BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory

Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen, Fazal Hameed, Jeronimo Castrillon, Jian-Jia Chen
TU Dortmund, Deaprtment of Computer Science, Dortmund, Germany
TU Dresden, Deaprtment of Computer Science, Dresden, Germany

Citation: To be published on 58th ACM/IEEE Design Automation Conference (DAC)

BibTeX:

```
@inproceedings{,
  author    = {Christian Hakert, Asif Ali Khan, Kuan-Hsun Chen, Fazal Hameed, Jeronimo Castrillon,
  Jian-Jia Chen},
  booktitle = {58th ACM/IEEE Design Automation Conference (DAC), accepted},
  title     = {BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory},
  year      = {2021}
}
```

BLOwing Trees to the Ground: Layout Optimization of Decision Trees on Racetrack Memory

Christian Hakert ( ), Asif Ali Khan ( ), Kuan-Hsun Chen ( ), Fazal Hameed ( ),
Jeronimo Castrillon ( ), Jian-Jia Chen ( )
: Technische Universität Dortmund
: Technische Universität Dresden
{christian.hakert, kuan-hsun.chen, jian-jia.chen}@tu-dortmund.de / {asif_ali.khan, fazal.hameed, jeronimo.castrillon}@tu-dresden.de

***Abstract*—Modern distributed low power systems tend to integrate machine learning algorithms, which are directly executed on the distributed devices (on the edge). In resource constrained setups (e.g. battery driven sensor nodes), the execution of the machine learning models has to be optimized for execution time and energy consumption. Racetrack memory (RTM), an emerging non-volatile memory (NVM), promises to achieve these goals by offering unprecedented integration density, smaller access-latency and reduced energy consumption. However, in order to access data in RTM, it needs to be *shifted* to the *access port* first, resulting in latency and energy penalties.**

In this paper, we propose B.L.O. (Bidirectional Linear Ordering), a novel domain-specific approach for placing decision trees in RTMs. We reduce the total amount of shifts during inference by exploiting the tree structure and estimated access probabilities. We further apply the state-of-the-art methods to place data structures in RTM, without exploiting any domain-specific knowledge, to the decision trees and compare them to B.L.O. We *formally* prove that the B.L.O. solution has an approximation ratio of 4, i.e., its number of shifts is guaranteed to be at most 4 times the optimal number of shifts for a given decision tree. Throughout the experimental evaluation, we show that for the realistic use case B.L.O. *empirically* outperforms the state-of-the-art data placement method on average by 54.7% in terms of shifts, 19.2% in terms of runtime and 19.2% in terms of energy consumption.

I. 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 embedded systems that are battery-powered i.e., "on the edge", maintenance cycles can be significantly enlarged 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 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 achieving both energy efficiency and performance optimization.

Racetrack memory (RTM) is a new class of NVMs, which features high integration density, low unit cost and low energy consumption at the cost of access pattern specific *shift* latencies [3]. 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 [7], [10]. 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 data objects all equally and therefore have to consider all data objects possibly being accessed pairwise consecutively.

In this paper, in contrast, we consider the domain-specific knowledge of decision trees and optimize the memory layout on RTM, such that the total amount of shifts can be minimized. To this end, we present a *Bidirectional Linear Ordering* (B.L.O.), which explicitly accounts for the parent relation of nodes within the trees. By knowing that only nodes, which have a direct parent-child relation, can be accessed pairwise consecutive in advance, the B.L.O. heuristic resembles an existing algorithm for solving the *Optimal Linear Ordering* (O.L.O.) problem for constrained rooted trees optimally with a time complexity of $O(m \log m)$.

Our novel contributions:

- A theoretical proof that the solution of the O.L.O. problem for rooted trees causes at most $4\times$ the amount of total shifts than the optimal placement, when applied on decision trees in RTM.
- A placement heuristic named B.L.O., eliminating the major cause for long shift distances between two inferences, where we show that the total amount of shifts is not increased.
- An empirical evaluation, considering realistic runtime and energy consumption models to compare B.L.O. to the state-of-the-art approaches presented in [7], [10].

