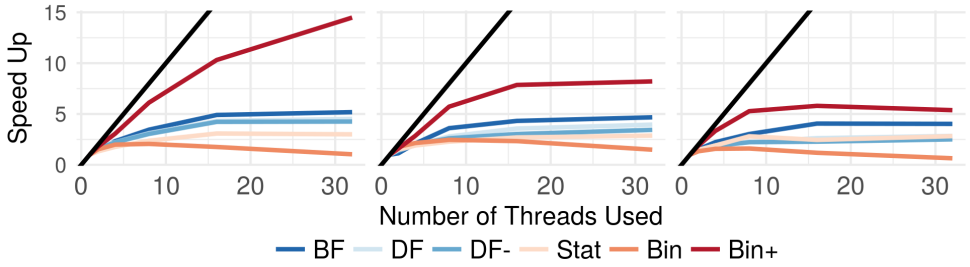
Shared-memory parallelism reduces latency effectively for each of the encodings (except Bin), but, despite being embarrassingly parallel, scaling is sublinear, Fig. 5(a) and (b). We expected this suboptimal scaling due to both increased contention in the memory subsystem and the skewed nature of observation traversal depths (Table 1). Multithread Bin performs poorly because threads receive extraneous nodes with each memory request because of the interleaved trees. This adds cache pollution, reduces cache locality, and requires multiple requests for cache lines with interleaved nodes.

To realize the full benefit of both threaded parallelism and the intra-thread parallelism provided by out-of-order execution, it is necessary that the number of trees in the forest be at least 16 to 32 times the number of threads used. This limitation can be seen in Fig. 6 where intra-thread parallelism is shown to work well with at least 16 to 32 trees per bin. In addition, the number of bins in a forest should be a multiple of the intended number of threads used for parallelism in order to ensure equal workloads are provided to each thread. In general, larger bin sizes perform better during multithread operation, but increasing bin sizes reduces the number of bins available for threaded parallelism.

**3.5 Comparison to Popular Tools.** We found no forests implementations that specifically target the reduction of inference latency, so instead comparisons will be made to two popular decision tree systems: XGBoost and RerF. Because both of these systems typically process observations in batches, we chose to use throughput as a measurement of comparison despite our system being optimized to reduce latency. We vary the size of forests in both the number of trees and size of trees (using max depth) to show the efficacy of Forest Packing for




(a) Per observations prediction time. The Y axis is log10 scale.




(b) Speed-up with additional threads.

Figure 5: Multithread characteristics. Bin+ performance with one thread is superior to the other encodings and scales better with additional resources.

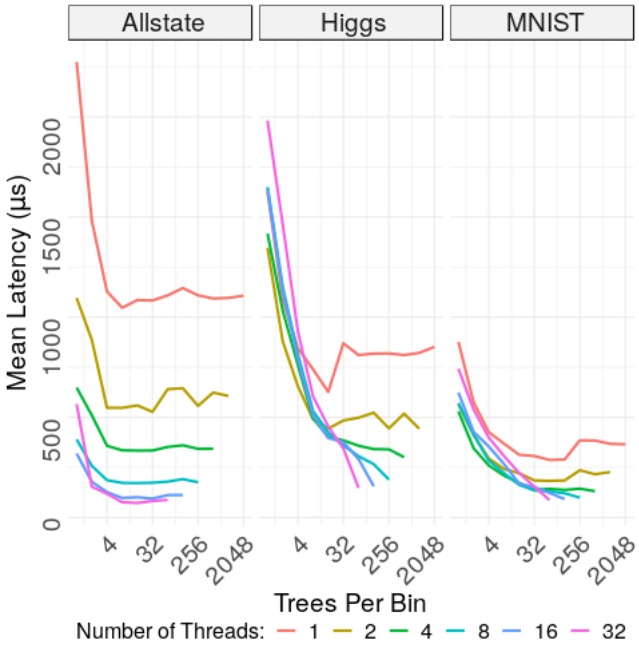



Figure 6: Effects of bin size on multithread performance for a 2048 tree forest. Increasing bin size allows for more intra-thread parallelism but limits thread level parallelism.

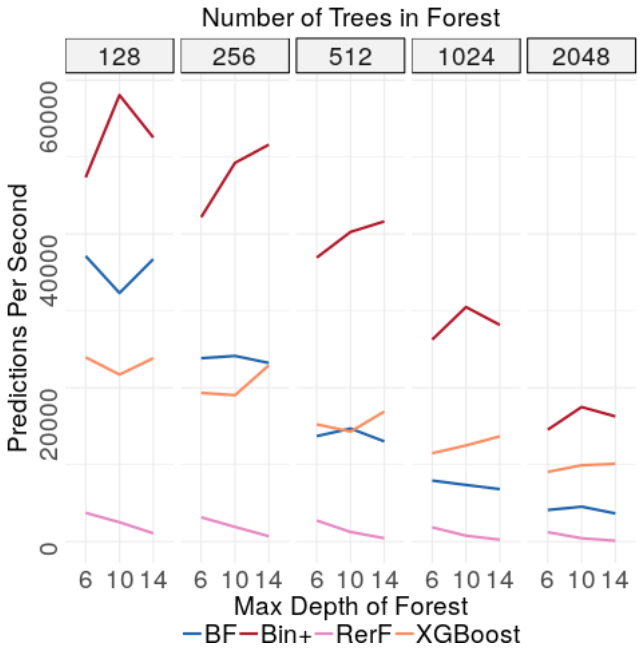



Figure 7: Inference throughput when varying number of trees in forest (defined above the chart) and tree depth. Forests were created using the MNIST dataset. Experiment uses multiple threads and batch size of 5000 where applicable.

