Efficient Realization of Decision Trees for Real-Time Inference

KUAN-HSUN CHEN, University of Twente, Netherlands
CHIAHUI SU, National Tsing Hua University, Taiwan
CHRISTIAN HAKERT and SEBASTIAN BUSCHJÄGER, Technical University of Dortmund, Germany
CHAO-LIN LEE and JENQ-KUEN LEE, National Tsing Hua University, Taiwan
KATHARINA MORIK and JIAN-JIA CHEN, Technical University of Dortmund, Germany

For timing-sensitive edge applications, the demand for efficient lightweight machine learning solutions has increased recently. Tree ensembles are among the state-of-the-art in many machine learning applications. While single decision trees are comparably small, an ensemble of trees can have a significant memory footprint leading to cache locality issues, which are crucial to performance in terms of execution time. In this work, we analyze memory-locality issues of the two most common realizations of decision trees, i.e., native and if-else trees. We highlight that both realizations demand a more careful memory layout to improve caching behavior and maximize performance. We adopt a probabilistic model of decision tree inference to find the best memory layout for each tree at the application layer. Further, we present an efficient heuristic to take architecture-dependent information into account thereby optimizing the given ensemble for a target computer architecture. Our code-generation framework, which is freely available on an open-source repository, produces optimized code sessions while preserving the structure and accuracy of the trees. With several real-world data sets, we evaluate the elapsed time of various tree realizations on server hardware as well as embedded systems for Intel and ARM processors. Our optimized memory layout achieves a reduction in execution time up to 75 % execution for server-class systems, and up to 70 % for embedded systems, respectively.

CCS Concepts: • **Computing methodologies → Classification and regression trees**; • **Computer systems organization → Embedded systems**; • **Software and its engineering → Software performance;**

Additional Key Words and Phrases: Architecture-Aware realization, optimized memory layout, random forest, cache-aware optimization, real-time inference

ACM Reference format:
Kuan-Hsun Chen, ChiaHui Su, Christian Hakert, Sebastian Buschjäger, Chao-Lin Lee, Jenq-Kuen Lee, Katharina Morik, and Jian-Jia Chen. 2022. Efficient Realization of Decision Trees for Real-Time Inference. *ACM Trans. Embedd. Comput. Syst.* 21, 6, Article 68 (October 2022), 26 pages.
https://doi.org/10.1145/3508019

This work has been partially funded by the Federal Ministry of Education and Research of Germany as part of the competence center for machine learning ML2R (project number 01IS18038A), supported by Deutsche Forschungsgemeinschaft (DFG) within the project OneMemory (project number 405422836), the SFB876 A1 (project number 124020371), and Deutscher Akademischer Austauschdienst (DAAD) within the Programme for Project-Related Personal Exchange (PPP) (project number 57559723).
Authors' addresses: K.-H. Chen, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands; C. Su, C.-L. Lee, and J.-K. Lee, National Tsing-Hua University, Department of Computer Science, No. 101, Section 2, Kuang-Fu Road, Hsinchu, Taiwan 30013; C. Hakert and J.-J. Chen, Technische Universität Dortmund, Fakultät für Informatik, Otto-Hahn-Str. 16, 44227 Dortmund; S. Buschjäger and K. Morik, Technische Universität Dortmund, Fakultät für Informatik, Otto-Hahn-Str. 12, 44227 Dortmund.

1539-9087/2022/10-ART68
https://doi.org/10.1145/3508019

1 INTRODUCTION

Data collection has become ubiquitous in the era of the Internet of Things, where sensors and restricted computing facilities are embedded into various physical objects [38]. A plenitude of such resource constrained devices gather volumes of data but also apply machine learning models in real-time at the edge. Decision tree ensembles as well as Deep Learning are currently among the state of the art in machine learning and are dominating machine learning competitions such as the ones hosted on www.kaggle.com. Kaggle is one of the largest online communities for data science challenges and machine learning competitions which frequently results in real-world applications of machine learning in, e.g., HIV research[1] or improving the search of the Higgs Boson[2]. For unstructured data such as text or images, Deep Learning is currently among the state of the art, whereas for structured data, decision-tree ensembles such as Gradient Boosting or Random Forest seem to work best[3]. Hence, it is no surprise that for real-time applications, tree ensembles have become important to augment our society in many fields, e.g., classification of celestial objects in astrophysics [11], pedestrian detection [32], 3D face analysis [17], noise signal analysis [37], nano-partical analysis [26, 40], etc.

Such applications have pushed the issue of efficient real-time inference forward [30]. In order to detect a particular event in these applications, a stream of sensor measurements must be classified. Such real-time scenarios therefore require the learned model deployed on edge computing devices to be efficiently executed. Consider an example setup, where battery powered sensor nodes are placed in a field and collect environment data. In order to save energy for radio transmission, the sensor nodes directly apply machine learning models to the raw sensor data and only submit the classification or regression result to a central instance. Optimizing the execution of the machine learning model in order to save execution time and thus energy, can help to extend the battery lifetime and increase maintenance cycles significantly. Since decision trees and random forests are popular candidates for such extreme resource constrained environments, this work provides means to optimize their execution time in such scenarios.

Whereas machine learning research has focused for a long time on purely algorithmic properties, the borderline between realizational details and algorithmic contributions has become blurred. It has been shown that the caching behaviour of realized algorithms determines the performance even more than algorithmic differences [34]. For real-time applications, the caching behaviour to **decision trees (DT)** is also critical. Such a problem is elevated further by ensembles of trees such as **Random Forests (RF)** that combine multiple trees into a single classifier and improve the classification accuracy, which challenge the cache significantly. It follows that the memory footprint for these models to host all tree nodes is often larger than the size of cache memories equipped, especially on embedded systems and therefore a clever *memory layout* is required. Several designs for DTs on memory layout have been proposed in the literature, e.g., [4, 22], but none considers the setting of real-time inference. The conference version of this work [9] is the first to focus on the optimization for real-time inference by taking the caching behavior into account.

In order to provide more intuition for the studied problem, we explain an extremely simplified example in Figure 1. On the left side of the figure, an extremely small decision tree is illustrated. The labels at the edges depict the empirical probability, which node of the tree is inferred subsequently. In this example, we assume that the tree is inferred on the most probable path, i.e., n0, n2, n6. The naive mapping would traverse the tree layer wise and place the nodes in memory, as illustrated in the top right of the figure. If we now assume that 4 nodes can fit into one cache line, two cache

[1] https://www.science.org/doi/10.1126/science.331.6018.698.
[2] https://www.symmetrymagazine.org/article/july-2014/the-machine-learning-community-takes-on-the-higgs.
[3] https://wandb.ai/site/articles/ama-with-anthony-goldbloom-ceo-of-kaggle.

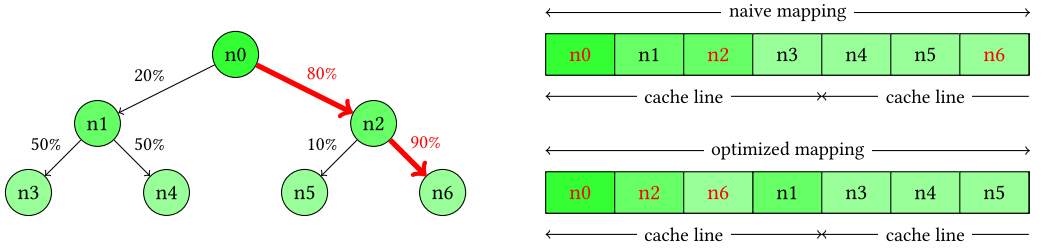

Fig. 1. Simplified tree mapping example.

lines need to be entirely loaded from main memory, in order to execute the tree. Consequently, an extremely simple optimization is to sort all tree nodes by their absolute access probability (n0: 100%, n1: 20%, n2: 80%, n3: 10%, n4: 10%, n5: 8%, n6:72%) and place this order in memory, as depicted in the bottom right part of the figure. It can be seen that only one cache line needs to be fully loaded from main memory in order to execute the tree along this path. Since cache lines are always entirely loaded from main memory to the cache, 50% of memory accesses can be saved in this scenario.

In this work, we reveal that two common DT realizations, i.e., 1) a loop to iterate over each tree node within a continuous data structure forming so-called *native tree*, and 2) a series of if-else blocks to unroll all possible iterations forming so-called *if-else tree*, may suffer from the memory-locality issues, especially when the trees are deep. Probabilities for accessing single nodes are usually not equal, e.g., the root has a higher probability to be accessed than a single leaf. Arranging trees in memory trivially as these common realizations neglect this probability distribution and thus potentially cause useless data- or instruction-prefetching, mitigating advantages of caches.

In general, after the initial training phase only the tree structure and its parameters is fixed, i.e., dependencies between nodes are static. However, the layout of the realization in memory remains open. For native trees, the layout is maintained by references to child nodes, which can be adjusted arbitrarily. For if-else trees, a similar concept can also be achieved by introducing `goto` statements to break the sequence of if-else blocks. Hence, our ultimate goal is to construct a mapping of tree nodes to memory locations, considering the probabilities of nodes to be accessed in such a way that the cache behaviour is optimized during inference. Since the cache memories equipped on different architectures might differ, our optimization algorithms particularly preserve some architecture-dependent parameters reflecting their impacts. Based on the preliminary results in [9], this manuscript further improves the code size estimation for if-else tree optimizations by analyzing the possible variances of generated instructions. To the best of our knowledge, this work is the first to accelerate the real-time inference of DTs by optimizing the memory layout to benefit the caching behaviors without sacrificing any accuracy.

Our Contributions in a Nutshell:

- We analyze the memory-locality issues of two common DT realizations, i.e., native and if-else trees (See Section 3)
- We present two cache-aware optimizations for each tree realization respectively, which utilizes the profiled probability of each node to arrange the memory layout of tree nodes in a path-wise manner (see Section 4)
- We provide a variance-aware node size estimation, based on the analysis of potential variance generated by the GNU compiler optimizations, to improve the efficiency of the if-else tree optimization (see Section 5).
- We conduct a comprehensive evaluation of cache misses and execution time reduction on server-class and embedded systems for the Intel and ARM CPU architectures over different

maximum tree depths, number of trees in an ensemble, and different budgets related to the size of the instruction-cache (see Section 6).

