Technical University of Crete

School of Electroinc and Computer Engineering

Diploma Thesis

# FPGA-based Data Mining from Streams

Pavlos Giakoumakis

Committee

Professor Apostolos Dollas (supervisor)
Associate Professor Ioannis Papaefstathiou
Professor Minos Garofalakis

October 15, 2014

**Abstract:**

*Data Stream Mining is a relatively new field of Data Mining and Hoeffding trees is a very popular classification algorithm on streaming data (instances). The basic Hoeffding tree algorithm, with some extensions, was originally presented as VFDT (Very Fast Decision Trees) by Domingos and Hulten.*

*We studied Hoeffding trees and the most of its available extensions. There is a large number of customizations and that makes it hard to study them all and select a "good case" to be implemented and optimized in custom hardware. Nevertheless, we found that the main behavior of the algorithm remains similar. Thus, we selected a configuration very close to the initial VFDT algorithm.*

*Hoeffding Trees are highly parallelizable and employ operations on the leaves that seem to fit well on FPGAs. We designed a simple processing unit (the Attribute Processor - AP) that could be used by very small embedded systems, and is capable of updating a leaf and computing the best (binary) split suggestions, using negligible resources.*

*By using a group of APs we designed the Attribute Processing Array (APA). It can easily take full advantage of the bandwidth of a single memory controller port (in our case the Convey Coprocessor MC port) with very low resources.*

*To exploit the processing power of Convey's HC-2$^{ex}$ we had to make a clever partitioning between hardware and software by selecting a custom instruction set and designing a collection of "utility" hardware modules, that all together compose an interesting architecture.*

***Keywords:* Data Stream Mining, Machine Learning, Decision Trees, Hoeffding Trees, VFDT, Hardware Accelerators, Field Programmable Gate Array, Convey Computer**

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With the emergence and evolution of computers we could solve increasingly larger problems, but collecting digital information to support a model was not easier than processing them. However, today in many domains we have an almost infinite amount of digital information available that is impossible to process with classic algorithms. Data Stream Mining is a relatively new field that aims to address the processing of data streams with new intelligent algorithms. Hoeffding trees is a data stream classification algorithm based decision trees that grow dynamically.

## 1.1 The Classification Problem

Classification is the process of assigning unseen instances to categories, based upon an existing model. To generate the needed model a training mechanism must process a set of instances with known labels. Moreover, we are trying to learn the underlying concepts under a set of instances, and generate a model $f$, to accurately predict the class $y = f(\mathbf{x})$ of any instance with attribute values $\mathbf{x}$.

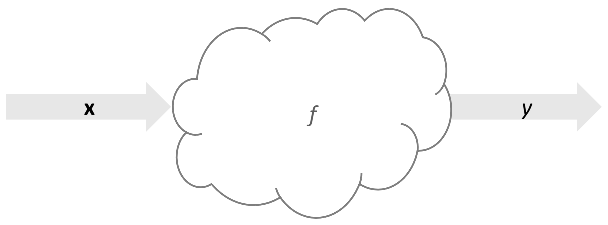

Figure 1.1: Using a model $f$ to classify an instance with attribute values $\mathbf{x}$.

## 1.2 Decision Trees

A decision tree is structure that can be used to ease decisions. An example decision tree is shown in Figure 1.2. Each node of the tree contains a test on an

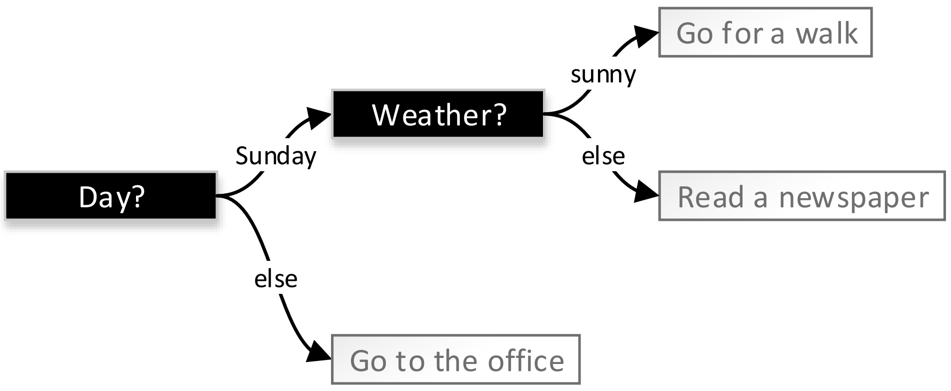

Figure 1.2: A very simple decision tree.

attribute and one branch for each possible outcome of the test. Leaves contain valuable information that describe the selected decision. Such information may be a class label or the sufficient statistics to make a class prediction. Decision tree learners induce decision trees to address classification problems. A newly arrived instance will begin from the root and will follow branches to lower levels until it reaches a leaf. The adequate leaf operations will be triggered either to assign a class label to the instance for test instances, either to update the leaf's structures for training instances.

## 1.3 Market Analysis

The market for hardware-based implementations of Hoeffding trees is divided into two segments, the embedded systems and the high performance computing (Figure 1.3). Nevertheless, an additional segment of hybrid systems may exist. Many embedded applications may need a low cost, power efficient classification system that is able to process real time data streams. Various combinations of cost, power consumption and performance are needed, depending on the application field. On the other hand, in high performance computing, the goal is to provide a system capable of swiftly processing enormous amounts of data. Power consumption stills a very significant factor. Furthermore, for long running systems, lowering the power consumption will considerably reduce maintenance cost.

### 1.3.1 Embedded Systems

We could easily distinguish a plethora of applications that would benefit from an embedded system that is capable of coping data streams. These systems must comply with some requirements such as:

- System cost.
- Real-time performance requirements.

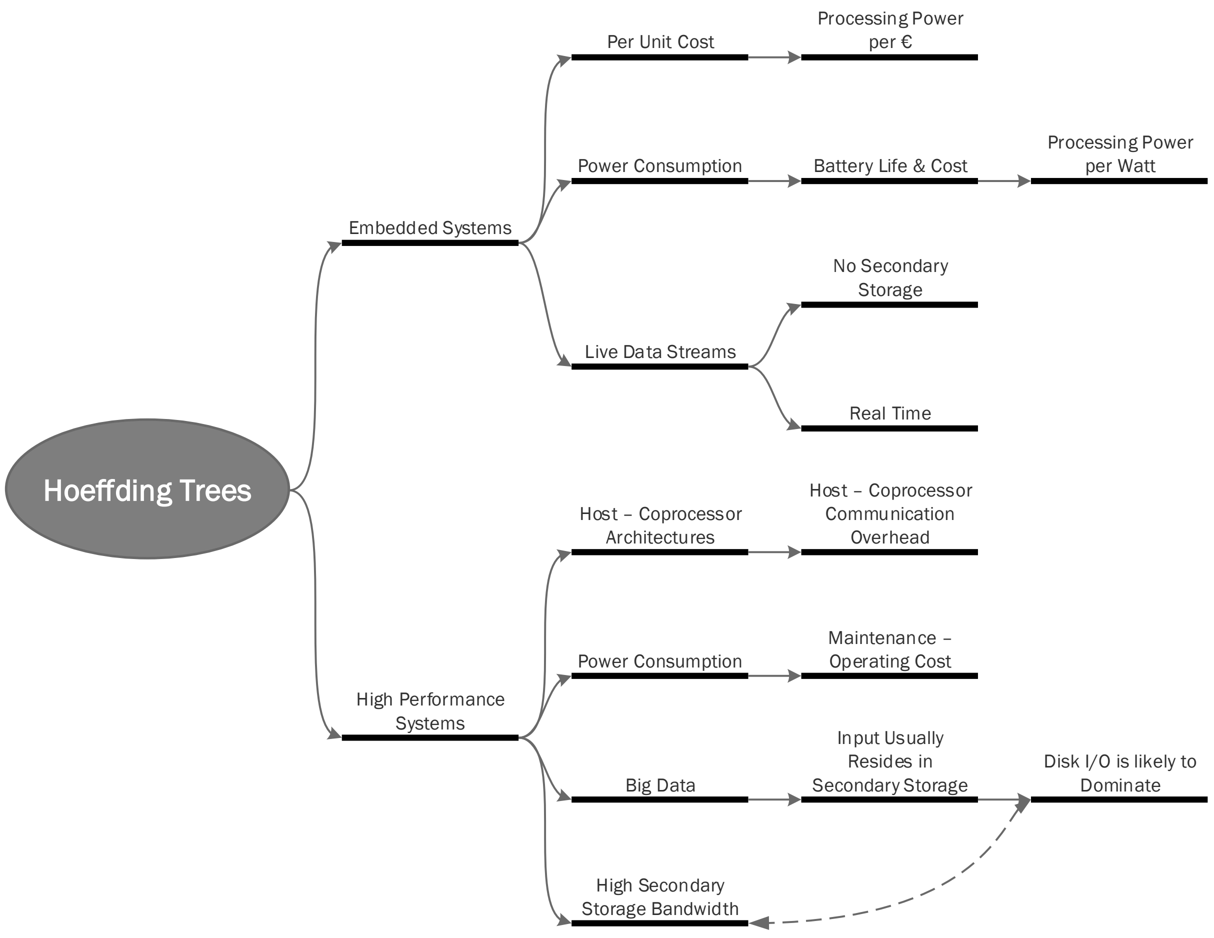
Hoeffding Trees
Embedded Systems
Per Unit Cost
Processing Power per €
Power Consumption
Battery Life & Cost
Processing Power per Watt
Live Data Streams
No Secondary Storage
Real Time
High Performance Systems
Host – Coprocessor Architectures
Host – Coprocessor Communication Overhead
Power Consumption
Maintenance – Operating Cost
Big Data
Input Usually Resides in Secondary Storage
Disk I/O is likely to Dominate
High Secondary Storage Bandwidth

Figure 1.3: Market segmentation for Hoeffding Trees classification algorithm.

- Power consumption specifications.
- Time to market.

The aforementioned requirements, all together along with other, make a potential reconfigurable hardware solution ideal for such applications. Moreover, micro-controllers and low power CPUs will fail to provide the required performance, while high-end CPUs will fail to provide the required performance per watt relation.

### 1.3.2 High Performance Computing

Many applications may favor from a high performance data stream classification hardware implementation. Generally, we could recognize two notable categories of applications that will take advantage of it:

- Very fast data streams.
- Very large databases.

## 1.4 Motivation

Our motivations included:

1. Examine a data stream mining algorithm and how it could be implemented in an FPGA using HDL languages.

2. Implement a data stream mining algorithm in a Convey computer, evaluate HC-$2^{ex}$ as a platform to implement data stream mining algorithm and rate the viability of such a system.

3. Examine the Dataflow paradigm. This paradigm results in well-performing FPGA systems. It is almost a synonym to what FPGAs can do.

4. Examine a simple version of Hoeffding tree algorithm and if it maps well on FPGAs. This algorithm has the benefit of having heavier refinements, or of being the basis for decision tree ensembles. A good result in the basic algorithm means that possibly a whole area of Hoeffding-tree-based algorithms will benefit from it.

## 1.5 Contribution

Contributions of this thesis include:

- To the best of our knowledge, this is the first implementation of a data stream mining algorithm in the Convey's platforms. Convey has an all-round versatile coprocessor memory subsystem that is able to provide nearly-constant streams of data even for non-sequential accesses. On the other side, our algorithm normally requests data sequentially. So we could use a system with a memory controller optimized only for only sequential accesses and possibly get bigger bandwidth. However, our implementation showed exceptional performance results and Convey HC-$2^{ex}$ seems as a viable solution for CPU-FPGA implementations of data stream mining algorithms.

- We analyzed the Hoeffding tree algorithm and noted the types of parallelism that exist on a simple Hoeffding tree version.

- We profiled the algorithm and found/confirmed that for large leaves the heaviest part on training the algorithm is the leaf processing.

- We analyzed the process to compute the best binary split suggestions on a nominal attribute of a Hoeffding tree and noted five refinements-optimizations. These optimizations result in simpler and smaller hardware. Two of them are able to shrink the cost to compute the split suggestions and at least one will be beneficial for software implementations too.

- We implemented a fully pipelined; stream based; Memory-controller-port oriented system to process leaves in hardware (nominal attributes only).
- We measured our single memory-controller-port FPGA based system and the results show that our system is able to obtain a speedup against the corresponding single threaded MOA procedure of up to more than 160x.
- We noted that with 64 memory controller ports available in the Convey platform, this speedups may result in a massive speedup or throughput increase of up to four orders of magnitude against the corresponding single-threaded MOA implementation.

## 1.6 Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 reviews the related literature. Chapter 3 provides a simplified description the Hoeffding tree algorithm, shows the profiling results and describes the available parallelism. Chapter 4 presents the architecture of our implementation. Chapter 5 verifies and evaluates the presented architecture. Finally, chapter 6 gives some ideas for future work.

# Chapter 2

# Literature Review

## 2.1 Introduction

In this chapter we review the most important literature related with our work. Excepting (1) in where its authors published the algorithm, most of the examined literature covers four fields. Each field tries to address very different issues:

- Work that refines the main algorithm.
- Refinements that try to address concept drift in data streams.
- Refinements that try to address noisy data.
- Hoeffding tree ensembles.

However this chapter does not follow this structure.

Furthermore, (2) and (3) are very good for someone to get introduced with, and understand the algorithm and the related field. They cover a vast area of related literature from evaluation of data stream mining algorithms and data stream generators, to Hoeffding trees, Hoeffding tree ensembles and concept drift.

## 2.2 Hoeffding Trees

Traditional learners that induce and use decision trees such as C4.5 (4), CART (5), SLIQ (6) and SPRINT (7) are batch algorithms and during the training phase they need to access each instance multiple times. Normally, when the training phase is complete, the induced tree will remain static for classification use. Hoeffding tree algorithm (1) is a decision tree algorithm for data stream classification with an incremental nature. It begins with a single node, the root, and employs the Hoeffding bound(8) to make split decisions. Furthermore, it will select the two best split suggestions (the two that maximize/minimize a function e.g. Information Gain or Gini index), and will uses the Hoeffding bound to decide if it is beneficial to apply the best split suggestion. In (1) Domingos et al.

also prove that the output of the algorithm is asymptotically nearly identical to that of a conventional learner. For more information about how the algorithm employs Hoeffding Bound please refer on Appendix A.

### 2.2.1 Selecting a Different Bound

In (9) Rutkowski et al. noted that Hoeffding inequality is incorrectly used in Hoeffding trees and proposed the use of McDiarmid's bound for both information gain and Gini index. In (10) Rutkowski et al. proposed another method based on the Gaussian Approximation, and in (11) they present a CART decision tree for mining data streams based on Gaussian approximation. However, our work can be applied regardless to which bound the algorithm will use.

### 2.2.2 VFDT

The VFDT (Very Fast Decision Tree learner) is a decision-tree learning system based on the Hoeffding tree algorithm. It was published together with the algorithm in (1). The refinements are listed below:

**Ties.** When the two attributes with the best score (Information Gain or Gini index) in a leaf have very small difference (smaller than a predefined threshold), then it is considered that there is a tie and VFDT splits on the best attribute.

**Score computation.** VFDT has a parameter to not check for split decisions for every instance, but to accumulate instances on leaves and check a leaf only when it observed enough (more than $n_{min}$) instances.

**Leaf deactivation.** The system deactivates the least promising leaves. During deactivation the main statistic structures are discarded and the corresponding memory is freed, so that it could be used for splitting other, more promising, leaves. A deactivated leaf may reactivated if it becomes more promising than currently active leaves.

**Poor Attributes.** When the difference between the score of an attribute and the best score observed from all attributes becomes greater than the Hoeffding bound, then it can be dropped from consideration and the corresponding memory could be freed.

**Initialization.** It can be initialized with a conventional RAM-based learner on a small subset of the data. This option can improve the "learning curve" and help the system to early achieve higher accuracy.

**Rescans.** VFDT can rescan previously observed instances. This is useful for slow streams.

Authors also measure and evaluate the VFDT system from various aspects.

### 2.2.3 Available Frameworks

#### VFML

VFML (Very Fast Machine Learning) (12) is a toolkit for mining high-speed data streams and very large data sets. It is written mainly in C and comes under a modified BSD license. Many algorithms are included in the same bundle, but someone could implement more by using the provided API. Unfortunately, VFML development stopped long time ago and even the compilation is tricky as it was written to be compiled with older tool versions. It implements a refined VFDT algorithm.

