

Code Generation from Regression Trees for Microsecond-Scale Decisions in Operating Systems

Birte Friesel
birte.friesel@uos.de
Universität Osnabrück
Osnabrück, Germany

Marcel Lütke Dreimann
marcel.luetkedreimann@uos.de
Universität Osnabrück
Osnabrück, Germany

Olaf Spinczyk
olaf@uos.de
Universität Osnabrück
Osnabrück, Germany

Abstract

In today's world of heterogeneous server hardware, deciding on suitable task and data placements is a far from trivial undertaking. Depending on system load and application behaviour, some compute and memory assignments improve performance, while others impair it. Yet, these increasingly complex decisions must be made quickly: spending 5 ms just to decide on a placement that reduces latency by merely 2 ms is a net loss. So far, developers have resorted to hand-crafted heuristics for this task. While these are founded in real-world experience, they are hard to reason about, hard to adjust, and rarely free from bugs or inefficiencies. In our opinion, a transition towards model-guided placement decisions is due – i.e., using machine learning models to predict compute/data placement from system status and application requirements. To this end, we examine the suitability of regression trees for fast (nano- to microsecond-scale) runtime decisions within operating systems. We examine three types of trees / forests with three different code generation methods, and show that, depending on tree complexity and data type, median latency is 23 ns to 79 μ s per placement decision. By using C++ template expansion to transform entire trees into equivalent machine code, we are able to reduce inference latency by up to an order of magnitude compared to conventional approaches, while (for 8-bit integer data) *also* halving the code size. At the same time, trees are compact and can easily be generated from, e.g., Python-based machine learning models.

Keywords: Decision Trees, Placement Strategies, Templates, C++, Code Generation, Executable Data Structures

ACM Reference Format:

Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. 2026. Code Generation from Regression Trees for Microsecond-Scale Decisions in Operating Systems. In *Workshop on Programming Languages and Operating Systems (PLOS '26)*, September 29–October 2, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3831586.3838141>



This work is licensed under a Creative Commons Attribution 4.0 International License.

PLOS '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2877-8/26/09

<https://doi.org/10.1145/3831586.3838141>

1 Introduction

Today's servers can combine a variety of heterogeneous hardware components. For instance, compute kernels can be scheduled on CPU cores, GPUs, FPGAs, or near-memory-computing (NMC) devices. Corresponding data can be placed on conventional DDR RAM, High-Bandwidth Memory (HBM), NMC-capable memory or CXL memory extensions [10].

As such, schedulers no longer just have to decide *which* task to run next, but also *where* to place it. Similarly, memory allocators must not just find *any* sufficiently large range of free memory, but also decide *where* to start looking for it. At the same time, there is no single best allocation strategy: optimal compute and data placement depends on application behaviour, and established heuristics such as using HBM as cache for DDR RAM or always preferring NUMA-local memory can, in fact, be counter-productive [11].

Runtime placement decisions must also be *fast* – otherwise, any performance benefits gained from better placement would be cancelled out by the decision algorithm's latency. As such, predicting the performance of all valid task/data placements and then choosing the best one is out of the question. Such an approach only works in *offline* scheduling scenarios, and even then, it is an NP-hard problem [21].

So far, runtime schedulers and memory allocators typically rely on hand-crafted heuristics in order to achieve fast (nano- to microsecond-scale) and hopefully good placement decisions. They cannot be easily adapted to system-specific hardware topologies, and may hide bugs or suffer from bad performance in pathological corner cases [19].

We envision operating systems that are less reliant on such hand-crafted and error-prone heuristics, and instead use benchmarks and machine learning models to optimize placement decisions. Specifically, given an input vector $\vec{x}$ that describes application kernel requirements (e.g., data dependencies and memory access patterns) and current system utilization, a set of models should directly predict suitable placements $f_{\text{compute}}(\vec{x})$ and $f_{\text{data}}(\vec{x})$ for tasks and data objects. Note that this is an orthogonal extension to the scheduler: the models *only* decide on placement, not *which* task to place.

Such a paradigm shift would have far-reaching ramifications that exceed the scope of this paper. Here, we only focus on one aspect of model-guided runtime decisions: are machine learning models fast enough for this approach to be viable in practice? Specifically, how and up to which level

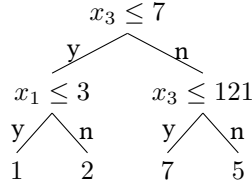


Figure 1. A regression tree with depth $d = 2$.

of model complexity can we obtain decisions within tens of nanoseconds to a few dozen microseconds? We do not cover learning algorithms or actual runtime placement decisions.

With these requirements, approaches such as (deep) neural networks are already out of the question. Depending on layer type and dimension, a single neuron requires dozens to thousands of math operations to compute its output value, and neural networks typically contain hundreds to millions of neurons. Instead, we focus on regression trees: a tree structure that takes a vector $\vec{x} \in \mathbb{R}^n$ (or $\mathbb{Z}^n$, when working with quantized models) as input, asks a single (typically boolean) question in each non-leaf node, and determines its output value (e.g., placement decision) from the leaf node that has been reached by the decision path determined by $\vec{x}$. Such a query on a regression tree is also referred to as *traversal* or *inference*. For instance, in Fig. 1, $f(1, 2, 3) = 1$ and $f(10, 20, 30) = 7$. Note that x_2 has no effect on its prediction.

