technische universität
dortmund

ROLLED: Racetrack Memory Optimized Linear Layout and Efficient Decomposition of Decision Trees

Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen, Fazal Hameed, Jeronimo Castrillon and Jian-Jia Chen
TU Dortmund University, Germany
TU Dresden, Germany
University of Twente, Netherlands

Citation: Accepted in IEEE Transactions on Computers

@article{hakert2022rolled,
    author = {Hakert, Christian and Khan, Asif Ali and Chen, Kuan-Hsun and Hameed, Fazal and Castrillon, Jeronimo and Chen, Jian-Jia},
    title = {ROLLED: Racetrack Memory Optimized Linear Layout and Efficient Decomposition of Decision Trees},
    journal = {IEEE Transactions on Computers},
    year = {2022}
}

ROLLED: Racetrack Memory Optimized Linear Layout and Efficient Decomposition of Decision Trees

Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen, Fazal Hameed, Jeronimo Castrillon, *Senior Member, IEEE*, and Jian-Jia Chen, *Senior Member, IEEE*

Abstract—Modern low power distributed systems tend to integrate machine learning algorithms. In resource-constrained setups, the execution of the models has to be optimized for performance and energy consumption. Racetrack memory (RTM) promises to achieve these goals by offering unprecedented integration density, smaller access latency, and reduced energy consumption. However, to access data in RTM, it needs to be *shifted* to the *access port* first. We investigate decision trees and develop placement strategies to reduce the total number of shifts in RTM.

Decision trees allow profiling during training, resulting in tree paths' access probabilities. We map tree nodes to RTM so that the total number of shifts is minimal. Concretely, we present two different placement approaches: 1) where tree nodes are closely packed and placed *uniformly* in a single RTM location and 2) where decision tree nodes are *decomposed* to separate RTM blocks. We discuss theoretical cost models for both approaches, we formally prove an upper bound of $4\times$ for the unified and an upper bound of $12\times$ for the decomposed organization towards the optimal placement. We conduct a thorough experimental evaluation to compare our algorithms to the state-of-the-art placement strategies Our experimental evaluations show that the unified and decomposed solutions reduce the number of shifts by 58.1% and 80.1%, respectively, leading to a 53.8% and 46.3% reduction in the overall runtime and 52.6% and 61.7% reduction in the energy consumption, compared to a naive baseline.

Index Terms—Non-volatile Memory, Decision Tree, Optimal Linear Ordering, Racetrack Memory

1 Introduction

The rise of non-volatile memories (NVMs) as SRAM and DRAM competitive memory technologies allows systems to benefit from their richer densities, lower per-bit cost and energy consumption and comparable access latencies. Especially in battery-powered embedded systems, maintenance cycles can be significantly increased by carefully exploiting the advantages of NVMs and reduce the overall system energy consumption. An important application for low power computing "on the edge" is data processing and gathering, e.g., for distributed sensor nodes. Such setups can be improved by executing machine learning models already on the edge. One popular candidate for resource-constrained and efficient classification models are decision trees, since they do not require complex arithmetic operations and are highly configurable with only a few parameters. Assuming a decision tree should be executed on the edge to classify data points on the fly, the memory layout of the decision tree has to be carefully considered to achieve both energy efficiency and performance optimization.

Racetrack memory (RTM) is a new class of NVM, which features high integration density, low unit cost, and low energy consumption at the cost of access pattern specific *shift* latencies [1]. In RTM, data cannot be randomly accessed; it needs to be *shifted* to an access port first before it can be read out. The distance, i.e., how far the data needs to be shifted, defines the additional *shift* latency. Researchers target the problem of optimally mapping data structures to RTM, with respect to the shift latency by proposing placement heuristics, since exhaustively searching for the optimal placement is often not feasible [2], [3]. The heuristics usually profile the access probabilities of the data objects either in advance or during runtime. The major shortcoming of such placement heuristics is that they treat all data objects equally and, therefore, consider all data objects possibly being accessed pairwise consecutively.

A single cell in RTM is a magnetic nanowire equipped with one or more access ports and can store up to 100 data bits. The nanowires are grouped into *domain block clusters* (DBCs) that allow accessing all bits of a data word in parallel. The RTM array then consists of multiple DBCs, where each DBC has multiple locations. The selection of a target DBC in RTM requires no shifting and allows random accessing while accessing DBC locations is still sequential and requires shift operations. This provides an additional tuning knob, i.e., how data objects are placed within DBCs to minimize the necessary shifts and how they are distributed across DBCs. Existing optimization approaches to reduce the shift overhead try to find relations and dependencies between data objects and try to place such objects close together in order to reduce the shift overhead. As these approaches however have to assume a very generic structure of data objects, achieving optimality is likely not feasible.

Hence, we study domain specific placement approaches for decision trees in this paper, which assume a more concrete and simpler structure of the data objects, limited to the structure of binary trees. We investigate two strategies for organizing decision trees across racetrack DBCs. In the

first approach, we place the entire decision tree into a single DBC [4]. The width of the DBC is chosen such that a single position contains an entire node of the decision tree. This approach uses the decision tree nodes' probabilities but considers the nodes themselves as black boxes.

The decision tree node data structure consists of pointers to child nodes (two in the case of binary trees) and the node data (the split decision value). In every iteration of the tree traversal, only one child's pointer needs to be retrieved from the RTM. However, since all elements in the unified node are tightly coupled, the entire DBC is shifted to retrieve a particular node. The second approach uses this observation to decouple the pointer and data elements and store them in separate DBCs, resulting in a split value DBC, a left pointer DBC, and a right pointer DBC. This decomposition enables accessing RTM at the DBC granularity, thereby avoiding unnecessary shifts. For instance, if the tree is traversed towards the left child, only the split value and the left pointer DBCs need to be shifted, and the right pointer DBC remains unaffected.

The unified and decomposed approaches impose different costs in terms of required racetrack shifts during the traversal of the decision trees. We develop theoretical cost models that allow further argumentation and discussion about optimal solutions for the placement problem. We introduce a domain-specific placement algorithm and compare it to the optimal placements for both approaches. For both cases, we also proof and find upper bounds, i.e., we make sure that our placement algorithm delivers a solution that never requires more than $4\times$ the number of shifts of an optimal solution on the unified organization. For the decomposed organization, we prove that our placement algorithm does not cause more than $12\times$ the number of shifts an optimal solution would cause. We further prove that any specific placement on the unified organization cannot cause more than $3\times$ shifts on the decomposed organization.

In addition to the theoretical proofs and reasoning, we conduct a thorough experimental evaluation to compare the unified and decomposed approaches and our domain-specific placement algorithm to the state-of-the-art RTM placement algorithms. We compare the different solutions in terms of shift operations, runtime, and energy consumption. Concretely, we make the following contributions:

- A unified and a decomposed nodes' organization approach for decision trees on racetrack memory, including their formal cost models.
- A domain-specific placement algorithm for decision trees, including formal proofs of the upper bound towards the optimal solution on both organization approaches.
- Experimental evaluation and comparison to state-of-the-art methods, including end-to-end latency and energy evaluation.

2 System Model and Problem Definition

In this work, we target low-power embedded systems for machine learning inference. A typical scenario for such systems could be the deployment of battery-powered sensor nodes. Instead of transmitting the raw sensor data via radio transmission, the system could locally perform the model inference and only submit the derived result, thereby considerably saving transmission energy. The target system is assumed to be equipped with a simple CPU core (e.g., few MHz clock rate), a small main memory (e.g. SRAM or DRAM) and integrated RTM scratchpad memory. The RTM scratchpad is assumed to not be covered by further caches and directly serve requests from the CPU core. The system architecture is illustrated in Figure 1. Mapping the RTM scratchpad to a certain memory location may reduce the average access latency, the energy consumption for accesses to that memory location can be drastically reduced. This work assumes that the decision tree model is mapped to this RTM scratchpad memory, so the access patterns of the tree nodes determine the access latency and energy consumption. This work further assumes that the execution of a single decision tree is not parallelized across multiple cores, since parallelism in random forests is usually achieved by executing different trees on different cores in parallel.

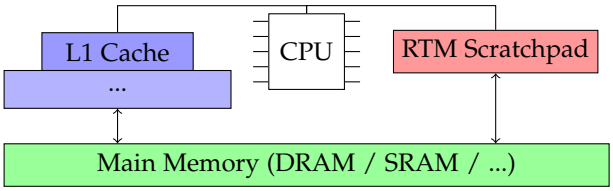


Fig. 1. System Memory Architecture

2.1 Decision Tree and Probabilistic Model

In this work, we consider *Decision Trees* as the inference model, where the leaf nodes contain the prediction values of the model under supervised learning. The input data is classified by its values for a fixed amount of features. Each inner node in the decision tree compares exactly one feature value from the input data with a fixed split value, deciding if the inference goes further to the left or the right child. Decision trees are a famous inference model for resource constrained machine learning. Furthermore, decision trees, in contrast to graph based networks, allow a probabilistic view on required data objects for the execution.

Each tree consists of nodes $N = \{n_0, n_1, ..., n_{m-1}\}$, divided into inner nodes N_i and leaf nodes N_l with $N = N_i \cup N_l$ and $N_i \cap N_l = \emptyset$, n_0 is the root node. Each node $n_x \in N \setminus \{n_0\}$ has exactly one parent node $P(n_x)$. Each node consists of three values: a split value, a pointer to the left child, and a pointer to the right child. In the unified organization, the entire node is mapped to a single array element in a consecutive array of size m. In the decomposed organization approach, we place each component of a node into a separate array, resulting in three arrays of size m. The indices of all components of a node in different DBCs, however, have to be synchronized. If the split value of node n_x is stored at index i, its corresponding left and right pointer values must also be stored at index i in their corresponding arrays. For a single array, the racetrack shifting cost of accessing index i and j with $0 \leq i, j < m$ is $|i - j|$. A valid placement of nodes N to array indices $I : N \to \{0, 1, ..., m-1\}$ must be bijective.

The inference model always starts at the root node and follows a certain path according to the comparisons

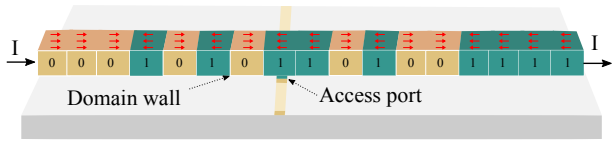


Fig. 2. RTM cell structure

at each node until reaching at a leaf node. By following the probabilistic model proposed in [5], each comparison is modeled as a Bernoulli experiment, by which each node is assigned a probability to be accessed from the parent node $prob : N \rightarrow [0,1] \subset \mathbb{Q}$ with $prob(n_0) = 1$ and $\forall n_p \in N_i : \sum_{n_x \in N : P(n_x)=n_p} prob(n_x) = 1$. That is, the sum of the probabilities of the children of the node n_p is 1.

2.2 RTM Cell Structure

The basic unit of storage in an RTM is a magnetic nanowire called *track*. Each track consists of multiple small magnetic regions (*domains*) which are separated by domain walls, and each of them has its own magnetization orientation as shown in Figure 2. A domain in a track represents a single bit (i.e., a "0" or "1") determined by its magnetization orientation. Each track is equipped with a single or multiple access ports responsible for performing a read or a write operation that requires the desired domain to be shifted along the track towards the access port by applying an electrical current. After aligning the desired domain to the respective access port, the relevant data is either read by sensing its magnetization orientation or written by updating its magnetization orientation.

2.3 RTM Architecture

The hierarchical organization of RTM, like other memory technologies, consists of banks, subarrays, domain wall block clusters (DBCs), tracks, and domains as depicted in Figure 3. Each structure at the highest level (e.g., bank) is decomposed into smaller structures at the next level (e.g., subarray). An RTM's essential structure is a DBC that contains T tracks, each comprising K domains. A single DBC can store K data objects with T-bit, where each object is stored in an interleaved pattern across the T tracks. Under a single port and K domains per track assumption, the shift cost to access a particular data object in a DBC may range from zero to $T \times (K-1)$.