#### MOA

MOA (Massive Online Analysis) (13) is the most popular open source framework for data stream mining. It is written in Java and is closed related with WEKA (14) (it also use some code from WEKA). One of the best features supported by MOA is the capability to easily write your own plugins. Its Hoeffding tree implementation is based on VFDT but also has many other refinements. However, it lacks some of the features implemented in VFML or some of the refinements presented in VFDT. Many researchers write their own refinements as MOA plugins.

#### SAMOA Framework

Scalable Advanced Massive Online Analysis (SAMOA) (15) is a framework for distributed streaming machine learning. It includes a variety of distributed implementations of known algorithms, but also comes with a powerful API for developing new algorithms in distributed systems and offers a very high level of system abstraction. Researchers can utilize the provided functionality to concentrate to their algorithms and ignore lower level issues. SAMOA aims for scalability and, as soon as the implemented algorithm and the circumstances allow it, the same implementation could be able to exploit systems with vastly different capabilities. One very important and almost unique characteristic of SAMOA is that it addresses big data problems in all of three dimensions (volume, velocity, variety) (16). SAMOA employs a Streaming Processing Engine (SPE), such as S4 (17) or Storm (18) to execute an algorithm and provides one more possibility of integrating new SPEs to it.

## 2.3 Coarse-Grained Parallelism

In (19, 20, 21, 16), authors describe three types of extract coarsed-grained parallelism in decision tree classification algorithms: the synchronous, the partitioned and the hybrid methods. In the synchronous method all workers process the same node, but on a different part of the dataset. Dataset can be partitioned horizontally or vertically. In horizontal partitioning each worker will process a different instance, while on vertical partitioning each worker will process a different attribute. The partitioned method implies that each worker will work on a different part of the tree. Finally, a hybrid method between the synchronous and partitioned tree construction is possible.

## 2.4 Parallel Software Implementations

### 2.4.1 Vertical Hoeffding Tree

Vertical Hoeffding Tree (VHT) algorithm is presented in (16). VHT is a parallel version of Hoeffding tree. An implementation of this algorithm is included in SAMOA. It is optimized for problems with a massive amount of attributes. Furthermore, its architecture is comprised of four PI (Processing Item – an abstracted notion of a processor) types as shown in figure 2.1. The source PI

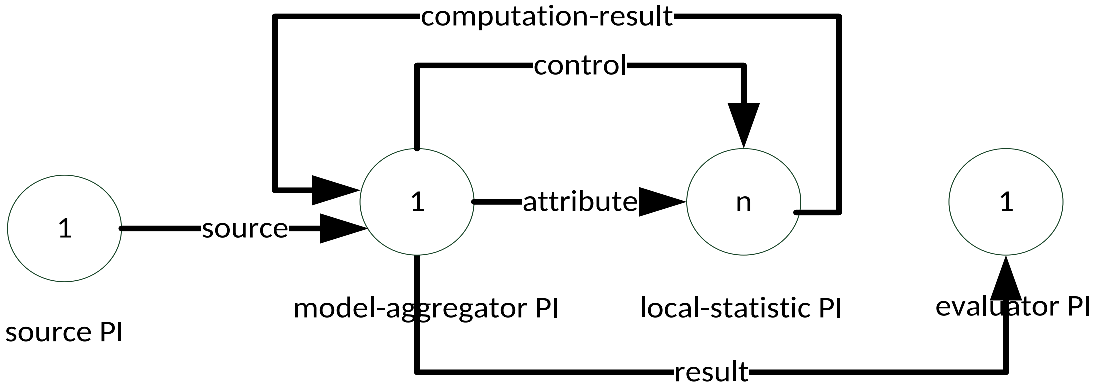

Figure 2.1: SAMOA - Vertical Hoeffding Tree (source: (16))

handles or generates the input data. The model-aggregator PI holds the tree model and coordinates the whole processing. The local-statistic PIs process leaves by either updating them or computing the best split suggestions. Each leaf is divided in multiple parts, where each part resides in a different local-statistic PI. To process a leaf each local-statistic PI processes its own part of the leaf. Evaluator PI is responsible for evaluating the classification results taken from model-aggregator PI. Five streams of content events (messages) are generated between the PIs. New instances from the source PI arrive to the model-aggregator PI via the source stream. Model-aggregator splits the instances and sends them to the local-statistic PIs through the attribute stream. It also sends control messages via the control stream to instruct local-statistic PIs to perform computations. Any results from a local-statistic PI arrive back to the model-aggregator PI via the computation-result stream. Finally, any classification results are sent to the evaluator PI through the result stream.

## 2.5 Related Hardware Implementations

### 2.5.1 HC-CART

Chrysos et al. in (21) present HC-CART, a CPU-FPGA system, based on Convey HC-1 (22), that implements the CART algorithm and shows impressive speedups of up to two orders of magnitude over single threaded solutions together with an exceptional power efficiency. To succeed this, HC-CART takes advandage of:

- The parallelism when evaluating different attributes
- The high memory bandwidth available in Convey's FPGAs.
- The high processing throughput offered by FPGAs.

Furthermore, HC-CART is the first combined software/hardware system where a multi-FPGA architecture is integrated with a popular software framework (rpart library of the R-project (23). From our aspect of view we also note that:

1. Computation of Gini-index is suitable for FPGA implementations and we could provide a Hoeffding tree solution based on Gini Index instead of Information Gain if needed.
2. The high level architecture used in HC-CART to combine multiple modules that process different attributes and the results are being compined by a tree structure of comparatores such as the solution proposed by Naranayan in (24). We could use a similar structure in our problem.

## 2.6 Other Work

Tong et al. in (25) develop an online traffic classifier, based upon the C4.5 algorithm, that classifies packet flows. They apply 10-fold cross validation to evaluate the accuracy of the classifier and they use Entropy-MDL discretization to handle continuous attributes. They also try different feature sets and different number of packets for training to find the best configuration for their problem. However, the trees are trained with WEKA, so they do not implement the training phase and they have very small models that fit completely in FPGA memory resources. They implemented two designs one for high performance, one for low cost. Their system has good clock frequencies but a small and field-restrictive design (the whole model resides in FPGA) and a Virtex 6 speed grade 2 device which is very fast.

Kestur et al. in (26) present SHARK (Streaming Hardware Accelerator with Run-time Reconfigurability). SHARK is a streaming model-framework for FPGA designs. Is layered hieratically in five levels of module categories:

- Utilities, which are basic arithmetic or logic functions.

- User-IPs, which are implement frame-level operations such as 2D convolution
- Cores, which are frame-level operations at a higher scale than user-IPs.
- Wrappers, which are basically composed of multiple cores but they also have to parse the configuration stream and route the data to the appropriate cores.
- Stream-pipe, which is the streaming pipeline composed of wrappers and custom logic. During configuration a stream pipe broadcasts the configuration data to the wrappers without parsing them.

Each module, except utility modules, interface with multiple input/output data streams and with one configuration stream. All streams use the LocalLink interface of Xilinx. SHARK is suitable for bottom-up design methodologies. Furthermore, authors also note that an accelerator can be reconfigured in three ways:

- Compile-time configuration.
- Run-time reconfiguration (they mean configuration through registers not partial reconfiguration).
- Partial reconfiguration.

Lastly, they implement an example system (the Saliency algorithm) where they have only one pipe-stream that is reconfigured multiple times to execute different parts of the algorithm. In this manner they save FPGA resources. The system measured to be nearly equivalent with an Nvidia GeForce 8800 GTX in throughput (5x compared with a CPU), but with 37x and 14x times higher performance-per-watt than the CPU and GPU implementations respectively.

# Chapter 3

# Understanding Hoeffding Trees

## 3.1 Introduction

In this chapter we examine Hoeffding trees from a different aspect. The aim is to gain an intuitive understanding about their behavior, and explore how a designer could benefit from parallelism. The algorithm is highly parameterizable and this makes understanding the effects of different parameters mandatory. However, all of its variations exhibit very similar behaviors, and it would be beneficial to study a more abstracted version that is easier to understand. We examine a binary Hoeffding tree with only nominal attributes and majority class classifiers on the leaves. To drive the split process we use the information gain. We ignore any memory management operations seen in Hoeffding tree implementations, such as poor attribute removal and leaf deactivation.

### 3.1.1 Chapter Structure

To understand Hoeffding trees we examine the algorithm from three different points of view:

- The simplified view. Here, we try to explore the main characteristics of the algorithm.
- The profiling view, presents measurements to quantify the benefits of any optimization discussed in the available parallelism section.
- The available parallelism view, presents the potential parallelism available in Hoeffding trees.

## 3.2 Simplified Algorithm View

Hoeffding trees as a data stream classification algorithm provide two methods, train-on-instance and test-on-instance, to handle train and test instances respectively. Under this interface, Hoeffding trees try to model the data stream by keeping statistics organized to the leaves of a decision tree structure. Leaves could be though as independent entities that provide an interface of three methods, train-on-instance, test-on-instance and get-best-split-suggestions. A Hoeffding tree will use these to operate. Two types of attributes exist, nominal and numeric, but we examine here only the nominals. The two types have very different needs from the leaves.

### 3.2.1 Train on Instance

Train on instance will run for every training instance. Figure 3.1 presents the corresponding UML activity diagram. When a new instance arrive, it will be

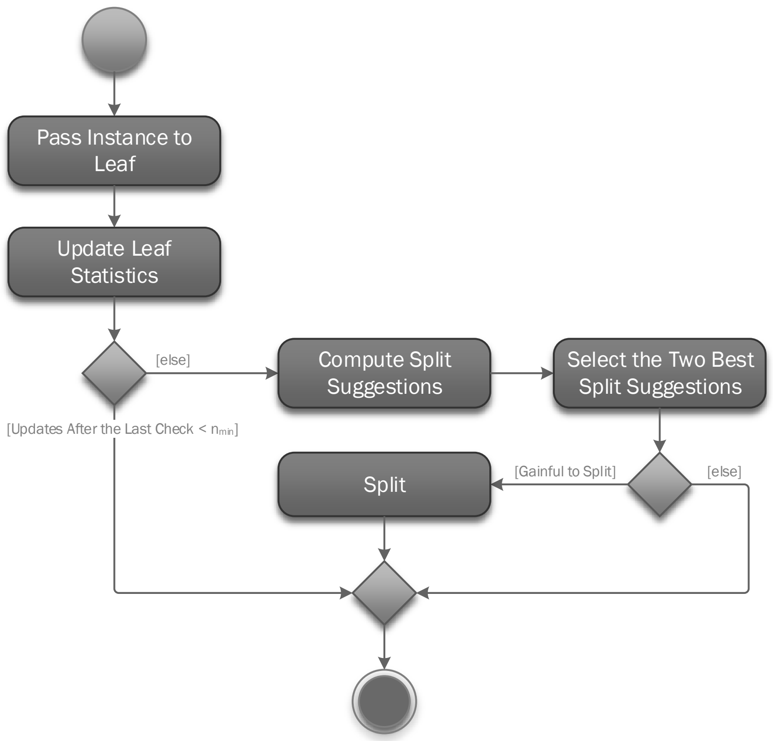
Pass Instance to Leaf
Update Leaf Statistics
[else]
Compute Split Suggestions
Select the Two Best Split Suggestions
[Updates After the Last Check < $n_{min}$]
Split
[Gainful to Split]
[else]

Figure 3.1: UML activity diagram for train on instance operation.

passed through the root to a leaf and its attribute values will be used to update the leaf. If number of instances seen on this leaf is greater than the parameter $n_{min}$, the algorithm will heuristically search for split suggestions and will evaluate them with aid of a function, such as information gain. If the best of them satisfy the Hoeffding inequality, then it will be applied to the leaf. The leaf will be replaced by a node that points to new leaves.

### 3.2.2 Test on Instance

Figure 3.2 shows the UML activity diagram for the generic test on instance function. The system will sort the instance to a leaf and the will use the leaf statistics and the attribute values of the instance to make a class prediction. Furthermore, some classifiers may also update some statistics after the prediction (e.g. adaptive methods may keep statistics about accuracy to select between different alternatives).

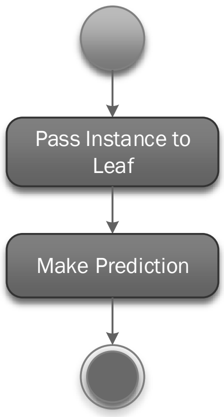

Figure 3.2: UML activity diagram for test on instance operation.

### 3.2.3 Leaves

Many types of leaves exist, but we concentrate on leaves - majority class classifiers, with only nominal attributes. In the case where all attributes have the same number of numbers, the statistics for these leaves form a cuboid. Furthermore, each instance defines a set of (attribute, class, value) vectors. Leaves count the occurrences of these vector, and statistics essentially form a cuboid structure, where the three dimensions represent the attribute, class and value and each point holds the number of its occurrences. Figure 3.3 shows a graphical 3D representation of the statistics. Many of these points may be zeros. Hence, many implementations use lists to store the statistics.

## 3.3 Profiling

To profile the algorithm we used MOA (13). MOA (Massive Online Analysis) is an open source framework for data stream mining. The profiling was done with Netbeans IDE 8.0 in a Q6600 2.4 GHz with 2GB RAM system. We used Java 7 update 65 (OpenJDK). The installed OS was Fedora 20 (Heisenbug). The MOA version was 2014.4.

Due to high overhead the execution was very slow limiting the number of instances that we could examine. Processing only some thousands of instances results in a tree with only a few nodes. This means that the sort-to-leaf operation will take less time than in a system with thousands of nodes. Thus, to overcome the case of examining only very small tree structures, we wrote two extension classes for MOA that help us to run the experiment into two phases.

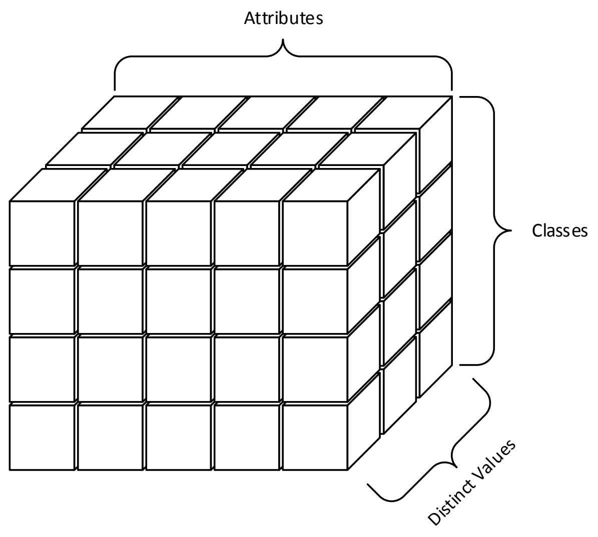

Figure 3.3: The statistics inside a leaf with only nominal attributes, represented as cuboid.

Each class undertakes one phase. In the first phase, we train a Hoeffding tree classifier, and generate a bucket of instances to be used for profiling. We store them both in a file through the Serializable Interface of Java. In the second phase, we load them from the file and use the instances to either train or test the classifier. The two phases can run in different machines, under a different environment. In this manner, we run the first phase in a server, and the second phase in aforementioned environment, where we profiled it.

We configured the algorithm to run as much close to the described algorithm as it was possible. We set the limit for the memory consumed by the tree and the memory estimation period to their maximum values, so that no memory estimation will be triggered. Moreover, we disabled poor attribute removal, we selected the majority class classifier as the leaf prediction method and we set the grace period $n_{min}$ (the number of instances seen by a leaf before we check for a split) to 1.

Two data streams have been used for profiling. Table 3.1 presents their main characteristics. We focus on characteristics that directly affect the per-

| | Configuration 1 | Configuration 2 |
|---|---|---|
| Stream Generator | RandomTreeGenerator[1,2] | |
| Number of Attributes | 10 | 32 |
| Number of Classes | 4 | 8 |
| Number of Attribute Values | 32 | 64 |

Table 3.1: The two configurations used for profiling.

[1]The maximum depth of the tree concept is 3.

[2]The first level of the tree that can have leaves, above the maximum depth, is 2.

formance. A random generator is used to generate the instances. Note that the nature of the stream may also have impacts in performance.

We concentrate on the training process, but we profile both the training and the testing procedures. We expect that training will be much more heavier that the testing as testing. When profiling the training procedure, we expect:

- The most time consuming part will be to compute the best split suggestions.
- The cost to traverse the tree structure normally increases with the number of processed training instances, as the tree grows and the paths from the root to the leaves become longer.
- The cost to update the leaf statistics (learn from instance) remains nearly constant. Moreover, it may execute for every instance too but it only updates one element of the leaf statistics for each attribute.
- The cost to compute the best split suggestions may decrease while the tree grows. The main reason is that splits drop attributes. For binary splits, where we have "attribute-equal-to-value" and "attribute-not-equal-to-value" branches, the sub-tree under the equal branch will only get instances that will have only the corresponding value for this attribute.
- The cost to compute the best split suggestions will be higher for the second configuration.