We contribute three different methods for transforming regression trees and forests into efficient C/C++ source code, including an *executable data structure* approach that exploits compile-time C++ template expansion. In all three cases, we automatically serialize the regression tree into a C/C++ array; the traversal function itself is independent of tree structure or attributes. We also contribute a systematic analysis of inference latency and ROM usage, and recommendations for training hyper-parameters to obtain models that provide low-latency queries for runtime decisions in operating systems.

The next section introduces the three different types of regression trees that we examine in this paper. Section 3 covers our code generation approaches, which we evaluate in Section 4 and discuss in Section 5. After examining related work in Section 6, we conclude in Section 7.

2 Regression Tree Variants

At its core, a regression tree is a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ that transforms an n -dimensional input vector $\vec{x}$ to a single output value y . It is a specialization of the concept of *decision trees* that uses an arbitrary n -dimensional input domain and an arbitrary output domain [5]. Decision trees with categorical output domains are also known as a *classification trees*. As computers must encode all (possibly categorical) values as numbers, we only examine data structures that build upon regression trees ($\mathbb{R}^n \rightarrow \mathbb{R}$ or $\mathbb{Z}^n \rightarrow \mathbb{Z}$). All of our methods can also be applied to generic decision tree data structures.

In our case, the input vector contains aspects such as the linearity and locality of a task’s data accesses (which can be obtained, e.g., via profiling), the current availability and utilization of compute and memory resources, and the placement of related tasks and data objects. Note that transformation of system status and application behaviour into such vectors as well as learning algorithms are out of scope here. We solely focus on the suitability of regression trees in terms of memory and latency requirements.

We examine three different flavours of regression trees: conventional regression trees (RT), regression model trees (RMT), and regression forests (RF).

2.1 Conventional Regression Trees

Our conventional regression tree (RT) models utilize the CART (“Classification and Regression Tree”) data structure and learning algorithm as proposed by Breiman et al. [5]. Each non-leaf node holds a query “ $x_i \leq z$ ” for some *feature* i and *threshold* z , and each leaf node holds a prediction $y \in \mathbb{R}$. The inference algorithm starts at the root, compares the value of input feature x_i to the node’s threshold z , and recursively continues in the corresponding sub-tree until it reaches a leaf. As such, for a tree with depth d , it returns a prediction after no more than d comparisons – see Fig. 1 for an example.

Nowadays, regression trees are a widely-used part of the machine learning toolbox, and have also been shown to perform well in the context of performance predictions for configurable software systems [16]. In fact, in this domain, research indicates that their prediction accuracy is comparable to neural networks, and far exceeds the accuracy of non-tree regression learners [1].

From a computational complexity point of view, an RT is defined by its input dimension n , depth d , and number of nodes m . Note that regression trees are not necessarily balanced, so d and m must be captured separately. We can only state that $2d + 1 \leq m \leq 2^{d+1} - 1$.

2.2 Regression Model Trees

Regression trees are especially well-suited for boolean features ($x_i \in \{0, 1\}$) – when dealing with a numeric input domain, they are limited by their inability to inter- or extrapolate beyond training data. Meanwhile, least-squares regression (e.g., $f(\vec{x}) = \beta_0 + \sum_{i=1}^n \beta_i x_i$) is well-suited for numeric features, but less so for boolean ones.

Regression model trees (RMT) aim to combine the best of both worlds [14]. Their decision nodes exclusively deal with boolean and *categorical* features (i.e., $x_i \in C_i \subset \mathbb{N}$, for example to indicate on which NUMA node a task or data object has been placed). In case of categorical features, they utilize *multi-way splits* where a single decision node has more than two sub-trees – one for each categorical value $z \in C_i$ encountered during training. Their leaf nodes do not hold constant values, but instead contain regression formulas that exclusively reference numeric features ($x_i \in \mathbb{R}$).

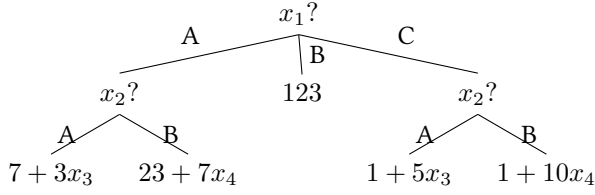


Figure 2. A regression model tree with two categorical and two numeric features.

For n_C boolean or categorical features and n_N numeric features, an RMT inference takes no more than n_C comparisons, and no more than n_N multiplications and additions. This gives a well-defined upper bound on inference latency.

Fig. 2 shows an example. Note that, while RMT also support non-linear functions, we do not consider those here. This helps ensure low latency operation, as no complex mathematical operations are involved. Additionally, it makes them compatible with linear model trees (LMT), which have also been shown to perform well when it comes to performance prediction of variable software and hardware systems [13].

