Immediate Split Trees: Immediate Encoding of Floating Point Split Values in Random Forests

Christian Hakert* (✉), Kuan-Hsun Chen†, Jian-Jia Chen*

* TU Dortmund University, Germany † University of Twente, Netherlands
christian.hakert@tu-dortmund.de, k.h.chen@utwente.nl,
jian-jia.chen@cs.tu-dortmund.de

Abstract. Random forests and decision trees are increasingly interesting candidates for resource-constrained machine learning models. In order to make the execution of these models efficient under resource limitations, various optimized implementations have been proposed in the literature, usually implementing either *native trees* or *if-else trees*. While a certain motivation for the optimization of if-else trees is to benefit the behavior of dedicated instruction caches, in this work we highlight that if-else trees might also strongly depend on data caches.
We identify one crucial issue of if-else tree implementations and propose an optimized implementation, which keeps the logic tree structure untouched and thus does not influence the accuracy, but eliminates the need to load comparison values from the data caches. Experimental evaluation of this implementation shows that we can greatly reduce the amount of data cache misses by up to $\approx 99\%$, while not increasing the amount of instruction cache misses in comparison to the state-of-the-art. We additionally highlight various scenarios, where the reduction of data cache misses draws important benefit on the allover execution time.

1 Introduction

Increasing focus on resource-constrained machine learning throughout the last years brings up random forests and decision trees in various shapes and flavours as premier candidates for resource-limited classification or regression models. Although decision trees can be tuned toward resource limitation on the tree structure already, by for instance limiting the amount of nodes and thus the total memory consumption, considering the underlying computing architecture turns out to allow even further optimization, e.g., [3,4,7,11,12]. As one particular aspect of the computing architecture, memory hierarchies and caches turn out to allow a certain interplay with decision trees. Caches are organized in a way, that they store a copy of intensively used memory contents in a small and fast memory, in order to reduce memory access latencies. The decision, which memory contents are stored in the cache, is made by the hardware, following certain prefetching and preemption strategies. Since the probability of accessing certain nodes in a decision tree can be profiled at training time, a certain prediction of memory accesses and memory access sequences can be made. This can be consequently

used to shape the trees in a way that the hardware caches automatically prefetch the data objects, which are likely used during further execution.

Towards this, implementation of decision trees yield two common realizations: 1) *native trees* and 2) *if-else trees* [1]. The former ones aim to encode tree nodes in a large data array and build up the tree structure by following pointers to the right or left child node at every node. The latter ones aim to translate every node of a tree to an if-else construct in the programming language and build up the tree structure by extensively nesting these if-else constructs. Generally, although most computers implement the von Neumann architecture and contain a unified data and instruction memory, the CPU implements different methods for data and instruction memory accesses. Within the CPU pipeline, data and instruction memory accesses can be performed at different stages and therefore utilize different hardware units. Furthermore, systems with cache hierarchies usually partition caches (mostly the first level caches) into instruction and data cache, leading to distinct access behavior and access latencies. Due to these aspects, considering data and instruction memory accesses of a program separately and focus on their optimization can lead to important performance improvements.

Since native trees intend to intensively utilize data memory and data caches, optimization of native trees targets to modify the node array structure in order to comfort data cache prefetching [4]. Contrarily, if-else trees intend to intensively use instruction memory and therefore optimization of if-else tree tries to reshape the structure of nested if-else blocks in order to comfort the prefetching of instruction caches [4]. When random forest models exceed the capacity of a cache, which likely happens for first level caches, compulsory cache misses in the instruction cache and data cache happen during the execution and cause increased latencies since caches are usually faster by orders of magnitude than main memory. Optimizing the prefetching of caches, as described before, can help to lower the introduced additional latency. When caches are partitioned into instruction and data caches, prefetching as well is separated for both caches, which highlights the need for separate optimization of instruction and data memory accesses.

Problem: In this work, we take a closer look to the memory behavior of if-else tree implementations and highlight that, counter intuitively to the design principle, if-else trees depend on intensive use of data memory and data caches as well, causing data cache misses, which is not considered and targeted by existing optimizations [3, 4].
Solution: Consequently, we propose an optimized implementation of if-else trees with floating point split values, where we eliminate large parts of the use of data caches. Thus, we can apply existing optimizations for if-else trees subsequently and improve their execution.

Experimental evaluation on X86 and ARMv8 based server and embedded systems highlights that our proposed optimization can reduce the amount of data cache misses by up to 99%, while not increasing the amount of instruction cache misses. In addition to the great reduction of data cache misses, we show

that the reduction of data cache misses directly contributes to a reduction of the allover execution time in various scenarios. End-to-end timing measurements demonstrate that we can reduce the allover execution time of decision trees with our proposed optimization, especially on server systems.