When profiling the training procedure, we expect:

- The most time consuming part will be to traverse the tree structure.
- The cost to traverse the tree structure normally increases with the number of processed training instances, as the tree grows and the paths from the root to the leaves become longer.
- The cost to find the majority class will be constant as it is proportional to the number of classes.

Table 3.2 presents the measurements to train a tree with a stream of the first configuration, while table 3.3 presents the measurements for the second configuration. As expected, the most time consuming part in processing a training instance is the computation of the split suggestions, its cost decreases while the tree grows, and it is much heavier for the second configuration where we have more attributes, classes, and values. The cost to update the leaf statistics (learn from instance) is not that stable. Especially in configuration 2, it shows an increasing trend together with the tree. Nevertheless, its growth is insignificant and it may be incidental. Moreover, for configuration 2 the cost to sort instances to leaves grows as expected, but for configuration 1, oddly, it decreases. This decrease may be incidental too.

Tables 3.2 and 3.3 presents the corresponding results where we profile the testing process. We took the same files generated to profile the training pro-

|  | Training Instances Seen Before Profiling | | | |
|---|---|---|---|---|
|  | 10K | 100K | 1M | 10M |
|  | Time for 10K instances (ms) | | | |
| → Train-on-instance | 6336.771 | 5188.522 | 5785.802 | 4235.383 |
| ↳ Attempt to split | 6241.265 | 5119.335 | 5639.917 | 4146.075 |
| ↳ Compute Best Split Suggestions | 6171.487 | 5051.770 | 5572.282 | 4080.867 |
| ↳ Learn from instance | 60.235 | 28.618 | 114.036 | 61.992 |
| ↳ Filter instance to leaf | 15.919 | 22.065 | 13.580 | 9.147 |

Table 3.2: Time to train 10K instances for configuration 1. The instances contain 10 attributes, 4 classes and 32 values.

|  | Training Instances Seen Before Profiling | | | |
|---|---|---|---|---|
|  | 10K | 100K | 1M | 10M |
|  | Time for 10K instances (ms) | | | |
| → Train-on-instance | 79941.498 | 79306.900 | 78547.427 | 76036.711 |
| ↳ Attempt to split | 79829.019 | 79164.291 | 78352.249 | 75751.794 |
| ↳ Compute Best Split Suggestions | 79474.482 | 78828.976 | 78006.590 | 75422.133 |
| ↳ Learn from instance | 59.424 | 69.097 | 117.534 | 201.724 |
| ↳ Filter instance to leaf | 11.674 | 30.462 | 36.427 | 43.138 |

Table 3.3: Time to train 10K instances for configuration 2. The instances contain 32 attributes, 8 classes and 64 values.

cedures, but we used the instance buckets as testing instances instead. As expected, the cost to sort an instance to a leaf determines the overall cost of the test-on-instance operation, the cost to find the majority class on a leaf is almost constant, and the cost to traverse the tree structure grows with every new training instance. Note that, paradoxically, the cost to sort an instance to a leaf seems different between the train-on-instance and test-on-instance operations. This seems even more strange because they begin sharing the same tree structure. Someone could argue that the reason for this is that when profiling the training process the tree continuous to grow. However, a tree trained with millions of instances, cannot be affected that much. Maybe the reason is that in test-on-instance operations a much larger portion of the memory accesses refer to the tree structure, and a larger portion of the caches will contain tree nodes.

| | Training Instances Seen Before Profiling | | | |
|---|---|---|---|---|
| | 10K | 100K | 1M | 10M |
| | Time for 10K instances (ms) | | | |
| → Test-on-instance | 16.436 | 48.168 | 176.344 | 68.965 |
| ↳ Filter instance to leaf | 5.150 | 40.932 | 165.951 | 61.461 |
| ↳ Find the Majority Class | 4.353 | 3.971 | 5.291 | 4.669 |

Table 3.4: Time to test 10K instances for configuration 1. The instances contain 10 attributes, 4 classes and 32 values.

| | Training Instances Seen Before Profiling | | | |
|---|---|---|---|---|
| | 10K | 100K | 1M | 10M |
| | Time for 10K instances (ms) | | | |
| → Test-on-instance | 19.249 | 51.076 | 96.326 | 98.335 |
| ↳ Filter instance to leaf | 6.845 | 43.004 | 88.275 | 89.064 |
| ↳ Find the Majority Class | 4.784 | 4.387 | 4.436 | 5.418 |

Table 3.5: Time to test 10K instances for configuration 2. The instances contain 32 attributes, 8 classes and 64 values.

## 3.4 Available Parallelism

The available parallelism could be divided into two categories, according to whether it primarily optimizes throughput or latency. For clarity, throughput and latency refer to instance processing. Due to the fact that usually we have countless instances available, improving throughput is more important than latency. Obviously, by decreasing latency, throughput increases and in cases where the rate of instance arrivals exceed the processing power, a throughput increase will decrease latency. Similarly, parallelism could be divided into parallelism between different instances and parallelism inside the processing of one instance. Figure 3.4 visualize all available types of parallelism. For different instances, we can traverse the tree structure in parallel, and if they reach different leaves we could also process them in parallel. Some leaf operations are permitted to execute simultaneously even if they refer to the same leaf. For example, the test-on-instance operation for implementations that do not update statistics for test operations. Finally, it is better to break up the computation into sub-problems and examine the parallelism of each new problem independently. The following subsections describe the available parallelism in

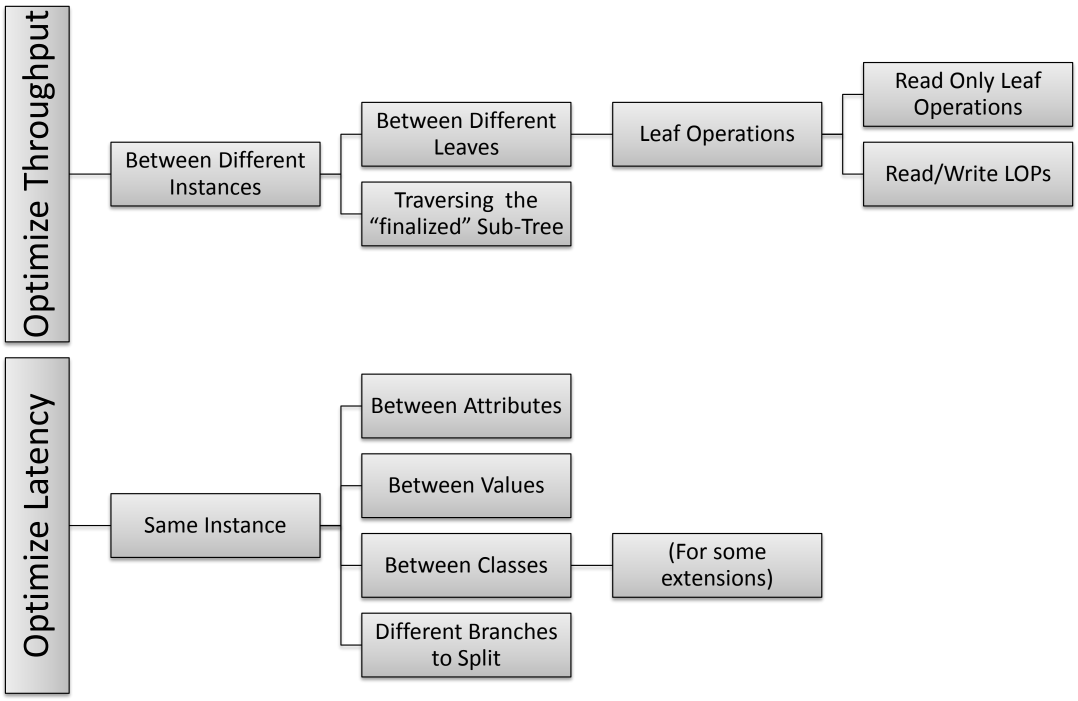

Figure 3.4: Condensed diagram of all types of parallelism present on Hoeffding trees.

various cases.

### 3.4.1 Traversing the Tree

A very important type of parallelism when traversing the tree structure is the parallelism between different parts of the tree. The partitioned method presented in section 2.3 exploits this type of parallelism.

However, this is not the best way to exploit parallelism in our case, where tree nodes are static and only leaves can be updated. Here, we have unlimited parallelism available and in a shared memory system we can just allow multiple workers to work on the whole tree except the leaves. In a distributed memory system, where workers work on different address spaces, a tree replication is possible.

Furthermore, leaves should be treated differently. There, in an update, it may be beneficial to wait for it to complete until sorting new instances to this particular leaf. Alternatively, we could submit the instance for processing in the particular leaf and continue with the next instance. However, instances sorted to different leaves can be processed simultaneously. The capability of computing on different leaves simultaneously is discussed in the next subsection.

### 3.4.2 Parallel Leaf Processing

Leaf operations that do not conflict could be executed in parallel. Processing of different leaves does not conflict. Operations that do not alter the leaves (read-only) does not conflict between each other, even if they refer to the same leaf, but they conflict with operations that update the leaf. In the later case, system must guarantee that read-only operations will always read consistent data and usually the order of execution must be preserved. Operations that modify the leaves generally conflict with any other processing to the same leaf.

### 3.4.3 Update Statistics

A training instance sorted to a leaf will trigger one or more routines to update its statistics. For nominal leaves, we have at least the statistics that form the cuboid presented in figure 3.3. An instance having several attributes could update these attributes in parallel. If we have a batch of instances we can do even more. Every training instance defines an ensemble of (attribute, class, value) pairs that define a different point in the statistics cuboid. Multiple instances define a larger set of (attribute, class, value) pairs, but some of them will refer to the same points of the cuboid. A solution is to generate (attribute, class, value, count) vectors with reduce operations and use them to update the statistics.

### 3.4.4 Compute Best Split Suggestions

To compute the best split suggestions actually we execute a number of smaller sub-tasks:

1. Heuristically select the candidates for a split on each attribute.
2. Evaluate all the candidates.
3. Select for each attribute the best candidate.
4. Select the two best candidates from all attributes.

Obviously, tasks 1, 2, and 3 could execute in parallel for different attributes. Task 1 and 2 (especially 2) could also execute in parallel for different split candidates even on the same attribute.

# Chapter 4

# System Architecture

## 4.1 Introduction

This chapter present a flexible architecture designed to be the basis for more complex architectures. Moreover, we could define an extra layer of hardware to utilize the hardware modules presented here, possibly extended, together with other hardware modules, to compose a field specific solution.

## 4.2 Hardware - Software Partitioning

The problem is composed of two independent entities. These are the main tree structure and the leaves. Consequently, a carefully designed solution should be partitioned into the two corresponding parts. We analyzed these two parts and found that it would be better to implement the leaf functionality into hardware, while leaving the host to handle the tree structure. Some reasons for this decision include:

- Leaf operations seem much heavier than traversing the tree.
- It is very easy to increase the throughput for tree traversals with off-the-shelf hardware. Hence, the host processor is enough to leverage the speedups on the overwhelmingly heavier leaf operations.
- A hardware based implementation of the whole algorithm, restricts the possible benefits from research, as it removes the flexibility to explore how a high throughput system could be utilized to refine the algorithm or to accelerate other algorithms that employ the same leaf operations.
- The tree structure could also contain scheduling functionality that is easier to implement in software.

Except from these, it is better to implement first a small kernel, observe the results and then expand it. By implementing leaf operations on hardware and letting the tree for software, we define a hardware - software partition. As we see, the aforementioned partitioning is the most appropriate, but in the future someone could also implement other modules that manage the tree in hardware.

For high performance computing a key property is that we can separate the tree structure from the main leaf data. Thus, the tree and the leaves could reside in two completely different address spaces. Figure 4.1 presents the case where the host holds the tree structure and a small fraction of each leaf, while a coprocessor has the rest of it. The host can run simple leaf operations and dispatch the heavy ones to the coprocessor. In our implementation, where we

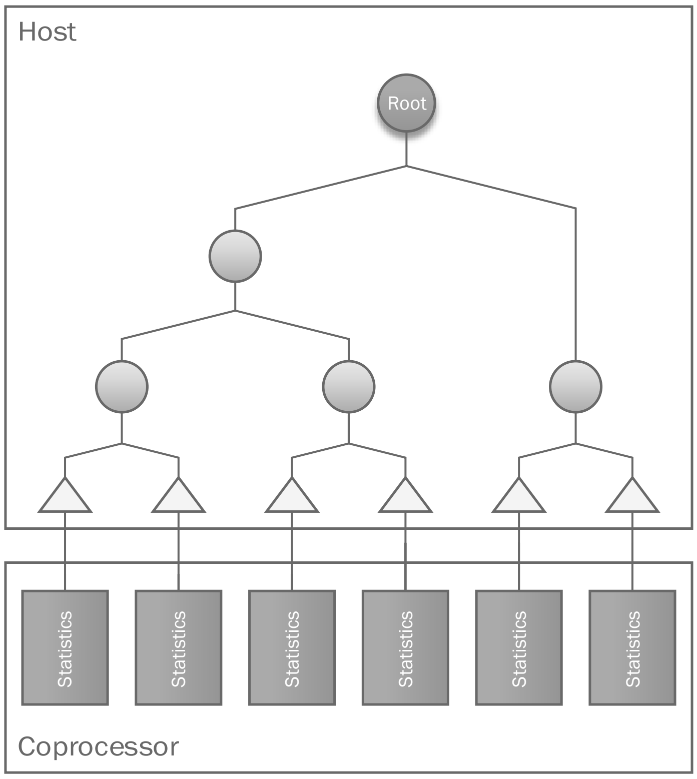

Figure 4.1: Hardware - software partitioning. Leaves consist of two parts the, with each part residing in a different address space.

use majority class classifiers on leaves, test-on-instance operations are completely served by the host, since these operations are very light. In contrast, train-on-instance operations employ both the host and the coprocessor. The host maps instances on leaves with the aid of the tree structure and then either it process them or it triggers leaf operations on the coprocessor, depending on if they are training or test instances.

## 4.3 Instruction Set Architecture

Hardware - software partitioning defines an Instruction Set Architecture (ISA), in its broader sense, as an interface between the software and a hardware module. Depending on the implementation, it may not provide assembly instructions, but a communication protocol instead. However, the two concepts are very close and usually interchangeable. Figure 4.2 presents how the ISA links the software with our hardware modules. We use an ISA consisting of two

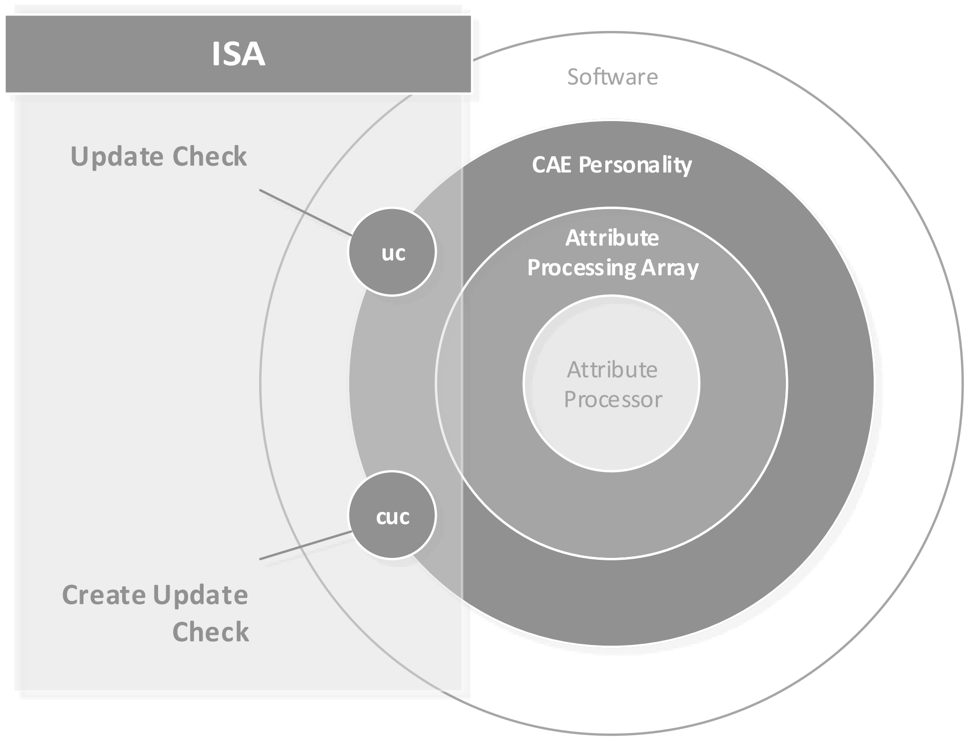

Figure 4.2: The Instruction Set Architecture.

instructions:

**Create, Update and Check (cuc).** The hardware initializes the statistics of the leaf to zeros, updates the leaf with a packet of instances and then computes the best split suggestions. The best suggestions are returned to the host for further processing. Simultaneously, it store the updated leaves to the memory.

