

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

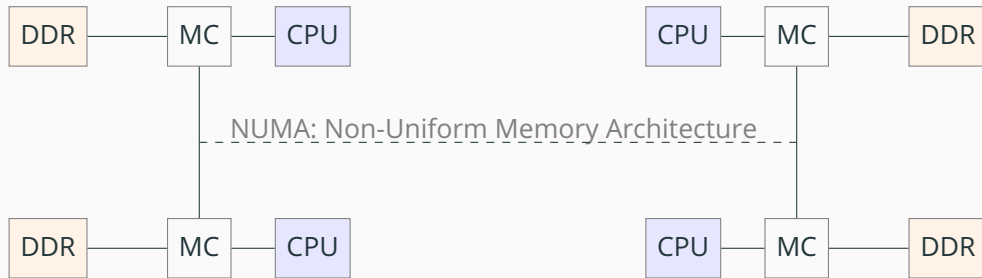
Birte Friesel, Marcel Lütke Dreimann, Olaf Spinczyk

September 29th, 2026

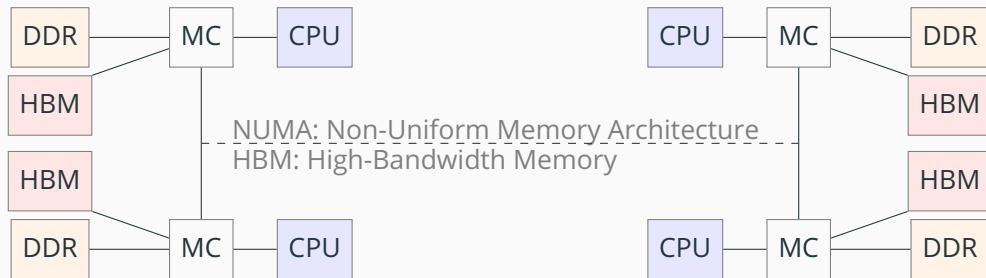
`ess.cs.uos.de/~bkf`

`birte.friesel@uos.de`

Motivation: Task/Data Placement



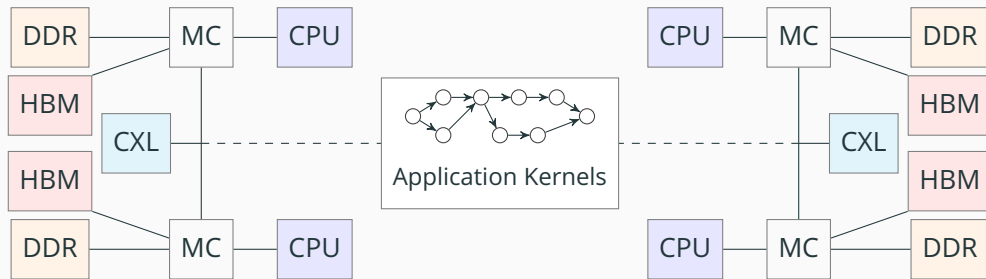
Motivation: Task/Data Placement



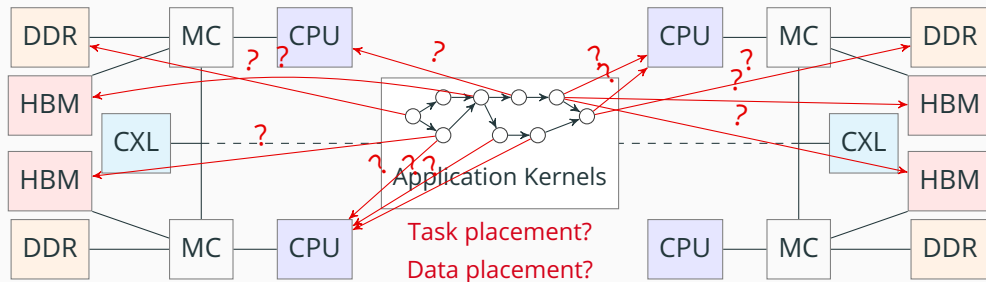
Motivation: Task/Data Placement



Motivation: Task/Data Placement



Motivation: Task/Data Placement



- Challenge: spatial placement of tasks and data (“where to run / allocate?”)
 - Optimal placement: NP-hard; requires knowledge about future
 - State of the art: manual heuristics (fast at cost of pathological edge cases)