II. 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 for saving the transmission energy. The target system is assumed to be equipped with a simple CPU core (e.g., few MHz clock rate, no caches), SRAM as main memory and integrated RTM scratchpad memory. Efficient data mapping to the RTM scratchpad may reduce the average access latency, and the energy consumption for accesses to RTM locations 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.

A. 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 performs a comparison of exactly one feature value from the input data with a fixed split value, which then decides if the inference further goes to the left or to the right child.

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

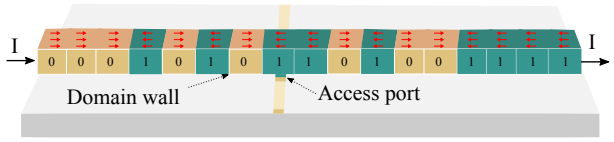

Fig. 1. RTM cell structure

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 $P(n_x)$. In memory, we place each and every node of the tree into a consecutive array of the size of m nodes, where the racetrack shifting cost of accessing index i and j with $0 \leq i, j < m$ in a sequence is $|i-j|$. A valid mapping of nodes N to array indices $I : N \to \{0, 1, ..., m-1\}$ must be bijective.

The inference model always starts paths from the root node and follows the children according to the comparisons at each node until reaching a leaf node. By following the probabilistic model proposed in [6], each comparison is modeled as a Bernoulli experiment, by which each node is assigned with a probability to be accessed from the parent node $prob : N \to [0, 1] \subset \mathbb{Q}$ 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.

B. 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 have its own magnetization orientation as shown in Figure 1. 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 port(s) for read or write operations which 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 out by sensing its magnetization orientation or written by updating its magnetization orientation.

C. RTM Architecture

The hierarchical organization of RTM, like other memory technologies, consists of banks, subarrays, Domain Block Clusters (DBCs), tracks and domains as depicted in Figure 2. Each structure at the highest level (e.g., bank) is decomposed into smaller structures at the next level (e.g., subarray). The essential structure of an RTM is a DBC which contains T tracks each comprising K domains. A single DBC is capable of storing 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 [3]. However, many recent designs consider $K = 64$ which is not only more realistic but also enables efficient utilization of the address bits. In this work, we also assume that 64 nodes of a decision tree can be placed within a single DBC, which can contain a subtree of the maximal depth of 5. As the decision trees we use in this paper are balanced, larger trees can be easily split into such subtrees by introducing dummy leaves, which point to the next subtree. Subtrees in different DBCs can be accessed without additional shifting costs.

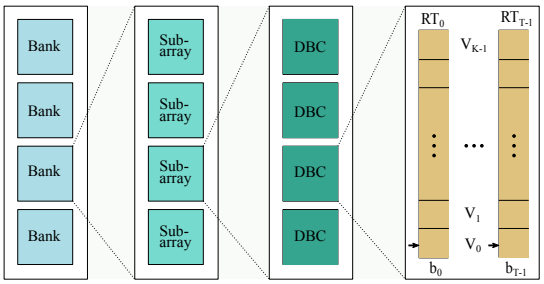

Fig. 2. An overview of the RTM hierarchical organization

D. State-of-the-art Data Placement in RTMs

Recent works [7], [10] propose compiler-guided approximate and optimal solutions for objects placement in RTMs. A memory access trace S can be represented with an undirected 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 [7] 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. Afterwards, 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 object 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 [10] uses 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.

E. Problem Definition

In this work, we focus on the placement optimization to minimize the amount of racetrack shifts for decision trees, which are trained beforehand with corresponding datasets. The studied **problem** is defined as follows:

- **Input:** A binary decision tree, consisting of m nodes, i.e., set N, where each node is associated with a probability to be accessed from its parent. The probability is profiled on the dataset for training. The information of the rooted tree is defined in Section II-A.
- **Output:** A bijective mapping of tree nodes to a memory array which minimizes the expected required racetrack shifts while accessing the tree nodes during inference, assuming the given probabilities of the nodes of the decision tree.