**Update and Check (uc).** Same as the cuc instruction but instead of initializing the leaf to zeros it loads the preexisting statistics from the memory.

In our case, where we use the Convey system, these to instructions are implemented as coprocessor assembly instructions.

## 4.4 Hardware Modules

In this section we describe our architecture in a top - down manner. All modules are completely implemented in pure VHDL/Verilog and do not contain any primitive modules nor other vendor provided modules.

### 4.4.1 Extended Attribute Processing Array

EAPA (Extended Attribute Processing Array) is designed to fully utilize a single memory controller port. It can be connected directly to a memory port without intervention of any additional modules, as it contains the required DMA engines. The interface to the memory controller port is the one that is defined from the Convey Computer. Except from the memory interface it also has an simple write-only interface, based upon valid and ready handshaking signals, from where it receives a stream of commands. The format of the commands is shown in figure 4.3. It is designed to utilize less routing resources. Note that actually, in the case where the bit 82 is 1, instruction is the Convey's CAEP instruction itself.

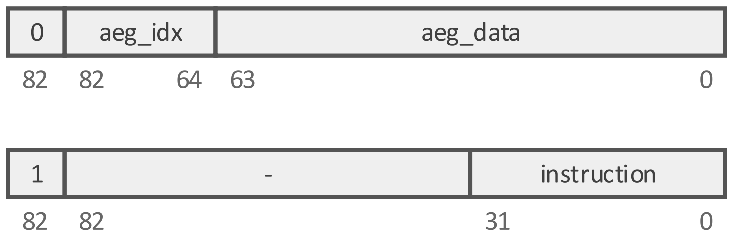

Figure 4.3: Two types of commands exist, write register commands and instructions.

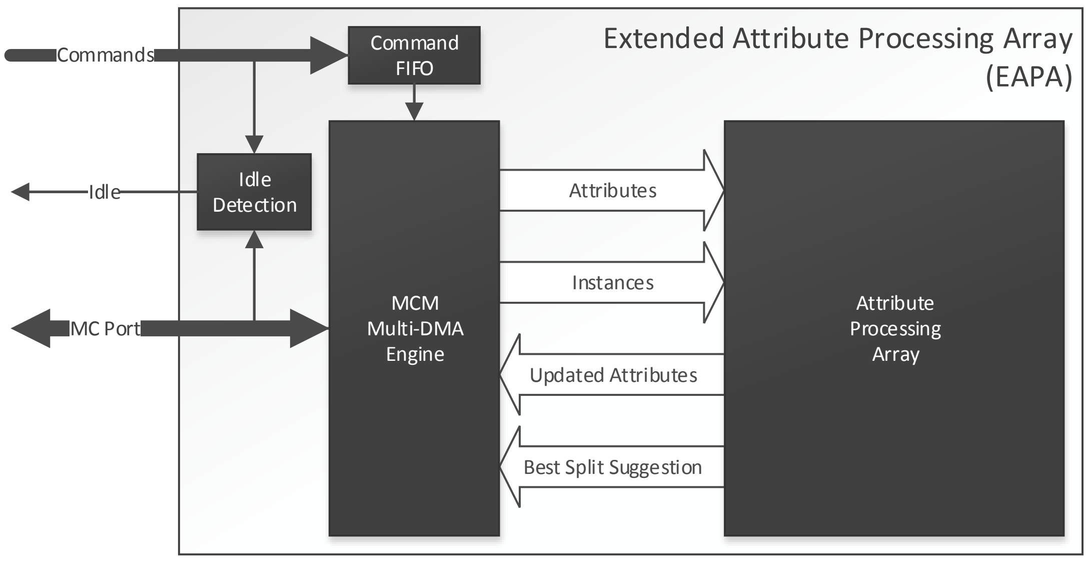

Figure 4.4: The architecture of an Extended Attribute Processing Array.

Figure 4.4 presents EAPA the architecture. Commands are queued in a FIFO and then sent to the Memory Controller Management Unit (MCMU). MCMU is essentially a DMA unit, which undertakes the communication with the memory port and manages four streams to send or receive data from the Attribute Processing Array. Attribute Processing Array or simply APA manipulates the attribute statistic and returns the best suggestions to apply a split. Convey system requires an idle signal to work. Idle Detection unit holds bookkeeping information to generate this signal.

Attribute Processing Array and its sub-modules are very carefully designed with many generic parameters. Memory Controller Management Unit is also carefully designed, but it does not have that level of flexibility from generic parameters.

### 4.4.2 Memory Controller Management Unit

Memory Controller Management Unit (MCMU) communicates with the memory controller through the provided by Convey, memory controller port interface. As figure 4.5 shows, it contains four distinct DMA engines, that handle the data streams to and from the memory controller. These are Leaf Fetch, Instance Fetch, Leaf Store, and BSS Store modules. Memory requests are multi-

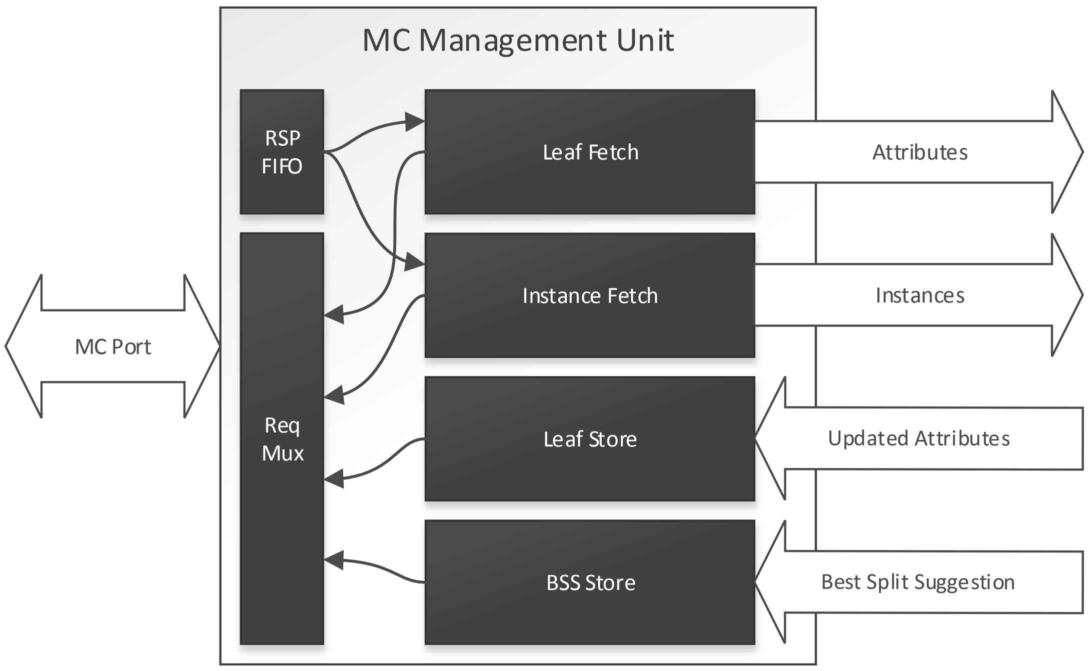

Figure 4.5: The architecture of Memory Controller Management Unit.

plexed by the Request Multiplexer and sent to the memory controller through a memory controller port. Request Multiplexer internally use a round robin arbiter to select who will transmit in the next clock period. However, the arbiter will not grant the MC Port to a module that is not ready to immediately send data. The responses from read requests are queued to the Response FIFO and wait to get accepted from the corresponding modules. All four DMA engines are designed to use 8-byte bursts when it is possible to exploit the Convey Coprocessor memory subsystem. Furthermore, the MCMU is designed to keep busy the a memory controller port that runs at the same frequency with it.

### 4.4.3 Attribute Processing Array

The Attribute Processing Array or APA is shown in figure 4.6. The DMA modules feed APA with attributes and instances. Dispensers are responsible for delivering the data to the Attribute Processors. APA receives the attributes one by

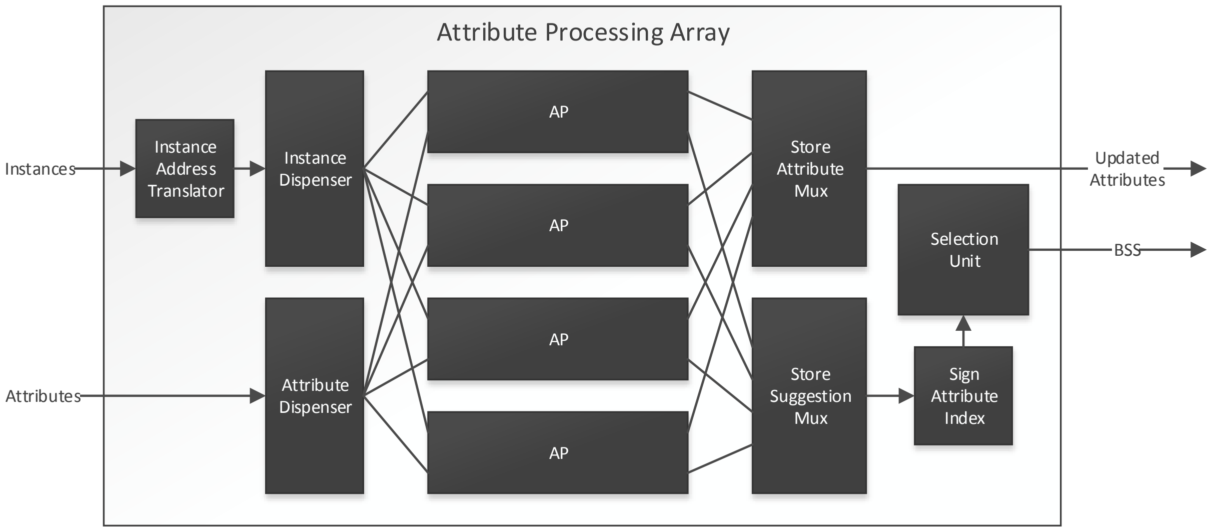

Figure 4.6: The architecture of Attribute Processing Array.

one. Similarly, it receives instance packets for the corresponding attributes one by one. Dispensers and Multiplexers work in a round robin manner and together control the execution order. Essentially, APA processes attributes in order, but hides any visible processing time behind IO operations. We need only a few of APs because APs also hide processing time in IO operations with pipelining. The number of APs can be changed through a generic parameter. Instances packets are <class index, value index> vectors dedicated to one attribute. Each vector derived from one instance. Multiple instances can be combined to generate instance packets. Instance Address Translator is needed to transform a vector to an address inside the BRAM that resides in Update Cache and holds the attribute statistics. Sign Attribute Index module signs the attribute index so that the Selection Unit could find which index generated the best split suggestion (BSS). Selection Unit compares the Split Suggestions and selects the BSS and the second best split suggestion. Updated attributes and the BSS are sent to the corresponding DMA modules dedicated for store operations.

### 4.4.4 Attribute Processor

This is the core of the system. It is responsible for updating attributes and computing the best attribute split suggestions. For clarity, with "best attribute split suggestions" we mean that every time it will process an attribute it will compute the best suggestion for this attribute. Figure 4.7 presents the architecture of an Attribute Processor. Update Cache is responsible for updating the attribute. The stream of the updated attributes is cloned into two streams,

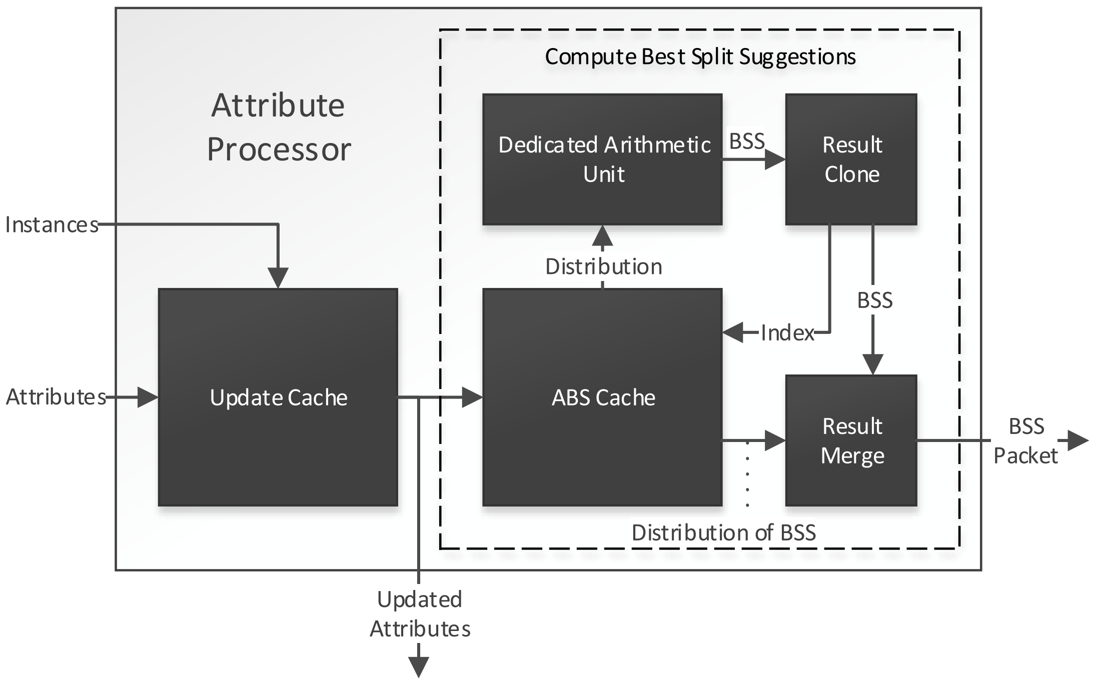

Figure 4.7: The architecture of Attribute Processor.

where one goes out to be stored in the main memory, while the other goes to the ABS Cache (Attribute Best Suggestion Cache). ABS Cache together with Dedicated Arithmetic Unit (DAU), Result Clone and Result Merge, cooperate to compute and return the best attribute split suggestion. ABS Cache holds the statistics to evaluate the split suggestions, but also transforms them in the correct form to be processed from the DAU. DAU computes the best attribute split suggestion from the provided statistics. Result Clone sends the index of this suggestion to ABS Cache. Then, ABS Cache will respond with the corresponding distribution. Result Merge will append the distribution to the suggestion packet. The best attribute split suggestion is sent out for further processing.

### 4.4.5 Update Cache

Update Cache is responsible for updating attribute statistics with instances. As it was mentioned in previous chapters, instances contain values for a number of attributes and may also have class labels. Training instances always contain a class. We can view them as a set of <attribute, value, class> vectors, where each vector refers to a different attribute. When having more than one instances that sort to the same leaf, we can combine these vectors. This results in having multiple vectors per attribute. We can partition these vectors on the attributes and thus sort packets of <value, class> vectors to each attribute of the leaf. Three facts make this the best choice for us:

- We can update attributes one by one before sending them to another module to check for split suggestions, all in an attribute-oriented pipeline.

- We need to design simple hardware. We could have an external pipe that updates the leaf when a new instance arrive. It seems appealing because each training instance alters only a few of the leaf statistics (just one value per attribute). Nevertheless, this will result in addressing very complex problems in areas such us memory coherence, task scheduling and communication.
- We are able to expose attribute locality by gathering multiple instances that sort to a leaf and applying all the updates for each attribute at the same time without a penalty.