An RMT is characterized by input dimensions n_C and n_D , depth d , maximum number of categories c (i.e., the maximum amount of elements in any set C_i), and number of nodes m .

2.3 Regression Forests

A regression forest (RF) is an *ensemble* $\mathcal{F} = \{f_1, \dots, f_K\}$ of K regression trees, where each tree is either learnt in isolation (“bagging”) or aims to correct mis-predictions made by other trees (“boosting”). Typically, each tree covers a random subset of the input domain, e.g., only one third of all n features. Again, this is a common machine learning approach, where multiple weak learners are combined to form a single model. In contrast to RMT, RF do not support categorical features or multi-way splits.

Predictions are typically calculated via $\mathcal{F}(\vec{x}) = \frac{1}{K} \sum_{i=1}^K f_i(\vec{x})$ (bagging) or $\mathcal{F}(\vec{x}) = \beta_0 + \sum_{i=1}^K f_i(\vec{x})$ (boosting). Regression forests are also known to perform well when it comes to performance prediction tasks in our target domain [1].

We examine boosted regression forests as trained by the *XGBoost* algorithm [6]. From an inference point of view, they are identical to bagged forests, except for a minor difference in how the sum of individual predictions is processed.

An RF is characterized by number of estimators K , input dimension n , maximum depth d across all trees, and total number of nodes m . We now move on to transforming these models into C/C++ source code for runtime inference.

3 Code Generation

The object-oriented text-book approach, where each node is represented as a separate instance of a decision or leaf node class, is out of the question: on a 64-bit system, this would require 16 Bytes per decision node just for pointers

```
struct node {
    const uint16_t rightChild;
    const uint8_t feat;
    const inout_t threshold;
};
```

Listing 1. Data structure for individual RT and RF nodes.

```
const struct node tree[7] = {
    {.rightChild = 4, .feat = 3, .threshold = 7}, // 0
    {.rightChild = 3, .feat = 1, .threshold = 3}, // 1
    {.rightChild = 0, .feat = 255, .threshold = 1}, // 2
    {.rightChild = 0, .feat = 255, .threshold = 2}, // 3
    {.rightChild = 6, .feat = 3, .threshold = 121}, // 4
    {.rightChild = 0, .feat = 255, .threshold = 7}, // 5
    {.rightChild = 0, .feat = 255, .threshold = 5}, // 6
};
```

Listing 2. C/C++ source code representation of the regression tree given in Fig. 1.

to sub-trees, and there would be no guarantees on spatial locality. In fact, even the Python libraries we utilize for RT and RF use a more concise, list-based internal representation.

Hence, for all code generation variants, we store each tree as a list of nodes in pre-order traversal. Each (non-leaf) decision node “ $x_i \leq z$ ” at index j holds a feature i , a threshold z , and the index of its right child, as shown in Listing 1. Thanks to pre-order traversal, the index of the left child is always $j + 1$, and thus not stored. Each leaf node encodes its output value in threshold, and sets `rightChild = 0`; `feat = UINT8_MAX` to indicate that it is a leaf node.

For example, Listing 2 encodes the regression tree given in Fig. 1. Note that this tree encoding is identical for all code generation methods, and the only part that depends on the actual regression tree (or RMT, or RF). The source code of the inference functions, which will be covered in the following sections, is independent of tree or forest structure.

We examine four `inout_t` definitions: *quantized* trees with 8-, 16-, or 32-bit signed integers, and conventional 32-bit float thresholds. For RMT, we split the input vector into an array of boolean / categorical values (`uint8_t`) and an array of numeric values (`inout_t`). Note that we assume that an input vector $\vec{x}$ holds no more than 255 features (i.e., $n < 256$) – support for more features can be obtained by switching to `uint16_t` `feat` and `UINT16_MAX`, respectively.

We assess three different types of code generation: plain C code that limits the amount of data-dependent branch instructions (*plainC*), C++ with a recursive traversal function embedded into `struct node` (*plainC++*), and C++ that uses template expansion to transform the entire tree or forest into nested branch statements (*templateC++*), i.e., into an *executable data structure* [20]. In all cases, we generate standard-compliant C/C++ code, and use gcc 15.3.0 with `-Os` (without custom patches or plugins) to compile it.

Due to limited space, we only cover code generation for regression trees. Regression forests extend those with a list of trees and a traversal function that takes the sum of each tree’s prediction, whereas regression model trees require

```

inout_t traverse(inout_t *vec) {
    uint16_t i = 0;
    while (tree[i].feat != 255) {
        bool cmp = vec[tree[i].feat] <= tree[i].threshold;
        i = cmp * (i + 1) + !cmp * tree[i].rightChild;
    }
    return tree[i].threshold;
}