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 $path(n_x)$ to denote the set which contains all nodes on the path from the root node down to n_x. With the help of this we declare the absolute access probability of node n_x as $absprob(n_x) = \Pi_{n_z \in path(n_x)} prob(n_z)$. In addition, every node $n_x \in N$ has a subtree with a subset of leaf nodes $leaves(n_x) \subseteq N_l$ where $\forall n_y \in leaves(n_x) : n_x \in path(n_y)$.

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

$$absprob(n_x) = \sum_{n_y \in leaves(n_x)} absprob(n_y) \qquad (1)$$

III. Decision Tree Specific Placement

Throughout this section, we present our novel approach for a decision tree specific placement heuristic. Given some valid mapping I, the expected cost for the inference of an input value, i.e., following a path from the root to a leaf, is given by Eq. (2):

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

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

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

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

$$C_{total} = C_{down} + C_{up} \quad (4)$$

An optimal mapping I^* for a decision tree on racetrack memory is a mapping which minimizes C_{total}. This problem is an instance of the *Optimal Linear Ordering* (O.L.O.) problem [1], [4], [8]. The O.L.O. 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 O.L.O. (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 O.L.O. problem for rooted trees with the root node on the leftmost position (i.e. only optimizing C_{down}) can be optimally solved in time complexity $O(m \log m)$ [1].

A. Towards Fast O.L.O. for Decision Trees

Throughout this section we use the notations defined in Table I:

Placement	Explanation
I	arbitrary mapping
I^*	optimal mapping which optimizes C^*_{opt}
$I^{*\downarrow}$	optimal mapping which optimizes $C^{*\downarrow}_{down}$
$\overleftarrow{I}$	arbitrary mapping with the root on the left
$\overleftarrow{I^*}$	optimal mapping with the root on the left and with expected down cost $\overleftarrow{C^*}_{down}$

TABLE I
Placement Notation

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

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.* □

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

$$C^{*\downarrow}_{down} \leq C^*_{opt} \quad (5)$$

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 [1] regarding the optimization of $I^{*\downarrow}$ when the root *has to be put* on the leftmost position.

Lemma 2 (Page 410 in [1])**.** *(restated) There exists an optimal unidirectional placement $\overleftarrow{I^*}$ for the O.L.O. 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 leftmost 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. To do so, we show that $C_{down} = C_{up}$. Since we know that I is unidirectional or bidirectional, we also know that a leaf node $n_x \in N_l$ is always the rightmost node or the leftmost node within its path $path(n_x)$ 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:

$$\forall n_y \in N_l : |I(n_y) - I(n_0)| = \sum_{n_z \in path(n_y) \setminus n_0} |I(n_z) - I(P(n_z))| \quad (6)$$

This leads to:

$$C_{up} = \sum_{n_y \in N_l} \left(absprob(n_y) \cdot \sum_{n_z \in path(n_y) \setminus \{n_0\}} |I(n_z) - I(P(n_z))| \right) \quad (7)$$

The summation is reorganized with respect to each node $n_x \in N$ by using the following observation: if n_z is in $path(n_y)$, then n_y is in $leaves(n_z)$. That is, a node $n_x \in N$ contributes to Eq. (7) exactly $|I(n_x) - I(P(n_x))| \cdot \sum_{n_y \in leaves(n_x)} absprob(n_y)$. Therefore,

$$C_{up} = \sum_{n_x \in N \setminus \{n_0\}} \left(|I(n_x) - I(P(n_x))| \cdot \sum_{n_y \in leaves(n_x)} absprob(n_y) \right) \quad (8)$$

Applying Definition 1 leads to Eq. (9):

$$C_{up} = \sum_{n_x \in N \setminus \{n_0\}} (|I(n_x) - I(P(n_x))| \cdot absprob(n_x) = C_{down} \quad (9)$$

□

In the following, we point out the relation between a mapping I and a mapping $\overleftarrow{I}$ 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 leftmost position by increasing the expected cost of $\overleftarrow{C}_{down}$ with at most a factor of 2:*

$$\overleftarrow{C}_{down} \leq 2 \cdot C_{down} \quad (10)$$

Proof. Suppose that the root of the decision tree is assigned at position r in the placement I. Due 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 $i = 1, 2, \ldots, r$.