Our novel contributions:

- Analysis of the state-of-the-art optimization for if-else trees regarding the usage of the data cache.
- An optimized implementation, reducing large portions of data cache misses in if-else trees
- Experimental evaluation of the proposed implementation and comparison to the state of the art

2 Related Work

Several techniques have been proposed in the literature to speed up the execution of inference for tree-based ensembles. For binary search trees, Kim et al. in [7] presents an optimized realization by using vectorization units on X86, considering the register sizes, cache sizes, and page sizes specifically. However, such a technique requires a specific support from the underlying architectures. The concept of vectoring the tree structures is also applied to the context of ranking models in [9], which enhances the QuickScorer algorithm for gradient boosted trees [5, 8]. Ye et al. in [13] further improve the scalability of such vectorization methods by encoding the node representation to compact the memory footprint. These techniques decompose the tree-ensembles into different data structures based on the feature values, which is especially effective for large ensembles of smaller trees. Without traversing trees one by one, however, the target applications are mainly limited to batch-processing.

Architecture-aware implementations for decision trees have started from [12], which optimizes the implementations of decision trees on different architectures, i.e., CPUs, FPGAs, and GPUs. By fixing the tree-depth, Prenger et al. in [11] further show an effective pipelining approach over these computing units, based on the CATE algorithm during training. However, the impact of cache misses was not taken into account. The two common implementations for decision trees, i.e., native trees and if-else trees, are first distinguished in [1], which provides the first attempt to increase data locality for native trees. By leveraging the probability model of accessing nodes during tree traversal in [2], Buschjäger et al. in [3] propose several optimizations for memory layout over different tree implementations to improve the memory locality and show the potential speed-ups can be up to 2-4x over different architectures. Chen et al. enhance this method further by compiler based binary size estimation [4]. Hence, we consider the proposed optimization for if-else trees from [4] as the state-of-the-art approach.

3 Problem Analysis

After training of a random forest model (e.g. with scikit-learn [10]), the model is derived in a logic representation (e.g. encoded in JSON). Executing this model without special operating system or library support requires a realization in a programming language and compilation to machine code. The realization of decision trees and random forests as if-else trees, as introduced by [1] intensively utilizes instruction caches during the execution, as the entire tree structure is encoded in instructions itself. Only loading of the data point for inference is mandatory from data memory and therefore uses data caches. Listing 1.1 depicts an example of the implementation of a single tree node as an if-else tree in C++.

```
1 if(pX[3] <= (float) 1.500000){
2     return 1;
3 } else { ...
```

Listing 1.1. C++ node example

It can be seen that the loading of the data point (stored in pX) is an array access and therefore a data memory access. The split value, which is used to decide in combination with the data point if the left or right subtree should be further traversed, is immediately encoded in the source code, also the prediction value is immediately encoded with the return statement. To illustrate the conversion to assembly code, we investigate the assembly code for an X86 machine, produced by the gcc compiler in version 11.1.0 in the following. Later in this paper we consider both X86 and ARMv8 architectures.

```
1 movss    0xe50(%rip),%xmm9
2 comiss   0xc(%rdi),%xmm9
3 jmp      2fa0
```

Listing 1.2. Assembly node example

Listing 1.2 illustrates the relevant assembly code for the implementation of the node from Listing 1.1. Line 1 is responsible for loading the split value, Line 2 loads the feature value from the data point and performs the comparison to the split value. Line 3 then performs the according jump. Counter intuitively to the C++ implementation, the split value is not encoded as an immediate value, but rather leads to a data memory load within the `comiss` (compare scalar ordered single-precision floating point) instruction. Since the X86 instruction set does not offer immediate values for floating point instructions, the compiler decides to place the split values at a central position in data memory and translate the accesses to regular data loads[1]. Please note that the `movss` instruction (which loads a floating point number to a register) uses the immediate encoding for the offset within main memory, but not for the floating point constant itself. In consequence, two out of the three relevant instructions for an if-else tree perform data accesses and utilize the data cache. The motivational concept of intensively utilizing instruction memory and caches for if-else trees does not hold all along.

[1] This observation is not necessarily bounded to the X86 architecture, the ARMv8 architecture neither does offer such a feature.