```

Listing 3. Regression tree inference implemented in plainC.

```

00: mov    %rdi,%r8          ; r8 := &vec[0]
03: xor    %eax,%eax          ; accu := 0
05: lea    0x...(%rip),%rcx    ; counter := &tree[0]
0c: movzwl %ax,%esi           ; i := accu
0f: movzbl 0x2(%rcx,%rsi,4),%edx; data := tree[i].feat
14: mov    0x3(%rcx,%rsi,4),%dil; dest := tree[i].thresh
19: cmp     $0xff,%dil          ; tree[i].feat <=> 255
1c: je      2b                  ; if .feat == 255: return .threshold
1e: inc     %eax                ; else: accu += 1
20: cmp     %dil,(%r8,%rdx,1);vec[tree[i].feat] <=> .thresh
24: cmovg   (%rcx,%rsi,4),%ax;if ">": accu := tree[i].right
29: jmp     0c                  ;next loop iteration: goto 0c
2b: mov     %edi,%eax           ;accu := tree[i].threshold
2d: ret                          ;return value stored in accu

```

Listing 4. Machine code for 8-bit integer inference in plainC, corresponding to the source code shown in Listing 3.

slightly more involved changes to the node structure and the traversal function. Source code and data are available at ess.cs.uos.de/git/artifacts/plos26-regression-trees and [9].

3.1 Plain C and C++

We implement our plainC traversal function in a way that avoids penalties associated with function calls and branch mis-predictions. It uses (branch-free) arithmetic operations to determine the ID of the next node, as shown in Listing 3. As such, it consists of a single loop without recursive function calls, where each loop iteration corresponds to one tree level.

For all `inout_t` definitions, the compiled traversal loop body contains no more than a dozen instructions, and just a single (from a pipeline perspective, unlikely) conditional jump that takes care of the loop's exit condition. See Listing 4 for the entire function body of the 8-bit integer variant.

For plainC++, we embed the traversal function into the node data structure, which is not possible in C. In contrast to plainC, the function body does not keep track of the current node's absolute ID. Instead, it resorts to a conventional if/else check for determining the next node, and uses recursive `traverse` calls rather than a for loop, as shown in Listing 5.

The compiler detects that this is tail recursion, and optimizes accordingly: the compiled function does not contain any actual function calls. Instead, it uses a loop that is quite similar to the one emitted for plainC (see Listing 6 for the 8-bit integer variant). Independent of `inout_t`, its body consists of eleven instructions, including two conditional jumps (one for the if/else statement and one for the loop's exit condition) and two unconditional jumps (next iteration).

For both of these variants, the traversal function is stored in the text segment, taking up just 44 to 62 B. With proper alignment, it fits into a single instruction cache line. The tree

```

struct node {
    /* ... */
    inout_t traverse(const node *tree, inout_t *vec) const {
        if (feat == 255) {
            return threshold;
        }
        if (vec[feat] <= threshold) {
            return (this+1)->traverse(tree, vec);
        }
        return tree[rightChild].traverse(tree, vec);
    }
};

```

Listing 5. Regression tree inference in plainC++.

```

00: lea     0x...(%rip),%rax    ; this := &tree[0]
07: mov     %rax,%rsi          ; src := &tree[0]
0a: movzbl 0x2(%rax),%edx       ; data := this.feat
0e: mov     0x3(%rax),%cl       ; counter := this.threshold
11: cmp     $0xff,%dil          ; this.feat <=> 255
14: je      2a                  ; if .feat == 255: return .threshold
16: cmp     %cl,(%rdi,%rdx,1);vec[.feat] <=> .threshold
19: jg      21                  ; if ">": goto 21
1b: add     $0x4,%rax           ; else: this++ (left child)
1f: jmp     0a                  ; goto 0a (tail recursion)
21: movzwl  (%rax),%eax          ; accu := this.rightChild
24: lea     (%rsi,%rax,4),%rax;  this := tree[.rightChild]
28: jmp     0a                  ; goto 0a (tail recursion)
2a: mov     %ecx,%eax           ; accu := this.threshold
2c: ret                          ; return value in accu

```

Listing 6. Machine code for 8-bit int inference in plainC++, corresponding to the source code shown in Listing 5.

array is stored in rodata, taking up 4 B (int8), 6 B (int16), or 8 B (int32, float) per node in both variants.

3.2 Executable Data Structures

For our *executable data structure* approach, we utilize compile-time C++ template expansion to transform `traverse()` from a recursive function into a series of nested if/else checks that employs neither function calls nor loops. All of this is done by the compiler – we do not do so during source code generation. In fact, the node struct and tree array are identical to plainC, apart from the fact that we now use the `constexpr` qualifier to identify the tree as a compile-time constant.

The structure of the templateC++ traversal function is similar to the one used within plainC++ (see Listing 5). However, we now implement it as a C++ template, using the index of the current tree node as a *template parameter*. As such, when compiling the source code, the compiler will first instantiate one traversal function for each node in the tree: one for the root, one for its left child, one for its right child, and so on.