- reassign every node in position $r+i$ in I to $2 \cdot r + i$ for $i = r+1, r+2, \ldots, m$.
- reassign every node in position $r-i$ in I to $r + 2 \cdot i - 1$ for $i = 1, 2, \ldots, r$.

After that, every node is then shifted by r positions towards the left and the root of the decision tree is on the leftmost position, i.e., 0.

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

$$\overleftarrow{I}(n_x) = \begin{cases} 2 \cdot (r - I(n_x)) - 1 & I(n_x) < r \\ 2 \cdot (I(n_x) - r) & r \le I(n_x) \le 2r \\ I(n_x) & 2r < I(n_x), \end{cases} \tag{11}$$

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 Eq. (11) to prove

$$|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| \le 2|I(n_x) - I(n_z)|. \tag{12}$$

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

- **Case 1a**: $I(n_x)$ and $I(n_z)$ are both $\ge r$: Then, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = 2|I(n_x) - I(n_z)|$, i.e., Eq. (12) holds.
- **Case 1b**: $I(n_x)$ and $I(n_z)$ are both $< r$: Then, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = 2|I(n_x) - I(n_z)|$, i.e., Eq. (12) holds.
- **Case 1c**: one of $I(n_x)$ and $I(n_z)$ is $< r$ and the other is $\ge r$: Suppose for the first sub-case that $I(n_x) > I(n_z)$. Then, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = 2 \cdot (I(n_x) - r) - 2(r - I(n_z)) + 1 < 2 \cdot (I(n_x) - r) - 2(r - I(n_z)) + 4(r - I(n_z)) = 2 \cdot (I(n_x) - I(n_z)) = 2|I(n_x) - I(n_z)|$, where $<$ is due to the assumption that $I(n_z) < r$ and $I(n_z)$ is an integer, i.e., $1 \le r - I(n_z)$. The other case that $I(n_z) > I(n_x)$ is symmetric. Therefore, the condition in Eq. (12) remains to hold.

Case 2: $I(n_z) > 2r$ and $I(n_x) > 2r$: In this case, the reassignment does not change their positions, i.e., $\overleftarrow{I}(n_z) = 2r + (I(n_z) - 2r) = I(n_z)$ and $\overleftarrow{I}(n_x) = 2r + (I(n_x) - 2r) = I(n_x)$. As a result, $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = |I(n_x) - I(n_z)|$, and Eq. (12) holds.

Case 3: $I(n_z) > 2r$ and $I(n_x) \le 2r$: When $I(n_x) \ge r$, we have $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = I(n_z) - 2|I(n_x) - r| = I(n_z) - 2I(n_x) + 2r \le 2 \cdot |I(n_z) - I(n_x)|$. When $I(n_x) < r$, we have $|\overleftarrow{I}(n_x) - \overleftarrow{I}(n_z)| = I(n_z) - 2r + 2I(n_x) + 1 < I(n_z) - 2r + 2I(n_x) + 4r - 4I(n_x) = I(n_z) + 2r - 2I(n_x) \le 2 \cdot |I(n_z) - I(n_x)|$, where $<$ above is due to the assumption that $I(n_z) < r$ and hence $r - I(n_z) \ge 1$. Therefore, Eq. (12) holds.

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

As a result, Eq. (12) holds for all cases and the lemma is proved. □

Suppose that $\overleftarrow{I^*}$ is an optimal unidirectional mapping 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 mapping which optimizes $C^{*\downarrow}_{down}$. We conclude the following corollary:

Corollary 1.

$$\overleftarrow{C^*}_{down} \le 2 \cdot C^{*\downarrow}_{down} \tag{13}$$

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 leftmost position. Therefore, $C^{*\downarrow}_{down}$ is a lower bound of any solution when the root is on the leftmost position. By Lemma 4, $I^{*\downarrow}$ can be converted into a placement $\overleftarrow{I}$, in which the root is put to the leftmost position, with a cost up to $\overleftarrow{C}_{down} \le 2 \cdot C^{*\downarrow}_{down}$. 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.

