technische universität
dortmund

Bachelorthesis

Analysis and Optimization for Hoeffding Tree

Xiaoshi Wang

13.04.2020

Supervisors:
Prof. Dr. Jian-jia Chen
Dr. Kuan-Hsun Chen

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Abstract

Data stream mining is an important topic of machine learning, and Massive Online Analysis (MOA) [3] as a popular framework specifically for mining data streams is well known. It implements different classifiers for the data stream mining including Hoeffding tree, which is the focus of this thesis. By checking the underlying implementation, we notice that the cache behavior of Hoeffding tree is not taken into consideration and the implementation can be further improved. We preliminarily explore if the proposed approaches in [4] can also benefit the Hoeffding tree in MOA and introduce the ideas of different approaches to implement the method proposed by them which uses the path probabilities of the nodes to reduce cache misses. Since Hoeffding tree continues growing during the whole process and the method is designed for a static decision tree, we convert Hoeffding tree to a static version of native tree and apply the method on it. We introduce a way using indices without changing the tree structure to store all Hoeffding tree nodes into a shared array. Then we implement the method in [4] based on the shared array and run the sorting algorithm every time MOA does the result sampling. From the results of MOA, we have a slight average speed-up of 2%, whereas the VTune results show that the cache behavior is still not under good control.

Contents

1 Introduction

Data stream mining is the process of extracting knowledge structures from continuous, rapid data record [6]. The goal is to predict the class or value of new instances in the data stream given some knowledge about the class membership or values of previous instances in the data stream. Massive Online Analysis (MOA) [3] is a popular open-source software specifically for mining data streams. Because MOA is written in Java, so it has also a great extensibility. There are plenty of classifiers and data streams that are already implemented for the user.

In this thesis we will focus on one of its classifier called Hoeffding tree, which is widely used in data stream mining. Originally, Hoeffding tree was introduced by Domingos and Hulten [5]. In MOA, the implementation combines the basic theoretical Hoeffding tree algorithm with plenty of practical enhancements from Very Fast Decision Tree (VFDT) algorithm. It can learn from the data stream and perform prediction by choosing the majority class at each leaf. The predictive accuracy can be increased by adding Naive Bayes models at the leaves of the trees [9]. There are also many variations from the original Hoeffding tree such as Hoeffding adaptive tree and Vertical Hoeffding tree. We only explore the binary Hoeffding tree in this thesis.

This thesis is structured as follows: To understand the thesis, we review three aspects of the related work: The first is the Hoeffding tree. We introduce the backbone of Hoeffding tree, the algorithm and other related concepts. Then we introduce the method from [4], which uses the probabilities to reduce the cache misses. The third aspect is the CPU caching as well as the type of cache misses (see Chapter 2). Afterwards, we introduce the ideas of different approaches for the implementation (see Chapter 3). In Chapter 4, we discuss the implementation in detail in two parts, which are the array implementation and the probability sorting method implementation. In Chapter 5, we show the results from the graphical user interface of MOA and from VTune. Afterall, Chapter 6 concludes the thesis.

1.1 Motivation

For decision trees, a general problem is that the data structure of split node requires recurrent memory allocations for saving the new generated split nodes and they are discontiguously allocated in memory which causes more overhead and weakens the data locality of the program [4]. Hoeffding tree is capable of learning in a short time for each instance and is suitable for high-volume data streams [5]. The question remains open: How to improve the performance by taking cache behavior into consideration. Considering Hoeffding tree is a type of decision tree, it has the same problem and even worse effects due to the need of growing tree on-the-fly. This reduces the performance due to penalty of addition cycle induced by cache loading. So cache-misses should be reduced by a method which enforces the nodes to be saved in a more compact way.

In the "Tending Your Garden - Realization of Random Forest for Real-Time Evaluation through Tree Framing" [4] Buschjäger et al. introduced a way of optimization for the decision tree execution, which makes the random forest have a better performance in the instruction cache by considering the path probability of all paths. Nodes on the path with higher probabilities can be loaded at first until the accumulated size of the selected nodes exceeds the budget.

In [4] the method is built for decision trees that are already trained and do not grow anymore. However, Hoeffding tree is a decision tree designed specifically for an on-line training scenario where the data stream continuously getting filtered into it and the tree never stops growing during the whole process, unlike a decision tree already located fixed in an array and do not change again later. Nodes inside the Hoeffding tree can also be deactivated, which means that we also have to take care of updating the tree structure during the whole time. In this thesis, we investigate how to apply the methodology of [4] as a backbone on the Hoeffding tree to improve the performance in terms of run time. And the node splitting is constrained to binary split. To implement and evaluate the results, we choose a framework called Massive Online Analysis which has different types of implementations of Hoeffding tree.

1.2 Framework Massive Online Analysis

Massive Online Analysis (MOA) is a open-sourced Framework written in Java [3]. One of the resons to study MOA in this thesis is that it is popular (cited by 1382 times so far) and it implements Hoeffding tree. With the graphical user interface of MOA, tasks can be easily created and configured.

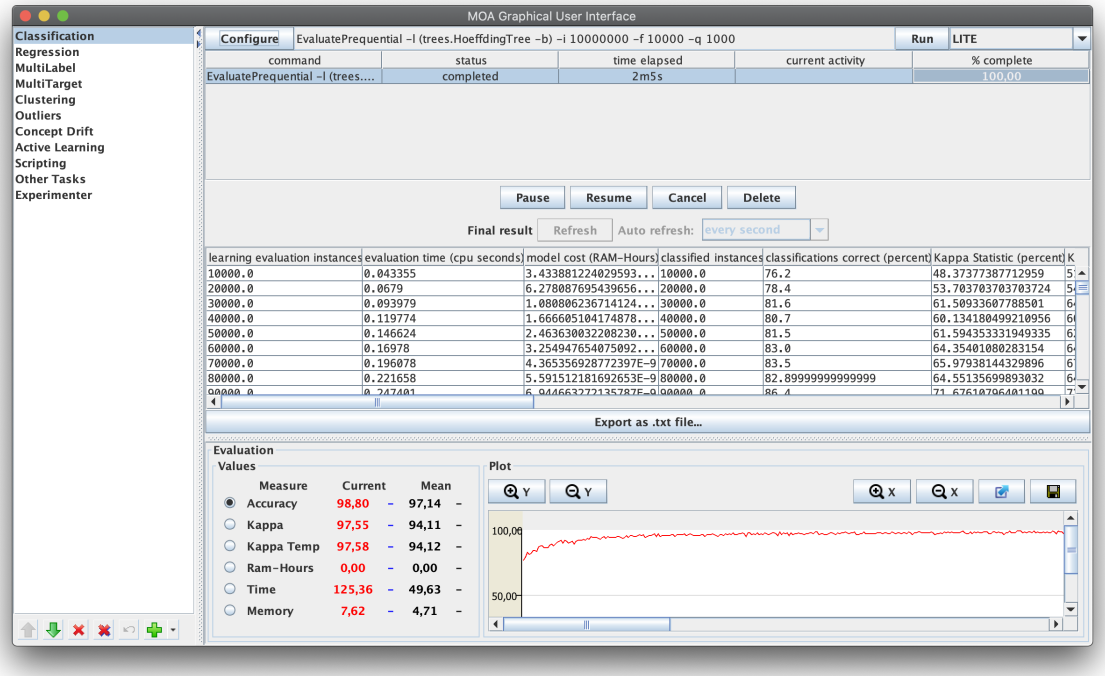

Figure 1.1: MOA Graphical User Interface

It supports the following expeimental setups:

- configurable settings for data stream
- plenty of different existing algorithms and
- new algorithms, streams and evaluation methods.