A DBC can store up to 100 data objects, i.e., K can be as high as 100 [1]. However, many recent designs consider $K = 64$, which is more realistic and enables efficient utilization of the address bits This work also assumes that 64 nodes of a decision tree can be placed within a single DBC, containing a subtree of the maximal depth of 5. Since we use balanced decision trees in this paper, larger trees can be easily split into such subtrees by introducing dummy leaves, pointing to the next subtree. Subtrees in different DBCs can be accessed without additional shifting costs.

2.4 State-of-the-art Data Placement in RTMs

Recent works [2], [3] propose compiler-guided approximate and optimal solutions for objects placement in RTMs. A memory access trace S is represented with an undirected

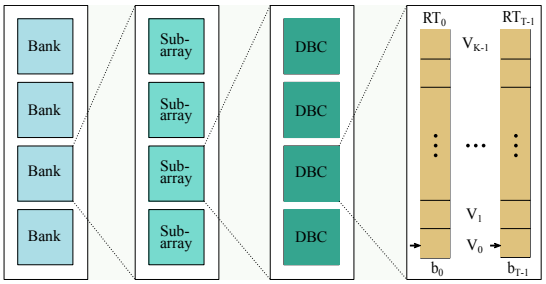


Fig. 3. An overview of the RTM hierarchical organization

graph of the form $G(V, E)$ where V is the set of vertices representing data objects and E is the set of edges between vertices. Each edge has an associated edge weight value corresponding to the number of consecutive occurrences of the connecting vertices. The heuristic in [3] maintains a single group g and assigns objects to it. In the first step, the data object with the highest access frequency (number of accesses) in S is assigned to it. Afterward, the remaining data objects (i.e., vertices in V) are appended to g one by one by prioritizing the vertex with the highest adjacency score. The chronological order in which vertices are added to the group determines the assignment of the corresponding data objects to the DBC, from left to right. However, this may lead to many costly long shifts because the data object with the highest frequency is placed on one end of the DBC. To overcome this problem, ShiftsReduce [2] uses a two-directional grouping to place the data objects with the highest access frequency in the middle of the DBC and places temporally close accesses at nearby locations inside the RTM.

2.5 Problem Definition

In this work, we focus on placement optimization to minimize the number of racetrack shifts for decision trees, which are trained beforehand, on memory devices with a single access port. This work is not about changing any logic structure of the decision tree, we take a logic representation of a trained tree as an input and determine a memory mapping, which maintains the logic structure. The **problem** is defined as follows:

- **Input:** A binary decision tree, consisting of a set N with m nodes, where each node is associated with a probability to be accessed from its parent. The probability is profiled on the training dataset. The information of the rooted tree is defined in Section 2.1.
- **Output:** A bijective placement of tree nodes to memory array indices that uses the node access probabilities and minimizes the required racetrack shifts while accessing the tree nodes during inference. The objective of minimized racetrack shifts is different for the unified and decomposed organizations.

Figure 4 illustrates a simplified instance of the problem. The input is the logic tree structure with profiled probabilities on the left, the output is a mapping of nodes to array indices on the right. The mapping results in a total expected shifting cost. For the upper mapping, the cost for shifting from the root to n_1 and back to the root is 2, the cost for shifting to n_2 and back is 4, thus weighted with the

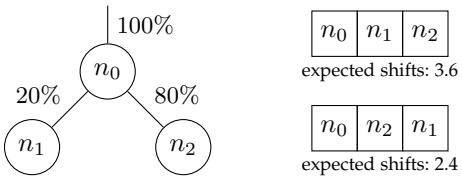


Fig. 4. Simplified example of optimized decision tree mapping

probability, the cost is $0.2 \cdot 2 + 0.8 \cdot 4$. Following the same consideration, the lower mapping is an optimized (yet not optimal) mapping, causing an expected cost of 2.4.

Due to the rooted tree structure, each node n_x in N has a unique access path from the root to n_x. We use $rlpath(n_x)$ to denote the set containing all nodes from the root node down to n_x. With the help of this, we declare the absolute access probability of node n_x as $absprob(n_x) = \Pi_{n_z \in rlpath(n_x)} prob(n_z)$. In addition, every node $n_x \in N$ has a subtree with a subset of leaf nodes $leafs(n_x) \subseteq N_l$ where $\forall n_y \in leafs(n_x) : n_x \in rlpath(n_y)$.

Definition 1. *For a given node $n_x \in N$, the sum of probabilities of its direct children must always be 1 (cf. Section 2.1). By definition, the absolute probability of n_x can be then expressed as:*

$$absprob(n_x) = \sum_{n_y \in leafs(n_x)} absprob(n_y) \tag{1}$$

3 Uniform Organization

This section presents our unified organization approach, i.e., placing all components of the decision tree node at one index in the DBC. We first define the cost model of decision tree execution for this approach and introduce our novel placement strategy subsequently. We deliver a formal proof, assessing the optimality of our strategy.

3.1 Cost Model

Given some valid placement I, the expected cost to infer an input value, i.e., following a path from the root to a leaf, is given by Eq. (2):

$$C_{down} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot |I(n_x) - I(P(n_x))| \tag{2}$$

After finishing one inference iteration, the DBC needs to be shifted back to the root node so that the next inference iteration can again start at the root. The expected cost of shifting from leaf nodes back to the root node is given by Eq. (3):

$$C_{up} = \sum_{n_x \in N_l} absprob(n_x) \cdot |I(n_x) - I(n_0)| \tag{3}$$

Combining them leads to the total expected shifting cost under the profiled dataset (Eq. (4)):

$$C_{total} = C_{down} + C_{up} \tag{4}$$

An optimal placement I^* for a decision tree on racetrack memory in the unified organization approach is a placement that reduces C_{total} to the absolute minimum. This problem is an instance of the *Optimal Linear Ordering* (OLO) problem [6]–[8]. The OLO problem, in general, is to map the nodes of a graph G to slots, where all slots are in a row, and adjacent slots are one unit apart, such that the total sum of arc weights multiplied with the distance between the nodes, connected by the arc, is minimal. The OLO (or also called *Optimal Linear Arrangement*) problem is an instance of the Quadratic Assignment Problem and is NP-complete [9]. As a special case, the OLO problem for rooted trees with the root node on the leftmost position can be optimally solved in time complexity $O(m \log m)$ [6]. Although decision trees are a rooted tree structure, the node access structure is a cyclic graph, since a leaf node is always followed by the root node for the next data tuple. Ignoring the cost of this arc in the access graph (i.e. only optimizing C_{down}) makes the optimization of the racetrack shifts within decision trees an instance of a rooted tree, but is not optimal for the total cost C_{total}. Therefore we analyze the optimally of the solution for C_{down} on C_{total} in the following.

3.2 Optimal Linear Ordering for Decision Trees

In this section, we prove an upper bound of the optimality of a placement, only considering C_{down}, on the studied problem of optimizing C_{total}. Therefore, we show how an optimal solution for C_{total} can be transformed into a solution, which has the form of the output of the optimal algorithm for optimizing C_{down}. For the different transformation steps, we explain the caused increase in shifting cost. Ultimately, we derive the upper bound from the fact that the transformed solution must not be better than the derived solution for C_{down}.

Throughout this section we use the notation defined in Table 1:

Placement	Explanation
I	arbitrary placement
I^*	optimal placement which optimizes C_{total}
C^*_{opt}	total cost C_{total} caused by I^*
$I^{*\downarrow}$	optimal placement which optimizes C_{down}
$C^{*\downarrow}_{down}$	C_{down} caused by $I^{*\downarrow}$
$\overleftarrow{I}$	arbitrary placement with the root on the left
$\overleftarrow{I^*}$	optimal placement with the root on the left
$\overleftarrow{C^*}_{down}$	C_{down} caused by $\overleftarrow{I^*}$

TABLE 1
Placement Notation

Suppose that C^*_{opt} is the minimum expected cost C_{total} of the optimal placement I^* of the decision tree. In the following, we show how to derive a sub-optimal placement, which at most causes 4 times the cost of C^*_{opt}. A root leaf path, defined as $rlpath(n_\ell)$, from the root node n_0 to a leaf node $n_\ell \in N_l$ in a placement I is *monotonically increasing* if $I(n_x) > I(P(n_x))$ for every node n_x in $rlpath(n_\ell) \setminus \{n_0\}$. Contrarily, such a path is *monotonically decreasing* if $I(n_x) < I(P(n_x))$ for every node n_x in $rlpath(n_\ell) \setminus \{n_0\}$.

Definition 2. *We define placement I **unidirectional** if all paths in the given decision tree are monotonically increasing in this placement.* □

Definition 3. *We define placement I **bidirectional** if every path in the decision tree is either monotonically increasing or monotonically decreasing.* □

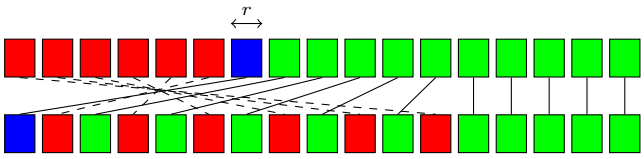


Fig. 5. Reassignment of nodes and root to the left

Lemma 1. *Let $I^{*\downarrow}$ be a placement which only minimizes $C^{*\downarrow}_{down}$ and ignores $C^{*\downarrow}_{up}$. Then,*

$$C^{*\downarrow}_{down} \leq C^{*}_{opt} \tag{5}$$

Proof. This comes from the definition as certain terms in the objective function are removed and all terms are positive. □

We now restate an existing property that was already used by Adolphson and Hu [6] regarding the optimization of $I^{*\downarrow}$ when the root *has to be put* on the leftmost position.

Lemma 2 (Page 410 in [6])**.** *(restated) There exists an optimal unidirectional placement $\overleftarrow{I^*}$ for the OLO problem when the input is a rooted tree, i.e., $\overleftarrow{C^*}_{down} = C^{*\downarrow}_{down}$, under the constraint that the root is on the left most position.*

Deriving a unidirectional or bidirectional placement induces the special property that optimizing C_{down} implicitly optimizes C_{up}, which is shown by the following lemma.

Lemma 3. *If a placement I is unidirectional or bidirectional, $C_{down} = C_{up}$.*

Proof. The full proof can be found in the appendix. Basically, in a unidirectional mapping the leaf is always the right most node, thus going from the root to the lead (down) is the same distance as going from the leaf to the root (up). □

In the following, we point out the relation between a placement I and a placement $\overleftarrow{I}$ which puts the root on the leftmost position.

Lemma 4. *Any placement I can be converted into a placement $\overleftarrow{I}$ which places the root on the left most position by increasing the expected cost of $\overleftarrow{C}_{down}$ with at most a factor of 2:*

$$\overleftarrow{C}_{down} \leq 2 \cdot C_{down} \tag{6}$$

Proof. For spatial and readability reasons, the full proof can be found in the appendix. The basic concept how to construct this transformation is illustrated in Figure 5, where the original mapping is illustrated in the top and the new mapping is illustrated in the bottom. The root is moved to the left most position. A symmetric amount of nodes around the original root is interleaved, such that the distance to the of each interleaved node is at most doubled. All other nodes can remain at their position, since the movement of the root increases their distance by a factor of less than 2.