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

□

We now explain how to derive an optimal unidirectional solution that minimizes $\overleftarrow{C^*}_{down}$ efficiently. Adolphson and Hu [1] proposed an algorithm to optimally solve this case. Specifically, according to [1], the O.L.O. 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 O.L.O. problem in $O(m \log m)$ time complexity.

B. Bidirectional Linear Ordering

Deriving a mapping by the algorithm from Adolphson and Hu at most causes $4\times$ the cost compared to the optimal solution for our placement problem. The algorithm from Adolphson and Hu has the main drawback that it places the root node to the leftmost slot in any solution, which is not optimal when the cost for going back from leaves to the root between inferences is considered. Our final algorithm computes a *Bidirectional Linear Ordering* (B.L.O.). We map the two subtrees underneath the root by the algorithm from Adolphson and Hu, which derives a mapping I_L for the left subtree and a mapping I_R for the right subtree. Both mappings cause an expected cost which is at least 2 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 path to a leaf and back to the root. We then form the final B.L.O. mapping by placing $I^\diamond = \{reverse(I_L), 0, I_R\}$. In this mapping two shifts are then added again to every path into and out of the right and left subtree, thus $C^\diamond_{total} \le C_{total}$.

Considering the exemplary decision tree in Figure 3, each access would start at the leftmost position in the first placement, target a leaf within the rest of the mapping and shift back to the leftmost position. In the second mapping, as long as leaves from the left and right subtree are accessed on a similar ratio, the expected shifting distance is divided by a factor of 2. 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 B.L.O. is $O(m \log m)$.

IV. Evaluation

In order to compare our Bidirectional Linear Ordering (B.L.O.) approach to the state-of-the-arts (i.e., ShiftsReduce [10] and Chen et al. [7]), we adopt an open-source framework published in [5] and select 8 typical machine learning classification datasets from the UCI Machine Learning Repository [14] and [13]: adult, bank, magic,

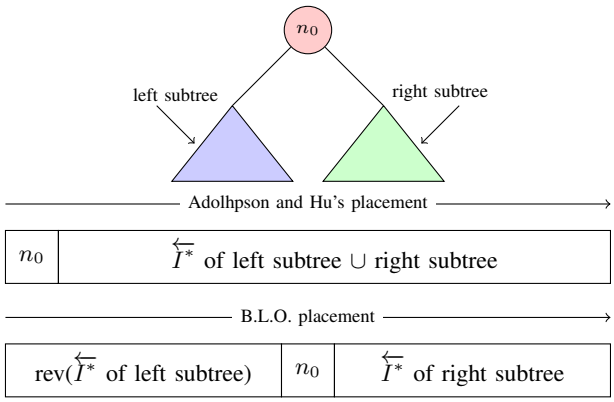

Fig. 3. Suboptimal Placement Correction

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 on the training data by using tree classifiers in the `sklearn` package [16].

To derive different sized trees, we specify the maximum depth of the trees, e.g., DT1 means that the tree has 2 levels and DT3 means that the tree has 4 levels. After the trees are generated, we profile the node probabilities on the training data by counting how often either the left child or the right child of each node is visited. This delivers us empirical branch probabilities and absolute node access probabilities, respectively. We further 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 map the trees to a memory layout with the compared approaches and replay the node access trace to derive the total amount of required racetrack shifts under the assumption that the entire tree is placed in a single DBC. Although this already allows a quantitative comparison of the placement approaches, we further compute the energy consumption and total runtime on a realistic model, derived from the various memory mappings. For the runtime, we use the per-access and per-shift latencies in Table II 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 is $runtime = \ell_R \cdot n_{accesses} + \ell_S \cdot n_{shifts}$. 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 II.