As Figure 1.1 shows, we can easily configure a task from multiple different task types. The result values, i.e. the accuracy and Kappa statistics are showed graphically. We can see how many nodes are created, how many leaf nodes are there, how big is the size of the tree along with others. The column above shows the complete configuration parameters of a task, which we can use to run the same task in command line for the VTune evaluation at the end. `trees.HoeffdingTree -b)` represents the classifier and the binary split because this thesis focus on the `HoeffdingTree` with the data stream from the default stream generator for Hoeffding tree `RandomTreeGenerator`.

1.2.1 Data Stream and MOA

MOA is built to solve the problem of a continuous supply of data, i.e., an input source that sends one training instances for inspection a single time only. `RandomTreeGenerator`

is one of the stream generator in MOA and it is designed to generate instances through a randomly generated decision tree.

The first step of `RandomTreeGenerator` is to generate a decision tree. By calling `generateRandomTree` it starts from the tree root, a random attribute is selected as the criterion for splitting at each internal node. At the end, each leaf takes one class label randomly and all the other nodes without classes are simply internal split node. After this the tree generating process is completed. Once we have a decision tree, the instances can be easily generated.

An instance $< x,\ y >$ for the data stream, where x is a set of attributes with values. For example $x = (62.1, 2.986, 3.3)$ which means that our instance contains three attribute and attribute1 has a value of 62.1 and so on. And the y in $< x,\ y >$ is representing the value of class. Since the basic structure of an instance is clear, now the generating process is executed with two small steps: First, a vector x containing multiple attribute values should be generated by using the Java class `Random`. Secondly it uses the generated vector x and compares them with the tests in the decision tree which is prepared. After this we have an instance ready for data stream.

For the repeatability and the reproducibility of the tests, the random generated decision tree should also be reproduced to ensure that the results of multiple tests with different configurations can be compared with the same input data stream. In MOA it is already well implemented, we can ensure that just by enabling some of the parameters: `-f` guarantees that every time the randomly generated decision tree is the same, and `-i` guarantees that the generated instances are always the same. With these parameters, we can easily evaluate the effects with the changed codes.

2 Background and Related Work

As mentioned in Section 1 there are two related techniques to this thesis.

The first one is Hoeffding tree, and the backbone of it: Hoeffding bound. First the algorithm of Hoeffding tree implemented in MOA is introduced. Then the end some of the relevant concepts and techniques are introduced and explained.

The second related work which we look at is the method proposed in [4]. Buschjäger et al. introduce a way of optimization for decision trees so that the cache misses can be reduced. They use the probabilities of each path to rearrange the nodes and achieve a significant speed-up. In this chapter we introduce this method and the differences between the decision tree considered in [4] and Hoeffding tree.

At the end we introduce the caching behavior of CPUs and three different types of cache misses.

2.1 Hoeffding Tree

Hoeffding trees were introduced by Domingos and Hulten in [5]. It is a type of decision tree designed for data stream mining. As the data stream sends instances continuously to the Hoeffding tree, the tree learns from the instances decides whether it is needed to split based on current collected statistics. It grows and splits dynamically during the whole learning process. In the following, the backbone behind the Hoeffding tree, Hoeffding bound is introduced firstly to show when does the Hoeffding tree split and grow:

2.1.1 Hoeffding Bound

Hoeffding bound (Chernoff-Hoeffding bound) is widely applied when dealing with large data sets in modern machine learning. It provides a convincing bound of splits in Hoeffding tree. Combined with other information the decision of splitting can be formally determined. In a Hoeffding tree every internal node contains a test related to different

attribute values. The input examples will be sent down to different paths through these tests. The crucial decision for growing the Hoeffding tree now is when to split a node, and what to use as our test for the node.

There are many popular criterias for choosing the spilt tests in a decision tree, and the most common one is *information gain*. Information gain measures how much information does an event carry when it happened. An event with smaller probability carries more information than the events with higher probabilities. So the information gain of an event is always inversely proportional to the probability that it will happen and also cannot be negative. It measures how much 'purity' is gained in each subset of a spilt. We use entropy to show the purity of subsets. Entropy shows us how much information we expect from a specific event before its happening.

Suppose we have a distribution of class labels with fractions

$$p_1, p_2, ..., p_n$$

which are the probabilities we need, the entropy will be calculated with:

$$entropy(p_1, p_2, p_3, ..., p_n) = \sum_{i=1}^{n} -p_i \log_2 p_i \tag{2.1}$$

the negative symbol makes sure that the entropy is a positive value or 0. And we take $\log_2 p_i$ not for a specific reason but just following the tradition of information theory. This information gain helps us to do split decision in every node. For an input as data stream, Domingos and Hulten introduce a way making the same decision with the help of the Hoeffding bound[8]. After n independent observations and if we have a probability of $1 - \delta$, the difference between the true mean and the estimated mean of the random variables in the range R cannot exceed:

$$\epsilon = \sqrt{\frac{R^2 \ln 1/\delta}{2n}} \tag{2.2}$$

The Range of value R is the vase 2 logarithm of number of possible class labels and δ is the desired probably that the correct attribute is chosen at every point in the tree. The split confidence parameter is one minus the desired probability, and this value is set default of 10^7. So as our Range of values R and the split confidence are both fixed, the Hoeffding bound will be only reversely proportional to the number of observations n. Now the separation will be related to the difference in gain between two attributes with the best information gain value and the Hoeffding bound. For example if we have two best attributes with the information gain difference 0.4, and the Hoeffding bound ϵ is 0.1. It

means that from now on it would be only a positive separation if the information gain difference is at least 0.2.

2.1.2 The Basic Algorithm

The tree starts with a single root node, which is also a leaf node. It will go through all the input training examples and for each input we do the following actions: First we flush all the examples down to leaf nodes, and then the statistics in each leaf for checking split criterion will be updated. After the estimated value $\bar{G}_l(X_i)$ according to the information gain for each attribute the example carries is calculated, two of the best performing attributes X_a and X_b with highest estimated value will be selected. Now the Hoeffding bound is carried out with $\epsilon = \sqrt{\frac{R^2 \ln 1/\delta}{2n_l}}$. If the $X_a \neq X_\emptyset$ and the difference between the best two estimated value $\bar{G}_l(X_a) - \bar{G}_l(X_b)$ is greater than the Hoeffding bound ϵ or $\epsilon < \tau$ where τ is a fixed value for tie-breaking, (Tie-breaking means the situation if we have two or more than two attributes with identical values, the tree will stop growing. To solve this the tie-breaking parameter τ is introduced. If the Hoeffding bound is smaller than the tie-breaking parameter, no matter how close the candidates are performing, the tree will split on the current best attribute.) the node l will be replaced with a split node that splits on X_a, which caused the tree to grow. The implementation will be explained in detail later in Section 3.2.