□

Suppose that $\overleftarrow{I^*}$ is an optimal unidirectional placement of the rooted tree (with the root on the leftmost position) and optimizes the cost $\overleftarrow{C^*}_{down}$. Further suppose that $I^{*\downarrow}$ is an optimal placement which optimizes $C^{*\downarrow}_{down}$. We conclude the following corollary:

Corollary 1.

$$\overleftarrow{C^*}_{down} \leq 2 \cdot C^{*\downarrow}_{down} \tag{7}$$

Proof. $I^{*\downarrow}$ is an unconstrained placement that achieves the optimal $C^{*\downarrow}_{down}$. By Lemma 2, we know that $\overleftarrow{I^*}$ is an optimal placement for the cost $\overleftarrow{C^*}_{down}$ under the condition that the root is on the left most position. Therefore, $C^{*\downarrow}_{down}$ is a lower bound of any solution when the root is on the left most position. By Lemma 4, $I^{*\downarrow}$ can be converted into a placement $\overleftarrow{I}$, in which the root is put to the left most position, with a cost up to $\overleftarrow{C}_{down} \leq 2 \cdot C^{*\downarrow}_{down}$. Therefore, $\overleftarrow{I}^*$, as the optimal placement under the root constraint, must not cause a higher cost $\overleftarrow{C^*}_{down}$ than $\overleftarrow{C}_{down}$. □

Theorem 1. *An optimal unidirectional placement has an approximation factor of 4 of the studied problem.*

Proof. Based on Lemma 3, we know that the expected cost, denoted as $\overleftarrow{C^*}_{total}$, of the optimal unidirectional placement for the decision tree (including the down- and up-parts) is exactly $2 \cdot \overleftarrow{C^*}_{down}$. Therefore, together with Corollary 1 and Lemma 5, we reach the conclusion.

$$\overleftarrow{C^*}_{total} = 2 \cdot \overleftarrow{C^*}_{down} \leq 4 \cdot C^{*\downarrow}_{down} \leq 4 \cdot C^{*}_{opt}.$$

□

We now explain how to derive an optimal unidirectional solution that minimizes $\overleftarrow{C^*}_{down}$ efficiently. Adolphson and Hu [6] proposed an algorithm to solve this case optimally. Specifically, according to [6], the OLO problem for rooted trees with the root mapped to the leftmost slot is to find an optimal allowable linear ordering of tree nodes. An allowable linear ordering in their terminology means that if node $n_p = P(n_x)$ is the parent of node n_x, it has to be left of n_x in the ordering. The algorithm from Adolphson and Hu always derives an optimal allowable linear ordering to minimize the OLO problem in $O(m \log m)$ time complexity. The algorithm is implemented by recursively condensing subtrees underneath every node. This means, the algorithm decides whether further nodes of the underlying subtree should be placed close to the node, or if another node with relative high access probability should be put close in the mapping. This is achieved by dynamically keeping track of internal weights, which relate the node access probability and the length of mapped subtree nodes underneath. The algorithm basically skips mapping subtree nodes, once the increasing expected cost of other nodes exceeds the gain in expected cost for subtree nodes. Please note that the OLO problem is studied further in the literature and even more efficient algorithms for rooted trees are proposed (e.g. Skodinis proposes an algorithm with $O(m)$ runtime complexity [10]). However, these algorithms differ in their time complexity, but all of them provide optimal solutions to the OLO problem for rooted trees. In this paper we base on the linear allowable property from Adolphson and Hu. In addition, we compute the tree layouts offline, thus both, $O(m \log m)$ and $O(m)$, are feasible for all our trees.

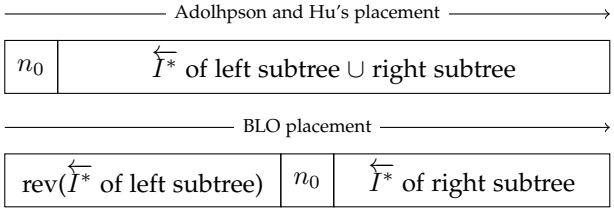


Fig. 6. Suboptimal Placement Correction

4 Bidirectional Linear Ordering

Deriving a placement by the algorithm from Adolphson and Hu at most causes $4\times$ the cost compared to the optimal solution for the unified organization approach. The algorithm from Adolphson and Hu has the major drawback of placing the root node to the leftmost slot in any solution, which is not optimal when the cost for going back from leafs to the root between inferences is considered. Our novel proposed

Algorithm 1 BLO Mapping Algorithm

Given a tree with root n_r
$T_L \leftarrow$ left subtree of n_r
$T_R \leftarrow$ right subtree of n_r
$I_L \leftarrow$ OLO mapping of T_L
$I_R \leftarrow$ OLO mapping of T_R

return $\{reverse(I_L), 0, I_R\}$

algorithm computes a *Bidirectional Linear Ordering* (BLO) (Algorithm 1). We map the two subtrees underneath the root by the algorithm from Adolphson and Hu, which derives a placement I_L for the left subtree and a placement I_R for the right subtree (Figure 6). Both placements cause an expected cost of at least two shifts less than the total expected cost of the entire tree since one node, and therefore a shift at least by one slot is missing on every root leaf path to a leaf and back to the root. We then form the final BLO placement by placing $I^\diamond = \{reverse(I_L), 0, I_R\}$. In this placement, two shifts are then added again to every root leaf path into and out of the right and left subtree, thus $C^\diamond_{total} \leq C_{total}$. In consequence, the upper bound of $4\times$ holds for BLO as well. The amount of shifts, however, is expected to be reduced by using BLO instead of OLO.

The reverse ordering can be done in $O(m)$, the placement of the root is performed with constant time overhead. Therefore, the time complexity of BLO is $O(m \log m)$.

5 Decomposed Organization

The last two sections explain the unified organization approach and discuss how the optimal linear ordering problem is related to our shifts minimization objective. This section focuses on the decomposed organization approach and analyzes how OLO and BLO perform for the decomposed trees. We revise the cost model and provide another formal proof about the solution's optimality for the OLO problem.

The decomposed approach is motivated by two major challenges in the unified organization approach: (1) it requires very wide DBCs and is less scalable (ii) leaf nodes that make $\approx 50\%$ of the total number of tree nodes do not need to store pointers for left and right child nodes. However, since the node information in the unified approach is tightly coupled, storage can not be optimized. This leads to storage wastage and yields suboptimal latency and energy consumption.

The DBC size is generally defined by two parameters, i.e., the number of (useful) domains per track and the number of tracks per DBC. Increasing the number of domains per track increases the capacity but at the cost of increased latency and increased position-error rate [11]. Similarly, the number of tracks per DBC affects the number of address bits, decoder's size, and ultimately performance and energy consumption. For a fixed size RTM, increasing the number of tracks per DBC reduces the number of DBCs and requires fewer address bits. However, this comes at the cost of storage wastage and increased energy consumption. Smaller width DBCs allow for storing different memory objects in different parts of the RTM that can be accessed and controlled independently. This also avoids wasting the RTM storage space.

We propose a decomposed approach to find a better solution to store decision trees in optimized width DBCs. We split every tree node into three components: (1) the split value/feature index, which is used to decide on an incoming data tuple to traverse the tree further to the left or right; (2) the left child pointer, and (3) the right child pointer. We place all these three components in separate DBCs at synchronized indices, leading to one DBC for right child pointers, one for left child pointers, and one for split values and feature indices. It should be noted here that we assume all DBCs to have the same width, such that they can be arbitrarily allocated to the split values or pointer values. As the indices need to be synchronized (i.e. the right pointer of node n_x has the same index in the right pointer DBC as the left pointer in the left pointer DBC), the placement I is modeled in the same manner as before. The central advantage of the decomposed DTs is that the width of the DBCs is reduced, and the right pointer and left pointer DBCs do not need to store leaf nodes which can result in a considerable reduction in the memory footprint of the DTs (of $\approx 33\%$). From the programming perspective, only few changes are required to access the decomposed organization during inference. In the unified organization, every tree node is stored as one object in an array, thus access to the three node elements require an access at the corresponding array index and the according offset within the object. For the decomposed organization, the three node components are stored as three different objects in three arrays. Thus, the array index for the current node stays the same, but instead of accessing different offsets within one object, accesses for the same index in different arrays need to be performed. This induces minor changes of the decision tree code.

Although the proposed decomposition can be realized straightforwardly, it yields a different optimization objective. The decision tree inference causes a different cost in the decomposed structure. Eventually, an optimal placement for a unified decision tree may not be optimal for its corresponding decomposed tree. Therefore, we need to revisit the upper bound of our proposed BLO algorithm, respecting the modified structure of an optimal placement. In order

to formalize the decomposition, we introduce following notation:

$C_{down}^{decomp}/C_{up}^{decomp}/$ C_{total}^{decomp}	denotes the cost for an unconstrained arbitrary placement I to traverse the tree in decomposed DBCs.
$C_{down}^{*decomp}/C_{up}^{*decomp}/$ C_{*total}^{decomp}	denotes the cost in decomposed DBCs for an optimal placement $I^{*decomp}$, which optimizes $C_{total}^{*decomp}$.
$\overleftarrow{C}_{down}^{*decomp}/\overleftarrow{C}_{up}^{*decomp}/$ $\overleftarrow{C}_{total}^{*decomp}$	denotes the cost for an optimal placement $\overleftarrow{I}^*$ with the root on the left most position, which is caused on decomposed DBCs and optimizes $\overleftarrow{C}_{down}^{*}$.

TABLE 2
Decomposition Notation

It should be noted here that we consider the cost as a number of shifts within the DBCs. A DBC shift in RTM is different from the bit shifts, which are dependent on the DBC width. We hereby count shifts for the unified organization scenario with the same weight as shifts for the decomposed organization scenario to make the cost definitions comparable and relate them. However, when it comes to the realization of the decomposed DBCs, every shift contributes $\frac{1}{3}$ to the bit shifts and energy consumption compared to a single shift in the unified DBC. Hence, if a placement results in $3\times$ the cost on decomposed DBCs as on unified DBCs, ultimately, the energy consumption penalty is roughly the same in both cases.

For the rest of this section, we first revisit the cost model for our decomposed approach and then define the objective. We subsequently analyze and adjust the upper bound on our BLO placement.

5.1 Revisited Cost Model

During inference of the decomposed tree, the split value always has to be checked first. Thus, the split value DBC has to be shifted to every node during inference and therefore features the same cost for traversing the tree down ($C_{split,down}^{decomp}$) and back to the root ($C_{split,up}^{decomp}$) as for the unified organization approach:

$$C_{split,down}^{decomp} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot |I(n_x) - I(P(n_x))| \tag{8}$$

$$C_{split,up}^{decomp} = \sum_{n_x \in N_l} absprob(n_x) \cdot |I(n_x) - I(n_0)| \tag{9}$$

For the right pointer and left pointer DBC, the decision to shift to a certain index depends on the previous decision on the split value. Indeed, only the right pointer DBC or the left pointer DBC needs to be shifted for any node, but not both. Constructing the cost for this requires additional definitions. In the following, we denote the left child of node n_x by $LC(n_x)$ and the right child $RC(n_x)$, respectively:

Definition 4. *We define $path(n_x, n_y) = \{n_{i_1}, n_{i_2}, ..., n_{i_m}\}$ as a part of a root leaf path where $n_{i_1} = n_x$ and $n_{i_m} = n_y$ and $P(n_{i_x}) = n_{i_{x-1}}$ or as the empty set if n_x is neither a direct, nor an indirect parent of n_y.*

Definition 5. *We define $isleft(n_x)$ for all nodes $n_x \in N \setminus \{n_0\}$ as 1 if $n_x = LC(P(n_x))$ and as 0 for all other cases. We symmetrically define $isright(n_x)$ for all nodes $n_x \in N \setminus \{n_0\}$ as 1 if $n_x = RC(P(n_x))$ and as 0 for all other cases.*

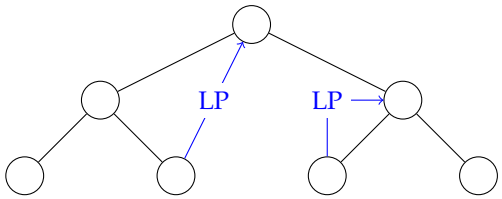


Fig. 7. Illustration of the left most parent of a node (2 exmaples)

Definition 6. *We define $LP(n_x)$ as the leftmost parent of node n_x for all nodes $n_x \in N \setminus \{n_0\}$:*
$\forall n_y \in path(LP(n_x), n_x) \setminus \{LP(n_x)\} : LC(n_y) \notin path(LP(n_x), n_x) \wedge LC(LP(n_x)) \in path(LP(n_x), n_x)$
If such a node does not exist, $LP(n_x) = \epsilon$. In other words, the leftmost parent is the closest node to n_x on its path from the root, where the left child is taken (illustrated in Figure 7).
We symmetrically define $RP(n_x)$ as the rightmost parent of node n_x for all nodes $n_x \in N \setminus \{n_0\}$:
$\forall n_y \in path(RP(n_x), n_x) \setminus \{RP(n_x)\} : RC(n_y) \notin path(RP(n_x), n_x) \wedge RC(RP(n_x)) \in path(RP(n_x), n_x)$
If such a node does not exist, $RP(n_x) = \epsilon$.