The source code of the developed framework is publicly available[4] on https://github.com/tudo-ls8/arch-forest.

2 DECISION TREES AND RANDOM FOREST

We consider supervised learning problems, in which we infer a model $\widehat{f} : \mathbb{R}^d \to \mathcal{Y}$ from labelled training data $\{(\vec{x}_i, y_i) | i = 1, \ldots, N\}$ to predict the value $\widehat{f}(\vec{x})$ of new, unseen observations. For $\mathcal{Y} = \mathbb{R}$ we have a regression problem, for $\mathcal{Y} = \{0, 1, \ldots\}$ we have a classification problem.

One of the most-used algorithms for these types of problems are decision tree ensembles. A **decision tree (DT)** is a binary search tree in which each inner node contains binary decision and the leaf nodes contain the predictions of the tree. The inner nodes use axis-aligned splits of the form $\mathbb{1}\{x_k \leq t\}$ where k is a pre-computed feature index and t is a pre-computed threshold. Depending on the outcome of $\mathbb{1}\{x_k \leq t\}$ either the left or the right child of the node is visited until a leaf node is found and its prediction is returned. Let $s_l(x) \colon \mathcal{X} \to \{0, 1\}$ be the series of splits which is '1' if x belongs to leaf l and '0' if not, then the prediction of a tree is given by $h(x) = \sum_{l=1}^{L_i} \widehat{y}_{i,l} s_{i,l}(x)$, where $\widehat{y}_{i,l} \in \mathbb{R}^C$ is the (constant) prediction value of leaf l and L_i is the total number of leaves in tree h_i. To compute the split features and split thresholds in each node the gini score (CART) or the information gain (ID3) is minimized which measure the impurity of a split. The induction starts with the root node and the entire dataset. Then the optimal splitting is computed and the training data is split into the left part ($1\{x_k \leq t\}$) and the right part ($1\{x_k > t\}$). The splitting is repeated until either a node is 'pure' (it contains only examples form one class) or another abort criterion, e.g., a maximum number of nodes in the tree, is reached. The predictions on the leave nodes are computed by estimating the class probabilities of all observations in that specific leaf.

A **Random Forest (RF)** extends a single DT by training a set of M axis-aligned decision trees and weighing them equally: $f(x) = \frac{1}{K} \sum_{i=1}^{K} h(x)$. Each tree is trained on a bootstrap sample of the original training and a subset of $d_i \ll d$ features to promote diversity among the trees. Algorithm 1 summarizes this approach.

In the classical Random Forest (RF) approach proposed by Breiman [8], the set of K DTs are trained with different samples of input features. In the literature, other RFs variations have been explored, such as those that train trees on samples of data (bagging) [5] or those that randomly generate trees without training at all [19]. However, it is common to all these methods, that they use tree-structured predictors as base learners and that they inject some form of randomness into the training.

In the theoretical analysis of these methods, we often encounter the fact that base learners should be as large as possible: Breiman has shown that bagging in general, and Random Forest specifically, reduce the variance of a biased learner [6]. Thus, for optimal performance, individual trees should minimize the bias error, which implies that they should not be restricted in size. In [7], Breiman extended his formal argument by empirical support. More recent theoretical analysis of RFs such as [2, 3, 14, 28] consistently support Breimans original claim that trees should be as large as possible. In short, recommendations for the optimal tree height range between $O(\log N)$ and $O(N)$, both, from a theoretical and empirical perspective. This makes RF fundamentally different

[4]The original framework was published in [9], which is available since 2018 on https://bitbucket.org/sbuschjaeger/arch-forest/.

ALGORITHM 1: Random Forest algorithm [8].

```
for i = 1, ..., K do
    X_i ← bootstrap_sample(X)
    h_i ← new_tree()
    nodes ← [(h_i, X_i)]
    while len(nodes) > 0 do
        n, X_n ← nodes.pop()
        split ← compute_best_split(X_n, d)
        n.set_split(X_n, split)
        X_l, X_r ← split_data(X_n, split)
        n_l, n_r ← new_children(n)
        if tree_not_done then
            nodes.append(n_l, X_l)
            nodes.append(n_r, X_r)
        end if
    end while
    trees.append(h_i)
    weights.append(1/K)
end for
```

from other ensemble learners such as Boosting [18], where the size of individual base learners are restricted to reduce over-fitting.

2.1 A Probabilistic View of DT Execution

For each DT, each node receives a unique identifier (e.g., in breath-first order) i. We denote the left child of i with $l(i)$ and the right child with $r(i)$. Let M denote the number of leaves in a DT. Since each leaf has a unique path from the root of the DT, there are M different paths from the root node to the leaves. Every node in the tree stores information about the feature, as well as a split-value against which the feature is compared to (split-point). Additionally, every leaf node stores its associated prediction (i.e., 0 or 1).

To classify a sample $\vec{x}$, the inference starts from the root node of the tree and follows the children according to the comparisons at each split node until a leaf node is reached. Afterwards, an associated prediction value of the leaf node is returned. Every observation takes exactly one path $\pi(\vec{x})$ from the root node to one leaf node. To simplify the notation, we drop the argument $\vec{x}$, if we are not interested in the path of a specific observation.

Following the probabilistic view of DT inference [10], we model each comparison at a split node i as a Bernoulli experiment in which one takes the path towards the left child with probability $p(i \to l(i))$ and, respectively, for the right child with $p(i \to r(i))$. It holds that $p(i \to l(i)) = 1 - p(i \to r(i))$. An example can be found in Figure 1. Please note that the probabilities $p(i \to l(i))$ and $p(i \to r(i))$ can be estimated during training by counting the number of samples at each node i taking the left and right path. Assume a path of length L with $\pi = (i_1, i_2, \ldots, i_L)$, where i_{j+1} is either the left or the right child of the j^{th} node on the path. Then, following this path consists of a series of Bernoulli experiments each with probability $p(i_j \to i_{j+1})$. Let $\mathcal{P}$ denote the set of all paths in the tree. The probability to take path $\pi \in \mathcal{P}$ is given by

$$p(\pi) = p(i_0 \to i_1) \cdot \ldots \cdot p(i_{L-1} \to i_L) = \prod_{j=0}^{L} p(i_j \to i_{j+1})$$

Again, let i be a node, then there is exactly one path $\pi = (0, \ldots, i)$ ending in node i. We denote the probability of the path leading to node i as $p(i) = p((0, \ldots, i))$. Let $\mathcal{T}$ be the set of all nodes in the tree, we define the probability for every subset of nodes $T \subseteq \mathcal{T}$ as:

$$p(T) = \sum_{i \in T} p(i)$$

2.2 Problem Definition

In this paper, we consider the performance optimization for executing a given tree ensemble model, i.e., a Random Forest with the probabilistic information of its DTs. We assume the accuracy of the learned model is ensured at the training phase, and the proposed optimizations at the *application layer* only focus on the efficient execution of DTs *without touching the tree structures*, i.e., dependencies between nodes are preserved. The aim of this work is to derive optimized code segments that optimize the memory locality to benefit the underlying caching behaviors. The considered performance metric is the *elapsed time* of executing the optimized realizations on a new, previously unseen observation. The optimization is applied with the knowledge of probabilities of the training data (known observations), hence the unseen observations may slightly differ in the access probabilities. This effect is therefore also included in the performance metric.

Consider a binary DT in Figure 1 as an example. When a tree node of the DT is executed, it either (a) reports the associated prediction if the node is a leaf, or (b) performs a comparison between a targeted feature with a threshold to decide the next destination, e.g., the left child or the right child. Two common DT realizations are considered here:

- A straightforward realization, named the *native* tree, uses a loop to iterate over each node of a tree within a continuous data structure, e.g., arranged by a one-dimensional array. An example code can be found in Listing 1.
- An alternative realization, named the *if-else* tree, statically generates if-else blocks. Here, the split values of a tree are all hard-coded as constant values into the instructions. An example code can be found in Listing 2.

```
1  struct Node {
2    bool isLeaf;
3    unsigned int prediction; // Predicted Label
4    unsigned char feature; // Targeted feature
5    float split; // Threshold
6    unsigned short leftChild;
7    unsigned short rightChild;
8  };
9  Node tree[] = {{0,0,0,8191,1,2},{0,0,1,2048,3,4},..]}
10 bool predict(short const x[3]){
11   unsigned int i = 0;
12   while(!tree[i].isLeaf) {
13     if (x[tree[i].feature] <= tree[i].split) {
14       i = tree[i].leftChild;
15     } else {
16       i = tree[i].rightChild;
17     }
18   }
19   return tree[i].prediction;
20 }
```

Listing 1. Example for Native tree structure in C++.

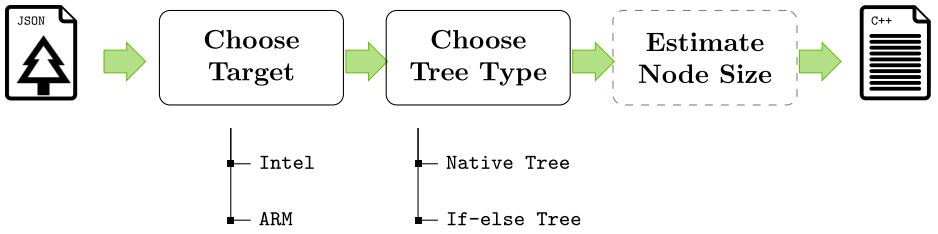

Fig. 2. Overview of the developed framework. The dashed block is only activated within the optimization of if-else tree. Here we take input with JSON format and output with C++ format to simplify the presentation.

Based on a given tree-ensemble model, which can be given as a JSON file, we develop a framework to generate optimized code sessions for *native* trees and *if-else* trees. As shown in Figure 2, firstly such abstracted models defined in Section 2 are derived from the given inputs. Afterwards, depending upon which target is chosen, the proposed optimizations (see Section 4) for different DT realizations will be conducted correspondingly. For the simplicity of the presentation, we assume the realization is generated in C++ for the rest of the paper.[5]

```
1  bool predict(short const x[3]){
2    if(x[0] <= 8191){
3      if(x[1] <= 2048){
4        return true;
5      } else {
6        return false;
7      }
8    } else {
9      if(x[2] <= 512){
10       return true;
11     } else {
12       return false;
13     }
14   }
15 }
```

Listing 2. Example for If-else structure in C++.