Ports/track, tracks/DBC, domains/track		1, 80, 64
Leakage power [mW]	p	36.2
Write / Read / Shift energy [pJ]	$e_W/e_R/e_S$	106.8 / 62.8 / 51.8
Write / Read / Shift latency [ns]	$\ell_W/\ell_R/\ell_S$	1.79 / 1.35 / 1.42

TABLE II
RTM PARAMETERS VALUES FOR A 128 KiB SPM

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

A. Result Discussion

Figure 4 depicts the experimental results for the reduction of the total amount of shifts by the different placement approaches. All results indicate the relative amount of racetrack shifts compared to a *naive* placement, which is derived by traversing the tree in breath-first order while placing the nodes consecutive in memory as they are traversed. Despite applying our proposed B.L.O. algorithm, ShiftsReduce [10] and Chen et al. [7], we also formulate the mapping problem as a mixed integer program (MIP), which optimizes Eq. (4). We implement this MIP in the Gurobi optimizer [2] and set a time limitation of 3 hours per dataset and tree configuration. For all datasets, the MIP converges to the optimal solution only for DT1 and DT3. For all other cases, the result is based on the Gurobi heuristic. Results which are worse than $1.2\times$ of the naive placement are not included.

Investigating the illustrated results, it can be observed that for the cases where the MIP finds an optimal mapping (for DT1 and DT3), B.L.O. achieves the same or only marginally worse results than the optimum. This supports the heuristic design principle of B.L.O. (Section III-B). Furthermore it can be observed that B.L.O. achieves the best reduction in shifts for most of the investigated cases. Considering the mean improvement over all evaluated datasets and trees, B.L.O. reduces the amount of required shifts by 65.9% compared to the naive placement. ShiftsReduce reduces the required amount of shifts by 55.6%. This implies that B.L.O. further improves the amount of required shifts by 18.7% upon ShiftsReduce.

Please note that deciding the placement based on the profiled probabilities from the training dataset does not necessarily result in the expected cost for the test dataset, when both datasets are too different. We determine the required amount of shifts when the training dataset is inferred on the decision tree, after the mapping is decided on the profiled probabilities of the same dataset. The results report minimal difference: B.L.O. on average reduces the required amount of shifts on the train dataset by 66.1%, and ShiftsReduce reduces the required amount of shifts by 55.7%.

The reduction of the total amount of shifts is an indicator, which does not immediately reflect a realistic improvement in runtime or energy consumption. Therefore we compute the improvement of the total runtime and energy consumption for the placement approaches. Section II-C points out that in a realistic setup, larger decision trees are split into smaller trees first and the placement heuristic is then executed on multiple DT5 sized trees. Therefore we present the average runtime and energy consumption improvements for all DT5 experiments: B.L.O. improves the overall runtime by 71.9% compared the naive placement, the total energy consumption by 71.3% respectively. ShiftsReduce, in comparison, improves the overall runtime by 60.3% and the total energy consumption by 59.8%. Thus, B.L.O. improves runtime and energy consumption by 19.2% compared to ShiftsReduce. Comparing this to the reduction of shifts for DT5 sized trees only, B.L.O. reduces the required shifts by 74.7%, ShiftsReduce by 48.3%, thus B.L.O. improves ShiftsReduce by 54.7%. This draws the conclusion that despite static energy consumption and read latency having a non-negligible influence, the reduction of the amount of racetrack shifts results in a significant improvement of the runtime and energy consumption.

V. Related Work

To reduce the number of shifts for RTMs, various techniques have been proposed in the literature, such as runtime data swapping [18], preshifting [18], intelligent instruction [15], and data placement [7], [10]. For data placement, Chen et al. in [7] present a heuristic