These definitions imply that for all nodes $n_y \in path(LP(n_x), n_x) \setminus \{LP(n_x)\}$ in between a node n_x and $LP(n_x)$, $isleft(n_y) = 0$. This also holds symmetrically for the RP definition. With the help of Definition 5 and Definition 6 we can investigate every node within the tree and compute the shifting distance in the left pointer and right pointer DBC if that specific node requires an inference of the right or left pointer DBC. This leads to the cost for traversing the right and left pointer DBC down:

$$C_{lptr,down}^{decomp} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot isleft(n_x) \cdot |I(P(n_x)) - I(LP(P(n_x)))| \tag{10}$$

$$C_{rptr,down}^{decomp} = \sum_{n_x \in N \setminus \{n_0\}} absprob(n_x) \cdot isright(n_x) \cdot |I(P(n_x)) - I(RP(P(n_x)))| \tag{11}$$

For simplicity, we denote that $|x, \epsilon| = 0$ for an arbitrary number x. The cost for going up the tree between two inferences is not necessarily the cost for shifting back to the root in the left pointer and right pointer DBC. Instead, there is a set of nodes, which are candidates to be accessed first in the right and left pointer DBCs, i.e. the nodes n_x where $LP(n_x) = \epsilon$ or $RP(n_x) = \epsilon$, respectively. Thus, for computing the estimated cost, all these candidates need to be considered with their respective absolute probabilities:

$$C_{lptr,up}^{decomp} = \sum_{n_x \in N_l} absprob(n_x) \cdot \sum_{n_r : LP(n_r) = \epsilon} absprob(n_r) \cdot prob(LC(n_r)) \cdot |I(n_r) - I(LP(n_x))| \tag{12}$$

$$C_{rptr,up}^{decomp} = \sum_{n_x \in N_l} absprob(n_x) \cdot \sum_{n_r : RP(n_r) = \epsilon} absprob(n_r) \cdot prob(RC(n_r)) \cdot |I(n_r) - I(RP(n_x))| \tag{13}$$

Combining these partial costs, the total cost can be deduced by adding all components:

$$C_{down}^{decomp} = C_{split,down}^{decomp} + C_{lptr,down}^{decomp} + C_{rptr,down}^{decomp} \tag{14}$$

$$C_{up}^{decomp} = C_{split,up}^{decomp} + C_{lptr,up}^{decomp} + C_{rptr,up}^{decomp} \tag{15}$$

$$C_{total}^{decomp} = C_{down}^{decomp} + C_{up}^{decomp} \tag{16}$$

5.2 Towards Optimal Decomposition

Due to the revisited cost model, the considerations about an optimal decision tree placement to the decomposed DBCs also need to be revisited. This section conducts a proof about the relation of the placement solution produced by the OLO algorithm to the optimal solution.

Throughout this section, we clarify the relation between placements for the unified organization approach, the cost they cause on the decomposed organization, and how a placement for unified DBCs can be constructed from a placement for decomposed DBCs. First, we have to clarify the relation between the cost C_{total} for an arbitrary placement I on a unified DBC and the cost C_{total}^{decomp} the exact placement causes on decomposed DBCs. Intuitively, the cost for the unified DBC can be seen as the cost for the DBC containing the split and feature values since this DBC has to access every node. In the following, a restructuring of the cost model is considered:

Lemma 5.

$$C_{split,down}^{decomp} = \sum_{n_l \in N_l} absprob(n_l) \cdot \sum_{n_x \in rlpath(n_l) \setminus \{n_0\}} |I(n_x) - I(P(n_x))| \tag{17}$$

$$C_{lptr,down}^{decomp} = \sum_{n_l \in N_l} absprob(n_l) \cdot \sum_{n_x \in rlpath(n_l) \setminus \{n_0\}} isleft(n_x) \cdot |I(P(n_x)) - I(LP(P(n_x)))| \tag{18}$$

$$C_{rptr,down}^{decomp} = \sum_{n_l \in N_l} absprob(n_l) \cdot \sum_{n_x \in rlpath(n_l) \setminus \{n_0\}} isright(n_x) \cdot |I(P(n_x)) - I(RP(P(n_x)))| \tag{19}$$

The cost for traversing the tree down in decomposed DBCs can be restructured as a per path cost, which is weighted with the absolute probability of the leaf node on this root leaf path.

Proof. From the definition of the tree structure, we know that probabilities are entirely inherited. Thus, summing up the absolute probabilities of all leaf nodes underneath a certain node n_x must result in the absolute probability of this node: $absprob(n_x) = \sum_{n_l \in leafs(n_x)} absprob(n_l)$. In Eq. (17), each distance between each node and the parent is weighted with exactly this sum of absolute probabilities of underlying leafs, since for every leaf the entire root leaf path is considered. Consequently, Eq. (17) can be rewritten to Eq. (8). The same principle can be applied to Eq. (18) (transofrms to Eq. (10)) and to Eq. (19) (transforms to Eq. (11)). □

Lemma 6.

$$C_{lptr,down}^{decomp} \leq C_{split,down}^{decomp} = C_{down} \tag{20}$$

$$C_{rptr,down}^{decomp} \leq C_{split,down}^{decomp} = C_{down} \tag{21}$$

The summed cost for shifting down in decomposed DBCs in the left and right pointer tree is smaller than the cost for shifting down in the split value DBC, which is equal to the cost for shifting down in the unified DBC case.

Proof. The full proof can be found in the appendix. Basically, the left and right pointer DBCs visit a subset of nodes from the split DBC, thus there cannot be more shifts. □

Lemma 7.

$$C_{down}^{decmp} \leq C_{total}^{decomp} \tag{22}$$

The cost for traversing the tree down in a decomposed placement is a part of the total shifting cost (compare to Lemma 1).

Proof. C_{total}^{decomp} is the sum of C_{down}^{decomp} and C_{up}^{decomp}, where C_{up}^{decomp} itself is a sum of non-negative terms. □

Lemma 8.

$$C_{down} \leq C_{down}^{decomp} \tag{23}$$

The summed cost for shifting through the decomposed DBCs while traversing the tree downwards is at at least the cost of shifting through a tree on a unified DBC downwards with the same placement.

Proof. From the definition of the cost function, we know that $C_{down} = C_{split,down}^{decomp}$. We further know that $C_{rptr,down}^{decomp}$ and $C_{lptr,down}^{decomp}$ only consists of a sum of terms which are either 0 or positive. According to Eq. (14), C_{down}^{decomp} is the sum of only these 3 components. Thus, $C_{down} = C_{split,down}^{decomp} \leq C_{down}^{decomp}$. □

Next, we need to consider the cost relation of a linear allowable placement produced by OLO. As reported by Adolphson and Hu, there is always a linear allowable placement, which features the optimal cost C_{down} under the constraint that the root is placed to the leftmost position [6]. Thus, we denote the cost of such an optimal linear allowable placement in the following by $\overleftarrow{C}^{*\cdots}_{\cdots}$.

Lemma 9.

$$\overleftarrow{C}^{*decomp}_{lptr,up} \leq \overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{down} \tag{24}$$

$$\overleftarrow{C}^{*decomp}_{rptr,up} \leq \overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{down} \tag{25}$$

The cost for shifting up in the left and right pointer DBCs in a linear allowable placement can be upper bounded by the cost for shifting up in the split value DBC, which is the same cost as shifting down in the unified DBC case.

Proof. This proof can be found in the appendix. The considerations are similar to Lemma 6. □

Corollary 2.

$$\overleftarrow{C}^{*decomp}_{total} \leq 6 \cdot \overleftarrow{C}^{*}_{down} = 3 \cdot \overleftarrow{C}^{*}_{total} \tag{26}$$

If a linear allowable placement is deployed to decomposed DBCs, the total cost for shifting through the decomposed DBCs is at most $6\times$ the cost of shifting the unified DBC downwards.

Proof. Eq. (26) follows from the definition of the cost model (Eq. (16)) and Lemma 9, Lemma 6 and Lemma 3: $\overleftarrow{C}^{*decomp}_{lptr,down} \leq \overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{rptr,up} \leq \overleftarrow{C}^{*}_{up}$, $\overleftarrow{C}^{*decomp}_{split,down} = \overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{lptr,up} \leq \overleftarrow{C}^{*}_{up} = \overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{rptr,down} \leq \overleftarrow{C}^{*}_{up} =$

$\overleftarrow{C}^{*}_{down}$, $\overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{up} = \overleftarrow{C}^{*}_{down}$. In total, $\overleftarrow{C}^{(}_{total} * decomp)$ consists of 6 terms, which are all upper bounded by $\overleftarrow{C}^{*}_{down}$. Lemma 3 further leads to $\overleftarrow{C}^{*}_{total} = 2 \cdot \overleftarrow{C}^{*}_{down}$. □

Combining the above considerations, we can construct the according upper bound.

Theorem 2.

$$\overleftarrow{C}_{down} \leq 2 \cdot C^{decomp}_{total} \tag{27}$$

Any placement for decomposed trees can be transformed into a placement with the root on the left most position, where the cost for traversing the tree downwards in a unified DBC is at most 2× the cost for traversing the entire tree on decomposed DBCs.

$$\overleftarrow{C}^{*decomp}_{total} \leq 12 \cdot C^{*decomp}_{total} \tag{28}$$

An optimal linear allowable placement for shifting downwards in a unified DBC, as obtained by OLO, is an upper bound of 12 of the optimal placement for decomposed DBCs.

Proof. Eq. (27) directly follows from Lemma 7,Lemma 8 and Lemma 4.

Eq. (28) can be proven by contradiction. Suppose that the optimal linear allowable placement for a unified DBC $\overleftarrow{C}^{*}_{down}$ would cause a cost $\overleftarrow{C}^{*decomp}_{total}$ larger than 12× of the optimal placement for decomposed DBCs $C^{*decomp}_{total}$. According to Corollary 2, we know that the optimal placement must have at least a cost of $\frac{1}{6}$ on the unified DBC then, thus $\overleftarrow{C}^{*}_{down} > 12 \cdot \frac{1}{6} \cdot C^{*decomp}_{total} \Leftrightarrow \overleftarrow{C}^{*}_{down} > 2 \cdot C^{*decomp}_{total}$. We further know that according to Eq. (27) we can build a solution for the unified DBC with a cost less than $2 \cdot C^{*decomp}_{total}$, which contradicts the optimality of $\overleftarrow{C}^{*}_{down}$. □

5.3 Towards Bidirectional Linear Optimization

The BLO heuristic (Section 4) can be applied to the decomposed organization scenario without any limitation. The consideration that the BLO extension does not introduce additional shifting cost, however, does not remain valid for this scenario. Potentially, the left or right pointer DBC can be shifted from a certain node within the right subtree to another node within the left subtree, without loading the root and vice versa. Thus, both nodes may be placed closer in the OLO placement as in the BLO placement. However, the proof upper bounds the cost for going up and down in the left and right pointer DBCs with the cost for the split value DBC, i.e. with the cost of starting at the root and ending at a leaf in Lemma 9. Theorem 2 consequently takes this bound in to determine the ultimate upper bound. Hence, under this worst-case scenario, upper bound of 12× is valid for BLO and OLO.