Although our optimizations are practiced at the application layer, which rely on the size of each node to manage the memory layout, architecture-dependent information should also be taken into consideration. Particularly, unlike in native trees, the node size in if-else trees depends on the targeted architecture and generated instructions, which can only be determined after the compilation phase. To this end, we analyze possible variances under O3 compilation option and employ `objdump` of the standard GNU binary utilities to capture the hardware-dependent information and improve the accuracy of node size estimation (see Section 5). Further optimizations can be achieved by the co-design of application and compilation, which is considered out of the scope, so left for the future work (see Section 8).

3 MEMORY LOCALITY

Due to the significant performance gap between the main memory (DRAM) and the processor, modern computer architectures have introduced a memory hierarchy. In addition to the main

[5]The insights of the proposed optimization can be generally applied for any common imperative programming language like C.

memory, smaller and faster memory subsystems next to the processors, in the forms of *cache* and *scratchpad memory*, are used to hide the *long* memory latencies of DRAM. Drepper provides very insightful discussions about the impact of memory hierarchy on the performance of programs [15].

In this paper, we consider modern computer systems with instruction and data-caches. The key assumption of the memory hierarchy is the *locality*:

- **Temporal locality:** Recently accessed items will be accessed in the near future, e.g., small program loops
- **Spatial locality:** Items at addresses close to the addresses of recently accessed items will be accessed in the near future, e.g., sequential accesses to elements of an array.

Unfortunately, two most common realizations of DTs *do not exploit such localities* when they classify a set of input data.

The benefit of the *native* tree realization is the temporal locality of the program, i.e., executing a tree is a simple loop with a few lines of codes. However, the accesses to the nodes of the tree do not have any spatial locality. The execution of a DT follows a *unique path* from the root to a leaf, which are stored in memory addresses that are unfortunately arranged discontinuously, if no attention is made. As a result, the cached data will not be further used, if the distance between each node of the path is greater than the number of nodes that can be loaded into a cache set at once.

As for the *if-else* tree realization, since the thresholds and the values needed for a split node of a tree are all hard-coded into the instructions, this avoids indirect memory accesses and has a clear advantage of the reduction of the latency. Therefore, the *if-else* tree realization does not suffer from missing data locality. However, without awareness of instruction-cache design, the hard-coded instructions may just be loaded into the instruction-cache once and only used once, so that the advantage of the temporal locality in the instruction-cache is completely abandoned.

There are three types of cache misses [23], namely compulsory, conflict, and capacity cache misses, which might be induced by the aforementioned locality issues. The *compulsory misses* are due to the first access to a memory block, which by definition is not in cache. The *capacity misses* occur when some memory blocks are discarded from the cache due to the limited cache capacity, i.e., the program working set is much larger than the cache capacity. The *conflict misses* occur in set associative or direct mapped caches when several blocks are mapped to the same cache set.

To optimize the caching behavior, the above cache misses should be reduced as much as possible. Therefore, the realization of a DT should take the layout of the data (in the *native* tree), the instructions of the branches (in the *if-else* trees), and the size of caches of the particular platform of execution into consideration.

4 OPTIMIZATIONS OF DECISION TREE

So far we have introduced the two standard approaches for realization DTs. In this section, we present how we optimize these two realizations. First, we discuss the downsides of each realization regarding their caching behaviour. Afterwards, we present how to optimize the memory layout at the application layer to benefit the underlying caching process.

4.1 Optimization of Native Tree

As shown in Listing 1 we can realize a DT by placing the nodes sequentially in an array and access this array by using a simple while loop. We observe that half of the nodes in a tree are leaf nodes, which only store a prediction value. The *naive native* realization, however, assumes the same data type for each node, leading to unnecessary overhead. Second, considering the usage of DTs for predicting classes, we notice that the data access pattern in the array is mostly non-sequential. The distance between each accessed element becomes bigger when the depth of targeted nodes in

the DT becomes greater. This phenomenon violates the spatial locality of the array and is harmful for locality, which may result in high cache misses.

Reducing compulsory cache misses. Nodes are prefetched into the cache sequentially. If we can reduce the amount of memory each node needs, we can fit more nodes into the cache and thus reduce compulsory cache misses. For the native realization we recognize that a leaf node only stores a prediction value, but does not use the pointer to its children, nor does it use the feature index or the split-value. An efficient way to reduce the memory consumption is to remove all leaf nodes from this array, and encode the prediction of leaves into the other fields. To achieve this, we abandon the `isLeaf` and `prediction` field of the native solution, but store the prediction of the left (right) child directly in the respective fields `left` (`right`) if the left (right) child is a leaf node. This method only requires us to layout one array, but offers the same size-reduction as using two arrays.

Reducing capacity and conflict cache misses. As mentioned in Section 3, if no attention is paid, the nodes stored in memory are arranged discontinuously. Thus, when a node is loaded into the cache, the nearby nodes should be on the same path to reduce capacity and conflict cache misses. A sensible way to exploit the data locality is to allocate as many nodes as possible on the same path into the same cache set.

To do so, we propose the following approach, where τ denotes the cache set size: Let `A` be the array in which we place all nodes of $\mathcal{T}$. Furthermore, let C be the candidate list of nodes in $\mathcal{T}$ which have not been placed in `A` yet and let $\mathcal{S}$ denote the nodes which should be placed in the same cache set. For each node, we greedily choose that child, which has the highest probability on the current path and try to place it in $\mathcal{S}$. Once $\mathcal{S}$ contains $\tau - 1$ elements (thus is full), we append all nodes form $\mathcal{S}$ to the array `A`, reset $\mathcal{S}$ and continue with the next cache set. Algorithm 2 summarizes this method. When adding a new node to $\mathcal{S}$, attention has to be paid, because there are two types of nodes (Line 7):

- The current node is a split node. Then we pick the next node based on the children's probabilities and put the more probable child into $\mathcal{S}$ and the other child into the candidate list C.
- The checked node is a leaf node, i.e., it is the end of the path: We pick up a sub-root with the highest probability from the candidate list C as long as it is not empty. The traverse starts again until $\mathcal{S}$ is full.

If the current $\mathcal{S}$ is full before finishing a traverse of a path (Line 14), two children should be put back to the candidate list C (Line 16). A sub-root which has the highest probability should be picked up from C for the next new set $\mathcal{S}$. Once a set is finished, the nodes in it will be allocated into the data array sequentially. To the end, the output of the algorithm is the data array with a path-oriented layout, in which path-oriented sets are sequentially allocated into the array.

Please note that the proposed approaches in a) and b) both may be applied while realizing the optimization for *native* trees. To do so, an additional field is required in the node structure that indicates whether the prediction is embedded in the respective fields `left(right)`. In Algorithm 2, the leaf-node case can be skipped technically, whereas the split-node case has to consider this additional field accordingly.

4.2 Optimization of If-Else Tree

As already mentioned, we can unroll the comparisons of a DT into conditional statements forming an if-else structure (cf. Listing 2). As the entire tree is transformed into if-else blocks, the execution features inherent locality compared to *native* trees, as only the code is accessed during inference. However, the code size can still exceed the instruction cache and cause cache misses.

ALGORITHM 2: Optimized *native* Tree
Input: Tree-nodes $\mathcal{T}$, maximum nodes per set τ
Output: A data array A with the path-oriented layout

```
1:  A = [ ]
2:  C ← {0}
3:  while C ≠ ∅ do
4:      i ← arg max_{j∈C}{p(π(j))}
5:      C ← C \ {i}
6:      S ← {i}
7:      while |S| ≠ τ do
8:          if i is leaf-node and C ≠ ∅ then
9:              i ← arg max_{j∈C}{p(π(j))}
10:             C ← C \ {i}
11:         else
12:             C ← C ∪ arg min{p(i → l(i)), p(i → r(i))}
13:             i ← arg max{p(i → l(i)), p(i → r(i))}
14:             if |S| = τ − 1 then
15:                 //this is the last node in S
16:                 C ← C ∪ {l(i), r(i)}
17:             end if
18:         end if
19:         S ← S ∪ {i}
20:     end while
21:     A.append(S)
22: end while
23: return A
```

Reducing compulsory cache misses. When an instruction cache miss takes place, several instructions are sequentially fetched into the instruction cache. When a branch is executed, these prefetched instructions will possibly not be utilized. If we can increase the chance of actually using prefetched instructions, we can reduce the number of the compulsory cache misses. However, DTs are naturally composed of many branches. To reduce the possibility of branch executions for tree $\mathcal{T}$, we can traverse all its paths and swap the children of every node i when $p(i \to l(i)) \geq p(i \to r(i))$. By this way, we can decrease the possibility to branch out of the current block, which in turn increases the utilization of prefetched code blocks.

Reducing capacity and conflict cache misses. The best case for exploiting the instruction-cache fully is having all the instructions of the *if-else* tree loaded into the instruction-cache. However, if the size of the instructions from the overall tree structure is greater than the size of the instruction-cache, the cached instructions may be evicted out by loading other instructions due to the capacity and conflict cache misses. Considering the usage of DTs, we can notice that keeping the instructions of those nodes utilized frequently in the instruction-cache can improve the utilization of the cached instructions, resulting in better performance.