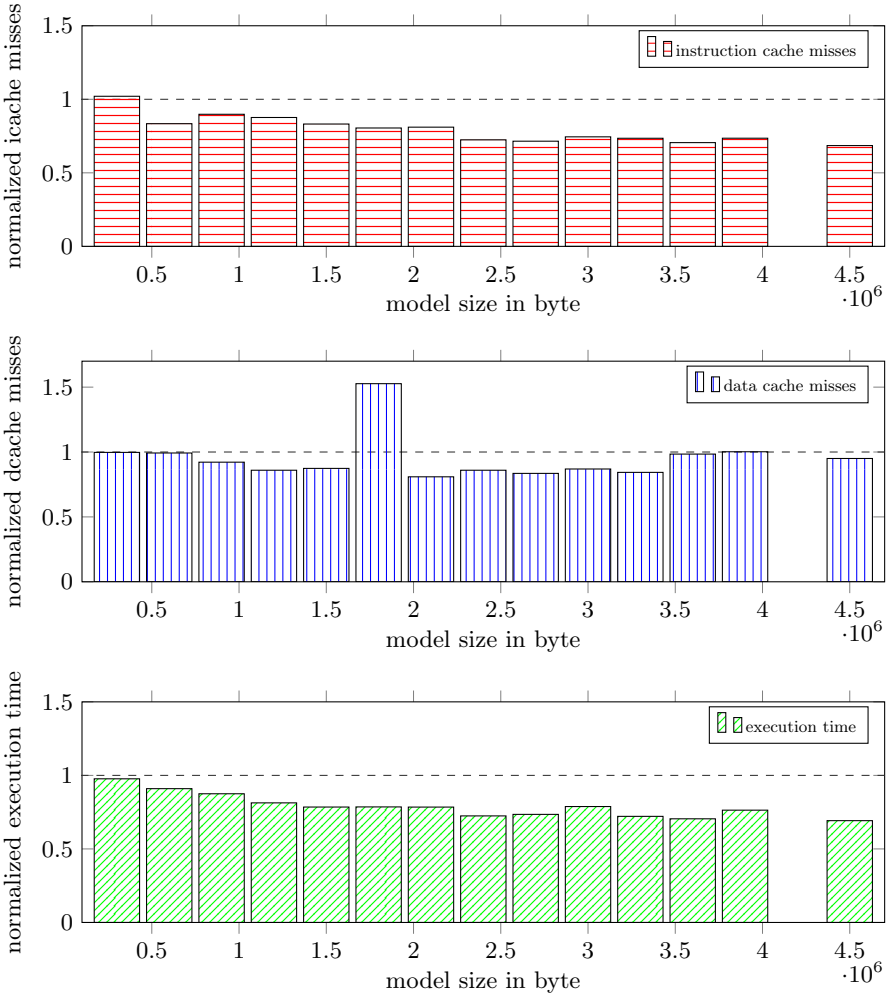

Fig. 1. Execution time, icache misses and dcache misses for if-else tree optimization

To further illustrate the impact of this condition, we investigated the state-of-the-art implementation of if-else trees from Chen et al. [4]. Specifically, we studied two possible implementations of the same logic model in the following: 1) a naive implementation of if-else trees, where every node in the logic tree structure becomes an if-else block and the left and right subtree is placed within the corresponding if or else block. 2) the generated trees with the state-of-the-art optimization [4], where the tree is reordered with regards to the branch probability within every node. This reordering aims to optimize cache prefetching and minimize the amount of cache misses. We generated a large set of random forests for data sets, which resulted in floating point split values for the naive

and the optimized if-else tree implementation. As datasets we chose from the UCI machine learning repository [6]: the *EEG Eye State Data Set* (eye), the *Gas Sensor Array Drift Data Set* (gas), the *MAGIC Gamma Telescope Data Set* (magic), the *Sensorless Drive Diagnosis Data Set* (sensorless) and the *Wine Quality Data Set* (wine), which are all classification data sets. We divided all datasets into 75% training data and 25% test data. We did not perform hyper-parameter tuning but rather tune the maximal depth of the decision trees in order to derive different sized models. We executed these models on a X86 server machine (2x AMD EPYC 7742, 32kB L1 i/dcache, 256GB RAM) and compare them with respect to their execution time, their amount of misses in the level 1 instruction cache and the amount of misses on the level 1 data cache.

Figure 1 depicts the recorded results from the execution of the implementations with the performance analysis tools for Linux (Perf). We normalize the results of the optimized implementation (applying the optimization method from Chen et al. [4]) to the naive implementation. We tuned two knobs: The maximal depth of single trees and the amount of trees within the ensemble. The resulting size of the model is based on the measurement of the binary size of the implementation after the compilation, which is illustrated along the x axis. Please not that the binary size of the model is only indirectly controlled by the maximal depth and the amount of trees, hence not for every size on the x axis also a model is generated. We consider the optimized implementation, even if it may result in a different binary size, to the original binary size from the naive implementation. Hence, even if optimizing the model increases the binary size, the performance still is compared to the corresponding naive implementation of the same tree structure. An increase in the binary size potentially causes a higher amount of cache misses, which is then reported in the normalized data. We further group the models in size groups (0kB-300kB, 300kB-600kB, ...) and compute the geometric mean of the normalized improvements. This value is the ultimately depicted in the figure. The green bars with diagonal lines indicate the reduction in total execution time, the red bars with horizontal lines indicate the reduction in L1 icache misses, and the blue bars with vertical lines indicate the reduction in L1 dcache misses.

It can be observed that, though the optimization reduces the total execution time and amount of icache misses for large models[2], the amount of L1 dcache misses is not reduced similarly. This stems from the fact that the if-else tree optimization proposed by Chen et al. [4] only modifies the sequence of the source code in order to reduce the amount of L1 icache misses. The placement and loading of the split values is not considered and thus not handled in the optimization. When the dcache misses can be reduced as well, a further reduction of the execution time can be possible. Furthermore, load can be released from the instruction memory, which may comfort other applications within the system.

[2] The optimization targets to optimize the memory layout, such that cache misses are reduced. Thus, effects likely only can be observed when the model size exceeds the cache capacity, which is only for larger models the case.

Observing this shortcoming in the existing optimization motivates us to develop a new optimization technique, which specifically focuses on the optimization of dcache misses by handling the loading of the split values in a dedicated manner. One trivial method is to round the floating point split values to integer values and subsequently encode them in the immediate field of the instructions itself, such that they do not need to be loaded from data memory at all. This, however, potentially induces a loss in accuracy due to the rounding of the split values. In this paper, we alternatively present an implementation, where the full floating point split value can be encoded in the immediate field of instructions and therefore also omits the need to load the split values from data memory.