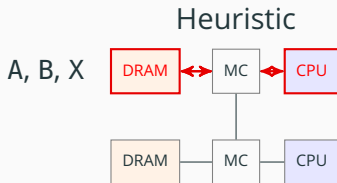
Pathological Edge Cases in Practice



Example (Vector Addition)

$\vec{x} = \vec{a} + \vec{b} \quad \hat{=}$ `for (i = 0; i < n; i++) { X[i] = A[i] + B[i] }`

Heuristic: A, B, X and compute on same NUMA node (local data)



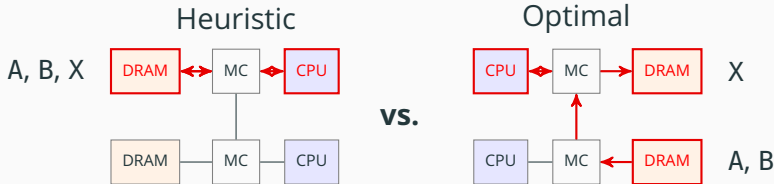
Pathological Edge Cases in Practice



Example (Vector Addition)

$\vec{x} = \vec{a} + \vec{b} \quad \hat{=}$ for (i = 0; i < n; i++) { X[i] = A[i] + B[i] }

Heuristic: A, B, X and compute on same NUMA node (local data)



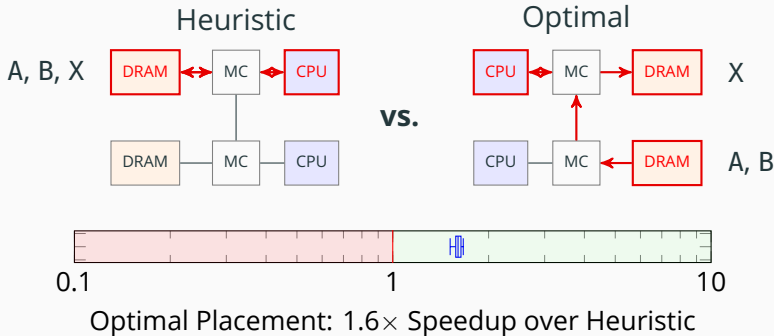
Pathological Edge Cases in Practice



Example (Vector Addition)

$\vec{x} = \vec{a} + \vec{b} \quad \hat{=}$ for (i = 0; i < n; i++) { X[i] = A[i] + B[i] }

Heuristic: A, B, X and compute on same NUMA node (local data)

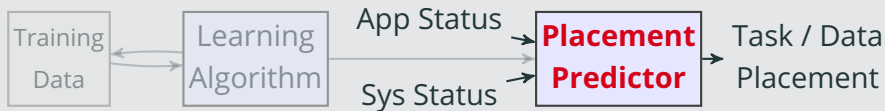


Heuristics vs. Machine Learning Models



- Heuristic: manually-distilled assumptions and past experience
- Model: learnt automatically from benchmarks and/or simulations

Example (Model-Guided Task / Data Placement)

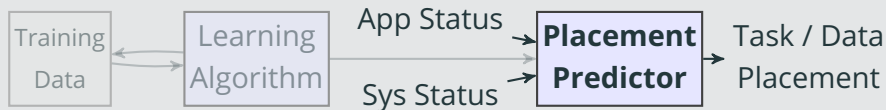


Heuristics vs. Machine Learning Models



- Heuristic: manually-distilled assumptions and past experience
- Model: learnt automatically from benchmarks and/or simulations