Update Cache is attribute oriented, as it loads one attribute from the input stream interface, it updates it in the aforementioned manner and then stores the updated attribute to the output stream interface. Then it can proceed to the next attribute. Figure 4.8 presents the architecture. It contains a BRAM-

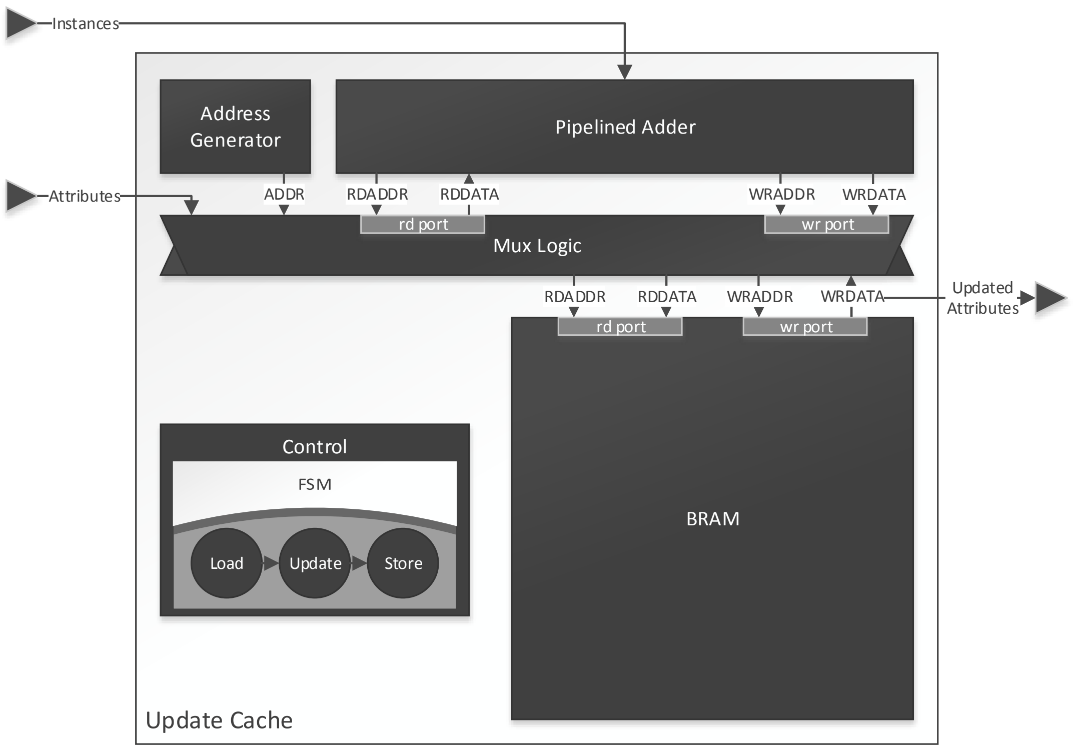

Figure 4.8: The abstracted architecture of Update Cache.

based memory, a pipelined adder, an address counter, the multiplexing logic, and the control. Control mainly comprises of an FSM. The interface of the module is composed of three streams, two input streams that deliver newly loaded attributes and instance packets, and one output stream, the stream of the updated attributes. The multiplexing logic is used to manage the accesses to the BRAM-based memory. The adder accepts instance packets and updates the correct memory entries. The address generator generates addresses to write or read from BRAM memory, when loading or storing attributes respectively. Notice, Update Cache does not accept commands to operate, but it contains some generic parameters (parameters that used to generate the module) and also contain some extra signals to get the size of attributes from the corresponding global register. However, these register must not change while Update Cache is not idle.

### 4.4.6 Compute Best Split Suggestions

As shown in figure 4.7, four modules cooperate to compute the best attribute split suggestion. Actually, ABS Cache and DAU make the computations while the other two modules just help to embed the distribution inside the packet with the best attribute split suggestion. Note that some implementations may omit the distribution. We currently use it only for debugging reasons.

#### Theory and Optimizations

Entropy (in bit units) of a discrete random variable $X$ is defined as:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x). \tag{4.1}$$

The conditional entropy of a random variable $Y$ given the occurrence of the value $x$ on an another random variable $X$ is defined as:

$$H(Y \mid X = x) = -\sum_{y \in Y} p(y|x) \log_2 p(y|x).$$

The conditional entropy of a random variable Y given a random variable X is:

$$H(Y \mid X) = \sum_{x \in X} p(x) H(Y \mid X = x).$$

Finally, information gain is defined as:

$$IG(Y \mid X) = H(Y) - H(Y \mid X).$$

There are currently two types of splits in the literature for nominal attributes:

- Binary splits. Each split node contain a marked attribute and a value for it. When an instance reaches a split node, in order to select the correct branch, we have to compare the instance's value of the market attribute to the value stored at the split node. The instance will proceed to one of the branches depending on whether the corresponding attribute has a value equal or not equal with the stored value.

- Multiway splits, where one branch exists for each possible outcome of the marked attribute.

We consider only binary splits.

Essentially, every attribute, including class attributes, is a random variable. We are interested in having a tree structure that is able to predict the class of a testing instance with high accuracy. Thus, it is meaningful to use information gain on class attributes to evaluate the candidates for splitting a leaf. The basic concept behind applying information gain, is the question "What would be the information gain from applying the split?". Let $IG_i(Y \mid X)$ the information gain of the candidate suggestion $i$.

The following transformation is useful as it results in simpler hardware and software.

**Optimization 1.** *We do not have to compute information gain to decide if it is beneficial to apply the best split suggestion.*

*Proof.*

$$\begin{aligned} IG_i(Y \mid X) - IG_j(Y \mid X) &> \epsilon \\ (H(Y) - H_i(Y \mid X)) - (H(Y) - H_j(Y \mid X)) &> \epsilon \\ H(Y) - H_i(Y \mid X) - H(Y) + H_j(Y \mid X) &> \epsilon \\ -H_i(Y \mid X) + H_j(Y \mid X) &> \epsilon \end{aligned}$$ ■

The following optimization is very important when evaluating the split suggestions.

**Optimization 2.** *The solution with that provides the best information gain is the one with the smallest conditional entropy.*

*Proof.* Assume that we compare two candidates for splitting a leaf, and thus $i \neq j$.

$$\begin{aligned} IG_i(Y \mid X) &> IG_j(Y \mid X) \\ H(Y) - H_i(Y \mid X) &> H(Y) - H_j(Y \mid X) \\ -H_i(Y \mid X) &> -H_j(Y \mid X) \\ H_i(Y \mid X) &< H_j(Y \mid X) \end{aligned}$$ ■

This optimization is far more important that optimization 1. It facilitates the selection of the best split suggestions from hardware modules. To select the best suggestions, we just have to find the ones that provide the smallest conditional entropy.

Let us examine the conditional entropy for binary splits. As shown in figure 4.9, we have two branches, the "equal" and the "not equal". Note that $X$ is now defined as a random variable with two outcomes one for each branch, because an instance sorted to this subtree will follow one these after the split. We denote the two outcomes with $x_1, x_2$. $Y$ has one outcome for each class. We have:

$$H(Y \mid X) = \sum_{x \in X} p(x) H(Y \mid X = x)$$

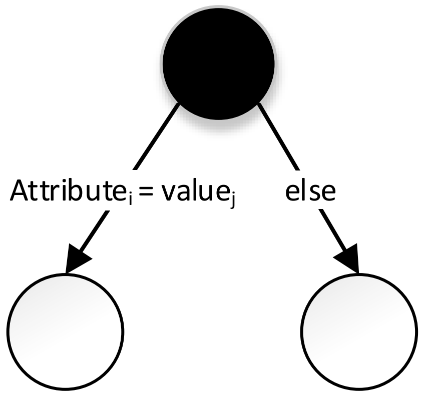

Figure 4.9: Binary split.

$$= -\sum_{x\in X} p(x) \sum_{y\in Y} p(y|x)\log_2 p(y|x).$$

As the attribute statistics are counts of occurrences of <class, value> vectors, we do not really have the probabilities available. We denote these counts with the letter $C$ with a proper index. We have:

$$\begin{aligned} p(y|x) &= \frac{p(y,x)}{p(x)} \\ &= \frac{\frac{C_{y,x}(y,x)}{C}}{\frac{C_x(x)}{C}} \\ &= \frac{C_{y,x}(y,x)}{C_x(x)}. \end{aligned}$$

$C_x(x)$ denotes the weight that the branch $x$ would observe if we had applied this split suggestion. $C_y(y)$ denotes the weight observed for the class $y$. $C_{y,x}(y,x)$ denotes the weight that the branch $x$ would observe for the class $y$ if we had applied this split suggestion. Obviously:

$$C_x(x) = \sum_y C_{y,x}(y,x) \tag{4.2}$$

and

$$C_y(y) = \sum_x C_{y,x}(y,x). \tag{4.3}$$

For a binary split suggestion where we have two branches, $C_{y,x}(y,x)$ is a 2D matrix, as shown in figure 4.10. However, the statistics for each attribute form a matrix that is different from this one. Figure 4.11 presents the attribute statistics as a matrix. We denote this matrix as $C_{y,z}(y,z)$, where $y$ and $z$ are the class and value indices respectively.

But how to compute the $C_{y,x}(y,x)$ matrix of each suggestion from $C_{y,z}(y,z)$ matrix? The answer depends from the split suggestions that we want to evaluate. For splits, such as the split presented in figure 4.9, for the "equal" branch we have

$$C_{y,x_1}(y,x_1) = C_{y,z}(y,v),$$

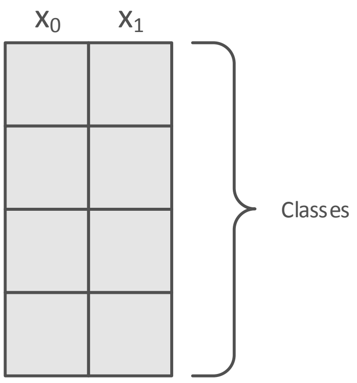

Figure 4.10: The statistics needed to evaluate a binary split suggestion. We have to compute this matrix from the attribute statistics.

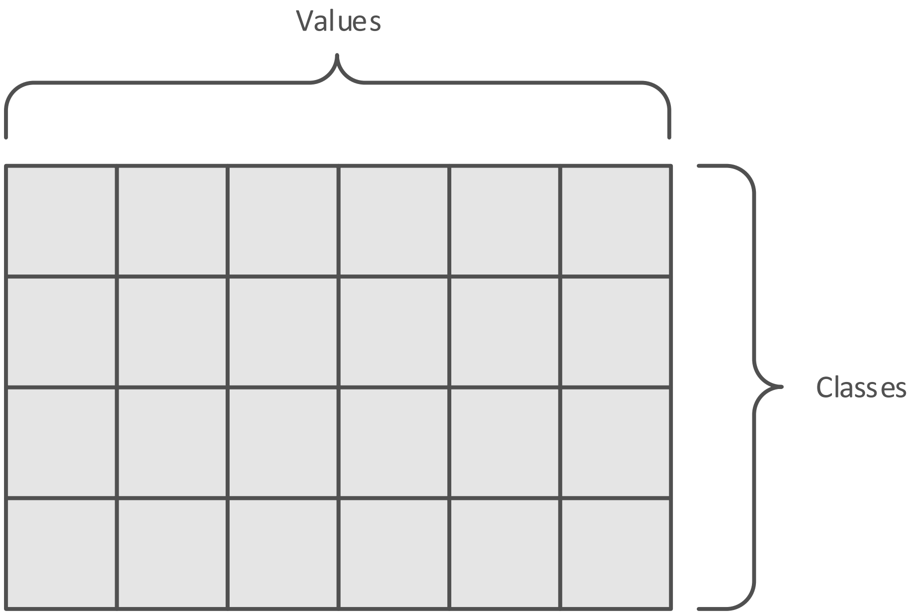

Figure 4.11: The statistics for a nominal attribute as a matrix.

and for the "not equal" branch

$$C_{y,x_0}(y, x_0) = \sum_{z \neq v} C_{y,z}(y, z)$$

where $v$ is the value that determines the split suggestion. So

$$C_{y,x}(y, x) = \begin{cases} \sum_{z \neq v} C_{y,z}(y, z) & \text{if } x = x_0 \\ C_{y,z}(y, v) & \text{if } x = x_1 \end{cases}$$

Now we are ready to reexamine the conditional entropy, as we have a way to compute the needed probabilities, and so:

$$H(Y \mid X) = -\sum_{x \in X} p(x) \sum_{y \in Y} p(y|x) \log_2 p(y|x)$$

$$
\begin{aligned}
&= -\sum_{x\in X}\frac{C_x}{C}\sum_{y\in Y}\frac{C_{y,x}}{C_x}\log_2\frac{C_{y,x}}{C_x}\\
&= -\frac{1}{C}\sum_{x\in X}\frac{C_x}{C_x}\sum_{y\in Y}C_{y,x}\left(\log_2 C_{y,x}-\log_2 C_x\right)\\
&= -\frac{1}{C}\sum_{x\in X}\sum_{y\in Y}\left(C_{y,x}\log_2 C_{y,x}-C_{y,x}\log_2 C_x\right)\\
&= -\frac{1}{C}\sum_{x\in X}\left(\sum_{y\in Y}C_{y,x}\log_2 C_{y,x}-\sum_{y\in Y}C_{y,x}\log_2 C_x\right)\\
&= -\frac{1}{C}\sum_{x\in X}\left(\sum_{y\in Y}C_{y,x}\log_2 C_{y,x}-\left(\sum_{y\in Y}C_{y,x}\right)\log_2 C_x\right)\\
&= -\frac{1}{C}\sum_{x\in X}\left(\sum_{y\in Y}C_{y,x}\log_2 C_{y,x}-C_x\log_2 C_x\right)\\
&= \frac{1}{C}\sum_{x\in X}\left(C_x\log_2 C_x-\sum_{y\in Y}C_{y,x}\log_2 C_{y,x}\right)\\
&= \frac{1}{C}\sum_{x\in X}\left(C_x\log_2 C_x-\sum_{y\in Y}C_{y,x}\log_2 C_{y,x}\right) \qquad (4.4)
\end{aligned}
$$

The complexity to compute $C_{y,x}$ together with $C_x$ is $O(cv)$, where $c$ is the number of classes and $v$ the number of values. Having $C_{y,x}$ available, the cost to compute $H(Y \mid X)$ for one binary split suggestion is $O(c)$. The total complexity is $O(c + cv) = O(cv)$. Nevertheless, we have one split suggestion for each attribute value. Thus, we have $O(cv^2)$ to compute the best attribute binary split suggestion for one attribute.

**Optimization 3.** *The complexity to evaluate all eqv/neqv*[1] *binary split suggestions for one attribute can drop from $O(cv^2)$ to $O(cv)$, by using row sums.*[2]

*Proof.* With $C_y$ available, we have:

$$
\begin{aligned}
C_{y,x}(y,x) &= \begin{cases}\sum_{z\neq v} C_{y,z}(y,z) & \text{if } x = x_0\\ C_{y,z}(y,v) & \text{if } x = x_1\end{cases}\\
&= \begin{cases}\sum_{z\neq v} C_{y,z}(y,z) + C_{y,z}(y,v) - C_{y,z}(y,v) & \text{if } x = x_0\\ C_{y,z}(y,v) & \text{if } x = x_1\end{cases}\\
&= \begin{cases}\sum_{z} C_{y,z}(y,z) - C_{y,z}(y,v) & \text{if } x = x_0\\ C_{y,z}(y,v) & \text{if } x = x_1\end{cases}\\
&= \begin{cases}\sum_{z} C_{y,z}(y,z) - C_{y,z}(y,v) & \text{if } x = x_0\\ C_{y,z}(y,v) & \text{if } x = x_1\end{cases}
\end{aligned}
$$

[1] One branch for equal, one for not equal like figure 4.9.

[2] It is notable that MOA framework does not use this optimization. The other two optimizations are not that important for software but this one seems very gainful.

$$= \begin{cases} C_y(y) - C_{y,z}(y,v) & \text{if } x = x_0 \\ C_{y,z}(y,v) & \text{if } x = x_1 \end{cases} \tag{4.5}$$

From equation (4.2) we have:

$$\begin{aligned} C_x(x) &= \sum_{y \in Y} C_{y,x}(y,x) \\ &= \begin{cases} \sum_y (C_y(y) - C_{y,z}(y,v)) & \text{if } x = x_0 \\ \sum_y C_{y,z}(y,v) & \text{if } x = x_1 \end{cases} \\ &= \begin{cases} \sum_y C_y(y) - \sum_y C_{y,z}(y,v) & \text{if } x = x_0 \\ \sum_y C_{y,z}(y,v) & \text{if } x = x_1 \end{cases} \\ &= \begin{cases} C - C_z(v) & \text{if } x = x_0 \\ C_z(v) & \text{if } x = x_1 \end{cases} \end{aligned} \tag{4.6}$$

So we can compute $C_y$, $C_z$, and $C$, once per attribute, all together in $O(cv)$. Then for each value compute $C_{y,x}(y,x)$ in $O(c)$, and $C_x(x)$ from $C$ and $C_z(v)$ in $O(1)$. Now, to compute $H(Y \mid X)$ for this attribute we need $O(c)$. Hence the total cost to compute $H(Y \mid X)$ for all attributes is now $O(cv)$, because we need $O(cv)$ to compute $C_y$, $C_z$, and $C$ once in the beginning and $O(cv)$ to compute $H(Y \mid X)$ for each suggestion. ■

**Optimization 4.** *We do not need to divide with $C$ until the very end, when applying the Hoeffding inequality. Then again, we could avoid it by comparing their difference with the result of the multiplication $C\epsilon$.*

*Proof.* It is obvious when looking at equation (4.4) because $C$ remains the same for all attributes. ■

Note that most optimizations presented here may have implications with missing values. Nevertheless, a possible solution for nominal attributes is to just define a new value to denote them, and handle them as values. Furthermore, we have one more optimization for multiway splits presented in appendix B, but we do not use it as we consider only binary splits.