With the above idea, we can define a computation kernel containing those nodes which are used most of time. For example, note that the root node of a tree is used in every case and thus it should be kept inside the cache all the time. Let $\mathcal{K}$ denote the kernel and let $s(i)$ be a mapping function returning the instruction size of node i. Our objective is to solve the following optimization problem:

$$\mathcal{K} = \arg\max \left\{ p(T) \middle| T \subseteq \mathcal{T} \text{ s.t. } \sum_{i \in T} s(i) \leq B \right\} \tag{1}$$

ALGORITHM 3: Optimized *if-else* tree
Input: Tree $\mathcal{T}$, Paths $\mathcal{P} = \{\pi_1, \ldots, \pi_M\}$
Output: Kernel $\mathcal{K}$, Label $\mathcal{L}$

```
swapChildren(T)
P ← sortByProbabilities(P)
b ← 0
for π ∈ P do
    for i ∈ π do
        if b + s(i) > B then
            Add i to L
        else
            Add i to K
            b ← b + s(i)
        end if
    end for
end for
```

where $\mathcal{B}$ is a given budget related to the size of the instruction-cache on the targeted architecture. Given $\mathcal{K}$, we can make sure, that these nodes are likely to remain in the cache, whereas the remaining nodes $\mathcal{L} = \mathcal{P} \setminus \mathcal{K}$ may be evicted more often.

In order to solve Equation 1 we need to iterate over all possible subsets of $\mathcal{T}$ which might be difficult for large trees. Thus, we propose a greedy approach in which we look at a complete path from root to leaf node: First, we swap the children depending on their probabilities as already explained in the former paragraph. Then, we sort all paths in the tree by their probability. After that, we greedily add a node one by one into $\mathcal{K}$ until the accumulated size of the added nodes $\sum_{i \in T} s(i)$ is greater than the given budget $\mathcal{B}$. The rest of nodes are all added into $\mathcal{L}$. Algorithm 3 summarizes the presented approach.

Once the nodes are grouped into $\mathcal{K}$ and $\mathcal{L}$ respectively, we can use `goto` statements to break the sequential generation of if-else blocks: First, we generate if-else blocks for all nodes in $\mathcal{K}$. Once the left/right child of one of those nodes is in $\mathcal{L}$, a goto statement is generated at the same position to replace the original if-else statement. Then, the corresponding if-else statements of this node and its children are all generated into a label block at the end. Listing 3 shows an example based on Listing 2 by applying Algorithm 3.

5 VARIANCE-AWARE NODE SIZE ESTIMATION

In the previous section, $s(\cdot)$ as a mapping function returning the instruction size of nodes, has to be designed as precise as possible, so that Algorithm 3 can group nodes into kernels efficiently. However, the question remains how to estimate the instruction size $s(\cdot)$ of each node. In [9], we count the number of generated instructions on the targeted architecture in an isolated example and form a static look-up table for two different types of nodes. By doing so, in fact we take a strict assumption that same type of nodes results in same code size. However, we should note that the actual number of instructions may greatly differ depending on how aggressive the compilation optimization is conducted.

To precisely capture the code size, first we analyze the possible variances from the disassembly, which can be obtained from the executable binaries by utilizing `objdump` of the standard GNU binary utilities. The considered families are x86-64 for Intel; ARMv7 and ARMv8 for ARM. We narrow our attention on the optimization option O3, which enables all supported optimization, and discuss the possible variances of generated nodes. Afterwards, we present how we automate the estimation for the instruction size of tree nodes.

Table 1. The Initial Size and Variance Thresholds for ARMv7 Processor

Type	Initial size	First threshold	Second threshold	Increase size
Int	8	31	1023	2
Short	10	4095	–	4
Char	8	31	4095	2
Float	24	255	–	4

```
bool predict(short const x[3]){
  if(x[0] > 8191){
    if(x[2] <= 512){
      return true;
    } else {
      return false;
    }
  } else {
    goto Label0;
  }
Label0:
  {
    if(x[1] <= 2048){
      return true;
    } else {
      return false;
    }
  }
}
```

Listing 3. If-else structure in C++ with goto statements.

5.1 Variance Analysis of Compiler Optimization

Split Node. We recall that the split nodes of if-else trees are realized by if-else blocks, where a feature of data is compared with a threshold. Given a split node, we analyze the potential variances for Intel and ARM processors:

- For Intel, without optimizations, i.e., if option O0 is enabled, there are two possible variances depending on the child of the split node. If both of the children are leaves, one compare-and-move instruction (`cmovlt` or `cmovge`) will be generated. Otherwise, one compare instruction (`cmpl`) and one jump instruction (`jg` or `jle`) will be generated. Depending on the value of the compared feature index and the threshold, the size of the spilt node is ranged from 5 to 12 bytes. Moreover, an optimization will further take place when the child of the split node is also a split node. If these two split nodes compare the same feature. Two sets of instructions will be combined into one load instruction and one compare instruction.
- For ARMv7, the generated instructions may differ greatly in terms of length and types, since the 16 bits thumb instruction set is involved. Figure 3 illustrates the estimation process. First, the size of a split node depends on the data type, split value, feature, and the children of this node. Without any optimization, the split node is formed by a load instruction (`ldr`), a comparison instruction (`cmp`), and a branch instruction (`bgt.n` or `ble.n`). The initial size of these instructions is shown in Table 1. If the split value is larger than 255, a longer comparison instruction will be generated. If the size of feature is larger than the first threshold, the load

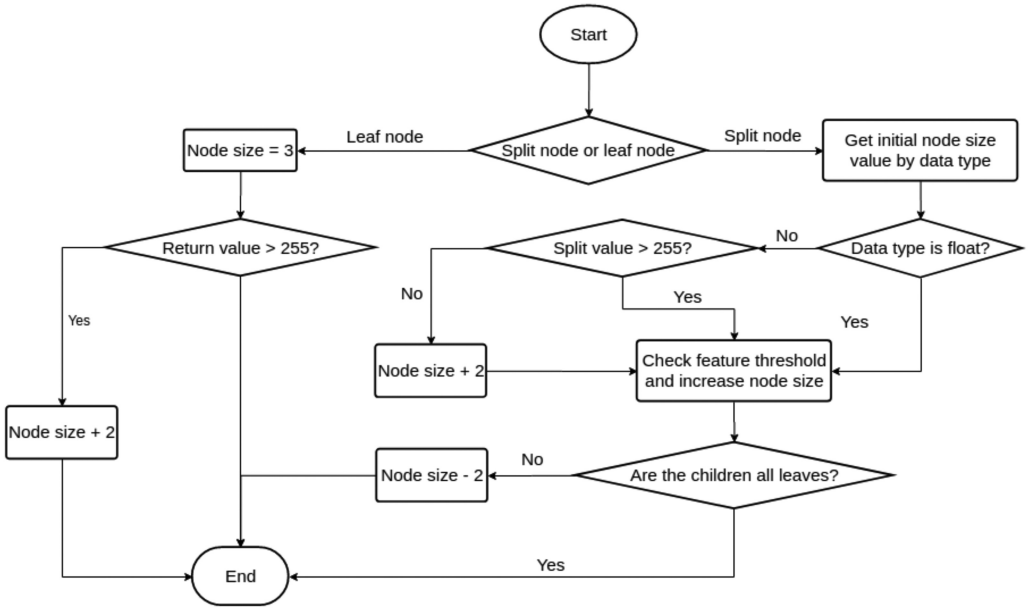

Fig. 3. Flowchart of ARMv7 node size estimation.

instruction is changed to a longer version. If the feature is larger than the second threshold, the compiler uses an add instruction (`add.w`) to assist the load instruction. Finally, if the two children of the split node are both leaf nodes, a 2 bytes if-then or if-then-equal instruction (`it` or `ite`) will be associated with the generated instructions of leaf nodes, instead of the 4 bytes branch instruction.

- For ARMv8, the variance of generated instructions is rather simple, and each instruction is always 4 bytes in our study. Usually a node consists of three instructions, load (`ldr`), compare (`cmp`), and branch (`b.gt` or `b.le`). Interestingly, if the split value is exactly a power of two, e.g., exactly equal to 2^{12}, a left-shift operation (`lsl`) will be embedded into the compare instruction. When the split value is greater than 2^{12}, one additional move instruction (`mov`) will be generated. Similarly, if the children of a split node are all leaves, a conditional set instruction (`cset`) will be associated with the generated instructions of leaf nodes without a branch instruction.

Leaf Node. We recall that a leaf node in the if-else tree is simply a return statement at the application layer. Given a leaf node in the compilation, however, its size might still differ depending upon the generated instructions.

- For Intel, if the return value is not 0, a move instruction (`mov`) will be generated to load the return value into a register. Otherwise, an XOR instruction (`xor`) operating on a register itself will be generated.
- For ARMv7, the estimation flow can also be found in Figure 3. If the sibling of the current leaf node is not a leaf node, the generated instructions will be a move instruction (`movgt` or `movle`) and a branch instruction (`bx`). If the return value is larger than 255, the size of the move instruction is increased with 2 bytes. If the sibling is a leaf node, the generated instruction is only one move instruction without a branch instruction, which is associated with the if-then instruction mentioned previously.

- For ARMv8, by default, a leaf node leads to one move instruction (`mov`) and one unconditional-branch instruction (`ret`). However, the node type of its sibling determines the possible optimization. If the sibling is a leaf node, the generated instruction is only one unconditional-branch instruction, which is associated with a conditional-set instruction (`cset`) from its parent split node. Moreover, the leaf nodes which return the same value might be omitted technically by branching the corresponding split nodes to the same destination.

Overall, the aforementioned optimizations are automatically conducted by the compiler as long as the condition is met. Hence, the node size estimation should be aware of the possible variances. In the following section, we present how to estimate the node size in a node-wise manner and take the possible variances into consideration.

5.2 Variance-Aware Estimation

To estimate the code size for each node precisely, we prepare a table containing the size of each node for s(·) before the optimization of if-else tree. For each DT, we iterate and estimate over all nodes sequentially according to their node types and contents, i.e., compared features and return value, reflecting the corresponding optimization variances. The real content of each node is pre-generated into a dummy function in C++ for the following analysis and automation. This node-wise estimation simplifies the analysis by only considering the content of the checked node and the type of its children without requiring further information.

```
1  __attribute__((section("leaf_5"))) unsigned int test5(float const pX[11]){
2      return 2;
3  }
4  __attribute__((section("leafEmpty"))) void emp(int const pX[11]){}
```

Listing 4. Examples of dummy and targeted functions for a leaf node.

We employ `objdump` from the standard GNU binary utilities with flag "-h", by which the size of each session can be profiled. Each node is realized as one annotated function, by which a self-defined section can be later recognized in the output of `objdump` and the corresponding instructions will be partitioned into. As we are interested in the increased size of each node for Algorithm 3, instead of the size of the whole function, dummy split and leaf nodes are prepared with empty functions, by which the size of instructions from the function frame can be captured as well.