2.1.3 Related Concepts

Hoeffding bound is the core idea of Hoeffding tree, but there are also a lot of important techniques that help the tree growth. We introduce in the following some of the techniques implemented in Hoeffding tree that are important and relevant to us:

Sufficient statistics

The sufficient statistics is a value indicating if it has enough statistics in this leaf at the moment to start calculating the information gain by executing the possible splits. This saves huge amount of resources if we do not calculate every time for each node to check if we have a better information gain with multiple options of splits. The statistics will be usually stored in a separate table to have a efficient storage instead of stored in the leafs.

```
Let HT be a tree with only a leaf (root)
for all instances do
  Sort instance to a leaf l using HT
  Update sufficient statistics in l
  Increment n_l, the number of instances seen at l
  if n_l mod n_min = 0 and instances seen at l not all of same class then
    Compute Ḡ_l(X_i) for each attribute
    Let X_a be attribute with highest Ḡ_l
    Let X_b be attribute with second highest Ḡ_l
    Compute Hoeffding bound ε = sqrt(R^2 ln 1/δ / 2n_l)
    if X_a ≠ X_∅ and Ḡ_l(X_a) − Ḡ_l(X_b) > ε or ε < τ then
      Replace l with an internal node that splits on X_a
      for all branches of the split do
        Add a new leaf with initialized sufcient statistics
      end for
    end if
  end if
end for
```

Algorithm 2.1: Algorithm of Hoeffding tree [5]

Grace Period

The algorithm shows that the tree has to evaluate the information gain of attributes for each input training example. This will obviously cost some unnecessary performance problems since that only one example can not be such important to make a significant effect on the calculation. So it will save the calculating resources if we could set a proper value that shows the number of examples the needs to wait after last calculation. With this parameter, the growth of Hoeffding tree will be slowed down to prevent that the tree grows too fast and splits too much. According to the experiments by Pedro Domingos and Geoff Hulten [5], with a 32MB memory limit, the accuracy of a tree with the grace period n_min is significantly better than a tree without grace period.

2.1.4 Implementation of Hoeffding Tree in MOA

The Structure of the whole `HoeffdingTree.java` is complicating. There is a high amount of inner classes inside of the Hoeffding tree, which reduces the code readability. To understand how Hoeffding tree in MOA grows and splits, it is important to know in which situation which node is applied. Due to different learning phases and position of nodes themselves, six different types of nodes are implemented in `HoeffdingTree.java`:

- **Found node**: A static helper class used for finding and returning a node and its parent node.
- **Node**: The basic type of node. It inherits from `AbstractMOAObject` and implements the basic method for the evaluation such as byte size calculation and the sub-tree recursion of this node. All the nodes below inherit from Node.
- **Split node**: The type of node representing inner nodes in Hoeffding tree. Most of the changes happens here and we will discuss its structure later in detail.
- **Learning node**: The leaf node. It contains an array of initial class observations and can also learn from the instances. There are two types of learning nodes:
- **Active learning node**: Normal leaves carrying statistics of data distribution for different attributes.
- **Inactive learning node**: The leaves that are disabled because of the memory management for Hoeffding tree in MOA. The cost of storage of sufficient statistics in the leaves is huge, so some of the leaves can be deactivated and the statistics of them are deleted.

So we have a decision tree with different node types, where all internal nodes are split nodes and all leaf nodes are represented through learning nodes and we know when does the tree growth and splits happen. The question remaining is how is this growth process implemented in code level in MOA. The split node has an attribute of dynamic array, which saves the children of this node itself. Once the split takes place, the new child nodes will be created and has also a new array with dynamic size. The nodes are therefore allocated not contiguously in memory and require new memory block allocations frequently. This is also the main focus in this thesis. We change the way of storage of child nodes to make sure that the nodes can be stored closer to each other and reduce the memory fragmentation.

```
public static class SplitNode extends Node {
    private static final long serialVersionUID = 1L;
    protected InstanceConditionalTest splitTest;
    protected AutoExpandVector<Node> children; // = new AutoExpandVector<Node>();
```

Figure 2.1: Split node structure

2.2 Probabilistic View of Decision Tree

In [4] the binary decision tree in a random forest is optimized by sorting the nodes with their probabilities. They reduce the cache misses in three different angles, which are explained in detail in the following. As the decision tree being executed, each node has its own probability of being called. If a node with high probability is always not loaded into cache, it can cause a high cache misses. They have an approach by looking at the complete path from root to leaf nodes:

First, all paths of a decision tree are saved for the operations later. Then we start from root and look at each time the path probability of left and right child and take the one with higher probability to the front of one big array, the other one node is stored temporarily in a max heap. The heap as a priority queue sorts the entries automatically and the node with maximum probability is always on the top of the heap. Once all the nodes on a complete path are sorted together to the front of the array, the heap pops out the node with the maximum path probability from all the paths that are not selected before. This makes sure paths with higher probabilities can be always quicker completely loaded.

One thing needs to be mentioned here is that the method from Buschjäger et al. is designed for a static scenario where the decision tree is already completely trained and does not grow any more. In the data stream mining scenario the tree grows dynamically and the probabilities changes also as more stream instances get classified. In this thesis we only explore the static scenario. This means that a way of running the method periodically is needed.

2.3 CPU Cache and Cache Misses

To understand the method employed by Buschjäger et al. we first have to understand how CPU caches work. This is necessary because the caching behavior of CPUs can strongly influence the performance characteristics of the code that is being run.

There are primarily two classes of caching we would have to consider: software caching and hardware caching. The focus is, however, primarily on hardware caching, as that seemed to be the most promising route.

It's important to note that there's a wide variety of CPU architectures, all of which handle caching differently. This means that we can't optimize the Hoeffding tree for all CPU architectures at once without forgiving at least some potential gain.

In this thesis, we focus on X86/X86_64 series of architectures. The general principle of hardware caches remain usually the same across different architectures.

In the realm of hardware caching, there's the Translation Look-aside Buffer (TLB), which acts as a kind of cache for translating between logical and physical addresses, and the CPU cache. When we speak about the "CPU cache" however, we have to keep in mind that we're dealing with many different caches, which are abstracted / "hidden away" from the user. For our purposes, we'll only consider the Instruction Cache (I-Cache) and the Data Cache (D-Cache).

Both work roughly the same way: Because fetching new instructions and data from main memory is slow, the CPU has volatile storage specifically for this purpose, which is located closer to the CPU than main memory. This grants us faster access times.

There's a trade-off through: because it's closer to the CPU, it also has to be smaller than the main memory, so as to still fit into the CPU package. Therefore both the I-cache as well as the D-Cache tend to be (depending on the CPU design and model) between one and four megabytes.

Another aspect that must be kept in mind is that all caches that are part of the CPU are used simultaneously by the OS and the application code. Not only does this make directly controlling the cache more difficult, we also can't use all of the cache for our application. In practice this means that we'll never be able to fit the whole Hoeffding tree into the caches at once, which makes selecting which parts of the tree to "force" into cache even more important. In this thesis we focus on the D-Cache and different types of cache misses and see if we can improve them by applying the methods proposed in [4].

2.3.1 D-Cache

The D-Cache typically has multiple tiers, most often three. In newer CPUs, the first tier is shared between multiple logical cores, which has to be considered when using eg multi threaded code or code running across different cores. If the data isn't contained in in the first level of d-cache, it's called an "l1 miss". If this occurs, the second tier will be checked. If an "l2 miss" then occurs, the third tier will be checked. If there was still no cache hit, the relevant / referenced data will be loaded from main memory.

One important characteristic is that the D-Cache is divided into different cache "lines". These are chunks of mostly 64 bytes of data which make up the whole of a single cache. If we suffer a cache miss, the CPU will fetch a whole cache line at once from main memory, even if the requested data is only (for example) 8 bytes big.

Optimizing for D-Cache behaviour is most commonly achieved by packing the data tightly together. That way, we ensure that no cache line is only partially filled, which would mean wasted cache space. It also involves packing the data more "linearly". That way, we won't have memory loads sprayed all over main memory, which lets the load predictor of the CPU more efficiently predict which pieces of data to load into the D-cache.

2.3.2 Type of Cache Misses

There are three different types of cache misses introduced in [4]. The first one is compulsory cache misses, which is caused by the first access to a memory block. This can be reduced by increasing the chance of using the prefetched instructions. For native trees just like Hoeffding tree, they remove the pointers for left and right child and the the split-values at leaf nodes. If the amount of memory can be reduced, then we can load more nodes into the cache, and the compulsory cache misses can also be reduced.

