Optimizing Random Forest Based Inference on RISC-V MCUs at the Extreme Edge

Optimizing Random Forest Based Inference on RISC-V MCUs at the Extreme Edge

https://doi.org/10.1109/TCAD.2022.3199903

Optimizing Random Forest Based Inference on RISC-V MCUs at the Extreme Edge

Enrico Tabanelli, Giuseppe Tagliavini, *Member, IEEE*, and Luca Benini, *Fellow, IEEE*

***Abstract*—Random Forests (RFs) use a collection of Decision Trees (DTs) to perform classification or regression. RFs are adopted in a wide variety of Machine Learning (ML) applications, and they are finding increasing use also in scenarios at the extreme edge of the IoT (TinyML) where memory constraints are particularly tight. This paper addresses the optimization of the computational and storage costs for running DTs on the microcontroller units (MCUs) typically deployed in TinyML scenarios. We introduce three alternative DT kernels optimized for memory- and compute-limited MCUs, providing insight into the key memory-latency trade-offs on an open-source RISC-V platform. We identify key bottlenecks and demonstrate that SW optimizations enable up to significant memory footprint and latency decrease. Experimental results show that the optimized kernels achieve up to 4.5 μs latency, 4.8× speedup and 45% storage reduction against the widely-adopted Naive DT design. We carry out a detailed performance and energy cost analysis of various optimized DT variants: the best approach requires just 8 instructions and 0.155 pJ per decision.**

***Index Terms*—Random Forest, Decision Tree, Machine Learning, RISC-V, Edge-Computing**

I. Introduction

Random Forests (RFs) are a family of learning methods widely adopted among a broad range of Machine Learning (ML) applications, such as Ranking Items [1], [2] and Click-Through Prediction [3], [4]. By aggregating many tree-like models acting as simple but weak learners, called Decision Trees (DTs), the algorithm builds a stronger learner capable of overcoming the overfitting-related issues that affect single learner approaches. Tree-based models are also being advocated among other ML approaches [5], [6] for applications demanding higher explainability and interpretability than ”black-blocks” ML algorithms [7]. Furthermore, the capacity to achieve good results with small training datasets [8] even with high-dimensional feature spaces [9] favors the adoption of RFs in applications where training data may be limited and feature selection unfeasible.

During the last years, the Internet-of-Things (IoT) ecosystem has experienced continuous growth, reaching 30 Billions of connected endpoints in 2020 and projected towards 74 Billions by 2025 [10]. Driven by this trend, the amount of sensor data generated at the network edge has increased dramatically [11]. So far, the limited end-nodes memory budget and computing power favored the solution of offloading data to the cloud for large-scale analytics, where resources are flexible and virtually unbounded. However, this cloud-centric paradigm poses several scalability issues [12], such as communication latency, energy efficiency, bandwidth, and privacy, which are severe for many IoT services. The paradigm shift toward near-sensor processing on resource-constrained devices is referred to as tinyML [13]–[15]. This approach opens up new challenges for the execution of complex tasks on edge devices. The main benefits of TinyML platforms include energy efficiency, data privacy protection, reduced bandwidth costs, and low-latency response [16].

In this contest, RFs have already been demonstrated as a viable solution in latency- (e.g., Health Monitoring Wearable Devices [17]) and privacy-sensitive IoT applications (e.g., Non-Intrusive Load Monitoring [18]). Monitoring systems extract bio-signals with sampling frequencies up to 10 kHz [19], thus enabling condition prevention on MCU-based devices leads to a latency requirement of about 100 μs. Non-Intrusive Load Monitoring (NILM) requires performing inference in a few μs since it leverages high-frequency features up to 1 MHz to distinguish switched-mode power supplies devices [20]. Recent works achieved an RF inference time of 49.9 ms with a 20 Inf/s throughput on the Raspberry Pi 3B [21]. The authors in [22] reached 120 μs and 8.3 kInf/s on the ARM Cortex-M4 architecture. Unfolding DTs structure into a sequence of nested if-then-else statements is a widely-adopted technique to target RF execution on MCU-class devices with embedded FLASH [23]. However, FLASH-less System-on-Chips (SoCs) integrate limited-size instruction caches (I$) which would become the bottleneck for execution time in case of massive code size filled with conditional branches. Alternative RF optimization approaches targeting TinyML platforms have been proposed, but at the cost of algorithm modifications and accuracy drops [24]. Thus, optimizing RFs to achieve fast inference without accuracy loss is a challenging task for traditional Von Neumann architectures [25]. Making a DT prediction requires traversing a tree structure via root-to-leaf paths and performing a comparison at each level to select the branch to take. This pattern demands non-uniform memory access to the DT model and input vector since comparison results are unpredictable and disclosed only at runtime. The inability to

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2022 and appeared as part of the ESWEEK-TCAD special issue.

This work was supported by the EU-funded BonsAPPs project (grant agreement 101015848) and the European PILOT (grant agreement 101034126).

E. Tabanelli, and L. Benini are with the Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, 40136 Bologna, Italy (e-mail: luca.benini@unibo.it, enrico.tabanelli3@unibo.it).

G. Tagliavini is with the Department of Computer Science and Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: giuseppe.tagliavini@unibo.it).

L. Benini is also with the Department of Information Technology and Electrical Engineering at the ETH Zurich, 8092 Zurich, Switzerland. (e-mail: lbenini@iis.ee.ethz.ch).

hide the access latency due to such unpredictability makes the RF algorithm memory-bound. By adopting high-end platforms to speed up the comparison-based workload, the state-of-the-art (SoA) focused on achieving low-latency inference at the cost of introducing significant constraints [26]. While FPGA-based accelerators provide low flexibility by hard-coding RF hyper-parameters into hardware blocks, GPU-based engines leverage algorithmic improvements not supported by largely-adopted ML frameworks. Furthermore, such approaches to RF demand HW resources unavailable on commercial battery-operated IoT devices.

The extra computational and memory resources required to aggregate DTs result with the majority voting block are almost negligible compared to the DTs runtime and memory requirements. Thus, we focus on improving DT execution, leading to the largest benefits in terms of memory and compute time for RF. This paper introduces a collection of optimized DT kernels designed to reduce the computational and memory costs required to execute RF models on MCUs. MCUs are general-purpose and offer higher programmability and faster prototyping times, making application design and on-the-fly upgrades far cheaper than HW-specialized engines. MCUs are the natural execution platform for TinyML applications since they operate in a power envelope of a few milliWatts meeting the limited budgets of battery-operated systems. Reduced Instruction Set Computer (RISC) CPUs embody the ideal architecture for edge applications by providing highly-optimized fixed-length instructions that allow capping the pipeline complexity of the cores. In addition, industry-standard MCUs widely deployed in embedded applications such as MSP430[1], STM32F4[2], and TriCore[3], integrate DSP-like operations. We perform an experimental assessment of our proposed kernels on PULPissimo [27], a RISC-V MCU supporting a wide set of ML- and DSP-centric instructions. The SoC delivers up to 2400 MOPS (187 MHz, 0.52 V) at the energy efficiency of 433 MOPS/W within a worst-case power envelope of 32.1 mW.