Listing 4 and 5 are the examples of targeted functions and a dummy functions for leaf and split nodes, respectively. In Listing 4, the increased size of a leaf node is only contributed by a return instruction "return 2;". Therefore, during the calculation for possible variances, according to the above analyses, the actual increased size contributed by this node can be obtained by subtracting the size of dummy leaf node. Likewise, the increased size of a split node "split_0" can be obtained by subtracting the size of a dummy split node from the calculated size of the targeted split node as well. Please note that the optimization analyzed in Section 5.1 might also take place on the targeted function of each single node. Therefore, the dummy function for split nodes contains a template of a split node, i.e., an if-else statement with a non-zero return value, and the real content of targeted split node is embedded in the if statement as a sub-level to ensure that the size estimation for each node is not further affected by the compilation optimization.

Alternatively, a variance-estimation can also be achieved without involving `objdump` and dummy functions to simplify the estimation procedure. For example, the possible generated instructions in `ARMv7`, can be identified earlier at the application layer with the flowchart shown in Figure 3. Both approaches are provided in the developed framework. Eventually, the node size s(·) for each tree node can be precisely derived and adopted in Algorithm 3. For the rest of the

Table 2. Evaluation Systems

	Intel Server	ARM Server	Intel Embedded	ARM Embedded
System	Dell PowerEdge R430	Gigabyte R181-T90	Digital Loggers Atomic PI	Hardkernel Odroid C2
CPU	Intel Xeon E5-2650L v4	Cavium ThunderX2 99xx	Intel Atom x5-Z8350	Amlogic S905
Spped	2.5 GHz	2.5 GHz	1.92 GHz	1.54GHz
RAM	62 GB	251 GB	1.9 GB	1.7 GB
L1 (i/d) cache	32 kB/32 kB	1.8 MB/1.8 MB	32 kB/24 kB	32 kB/32 kB
L2 cache	256 kB	14 MB	1 MB	512 kB
L3 cache	36 MB	64 MB	-	-
Compiler target	x86_64-linux-gnu	aarch64-linux-gnu	x86_64-linux-gnu	aarch64-linux-gnu
Compiler version	gcc 8.3	gcc 9.3	gcc 8.3	gcc 9.3

```
1  __attribute__((section("split_0"))) unsigned int test0(float const pX[11])
2  {
3      if(pX[0] <= 20){
4          if(pX[7] <= 0.9917550086975098){
5              return 10;
6          }
7          else return 40;
8      }
9      else return 30;
10 }
11 __attribute__((section("splitEmpty"))) unsigned int sp_emp(float const pX[11])
12 {
13     if(pX[0] <= 20){
14        return 10;
15     }
16     else return 30;
17 }
```

Listing 5. Examples of dummy and targeted functions for a split node.

paper, we only consider ARMv8 instruction set, so that we can compare the results of optimized realizations on server-class and embedded systems with ARM processors.

6 EVALUATION

In order to evaluate the aforementioned optimizations for DTs, we generate various configurations and compare them on different hardware platforms. We focus on both, the server domain and the embedded systems domain. For both system classes, we consider Intel and ARM, which are the two most common architectures nowadays.

Table 2 details the configuration of the four considered systems, which are compared throughout this evaluation. Please note that the considered embedded systems not only feature smaller main memory but also have lower clocking frequencies than the server-class systems. Furthermore, they also lack an L3 cache memory. To minimize external noise, we execute inference of test datasets one by one in single core execution in isolation on each system. We compile each experiment as a

Table 3. Summary of Data Sets for Our Experiments Based on UCI Data Sets [27]

Dataset	# Examples	# Features	Accuracy
adult	8141	64	0.85 - 0.86
bank	10297	59	0.88 - 0.89
letter	5000	16	0.71 - 0.95
magic	4755	10	0.82 - 0.85
satlog	2000	36	0.87 - 0.91
sensorless	14628	48	0.90 - 0.99
spambase	1151	57	0.93 - 0.95
wine-quality	1625	11	0.57 - 0.70

self containing binary and execute it under architectural performance measurement with the perf framework [20]. We extract the total number of L1 instruction cache misses[6] and the total elapsed time from each experiment and use them for comparison.

For our experiments we train random forests (RF) with a variable maximum tree depth and a variable amount of trees. Table 3 shows the considered datasets from the UCI Machine Learning Repository [27]. We limit our selection of datasets since we conduct evaluation on a broad landscape of test systems[7]. We train each RF configuration on each dataset and store the trained forests as generic JSON files. For training, we use the CART algorithm with the Gini-Score criterion for node-splitting and trained models through the `sklearn` package [35]. If the respective data-set comes with a pre-computed train/test split we use this during training. Otherwise, we use 75% of the data for training and 25% of the data for testing. The probability of each node is profiled during training. In addition to the number of features and the number of examples during test-time, we also report the range of accuracy. Please note that we do not perform any hyperparameter optimization with respect to the classification accuracy, but report the accuracy here to validate our tool-chain.

Subsequently, we import these JSON files into our optimization framework to generate a self-containing C++ realization for each dataset and ensure that optimized trees retain their accuracy:

- **naive:** This naive native placement takes the trees as they are and places them as a native tree in subsequent memory arrays in a breadth-first-search manner. Tree nodes are hereby placed layer wise, regardless of any profiled probability.
- **optnative:** This optimized native tree realization removes `isLeaf` and `prediction` field from the node structure, removes all leaf nodes from the node array, and encodes these leaf contents into the respective split nodes. Afterwards, Algorithm 2 is employed to optimize the memory layout of the native tree.
- **stdifelse:** This realization transforms the trained tree model into a standard realization of an if-else tree, ignoring profiled probabilities from the training.
- **optifelse:** This realization employs Algorithm 3 for optimizing the memory layout of the respective standard if-else tree. This includes the variance-aware estimation of node sizes presented in Section 5.

[6]The ARM embedded system (Odroid C2 from Hardkernel) does not implement L1 icache miss counting in perf.
[7]Some datasets used in the baseline are omitted, since the corresponding models cannot be simply executed on resource constrained embedded devices.

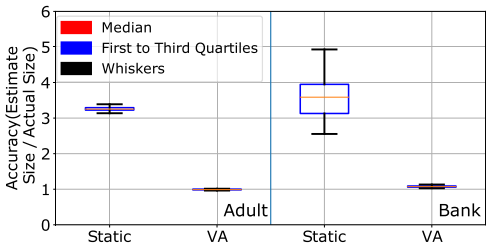

Fig. 4. Accuracy for Intel processor.

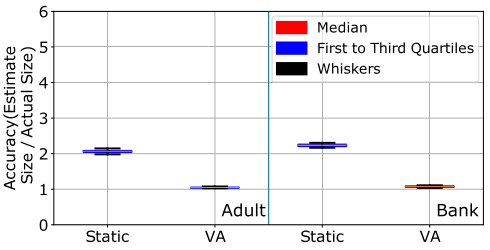

Fig. 5. Accuracy for ARMv8 processor.

Please note that, `sklearn` uses a probability-based majority vote, whereas we weigh all votes equally. Thus, final predictions may differ, but we could not detect any significant change in the final accuracy. Also note that, `sklearn` always produces floating-point split-values. For data-sets with integer features (e.g., the letter dataset) this is rounded down towards the next integer to circumvent the use of floating-point, which does not change the accuracy neither.

We note that the performance of our realization compared to `sklearn` might be of interest, since `sklearn` is arguably one of the most-used machine learning library and thus well-known to many practitioners. We found that our realization is on average $500 - 1500$ times faster than `sklearn`. However, we admit that this comparison is biased, because large parts of `sklearn` are written in Python and optimized for batch execution instead of real-time inference. Therefore, we focus the remaining evaluation on the analyzed realizations and optimizations.

6.1 Accuracy of Code Size Estimation

First, we evaluate the efficiency of the **variance-aware** estimation (**VA**) by comparing the estimated size of if-else trees with the static node size table as derived in [9] (**Static**), where the estimated size is accumulated by the estimated size of each node. As a baseline, we consider the actual code size reported by the `objdump` and normalize both estimated values with it to form the accuracy metric.

Figure 4 and 5 show the estimation accuracy, i.e., the estimated size normalized with the actual size, in box plots for given DTs in RFs, under two datasets, i.e., adult and bank, for Intel and ARMv8, respectively. The median of those derived DTs from RF is colored red. The blue box represents the interval from the first to the third quartile, while the black whiskers show the minimum and maximum of all of the data.

As shown in Figure 4, we can see that **VA** is way more accurate than **Static** for Intel processor. Specifically, under Bank dataset, **Static** may perform really worse (up to $\approx$ 5x), whereas the accumulated size of **VA** is really close to the actual tree size. For ARMv8, **VA** also outperforms **Static** as shown in Figure 5. Interestingly, **Static** performs closer to the actual code size for ARMv8, comparing to the derived results for Intel processor.

In general, we can conclude that **VA** outperforms **Static** regardless of targeted architectures. We empirically show that using constant values statically for the mapping function $s(\cdot)$ is not adequately accurate, which might consequently reduce the efficiency of if-else tree optimization, i.e., Algorithm 3. For the rest of the evaluation, we always adopt **VA** for the mapping function $s(\cdot)$ to maximize the efficiency of if-else tree optimization.