The second type of cache misses are capacity cache misses. They happen when the loaded blocks are discarded due to cache capacity. If a program has a larger working set than the cache, some of the instructions have to be loaded multiple times and again discarded for loading other instructions.

And the third type are conflict cache misses. It occurs if we have a set associative or direct mapped cache and different cache blocks are always replaced with each other because they have the same index. In [4] They propose that improving the data locality helps reducing the capacity cache misses and the conflict cache misses. Without optimization, the nodes are stored discontiguously in memory. So if we can store the node that are often together used, near to each other, it improves the both cache misses. So they rearrange the nodes on the same path together so that the nodes needed on a sequence can have a better chance to be loaded together at once into cache and the distance between nodes to access is reduced as well as the cache misses.

3 Ideas of Different Approaches

There are basically two different approaches to optimizing the memory layout of the Hoeffding trees currently implemented in MOA. We explore first where should the nodes and the tree itself be allocated. Furthermore, these approaches can once again be differentiated into those affecting the algorithmic aspects of the implementation, and those affecting purely the CPU- and cache-related aspects (i.e. those that are hardware related). We come up with a lot of different methods to achieve the same or at least similar to the array allocation like the implementation of Buschjäger et al. [4]. Due to the Java Virtual Machine (JVM) even if we saved elements together in one array, it can not guarantee that in the physical memory they have contiguous memory addresses. The JVM will allocate the new objects in an area that's available in heap and the memory for the heap does not have to be continuous. So at first we look for the help of Off-Heap Memory:

3.1 Off-Heap Memory

The first class of approaches targets the way nodes are stored in memory. Currently, as the nodes are stored using native Java arrays, they are located in the heap section of the JVM-allocated memory. This section is also managed by JVM garbage collection and programmers do not have control of the memory allocation. The new memory blocks are allocated and released all by JVM and this process is not visible to the user.

Figure 3.1: Java Off-heap

One approach would therefore be to move (at least) the arrays containing the child nodes to off-heap memory as shown in Figure 3.1, so that we might have more control over how they are allocated and cached by the CPU and/or OS. We are interested to try

off-heap memory method, but it doesn't work. For the completeness of the thesis, we introduce our ideas in the following:

3.1.1 Unsafe in Java

By default, Java doesn't allow accessing the off-heap memory. This serves memory safety (by preventing eg leaking object references), but sacrifices flexibility. Fortunately, Java also provides the `Sun.misc.Unsafe` class, which allows exerting more control over the internals of the JVM's object management.

To use `Sun.misc.Unsafe`, We first create an static Object of Unsafe and use it to instantiate a singleton "theUnsafe" by reflection in Java, which already breaks the singleton class pattern. After that we can use the `unsafe` to view the virtual memory address of an object or allocate a new memory block with a given size. Because the off-heap memory is not managed by the JVM garbage collection, the memory block which we allocated by ourselves has to be released also manually by calling `freeMemory()`.

```
1        Field field = Unsafe.class.getDeclaredField("theUnsafe");
2        field.setAccessible(true);
3        unsafe = (Unsafe)field.get(null);
```

Listing 3.1: Instantiate unsafe

With `Unsafe` we do have more control over memory allocation to store all the nodes of the Hoeffding tree together or closer to each other and it reduces the overhead of the garbage collector. However, the time for an allocation is significantly longer than allocate an object into on-heap memory.

3.1.2 Direct Byte Buffers

Another "safer" option provided by Java is the usage of "Direct Byte Buffers". It is introduced with the Java New IO (NIO) from version Java 1.4 [7]. These can be thought of as contiguous regions of memory, which are allocated manually and thus also have to be freed manually. However, they are also guaranteed to be off-heap memory. This means that by placing the nodes in such Byte Buffers, we theoretically have more control over how they are cached. This also means that Java's standard classes, such as ArrayLists, can't be used, as they rely on the JVM's automatic memory management.

To use the direct byte buffer is relatively simple. We declare a size for the buffer, and then use the `put()` and `get()` methods to manage the elements in this allocated buffer.

All elements that are stored in direct byte buffers have to be retrieved through a c-like pointer `ByteBuffer`. So it is also important to manage the position of ByteBuffer manually, which would be not so elegant for a situation where we have so many nodes to store and retrieve. The cost of maintaining the byte buffer is one of the biggest disadvantages of direct byte buffers.

While this approach would give us more control over the low-level details of memory management (and thus caching behaviour), it would also mean giving up the optimizations that the native Java data structures would provide us with: as most standard data structures are (partially) implemented using native code, they have a natural advantage over any implementation we might be able to come up with in java code. Apart from that, they also enjoy additional, data-structure-specific optimization given to them by the JVM.

In fact, initial tests conducted up-front have shown that simply storing and retrieving a list of integers from a byte buffer is twice as slow as just using a native java array as the results shown in Figure 3.2:

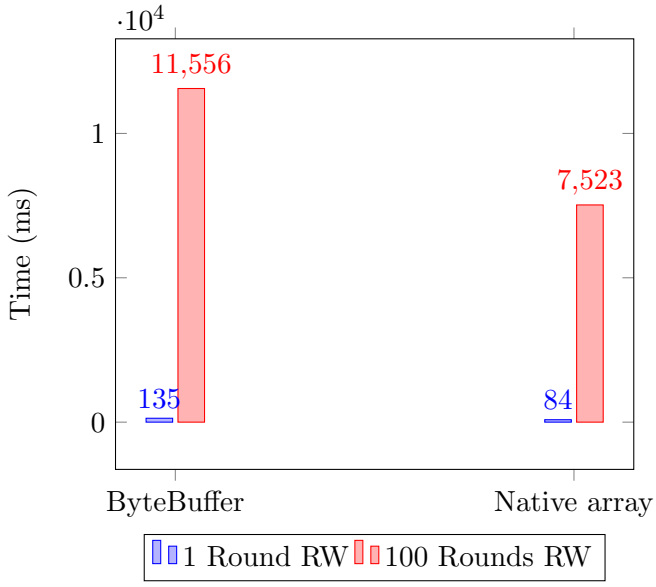

Figure 3.2: Results of random read and write speed between direct buffer and native array

Therefore, this option didn't seem like it might be able to provide any benefit at all, at least not as long as we implement the memory-management code in pure Java, and it also brings all risks of `Unsafe`. It means that we have to accept that controlling the memory allocation in Java does not gain any benefits in this scenario.

3.2 On-Heap Memory

The other class of approaches is based on accepting the Java-imposed memory-management limitations and focusing on optimizing the data structure for caching. While this involves working exclusively with on-heap memory, it also means that we can make use of the classes Java already provides to store the nodes of the Hoeffding tree.

So the problem remaining now is how to alter the data structure of the Hoeffding tree. Similarly, we want to have all the nodes of a decision in a compact and easily accessible structure, such as an array or a map, as adopted in [4].

3.2.1 Multilayered Shared Array

One approach would be to take the currently implemented structure and use a big, shared array as a "substrate" in which to save it; similar to how we implement a binary tree using an array as the underlying storage "medium".

Figure 3.3 shows how are the decision tree combined with the shared array and the node structure. First of all we take each "Vector" of each node, in which the children of that node are saved. Then, the index-value pairs of each vector in a binary search tree are stored. The sorting criteria for determining parent-child nodes is the index of that index-value pair. This is because the vector might be sparsely populated (i.e. "holes" / empty spaces between elements). Each time an element is added to the vector, a new index-value pair is created and inserted into the binary search tree. Considering there is memory management of MOA which sets frequently some of nodes to inactive, we always allocate 2 Blocks to leave the blank space free for those nodes even if we don't need them in the next learning process.