The main contributions of this paper are:

- We introduce the design of three alternative DT kernels (DT-Arr, DT-Loop, and DT-Rec) optimized to execute on memory- and compute-constrained MCUs. We leverage dedicated node structure in each kernel which determines the model memory footprint and the computing time necessary to traverse the DT. To this purpose, we evaluate time and space complexity, decomposing the last one into code and data contributions. The set of kernels has been developed in a high-level machine-independent C description to enable deploying the functions on alternative architectural targets and is currently available online[4].
- We optimized the DT kernels to maximize the performance on a RISC-V MCU using the baseline RV32IMFC Instruction Set Architecture (ISA). We leveraged the dedicated XpulpV2 extension supported by the MCU to further improve the Cycles per Instruction (CPI). The optimized kernels mainly benefit from hardware loop, post-incrementing load/store, immediate branching, and MAC instructions.
- We present a per-kernel fine-grained analysis breaking down storage and compute costs to reveal runtime bottlenecks on a RISC-V MCU. By leveraging the exploration results, we pinpoint a set of SW optimizations and data structure improvements to lower memory and runtime requirements. The proposed optimizations are typically hardware-agnostic; in some cases, we also propose optimizations requiring the support of platform-specific HW features. Avoiding memory alignment in DT-Rec data structures leads to 26% storage reduction compared to the unaligned kernel version, thus requiring only 43 kB to store the model. We reach a 21% latency decrease regarding the DT-Arr kernel baseline by exploiting offset addressing and reducing memory accesses, resulting in 2.04 kCycles per inference.
- We compare the alternative DT kernels proposed in this work against the largely-adopted Naive DT design, showing significant memory-latency trade-offs. With respect to such representation, we decrease the model memory footprint by 13.6% deploying the DT-Loop kernel while up to 45.4% with the DT-Rec approach that requires only 45.77 kB. Instead, the DT-Arr design can be optimized reaching a $4.8\times$ speedup with 1.8 kCycles. Along with that, we report energy and throughput measurements showing our approaches require down to 15.6 pJ per inference while enabling 223 kilo inferences per second (kInf/s).
- We provide insight into the kernel potential to fit resource-constrained MCUs, focusing on the architectural factors that affect execution time. We also illustrate the computational and storage resource costs demanded by the alternative designs extrapolating kernel-dependant metrics, such as memory accesses per decision (Mem/Dec), number of nodes (N), and Bytes per node (Bytes/Node), along with platform-specific instructions per decision (Instr/Dec) and pJ per decision (pJ/Dec). Thus, we highlight the 8 Instr/Dec and 0.155 pJ/Dec reached by the DT-Rec approach design.

The rest of this paper is organized as follows. In Section II, we present related works and discuss the latest approaches on RF acceleration. Section III describes the alternative DT kernels design introduced. Experimental results of our per-kernel fine-grained analysis and comparison against the Naive DT design are reported in Section IV. Section V concludes this article.

II. Related work

Targeting RF execution on MCUs, Sudharsan et al. [23] presented an SRAM-optimized strategy that does not allocate tree models in SRAM, exploiting instead FLASH memory which is generally more abundant in MCUs with embedded

[1] https://www.ti.com/microcontrollers-mcus-processors/microcontrollers/msp430-microcontrollers/overview.html

[2] https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[3] https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/

[4] https://github.com/EEESlab/RF-on-RISCV

FLASH. The proposed method hard-codes DT decision nodes into the C program (located in the FLASH memory) without allocating external variables and thus relies only on a large code. This approach effectively corresponds to the DT kernel that we use as a comparison baseline. Sudharsan key insight enables MCUs with embedded FLASH to exploit its density for storing a large executable while saving on the scarcer SRAM space. However, we focus on MCUs without embedded FLASH. In his setting, the massive code size intuitively leads to high pressure on the I$ and a related high miss rate, causing significant central processing unit (CPU) stalls and inefficient execution. Daghero et al. [24] proposed an adaptive inference strategy to improve RF efficiency by terminating the inference as soon as high-enough classification confidence is reached. This early-stopping mechanism introduces a trade-off between efficiency and precision. In contrast, our work focuses on optimizing efficiency without compromising inference accuracy. Thus, Daghero's technique is complementary and possibly synergistic to ours.

HW/SW co-design techniques have also been investigated to accelerate tree-based algorithms inference. These strategies usually differ in the HW target used to reduce DT traversal time and SW-algorithmic modifications introduced to fit the HW design better. High-performance FPGAs enable the rapid design of HW-specialized accelerators targeting domain-specific workloads while drastically reducing algorithm bottlenecks. For that purpose, FPGAs have been a widely adopted solution in literature [28], [29] to improve RF latency, thus allowing to meet the timing requirements of real-time tasks. Zhao et al. [30] proposed a flexible FPGA-based RF accelerator that encodes DTs structure information into instructions from a newly designed reduced ISA. Although achieving a 156 Msample/s throughput, the accelerator strength lies in deploying a novel RF representation yet not supported by commonly deployed ML frameworks. By introducing a 32-bit format to reduce nodes representation memory footprint, Alcolea et al. [31] presented a DT accelerator that minimizes data transfers with external memory. The FPGA-based accelerator achieves a $2\times$ speedup and $72\times$ energy improvement w.r.t. to an Intel I5-8400 CPU while decreasing the execution time by $30\times$ and the energy consumption by $23\times$ compared to an ARM Cortex-A9 core. Unfortunately, such approaches require a large amount of computing resources to hard-wire the computations of the overall RF structure into HW. As a result, the FPGA-based accelerator fits only a specific model demanding a reconfiguration phase with high latency costs whenever updating the algorithm.

The recent need for deploying large RFs in large-scale scenarios within small-time budgets has favored the exploitation of aggressive parallel strategies on GPGPUs. In [32], the authors investigate several multi/many-core parallelization strategies to speed up the DT traversal time by comparing an Intel Xeon CPU against an NVIDIA GTX GPU. The proposed GPU-based parallelization approach achieves the maximum performance with up to $100.8\times$ speedup w.r.t. a sequential execution of the algorithm. Van Essen et al. [33] presented a comparative study of alternative HW targets for accelerating Compact Random Forest (CRF), an algorithmic RF variation that trains fixed height DTs. The results showed that GPGPUs have hard resource bounds that are highly sensitive to the classifier and sample size, thus leading to a performance degrading with larger classifiers. Furthermore, Nakahara et al. [34] remarked that GPU architectures do not represent the optimal target for RF acceleration due to the higher cost of all-to-all communications.

In the last years, In-Memory Computing (IMC) accelerators have been gaining momentum by leveraging Non-Volatile Memory (NVM) crossbar arrays to accelerate workloads directly. The capability to operate at low power while achieving high throughputs has favored IMC technology adoption to accelerate RF inference among several recent works [35], [36]. Kang et al. [37] introduced an energy-efficient and high-throughput RF multi-class inference accelerator demonstrating a $6.8\times$ lower Energy-Delay Product (EDP) w.r.t. 8-bit digital implementation. Unfortunately, the IMC accelerator requires an optimal voltage tuning on the number of DTs to avoid accuracy drops, thus impeding on-the-fly model updating. Furthermore, the proposed approach lacks the support from ML frameworks to produce the algorithmic modifications needed to fit the IMC design. In [38], the authors introduced a DT-based IMC accelerator utilizing an analog memristor-based Content Addressable Memory (CAM). While achieving an impressive performance of 333 Million decisions per second (MDec/s) throughput and 1.28 nJ per decision (nJ/Dec) energy consumption, the CAM-based accelerator suffers from IMC-related drawbacks such as computational inaccuracy due to conductance noise.

HW/SW co-design techniques reviewed above provide significant throughput and latency performance but fail to meet the limited TinyML budgets. For this purpose, adopting SW-optimized DT kernels targeting general-purpose RISC-based platforms becomes crucial to enabling low-latency RF inference at the edge. We focus on this approach; more specifically, we aim to minimize latency and storage boundary for RF execution on low-cost MCUs.