6.2 Budget Evaluation

Although the **optifelse** mapping algorithm profiles the size of single nodes precisely, the budget $\mathcal{B}$ used in Algorithm 3 remains a system dependent parameter for each kernel. This parameter cannot

Table 4. Random Forest Model Sizes

Max. tree depth	Number of trees	x86_64-linux-gnu size	aarch64-linux-gnu size
6	150	88 kB - 260kB	96 kB - 259 kB
8	150	185 kB - 758 kB	190 kB - 762 kB
10	150	294 kB - 1.64 MB	296 kB - 1.65 MB
12	150	388 kB - 2.71 MB	390 kB - 2.76 MB
14	150	474 kB - 3.71 MB	480 kB - 3.82 MB
16	150	548 kB - 4.47 MB	558 kB - 4.63 MB
18	150	603 kB - 4.91 MB	623 kB - 5.52 MB
20	150	640 kB - 6.22 MB	660 kB - 5.51 MB
15	5	9.73 kB - 87.1 kB	10.3 kB - 89.3 kB
15	10	63.2 kB - 501 kB	64.2 kB - 517 kB
15	25	128 kB - 1.00 MB	129 kB - 1.04 MB
15	50	252 kB - 2.04 MB	257 kB - 2.11 MB
15	100	379 kB - 3.09 MB	389 kB - 3.19 MB
15	150	511 kB - 4.17 MB	524 kB - 4.31 MB
15	200	648 kB - 5.15 MB	661 kB - 5.32 MB
15	400	775 kB - 6.20 MB	790 kB - 6.40 MB

directly be deferred from the system datasheet, since cache replacement policies and scheduling decisions can have potential influence on the final behaviour of the caches. Therefore, we perform a series of experiments with various budget sizes on all systems to experimentally determine the appropriate budget size. We split the experiments into two sets, one where we fix the number of trees within the ensemble to 150 and only increase the maximal depth of each single tree from 6 to 20. This set investigates the impact in tree shape and single tree size. In the second set, we fix the maximum tree depth to 15 and increase the number of trees within the ensemble from 10 to 400. This set increases the total model size, while keeping the shapes of single trees similar. To provide intuition on the total model sizes of these parameter configurations, we report the real model size for all tested configurations in Table 4. We acquire these numbers by scanning the total size of all tree inference functions in the compiled binary files. We average this size over all tree optimization strategies and report the smallest and biggest model each within all datasets. It can be seen that for both sets we have configurations which at most use few kilobytes up to configurations, which use few megabytes and therefore exceed the capacity of L1 Caches.

Figure 6 depicts the recorded amount of L1 instruction cache misses over various tree depth and 150 trees in the ensemble as a normalized value to the amount of L1 icache misses of the **stdifelse** realization. Since the ARM embedded system does not implement counting of L1 icache misses directly, we omit the discussion of the ARM embedded system here. The plotted points indicate the mean value across all datasets, the vertical bars indicate the deviation, respectively. To account for large variances across the different datasets we normalize the number of icache misses with respect to **stdifelse**, e.g., a point at 0.5 means 50% of the L1 icache misses of the baseline. In addition to the **stdifelse** realization, the **optifelse** realization with different budgets is included in the results. We conduct the experiments on a wide range of budgets. In order to provide better clarity in the figures, however, we only include three representative budget sizes for each systems, i.e., 128, 1024, and 16384. For the server-class systems, a budget size of 1024 achieves the best reduction of L1 icache misses, especially for deeper trees, while for the embedded systems a budget size of 256 delivers a good reduction, especially for smaller trees. In general, the budget size has a higher influence on the server-class systems than on the embedded systems.

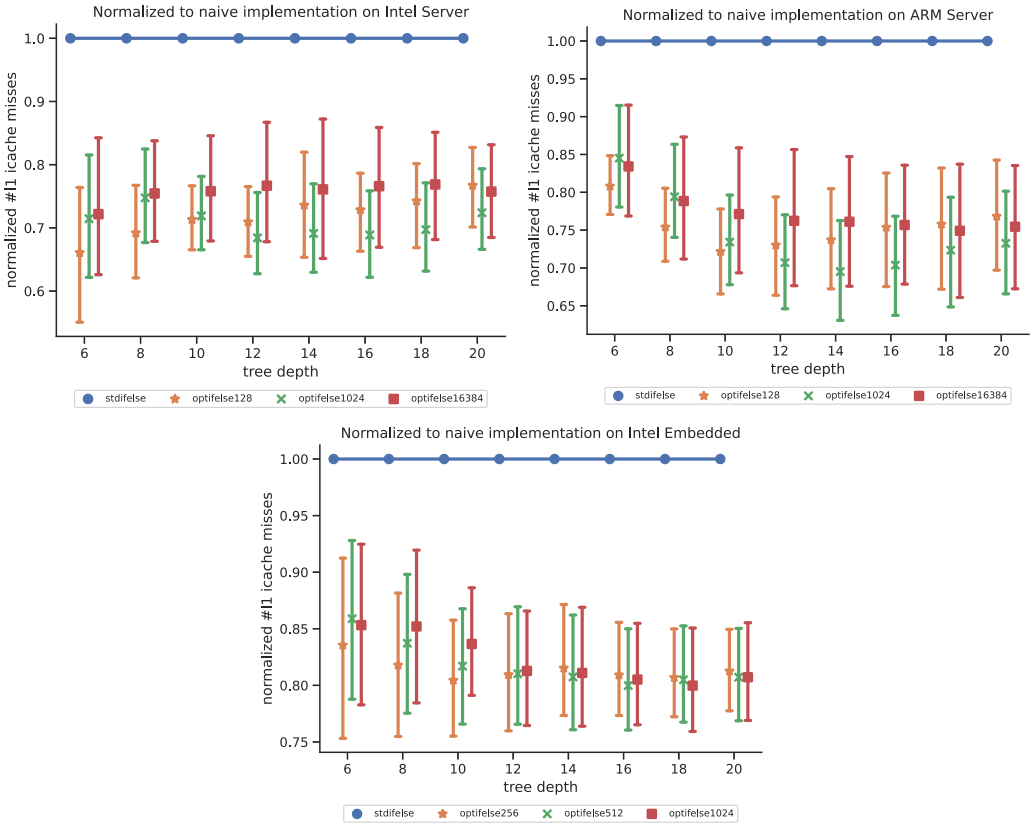

Fig. 6. Budget evaluation for variable tree depth.

Moreover, when fixing the tree depth and varying only the number of trees within the ensemble (Figure 7), similar results can be observed. For the server-class systems, similarly, a budget size of 1024 turns out to reduce L1 icache misses the most, for embedded systems only small differences in the chosen budget size can be observed. However, 256 turns out to be a good choice again. For the rest of the evaluation, therefore, 1024 and 256 are set as the budget size $\mathcal{B}$ for server-class systems and embedded systems, respectively.

6.3 Execution Time Evaluation

The previous evaluation on the reduction of L1 instruction cache misses reveals that our proposed optimized if-else tree placement reduces L1 icache misses by up to 40% already, compared to the **stdifelse** realization. The improvement in terms of execution time, however, does not solely depend on reduction in these L1 icache misses. Indeed, it is not obvious in which relation L1 icache misses and execution time resides. To this end, we measure the allover elapsed time of our executed experiments under different RF configurations. We stick to the experimentally determined budget size of 1024 for server-class systems and 256 for embedded systems. As a baseline, we consider the **naive** realization, since this is the simplest considerable realization. We compare this baseline to the **optnative** realization, the **stdifelse** realization and our **optifelse** realization.

Figure 8 illustrates the reduction in execution time when keeping the number of trees fixed and the maximal tree depth varying. It can be observed that for the server-class systems, the optimized

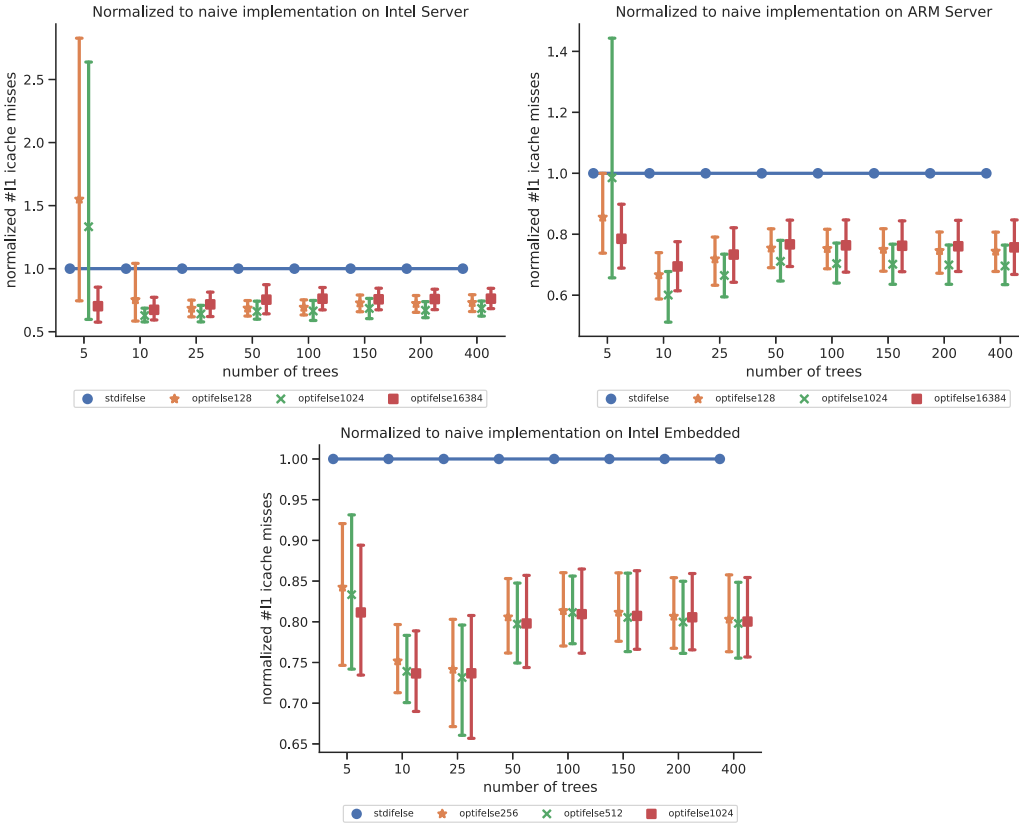

Fig. 7. Budget evaluation for variable number of trees.

if-else tree realization clearly dominates the other realizations. The execution time can be reduced by up to 75% in comparison to the **naive** realization, which makes a speedup of 4x. On the ARM Server it can be observed that the **optnative** can achieve a better execution time in comparison to **optifelse**, when it comes to deeper trees. The **optifelse** realization in this case, however, still saves more execution time. For embedded systems, in contrast, the **optifelse** realization only saves more execution time for rather small trees. For deeper trees, the **optnative** realization can achieve better results as the **optifelse** realization, i.e., 3x speed up on average. It should, however, be noticed, that the results of the **optnative** realization reside in the deviation band of the **optifelse** realization for most configurations. Thus, this effect also depends on the considered dataset.