As applications only ever call `traverse` on the root node, the compiler optimizes the resulting machine code by inlining each nested traversal function call into the traversal function for the root node. Essentially, this transforms the traversal function's body into a series of nested if/else statements that resemble the structure of the tree. For the tree given in Fig. 1, this is equivalent to the following snippet: `if (vec[3] <= 7) { return (vec[1] <= 3) ? 1 : 2; } else { return (vec[3] <= 121) ? 7 : 5; }.`

```
constexpr const struct node tree[N_NODES] = { /* ... */ };
template<int index> inout_t traverse(inout_t *vec) {
    if (tree[index].feat == 255) {
        return tree[index].threshold;
    }
    if (vec[tree[index].feat] <= tree[index].threshold) {
        return traverse<index+1>(vec);
    }
    return traverse<tree[index].rightChild>(vec);
}
template<> inout_t traverse<sizeof(tree)/sizeof(node)>
(inout_t *vec) { (void)vec; return 0; }
```

Listing 7. Regression tree inference in templateC++.

```
00: xor    %eax,%eax    ; unused, for traverse<N_NODES>
02: ret     ; traverse<0> entry point at 03
03: mov    0x3(%rdi),%al ; accu := x3
06: cmp    $0x7,%al     ; x3 <=> 7
08: jg     15            ; if x3 > 7 : goto 15
0a: mov    $0x2,%al     ; accu := 2 (for "x1 > 3")
0c: cmpb   $0x3,0x1(%rdi) ; x1 <=> 3
10: jg     1e            ; if x1 > 3: goto 1e -> return 2
12: mov    $0x1,%al     ; else: accu := 1 ("x1 <= 3")
14: ret     ; return value stored in accu
15: cmp    $0x79,%al    ; x3 <=> 121
17: mov    $0x7,%dl     ; data := 7 (for "x3 <= 121")
19: mov    $0x5,%al     ; accu := 5 (for "x3 > 121")
1b: cmovle %edx,%eax    ; if x3 <= 121: accu := 7
1e: ret     ; return value stored in accu
```

Listing 8. templateC++ machine code for the tree in Fig. 1.

Listing 7 shows the traversal function template. The template parameter “<int index>” refers to the current node ID, and the feature vector *vec* is passed along as a conventional function argument. So, the node ID must be a compile-time constant (which is why the tree must be annotated as `constexpr`), whereas the feature vector is an arbitrary runtime variable. The second `traverse` definition (prefixed with `template<>`) is required to catch out-of-bounds tree array accesses during template expansion. It is present in the compiled machine code, but never executed.

From an application interface perspective, there are just two changes: the tree can no longer be exchanged at runtime, and applications must call `traverse<0>(vec)` rather than `traverse(vec)` or `tree[0].traverse(vec)`.

As Listing 8 shows, the generated machine code does indeed directly represent the structure (and thresholds) of the underlying regression tree. It contains neither loop nor function call instructions, and – for our example with seven tree nodes and 8-bit integer quantization – takes up just 31 Bytes for inference code and tree data combined. This is less than half the space requirements of `plainC` and `plainC++`.

When using 8-, 16- or 32-bit integers for `inout_t`, the compiler uses `x86_64` immediate operands to embed each threshold directly into the `cmpb` / `cmp` / `cmpl` instruction. Hence, the data and rodata segments are empty – everything is stored in the text segment, thus improving cache locality. Only for `float`, thresholds (4 B per node) are stored separately in the rodata segment.

Note that all of this works with standard C++ features and g++ optimizations; there are no custom patches, plugins or

optimization passes involved. Our compiler invocation is a straightforward g++ `-g -O3 -std=c++14 -c -o tree.o tree.cc` call (i.e., include debug information, optimize for code size, and use the C++14 standard).

4 Evaluation

We use a custom Python script to build regression trees, regression model trees, and regression forests with variable attributes, automatically transform them into corresponding C or C++ source code, and measure their inference latency on an Intel Xeon E5-2650 v2 CPU clocked at 2.6 GHz, running Linux 6.12. This CPU was released back in 2013, so our latency measurements are rather conservative – we expect that more recent CPUs will achieve even lower values.

The Python script relies on scikit-learn’s `sklearn.tree.DecisionTreeRegressor` for RT, `xgboost` for RF, and custom RMT code. Both `sklearn` and `xgboost` are state of the art modules for machine learning in Python; RMT are quite young and thus not part of the Python module ecosystem.

Our configuration space covers six dimensions, which we explore systematically for RT, RMT, and RF:

1. number of numeric features ($n \in [1, 11]$ for RT and RF, $n_N \in [1, 4]$ for RMT),
2. number of categorical features ($n_C \in [1, 4]$, RMT only),
3. maximum tree depth d (8 to 16, RT and RF only),
4. number of trees K (20 to 200, RF only),
5. input data type (8-/16-/32-bit integer or float), and
6. code generation (plainC, plainC++, or templateC++).

For each unique combination of configurations in dimensions 1 through 4, we train a single RT / RMT / RF data structure. As such, each pair of benchmark targets that only differs in dimension 5 and/or 6 has the exact same tree structure and number of nodes, meaning that we can determine the speedup of one data type or code generation method over another by comparing the measured latencies of pairs of otherwise identical configurations.

We take several hundred measurements for each configuration, and use the median of those for latency and speedup analysis. Since we use a standard Linux system, our latency data is noisy: when comparing the median to all measurements within the second and third quartile, we observe an average deviation of 9 % for RT and RMT, and 3 % for RF.