4 Immediate Encoding in If-Else Trees

As mentioned before, when it comes to the optimization of the cache behavior of if-else implementations of decision trees, both cache types, i.e. the instruction cache and the data cache, need to be handled. In general, optimization methods profile the execution of the decision tree on the training dataset and determine empirical branch probabilities. These probabilities are used subsequently to shape the tree implementation in an optimized manner. When the total model size exceeds the capacity of a cache, which likely happens for kilobyte sized level 1 caches, cache misses cannot be avoided during execution of the tree.

Hence, the optimization target is to reduce the amount of cache misses in order to improve the total execution time of the decision tree. Such optimizations usually can exploit two aspects: 1) the tree is shaped in a way that frequently accessed parts of the decision tree are less likely evicted from the cache as in a naive implementation and therefore do not cause cache misses on access, and 2) the tree is shaped in a way that automatic prefetching of (spatial) local memory contents is utilized to load parts of the tree into caches before they are accessed and thus omit cache misses at the access time itself. To shape the tree itself, data memory and instruction memory needs to be distinguished. Data memory is usually used to store variables and arrays. If a tree implementation uses large arrays, changing the layout of the array allows to shape the tree. Instruction memory is used to store the instruction sequence of the tree itself. If the tree implementation uses many instructions, changing the sequence of instructions allows to shape the tree regarding the behavior of instruction caches.

As motivated before, the naive implementation of an if-else tree in C++ uses data memory to load both feature values and split values. Access to the feature values cannot be omitted and hardly be optimized, since the input tuple is not created by the tree implementation itself. Thus, data memory accesses to the feature values are compulsory. In consequence, optimization of the data memory accesses for the split values is challenging, since these accesses are necessarily interleaved with the accesses to the feature values. Therefore, the implementation we propose in this paper alters the loading of the split value from data memory to instruction memory. Subsequently, the tree is shaped by ordering the instruction sequence with respect to the behavior in the instruction cache.

```
1  __rtitt_lab_27_0:
2      movss   12(%1),          %%xmm1
3      //0x3fc00000=1.5
4      mov     $0x3fc00000,     %%eax
5      movd    %%eax,           %%xmm2
6      comiss  %%xmm1,          %%xmm2
7      jnb     __rtitt_lab_29_0
```

Listing 1.3. Optimized assembly implementation (X86)

```
1  "__rtitt_lab_27_0:"
2      ldr     s1,              [%1, 12]
3      //0x3fc00000=1.5
4      movz    w2,              #0x0000
5      movk    w2,              #0x3fc0,     lsl 16
6      fmov    s2,              w2
7      fcmp    s1,              s2
8      b.le    __rtitt_lab_29_0
```

Listing 1.4. Optimized assembly implementation (ARMv8)

Based on the arch-forest framework[3], used in [4], we implement a new code generator module for the generation of our optimized if-else tree. The code generator does not generate C or C++ code, but rather directly generates X86 or ARMv8 assembly code, which is embedded by inline assembly to the rest of the framework. In order to explain the assembly implementation, we focus on the example node from Listing 1.1.

Listing 1.3 illustrates the output of our code generator for the example node. In line 2, similarly as in the compiler generated code, the feature value is loaded from data memory, which cannot be omitted. Afterwards, the split value (1.5) is converted to IEEE-754 32 bit representation in line 4 and loaded as a bitmask to a general purpose register[4]. The `movd` instruction subsequently copies the register content without conversion to a floating point register and in line 6 and 7 the according comparison and jumps are executed.

Listing 1.4 similarly depicts an example of the ARMv8 code, which is generated by our code generator. The key difference is that ARMv8 does not offer pseudo instructions to load 32 or 64 bit immediate values, thus we decompose these into a set of `movz` (move and zero contents before) and `movk` (move and keep contents) instructions with according bit shifts. The `fmov` instruction in ARMv8 is the respective instruction to move contents from a general purpose register to a floating point register without conversion.

[3] `https://github.com/tudo-ls8/arch-forest`

[4] Our generator also supports double precision floating points; the code is generated accordingly on demand.

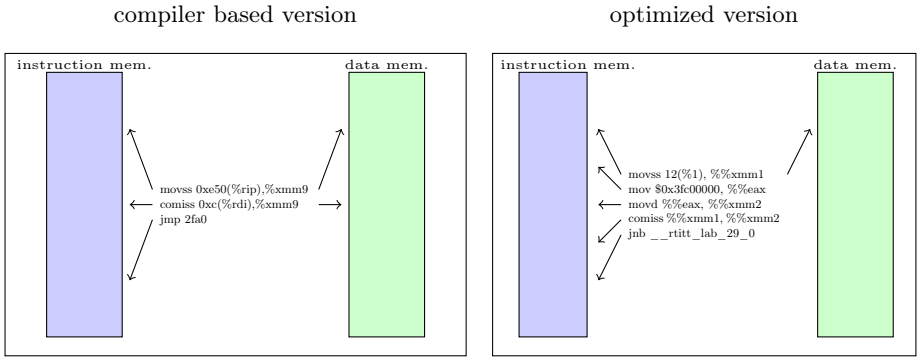

Fig. 2. Optimized loading of constants from memory

Figure 2 illustrates the difference in memory accesses between the compiler generated code and our generated code for X86. All instructions, by default, access instruction memory, since the instruction has to be loaded from instruction memory. In the compiler generated version, two out of three instructions in addition access data memory, in our optimized version only one out of five instructions additionally accesses data memory. Despite moving the split value entirely to the instruction memory, we also inherit the code sequence optimization from [4] in our code generator. For every node, the relative probability to visit the left or right child is compared and the more probable child is placed as the subsequent instructions. The less probable child hence is labeled and targeted by the jump / branch instruction. Implementation wise, this requires a swap of the branch condition, since the branch must be taken either on the $\leq$ or on the $>$ condition. We achieve this by either generating a `jnb` (jump if not below) / `b.le` (branch if less or equal) or a `jb` (jump if below) / `b.gt` (branch if greater than) instruction.