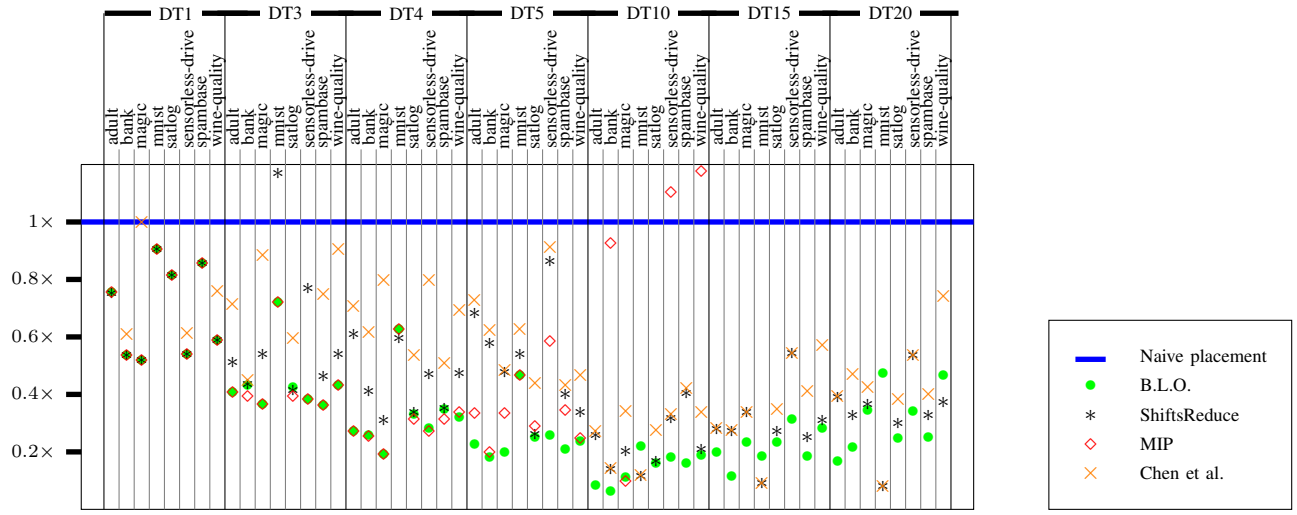

Fig. 4. Comparison of Total Shifts During Inference

appending data objects according to the adjacency information sequentially. Khan et al. in [10] 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 that the data objects of decision trees, such that possible access patterns of objects are strictly limited by the dependencies between tree nodes.

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 [11]. In fact, the studied problem can be treated as an instance of the quadratic assignment problem (QAP), which was introduced in 1957 [12], 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 [4]. 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 [17]. For directed trees, Adolphson and Hu in [1] 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 a track back from leaves to the root between two inferences needs to be considered.

VI. Conclusion

In this paper we present B.L.O., a domain specific placement heuristic for decision trees on RTM. B.L.O. 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. B.L.O. bases on an optimal algorithm to solve the O.L.O. problem for rooted trees [1] and eliminates the main reason for improper placements on RTM. B.L.O. hereby causes at most $4\times$ of the RTM shifts than the optimal placement and features a time complexity of $O(m \log m)$, which makes it feasible for large decision trees. Our empirical evaluation points out that for the most realistic use case of decision trees with a maximal depth of 5, B.L.O. outperforms the state-of-the-art placement heuristics by 54.7%, 19.2%, and 19.2% in terms of RTM shifts, runtime, and energy consumption, respectively.

Acknowledgement

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

References

[1] D. Adolphson and T. C. Hu. Optimal linear ordering. *SIAM Journal on Applied Mathematics*, 25(3):403–423, 1973.
[2] B. Bixby. The gurobi optimizer. *Transp. Re-search Part B*, 2007.
[3] 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*, 2020.
[4] R. E. Burkard, E. Çela, P. M. Pardalos, and L. S. Pitsoulis. *The Quadratic Assignment Problem*, pages 1713–1809. 1998.
[5] 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)*, pages 19–28, 2018.
[6] 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*, 65(1):209–222, 2018.
[7] 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.*, 2016.
[8] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Comput. Surv.*, 34(3):313356, Sept. 2002.
[9] M. R. Garey and D. S. Johnson. *Computers and intractability*. 1979.
[10] 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.*, 16(4), Dec. 2019.
[11] 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*, pages 5–18, 2019.
[12] T. C. Koopmans and M. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25(1):53–76, 1957.
[13] Y. LeCun. The mnist database of handwritten digits. 1998.
[14] M. Lichman. UCI machine learning repository, 2013.
[15] 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*, 2019.
[16] 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*, 12:2825–2830, 2011.
[17] Y. Shiloach. A minimum linear arrangement algorithm for undirected trees. *SIAM Journal on Computing*, 8(1):15–32, 1979.
[18] 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)*, pages 1–6, 2013.