This approach is called "multilayered" because that binary search tree is then saved in an array(list), together with the Binary Search Trees (BSTs) of the Vectors of the other nodes. Saving all of them in a single, shared array reduces memory fragmentation and increases the chance that we won't fill half cache lines.

However, this approach also imposes a great deal of additional implementation complexity, leading to more code (and thus way worse I-cache behavior) and less performance (because this indirection also necessitates a lot of nested method calls).

The next argument against this approach would be the enormous overhead of saving everything in multiple (virtual) data structures at once. As can be seen in the graphic, we have to save the attributes of not only the vector itself, but also those of the (virtual)

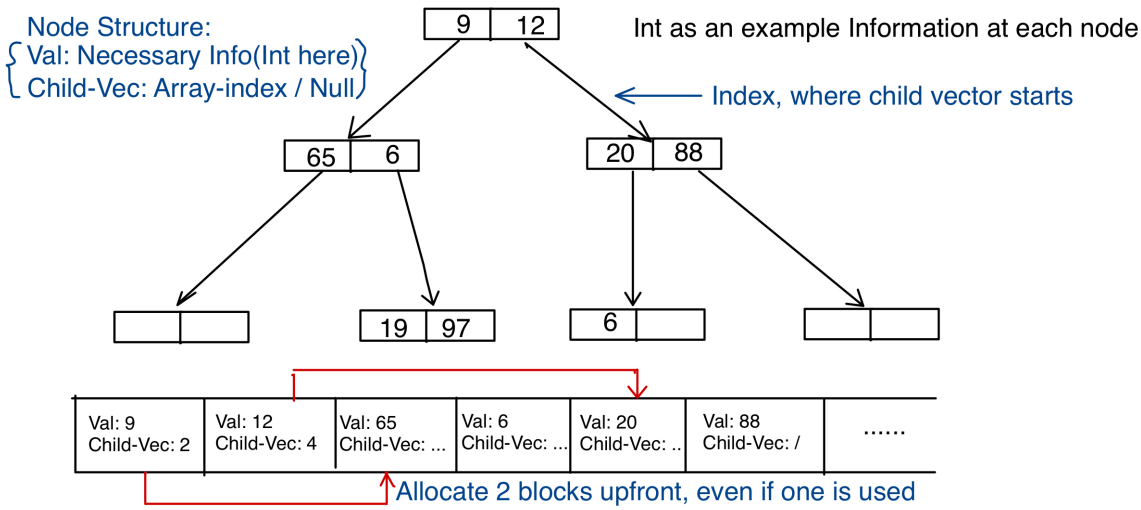

Figure 3.3: Multilayered node structure

BST in which we save it, and then also the attributes necessary for locating the elements in the big, shared array (In example, "where is the root of each BST inside this shared array?").

3.2.2 Direct Static Array

The last approach would be to remove the vectors altogether and directly save the Hoeffding tree itself (not just the vectors containing the children) in a big, shared array (array list). It is more like a simplification of last approach, without introducing new structure and fewer indices that need to be maintained.

This approach obviously has the least overhead of all the alternatives considered, not only in code size (and thus I-cache impact), but also in additional data that might have to be stored. Further, it's also relatively memory-efficient, as it still relies on the JVM's optimized memory layout routines. Furthermore, these routines are implemented in native code, leading to overall faster code execution. There's no complicating data structure introduced and all nodes are stored in a flatter way. And the codes can be also more readable than the other approaches with unrecommended memory operations in Java, which sadly doesn't gain us more benefits like in C language. And we can not have a precise control of the data locality because the nodes physically do not have to be stored contiguously in memory according the JVM's specification [1], which can lead to a not so surprising result when we implement the same method from [4].

To convert a binary decision tree into a array list of nodes, each node carries information about its own position, its parent's position and the positions of its children. With

these information each time a new node is created, it will instantly be added to the tail of this array list and the parent index can be easily traced. Because the node list is static, it is cleared every time in the constructor of `HoeffdingTree`.

Once we store all nodes in an array list, we can do the second part of this thesis: Sort the nodes with their path probabilities. The probabilities can be calculated recursively and we will rearrange the nodes on paths that have higher probabilities closer to each other and sorted in decending order.

4 Implementation

Essentially there are two steps for the implementation. We first implement the shared array to store all nodes together as we introduce in Section 3.2.2. It requires a way of conversion so that we can maintain the tree structure with the shared array. So we modify all nodes in `HoeffdingTree` to store the key information for the tree structure as explained in the following. The second step of the implementation is applying the method proposed in [4] in `HoeffdingTree`. After all the nodes of Hoeffding tree are stored in an array, we have a pure native tree which is suitable for the probability sorting method. Here after we introduce how is this exactly implemented and the changes we make to fit the implementation of Hoeffding tree.

4.1 Node Modification for Shared Array

As we consider specifically using binary splits, we had the opportunity to reduce the `SplitNode` from using vectors to store the children for each split node, to using only two int's per node, for storing the index of left and right child in the big shared array. We mentioned in Section 2.1.4 that there are several different types of node. The basic type `Node` should be the generic data type for the static node array, so that all different nodes can be stored in this array without losing any information.

Node splitting take place all inside of `SplitNode`, so we change the way of creating child nodes in split node. The first thing is to replace the `AutoExpandVector` that is used for storing children, with two simple attributes with type of int indicating the position of children. So the split node itself does not carry its children objects anymore but just an index.

```
        class Node extends AbstractMOAObject {
            int parentIndex = -1;
            int indexOfThisNode = -1;
            ...
        }

```

```
        class SplitNode extends Node {
            InstanceConditionalTest splitTest;
            int indexOfChild1;
            int indexOfChild2;
            ...
        }
```

Listing 4.1: Split node attributes

The `setChild()` and `getChild()` methods are also replaced by updating index values. Since we only consider binary splits here, we have only two possible child positions: Local index 0 represents the left child of this split node, and local index 1 for the right child. Due to the designed two-way index, we update the index information of both parent and children. If a child is being placed at an index greater than 1, it means that a non-binary split takes place and we can not accept it. Once we know the index of the node we need, it is quiet easy to get this node by just calling `HoeffdingTree.nodeList.get(indexOfNodeToGet)`. While doing this, it is very important to keep in mind that we focus only on binary splits here, so that we can have the potential benefits of only having a maximum of two children for each split node.

We add one int attribute `indexOfThisNode` to every node to simplify the tree traversal, as well as the implementation of using one big, shared array for all of the Hoeffding tree. This is not necessary because we can use the `indexOf()` method provided by Java which works the same as this attribute does. The goal is to reduce the unnecessary searching in node list. If we have the sufficient information already inside the node, it is faster than looking into the array list with all nodes again and finding the wanted node. So the nodes are now all stored in an array list as the Figure 4.1 shows and through the indices the tree remains functional.

For the structural integrity of the Hoeffding tree, we introduce these indices and they help rebuilding the tree from a flat array. They have to be always up-to-date with new tree changes such as adding new nodes into the tree or rearranging all nodes in the array. To update the indices, there are always three parts we consider: the node which is currently being updated, the parent of the node and the children of the node if it is a split node. This could also be a problem because we retrieve more nodes from the array for just updating one node. And as mentioned Hoeffding tree grows during the whole process which means we have always nodes to update and it brings already potential overheads. But we do not come up with a better solution for this problem and it is also a straight-forward way to implement.