6 Evaluation

In addition to the proven upper bound of our BLO algorithm on unified and decomposed organisation, this section presents experimental evaluation of the BLO algorithm and provides a comparison to the state-of-the-art. The proven upper bounds for BLO consequently hold for the state-of-the-art methods, since these cannot achieve better performance than the optimum. The relation between these approach in realistic scenarios, however, is empirically studied in this section. We first discuss the shifts reduction of different solutions and then show the impact of shifts reduction on the runtime and energy consumption.

6.1 Experimental Setup

In order to compare our Bidirectional Linear Ordering (BLO) approach to the state-of-the-art (i.e., ShiftsReduce [2] and Chen et al. [3]) on unified and decomposed organization, we adopt an open-source framework published in [12] and select eight typical machine learning classification datasets from the UCI Machine Learning Repository [13] and [14]: adult, bank, magic, mnist, satlog, sensorless-drive, spambase, and wine-quality. For each dataset, we use 75% of the data for training and 25% for testing. We train decision trees by using tree classifiers in the `sklearn` package [15]. We run the default configuration of sklearn, without tuning hyper-parameters.

To derive differently sized trees, we specify the maximum depth of the trees, e.g., DT1 means that the tree has a maximum depth of 1, thus two levels, and DT3 means that the tree has four levels. After the trees are generated, we profile the node probabilities on the training data by counting how often each node's left child or the right child is visited. This delivers us empirical branch probabilities and absolute node access probabilities. For further evaluation, we simulate the execution of the decision trees by generating a code implementation, which produces a trace of visited nodes during the data inference. We infer the data points from the test data on the trees and generate a node access trace, which provides the node access paths on a logic level. Subsequently, we place the trees to RTM with different layouts and compute the required amount of shifts by considering the node access trace and the node mapping. Based on the amount of shifts, wen can also compute the latency and energy consumption. Concretely, we compare the following.

- *Naive / NaiveD:* A baseline breadth-first order placement in which indices are assigned to tree nodes layer-wise in increasing order. The placement is used for the unified (Naive) and decomposed organization (NaiveD).
- *ShiftsReduce / ShiftsReduceD:* The state-of-the-art data placement algorithm from [2]. We evaluate the heuristic on the unified organization (ShiftsReduce) and the decomposed organization (ShiftsReduceD).
- *Chen / ChenD.:* The data placement algorithm from [3], evaluated on the unified organization (Chen) and the decomposed organization (ChenD).
- *BLO / BLOD:* Our proposed bidirectional linear ordering solution for unified trees. It is evaluated on the unified and decomposed organization.
- *MIP / MIPD:* The mixed integer programming formulation of the cost model (Eq. (4) for unified organization and Eq. (16) for decomposed organization). The solver, in case it converges, returns the optimal tree placement.

We replay the node access trace for all configurations to derive the total amount of required racetrack shifts. For the decomposed trees, the performance and energy numbers reported in this section consider all, i.e., the split value and pointers DBCs. Although the number of RTM shifts

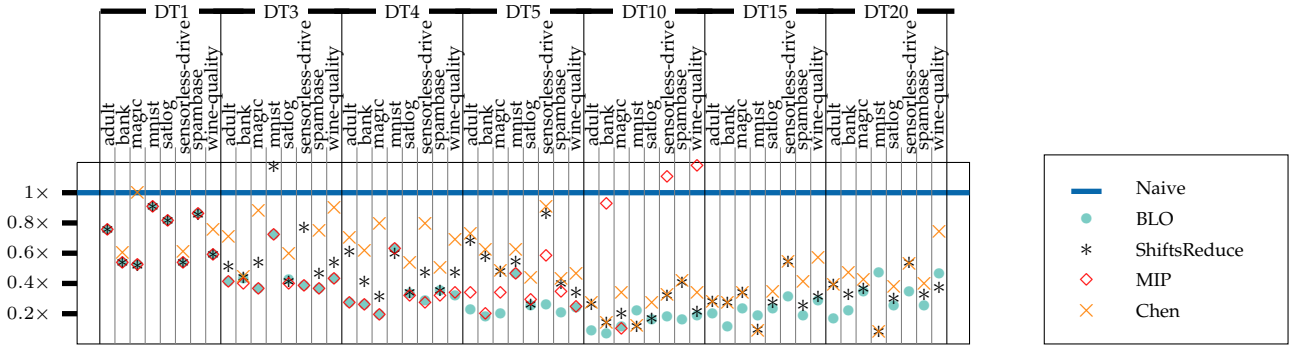


Fig. 8. Comparison of Total Shifts During Inference

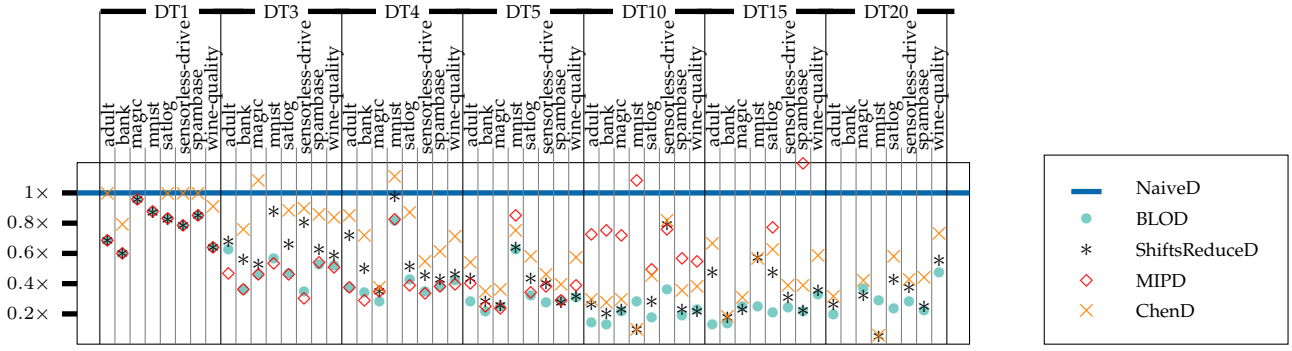


Fig. 9. Comparison of Total Shifts During Inference on Decomposed Trees

already allows a quantitative comparison of the different placement approaches, we further compute the energy consumption and total runtime on a realistic model derived from the various memory placements. For the runtime, we use the per-access and per-shift latencies in Table 3 and compute the overall runtime. Given the amount of RTM accesses $n_{accesses}$ and the total amount of shifts in between n_{shifts}, the total runtime for the unified organization is $runtime = \ell_R \cdot n_{accesses} + \ell_S \cdot n_{shifts}$. In the case of decomposed trees, since the DBCs are not moved synchronously, the total runtime also includes the penalty to align pointer DBCs. The total energy consumption is derived from read and shift dependent dynamic energy consumption and from the runtime dependent static energy consumption (leakage): $energy = e_R \cdot n_{accesses} + e_S \cdot n_{shifts} + p \cdot runtime$, where the parameters can be found in Table 3.

Ports per track, domains per track		1, 64
Tracks per DBC: unified, decomposed		96, 32
Leakage power [mW]: unified, decomposed	p	36.2, 36.9
Write energy [pJ]: unified, decomposed	e_W	106.8, 40.7
Read energy [pJ]: unified, decomposed	e_R	62.8, 23.4
Shift energy [pJ]: unified, decomposed	e_S	51.8, 17.3
Write latency [ns]: unified, decomposed	l_W	1.79, 1.75
Read latency [ns]: unified, decomposed	l_R	1.35, 1.32
Shift latency [ns]: unified, decomposed	l_S	1.42, 1.39

TABLE 3
RTM parameters values for a 128 KiB SPM

As previously mentioned, we only investigate the racetrack shifts caused when inferring data points on the decision trees. Since we assume that the decision trees are mapped to an isolated scratchpad memory for our target system, the memory accesses to the decision trees are not disrupted by any operating system interaction. However, the total energy consumption and latency still strongly depend on concurrent applications and the underlying system software. This could be investigated by further full-system simulation, which is out of the scope of this paper.

6.2 RTM Shifts Analysis

Figure 8 and Figure 9 compare the total amount of RTM shifts for different placements for the unified and decomposed DBCs, respectively. All results are normalized to the *naive* placement. The MIP formulation is implemented in the Gurobi optimizer [16] and is given a time limit of 8 hours per dataset and per tree configuration. For the DT1 and DT3 instances in all datasets, the MIP converges to the optimal solution. In all other cases, the results are based on the Gurobi heuristic. Results which are worse than $1.2\times$ of the baseline are not illustrated in the figures.

A detailed analysis of the results shows that for the cases where the MIP and MIPD finds an optimal placement (for DT1 and DT3), BLO and BLOD achieves the same or only marginally worse results than the optimum. This supports the heuristic design principle of BLO (Section 4). Compared to state-of-the-art solutions, it can be observed that BLO and BLOD achieve the best reduction in shifts for most of the investigated cases. This supports the design concept of a domain specific placement approach, which can achieve better results by assuming a simpler structure. Considering the geometric mean (geomean) improvement over all evaluated datasets and trees, BLO reduces the amount of bit shifts by 58.1% compared to the naive placement (see Figure 8). ShiftsReduce reduces them by 50.8%. BLO thus further reduces the amount of necessary bit shifts by 14.3% upon ShiftsReduce.

In the decomposed trees (BLOD), the absolute number of RTM bit shifts compared to the unified trees reduces (BLO) by an average of 37.6%. However, for the same unified naive

placement baseline, BLOD reduces the amount of RTM bit shifts by a geomean 80.1%, compared to 58.13% by BLO (see Figure 9). Compared to the MIPD solution, BLOD performs slightly better than the unified BLO placement in terms of RTM shifts. The ShiftsReduceD and ChenD solutions report comparable improvement for the decomposed and the unified trees. Note that the placement decisions in all heuristics are based on the training dataset while they are evaluated on the test dataset.

The reduction of the total shifts does not directly imply a similar improvement in runtime and energy consumption. To estimate the shifts reduction impact on the runtime and energy consumption, we consider a realistic setup as explained in Section 2.3. Larger decision trees are first split into smaller trees, and the placement heuristic is then executed on multiple trees of maximal depth of 5. Note that the assignment of these smaller trees to different DBCs may affect the cost of the overall shift. Techniques such as [17] can be applied to distribute tree nodes to different DBCs intelligently, but this is beyond the scope of this work. For the runtime and energy consumption results, we use decision trees up to DT5 and present the results in Section 6.4.

6.3 Unified vs. Decomposed DTs

Although the previous results report the performance of the BLO and BLOD algorithm on the unified and decomposed trees, the question which of both realizations should be used for a concrete system remains open. Eq. (26) implies that any linear allowable placement cannot cause more than $3\times$ shifts on the decomposed DBCs as on the unified DBCs. Under the ideal assumption that each single DBC in the decomposed setup only needs $\frac{1}{3}$ of bit-lines and therefore also only yields $\frac{1}{3}$ of the energy consumption, the decomposed setup cannot be worse than the unified setup in no scenario. In reality, however, constructing the decomposed setup may create additional static overheads or consume additional resources (such as chip space or leakage power), which is only desirable if the decomposed setup can significantly reduce the resource consumption.

In order to assess the resource savings when considering the decomposed setup, we take the placement of all configurations and replay the node access traces on the unified and decomposed organizations. We compute the relation of the total amount of shifts for all configurations in the unified and decomposed approaches. Theoretically, the ratio between the unified shifts and the decomposed shifts must range between $1\times$ and $3\times$. We evaluate this and show the ratios based on experimental results in Figure 10. For trees with a maximum depth of 1 i.e., DT1, the decomposed and unified approaches result in exactly the same amount of shifts in all placements. This is because a DT1 has 2 levels, thus only a single node with pointers which is mapped to the first location in a single DBC (unified) or multiple DBCs (decomposed). Therefore, no shifts in the right and left pointer DBC are required. Note that we assume that the access ports in all DBCs are initially aligned to the first position. For deeper trees, the increase in shifts ratio shows similar trend for all placement approaches. For the deepest trees considered in this evaluation, the number of shifts in the decomposed trees can be as high as 2.59 for the BLO algorithm.