#### Architecture Review

Now we can review "Compute Best Split Suggestions" logic in figure 4.7. ABS Cache will load one attribute and will compute $C_x$, $C_y$ and $C$. Then it will feed the Dedicated Arithmetic Unit with statistics in the appropriate format. DAU evaluates them with the aid of equation 4.4.

### 4.4.7 ABS Cache

Figure 4.12 shows the architecture of ABS Cache. This module is responsible for preparing the statistics to be sent to Dedicated Arithmetic Unit. Control

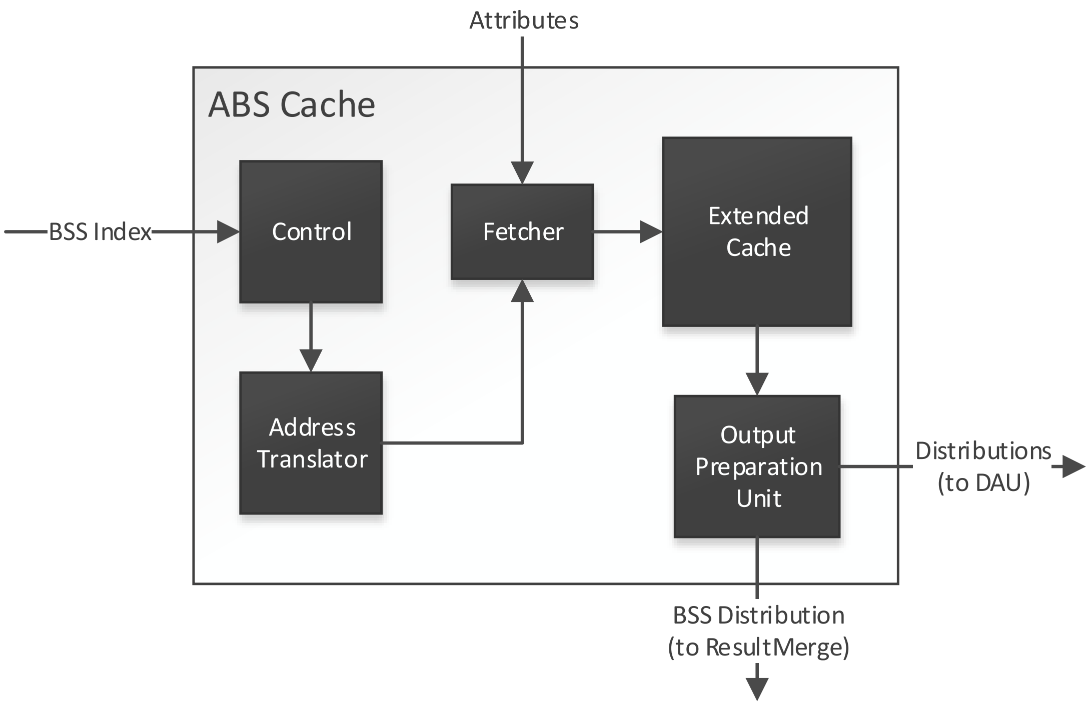

Figure 4.12: The architecture of ABS Cache.

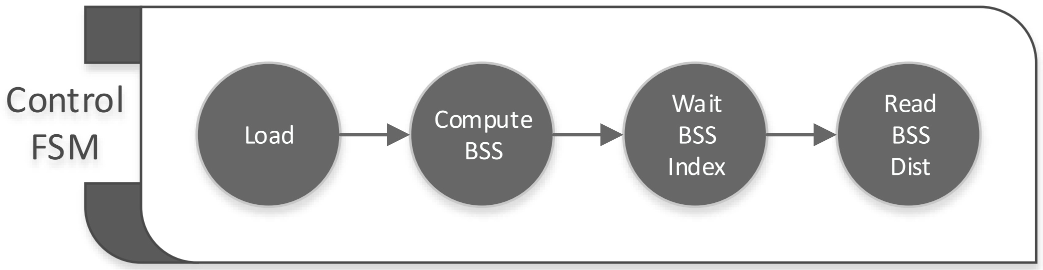

Figure 4.13: The FSM inside Control sub-module.

generates commands that pass all other sub-modules in a pipelined manner. It contains the FSM that is shown in figure 4.13. Four main FSM states are defined:

- In “Load” state it generates commands to load a new attribute.
- In “Compute BSS” state the commands will lead into sending data to the Dedicate Arithmetic Unit.
- In “Wait BSS Index” state the control waits to receive the resulting index of the best attribute split suggestion.
- In “Read BSS Dist” state the control will create the commands needed to emit the distribution of bss.

The generated commands hold the class and value indices, generated with the help of a counter. Commands also contain some bits that describe them.

The indices have to be translated into BRAM addresses. Address Translator always translate the indices to BRAM addresses. Fetcher will recognize load operations and will embed one data word from the input data stream in each load command. Eventually, commands arrive to Extended Cache, which holds and manages the attribute data. Moreover, it is responsible for computing $C_z$, $C_y$, and $C$. Output Preparation Unit is a very simple module that makes some customization to any output data according to their destination.

### 4.4.8 Extended Cache

Extended Cache is the core module of ABS Cache. Figure 4.14 presents its architecture. It is comprised of four main parts:

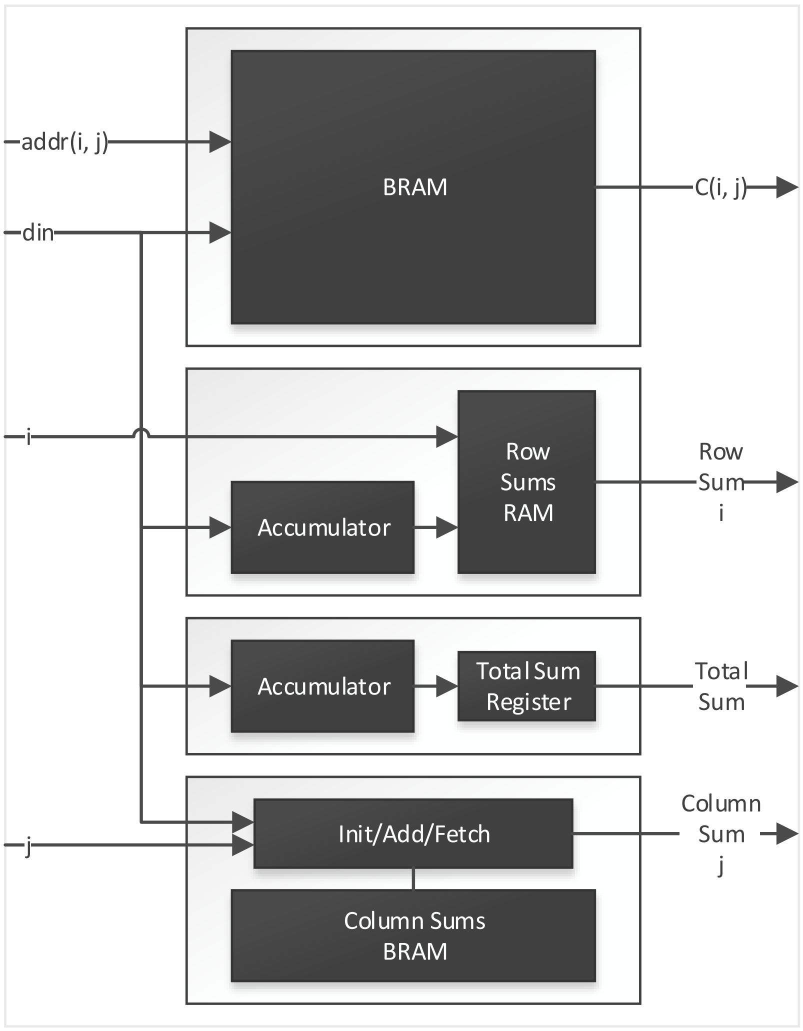

Figure 4.14: The architecture of Extended Cache.

- The BRAM that holds the attribute counts.
- The Accumulator which computes the row sums ($C_y$) and the RAM that stores them.

- The Accumulator that computes the total sum ($C$) and the register that holds it.
- An module to initialize, update, and fetch the column sums ($C_z$) from a dedicated BRAM that holds them.

Extended Cache accepts and serves write and read requests. On write requests, the destination address will be overwritten with the new data. Write requests must be sent ordered row by row. The accumulator that is responsible for computing the row sums will store the row sum on the appropriate position at the end of the row. The accumulator that computes the total sum generates and stores it in the appropriate register at the end of the load procedure. Logic that is responsible for computing the column sums is essentially a BRAM with an adder.

### 4.4.9 Dedicated Arithmetic Unit

Figure 4.15 abstracts the architecture of a Dedicated Arithmetic Unit (DAU). DAU evaluates all split suggestions one by one and returns the conditional en-

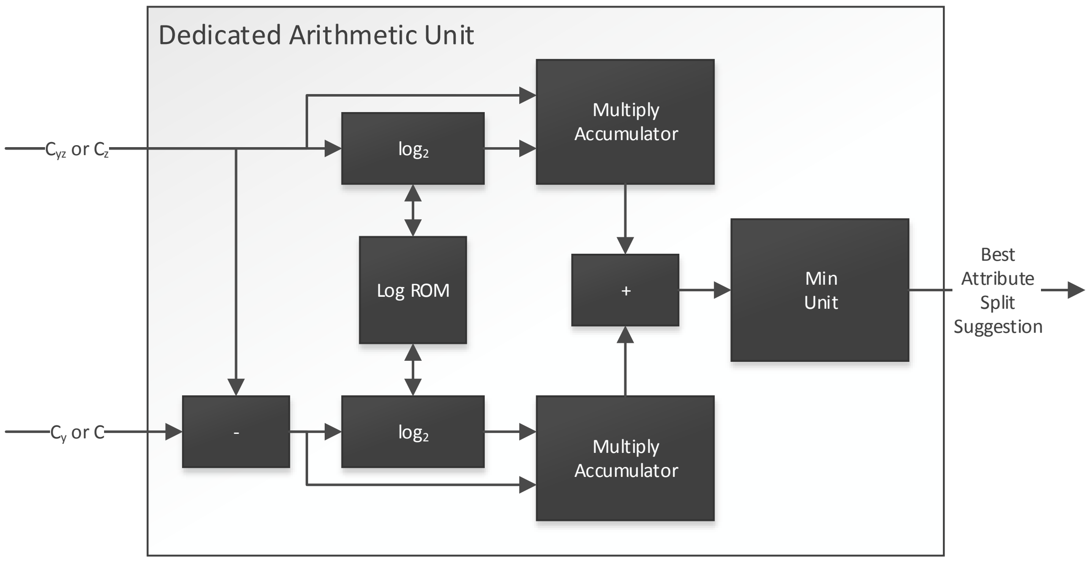

Figure 4.15: T he architecture of a Dedicated Arithmetic Unit.

tropy and the value index of the best attribute split suggestion. As can see from figure 4.15 the module is symmetric. The upper part computes the part of the sum in equation (4.4) where $x = x_1$. The bottom part computes the part of the same sum $x = x_0$. At this point we should clarify that ABS Cache sends a stream of two data words. First, it will send $C_z(v)$ and $C$. Then it will send $C_{y,z}$ and $C_y(y)$ for every $y \in Y$. The subtractor is used to compute $C_x(x)$ and $C_{y,x}(y, x_0)$ from the input stream using equations (4.6) and (4.5) respectively. The two units that compute the logarithms internally use a custom floating point format and a port of the Log ROM to compute the logarithms of the corresponding streams. Multiply accumulators compute all multiplications and summations inside the parenthesis in equation (4.4). The adder computes the outer sum in equation (4.4) by adding the outputs from the multiply accumulators. Finally, the Min Unit selects the minimum entropy and computes its value index.

# Chapter 5

# Verification and Evaluation

## 5.1 Introduction

In the previous chapter we described a general system architecture and a hardware module dedicated to fully exploit one single memory controller port. In this chapter we verify and evaluate our architecture. Although we did not implement a system that employs more memory ports, our system is designed to be generalized in this manner. However, not only we do not need to use more than one modules to verify and evaluate our architecture, but it is also a good design strategy even if we plan to evaluate in multi-memory-port architecture.

## 5.2 Convey HC-1 and HC-$2^{ex}$

It is not in our intentions to describe here neither HC-1 nor HC-$2^{ex}$ in detail, but we need to provide some very general information to clarify the evaluation environment. More information for them can be found in (22, 27, 28, 29, 30).

Convey uses a NUMA memory architecture. Host and coprocessor can access all of the available memory (HCGSM - Hybrid-Core Globally Shared Memory), but each one can access faster its own part of the memory. Figure 5.1 presents the abstracted architecture of a Convey coprocessor with its memory subsystem. Coprocessor is mainly comprised of Application Engine Hub (AEH), Application Engines (AE) and the Memory Subsystem. AEH embeds an implementation of a scalar processor. In AEs the designer implements the custom logic needed to accelerate an application. The custom logic is called personality and the corresponding HDL file is called cae_pers.

Figure 5.2 presents another view of Convey's architecture. When a thread that runs on the host calls a function that was designed to run on coprocessor, Convey will do whatever is needed to prepare the execution of this code in AEH and then the AEH will begin to run the corresponding code. The coprocessor program for a custom architecture usually has to include simple assembly code to interface with Application Engines (AE) FPGAs.

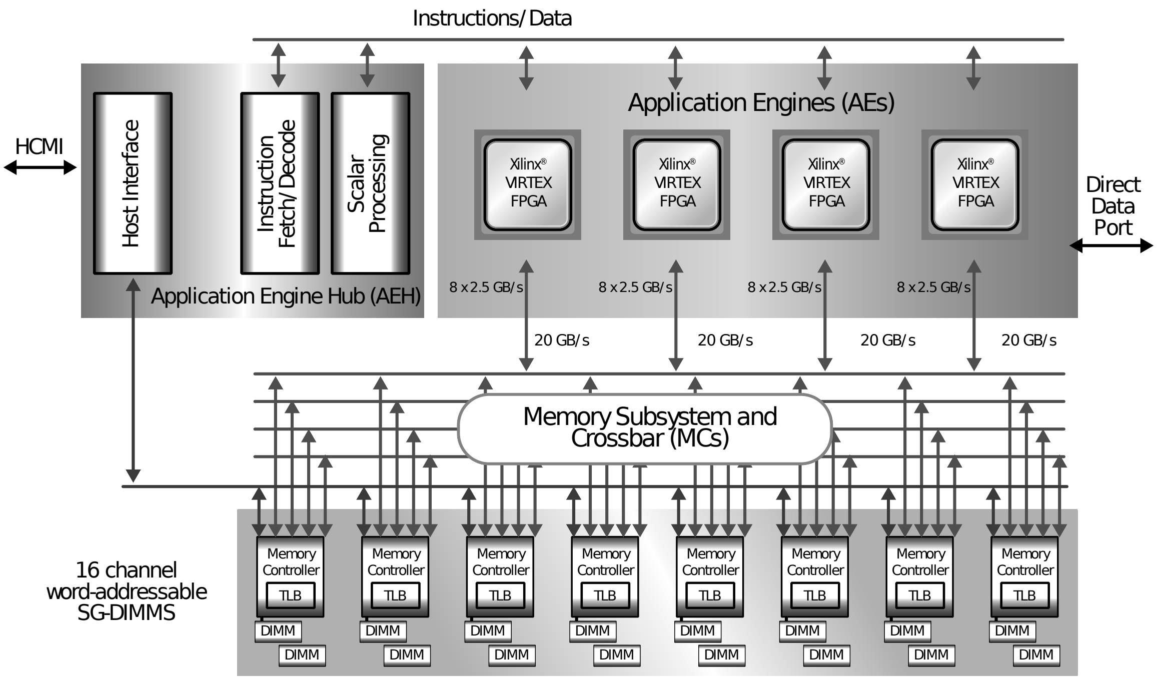
Instructions/ Data
HCMI
Host Interface
Instruction Fetch/ Decode
Scalar Processing
Application Engine Hub (AEH)
Application Engines (AEs)
Xilinx® VIRTEX FPGA
Direct Data Port
8 x 2.5 GB/s
20 GB/s
Memory Subsystem and Crossbar (MCs)
16 channel word-addressable SG-DIMMS
Memory Controller
TLB
DIMM

Figure 5.1: The abstracted memory architecture of a Convey Computer (source: (27)).