In order to integrate our code generation in a generally applicable shape, we implement all possible combinations for datatypes within if-else trees in the code generator. This includes various combinations of datatypes for the feature and for the split values, since the comparison has to be realized accordingly. Our code generator allows to generate if-else trees for 32 and 64 bit floating point split values, including the optimization from [4], in assembly code and eliminates a large portion of data memory loads, at the cost of few additional instructions, which are used to encode the data directly in the immediate field. Thus, the data cache misses are likely reduced when employing this implementation.

Beyond our implementation, this method is applicable to other models and structures as well. Floating point constants are required for a large set of machine learning models, e.g. neural networks or simple regression models. Such models are usually trained by adjusting a set of constants (weights, parameters, etc.), which are then incorporated for computation during inference. Since the computation is implemented as code execution in a CPU based variant, con-

stants can be similarly immediately encoded and possibly allow a performance improvement of other models.

5 Evaluation

In order to evaluate our proposed implementation of encoding the split values in the immediate fields of integer instructions, we focus on two central aspects: 1) the reduction of data cache misses and 2) the effect of the reduction on the total execution time. For the evaluation, we investigate again the data sets from the UCI machine learning repository [6]: The *EEG Eye State Data Set* (eye), the *Gas Sensor Array Drift Data Set* (gas), the *MAGIC Gamma Telescope Data Set* (magic), the *Sensorless Drive Diagnosis Data Set* (sensorless) and the *Wine Quality Data Set* (wine). These data sets are all classification data sets. We used the arch-forest framework together with our custom code generator to generate ensembles of different amount of trees and tree sizes for all data sets. We generated subsequently three implementations for every tree: 1) a naive if-else tree implementation without any optimization, 2) the optimized if-else tree implementation from Chen et al. [4] as the state of the art and 3) our assembly-based implementation, as presented in Section 4. As test platforms, we chose four different systems, two server systems with X86 and ARMv8 architectures and two embedded systems with X86 and ARMv8 architectures. The system details can be found in Table 1. We executed all generated ensembles on all of the systems

Table 1. Test system details

	CPU	**L1 icache**	**L1 dcache**	**Memory**
X86 Server	2x AMD EPYC 7742	32 kB	32 kB	256 GB DDR4
X86 Embedded	Intel Atom x5-Z8350	32 kB	24 kB	2GB DDR3
ARMv8 Server	2x Cavium Thunder X2	32 kB	32 kB	256 GB DDR4
ARMv8 Embedded	Amlogic S9052	32 kB	32 kB	2 GB DDR3

and used the performance analysis tools for Linux (Perf) to record instruction cache misses, data cache misses and the total execution time for every configuration. We again determined the model size in bytes after compilation to compare the different configurations regarding their final size. Similarly, we consider the optimized implementations for the binary size of the naive implementation and build size groups, which we use to compute the geometric mean and present the results. Thus, even if the model size is increased by the optimization, the normalized ratio still is depicted for the same logic model structure.

Figure 3 depicts the icache misses for the server systems, Figure 4 depicts the dcache misses, and Figure 5 depicts the execution time, respectively. Figure 6