III. DT Kernels Design

This section introduces the DT kernels designed to accelerate RF execution on memory- and compute-constrained MCUs. A binary DT doubles the number of nodes at each level leading to $2^{(H+1)}-1$ nodes with H representing the tree depth. Each kernel features a specific node structure layout that determines the associated memory footprint and a set of optimizations aimed at reducing the DT traversal time. We estimate time complexity by considering data loading, comparison, and conditional branching as elementary operations while determining the space complexity by breaking down code and data contributions. The complexities of the introduced kernels have been summarized in Table I. Lastly, we take as a comparison baseline a Naive DT kernel, which represents a widely adopted solution in literature for embedded applications [18], [39]–[41] and leverages the support of several code-generation libraries and frameworks [42]–[45].

TABLE I: Kernels code, data, and time complexity. H: Tree depth, L: Traversed branches.

	Code Complexity	Data Complexity	Time Complexity
Naive	$O(4 \times 2^{H+1})$	$O(2^{H+1})$	$O(4L)$
DT-Loop	$O(1)$	$O(4 \times 2^{H+1})$	$O(13L)$
DT-Rec	$O(1)$	$O(6 \times 2^{H})$	$O(9L)$
DT-Arr	$O(1)$	$O(4 \times 2^{H+1})$	$O(15L)$

A. Naive DT Kernel

The Naive approach consists of fully unfolding the DT structure into a sequence of nested *if-then-else* statements until reaching leaf nodes. Inside each decision node i, the algorithm accesses a feature f_i of the input vector X_{in} and compares it with the corresponding threshold t_i, choosing the left or right branch accordingly. Since the feature f_i and threshold t_i are already known at compile-time from the training process, this design leverages their constant values to unroll the overall tree structure. As observed from Listing 1, the kernel features $O(4L)$ time complexity with L denoting the branches traversed at inference time and 4 the basic operations required per decision node. By assuming storing threshold values in memory, the Naive method presents $O(2^{H+1})$ data space complexity since thresholds count matches the tree nodes. Due to fully unrolling the DT structure, the basic operations per decision along with the nodes count bound the program dimension determining the overall operations required and leading to $O(4 \times 2^{H+1})$ code space complexity. The lack of additional parameters external to the kernel and its straightforward algorithmic structure made the Naive DT method a widely adopted solution.

Listing 1: Naive DT Kernel

```
if(X_in[11] <= 332.0)              /* [f0=11,t0=332.0] */
{
    if(X_in[17] <= 198.0)          /* [f1=17,t1=198.0] */
    {
        classes[1]++;              /* Leaf  Class = 1 */
    }
    else
    {
        if(X_in[13] <= 78.01)      /* [f3=13,t3=78.01] */
        { ...
```

B. DT-Loop Kernel

By representing the tree node as a recursive data structure and defining the DT layout as a collection of nodes, this algorithmic variant allows traversing trees from root to leaf through a *while-loop* statement. As shown in Listing 3, the data structure encloses the mandatory node attributes to represent a tree, that are feature, threshold, and children. The children properties present two variables sharing the same space address representing the pointer to the child node (right$_{\text{child}}$, left$_{\text{child}}$) and the leaf class (right$_{\text{class}}$, left$_{\text{class}}$). In Listing 2, we reported the main body of the routine executed to decide the branch to take whenever entering a new node. The kernel starts accessing the root node from the top of the tree and enters the *while-loop* control flow statement. After evaluating the input feature and threshold comparison, we update the node with left$_{\text{child}}$ or right$_{\text{child}}$ depending on the selected branch. To discriminate leaf from decision nodes and determine the input class, we tag leaf node thresholds with a dedicated out-of-range value δ and check this value in the *while-loop* condition before traversing the subsequent nodes. When the runtime reaches a leaf node, we interchangeably access the left$_{\text{class}}$ or right$_{\text{class}}$, which stores the input class at this tree-level. By requiring a 4 word-sized fields to represent the decision node, the DT-Loop kernel delivers $O(4 \times 2^{H+1})$ data space complexity and features $O(1)$ code complexity due to its simple structure. For what concerns the prediction time, the design demands loading 6 values while executing 4 arithmetic operations, 2 comparisons, and a conditional branch for branch evaluation, thus leading to a $O(13L)$ complexity.

Listing 2: DT-Loop Kernel

```
node = node_root;
while (node->threshold != δ)
{
    cmp = X_in[node->feature] <=
          node->threshold;
    node = (cmp * node->left_child) +
           (!cmp * node->right_child);
}
class =  node->right_class;
```

C. DT-Rec Kernel

An H-level tree features $2^{(H+1)} - 1$ nodes divided between $2^H - 1$ decision nodes and 2^H leaf nodes, implying a significant memory allocation reserved for leaf nodes. For that purpose, the DT-Rec kernel embeds leaf nodes into parent decision nodes, thus requiring to store in memory only $2^H - 1$ nodes against the original $2^{(H+1)} - 1$. Such optimization allows roughly a 50% nodes decrease but demands extending the node data structure by two additional fields as shown in Listing 4. The integration of left$_{\text{class}}$ and right$_{\text{class}}$ leaf attributes induces a 50% memory increase for a single node, but the overall model space requirement drops by 25%.

Listing 3: DT-Loop Struct

```
struct Node{
    uintn_t feature;
    float threshold;
    union {
      Node *left_child;
      uintn_t left_class
    };
    union {
      Node *right_child;
      uintn_t right_class
    };
}
```

Listing 4: DT-Rec Struct

```
struct Node{
    uintn_t feature;
    float threshold;
    Node *left_child;
    Node *right_child;
    uintn_t left_class;
    uintn_t right_class;
}
```

In Listing 5, we reported the DT-Rec kernel routine consisting of a recursive function calling itself whenever accessing new decision nodes. Invoking the function with X$_{\text{in}}$ and root node as actual parameters, the routine starts evaluating the feature-threshold comparison and selects which branch to take depending on the result. Until reaching leaf nodes tagged by a specific value δ stored in $\text{left}_{\text{child}}$ and $\text{right}_{\text{child}}$, the kernel infers recursive calls using as parameters X_{in} and the selected child. By requiring to store in memory only $2^H - 1$ nodes represented by 6 word-sized fields, the kernel design demands $O(6 \times 2^H)$ data space complexity while featuring a negligible program size. Instead, the kernel routine loads 5 values and executes 2 comparisons and 2 conditional branches at each node, leading to a time complexity of $O(9L)$.

Listing 5: DT-Rec Kernel

```
function DT2Rec(X_in, Node){
    if(X_in[Node->feature] <= Node->threshold){
        if(Node->left_child != δ){
            return Dt2Rec(X_in, Node->left_child);
        } else {
            return Node->left_class;
        }
    } else {
        if(Node->right_child != δ){
            return Dt2Rec(X_in, Node->right_child);
        } else {
            return Node->right_class;
        }
    }
}
```

D. DT-Arr Kernel

Due to memory alignment in structure-based tree representations, the previous approaches prevent saving storage resources by adopting a fine-grain tuning at the byte level. For that purpose, the DT-Arr kernel adopts an array-based tree representation consisting of storing node attributes into three arrays, as reported in Listing 6. While features and thresholds represent tree nodes elements, the child array contains left and right child nodes as consecutive elements resulting in double the size. While threshold values always require the single-precision floating-point format, features and child arrays represent a non-negative integer domain potentially restricting the representation range to 1 Byte.