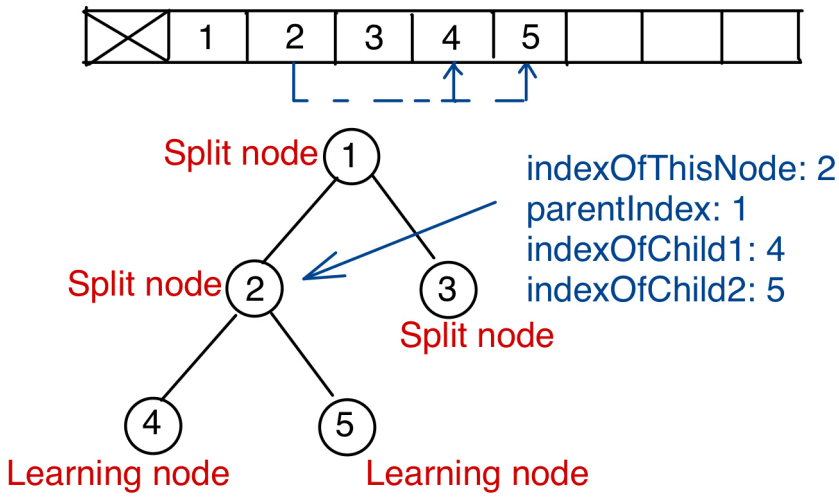

Figure 4.1: Node list structure

To simplify development and testing, we also added methods for printing the tree recursively. The methods for managing the tree layout and tree traversal are also modified and added. This part takes a lot more time to implement, because there is no provided written methods in MOA to view the trained decision tree. We start from root and go to the subtree of current node by recursion. Then we can show the information of this node's index in the shared array list, index of its parent and indices of its children if it is a split node. There are also many other helper methods implemented. In example `numOrphanNodes()` returning how many nodes currently are not reachable from the Hoeffding tree, which means they are wrongly located in the array or some abandoned nodes are still not removed from the array.

One thing needs to be explained is the first element of the shared array list. As Figure 4.1 shows, the first node element stored in the array list is not being used. The reason for that is tree root splitting. At the beginning of a task, there is only one node, the tree root and it acts also as a learning node. But if the tree root has sufficient statistics for a split at a time, it creates a new tree root for the Hoeffding tree and start splitting from the new root. This means that the first node is not the working tree root after splits, which can be tested by using the customized tree printing method.

In order to check if the tree remains complete and correct, we can call the method `numOrphanNodes()` and the tree printing method. We modified the `describeNode()` for each type of nodes, to see which node exactly is used. For the next step, implementing the probability sorting, "adding nodes" is refactored as well, so that we could more easily alter the condition (and method) for sorting, without having to change unrelated parts of the code.

4.2 Probability Sorting

After all these changes, the tree is now fully converted into a flat array and the ways of get/set are also refactored specifically for binary split. The next thing is to implement the method proposed by Buschjäger et al.: The nodes are rearranged in the order of their path probabilities, to further reduce the cache misses.

MOA provides observed class distribution, which is a array of two int numbers indicating how many instances are filtered to the left child and the right child. The number of instances classified in this node can be calculated by adding the two values. When the probability for both left and right child of a split node is found, we do the calculation recursively and multiply the probability of parent node with the current node, to get the path probability. In the end the probabilities that are stored on leaf nodes represent the probabilities of the whole path. Therefore, leaf nodes should be found and sorted in the descending order of their probabilities as path probabilities. `getAllLeaves()` returns all leaves in an array list where the nodes are also already sorted by using a customized comparator as Listing 4.2 shows. To be able to sort all the leaf nodes, the program requires the natural ordering of `Node`. And the natural ordering for the nodes is obviously the probability of each node. We use the `sort()` method provided in Java and overwrite its `compare()` method by comparing the attribute `probabilityOfThisNode`. The positive return value means that the second node has a greater probability and since we want a descending order at the end, it should be switched to the front and vise versa.

```
        Collections.sort(leafList, new Comparator<Node>() {
            @Override
            public int compare(Node o1, Node o2) {
                double diff = o1.probabilityOfThisNode - o2.
                    probabilityOfThisNode;
                if (diff < 0) {
                    return 1;
                } else if (diff > 0) {
                    return -1;
                }
                return 0;
            }
        });
```

Listing 4.2: Sorting method

In [4] the implementation uses a heap to store the candidates that are not selected before, and after a complete path is sorted, the node in heap with maximum path probability can be selected as the next node to be stored. This is definitely an elegant way to

solve the problem, but not quiet suitable for the implementation we have here, because the path probabilities the sorting needs are all saved on leaf nodes and the other nodes contains only the node probabilities. Of course we can update the probability attributes of all nodes with a for loop, but it is not necessary since we know which of all nodes are leaf nodes with the true path probabilities that are needed.

As Algorithm 4.1 shows, the search begins from leaves to root. For the algorithm we need four new attributes which memorizes the new position of current node, the new position of parent and the new positions of children if current node is a split node. In a loop through all leaf nodes in the descending order of probabilities, as long as the node is not updated with new indices (pre-moved), the new positions for itself, parent and children are set, as well as a Boolean value `isMoved` which means is virtually moved but actually just updated without real move. This is for situations like if we have a node whose parent has been moved to the new array, it will cause inconsistency when we update this node's parent in the old array. And the costs to check whether its parent and children are moved is also higher than this implementation. Moving to the new array takes place after all nodes are updated and the tree root is set to be the last node to update. After that, the node list of Hoeffding tree is replaced by the new array and waits for the next sampling.

The probability sorting method is called every time when the MOA starts sampling for the results. This setting allows us to configure in the GUI of MOA easily, how freqeuntly the array list of nodes should be sorted, by just entering parameter values.

```
Input: Array of all nodes in Hoeffding tree nodeList
  newArr : node[ ]
  leafList = getAllLeaves(nodeList)
  for (leaf : leafList) do
    currentNode = leaf
    while (!currentNode.isMoved and currentNode! = root) do
      update currentNode's index information about itself, parent and children
      currentNode = nodeList[currentNode.parentIndex]
    end while
  end for
  update tree root's index information
  for (node : nodeList) do
    newArr[node.newIndex] = node
    newArr[node.newIndex].isMoved = false
  end for
  nodeList = newArr
```

Algorithm 4.1: Algorithm of node rearrangement

5 Evaluation

To evaluate the performance of our implementations, we compare values directly from the graphical user interface of MOA and use the VTune profiler to see the hardware events and more statistics in detail. We have two different platforms for the testing: A MacBook Pro running Mac OS with Intel i7-9750H and 16GB of RAM, and a PC running Ubuntu 19.10 with Intel E5-2697 and 1GB of RAM as the remote server of VTune. All approaches are based on MOA release version 2019.05.0 and VTune version 2020 update1. Both test systems are using Java 13.0.1.

We use three different versions of the implementation for the evaluation. The first one is the original implementation of MOA without any changes(hereinafter called the Original). And in the second version we store all the nodes in a shared array list so that the Hoeffding tree is converted into a native tree. The third version is based on the second one. We implement the probability sorting method from [4] and rearrange all nodes in the array list.

5.1 Task Results From GUI

If the implementation is all correct and without mistakes, the results should also be exactly the same as the original MOA implementation except the time elapsed, because the mechanical changes should not have any effects to the learning accuracy and the information of each node. The test is set to a data stream with 1,000,000 instances and sample frequency at 10000. Each of the these three versions are tested with 10 repeated tasks: First the original MOA implementation without any changes shows that the run time averages 4.179 seconds and it has an accuracy of 95.5% as Figure 5.1 shows. We switch the `moa.jar` to the jar that is compiled with our changes: If all nodes are stored in one shared array list, we notice that the run time is very slightly improved a little bit with an average of 4.11 seconds. With the probability sorting, we can see that there is a noticeable improvement in speed in average 4.097 seconds and some of the tasks are completed in 4 seconds. Accuracy remains always the same as we expected in all three versions which means the implementation is correctly working.