Example (Model-Guided Task / Data Placement)



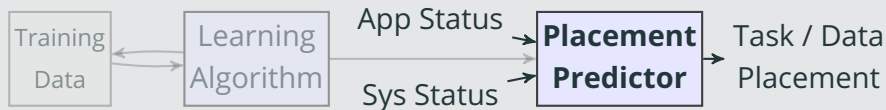
- Challenge: model predictions must be **fast**
- Heuristics: shallow nested if-else chain → near-instantaneous (≈ 10 ns)

Heuristics vs. Machine Learning Models



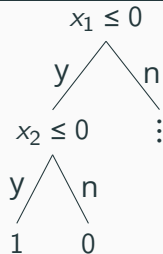
- Heuristic: manually-distilled assumptions and past experience
- Model: learnt automatically from benchmarks and/or simulations

Example (Model-Guided Task / Data Placement)

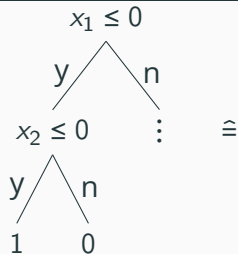


- Challenge: model predictions must be **fast**
- Heuristics: shallow nested if-else chain → near-instantaneous (≈ 10 ns)
- ML models: arbitrarily complex (e.g. neural networks: $\approx$ ms)

Approach: Decision Trees



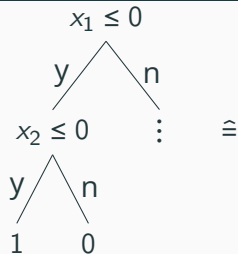
Approach: Decision Trees



```
if ((task_node == 0) && mem_bound)
    data_node = 1; // remote data
else if (task_node == 0)
    data_node = 0; // local data
else if (/* ... */)
    /* ... */
```

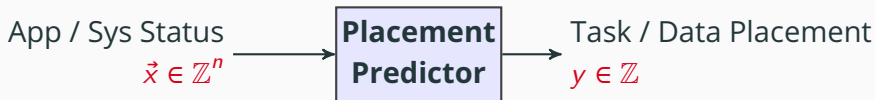
- Fast and compact machine learning model; structure $\hat{=}$ if/else chains

Approach: Decision Trees

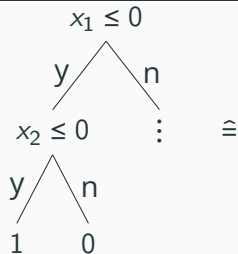


```
if ((task_node == 0) && mem_bound)
    data_node = 1; // remote data
else if (task_node == 0)
    data_node = 0; // local data
else if (/* ... */)
    /* ... */
```

- Fast and compact machine learning model; structure $\hat{=}$ if/else chains



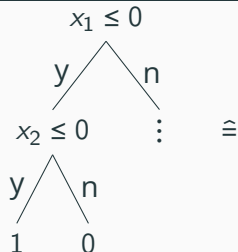
Approach: Decision Trees



```
if ((task_node == 0) && mem_bound)
    data_node = 1; // remote data
else if (task_node == 0)
    data_node = 0; // local data
else if (/* ... */)
    /* ... */
```

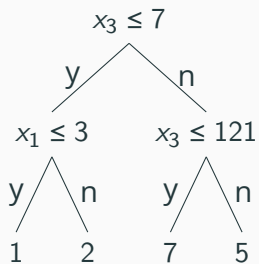
- Fast and compact machine learning model; structure $\hat{=}$ if/else chains
- Decision tree can be learnt, updated and evaluated automatically
 - Traceable runtime decisions in OS; backed by training data