a variety of forest sizes, Fig. 7. We set the batch size to 5000 for XGBoost and RerF. All systems except Forest Packing use all 96 threads of our machine—XGBoost uses forest characteristics to dynamically choose how many of the 96 threads to use. We limit Forest Packing to use only 16 threads in order to maximize intra-thread parallelism, a limitation described in Sec. 3.4. The use of 16 threads requires the use of at least 16 bins, which, for 2048 trees, limits the bin size to 128 trees. For a forest of 128 trees, the use of 16 threads limits the bin size to 8 trees. Despite the limitation on thread use, Forest Packing continues to outperform all other systems.

## 4 Significance and Impact.

The two major contributions provided by Forest Packing are the reduced memory footprint of a forest model and the novel tree traversal process made possible by tree binning. The removal of redundant leaf nodes halves decision forest memory requirements, which is a critical improvement for machine learning applications on memory limited devices such as mobile devices or stand-alone sensors. The tree traversal modification provides a faster, more scalable, system with multi-thread performance more than an order of magnitude faster than the naive solution. This reduced latency can make decision forests a more competitive option for real-time vision, recommender, and anomaly detection systems.

## References


[1] *"Allstate Claim Prediction Challenge"*. https://www.kaggle.com/c/ClaimPredictionChallenge. Accessed February 13, 2018.

[2] *"Higgs Boson Machine Learning Challenge"*. https://www.kaggle.com/c/higgs-boson. Accessed February 13, 2018.

[3] N. Asadi, J. Lin, and A. P. de Vries, *"Runtime Optimizations for Tree-Based Machine Learning Models"*, IEEE Transactions on Knowledge and Data Engineering, 26 (2014), pp. 2281–2292, https://doi.org/10.1109/TKDE.2013.73.

[4] L. Breiman, *Random forests*, Machine Learning, 45 (2001), pp. 5–32, https://doi.org/10.1023/A:1010933404324, https://doi.org/10.1023/A:1010933404324.

[5] T. Chen, I. Cano, and T. Zhou, *RABIT : A Reliable Allreduce and Broadcast Interface*, vol. 3, 2015, p. 2, https://pdfs.semanticscholar.org/cc42/7b070f214ad11f4b8e7e4e0f0a5bfa9d55bf.pdf.

[6] T. Chen and C. Guestrin, *XGBoost: A Scalable Tree Boosting System*, in Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, New York, NY, USA, 2016, ACM, pp. 785–794, https://doi.org/10.1145/2939672.2939785, http://doi.acm.org/10.1145/2939672.2939785.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus, *Making Pointer-Based Data Structures Cache Conscious*, Computer, 33 (2000), pp. 67–74, https://doi.org/10.1109/2.889095.

[8] H. Cho and M. Li, *Treelite: toolbox for decision tree deployment*, (2018), http://hyunsu-cho.io/preprints/treelite_sysml.pdf.

[9] A. Criminisi, E. Konukoglu, and J. Shotton, *Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning*, tech. report, October 2011, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/decisionForests_MSR_TR_2011_114.pdf.

[10] B. V. Essen, C. Macaraeg, M. Gokhale, and R. Prenger, *"Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?"*, in 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, April 2012, pp. 232–239, https://doi.org/10.1109/FCCM.2012.47.

[11] L. Heutte, C. Petitjean, and C. Désir, *"Pruning Trees in Random Forest for Minimizing Non Detection in Medical Imaging"*, in Handbook of Pattern Recognition and Computer Vision, World Scientific, 2016, pp. 89–107.

[12] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, *Lightgbm: A highly efficient gradient boosting decision tree*, in Advances in Neural Information Processing Systems, 2017, pp. 3146–3154.

[13] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, *"FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs"*, in Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, New York, NY, USA, 2010, ACM, pp. 339–350, https://doi.org/10.1145/1807167.1807206, http://doi.acm.org/10.1145/1807167.1807206.

[14] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte, *"SIMD Parallelization of Applications that Traverse Irregular Data Structures"*, in Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Feb 2013, pp. 1–10, https://doi.org/10.1109/CGO.2013.6494989.

[15] T. Ye, H. Zhou, W. Y. Zou, B. Gao, and R. Zhang, *Rapidscorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization*, in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, 2018, pp. 941–950.