Realization of Random Forest for Real-Time Evaluation through Tree Framing

{sebastian.buschjaeger,kuan-hsun.chen,jian-jia.chen,katharina.morik}@tu-dortmund.de

technische universität dortmund
Artificial Intelligence Group
computer science 12

Project setting

Goal Hardware-awareness of Machine Learning
Why does this matter?

- Reduce energy costs by reducing hardware requirements
- Reduce training/prediction time by better hardware utilization

Focus here How can we concurrently apply a given model on a small device in real-time?

Abstract

Fact Random Forests are still one of the best blackbox learners available

Question How to optimize RF execution?

Basic idea Utilize the structure of trained tree

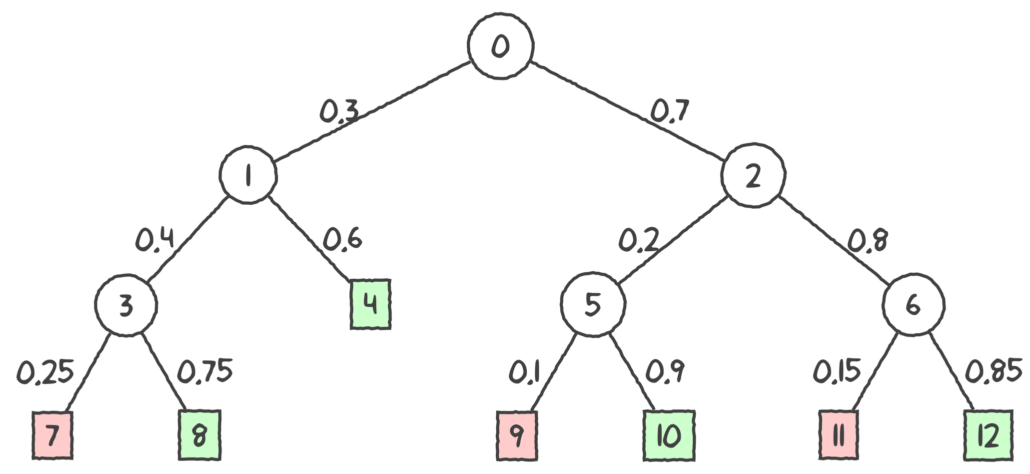
→ **Branch-probability** $p_{i \to j}$

→ **Path-probability** $p(\pi) = p_{\pi_0 \to \pi_1} \cdot \ldots \cdot p_{\pi_{L-1} \to \pi_L}$

→ **Expected path length** $\mathbb{E}[L] = \sum_\pi p(\pi) \cdot |\pi|$

Idea Use $E[L]$ to optimize memory-layout of trees

Example



$$p((0,1,3)) = 0.3 \cdot 0.4 \cdot 0.25 = 0.03$$
$$p((0,2,6)) = 0.7 \cdot 0.8 \cdot 0.85 = 0.476$$

Implementation 1: Native Tree

```
Node t[] = {/* ... */};
bool predict(short const * x){
    unsigned int i = 0;
    while(!t[i].isLeaf) {
        if (x[t[i].f] <= t[i].s) {
            i = t[i].l;
        } else {
            i = t[i].r;
        }
    }
    return t[i].pred;
}
```

Idea Iterate array of tree-nodes

+ Simple to implement
+ Small 'Hot'-Code
- Requires D-Cache (array)
- Requires I-Cache (code)
- Requires indirect mem. access

Implementation 2: If-Else Tree

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

Idea Unroll tree into `if-else`

+ No indirect mem. access
+ Compiler optimizes aggresivly
+ Only I-Cache required
- Code does not fit I-Cache
- No 'hot'-code

Optimization for Native Tree

Compulsory cache misses
→ Cache memory is not enough to hold complete array
→ Leaf-nodes only store the prediction. Pointer to children not necessary
Solution Store prediction directly in 'parent' node

Capacity and conflict cache misses
→ Pre-fetching does not work, if nodes are discontinuously arranged
→ Layout nodes in array so that they respect access pattern
Solution Greedily put nodes with highest probability in same cache set

- Put the root node into current working set $\mathcal{C}$. Set $i = 0$
- If $|\mathcal{C}| \leq \tau$: $\mathcal{C} = \mathcal{C} \cup \arg\max(p(i \to l(i)), p(i \to r(i)))$
- Continue until $|\mathcal{C}| \geq \tau$
- Place nodes in $\mathcal{C}$ continously in array

Optimization for If-Else Tree

Compulsory cache misses
→ Cache memory is not enough to store all code
→ Increase chance, that nodes with higher probabilities are in the cache
Solution Swap nodes if $p(i \to l(i)) \geq p(i \to r(i))$