Considering the fixed tree depth and a variable number of trees (Figure 9), similar observations can be made. For server-class systems the optimized if-else tree realization achieves the best reduction in terms of execution time. For embedded systems, however, this can be observed at most for rather small models. The **optnative** and **optifelse** realizations achieve similar improvements for bigger models here as well.

6.4 Discussion

The experiments and results highlight several interesting trends and insights. First of all, it can be observed that the Intel and ARM CPU architecture confront similar trends for servers, and also for embedded systems. The main difference in the systems we consider here is the presence of

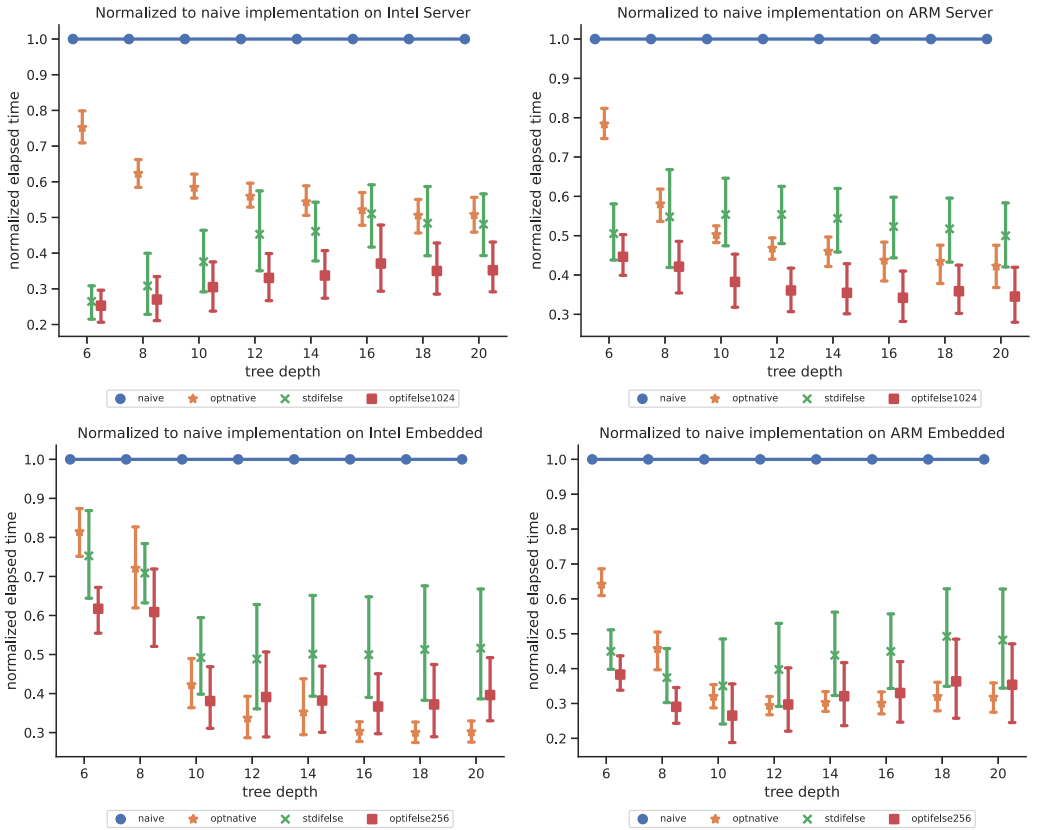

Fig. 8. Time evaluation for variable tree depth.

significant L3 cache for the server-class systems. The server-class systems are farther equipped with fast DDR4 RAM, while the embedded systems load memory contents from low power memory when a cache miss occurs. The evaluation results reveal that the **optifelse** realization exploits this additional potential on the server-class systems well and achieved a better performance improvement compared to the **optnative** realization.

Generally, it can be observed that the improvement in terms of elapsed time for server-class systems follows different trends. Please note that even the biggest models entirely fit into the L3 cache on both of server-class systems. For the embedded systems, in contrast, it can be observed that the optimization strategies improve the execution time further for bigger model sizes. These bigger models do not fit into the L2 cache of both of embedded systems and thus expensive memory accesses have to be performed, which are reduced by the presented optimization.

For the embedded systems, the **optifelse** realization cannot exploit hardware properties so far and achieves comparable low, partially worse results than the **optnative** realization. This draws the conclusion that the question of whether using **optifelse** or **optnative** is not necessarily an architecture-dependent question, but rather a question of available system resources. For trees with a small depth, for instance, if-else trees can be still better optimized on embedded systems than native trees. Therefore, also the question of model size and model complexity needs to be considered when choosing between if-else and native trees.

Another system dependent aspect, which has to be considered, is the choice of the appropriate budget size for the **optifelse** realization. Our experiments report that the budget size can have

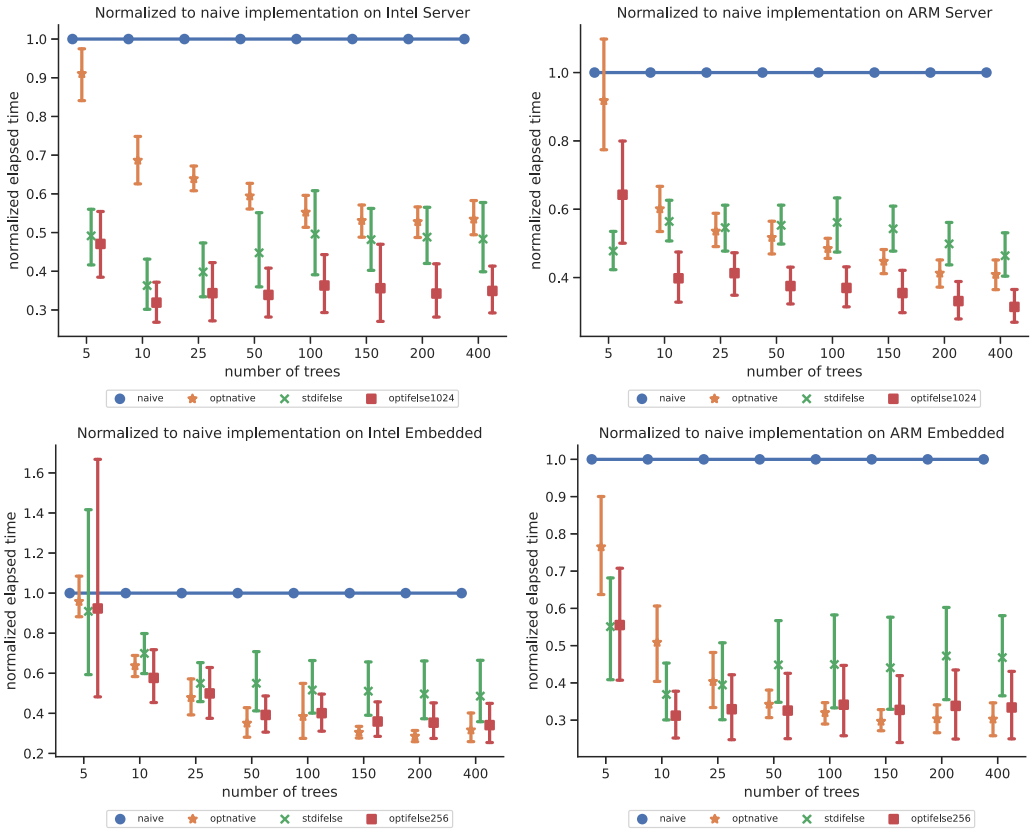

Fig. 9. Time evaluation for variable number of trees.

a rather large influence on the reduction of L1 icache misses. If this size is not chosen carefully, improvement in the execution time can be marginal. Thus, the budget size has to be profiled, to achieve significant results, especially for smaller models.

7 RELATED WORK

The realization of a given DT ensemble has been studied in literature. For example, Van Essen et al. present a comprehensive study of different architectures for realizing RFs on CPUs, FPGAs, and GPUs [39]. Based on the CATE algorithm, Prenger et al. train an RF with DTs constrained by a fixed height and show an effective pipelining approach for executing DTs on CPUs, FPGAs, and GPUs [36]. Nakandaka et al. propose a framework, which compiles decision trees into tensor operations in order to exploit parallelism [33].

Asadi et al. introduce different realization schemes of tree-based models in the context of learning-to-rank tasks [1]. They mainly introduce the two different realization schemes: The first one uses a while-loop to iterate over individual nodes of the tree (native tree), whereas the second approach decomposes each tree into its if-else structure (if-else tree). For the first realization, the authors also consider a continuous data-layout (i.e., an array of *structs*) to increase data locality but do not directly optimize each realization. Also, note that the authors mainly consider gradient boosted trees, at which the individual trees are usually “weak” in a sense, that they are comparably small, as opposed to larger trees in RFs.

In the context of ranking models, Lucchese et al. present the QuickScorer algorithm for gradient boosted trees [12]. In this approach, the authors discard the tree structure but decompose each tree into its comparisons. Then, they sort the comparisons of the entire ensemble according to the feature value and perform them one after another instead of traverse trees in a classical sense. To do so, they introduce a 2^{Δ} dimensional bit vector, where Δ is the height of a tree in which the **most significant bit (MSB)** signifies the prediction leaf node of that tree. This way, the algorithm can reuse comparisons across all ensemble members while minimizing cache misses. Lucchese et al. further enhance their method by adding vectorization over multiple examples for more efficient batch-processing [31]. To mitigate the limitations of a fixed height, Ye et al. propose to use an encoding scheme called epitome which they decode on-the-fly while also keeping the vectorization over the examples [41]. We note that while these methods usually offer a tremendous speed-up, they execute *all* possible comparisons in the entire ensemble in the worst case. Thus, they are especially effective for large ensembles of smaller trees commonly produced by gradient boosting algorithms. In a sense this approach is complementary to our approach which focuses on larger trees as commonly found in random forests.