Listing 6: DT-Arr Arrays

```
uintn_t features[Nodes_tree] = { 24, 16, ..., 0, 0 };
float threshold[Nodes_tree] = { 3.63, 7.1, ..., -2 };
uintn_t child[Nodes_tree×2] = { 4, 224, ..., 2 };
```

As shown in Listing 7, the DT-Arr kernel starts reading the root node index and feature while deploying the threshold to distinguish leaf from decision nodes in the *while-loop* condition. The instructions in the loop body read the input feature and compare it against the threshold value while updating the node with child depending on the selected branch. After reading the new feature based on the fresh node index, the kernel checks if a leaf node is reached and eventually retrieves the input class. While the DT-Arr routine features a minimal code dimension, the array-based tree representation enables $O(4 \times 2^{H+1})$ data complexity. Instead, evaluating a decision node requires executing 6 loads, 6 arithmetic operations, 2 comparisons, and a conditional branch for a total of a $O(15L)$ prediction time complexity.

Listing 7: DT-Arr Kernel

```
node = 0;
feature = features[0];
while(threshold[node] != δ)
{
    cmp = (X_in[feature] <= threshold[node]);
    node = cmp * child[2*node] +
          !cmp * child[2*node + 1];
    feature = feature[node];
}
class = child[2*node];
```

IV. Experiments

In this section, we perform an experimental assessment of the DT kernels described in Section III. Section IV-A introduces the experimental setup along with the ML framework deployed to train the RF models. Section IV-B presents a fine-grained analysis of each kernel to determine limits and benefits at the memory and latency level; this analysis includes a breakdown of platform-dependent compute non-idealities and storage usage. Section IV-C provides an in-depth comparison between the alternative DT kernels, pointing out the performance and memory improvements compared to the Naive baseline.

A. Setup

The experiments have been conducted on PULPissimo [27], an open-source System-on-Chip (SoC) integrating a 32-bit RISCV-based processor tailored for near-sensor computing applications. The RISCY core features a 4-stage in-order single-issue pipeline supporting the extended RV32IMFCXpulpV2 ISA, which delivers highly energy-efficient custom ML- and DSP-centric instructions. In particular, our optimized kernels mainly benefit from hardware loop (p.setup), post-incrementing load/store (p.lw/p.sw), immediate branching (p.bnei), and MAC (p.mac) instructions. The support for hardware loops includes dedicated hardware blocks to reduce the control code overhead, removing the instructions to update the counter register and related branch stalls. Post-incrementing memory accesses perform the automatic update of address registers by a constant offset, reducing the overhead of pointer arithmetic. While MAC operations perform multiplication and addition in a single clock cycle, immediate operand in branching instructions removes the additional compare operation. To reduce energy and area consumption, PULPissimo does not include data caches but features a 520 kB L2 multi-bank memory and a ROM storing the boot code. In our setup, the RI5CY core fetches instructions from a 4 kB I$ providing optimal performances and energy efficiency.

Algorithms and tests are implemented in C language and compiled using the open-source PULP GCC toolchain supporting the RV32IMFCXpulpV2 ISA. The high-level machine-independent description enables deploying the functions on alternative architectural targets without any ISA constraint. After optimizing the DT kernels to maximize the performance on the baseline RV32IMFC ISA, we leveraged the platform-specific XpulpV2 extension to reduce the CPI. We then fine-tuned the DTs design to optimize the performance on PULPissimo by leveraging hardware-agnostic optimizations and a set of platform-specific features. To evaluate the performance, we emulated the PULPissimo platform with a hardware emulator running on a Xilinx UltraScale+ VCU118 FPGA setting the clock frequency to 400 MHz. The HW design also includes a set of Special-Purpose Registers (SPRs) to keep track of hardware-related events at the core level mandatory for enabling fine-grained performance analysis. In our experiment, we report events related to pipeline non-idealities in memory accesses, primarily load stalls (LD STALL), misaligned loads (MIS-LD), and instruction cache misses (I$ MISS). While TKN-BR refers to control hazards due to taken branches, ALU counts the number of arithmetic and bitwise instructions executed by the core. The FPGA emulator is cycle-accurate but cannot provide a power estimation representative of an ASIC solution. For this reason, the power figures used to evaluate the energy consumption of the algorithms are derived from post place-&-route models in 22 nm FDX technology.

TABLE II: Dataset Properties

	Dimension	Classes	Samples
Vehicle	18	4	1000
MFeat	216	10	2000

To characterize the performance, we have targeted two standard datasets representing general IoT applications, whose properties are summarized in Table II. In Section IV-B, we leverage the Vehicle dataset that was designed to classify car brands based on a set of scale-independent features extracted using classical moment- and heuristic-based measures. By featuring about 1000 instances and 18 features, the dataset supports recognizing four vehicle classes and represents a potential real-world IoT scenario. We also targeted the MFeat database in Section IV-C, a medium-sized dataset consisting of features extracted from hand-written digits ('0'-'9'). The 2K samples present digit profile correlations and morphological attributes leading to a 216-dimensional features space and ten supported classes. Training has been performed using the Scikit-Learn ML framework, also relying on its front-end to support the code generation of model parameters and structures. RF models feature the alternative DT designs but share a standard software module used for majority voting, which combines all the predictions and retrieves the output class. We configured the RF model to be populated by 16 DTs, which are enough to fit the on-chip memory of the RISCV-based platform. However, the L2 memory size poses no strict limitation on the model dimension since the platform can be easily extended with an external memory level (L3). By placing data into the off-chip memory, a double-buffering mechanism becomes mandatory to overlap L2-L3 DMA transfers with kernel processing optimally. This can be implemented as a wrapper around the algorithms, and we verified that it does not impact our performance measurements.

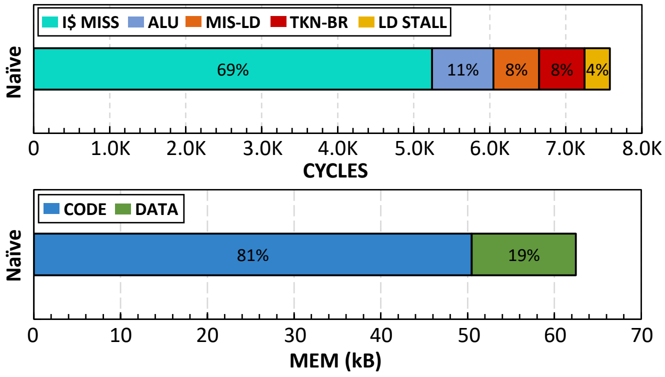


Fig. 1: Naive Kernel Resources

B. Kernels Analysis

In Figure 1, we reported the computing time and memory footprint required by the Naive DT kernel when running on PULPissimo. By unrolling the overall tree structure, the DT Naive approach demands about 7.5 kCycles largely bounded by I$ MISS (69%). The massive code size accounts for 81% of the total memory footprint (62.5 kB), leading to a high pressure onto the I$ and a related high miss rate. This effect causes a degradation of the CPI to 7.16. As shown in Listing 10, traversing a node involves evaluating a RISC-V assembly routine accounting for 5 Instr/Dec. Furthermore, this approach requires 3 read operations per decision to load the floating-point immediate and input feature into registers.