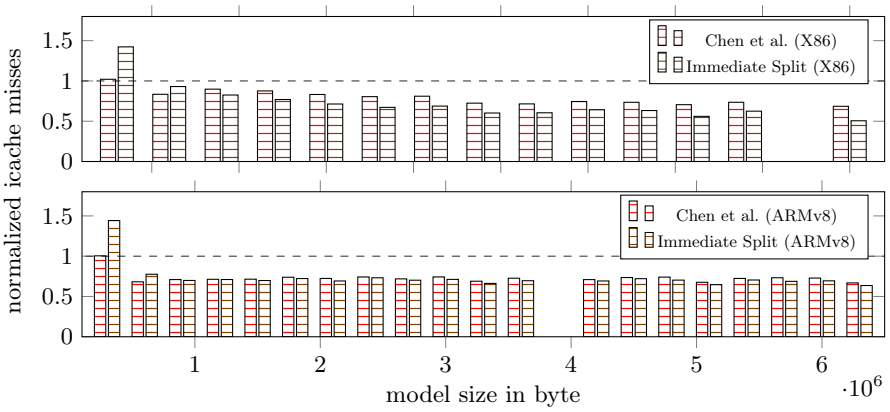

Fig. 3. Instruction cache misses of immediate encoded split values - server

depicts the icache misses for the embedded systems, Figure 7 the dcache misses and Figure 8 the execution time, respectively. We again compute the normalized ratio between the optimized and the naive implementation. Thus, a number larger than 1 indicates worse performance in comparison to the naive implementation. Each figure includes results for the X86 architecture and for the ARMv8 architecture. Comparing the reduction for the optimization from the state of the art and our optimization leads to another, relative improvement, which is illustrated in Table 2. The #IMPROVED and #IMPROVED($> 900k$) values describe in how many of the tested models our optimization performs better regarding instruction cache misses, data cache misses or execution time than the optimization from Chen et al. The latter value only considers models, which lead to a binary size of more than $900kB$. We further compute the improvement ratio by $1 - \frac{\text{Immediate Split}}{\text{Chen et al.}}$. Hence, a number of $+100\%$ for the cache misses would mean that our optimization eliminates all cache misses, which are left after the optimization from the state of the art. We compute this improvement for all data sets and report the geometric mean for models larger than $900kB$ and the peak value in the table.

5.1 Discussion

Generally, it can be observed that for rather small ensembles (up to $\approx 900kB$) a diminished performance can be observed for most configurations. If a small model anyway can be held entirely in the level 1 cache, there is no requirement for any optimization. The optimization, however, induces certain overheads by introducing more instructions, which leads to an ultimate performance decrease. In consequence, this draws the conclusion that the optimization should only be applied in meaningful scenarios, where the ensemble size exceed the level 1 cache size and necessarily produces cache misses. Therefore, we focus on these meaningful scenarios only.

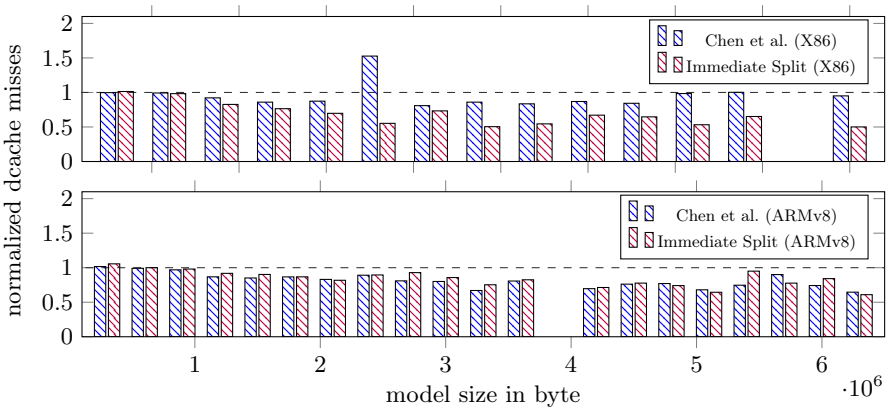

Fig. 4. Data cache misses of immediate encoded split values - server

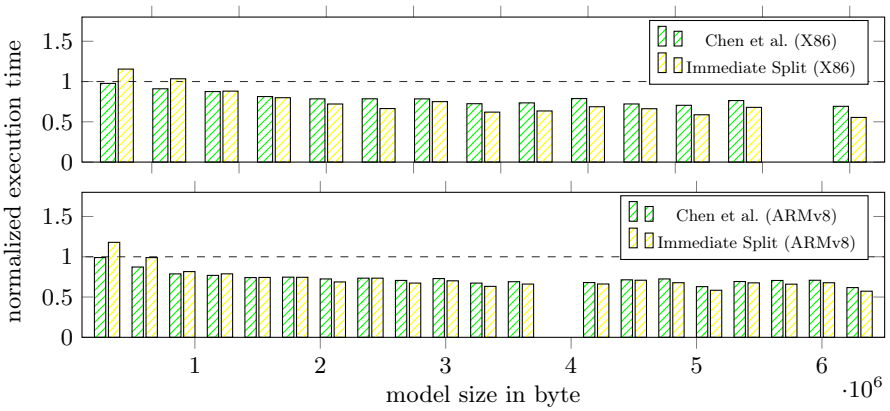

Fig. 5. Execution time of immediate encoded split values - server

Focusing on the instruction cache misses only, it can be seen that for most configurations with large model sizes the amount of icache misses is further decreased by our proposed optimization, compared to the state of the art (on the X86 server system in 95% of the relevant cases in geomean by 14.7%). Considering the data cache misses, considerable reductions can be observed for larger ensembles in comparison to the state of the art as well. For the X86 server, the amount of data cache misses for large ensembles is even reduced in 84% of the relevant cases by up to $\approx 92\%$ in peak. In case of the ARMv8 server, a slighter reduction of dcache misses can be observed, up to $\approx 65\%$ in peak and even an increase of $\approx 4\%$ in geomean for large ensembles. Focusing on the embedded systems, similar behavior can be observed for the data cache, the behavior for the

[6] The geomean values in this table are computed for models only, which are larger than 900 kB.