## 5.3 System Architecture for Evaluation

Figure 5.3 shows our cae_pers module for testing, verification and evaluation. It is a modified version of the corresponding module of an example distributed together with Convey software and manuals. As we evaluate only one EAPA module alone, we needed only one memory port and we left all other memory ports unconnected. EAPA was designed to be connected directly to the provided memory port interface, so we did not implement any additional logic to connect it. Nevertheless, we needed some very small logic to interface it with the dispatch interface. This extra logic was, more or less, part of the aforementioned Convey's example.

## 5.4 Resource Analysis

This system was designed primarily for HC-2$^{ex}$. In HC-2$^{ex}$ we have four Virtex 6 LX760 (31) available. However, we also prepared a bitfile for HC-1. HC-1 has four Virtex 5 LX330 (32) available instead. To generate the resource reports presented here we run a synthesis under Xilinx ISE 14.7 with the default settings for each device. Tables 5.1 and 5.2 show the reports for a Virtex 6 LX760 and a Virtex 5 LX330 respectively. Available resources in each device were retrieved from the xst reports but were also checked with the descriptions from (31) and (32). The needed resources are very close for the two devices, but the Virtex 6 has a lot of more available. Each APA (Attribute Processing Array) contains by

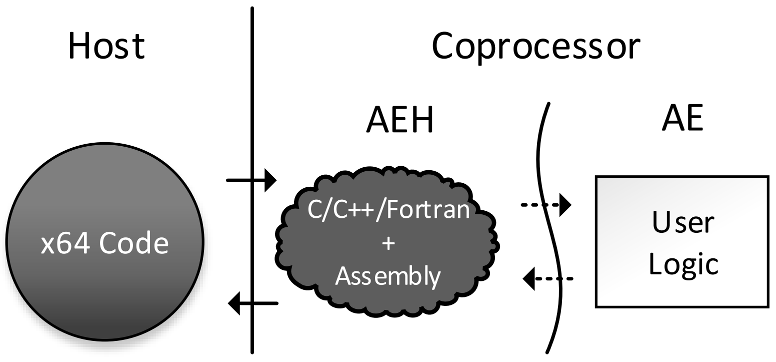

Figure 5.2: The abstracted software architecture in a Convey Computer.

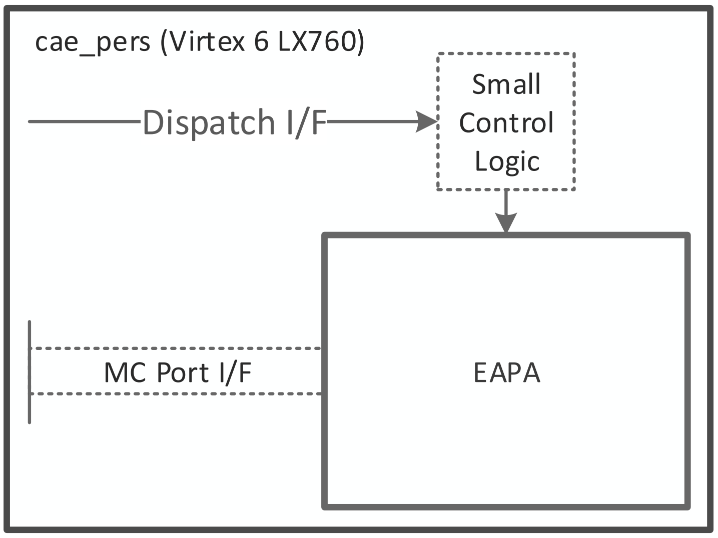

Figure 5.3: Evaluation personality architecture.

default four AP (Attribute Processors), unless the corresponding generic value is set to a different value.

## 5.5 Clock Results

User logic currently runs at 150 MHz but all modules where designed to run at 300+ MHz. The main reason for using a 150 MHz clock is that we do not need more than 150 MHz to handle one MC port. Nonetheless, a higher clock using the same FPGA resources may mean a better hardware utilization, as soon as the used resources remain the same.

## 5.6 Software

We needed a set of software tools to be able to debug, verify, and evaluate our architecture. For this reason, we wrote two MOA plugins, two C-based pro-

| Type | EAPA | APA | AP | Available |
|---|---|---|---|---|
| Slice Registers | 11720 | 9697 | 1816 | 948480 |
| Slice LUTs | 9739 | 7554 | 1782 | 474240 |
| RAMB36E1s | 33 | 24 | 5 | 720 |
| DSP48E1s | 17 | 16 | 1 | 864 |

Table 5.1: Utilization Report for Virtex 6 LX760.

| Type | EAPA | APA | AP | Available |
|---|---|---|---|---|
| Slice Registers | 11700 | 9699 | 1847 | 207360 |
| Slice LUTs | 9619 | 7592 | 1725 | 207360 |
| RAMB36s | 35 | 24 | 6 | 288 |
| DSP48Es | 17 | 16 | 4 | 192 |

Table 5.2: Utilization Report for Virtex 5 LX330.

grams, and one program for Convey together with some Perl and Bash scripts.

### 5.6.1 Test Data Generator

To generate test cases for verification or evaluation we implemented a MOA plugin, named "Test Data Generator". This plugin accepts some configuration arguments and generates the corresponding files for measurements or verification.

We have leaves and commands. Commands are packets of instances with a leaf index and an indicator to know if the leaf has been initialized. We have packets of instances and not single instances because our modules allow to update a leaf with multiple instances together before computing the best split suggestions. Each command references to a different leaf, so that we could easily verify any implemented system that would use parallelism between different leaves. All data are generated with a simple random number generator. This is not a problem because the amount of computational work that has to be done is irrelevant with the selected numbers. The most important input arguments are the number of leaves to generate, the number of attributes per leaf, the number of classes, the number of values per attribute, and the number of instances per command. The generated date are stored in four files:

1. One file contain the leaves.
2. Another file holds the commands. In the end-system these will be the only data that the host will send to the coprocessor.
3. A file to has the leaves after altering them with the corresponding commands.

4. Finally, we have a file to store the return structures. In the end-system these will be the only data that would return to the host.

The generator in a loop will populate a new leaf with random data and will append it in the leaves-before-update file. Then it will create a command with the corresponding instances and will store it to the commands file. Afterwards, it will update the leaf with the command and will save the updated leaf into the leaves-after-update file. Finally, it will compute the best split suggestions and will store the return data to the return-data file.

### 5.6.2 MOA Benchmark

MOA Benchmark is another plugin for MOA that we designed to measure the processing time of the leaves. It will first load the leaves-before-update and the commands files and then it will measure the needed time to process them all.

### 5.6.3 Convey Software Core Structure

Our main software was divided into two parts, a very small assembly part that runs in the coprocessor and a C part that runs in the host. The assembly part is just three small wrapper functions that interface the software with our hardware modules. The two functions are there to interface our two instructions (create-update-check and update-check). The other is to configure any registers that our modules needs, but they remain fixed during an experiment. The C is responsible:

1. To load leaves from a file.
2. Copy them to coprocessor.
3. Load instructions (packets of instances together with a leaf ID and if the referenced leaf will be treated as initialized so we will run create-update-check and not update-check).
4. Initialize the coprocessor. Then set the needed registers by instructing convey to call the adequate initialization method that we implemented in assembly.
5. Begin measuring the time.
6. Dispatch each instruction separately within a loop by coping the packet of instances to the coprocessor memory and ordering convey to call the adequate assembly function.
7. Instruct Convey to execute a fence on the coprocessor outside the loop. This fence does not needed because we use the simple synchronous dispatches to the coprocessor and Convey inserts automatically a fence before returning from the coprocessor.

8. Stop measuring the time.
9. Copy the leaves and the return structures from the coprocessor to the host.
10. If needed execute verification by comparing the results with the expected ones.

Note that, firstly, we have to copy our leaf data into coprocessor as Convey has a NUMA memory architecture and leaves would normally reside in the coprocessor. Secondly, although our architecture is designed to handle different leaves without needing a fence, we measure an automatically inserted fence for each dispatch together with one not needed fence outside the loop.

### 5.6.4 C

Because MOA is written in Java and has a lot of abstraction layers, we also implemented a very "unofficial", simple C, x86-64 version of the benchmark, to catch any cases where the overheads in Java where too high. It is basically the same software like section 5.6.3, but it executes completely in the CPU and it does not use optimization 3. It differs remotely from MOA mainly in that it uses optimizations 1, 2, and 4. Furthermore, MOA when examines an attribute, it will compute the distributions for all of its split suggestions and it will evaluate them to select the one with the best information gain. Our C implementation compute the distribution for one split suggestion and will evaluate it by comparing it to the previous best one. To differentiate from software in section 5.6.3 we used the C preprocessor in the same source files.

### 5.6.5 C-Opt

While we were debugging our architecture using the software presented in sections 5.6.1, 5.6.2, 5.6.3, and 5.6.4, we noticed an impressive speedup of two orders of magnitude for our hardware architecture. Knowing that the optimization 3 could help CPU implementations in cases where we have many values per attribute we implemented a C version of the algorithm like section 5.6.4, but in this case we included optimization 3. Again, to differentiate from software in section 5.6.3 and 5.6.4 we used the C preprocessor in the same source files.

### 5.6.6 Scripts

For verification, debugging and early evaluation we wrote a Perl script that executes the aforementioned software. Later to be able to exhaustively measure the system with a single command, we wrote two Bash scripts. These scripts also use the software presented in sections 5.6.1, 5.6.2, 5.6.3, 5.6.4, and 5.6.5.

## 5.7 System Level Verification

Despite that we ran multiple simulation tests for each module at the design process, some final system-level verification steps were required to fix any problems left in the design and verify the correctness of the results. The simulation ran with the aid of Perl scripts and the provided by Convey workbench to simulate the whole system using behavioral models. Nonetheless, to fix an issue with code that did not translate well in FPGA resources, we also had to run post translate simulation. Finally, we ran and verified our implementation in HC-$2^{ex}$. The system was verified that produces correct results in both system level simulation and real hardware.

## 5.8 Evaluation

Although our architecture is pointing to increase the throughput of leaf processing, a fence is automatically inserted between the processing of different leaves, so this figure presents latency not throughput. The corresponding measurements are in a table format in Appendix C.

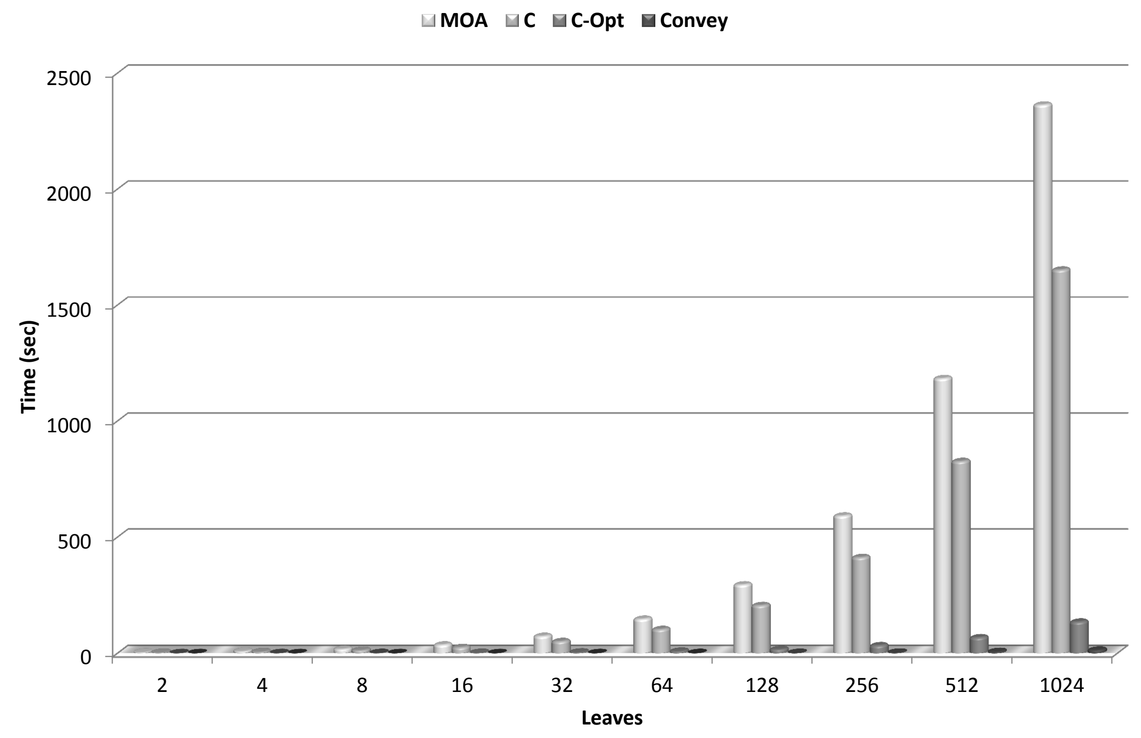

Figure 5.4: Time to process multiple leaves (1 instance per leaf, 1024 attributes, 8 classes, 256 values per attribute.

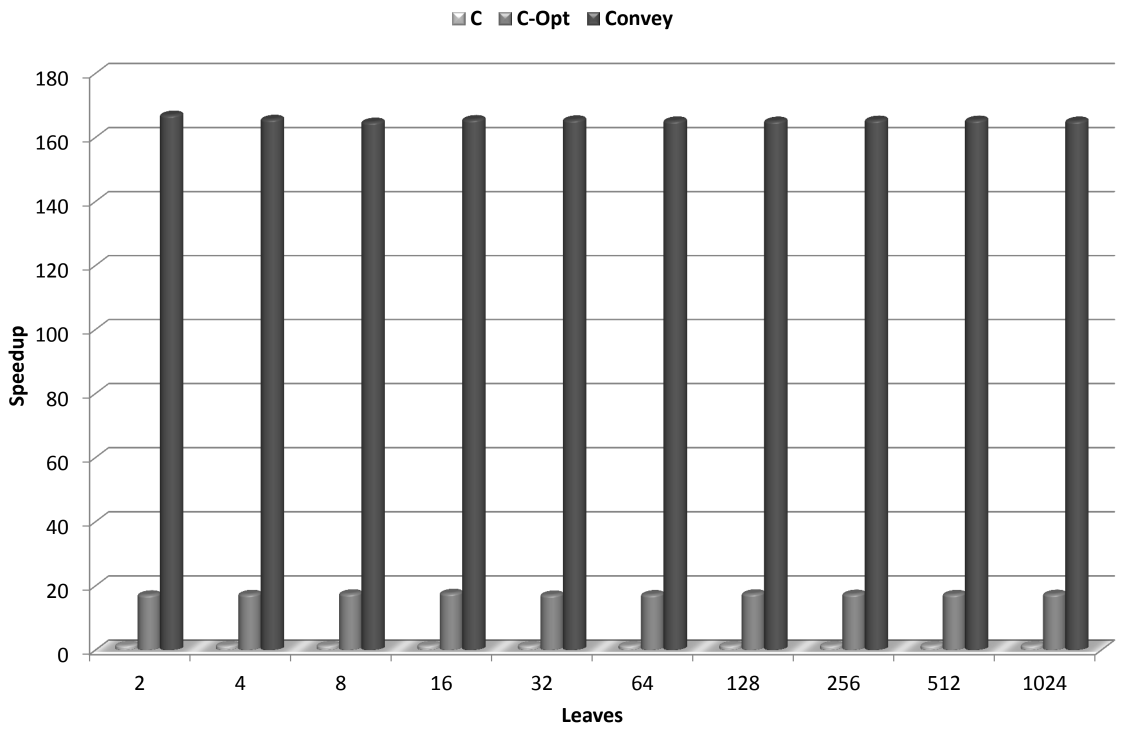

Figure 5.5: Speedup against MOA when processing multiple leaves (1 instance per leaf, 1024 attributes, 8 classes, 256 values per attribute.