Listing 8: DT-Rec+US

```
0x4c: p.lw   a4,4(a1!)
0x50: lbu    a5,0(a4)
0x54: lw     a3,4(a4)
0x56: slli   a5,a5,0x2
0x58: p.lw   a5,a5(a0)
0x5c: fle.s  a5,a5,a3
0x60: beqz   a5,0x7a
0x62: lw     a5,8(a4)
0x64: beqz   a5,0xc0
```

Listing 9: DT-Rec+PS

```
0x4c: p.lw   a4,4(a1!)
0x50: lbu    a5,0(a4)
0x54: lw     a3,1(a4)
0x56: slli   a5,a5,0x2
0x58: p.lw   a5,a5(a0)
0x5c: fle.s  a5,a5,a3
0x60: beqz   a5,0x7a
0x62: lw     a5,5(a4)
0x64: beqz   a5,0xc0
```

Listing 10: Naive

```
0x80: lui    a3,0x1c001
0x84: lw     t4,36(a0)
0x88: lw     a3,-12(a3)
0x8c: fle.s  a3,t4,a3
0x90: beqz   a3,0xde
```

By deploying a struct-based approach, the DT2Loop kernel executes in 2.58 kCycles, as reported in Figure 2. Due to the prevailing ALU instructions (43%) over the computing

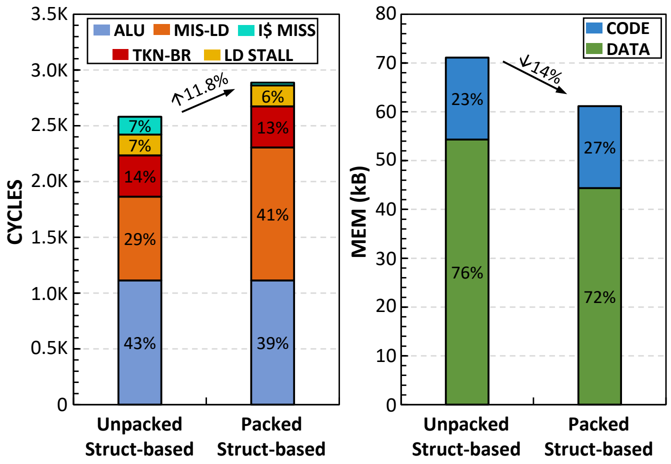


Fig. 2: DT-Loop Kernel Resources

time and pipeline factors, the DT-Loop method increases CPU usage by delivering a 1.3 CPI. However, the GCC toolchain handles the struct-based tree representation by adding a proper padding to align unpacked structure members to 4 Bytes address boundary. Consequently, storing a DT-Loop node requires 16 Bytes regardless of the effective variable range leading to 71.3 KB memory footprint largely owed to data (76%). To trade computing time for memory saving, we evaluate an optimized kernel version with *packed* fields; this solution avoids the adoption of padding to enforce memory alignment without increasing the memory size. Such an approach leads to an overall model storage requirement of 61.3 kB due to an 18% data memory reduction while leaving constant the code size. In this regard, no memory alignment in structure members involves performing 440 additional MIS-LD, resulting in a slow-down to 2.89 kCycles. The SW optimization introduced is platform-specific since only architectures supporting misaligned memory accesses can leverage it. Lastly, we show in Listings 11 and 12 that both kernel versions require 13 Instr/Dec, but the unpacked version presents unaligned offsets in 3 memory read operations.

Listing 11: DT-Loop+US

```
0x66: lbu    a5,0(a4)
0x6a: lw     a7,8(a4)
0x6e: slli   a5,a5,0x2
0x70: p.lw   a5,a5(a0)
0x74: lw     a6,12(a4)
0x78: fle.s  a5,a5,a3
0x7c: andi   a5,a5,255
0x80: xori   a3,a5,1
0x84: mul    a4,a5,a7
0x88: p.mac  a4,a3,a6
0x8c: lw     a3,4(a4)
0x8e: feq.s  a5,a3,t1
0x92: beqz   a5,0x66
```

Listing 12: DT-Loop+PS

```
0x66: lbu    a5,0(a4)
0x6a: lw     a7,5(a4)
0x6e: slli   a5,a5,0x2
0x70: p.lw   a5,a5(a0)
0x74: lw     a6,9(a4)
0x78: fle.s  a5,a5,a3
0x7c: andi   a5,a5,255
0x80: xori   a3,a5,1
0x84: mul    a4,a5,a7
0x88: p.mac  a4,a3,a6
0x8c: lw     a3,1(a4)
0x8e: feq.s  a5,a3,t1
0x92: beqz   a5,0x66
```

As depicted in Figure 3, the *unpacked* struct-based DT-Rec kernel reduces memory usage to 57.9 kB due to merging leaf nodes into parent decision nodes while taking 2.03 kCycles per inference. The high usage of conditional statements significantly impacts control hazards (TKN-BR), equal to 22%; together with increased LD STALL (15%), this effect contributes to a suboptimal 1.44 CPI. The recursive design would also imply a high stack usage by adding new stack frames for each function call, requiring $O(H)$ space complexity. However, since we do not perform any computations on the returned value, the kernel represents a tail-recursive routine with no more need to preserve the stack frame for that call. The GCC toolchain supports a tail-call optimization pass enabling the removal of the recursion memory overhead and forcing the reuse of the same stack frame, decreasing the space complexity to $O(1)$. This improvement is unrelated to the adopted architecture but depends on the support provided by the specific toolchain deployed to compile the source code. To further reduce the algorithm memory footprint to 42.99 kB, we support a *packed struct* version that decreases data storage by 36.7%. Due to the lacking of memory alignment, the core performs 48.2% extra MIS-LD to access structure fields, inducing a latency increase to 2.32 kCycles. Such slow-down can be observed comparing Listing 8 and 9, where we reported the assembly routine of both versions executed when traversing a node. The kernels evaluate the same 9 Instr/Dec, including 5 memory access operations, of which 2 features unaligned offsets in the packed struct-based version.

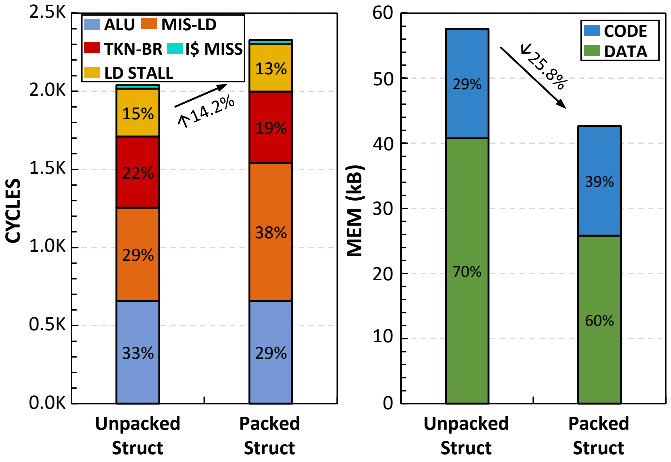


Fig. 3: DT-Rec Kernel Resources

Adopting an array-based tree representation permits storing node attributes in separate data structures, leveraging the effective variable range to save memory space without increasing MIS-LD. Figure 4 shows the computing time and memory footprint breakdown of three DT-Arr kernel versions developed trading storage demands for inference acceleration. The *baseline* design fully reaches an optimal memory reduction requiring 41.25 kB to store the overall RF model where data storage accounts for only 59%. Running an inference on such kernel demands 2.57 kCycles mainly consisting of ALU operations (50%), delivering a high CPU utilization featured by a nearly optimal CPI of 1.19. To reduce the number of instructions per decision node and improve the runtime, we apply two platform-agnostic optimizations on the DT-Arr kernel. As shown in Listing 13, the Baseline version involves executing 14 Instr/Dec containing 5 memory accesses but also shifting operations to extend array indexes to offset addresses. Since array indexes are already known at compile-time, we can precompute off-line offset addresses by multiplying indexes with data type byte-width. The *Shift-Less* kernel version saves a shifting operation lowering to 13 Instr/Dec by directly expli-