Approach: Decision Trees



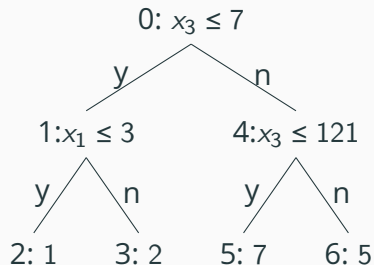
```
if ((task_node == 0) && mem_bound)
    data_node = 1; // remote data
else if (task_node == 0)
    data_node = 0; // local data
else if (/* ... */)
    /* ... */
```

- Fast and compact machine learning model; structure $\hat{=}$ if/else chains
- Decision tree can be learnt, updated and evaluated automatically
 - Traceable runtime decisions in OS; backed by training data
- Applicable to $\approx$ any in-OS heuristic, not just task/data placement (if decision trees are fast enough)





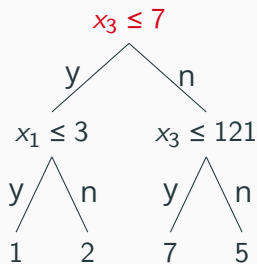
- Store Tree as list of nodes (pre-order traversal)



```
tree = {node0, node1, node2, node3, node4, node5, node6};
```



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data



```
struct node {
```

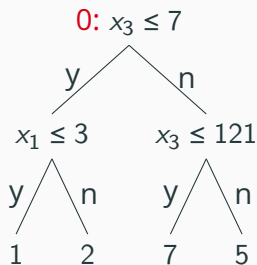
```
}
```

```
tree = {node0, node1, node2, node3, node4, node5, node6};
```

Textbook Approach: Data Serialization



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data



```
struct node {  
    // implicit: node index (starting at 0)
```

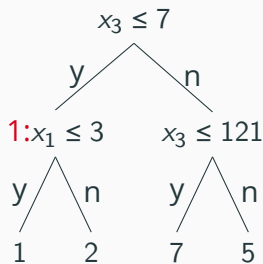
```
}
```

```
tree = {node0, node1, node2, node3, node4, node5, node6};
```

Textbook Approach: Data Serialization



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data

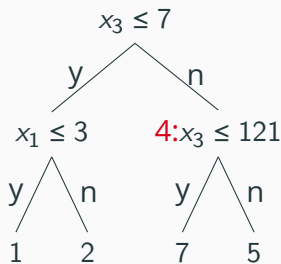


```
struct node {  
    // implicit: node index (starting at 0)  
    // implicit: left child is node index + 1  
};
```

```
tree = {node0, node1, node2, node3, node4, node5, node6};
```



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data

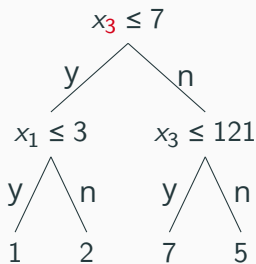


```
struct node {
    // implicit: node index (starting at 0)
    // implicit: left child is node index + 1
    const uint16_t rightChild;
}
```

```
tree = {node0, node1, node2, node3, node4, node5, node6};
```



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data

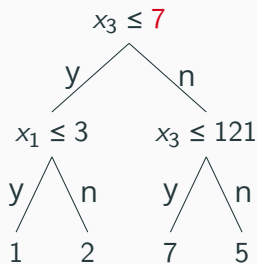


```
struct node {
    // implicit: node index (starting at 0)
    // implicit: left child is node index + 1
    const uint16_t rightChild;
    const uint8_t feat;
}
```

```
tree = {node0, node1, node2, node3, node4, node5, node6};
```



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data

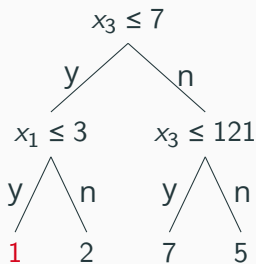


```
struct node {
    // implicit: node index (starting at 0)
    // implicit: left child is node index + 1
    const uint16_t rightChild;
    const uint8_t feat;
    const inout_t threshold;
}
```

```
tree = {node0, node1, node2, node3, node4, node5, node6};
```