In the decomposed organization, the highest shift reduction is expected from scenarios where the pointer DBCs are rarely shifted. For DT1, the best case is achieved because the left and right pointer DBCs do not need to be shifted at all. As the trees get deeper, the probability of frequently accessing left and right pointers also increases. Thus, for deeper trees the shifts reduction in the decomposed setup is reduced, which can be seen in the reported results as well.

However, focusing on the realistic tree sizes of at most 3 or 4 layers, which can be placed into a single DBC, the experimental data suggests that the amount of shifts is increased by at most a factor of $2\times$ when switching to the decomposed setup. This is a considerable margin to leverage static overheads from the the decomposition and provide a reduction in the total resource consumption.

6.4 Runtime and Energy

BLO reduces the total runtime by 53.8% compared to the naive placement, as shown in Figure 11. In comparison, for the same baseline, ShiftsReduce and BLOD reduce the total runtime by 45.7% and 46.3%, which are 13.3% and 13.9% longer compared to the BLO, respectively. Comparing this to the reduction of shifts for trees with maximum depth of 5 only, BLOD reduces the required shifts by 85.1%, BLO by 77.5%, and ShiftsReduce by 72.4%. Thus BLOD, compared to BLO and ShiftsReduce, further reduces the amount by shifts by 9.8% and 17.5% respectively. This suggests that a reduction in shifts may not necessarily result in the runtime reduction, or at least not with the same proportion. When comparing Figure 8 and Figure 9 to Figure 11 and Figure 12, please note the different scaling on the y axis and that results are averaged across datasets for the latter figures.

In the decomposed placement approach, the total runtime increases due to the alignment time in the pointers DBCs. The split value DBC is checked first to determine whether a pointer DBC needs to be accessed or not. Subsequently, depending on the node access probabilities, a shift request may be sent to the left or the right pointer DBC. The lazy shift approach in pointers DBCs improves the overall shift energy due to the reduced amount of shifts. However, this negatively impacts the runtime due to the shift penalty required to align the access port to the desired location if it is not aligned with the split value DBC. To quantify the impact of the decomposed approach on the runtime, we compare it with other methods, as presented in Figure 11. For the same baseline (naiveD), BLOD has an average runtime overhead of 7.5% compared to BLO. Consequently, BLOD also increases the leakage energy compared to BLO However, this deterioration in the leakage energy is offset by the reduction both in the shift and access component of the energy (cf. Figure 13). Similarly, other decomposed approaches (e.g., naiveD, MLPD) induces a runtime penalty compared to their unified counterparts (e.g., naive, MLP).

BLOD achieves the most reduction in energy consumption compared to all other approaches. This is because the total energy consumption of RTM is largely dependent upon the number of bit shifts, which affect the shift energy and the runtime, which determine the leakage energy. Figure 12 and Figure 13 show the overall energy consumption and the energy breakdown of different placement approaches

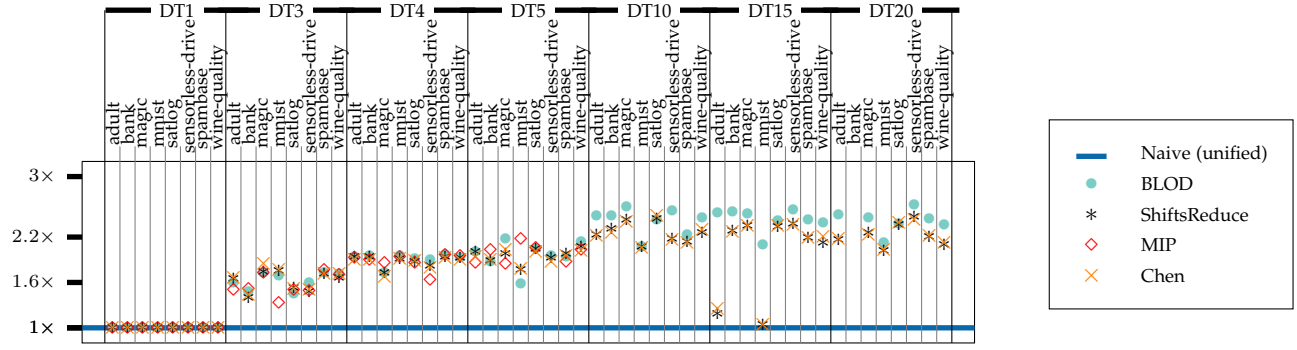


Fig. 10. Increase of Total Shifts During Inference between Unified and Decomposed Trees

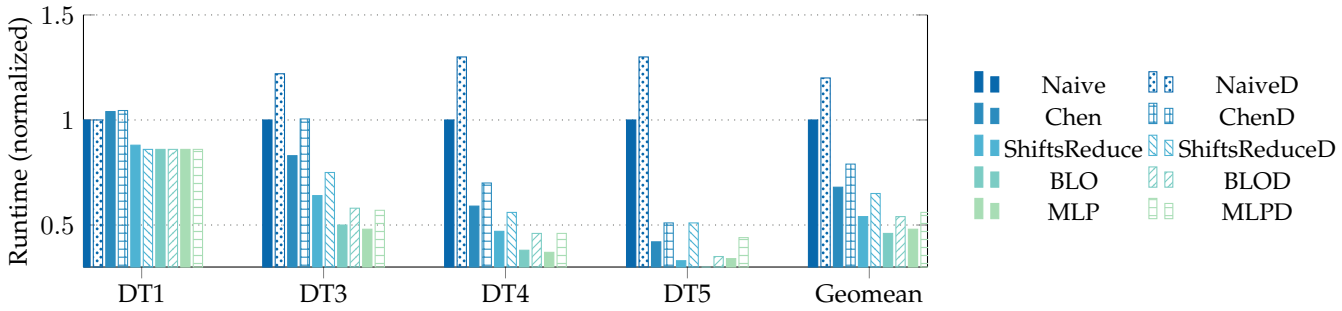


Fig. 11. Runtime of different configurations for different tree size. The results are average across all benchmarks.

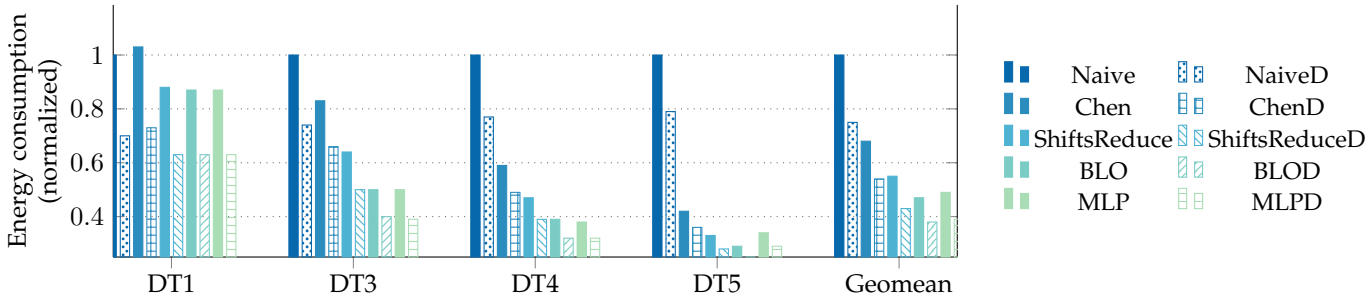


Fig. 12. Energy consumption of different configurations for different tree size. The results are average across all benchmarks.

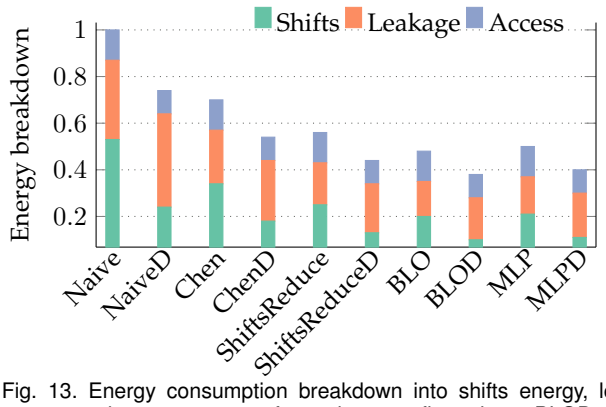


Fig. 13. Energy consumption breakdown into shifts energy, leakage energy and access energy for various configurations. BLOD records the lowest shift and total energy consumption compared to all other configurations.

for the unified and the decomposed DBCs normalized to the naive placement. Compared to the naive solution, BLOD delivers a 61.7% reduction in the RTM energy consumption, compared to 52.6% in BLO and 45.8% in ShiftsReduce for the same baseline.

Figure 13 highlights that the energy efficiency of BLOD compared to existing unified approaches is achieved via a significant reduction in the energy consumed by the shift operation and a slight reduction in the access energy. The leakage energy, compared to the naive solution (NaiveD), is also reduced by 44.7%. The improvement in the shift energy is due to reduced shift cost, while the reason for the leakage energy saving is the reduced runtime (cf. Figure 11). Compared to the unified BLO solution, despite an increase in the leakage energy by 16.2%, the decomposed approach consumes 17.3% less energy. Overall, for the naive baseline (Naive), BLOD on average achieves (95.3%, 35%, 21.5%, 17.3%, 150%, 1.7%) more energy reduction compared to (Chen, ShiftsReduce, MLP, BLO, naiveD, MLPD).

7 Related Work

A rich body of research has explored the efficient employment of RTM at various levels in the memory hierarchy for numerous application domains and system setups. In this context, optimization techniques for RTM have been proposed to facilitate their adoption in the register file [18]–[20], scratchpads [2], [21], [22], caches [23]–[30], network-on-chip [31], off-chip memory [32], and solid state drives [33]. Therefore, RTM can be fitted in all levels of the memory hierarchy, making it a promising candidate for universal memory.

To provide performance, area, and energy benefits, various optimizations have been proposed in the literature at

cell-level [28], circuit-level [29], layout-level [27], [30], [34], and cross-level [35]. RTM's leakage power and capacity advantages give it a competitive edge over existing memory technologies, but the expensive shift operations present a daunting challenge. In this context, various techniques for RTM shift cost reduction have been proposed, such as runtime data swapping [25], [28], [36], data compression [26], [37], preshifting [18], [38], access port management [24], [25], [28], intelligent instruction [39], and data placement [2], [3]. For data placement, Chen et al. in [3] present a heuristic appending data objects according to the adjacency information sequentially. Khan et al. in [2] formulate the data placement problem with an integer linear programming and further propose ShiftsReduce heuristic to enhance the previous heuristic by introducing a tie-breaking scheme and a two-directional objects grouping mechanism assuming a single access port RTMs. Whereas the above techniques are generalized solutions, this work considers the data objects of decision trees where the dependencies between tree nodes strictly limit possible access patterns.

Recently, it has been shown that domain-specific approaches not only guarantee better performance and energy consumption but also enable better predictability of the runtime [21]. In fact, the studied problem can be treated as an instance of the quadratic assignment problem (QAP), which was introduced in 1957 [40], considering the problem of allocating a set of facilities to a set of locations. When the facilities are all in a line (like the locations within in a DBC), such a special case is named the *linear ordering/arrangement* problem [7]. Suppose that the number of vertices is m and the length of an edge is defined as the linear distance between the vertices involved. Specifically, for tree graphs, the common objective is to minimize the sum of edge lengths as the total shift cost in this work. For undirected trees, Shiloach proposes an $\mathcal{O}(m^{2.2})$ algorithm [41]. For directed trees, Adolphson and Hu in [6] present an algorithm to derive an optimal placement in $\mathcal{O}(m \log m)$. For the studied problem of this work, Adolphson and Hu's algorithm is no longer optimal since the additional distance induced by shifting back a nanowire from leaves to the root between two inferences needs to be considered.