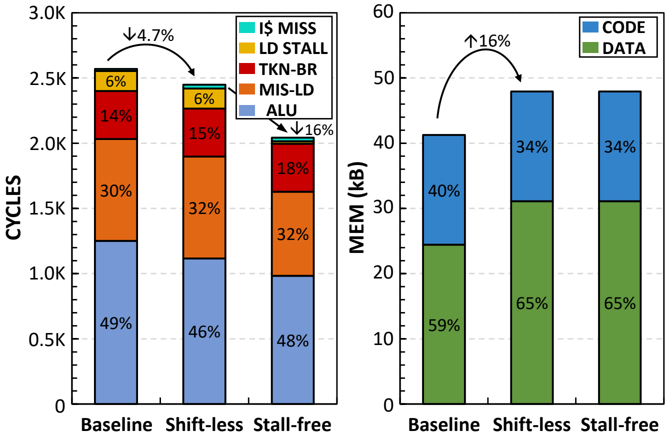


Fig. 4: DT-Arr Kernel Resources

cating the `base + offset` addressing mode, as presented in Listing 14. This code optimization diminishes the computing time to 2.45 kCycles due to a 10.8% decrease of ALU instructions but at the expense of a 16% memory increment resulting from the need to use longer data types for extended indexes. This design still relies on reading both child nodes, demanding 5 memory read operations per decision and featuring a high count of LD STALL covering 6% of the runtime. For that purpose, the *Stall-Free* version aims at decreasing stalls by reading both child nodes with a single memory access and retrieving the effective child through the comparison result. Listing 15 shows the reduction to 11 Instr/Dec achieved due to removing 1 memory access and adopting the new procedure to pick up the child node. The runtime optimization allows reaching a near-zero LD STALL execution with substantial decreases in ALU operations (11.9%) and MIS-LD (17.4%), leading to a 2.04 kCycles inference time featured by a near-ideal 1.16 CPI. In addition, such performance improvement comes at the expense of no increase in memory footprint.

Listing 13: DT-Arr Baseline

```
0x30: slli   a6,a6,0x2
0x32: slli   a5,a5,0x2
0x34: p.lw   t1,a6(t6)
0x38: p.lw   a5,a5(a0)
0x3c: add    a6,a6,a7
0x3e: fle.s  a5,a5,t1
0x42: lhu    t4,0(a6)
0x46: lhu    t3,2(a6)
0x4a: andi   a5,a5,255
0x4e: xori   t1,a5,1
0x52: mul    a6,t4,a5
0x56: p.mac  a6,t3,t1
0x5a: p.lb   a5,a6(t5)
0x5e: p.bnei a5,-8,30
```

Listing 14: DT-Arr Shift-Less

```
0x3c: p.lw   a6,a7(t6)
0x40: p.lw   a5,a5(a0)
0x44: add    a7,a7,t1
0x46: fle.s  a5,a5,a6
0x4a: lhu    t4,0(a7)
0x4e: lhu    t3,2(a7)
0x52: andi   a5,a5,255
0x56: xori   a6,a5,1
0x5a: mul    a7,t4,a5
0x5e: p.mac  a7,t3,a6
0x62: srli   a6,a7,0x2
0x66: p.lb   a5,a6(t5)
0x6a: p.bnei a5,-8,3c
```

Listing 15: DT-Arr Stall-Free

```
0x34: p.lw   a7,a6(t4)
0x38: p.lw   a5,a5(a0)
0x3c: p.lw   a6,a6(t1)
0x40: flt.s  a5,a7,a5
0x44: slli   a5,a5,0x4
0x46: srl    a6,a6,a5
0x4a: srli   a5,a6,0x2
0x4e: and    a5,a5,t5
0x52: p.lb   a5,a5(t3)
0x56: and    a6,a6,t6
0x5a: p.bnei a5,-8,34
```

C. *Kernels Comparison*

In this section, we propose a comparison between the alternative DT kernel designs. Moreover, we pinpoint the optimal kernel solution to enable inference at the edge when targeting memory-constrained MCUs executing latency-sensitive applications.

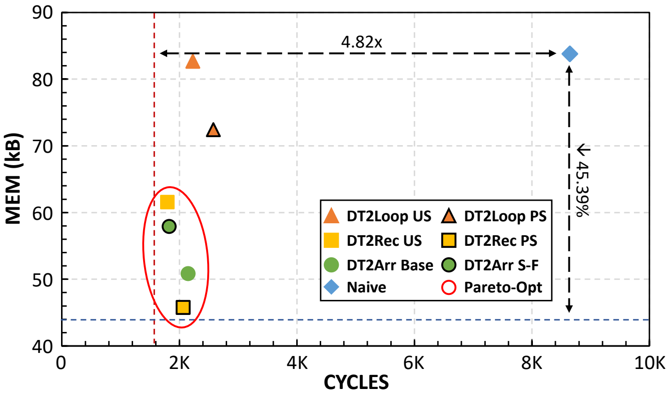


Fig. 5: Kernels Mem-Cycles Trade-off Comparison. Abbreviations: Unpacked Struct (US), Packed Struct (PS), Baseline (Base), Stall-Free (S-F), Pareto-Optimal (Pareto-Opt).

Figure 5 represents the compute and storage resources required to execute the alternative DT designs on Pulpissimo. At the upper-right chart corner, the Naive method represents the most resource-demanding design with 83.8 kB memory usage and 8.6 kCycles per inference, as reported in Table IV. Moreover, the kernel offers the lowest CPU performance by delivering a very poor 6.03 CPI that reveals the massive code size exceeding MCUs I$ limited capability as the bounding factor. Adopting the newly introduced DT kernels moves to less-demanding inference constraints, enabling efficient execution on even highly resource-constrained edge devices. We highlighted in red the Pareto-Optimal solutions that allow reaching speedups ranging from 4.03× to 4.82× while achieving between 26.52% and 45.39% memory footprint reduction. By taking about 1.8 kCycles per inference, the unpacked struct-based DT-Rec and stall-free DT-Arr kernels lead to about a 4.8× speedup compared to the Naive execution. Due to the runtime optimizations to minimize pipeline non-idealities, the stall-free DT-Arr method also delivers the highest core utilization featuring a 1.17 CPI. Although such designs do not provide the most memory-efficient DT representation, they deliver a storage utilization decrease between 26% and 31% of the baseline. To push the memory occupation improvement even further, the adoption of a packed struct-based approach on top of the DT-Rec kernel becomes mandatory. The improved tree representation decreases the storage demand by about 45.39% regarding the Naive memory usage, requiring only 45.8 kB. Due to the reduced Instr/Dec previously shown, the kernel still reaches high execution performance demanding 2.07 kCycles per inference, which results in a 4.19× speedup.

Our optimized DT kernels reach ultra-low-latency RF inference on PULPissimo by taking in between 4.49 and 6.45 μs instead of the 21 μs required by the Naive version. Such runtime improvements lead to almost a 223 kInf/s throughput with the unpacked struct DT-Rec approach, while 46 Inf/s bounds the peak processing rate of the Naive method. Concerning the energy performance, the optimally designed kernels reduce the energy usage dramatically, down to 15.62 pJ to execute the RF model on the RISCV-based processor.