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data

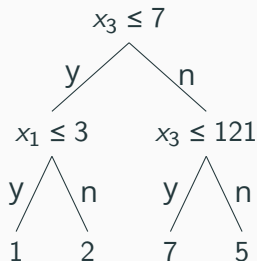


```
struct node {
// implicit: node index (starting at 0)
// implicit: left child is node index + 1
const uint16_t rightChild; // = 0
const uint8_t feat; // = 255
const inout_t threshold; // output
}
```

```
tree = {node0, node1, node2, node3, node4, node5, node6};
```



- Store Tree as list of nodes (pre-order traversal)
- Minimize per-node data



```
struct node {  
    // implicit: node index (starting at 0)  
    // implicit: left child is node index + 1  
    const uint16_t rightChild;  
    const uint8_t feat;  
    const inout_t threshold;  
}
```

⇒ 4 Bytes per tree node with `inout_t = int8_t` (8-bit quantization)

Textbook Approach: Prediction



```
inout_t predict(struct node *tree, inout_t *vec) {  
    uint16_t i = 0;           // start at root node  
    while (tree[i].feat != 255) { // while current node != leaf  
        bool cmp = vec[tree[i].feat] <= tree[i].threshold; // find child  
        i = cmp * (i + 1) + !cmp * tree[i].rightChild; // continue there  
    }  
    return tree[i].threshold; // return value stored in leaf  
}
```



```
inout_t predict(struct node *tree, inout_t *vec) {
    uint16_t i = 0;                // start at root node
    while (tree[i].feat != 255) { // while current node != leaf
        bool cmp = vec[tree[i].feat] <= tree[i].threshold; // find child
        i = cmp * (i + 1) + !cmp * tree[i].rightChild; // continue there
    }
    return tree[i].threshold;      // return value stored in leaf
}
```

- gcc -Os: < 64 B ROM, 4 .. 8 B per node RAM,
prediction 13 .. 23 ns per tree level (depending on quantization)



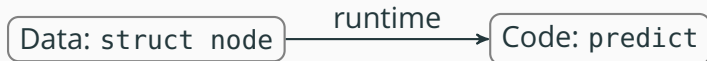
```
inout_t predict(struct node *tree, inout_t *vec) {  
    uint16_t i = 0;           // start at root node  
    while (tree[i].feat != 255) { // while current node != leaf  
        bool cmp = vec[tree[i].feat] <= tree[i].threshold; // find child  
        i = cmp * (i + 1) + !cmp * tree[i].rightChild; // continue there  
    }  
    return tree[i].threshold; // return value stored in leaf  
}
```

- gcc -Os: < 64 B ROM, 4 .. 8 B per node RAM,
prediction 13 .. 23 ns per tree level (depending on quantization)
- Main bottleneck: rightChild → non-prefetchable data segment access

More Efficient Tree Traversal



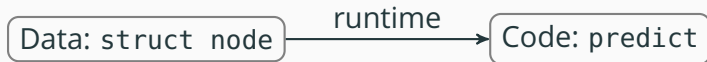
- Textbook approach: data (tree) and code (traversal) stored separately
- Traversal delayed by waiting for cache or RAM reads



More Efficient Tree Traversal



- Textbook approach: data (tree) and code (traversal) stored separately
- Traversal delayed by waiting for cache or RAM reads

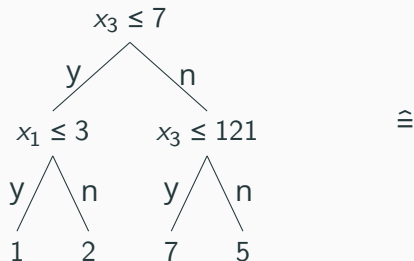


- Better: embed data into code; use immediate operands if possible
- No more data segment reads; everything is part of instruction stream