The imperfection in the fabrication technologies and fluctuation in the current density required for the shift operation may cause pinning faults and position errors in RTMs. Of late, many position error detection and correction schemes have been proposed to guard RTMs against such errors and improve their reliability [11], [42], [43]. This work focuses on reducing the shift operations in RTMs, which indirectly reduces the probability of position error but does not explicitly consider this aspect.

8 Conclusion

In this paper, we present BLO, a domain-specific placement heuristic for decision trees on RTM. BLO exploits the knowledge of the internal structure of decision trees and the profiled probabilities for nodes being accessed, which are gathered on a previously known dataset. BLO bases on an optimal algorithm to solve the OLO problem for rooted trees [6] and eliminates the main reason for improper placements on RTM. We introduce two different approaches to organization decision trees on racetrack memory. The decomposed organization decouples the storage of decision tree nodes and allows optimization regarding memory space consumption.

BLO causes at most $4\times$ of the RTM shifts than the optimal placement on the unified organization. The upper bound is proven to be $12\times$ on the decomposed organization approach. Our empirical evaluations show that BLOD delivers the best bit shifts reduction for the most realistic use-case of decision trees (depth 5) (geomean of 80%). In terms of runtime, BLO compared to BLOD performs better due to the longer stalls in BLOD pointers' DBCs. In terms of energy consumption, BLOD outperforms all other configurations.

Acknowledgement

This paper has been supported by Deutsche Forschungsgemeinshaft (DFG), as part of the project OneMemory (405422836), SFB876 A1 (124020371), DART-HMS (437232907), TraceSymm (366764507) and CO4RTM (450944241).

References

[1] R. Bläsing, A. A. Khan, P. C. Filippou, C. Garg, F. Hameed, J. Castrillón, and S. S. P. Parkin, "Magnetic racetrack memory: From physics to the cusp of applications within a decade," *Proceedings of the IEEE*, vol. 108, 2020.

[2] A. A. Khan, F. Hameed, R. Bläsing, S. S. P. Parkin, and J. Castrillon, "Shiftsreduce: Minimizing shifts in racetrack memory 4.0," *ACM Trans. Archit. Code Optim.*, vol. 16, Dec. 2019.

[3] X. Chen, E. H.-M. Sha, Q. Zhuge, C. J. Xue, W. Jiang, and Y. Wang, "Efficient Data Placement for Improving Data Access Performance on Domain-Wall Memory," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 24, Oct. 2016.

[4] C. Hakert, A. A. Khan, K.-H. Chen, F. Hameed, J. Castrillon, and J.-J. Chen, "BLOwing trees to the ground: Layout optimization of decision trees on racetrack memory," in *Proceedings of the 58th Annual Design Automation Conference (DAC'21)*. ACM, Jul. 2021.

[5] S. Buschjäger and K. Morik, "Decision tree and random forest implementations for fast filtering of sensor data," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, 2018.

[6] D. Adolphson and T. C. Hu, "Optimal linear ordering," *SIAM Journal on Applied Mathematics*, vol. 25, 1973.

[7] R. E. Burkard, E. Çela, P. M. Pardalos, and L. S. Pitsoulis, *The Quadratic Assignment Problem*, 1998, pp. 1713–1809.

[8] J. Díaz, J. Petit, and M. Serna, "A survey of graph layout problems," *ACM Comput. Surv.*, vol. 34, Sep. 2002.

[9] M. R. Garey and D. S. Johnson, *Computers and intractability*.

[10] K. Skodinis, "Computing optimal linear layouts of trees in linear time," in *European Symposium on Algorithms*. Springer, 2000.

[11] S. Ollivier, D. K. Jr., K. A. Roxy, R. G. Melhem, S. Bhanja, and A. K. Jones, "Leveraging transverse reads to correct alignment faults in domain wall memories," in *DSN*. IEEE, 2019.

[12] S. Buschjäger, K.-H. Chen, J.-J. Chen, and K. Morik, "Realization of random forest for real-time evaluation through tree framing," in *IEEE International Conference on Data Mining (ICDM)*, 2018.

[13] M. Lichman, "UCI machine learning repository," 2013.

[14] Y. LeCun, "The mnist database of handwritten digits," 1998.

[15] 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, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, 2011.

[16] B. Bixby, "The gurobi optimizer," *Transp. Re-search Part B*, 2007.

[17] A. A. Khan, A. Goens, F. Hameed, and J. Castrillon, "Generalized data placement strategies for racetrack memories," in *Design, Automation and Test in Europe Conference (DATE)*, 2020.

[18] E. Atoofian, "Reducing Shift Penalty in Domain Wall Memory Through Register Locality," in *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '15, 2015.

[19] M. Moeng, H. Xu, R. Melhem, and A. Jones, "ContextPreRF: Enhancing the Performance and Energy of GPUs With Nonuniform Register Access," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, 2016.

[20] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li, "Exploration of GPGPU Register File Architecture Using Domain-wall-shift-write Based Racetrack Memory," in *Proceedings of the 51st Annual Design Automation Conference on Design Automation Conference*, 2014.

[21] A. A. Khan, N. A. Rink, F. Hameed, and J. Castrillon, "Optimizing tensor contractions for embedded devices with racetrack memory scratch-pads," in *International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019.

[22] H. Mao, C. Zhang, G. Sun, and J. Shu, "Exploring Data Placement in Racetrack Memory Based Scratchpad Memory," in *2015 IEEE Non-Volatile Memory System and Applications Symposium*, Aug 2015.

[23] H. Xu, Y. Alkabani, R. Melhem, and A. K. Jones, "Fusedcache: A naturally inclusive, racetrack memory, dual-level private cache," *IEEE Trans. on Multi-Scale Computing Systems*, vol. 2, April 2016.

[24] R. Venkatesan, V. Kozhikkottu, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan, "TapeCache: A High Density, Energy Efficient Cache Based on Domain Wall Memory," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12, 2012.

[25] R. Venkatesan, V. J. Kozhikkottu, M. Sharad, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan, "Cache Design with Domain Wall Memory," *IEEE Trans. on Computers*, vol. 65, 2016.

[26] H. Xu, Y. Li, R. Melhem, and A. K. Jones, "Multilane Racetrack Caches: Improving Efficiency Through Compression and Independent Shifting," in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015.

[27] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao, "Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power," in *The 20th Asia and South Pacific Design Automation Conference*, Jan 2015.

[28] Z. Sun, X. Bi, W. Wu, S. Yoo, and H. . Li, "Array Organization and Data Management Exploration in Racetrack Memory," *IEEE Transactions on Computers*, vol. 65, April 2016.

[29] S. Motaman, A. Iyengar, and S. Ghosh, "Synergistic Circuit and System Design for Energy-efficient and Robust Domain Wall Caches," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '14, 2014.

[30] Z. Sun, X. Bi, A. K. Jones, and H. Li, "Design Exploration of Racetrack Lower-Level Caches," in *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2014.

[31] D. Kline, H. Xu, R. Melhem, and A. K. Jones, "Domain-wall memory buffer for low-energy nocs," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[32] Q. Hu, G. Sun, J. Shu, and C. Zhang, "Exploring Main Memory Design Based on Racetrack Memory Technology," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, May 2016.

[33] E. Park, S. Yoo, S. Lee, and H. Li, "Accelerating Graph Computation with Racetrack Memory and Pointer-assisted Graph Representation," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014.

[34] S. Motaman, A. S. Iyengar, and S. Ghosh, "Domain wall memory-layout, circuit and synergistic systems," *IEEE Transactions on Nanotechnology*, vol. 14, 2015.

[35] G. Sun, c Zhang, H. Li, Y. Zhang, W. Zhang, Y. Gu, Y. Sun, J. Klein, D. Ravelosona, Y. Liu, W. Zhao, and H. Yang, "From Device to System: Cross-Layer Design Exploration of Racetrack Memory," in *DATE*. ACM, 2015.

[36] Z. Sun, Wenqing Wu, and Hai Li, "Cross-layer racetrack memory design for ultra high density and low power consumption," in *Design Automation Conference (DAC)*, 2013.

[37] A. Ranjan, S. G. Ramasubramanian, R. Venkatesan, V. Pai, K. Roy, and A. Raghunathan, "Dyrectape: A dynamically reconfigurable cache using domain wall memory tapes," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.

[38] A. Colaso, P. Prieto, P. Abad, J. A. Gregorio, and V. Puente, "Architecting Racetrack Memory Preshift through Pattern-Based Prediction Mechanisms," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[39] J. Multanen, P. Jääskeläinen, A. A. Khan, F. Hameed, and J. Castrillon, "Shrimp: Efficient instruction delivery with domain wall memory," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2019.

[40] T. C. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica*, vol. 25, 1957.

[41] Y. Shiloach, "A minimum linear arrangement algorithm for undirected trees," *SIAM Journal on Computing*, vol. 8, 1979.

[42] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, Y. Wang, and J. Shu, "Hi-fi playback: Tolerating position errors in shift operations of racetrack memory," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[43] A. Vahid, G. Mappouras, D. J. Sorin, and A. R. Calderbank, "Correcting two deletions and insertions in racetrack memory," *CoRR*, vol. abs/1701.06478, 2017. [Online]. Available: http://arxiv.org/abs/1701.06478

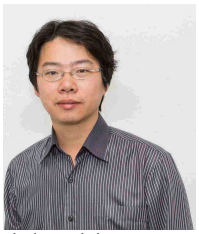
Christian Hakert is a research associate at TU Dortmund in the group of Design Automation for Embedded Systems with Prof. Jian-Jia Chen. He received his Master degree in Computer Science from TU Dortmund in 2019 and received the best student award "Jahrgangsbestenpreis" for his master degree in 2019. His research interest is the support and application of non-volatile main memories in system software and operating systems.

Asif Ali Khan is a researcher at the Chair for Compiler Construction in the Computer Science Department of the TU Dresden, Germany. His research interests include Computer architecture, heterogeneous memories, and compiler support for the memory system. Currently, Asif's research mainly focuses on exploring the emerging nonvolatile memory technologies in the memory subsystems and their optimizations for various metrics.

Kuan-Hsun Chen is a tenured assistant professor in the Department of Computer Science at University of Twente in the Netherlands. From Aug. 2019 to Aug. 2021, he was a postdoc at TU Dortmund University in Germany. He earned his Ph.D. (Dr.-Ing.) in Computer Science from TU Dortmund University with a distinction "Summa cum Laude" in 2019. He earned his master's degree in Computer Science from National Tsing Hua University (Taiwan) in 2013. His research interests include real-time embedded systems, architecture-aware software design, and dependable computing.

Fazal Hameed received his Ph.D. (Dr.-Ing.) degree in computer science from the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, in 2015. He joined the chair for Compiler Construction at the TU Dresden (Dresden, Germany) as Post-doctoral researcher in March 2016. Before, he worked on a similar position at the Chair of Dependable and Nano Computing (CDNC) Karlsruhe Institute of Technology (KIT), Germany. He is currently affiliated with Institute of Space Technology, Islamabad, Pakistan. He mainly works in the architecture group with a focus on memories. Mr. Hameed was a recipient of the CODES+ISSS 2013 Best Paper Nomination for his work on DRAM cache management in multicore systems. He has served as an External Reviewer for major conferences in embedded systems and computer architecture.

Jeronimo Castrillon is a professor in the Department of Computer Science at the TU Dresden, where he is also affiliated with the Center for Advancing Electronics Dresden (CfAED). He is the head of the Chair for Compiler Construction, with research focus on methodologies, languages, tools and algorithms for programming complex computing systems. He received the Electronics Engineering degree from the Pontificia Bolivariana University in Colombia in 2004, his masters degree from the ALaRI Institute in Switzerland in 2006 and his Ph.D. degree (Dr.-Ing.) with honors from the RWTH Aachen University in Germany in 2013. In 2014, Prof. Castrillon co-founded Silexica GmbH/Inc, a company that provides programming tools for embedded multicore architectures.