To finalize our study on the alternative DT designs, we summarise in Table III the major outcomes of each kernel reporting the demanded resources in the shape of trends. While the ↑ symbol represents a workload highly fittable into resource-constrained MCUs, the ↓ symbol highlights a tendency highly likely to make unfeasible the adoption of the model on-the-edge with different scenarios.

TABLE III: Kernel Resources Demand Trend

	Memory	Compute
Naïve	↓↓	↓↓
DT-Loop + US	↓↓	↑↑
DT-Loop + PS	↓	↑
DT-Rec + US	↑	↑↑↑
DT-Rec + PS	↑↑	↑↑
DT-Arr Base	↑↑	↑↑
DT-Arr Stall-Free	↑	↑↑↑

Table V reports the compute and memory resource costs extrapolated from the previous experiments for each kernel. While Mem/Dec, N, and Bytes/Node costs are kernel-related, Instr/Dec and Energy/Dec are algorithmic- and platform-

TABLE IV: DT Kernels Statistics and Measurements Comparison

Kernel	Cycles	Instr	CPI	SpeedUp	Mem (kB)	Mem Red. (%)	Latency (μs)	Throughput (kInf/s)	Energy (pJ)
Naive	8.64k	1.43k	6.03	-	83.81	-	21.61	46.27	75.20
DT-Loop + US	2.23k	1.74k	1.28	3.88×	82.70	1.32	5.567	179.6	19.38
DT-Loop + PS	2.58k	1.74k	1.48	3.35×	72.45	13.56	6.447	155.1	22.44
DT-Rec + US	1.80k	1.21k	1.48	4.82×	61.58	26.52	4.487	222.8	15.62
DT-Rec + PS	2.07k	1.21k	1.70	4.19×	45.77	45.39	5.162	193.7	17.97
DT-Arr base	2.15k	1.77k	1.21	4.03×	50.86	39.32	5.375	186.5	18.71
DT-Arr S-F	1.83k	1.56k	1.17	4.73×	57.93	30.88	4.575	218.6	15.92

dependent since the ISA support might change the instruction count and the energy performance. Although presenting the lowest Instr/Dec and Mem/Dec count, the Naive kernel high pressure onto the I$ results in an expensive 0.745 pJ/Dec. By adding a few instructions and memory read operations, SW optimizations decrease the architectural factors reaching about 0.15 pJ/Dec with the adoption of DT-Rec and DT-Arr designs. Regarding the memory-related metrics, the Naive approach requires storing five 32-bit instructions and a single-precision floating-point threshold per decision node, totaling 24 Bytes/Node. Moving towards DT-Loop and DT-Arr designs reduces the node memory to 16 Bytes/Node and beyond depending on the node attributes range. Instead, the DT-Rec approach decreases the overall stored nodes to $2^H - 1$ against the $2^{H+1} - 1$ demanded by the other designs.

The final goal of this analysis is to select proper DT kernels to optimally fit resource-scarce devices resources depending on the application constraints. Latency-sensitive IoT tasks will target inference acceleration to meet the timing requirements, preferring faster kernels such as Stall-Free DT-ARR and DT-Rec standard versions. Instead, applications running on storage-limited end-nodes will favour memory-efficient kernels such as standard DT-Arr and DT-Rec, with particular attention to the packed struct option. Such optimization enables a large memory reduction at the expense of a severe MIS-LD increment, which negatively affects the computing time.

TABLE V: Kernels Compute and Memory Resource Costs. Abbreviations: DT-Loop (DT-L), DT-Rec (DT-R), DT-Arr (DT-A).

	Instr/Dec	Mem/Dec	pJ/Dec	Nodes	Byte/Node
Naïve	5	3	0.745	$2^{H+1}-1$	24
DT-L+US	13	5	0.192	$2^{H+1}-1$	16
DT-L+PS	13	5	0.222	$2^{H+1}-1$	≤16
DT-R+US	8	5	0.155	$2^{H}-1$	24
DT-R+PS	8	5	0.178	$2^{H}-1$	≤24
DT-A Base	14	5	0.185	$2^{H+1}-1$	≤16
DT-A S-F	11	4	0.158	$2^{H+1}-1$	≤16

V. Conclusion

The paper studies the design of DT kernels to optimize the deployment of RF inference on highly memory-constrained MCUs executing latency-sensitive applications. After introducing the new designs and their complexity along with the widely adopted Naive method, we evaluated the performance on a RISC-V platform. This fine-grained analysis determines limits and benefits at the memory- and latency-level while also breaking down platform-dependent non-idealities and storage usage. By adopting runtime optimizations to reduce pipeline factors and memory-efficient node representations to decrease storage usage, we improve resources requirement revealing significant memory-latency trade-offs. Furthermore, we propose an overall DT kernel comparison demonstrating that deploying our designs can reach a 4.82x speedup and 83.12% memory reduction with respect to the widely-used Naive method. Lastly, we summarize the major outcomes of our work supported by time and memory costs to determine the optimal kernel fitting the application constraints.

Future work will include exploring extreme quantization techniques pushing even towards sub-byte representations to decrease RF memory usage further. Furthermore, we will investigate the adoption of dedicated RISCV ISA extensions to improve inference on MCUs without relying on a full-fledged accelerator.

References

[1] O. Chapelle and Y. Chang, "Yahoo! learning to rank challenge overview," in *Proceedings of the 2010 International Conference on Yahoo! Learning to Rank Challenge - Volume 14*, YLRC'10, p. 1–24, JMLR.org, 2010.

[2] D. Sorokina and E. Cantu-Paz, "Amazon search: The joy of ranking products," in *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, (New York, NY, USA), p. 459–460, Association for Computing Machinery, 2016.

[3] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. n. Candela, "Practical lessons from predicting clicks on ads at facebook," in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14, (New York, NY, USA), p. 1–9, Association for Computing Machinery, 2014.

[4] I. Segalovich, "Machine learning in search quality at yandex," *Invited Talk, SIGIR*, vol. 125, 2010.

[5] A. Adadi and M. Berrada, "Peeking inside the black-box: a survey on explainable artificial intelligence (xai)," *IEEE access*, vol. 6, pp. 52138–52160, 2018.

[6] D. Gunning and D. Aha, "Darpa's explainable artificial intelligence (xai) program," *AI Magazine*, vol. 40, no. 2, pp. 44–58, 2019.

[7] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee, "From local explanations to global understanding with explainable ai for trees," *Nature machine intelligence*, vol. 2, no. 1, pp. 56–67, 2020.

[8] Z.-H. Zhou and J. Feng, "Deep forest: Towards an alternative to deep neural networks," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 3553–3559, 2017.

[9] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[10] B. Gupta, "Analysis of iot concept applications: Smart home perspective," in *Future Access Enablers for Ubiquitous and Intelligent Infrastructures: 5th EAI International Conference, FABULOUS 2021, Virtual Event, May 6–7, 2021, Proceedings*, vol. 382, p. 167, Springer Nature, 2021.

[11] Cisco, "Global Cloud Index: Forecast and Methodology, 2016–2021," 2016.

[12] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *2013 Proceedings IEEE INFOCOM*, pp. 1285–1293, 2013.

[13] R. Sanchez-Iborra and A. F. Skarmeta, "TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities," *IEEE Circuits and Systems Magazine*, vol. 20, no. 3, pp. 4–18, 2020.