To see the run time difference the number of instances is set to 10,000,000 as Figure 5.2 shows. Still the changes doesn't differ from each other too much. All three versions have a run time at around 125 seconds. Since we cannot visualize the difference from the run time, it has to be analyzed deeper to the hardware level, which is why the VTune profiler is used in this thesis:

5.2 VTune Profiler

VTune Profiler (formerly VTune Amplifier) is a software from Intel which provides many powerful features that support the developers in analyzing the performance of their programs. It was commercial before and is now free to use. We use VTune to analyze the

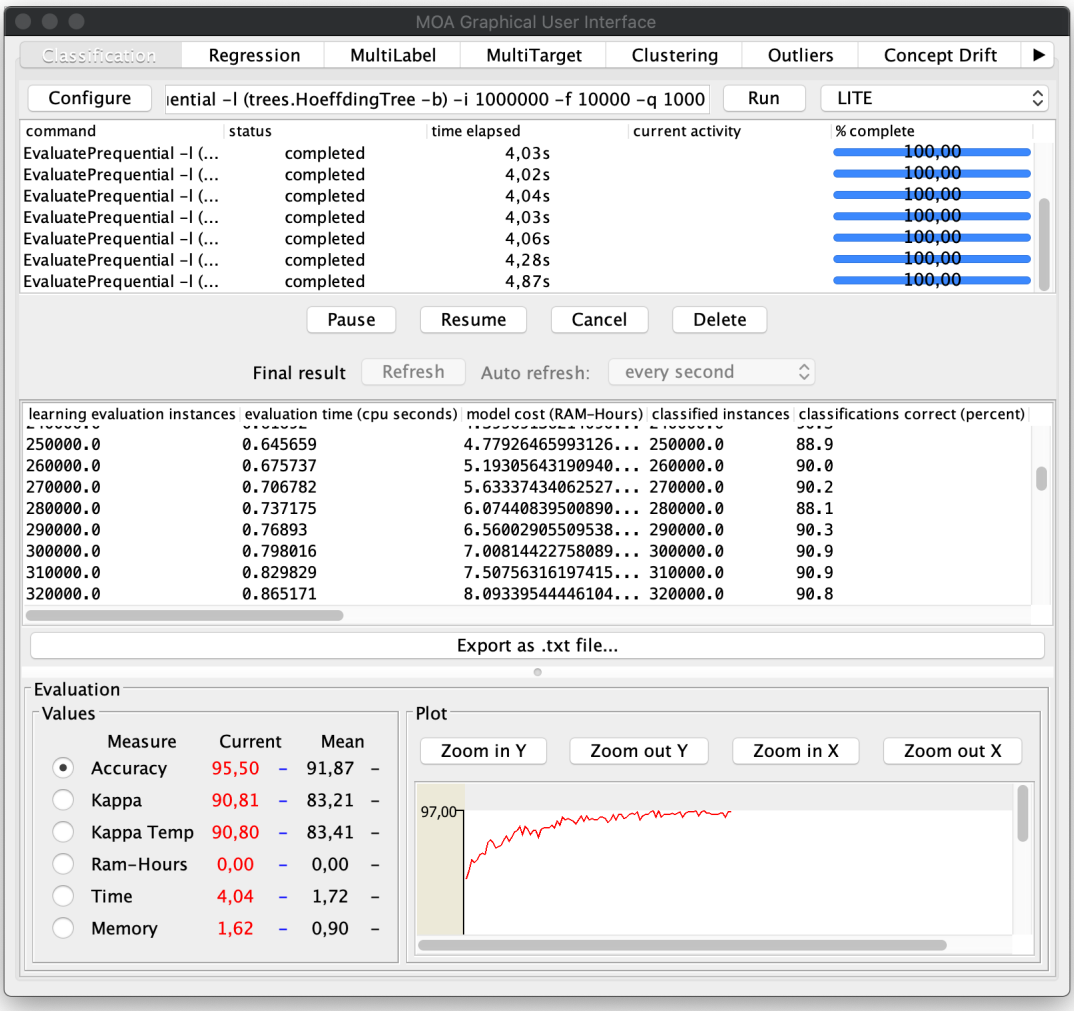

Figure 5.1: Results in GUI

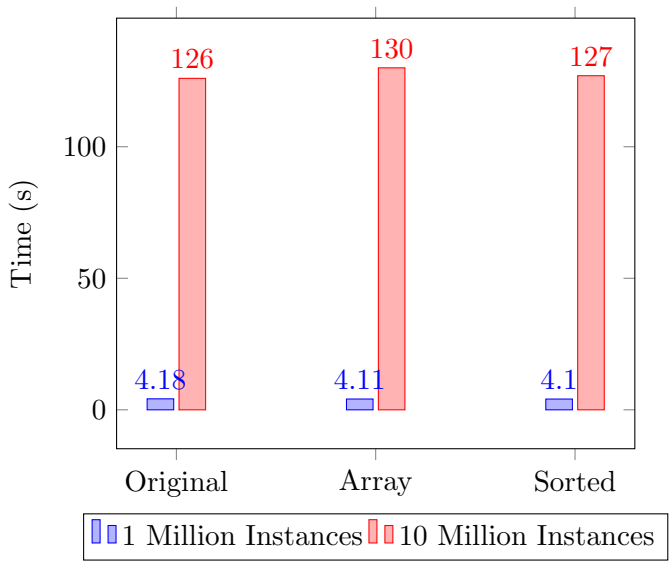

Figure 5.2: Results of run-time for three versions

data cache-misses of MOA before and after our implementation, so that we can see if the changes provide advantages regarding the cache behavior.

5.2.1 Introduction of VTune

We can choose from plenty of different options of analysis directions in example Hotspots, Micro-architecture or parallelism. And the hardware-level analysis can help us find out how our java programs are actually performing since we do not have the direct access to hardware with java. Through the bottom-up function tables we have the possibilities to check the performance of each specific function (CPU Time, Instructions retired, Micro-architecture usage and more). The most important data for this thesis, Hardware events about cache misses can be retrieved with the help of micro-architecture exploration.

There is a critical bug also found in VTune on MacOS. For the VTune setup we configure first the analysis on the local host machine and then connect to the remote Linux server. After the task is complete, the results will be downloaded back to the local client. But on MacOS, the VTune can not load the results from remote, so we use another Windows PC as our local client and ssh connect to the remote server. The analysis direction is set to micro-architecture exploration as mentioned before. After the collection and processing of results, we can check every hardware event in detail.

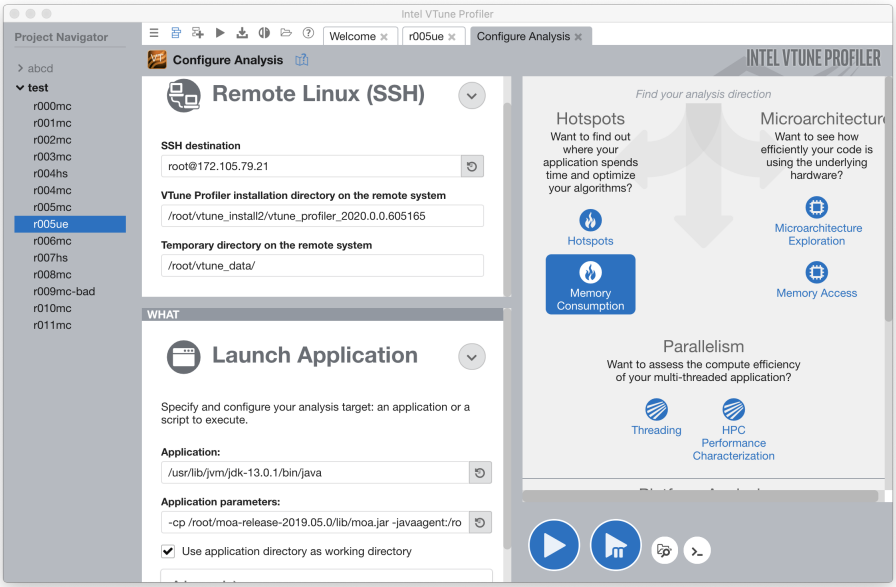

Figure 5.3: VTune GUI

5.2.2 Results in VTune

Same as the testing in MOA GUI, we test the three different versions mentioned at the beginning of Section 5, and with two different number of instances. Application is the Java program and the different jar packages are given as an application parameter. The other settings remain the same as above in GUI.