In addition to the three code generation variants, we also perform RT, RMT, and RF inference within the Python script. On the one hand, this allows us to compare the latency of our C/C++ code with pure-Python inference. On the other hand, we use Python inference results to verify the correctness of C/C++ inference, and thus of the code generation process.

4.1 Plain C and C++

First off, we compare the latency of `plainC` to inference within our Python script. As Fig. 3 shows, compiled C/C++ code is indeed faster by two to three orders of magnitude,

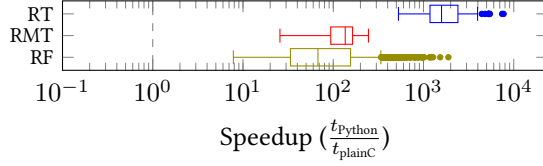


Figure 3. Speedup distribution of plainC over Python-based inference for all examined RT, RMT, and RF configurations.

Table 1. Median latency (ns) for regression tree and regression forest inference, modelled via least-squares regression.

Data Type	plainC	plainC++	templateC++
int8 (RT)	$13.9 \cdot d$	$13.2 \cdot d$	$7.8 \cdot d$
int16 (RT)	$17.8 \cdot d$	$14.8 \cdot d$	$10.6 \cdot d$
int32 (RT)	$14.2 \cdot d$	$12.8 \cdot d$	$7.9 \cdot d$
float (RT)	$23.2 \cdot d$	$13.5 \cdot d$	$9.0 \cdot d$
int8 (RF)	$15.4 \cdot Kd$	$13.0 \cdot Kd$	$1.5 \cdot Kd$
int16 (RF)	$18.0 \cdot Kd$	$14.2 \cdot Kd$	$11.8 \cdot Kd$
int32 (RF)	$14.2 \cdot Kd$	$14.3 \cdot Kd$	$11.6 \cdot Kd$
float (RF)	$22.4 \cdot Kd$	$14.4 \cdot Kd$	$17.8 \cdot Kd$

with RT showing the greatest speedup. Improvements for RF are less pronounced, as xgboost uses multi-threading during inference, whereas our C/C++ implementations are single-threaded. We do not observe any significant influence of number of features, tree depth, number of trees, or data type on latency reduction.

Median latency for the Python baseline ranges from 140 to 320 μ s for RT, 2 to 7 μ s for RMT, and 600 μ s to 1.4 ms for RF. Median latency for plainC ranges from 36 to 500 ns (RT, RMT) and 700 ns to 84 μ s (RF), respectively. These findings are not surprising, but nevertheless an important sanity check.

Next, we use least-squares regression to build latency prediction models for all RT and RF configurations. We find that depth d and number of trees K are sufficient to predict inference latency with 20 % (RT) and 38 % (RF) average error. While this is not stellar accuracy, the resulting models are simple enough for manual analysis, and also agree with our speedup observations. Table 1 shows the model functions.

We find that there is little difference between plainC++ and plainC, both in terms of code size (see Section 3.1) and inference latency. Most differences in Table 1 are within each other’s error range, and thus not suitable for assessing which method is faster. An analysis of raw measurement data (see Fig. 4) shows that plainC++ is slightly better for most regression forest variants. We observe the highest speedup (up to two-fold) for floating-point forests, lesser improvements for 8- and 16-bit integer-quantized forests, and (on average) equal latency for 32-bit integers. Meanwhile, plainC is a slightly better option for most (but not all) integer-quantized

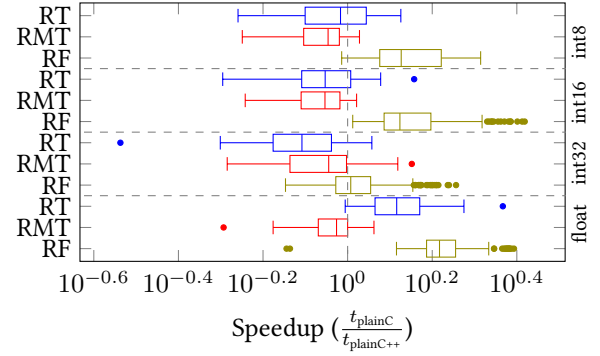


Figure 4. Speedup distribution of plainC++ over plainC.

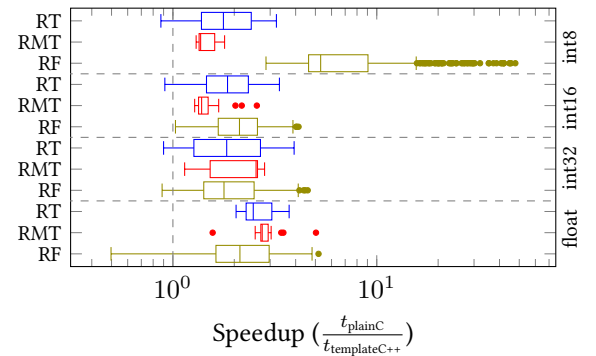


Figure 5. Speedup distribution of templateC++ inference over plainC for all examined RT, RMT, and RF configurations.