[14] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, *et al.*, "Benchmarking TinyML systems: Challenges and direction," *arXiv preprint arXiv:2003.04821*, 2020.

[15] TinyML foundation, "TinyML reasearch community."

[16] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the Internet of Things," *IEEE access*, vol. 6, pp. 6900–6919, 2017.

[17] T. M. Ingolfsson, A. Cossettini, X. Wang, E. Tabanelli, G. Tagliavini, P. Ryvlin, and L. Benini, "Towards long-term non-invasive monitoring for epilepsy via wearable eeg devices," *arXiv preprint arXiv:2106.08008*, 2021.

[18] E. Tabanelli, D. Brunelli, A. Acquaviva, and L. Benini, "Trimming feature extraction and inference for mcu-based edge nilm: a systematic approach," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 2, pp. 943–952, 2021.

[19] V. C. Pezoulas, T. P. Exarchos, and D. I. Fotiadis, "Chapter 2 - types and sources of medical and other related data," in *Medical Data Sharing, Harmonization and Analytics*, pp. 19–65, Academic Press, 2020.

[20] T. Bernard, M. Verbunt, G. vom Bögel, and T. Wellmann, "Non-intrusive load monitoring (nilm): Unsupervised machine learning and feature fusion: Energy management for private and industrial applications," in *2018 international conference on smart grid and clean energy technologies (ICSGCE)*, pp. 174–180, IEEE, 2018.

[21] M. T. Yazici, S. Basurra, and M. M. Gaber, "Edge machine learning: Enabling smart internet of things applications," *Big data and cognitive computing*, vol. 2, no. 3, p. 26, 2018.

[22] E. Tabanelli, G. Tagliavini, and L. Benini, "Dnn is not all you need: Parallelizing non-neural ml algorithms on ultra-low-power iot processors," *arXiv preprint arXiv:2107.09448*, 2021.

[23] B. Sudharsan, P. Patel, J. G. Breslin, and M. I. Ali, "Ultra-fast machine learning classifier execution on iot devices without sram consumption," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 316–319, 2021.

[24] F. Daghero, A. Burrello, C. Xie, L. Benini, A. Calimera, E. Macii, M. Poncino, and D. J. Pagliari, "Adaptive random forests for energy-efficient inference on microcontrollers," in *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 1–6, 2021.

[25] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the automata processor," in *International Conference on High Performance Computing*, pp. 200–218, Springer, 2016.

[26] N. Asadi, J. Lin, and A. P. De Vries, "Runtime optimizations for tree-based machine learning models," *IEEE transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2281–2292, 2013.

[27] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an ultra-low-power pulpissimo soc in 22nm fdx," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pp. 1–3, 2018.

[28] 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, no. 1, pp. 209–222, 2018.

[29] D. Watanabe, Y. Yano, S. Izumi, H. Kawaguchi, K. Takeuchi, T. Hiramoto, S. Iwai, M. Murakata, and M. Yoshimoto, "An architectural study for inference coprocessor core at the edge in iot sensing," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 305–309, 2020.

[30] S. Zhao, S. Chen, H. Yang, F. Wang, and Z. Wei, "Rf-risa: A novel flexible random forest accelerator based on fpga," *Journal of Parallel and Distributed Computing*, vol. 157, pp. 220–232, 2021.

[31] A. Alcolea and J. Resano, "Fpga accelerator for gradient boosting decision trees," *Electronics*, vol. 10, no. 3, 2021.

[32] F. Lettich, C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "Parallel traversal of large ensembles of decision trees," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2075–2089, 2019.

[33] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga?," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 232–239, 2012.

[34] H. Nakahara, A. Jinguji, S. Sato, and T. Sasao, "A random forest using a multi-valued decision diagram on an fpga," in *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, pp. 266–271, 2017.

[35] X. Yin, F. Müller, A. F. Laguna, C. Li, W. Ye, Q. Huang, Q. Zhang, Z. Shi, M. Lederer, N. Laleni, *et al.*, "Deep random forest with ferroelectric analog content addressable memory," *arXiv preprint arXiv:2110.02495*, 2021.

[36] M. Zamboni, M. Graziano, P. D. G. Turvani, and D. Uccellatore, "Logic-in-memory implementation of random forest algorithm," *Politecnico Torino*, 2018.

[37] M. Kang, S. K. Gonugondla, S. Lim, and N. R. Shanbhag, "A 19.4-nj/decision, 364-k decisions/s, in-memory random forest multi-class inference accelerator," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 7, pp. 2126–2135, 2018.

[38] G. Pedretti, C. E. Graves, S. Serebryakov, R. Mao, X. Sheng, M. Foltin, C. Li, and J. P. Strachan, "Tree-based machine learning performed in-memory with memristive analog cam," *Nature communications*, vol. 12, no. 1, pp. 1–10, 2021.

[39] Y. Idelbayev, A. Zharmagambetov, M. Gabidolla, and M. A. Carreira-Perpinan, "Faster neural net inference via forests of sparse oblique decision trees," 2022.

[40] A. Krishnakumar and U. Y. Ogras, "Performance analysis and optimization of decision tree classifiers on embedded devices: Work-in-progress," in *Proceedings of the 2021 International Conference on Embedded Software*, EMSOFT '21, (New York, NY, USA), p. 37–38, Association for Computing Machinery, 2021.

[41] L. T. da Silva, V. M. Souza, and G. E. Batista, "Embml tool: Supporting the use of supervised learning algorithms in low-cost embedded systems," in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 1633–1637, IEEE, 2019.

[42] Eloquent Arduino blog, "MicroML."

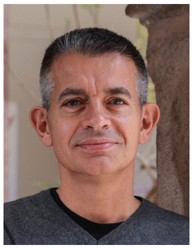
[43] D. Morawiec, "sklearn-porter." Transpile trained scikit-learn estimators to C, Java, JavaScript and others.

[44] J. Nordby, "emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices," 2019.

[45] H. Cho and M. Li, "Treelite: toolbox for decision tree deployment," in *Proc. Conf. Syst. Mach. Learn.(SysML)*, 2018.

Enrico Tabanelli received the M.S. (cum laude) degree in electronics engineering from the University of Bologna, Bologna, Italy, in 2019. He is currently a third-year Ph.D. at the Department of Electrical, Electronic and Information Engineering (DEI), University of Bologna, Bologna. His research interests lie in the area of Non-Intrusive Load Monitoring, parallelization of Machine Learning workloads on multi-core systems, and design of software stacks for emerging computing architectures.

Giuseppe Tagliavini received the Ph.D. degree in electronic engineering from the University of Bologna, Bologna, Italy, in 2017. He is currently an Assistant Professor with the Department of Computer Science and Engineering (DISI) at the University of Bologna. He has co-authored over 40 papers in international conferences and journals. His research interests include code analysis and transformation through compiler extensions, HW/SW co-design of programmable accelerators, machine learning algorithms and design tools for resource-constrained IoT devices.



Luca Benini (Fellow IEEE) holds the chair of digital Circuits and systems at ETHZ and is Full Professor at the University of Bologna. Dr.Benini's research interests are in energy-efficient parallel computing systems, smart sensing micro-systems and machine learning hardware. He has published more than 1000 peer-reviewed papers and five books. He is a Fellow of the ACM and a member of the Academia Europaea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg award, the 2020 EDAA achievement Award and the 2020 ACM/IEEE A. Richard Newton Award.