Table 2. Average and peak improvements compared to [4][6] for server and embedded systems

	X86		**ARMv8**	
	Server	Embd.	Server	Embd.
Time				
#IMPROVED	33.7%	09.6%	34.6%	06.1%
#IMPROVED(>900k)	80.9%	10.7%	68.3%	06.5%
GEOMEAN(>900k)	+08.4%	−16.4%	+01.5%	−17.8%
Peak	+39.5%	+36.7%	+29.7%	+38.7%
ICache				
#IMPROVED	60.0%	33.7%	55.5%	-
#IMPROVED(>900k)	95.2%	60.7%	76.9%	-
GEOMEAN(>900k)	+14.7%	+00.3%	+02.6%	-
Peak	+90.4%	+26.8%	+61.8%	-
DCache				
#IMPROVED	64.1%	76.8%	41.6%	37.5%
#IMPROVED(>900k)	84.5%	100.0%	39.4%	27.8%
GEOMEAN(>900k)	+26.1%	+96.1%	−4.7%	+4.6%
Peak	+92.3%	+99.8%	+65.5%	+87.7%

icache misses contrarily differs[7]. Data cache misses are reduced by up to $\approx 99\%$ in peak and $\approx 96\%$ in average for X86. Instruction cache misses, however, are not significantly reduced on the X86 embedded system. For the ARMv8 embedded system, the improvement of dcache misses as well is comparably lower to the X86 embedded system.

Despite reducing icache and dcache misses, the allover execution time of the optimized implementation matters. In general, it can be observed that a high reduction in dcache misses does not necessarily result in a high reduction in execution time. For large ensemble sizes on the server machines, a consistent reduction of execution time can be however observed for our proposed optimization. The majority of relevant cases (more than 65%) yields an improvement in execution time on the X86 and ARMv8 servers. The improvement is up to $\approx 40\%$ in peak for X86 and ARMv8, compared to the state of the art. For small ensemble sizes, it can be observed that the execution time is increased beyond the naive implementation with our optimized implementation. In theses cases, the additional overheads due to the immediate encoding cannot be leveraged by the improvement. Investigating the embedded systems, the execution time can only be improved for few cases ($\approx 10\%$ on the X86 and $\approx 6\%$ on the ARMv8 system). In geomean, the execution time is enlarged for the relevant cases, although the amount of dcache misses is drastically reduced for X86. This suggests that dcache misses are not the limiting factor for the execution in this scenario. Furthermore, this also implies that the CPU architecture is an important factor to the intended reduction of dacache misses with our optimization.

[7] The ARMv8 system we use does not allow tracking of icache misses with perf.

Although the results reveal that our proposed optimization cannot improve performance unconditionally, especially for small model sizes and embedded systems, we can report scenarios with a massive reduction of dcache misses and also a reduction of icache misses. Such a reduction can be useful to comfort parallel running applications. In several cases, the reduction of cache misses further directly relates to reduction of total execution time. When generating implementations, various versions can be profiled on the training data set so the best implementation can be chosen. Thus, for the cases where we achieve a worse result, the implementation of Chen et al. can still be chosen. Similarly for small models, where the optimized implementation induces a high overhead, the native implementation can be chosen.