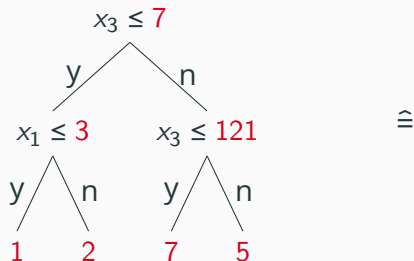
- Inspired by **Executable Data Structures** in Synthesis OS [PMI88; Mas92]

Executable Data Structure: Idea



```
inout_t heuristic(inout_t *vec) {  
    if (vec[3] <= 7)  
        if (vec[1] <= 3)  
            return 1;  
        else  
            return 2;  
    else  
        if (vec[3] <= 121)  
            return 7;  
        else  
            return 5;  
}
```

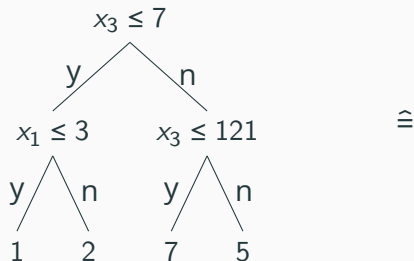
Executable Data Structure: Idea



- `cmp, return`: 8-bit immediate values $\rightarrow$ empty data segment

```
inout_t heuristic(inout_t *vec) {
    if (vec[3] <= 7)
        if (vec[1] <= 3)
            return 1;
        else
            return 2;
    else
        if (vec[3] <= 121)
            return 7;
        else
            return 5;
}
```

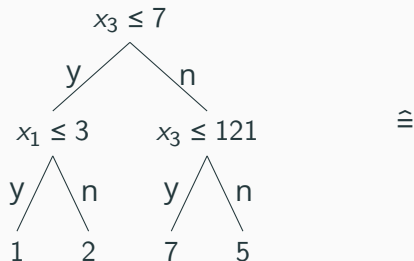
Executable Data Structure: Idea



- `cmp, return`: 8-bit immediate values → empty data segment
- But: code generator required :(

```
inout_t heuristic(inout_t *vec) {
    if (vec[3] <= 7)
        if (vec[1] <= 3)
            return 1;
        else
            return 2;
    else
        if (vec[3] <= 121)
            return 7;
        else
            return 5;
}
```

Executable Data Structure: Idea



```
inout_t heuristic(inout_t *vec) {
    if (vec[3] <= 7)
        if (vec[1] <= 3)
            return 1;
        else
            return 2;
    else
        if (vec[3] <= 121)
            return 7;
        else
            return 5;
}
```

- `cmp`, `return`: 8-bit immediate values → empty data segment
 - But: code generator required :(
- C++ template metaprogramming

Executable Data Structure: Implementation



```
constexpr const struct node tree[N_NODES] = {/*...*/};
```

Executable Data Structure: Implementation



```
constexpr const struct node tree[N_NODES] = {/*...*/};
```

```
template<int index> int8_t predict(int8_t *vec) {  
    if (tree[index].feat == 255)  
        return tree[index].threshold;           // leaf  
    if (vec[tree[index].feat] <= tree[index].threshold)  
        return predict<index+1>(vec);           // left child  
    return predict<tree[index].rightChild>(vec); } // right child
```

Executable Data Structure: Implementation

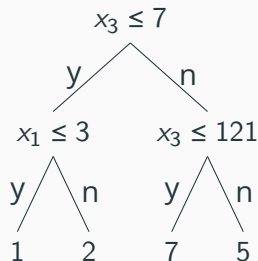


```
constexpr const struct node tree[N_NODES] = {/*...*/};
```

```
template<int index> int8_t predict(int8_t *vec) {  
    if (tree[index].feat == 255)  
        return tree[index].threshold;           // leaf  
    if (vec[tree[index].feat] <= tree[index].threshold)  
        return predict<index+1>(vec);           // left child  
    return predict<tree[index].rightChild>(vec); } // right child
```