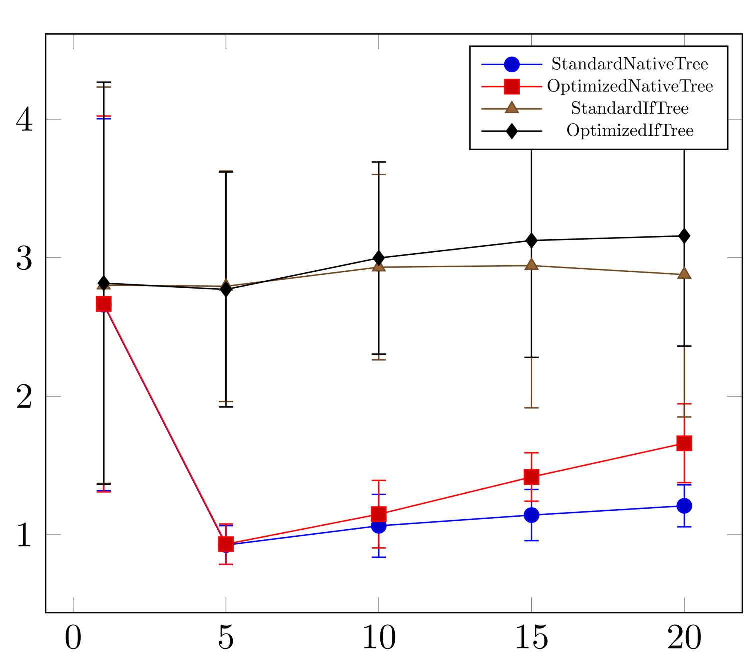
Capacity and conflict cache misses
→ Cache memory is not enough to store all code
→ Computation kernel of tree might fit into cache
Solution Compute computation kernel for budget β

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

- Start with the root node
- Greedily add nodes until budget exceeded

Note Estimate $s(\cdot)$ based on assembly analysis

Results on X86 CPUs



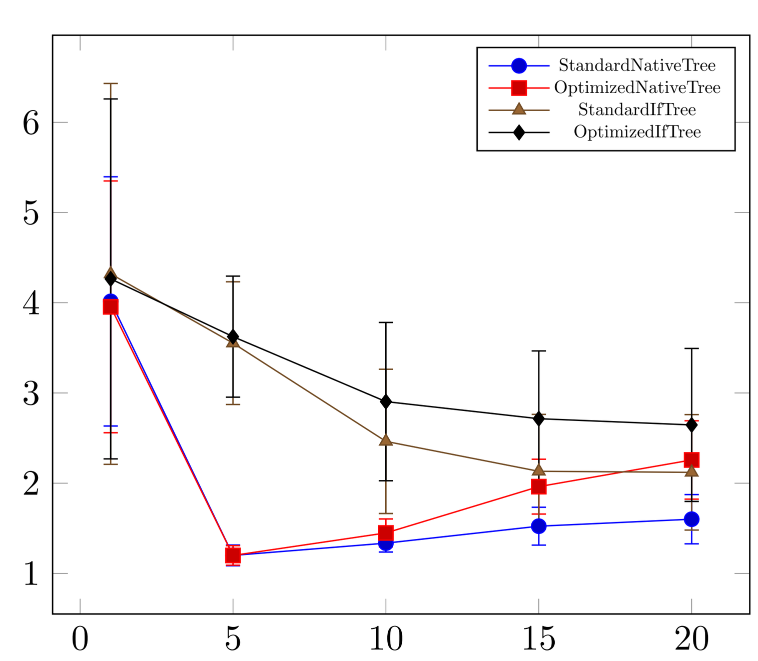
Results

- Optimizations improve performance
- `if-else` trees are clear winner

Interpretation

- Large I-Cache (256 KiB) favors `if-else`
- Compiler can utilize CISC architecture for `if-else`
- `Native` trees do not benefit from I-Cache and CISC

Results on ARM CPUs



Results

- Optimizations improve performance
- No clear winner for larger trees

Interpretation

- Smaller I-Cache (32 KiB) only fits small trees
- Smaller D-Cache (512 KiB) only fits small trees
- Requires more instructions than CISC

Conclusion

Take-away There are multiple ways of implementing Decision Trees on modern hardware

Thus Use code generator to automatically generate *all* possible implementations for a given architecture

We emperical evaluated our generator with a total of 1.800 experiments on 3 architectures

Results Speed-up around ≥ 3 on all architectures (`X86`, `ARM`, `PPC`)

Future Research and Improvements

- Improve compilation time → Generate intermediate language code
- Reduce memory footprint → Re-use common tree parts (subtree matching)
- Mix different implementation types → Switch from if-else to native when branching to deep

References

Part of the work on this paper has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", projects A1, B2 and C3.

Find us on bitbucket
`https://bitbucket.org/sbuschjaeger/arch-forest/`