RT and RMT configurations, with 32-bit integer trees showing the highest median speedup and floating-point regression trees performing better with plainC++ instead.

4.2 Executable Data Structures

For templateC++, Table 1 indicates a two- to ten-fold speedup over plainC and plainC++ with 8-bit integer thresholds, and lesser improvements over almost all other variants. Floating-point forests are the only exception, as float values cannot be embedded as immediates. Overall, less than 1.7 % of all configuration pairs indicate a slowdown. Fig. 5 confirms this: templateC++ is generally faster, with an up to 14-fold improvement for 8-bit forests. Median latency now ranges from 23 to 390 ns (RT, RMT) and 200 ns to 79 μ s (XGB).

In terms of code size, Fig. 6 shows that templateC++ can be a double-edged sword. For 8-bit integer data, executable data structures enable faster *and smaller* binaries than conventional approaches in all configurations. For 16-bit integer as well as 32-bit float thresholds, regression trees and forests are slightly larger, whereas regression model trees are more compact. For 32-bit integers, except for RMT, there is virtually no size difference between the three methods.

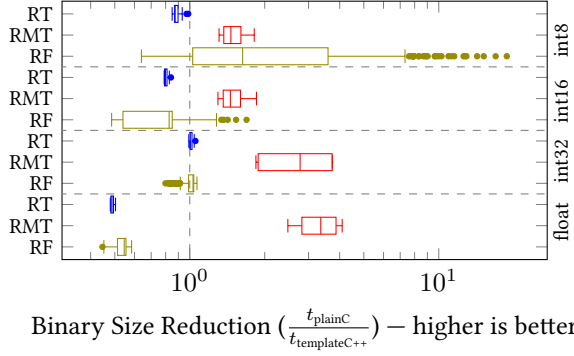


Figure 6. Binary size reduction of templateC++ over plainC.

5 Discussion

As we have seen, data-to-code transformation via compile-time C++ template expansion provides significant benefits: inference code is two to 14 times faster and, in many cases, *also* two to four times smaller. While we do observe an up to six-fold increase in binary size for floating point data, we consider this to be acceptable. On the one hand, we expect that practical applications will almost always use integer-quantized data. On the other hand, we are talking about 8 B per tree node vs. 48 B per tree node – for reasonably small trees and forests, we consider both to be acceptable.

However, there is a drawback: plainC and plainC++ support runtime training, whereas templateC++ requires struct node tree[] to be a compile-time constant. So, placement algorithms must either be provided with compile-time models, or utilize dynamic recompilation in order to reap the benefits of executable data structures via template expansion.

As for inference latency, Table 1 shows that we can expect 1.5 to 23 ns per tree level, depending on data type and implementation variant. Assuming that a placement decision must take less than 1 μ s, templateC++ allows us to use trees with a maximum depth of up to 128, or forests with a Kd factor of up to 650. In our experience, this is well within the range of training hyper-parameters used in practice. For less aggressively quantized data, trees must be more compact, but this still leaves room for capable machine learning models.

6 Related Work

With the advent of large language models (LLMs), a plethora of publications deal with “AI in operating systems” in general. When it comes to using machine learning models for (fast) runtime decisions, the body of works is thinner. We observe two different research directions: scheduler (i.e., task placement) and allocator (i.e., data placement) improvements.

On the scheduling side, according to a survey, researchers have already been considering model-guided strategies for more than two decades [8]. However, all six referenced documents focus on *which* task to schedule next or for *how long* to schedule it, and not *where* to place it. Nevertheless, five

of them have examined regression trees for runtime decisions, and four find that they provide higher accuracy than methods such as multi-layer perceptrons (i.e., simple neural networks). We are not aware of any fine-grained analyses of inference latency or C/C++ code generation variants.

KMLib provides a framework for usage and *runtime learning* of models within the kernel [2]. However, it has only been evaluated with linear least-squares regression, and latency data is not available.

As for data placement, research so far has focused on model-guided, hand-crafted heuristics and *offline* algorithms that cannot be used at runtime. For instance, several works address data placement on systems with heterogeneous main memory, but none of them directly use a machine learning model for making runtime placement decisions [4, 7, 17, 18].

Combined compute and data placement in an NMC context faces a similar issue: while past research has used models or model-guided simulations for placement decisions, the algorithms themselves are still hand-crafted, and their decision latency has not been evaluated [3, 12, 15].

Overall, we see that regression trees often outperform other machine learning models for runtime decisions within operating systems. Yet, we are not aware of any publication that provides a fine-grained performance analysis of the trade-offs between tree variant, tree complexity, and code generation method on inference latency and ROM usage. Our contribution bridges just this gap.

7 Conclusion

We have examined the suitability of regression trees and forests in C/C++ code for microsecond-scale runtime decisions in operating systems, using three implementation variants and four data types (8-/16-/32-bit int and 32-bit float).

We find that these data structures are viable, with an inference latency of 1.5 to 23 ns per tree level even on a one decade old CPU. Our contribution, a data-to-code transformation using compile-time C++ template expansion, achieves up to ten times lower latency *and* lower code size than conventional approaches for 8-bit integer data. In other configurations, it is also generally faster, but not necessarily more compact.