- Generated machine code resembles nested if/else tree structure
- gcc -O3, int8 quantization: ≤ 4.5 B per node ROM, 0 B RAM,
prediction: as low as 1.5 ns per tree level

Executable Data Structure: Machine Code

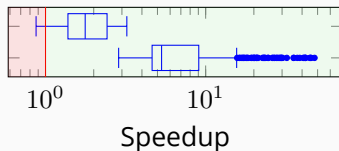


```
03: mov    0x3(%rdi),%al
06: cmp    $0x7,%al      ;x3 <=> 7
08: jg     15             ;"x3 > 7" -> jump
0a: mov    $0x2,%al      ;"x1 > 3" -> 2
0c: cmpb   $0x3,0x1(%rdi);x1 <=> 3
10: jg     1e             ;"x1 > 3" -> ret 2
12: mov    $0x1,%al      ;"x1 <= 3" -> ret 1
14: ret
15: cmp    $0x79,%al      ;x3 <=> 121
17: mov    $0x7,%dl       ;"x3 <= 121" -> 7
19: mov    $0x5,%al      ;"x3 > 121" -> ret 5
1b: cmovle %edx,%eax      ;"x3 <= 121" -> ret 7
1e: ret
```

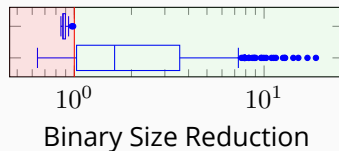
Performance Comparison



- 8-bit quantization: $\approx 2/5\times$ lower latency; $\approx 0.9/2\times$ lower binary size



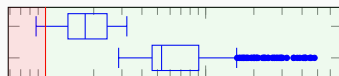
Tree
Forest



Performance Comparison

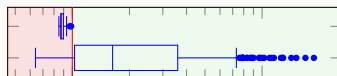


- 8-bit quantization: $\approx 2/5\times$ lower latency; $\approx 0.9/2\times$ lower binary size



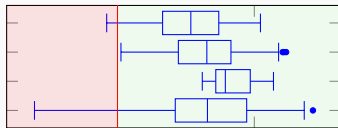
Speedup

Tree
Forest



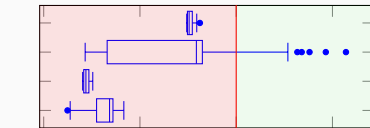
Binary Size Reduction

- Other data types: up to $2\times$ lower latency, but $\approx 2\times$ larger binary



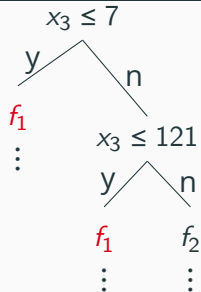
Speedup

i16 Tree
i16 Forest
f32 Tree
f32 Forest



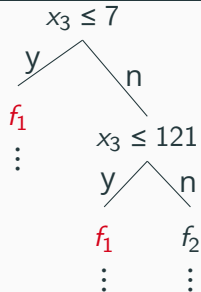
Binary Size Reduction

Side Note: Data Structure De-Duplication



- Decision trees may contain **identical sub-trees**

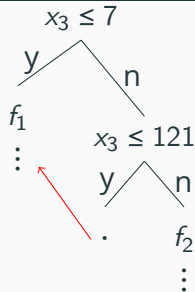
Side Note: Data Structure De-Duplication



- Decision trees may contain identical sub-trees
- All duplicates must be serialized

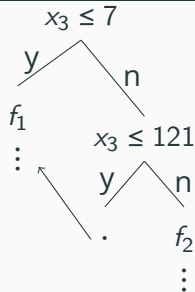
```
tree = {  
    { /* x3 <= 7 */ },  
    /* f1 ... */,  
    { x3 <= 121 /* },  
    /* f1 ... */,  
    /* f2 ... */  
}
```