First value to compare is the average CPU utilization. It shows average CPU utilization by computations of the application. Spin and Overhead time are not counted. Ideal average CPU utilization is equal to the number of logical CPU cores [2]. The test server's CPU has only one core so the ideal average CPU utilization value is 100%. When we use the original MOA implementation, this value is 79.2%. VTune indicates that it is already a low metric value, and the possible reasons can be load imbalance, run-time overhead, contended synchronization, or thread/process under-utilization. And when the nodes are stored in a shared array list, the average CPU utilization value comes to 80.7%, which doesn't differ too much. If the nodes are sorted in the order of their path probabilities, this values drops to 78.8%.

If we change the summary window to the "Hardware events", all monitored hardware events are listed here with values in detail. The following values more interesting to us:

- MEM LOAD UOPS RETIRED.L1 HIT and MEM LOAD UOPS RETIRED.L2 HIT show how many L1 and L2 cache hits take place

- MEM LOAD UOPS RETIRED.L1 MISS shows how many L1 cache misses take place

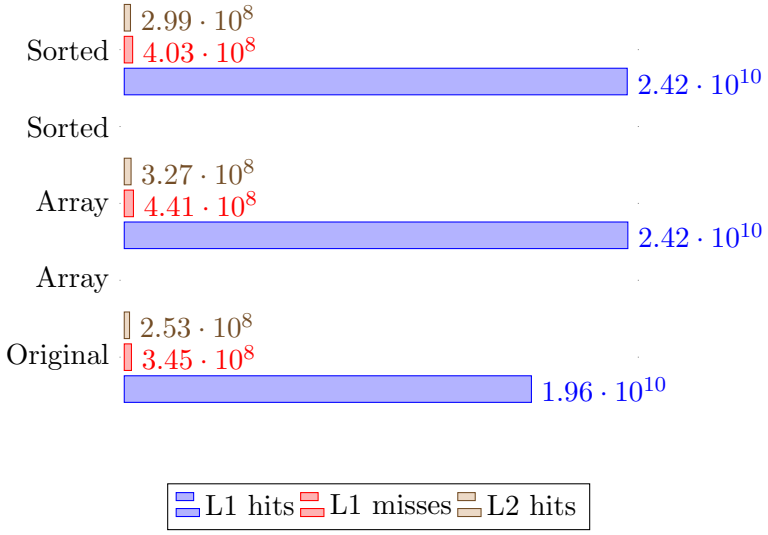

Figure 5.4: Results of L1 and L2 cache events

As we can see in Figure 5.4, the results are not good as expected. We do have 23.68% more L1 cache hits with the proposed implementation, but meanwhile the number of L1 cache misses are also higher than the original. With only node array the L1 cache misses are 27.9% higher than before, and if the path probability sorting method from [4] is used, the number of L1 cache misses can be reduced again. For the L2 cache hits, when the nodes are stored directly in an array list without sorting, the L2 cache hits increases already 29.33% than the original implementation, but sadly once we add the path probability sorting method from [4], the number drops again 8.12%.

There is another interesting metric that is dramatically affected by the implementation. Through micro-architecture exploration we notice that the Front-End bound changes significantly with these three different implementation. This metrics indicates such as instruction-cache misses, ITLB misses or fetch stalls after a branch misprediction. When stalls take place due to instruction-cache misses, it would be categorized as Front-End Bound. With the original, the Front-End bound is 13.54%. Once we store all nodes into a shared array list, this value is strongly affected and increased to 25.7%, which is already unacceptably high. And if we rearrange the node in this array list with their path-probabilities, it drops back to 11.7%, which is 13.78% better than the original.

One possible reason of the results can be the additional overheads introduced by the indices. To confirm this assumption we can use another feature of VTune which is the bottom-up analysis. We can choose from different program units for the grouping. If the shared array does introduced more overheads, so the changes should be mostly observable

in split node. Between the original version and the version with shared array implemented, we observe a significant longer CPU Time at `getChild()` function in Hoeffding tree, where we replace all small arrays with the shared array. The function has a CPU Time of 160 milliseconds where the time for original version is 35 milliseconds. This is an evidence that shows the overheads of retrieving nodes for Hoeffding tree are greater due to the implementation of the shared array. We access the nodes more frequently than the original to update the indices as the tree grows and the sorting method take place.

6 Conclusion

In this thesis, we rearrange the nodes in Hoeffding into one unique shared array. To do so we compare off-heap memory and on-heap memory, and choose the on-heap memory to store all the information at the end. We used a static array list and share it across all node instead of maintaining plenty of small array lists for each node. The nodes are modified with a set of indices which helps indexing the tree structure. After that we implement the method in [4] for the Hoeffding and sort the nodes in this array in the order of different path probabilities.

As the evaluation result shows, the optimization does not improve the performance of Hoeffding tree too much. This can have multiple possible reasons: First the implementation uses a lot of attributes i.e. the index information for each nodes and they always requires updates. This brings again much overheads. The updating process is executed every time that the sampling of MOA takes place, and it stalls the whole task. Another reason can be the Java Virtual Machine. Even if we implement that the nodes should be together in this array and close to each other, we can not know if they are really physically stored contiguously in memory due to JVM. JVM does not guarantee that an array has always the continuous physical memory address. So we may still suffer from the poor data locality. It can also be possible that the unsatisfying result is just because of the code quality from the implemented changes. We didn't manage to avoid some for loops when going through all the nodes. This could cost much time to load all nodes multiple times into cache.

But one thing for certain is that, trying to achieve the low level data allocation manipulating in Java is a painful and fruitless way. We do not gain much benefits from the method which already works in Python and C, but introduce more unnecessary overheads to the codes. So the further improvements should be focused on the algorithmic view instead of mechanical view in codes. And we can also try to implement the Hoeffding tree in other programming languages that are natively compiled such as C and Rust.

Appendix

List of Figures

Algorithmenverzeichnis

List of Source Codes

Bibliography

[1] *JVM Memory Specification*, 2020.

[2] *VTune CPU Utilization*, 2020.

[3] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. MOA: massive online analysis. *J. Mach. Learn. Res.*, 11:1601–1604, 2010.

[4] S. Buschjäger, K.-H. Chen, J.-J. Chen, and K. Morik. Realization of random forest for real-time evaluation through tree framing. *2018 IEEE International Conference on Data Mining (ICDM)*, 2018.

[5] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD 00, page 7180, New York, NY, USA, 2000. Association for Computing Machinery.

[6] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):1826, June 2005.

[7] R. Hitchens. *Java Nio*. OReilly Associates, Inc., USA, 2002.

[8] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[9] G. Holmes, R. Kirkby, and B. Pfahringer. Stress-testing hoeffding trees. In A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *Knowledge Discovery in Databases: PKDD 2005*, pages 495–502, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.