Kim et al. present a realization for binary search trees using vectorization units on Intel CPUs and compare their realization against a GPU realization [24]. The authors provide insight on how to tailor the realization to Intel CPUs by taking into account register sizes, cache sizes, and page sizes. Their work is specialized for Intel CPUs and thus it is not directly applicable for different CPU architectures. Lucchese et al. already have noticed that many nodes are seldom visited [29]. Buschjäger and Morik formalize this observation by estimating the probabilities of specific paths during tree traversal [10]. Based on this probabilistic view of model execution, the authors consider different realization schemes for tree traversal and theoretically analyze their execution time. Note, however, that this model of computation remains at the software level and does not include the memory layout.

Estimation of the program code size has been an effective approach for compiler optimizations. Debray et al. [13] and Dreweke et al. [16] also demonstrate that the estimation of the code size can also be used for the procedural abstraction techniques for code size reduction in the compiler optimizations. Similar as in Section 5, Hakert et al. also annotate the observed sessions [21]. In order to recognize the sessions for further memory access analyses, debug symbols are added during compilation, whereas we annotate directly on each function representing each node at the application layer.

Designing algorithms to be cache-aware and exploit the memory architecture is a widely studied field, a summary of commonly used approaches is described by Kowarschik and Weiß [25]. The general concept of benefiting locality of memory accesses has to be applied specifically for certain algorithms, as presented in this paper.

8 CONCLUSION

Data collection has become ubiquitous, and a plenitude of resource-constrained devices are now not only gathering but also processing data. Thus, the efficient application of pre-trained models becomes increasingly important. Particularly, tree-ensembles have been widely used for many real-time applications.

In this paper, we optimized the memory layout for two most common tree realizations, i.e., native and if-else trees, at the application layer. In a nutshell, we observe that the allocation of nodes in the memory does not align with the assumption of using cache memories. Therefore, we propose the cache-aware realizations and systematically create the memory-locality by following the probabilistic view of DT execution.

Since the properties of provided cache memories and the size of generated code may greatly differ depending on the underlying CPU architectures, our optimizations employ a few architecture-dependent parameters to take their impacts into account. Towards this, we further develop an automatic size estimation to capture possible variances induced by GNU architecture-dependent compiler optimizations and estimate the size of tree nodes precisely.

Our evaluation compares the cache misses and the elapsed time after applying our optimizations with the baseline naive tree realization over server-class and embedded systems for Intel and ARM processors. The proposed optimization for if-else tree realizations indeed can reduce L1 instruction cache misses. For the elapsed time, our optimizations achieve 75% reduction for server-class systems with optimized if-else trees, and 70% reduction for embedded systems with optimized native trees.

Outlook: Although our optimizations can greatly reduce the elapsed time for two common DT realizations, we have not yet automated the selection of parameters according to the size and complexity of trained models. As can be observed from the evaluation, there are many parameters, i.e., tree depth, number of trees in the ensemble, and budget, that can eventually result in various speedups. The selection procedure for such parameters can be further automated in future work, together with the optimization conducted offline. A more thoughtful co-design between application and compiler should be further achieved to take more hardware dependent information into account, and which would result in more efficient tree realizations eventually.

REFERENCES

[1] N. Asadi, J. Lin, and A. P. de Vries. 2014. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering* 26, 9 (Sept 2014), 2281–2292. https://doi.org/10.1109/TKDE.2013.73

[2] G. Biau. 2012. Analysis of a random forests model. *Journal of Machine Learning Research* 13, Apr (2012), 1063–1095.

[3] G. Biau and E. Scornet. 2016. A random forest guided tour. *Test* 25, 2 (2016), 197–227.

[4] Jeffrey P. Bradford and Jose A. B. Fortes. 2001. Characterization and parallelization of decision-tree induction. *J. Parallel and Distributed Computing* 61, 3 (2001), 322–349.

[5] L. Breiman. 1996. Bagging predictors. *Machine Learning* 24, 2 (1996), 123–140.

[6] L. Breiman. 1996. Bias, variance, and arcing classifiers.

[7] L. Breiman. 2000. *Some Infinity Theory for Predictor Ensembles*. Technical Report. Technical Report 579, Statistics Dept. UCB.

[8] L. Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.

[9] Sebastian Buschjager, Kuan-Hsun Chen, Jian-Jia Chen, and Katharina Morik. 2018. Realization of random forest for real-time evaluation through tree framing. In *2018 IEEE International Conference on Data Mining (ICDM)*. 19–28. https://doi.org/10.1109/ICDM.2018.00017

[10] S. Buschjäger and K. Morik. 2017. Decision tree and random forest implementations for fast filtering of sensor data. *IEEE Transactions on Circuits and Systems I: Regular Papers* PP, 99 (2017), 1–14. https://doi.org/10.1109/TCSI.2017.2710627

[11] Jens Buss, Christian Bockermann, Katharina Morik, Wolfgang Rhode, and Tim Ruhe. 2016. FACT-Tools – Processing high-volume telescope data. In *26th Astronomical Data Analysis Software and Systems Conference (ADASS)*. ADASS.

[12] Domenico Dato, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2016. Fast ranking with additive ensembles of oblivious and non-oblivious regression trees. *ACM Transactions on Information Systems* (2016). https://doi.org/10.1145/2987380

[13] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 2 (2000), 378–415.

[14] M. Denil, D. Matheson, and N. De Freitas. 2014. Narrowing the gap: Random forests in theory and in practice. In *International Conference on Machine Learning (ICML)*.

[15] Ulrich Drepper. 2007. What Every Programmer Should Know About Memory.

[16] Alexander Dreweke, Marc Worlein, Ingrid Fischer, Dominic Schell, Thorsten Meinl, and Michael Philippsen. 2007. Graph-based procedural abstraction. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 259–270.

[17] Gabriele Fanelli, Matthias Dantone, Juergen Gall, Andrea Fossati, and Luc J. Van Gool. 2013. Random forests for real time 3D face analysis. *International Journal of Computer Vision* 101, 3 (2013), 437–458. https://doi.org/10.1007/s11263-012-0549-0

[18] Yoav Freund and Robert E. Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. System Sci.* 55, 1 (1997), 119–139. http://www.research.att.com/~schapire/papers/FreundSc95.ps.Z.

[19] P. Geurts, D. Ernst, and L. Wehenkel. 2006. Extremely randomized trees. *Machine Learning* 63, 1 (2006), 3–42.

[20] T. Gleixner and I. Molnar. 2012. Perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2021-05-31.

[21] Christian Hakert, Kuan-Hsun Chen, Simon Kuenzer, Sharan Santhanam, Shuo-Han Chen, Yuan-Hao Chang, Felipe Huici, and Jian-Jia Chen. 2020. Split'n trace NVM: Leveraging library OSes for semantic memory tracing. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6. https://doi.org/10.1109/NVMSA51238.2020.9188136

[22] Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen, Fazal Hameed, Jeronimo Castrillon, and Jian-Jia Chen. 2021. BLOwing trees to the ground: Layout optimization of decision trees on racetrack memory. In *58th ACM/IEEE Design Automation Conference (DAC), accepted*.

[23] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[24] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 339–350.

[25] Markus Kowarschik and Christian Weiß. 2003. An overview of cache optimization techniques and cache-aware numerical algorithms. *Algorithms for Memory Hierarchies* (2003), 213–232.

[26] Pascal Libuschewski. 2017. *Exploration of Cyber-physical Systems for GPGPU Computer Vision-based Detection of Biological Viruses*. Ph.D. Dissertation. Technical University of Dortmund, Germany. http://hdl.handle.net/2003/35929.

[27] M. Lichman. 2013. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml.

[28] G. Louppe. 2014. Understanding random forests: From theory to practice. *arXiv preprint arXiv:1407.7502* (2014).

[29] C. Lucchese, F. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini. 2015. QuickScorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 73–82.

[30] Claudio Lucchese, Franco Maria Nradini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2017. QuickScorer: Efficient traversal of large ensembles of decision trees. In *Procs. ECML PKDD 2017*. Springer, 383–387.

[31] Claudio Lucchese, Raffaele Perego, Franco Maria Nardini, Nicola Tonellotto, Salvatore Orlando, and Rossano Venturini. 2016. Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles. In *SIGIR 2016 - Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*. https://doi.org/10.1145/2911451.2914758

[32] Javier Marín, David Vázquez, Antonio M. López, Jaume Amores, and Bastian Leibe. 2013. Random forests of local experts for pedestrian detection. In *IEEE International Conference on Computer Vision, ICCV*. 2592–2599. https://doi.org/10.1109/ICCV.2013.322

[33] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 899–917.

[34] Siegfried Nijssen and Joost N. Kok. 2006. Frequent subgraph mines: Runtimes don't say everything. In *Procs. 4th Workshop on Mining and Learning with Graphs (MLG)*. 173–180.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[36] R. Prenger, B. Chen, T. Marlatt, and D. Merl. 2013. *Fast MAP Search for Compact Additive Tree Ensembles (CATE)*. Technical Report. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[37] Fatemeh Saki, Abhishek Sehgal, Issa M. S. Panahi, and Nasser Kehtarnavaz. 2016. Smartphone-based real-time classification of noise signals using subband features and random forest classifier. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*. 2204–2208.

[38] Marco Stolpe. 2016. The Internet of Things: Opportunities and challenges for distributed data analysis. *SIGKDD Explorations* 18, 1 (June 2016), 15–34.

[39] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger. 2012. Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium*. IEEE, 232–239.

[40] Mikail Yayla, Anas Toma, Kuan-Hsun Chen, Jan Eric Lenssen, Victoria Shpacovitch, Roland Hergenröder, Frank Weichert, and Jian-Jia Chen. 2019. Nanoparticle classification using frequency domain analysis on resource-limited platforms. *Sensors* 19, 19 (2019). https://doi.org/10.3390/s19194138

[41] Ting Ye, Hucheng Zhou, Will Y. Zou, Bin Gao, and Ruofei Zhang. 2018. RapidScorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. https://doi.org/10.1145/3219819.3219857

Received 28 June 2021; revised 2 November 2021; accepted 13 December 2021