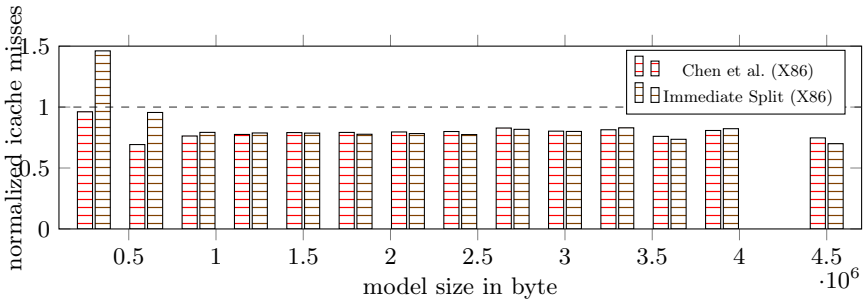

Fig. 6. Instruction cache misses of immediate encoded split values - embedded

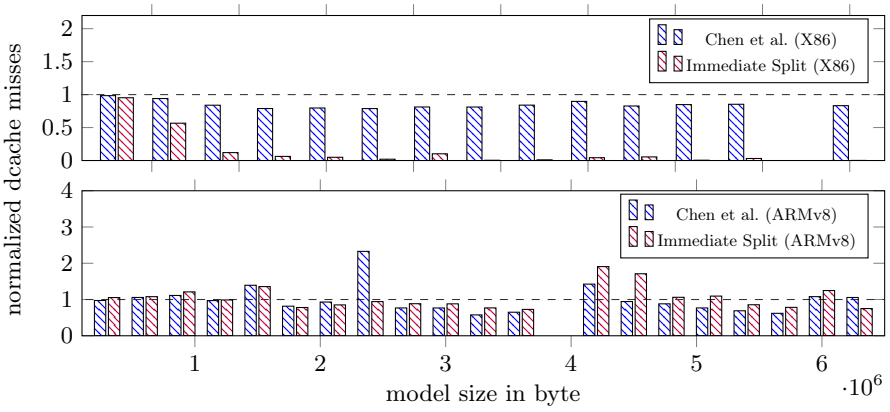

Fig. 7. Data cache misses of immediate encoded split values - embdedded

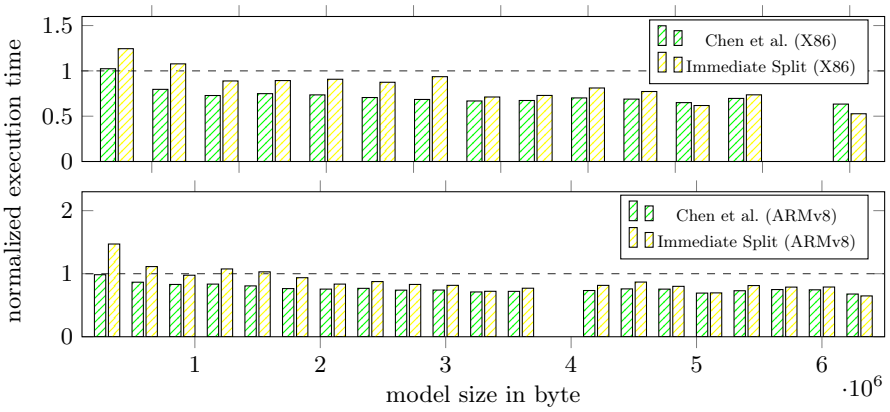

Fig. 8. Execution time of immediate encoded split values - embedded

6 Conclusion and Outlook

In this paper, we investigate the realization of random forests as if-else trees, which is one popular way of implementing random forests. We show that, counter intuitively to the design principle, data memory accesses and therefore potentially data cache misses play a big role in if-else tree implementations. The state-of-the-art optimization for if-else trees does not target data memory accesses specifically, therefore we propose an optimized implementation in this paper, where we eliminate a huge portion of data memory accesses. Experimental evaluation shows that our optimization can reduce the amount of data cache misses by up to 99% upon the state-of-the-art and can even lower the allover execution time by up to 40% on server systems. We further conclude that the overheads, which are introduced by our optimization, can only be leveraged for model sizes, which exceed the size of the level 1 caches. Thus, the optimization should be only applied in these cases. On embedded systems, the execution time is overall not significantly lowered, although the amount of cache misses can be drastically reduced. Hence, different aspects should also be explored. The implementation of our code generation fully supports X86 and ARMv8 architectures with different width integer and floating point data types. The source code is available at `https://github.com/tu-dortmund-ls12-rt/arch-forest/tree/immediatesplittrees`.

For future work, we plan to include model-based optimizations to our implementation, where we try to estimate the cache behavior during compile time with precise models and layout the trees accordingly. We also plan to investigate the relation between cache misses and execution time more intensively.

Acknowledgement

This work has been supported by Deutsche Forschungsgemeinschaft (DFG) within the project OneMemory (project number 405422836), the SFB876 A1 (project number 124020371), and Deutscher Akademischer Austauschdienst (DAAD) within the Programme for Project-Related Personal Exchange (PPP) (project number 57559723).

References

1. Asadi, N., Lin, J., de Vries, A.P.: Runtime optimizations for tree-based machine learning models. IEEE Transactions on Knowledge and Data Engineering **26**(9) (Sept 2014)
2. Buschjäger, S., Morik, K.: Decision tree and random forest implementations for fast filtering of sensor data. IEEE Transactions on Circuits and Systems I: Regular Papers **PP**(99), 1–14 (2017). https://doi.org/10.1109/TCSI.2017.2710627
3. Buschjäger, S., Chen, K.H., Chen, J.J., Morik, K.: Realization of random forest for real-time evaluation through tree framing. In: 2018 IEEE International Conference on Data Mining (2018). https://doi.org/10.1109/ICDM.2018.00017
4. Chen, K.H., Su, C., Hakert, C., Buschjäger, S., Lee, C.L., Lee, J.K., Morik, K., Chen, J.J.: Efficient realization of decision trees for real-time inference. ACM Transactions on Embedded Computing Systems (TECS) (2022)
5. Dato, D., Lucchese, C., Nardini, F.M., Orlando, S., Perego, R., Tonellotto, N., Venturini, R.: Fast ranking with additive ensembles of oblivious and non-oblivious regression trees. ACM Transactions on Information Systems (2016)
6. Dua, D., Graff, C.: Uci machine learning repository (2017), `http://archive.ics.uci.edu/ml`
7. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A., Kaldewey, T., Lee, V., Brandt, S., Dubey, P.: FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM (2010)
8. Lucchese, C., Nardini, F., Orlando, S., Perego, R., Tonellotto, N., Venturini, R.: Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 73–82. ACM (2015)
9. Lucchese, C., Perego, R., Nardini, F.M., Tonellotto, N., Orlando, S., Venturini, R.: Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles. In: SIGIR 2016 - Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (2016). https://doi.org/10.1145/2911451.2914758
10. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Édouard Duchesnay: Scikit-learn: Machine learning in python. Journal of Machine Learning Research **12**(85) (2011)
11. Prenger, R., Chen, B., Marlatt, T., Merl, D.: Fast map search for compact additive tree ensembles (cate). Tech. rep., Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2013)

12. Van Essen, B., Macaraeg, C., Gokhale, M., Prenger, R.: Accelerating a random forest classifier: Multi-core, gp-gpu, or fpga? In: Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on. pp. 232–239. IEEE (2012)
13. Ye, T., Zhou, H., Zou, W.Y., Gao, B., Zhang, R.: RapidScorer: Fast tree ensemble evaluation by maximizing compactness in data level parallelization. In: Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (2018). https://doi.org/10.1145/3219819.3219857