Note that we used a standard gcc 15.3.0 with the -Os flag to obtain these results. We did not implement custom compiler passes or similar; all the tools we need are already there.

In general, independent of data type and implementation variant, we find that regression trees and forests are suitable for microsecond-scale runtime decisions. As such, we hope to see more operating systems that use model-guided placement algorithms rather than (often error-prone and hardly understandable) hand-crafted heuristics in the future.

Acknowledgments

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 502565817.

References

- [1] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais, and Juliana Alves Pereira. 2022. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *Proceedings of the 26th International Systems and Software Product Line Conference - Volume A (SPLC '22)*. Association for Computing Machinery, 85–96. doi:10.1145/3546932.3546997
- [2] Ibrahim Umit Akgun, Ali Selman Aydin, and Erez Zadok. 2020. KMLib: Towards Machine Learning for Operating Systems. In *Proceedings of the On-Device Intelligence Workshop, co-located with the MLSys Conference*.
- [3] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Adaptive Query Compilation with Processing-in-Memory. In *Proceedings of the 39th International Conference on Data Engineering Workshops (ICDEW '23)*. 191–197. doi:10.1109/ICDEW58674.2023.00035
- [4] Anne Benoit, Swann Perarnau, Loïc Pottier, and Yves Robert. 2018. A Performance Model to Execute Workflows on High-Bandwidth-Memory Architectures. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. Association for Computing Machinery, Article 36, 10 pages. doi:10.1145/3225058.3225110
- [5] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. 1984. *Classification and Regression Trees* (1 ed.). Routledge. doi:10.1201/9781315139470
- [6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, 785–794. doi:10.1145/2939672.2939785
- [7] Rathish Das, Kunal Agrawal, Michael A. Bender, Jonathan Berry, Benjamin Moseley, and Cynthia A. Phillips. 2020. How to Manage High-Bandwidth Memory Automatically. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20)*. Association for Computing Machinery, 187–199. doi:10.1145/3350755.3400233
- [8] Siddarth Dias, Sidharth Naik, K Sreepraneeth, Sumedha Rmaan, and M Namratha. 2017. A Machine Learning Approach for Improving Process Scheduling: A Survey. *International Journal of Computer Trends and Technology* 43, 1 (2017). doi:10.14445/22312803/IJCTT-V43P101
- [9] Birte Friesel. 2026. *Code Generation from Regression Trees for Microsecond-Scale Decisions in Operating Systems (Artefact)*. doi:10.5281/zenodo.21410991
- [10] Birte Friesel, Marcel Lütke Dreimann, Mario Porrmann, and Olaf Spinczyk. 2026. Towards Task-Based Scheduling on Truly Heterogeneous Systems. In *Tagungsband des FG-BS Frühjahrstreffens 2026*. Gesellschaft für Informatik e.V. doi:10.18420/fgbs2026f-03
- [11] Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. 2024. Performance Models for Task-based Scheduling with Disruptive Memory Technologies. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems (DIMES '24)*. Association for Computing Machinery, 1–8. doi:10.1145/3698783.3699376
- [12] Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. 2025. Lightning Talk: Feasibility Analysis of Semi-Permanent Database Offloading to UPMEM Near-Memory Computing Modules. In *Datenbanksysteme für Business, Technologie und Web – Workshopband*. Gesellschaft für Informatik, 355–366. doi:10.18420/BTW2025-140
- [13] Birte Friesel and Olaf Spinczyk. 2022. Black-Box Models for Non-Functional Properties of AI Software Systems. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI (CAIN '22)*. Association for Computing Machinery, 170–180. doi:10.1145/3522664.3528602
- [14] Birte Friesel and Olaf Spinczyk. 2022. Regression Model Trees: Compact Energy Models for Complex IoT Devices. In *Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench '22)*. IEEE, 1–6. doi:10.1109/CPS-IoTBench56135.2022.00007
- [15] Birte Friesel and Olaf Spinczyk. 2025. Overhead Prediction for PIM-Enabled Applications with Performance-Aware Behaviour Models. In *Proceedings of the 1st Cross-disciplinary Conference on Memory-Centric Computing (CCMCC '25)*. IEEE, 1–8. doi:10.1109/CCMCC67628.2025.11380800
- [16] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Software Engineering* 23, 3 (June 2018), 1826–1867. doi:10.1007/s10664-017-9573-6
- [17] Mohammad Laghari and Didem Unat. 2017. Object Placement for High Bandwidth Memory Augmented with High Capacity Memory. In *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '17)*. 129–136. doi:10.1109/SBAC-PAD.2017.24
- [18] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. 2022. Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory. *Proc. VLDB Endow.* 15, 11 (July 2022), 2867–2880. doi:10.14778/3551793.3551837
- [19] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, Article 1, 16 pages. doi:10.1145/2901318.2901326
- [20] Calton Pu, Alexia Massalin, and John Ioannidis. 1988. The Synthesis Kernel. *Computing Systems* 1, 1 (Winter 1988), 11–32.
- [21] Jeffrey Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.