Side Note: Data Structure De-Duplication



- Decision trees may contain identical sub-trees
- All duplicates must be serialized
- Executable data structure: `cmp` and `jmp` instructions
 - Compiler emits jump to already-encoded sub-tree
 - Data structure de-duplication free of charge

Side Note: Data Structure De-Duplication



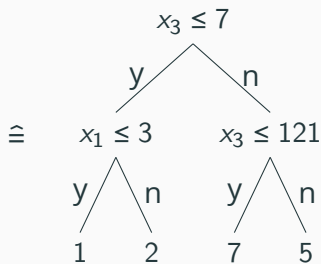
- Decision trees may contain identical sub-trees
- All duplicates must be serialized
- Executable data structure: `cmp` and `jg` instructions
 - Compiler emits jump to already-encoded sub-tree
 - Data structure de-duplication free of charge

- 8-bit integer quantization has highest chance of redundancy; hardly any in 16-bit integer or 32-bit float trees
- Significantly lower binary size for large, 8-bit decision forests
 - Effectiveness increases with # trees and maximum depth



- Heuristics for runtime decisions: fast (≈ 10 ns), but tedious
- Use decision trees instead: explainable and traceable runtime decisions

```
inout_t heuristic(inout_t *vec) {  
    if (vec[3] <= 7)  
        if (vec[1] <= 3) return 1;  
        else return 2;  
    else  
        if (vec[3] <= 121) return 7;  
        else return 5;  
}
```

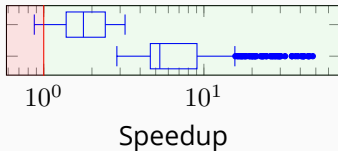




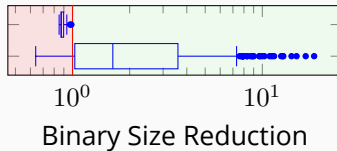
- Heuristics for runtime decisions: fast (≈ 10 ns), but tedious
- Use decision trees instead: explainable and traceable runtime decisions
- Textbook implementation: 13 .. 23 ns per tree level



- Heuristics for runtime decisions: fast (≈ 10 ns), but tedious
- Use decision trees instead: explainable and traceable runtime decisions
- Textbook implementation: 13 .. 23 ns per tree level
- **Executable data structures** (C++ template metaprogramming):
 - 8-bit quantization: latency as low as 1.5 ns per tree level (up to 10x reduction)
 - Compiler merges duplicate sub-trees → compression for large forests



Tree
Forest





- Heuristics for runtime decisions: fast (≈ 10 ns), but tedious
- Use decision trees instead: explainable and traceable runtime decisions
- Textbook implementation: 13 .. 23 ns per tree level
- **Executable data structures** (C++ template metaprogramming):
 - 8-bit quantization: latency as low as 1.5 ns per tree level (up to 10x reduction)
 - Compiler merges duplicate sub-trees → compression for large forests
- Replacing (hard-to-manage) heuristics with machine learning is feasible
- See paper and poster for details, implementation, and benchmark data



- Heuristics for runtime decisions: fast (≈ 10 ns), but tedious
- Use decision trees instead: explainable and traceable runtime decisions
- Textbook implementation: 13 .. 23 ns per tree level
- **Executable data structures** (C++ template metaprogramming):
 - 8-bit quantization: latency as low as 1.5 ns per tree level (up to 10x reduction)
 - Compiler merges duplicate sub-trees → compression for large forests
- Replacing (hard-to-manage) heuristics with machine learning is feasible
- See paper and poster for details, implementation, and benchmark data
- Future work: (online) training and evaluation of such models



- [Mas92] Alexia Massalin. **“Synthesis: An Efficient Implementation of Fundamental Operating System Services”**. PhD thesis. Columbia University, 1992.
- [PMI88] Calton Pu, Alexia Massalin, and John Ioannidis. **“The Synthesis Kernel”**. In: Computing Systems 1.1 (Winter 1988), pp. 11–32.