Jian-Jia Chen is Professor at Department of Informatics in TU Dortmund University in Germany. He was Juniorprofessor at Department of Informatics in Karlsruhe Institute of Technology (KIT) in Germany from May 2010 to March 2014. He received his Ph.D. degree from Department of Computer Science and Information Engineering, National Taiwan University, Taiwan in 2006. He received his B.S. degree from the Department of Chemistry at National Taiwan University 2001. Between Jan. 2008 and April 2010, he was a postdoc researcher at ETH Zurich, Switzerland. His research interests include real-time systems, embedded systems, energy-efficient scheduling, power-aware designs, temperature-aware scheduling, and distributed computing. He received the European Research Council (ERC) Consolidator Award in 2019. He has received more than 10 Best Paper Awards and Outstanding Paper Awards and has involved in Technical Committees in many international conferences.

ROLLED: Racetrack Memory Optimized Linear Layout and Efficient Decomposition of Decision Trees - **Appendix**

Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen,
Fazal Hameed, Jeronimo Castrillon and Jian-Jia Chen

Lemma 3. *If a placement I is unidirectional or bidirectional, $C_{down} = C_{up}$.*

Proof. As I is unidirectional or bidirectional, we know that a leaf node $n_x \in N_l$ is always the rightmost node or the leftmost node within its root leaf path $rlpath(n_x)$ if the path is monotonically increasing or decreasing, respectively. We further know that following the path from parents to their children must always be a movement monotonically to the right or monotonically to the left. Therefore we can follow that the distance from the root to a leaf node is equal to the sum of all distances on the path:

$$\forall n_y \in N_l : |I(n_y) - I(n_0)| = \sum_{n_z \in rlpath(n_y) \setminus n_0} |I(n_z) - I(P(n_z))| \tag{1}$$

This leads to:

$$C_{up} = \sum_{n_y \in N_l} \left(absprob(n_y) \cdot \sum_{n_z \in rlpath(n_y) \setminus \{n_0\}} |I(n_z) - I(P(n_z))| \right) \tag{2}$$

The summation is reorganized with respect to each node $n_x \in N$ by using the following observation: if n_z is in $rlpath(n_y)$, then n_y is in $leaves(n_z)$. That is, a node $n_x \in N$ contributes to Equation (2) exactly $|I(n_x) - I(P(n_x))| \cdot \sum_{n_y \in leaves(n_x)} absprob(n_y)$. Therefore,

$$C_{up} = \sum_{n_x \in N \setminus \{n_0\}} \left(|I(n_x) - I(P(n_x))| \cdot \sum_{n_y \in leaves(n_x)} absprob(n_y) \right) \tag{3}$$

Applying Definition 1 leads to Equation (4):

$$C_{up} = \sum_{n_x \in N \setminus \{n_0\}} (|I(n_x) - I(P(n_x))| \cdot absprob(n_x) = C_{down} \tag{4}$$

□

Lemma 4. *Any placement I can be converted into a placement $\overleftarrow{I}$ which places the root on the left most position by increasing the expected cost of $\overleftarrow{C}_{down}$ with at most a factor of 2:*

$$\overleftarrow{C}_{down} \leq 2 \cdot C_{down} \tag{5}$$

Proof. Suppose that the root of the decision tree is assigned at position r in the placement I. Due to space limitation, we present only the proof of the case that $m - r \geq r$, as the other case is symmetric. The placement is replaced as follows:

- reassign every node in position $r + i$ in I to $r + 2 \cdot i$ for $i = 1, 2, \ldots, r$.
- keep all nodes in position $r + i$ in I for $i = r + 1, r + 2, \ldots, m$ at their original position, which becomes $2 \cdot r + i$ relative to the root.
- reassign every node in position $r - i$ in I to $r + 2 \cdot i - 1$ for $i = 1, 2, \ldots, r$.

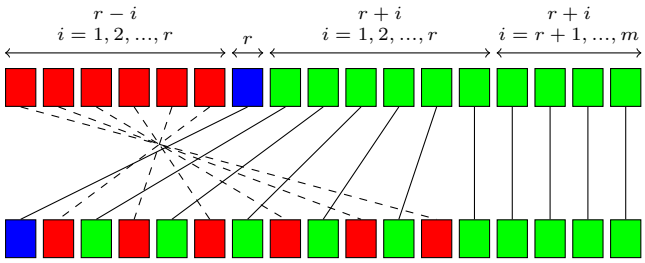


Figure 1: Reassignment of nodes and root to the left

Figure 1 gives a concrete example of the remapping of different nodes. After that, every node is then shifted by r positions towards the left, and the root is on the leftmost position, i.e., 0.

For notational brevity, we denote $P(n_x)$ as n_z for the rest of this proof. Due to the above reassignment, we have

$$\overleftarrow{I}(n_x) = \begin{cases} 2 \cdot (r - I(n_x)) - 1 & I(n_x) < r \\ 2 \cdot (I(n_x) - r) & r \leq I(n_x) \leq 2 \cdot r \\ I(n_x) & 2 \cdot r < I(n_x), \end{cases} \tag{6}$$

which also holds in the same manner for $\overleftarrow{I}(n_z)$. We analyze four cases for different conditions of $I(n_z)$ and $I(n_x)$ based on Equation (6) to prove

$$|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| \leq 2 \cdot |I(n_x) - I(n_z)|. \tag{7}$$

Case 1: $I(n_z) \leq 2 \cdot r$ and $I(n_x) \leq 2 \cdot r$: We further consider the following scenarios:

- **Case 1a**: $I(n_x)$ and $I(n_z)$ are both $\geq r$: Then, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = 2 \cdot |I(n_x) - I(n_z)|$, i.e., Equation (7) holds.
- **Case 1b**: $I(n_x)$ and $I(n_z)$ are both $< r$: Then, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = 2 \cdot |I(n_x) - I(n_z)|$, i.e., Equation (7) holds.
- **Case 1c**: one of $I(n_x)$ and $I(n_z)$ is $< r$ and the other is $\geq r$: Suppose for the first sub-case that $I(n_x) > I(n_z)$. Then, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = 2 \cdot (I(n_x) - r) - 2 \cdot (r - I(n_z)) + 1 < 2 \cdot (I(n_x) - r) - 2 \cdot (r - I(n_z)) + 4 \cdot (r - I(n_z)) = 2 \cdot (I(n_x) - I(n_z)) = 2 \cdot |I(n_x) - I(n_z)|$, where $<$ is due to the assumption that $I(n_z) < r$ and $I(n_z)$ is an integer, i.e., $1 \leq r - I(n_z)$. The other case that $I(n_z) > I(n_x)$ is symmetric. Therefore, the condition in Equation (7) remains to hold.

Case 2: $I(n_z) > 2 \cdot r$ and $I(n_x) > 2 \cdot r$: In this case, the reassignment does not change their positions, i.e., $\overleftarrow{I}(n_z) = 2 \cdot r + (I(n_z) - 2 \cdot r) = I(n_z)$ and $\overleftarrow{I}(n_x) =$

$2 \cdot r + (I(n_x) - 2 \cdot r) = I(n_x)$. As a result, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = |I(n_x) - I(n_z)|$, and Equation (7) holds.

Case 3: $I(n_z) > 2 \cdot r$ and $I(n_x) \leq 2 \cdot r$: When $I(n_x) \geq r$, we have $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = I(n_z) - 2 \cdot |I(n_x) - r| = I(n_z) - 2I(n_x) + 2 \cdot r \leq 2 \cdot |I(n_z) - I(n_x)|$. When $I(n_x) < r$, we have $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = I(n_z) - 2 \cdot r + 2 \cdot I(n_x) + 1 < I(n_z) - 2 \cdot r + 2 \cdot I(n_x) + 4 \cdot r - 4 \cdot I(n_x) = I(n_z) + 2 \cdot r - 2 \cdot I(n_x) \leq 2 \cdot |I(n_z) - I(n_x)|$, where $<$ above is due to the assumption that $I(n_z) < r$ and hence $r - I(n_z) \geq 1$. Therefore, Equation (7) holds.

Case 4: $I(n_z) \leq 2 \cdot r$ and $I(n_x) > 2 \cdot r$: This is the symmetric case of Case 3.

As a result, Equation (7) holds for all cases, so the lemma is proved. □

Lemma 6.

$$C^{decomp}_{lptr,down} \leq C^{decomp}_{split,down} = C_{down} \tag{8}$$

$$C^{decomp}_{rptr,down} \leq C^{decomp}_{split,down} = C_{down} \tag{9}$$

The summed cost for shifting down in decomposed DBCs in the left and right pointer tree is smaller than the cost for shifting down in the split value DBC, which is equal to the cost for shifting down in the unified DBC case.

Proof. We investigate entire root leaf paths from the root to a leaf node. According to Lemma 5, each path contributes to the total cost with the shifts along the path and the absolute leaf probability. From the definition of the cost function we know that $C_{down} = C^{decomp}_{split,down}$. Investigating the cost in the left and right pointer DBCs (Equation 18 and Equation 19) two cases need to be distinguished. As this consideration is symmetric for $C^{decomp}_{lptr,down}$ and $C^{decomp}_{rptr,down}$, we only discuss the left pointer case here. Considering an arbitrary root leaf path from the root node to a leaf node $rlpath(n_l) \setminus \{n_0\} = \{np_0, np_1, ..., np_m\}$, we know from the definition of $isleft$ and LP that for all positions $i_0, i_1, ...$ on the path where $isleft(np_{i_x}) = 1$, $LP(P(np_{i_x})) = P(np_{i_{x-1}})$, i.e. the left-most parent LP of the parent is always the immediate previous parent node which contributes to Equation 18. Further, $|I(P(np_x)) - I(LP(P(np_x)))| \leq \sum_{np_y \in path(np_x, LP(np_x)) \setminus \{np_x, LP(pn_x)\}} |I(np_y) - I(P(np_y))|$ since an arbitrary path between two indices cannot be longer than the direct path. Thus, the contributed cost to $C^{decomp}_{lptr,down}$ for this specific node is at most the contributed cost of this node and the omitted nodes ($isleft = 0$) to $C^{decomp}_{split,down}$. In total, $C^{decomp}_{lptr,down} \leq C^{decomp}_{split,down}$ and $C^{decomp}_{rptr,down} \leq C^{decomp}_{split,down}$. □

Lemma 9.

$$\overleftarrow{C}^{*decomp}_{lptr,up} \leq \overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{down} \tag{10}$$

$$\overleftarrow{C}^{*decomp}_{rptr,up} \leq \overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{down} \tag{11}$$

The cost for shifting up in the left and right pointer DBCs in a linear allowable placement can be upper bounded by the cost for shifting up in the split value DBC, which is the same cost as shifting down in the unified DBC case.

Proof. $\overleftarrow{C}^{*decomp}_{split,up} = \overleftarrow{C}^{*}_{up}$ directly follows from the definition of the cost functions (Equation 9 and Equation 2). $\overleftarrow{C}^{*}_{up} = \overleftarrow{C}^{*}_{down}$ follows from Lemma 3. By investigating Equation 9 and Equation 12 the outer sum is over the same (leaf) nodes. In a linear allowable placement, n_0 must have the left most position, further $I(LP(n_x)) < I(n_x)$ since LP is an indirect parent relation. Thus, all terms $|I(n_r) - I(LP(n_x))| \leq |I((n_x) - I(n_0)|$. The nodes considered in the inner sum of Equation 12, namely $n_r : LP(n_r) = \epsilon$, must form a single consecutive path of nodes where always the right child is taken by definition. Each node on the path contributes a certain portion of their absolute probability ($absprob(n_r) \cdot prob(LC(n_r))$), the remaining part is inherited to the right child by definition, which then itself contributes a part of the inherited probability. Thus, $\sum_{n_r:LP(n_r)=\epsilon} absprob(n_r) \cdot prob(LC(n_r)) \leq absprob(n_0) = 1$ Consequently, the inner sum is a weighted average of upper bounded terms, thus the entire sum can be upper bounded by $|I((n_x) - I(n_0)|$. The case for the right pointer DBC is symmetric. □