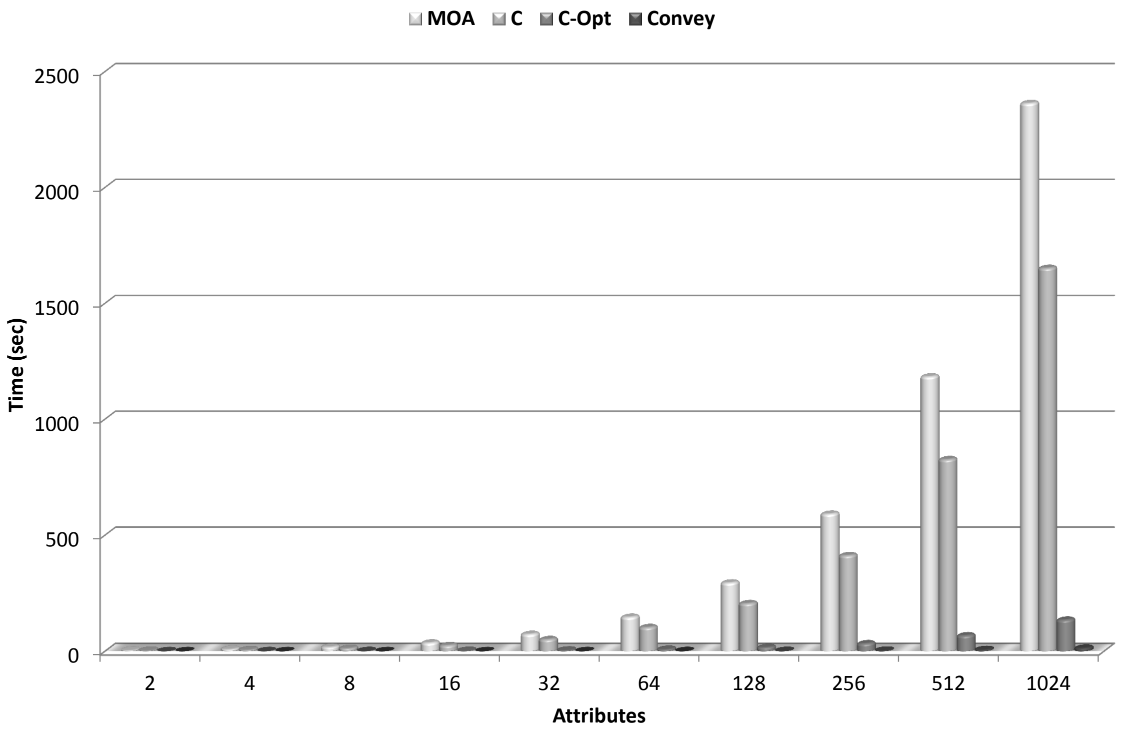

Figure 5.6: Time to process multiple leaves for different number of attributes (1 instance per leaf, 1024 leaves, 8 classes, 256 values per attribute).

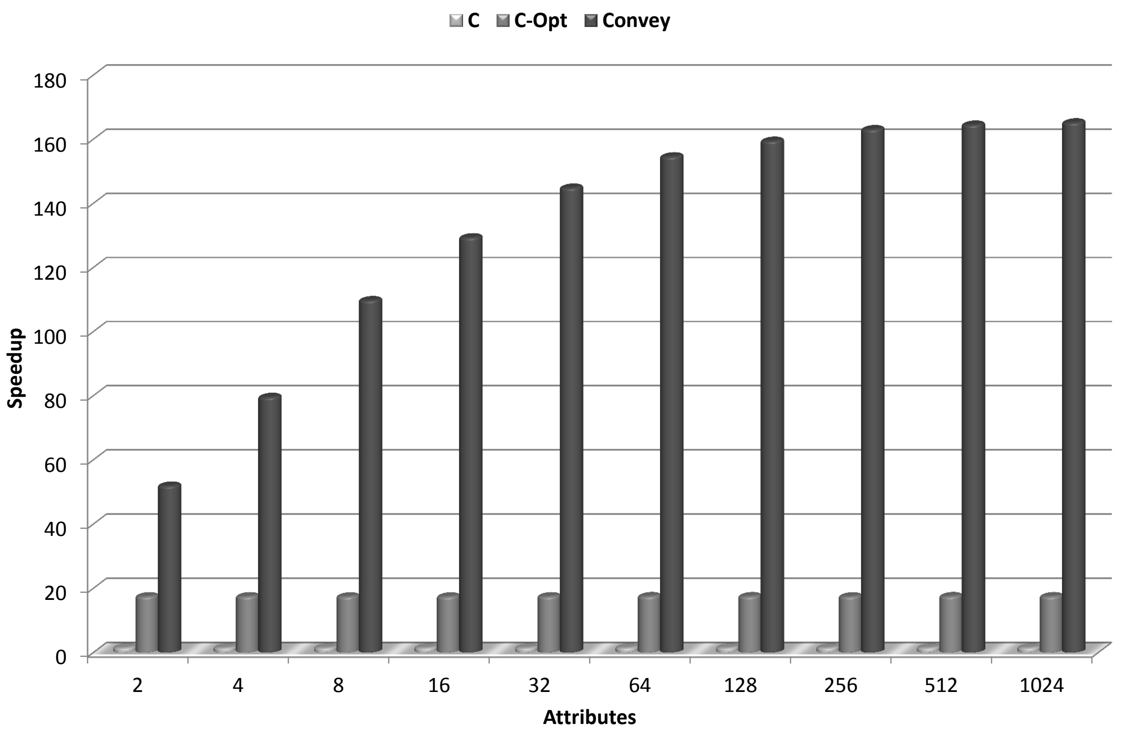

Figure 5.7: Speedup against MOA when processing multiple leaves for different number of attributes (1 instance per leaf, 1024 leaves, 8 classes, 256 values per attribute).

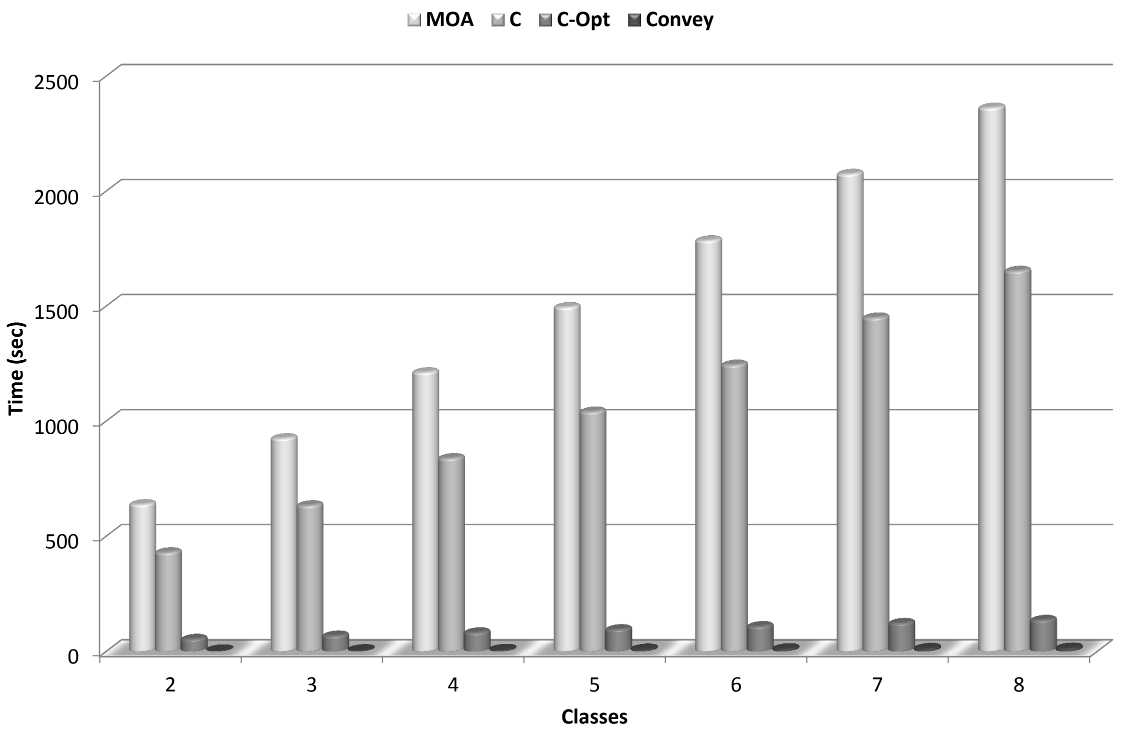

Figure 5.8: Time to process multiple leaves for different number of classes (1 instance per leaf, 1024 leaves, 1024 attributes, 256 values per attribute).

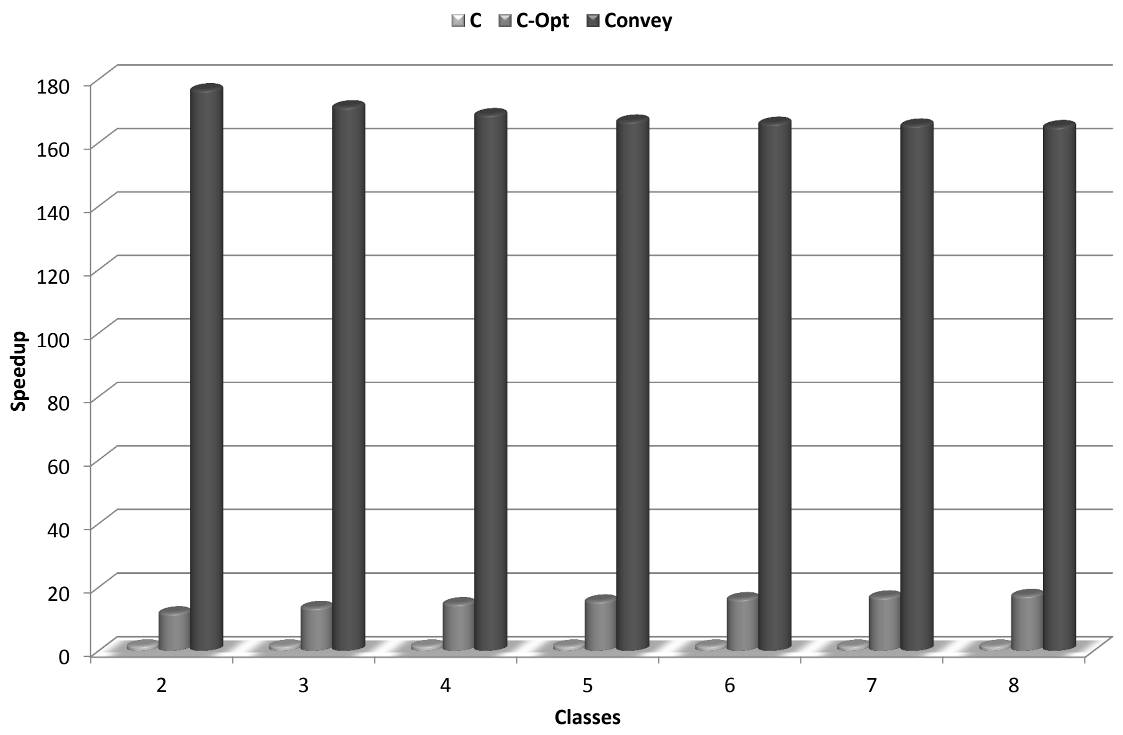

Figure 5.9: Speedup against MOA when processing multiple leaves for different number of classes (1 instance per leaf, 1024 leaves, 1024 attributes, 256 values per attribute).

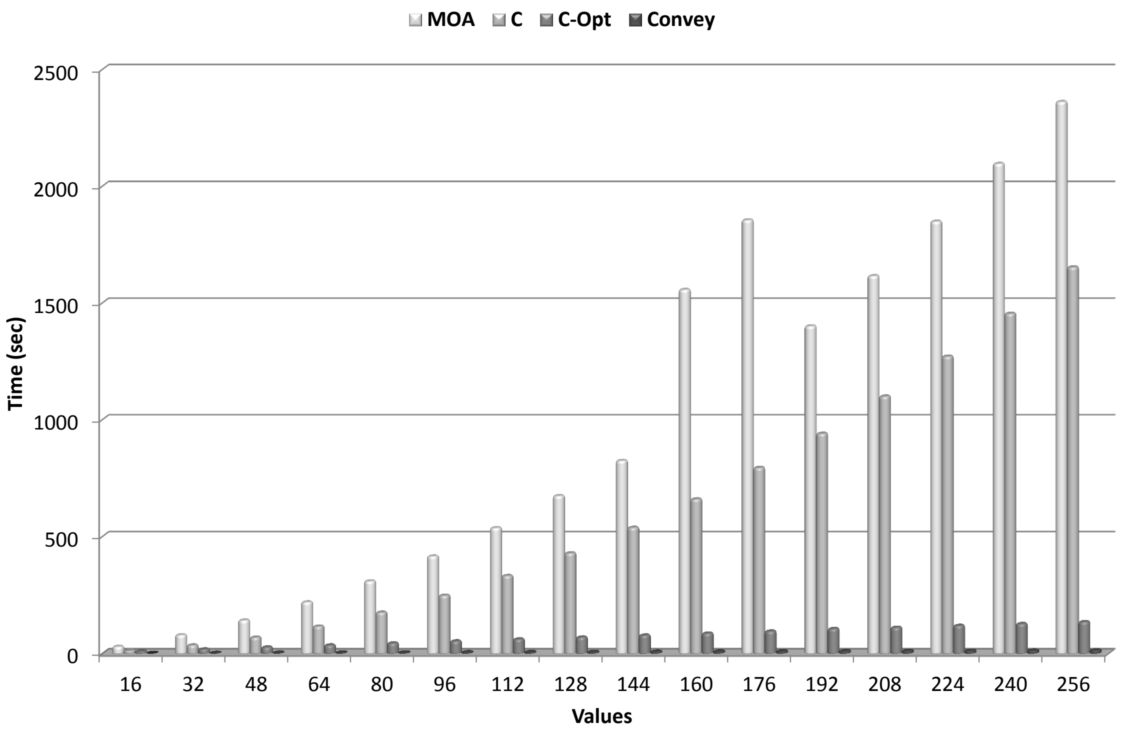

Figure 5.10: Time to process multiple leaves for different number of values per attribute (1 instance per leaf, 1024 leaves, 1024 attributes, 8 classes).

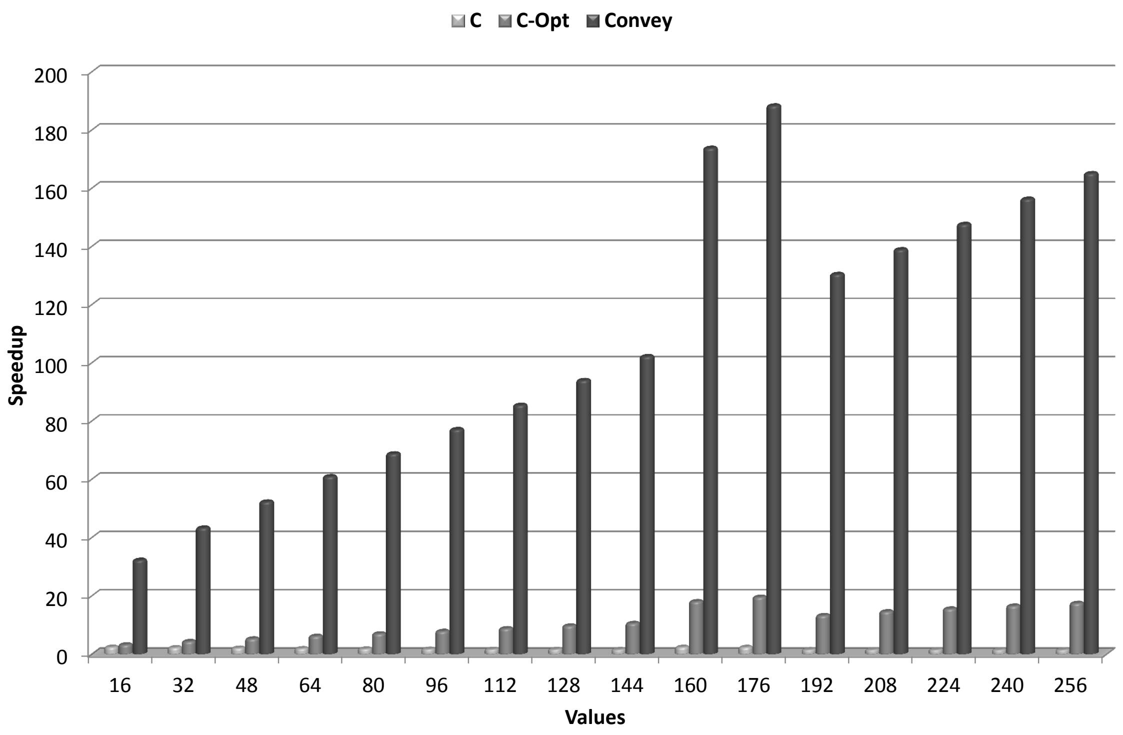

Figure 5.11: Speedup against MOA when processing multiple leaves for different number of values per attribute (1 instance per leaf, 1024 leaves, 1024 attributes, 8 classes).

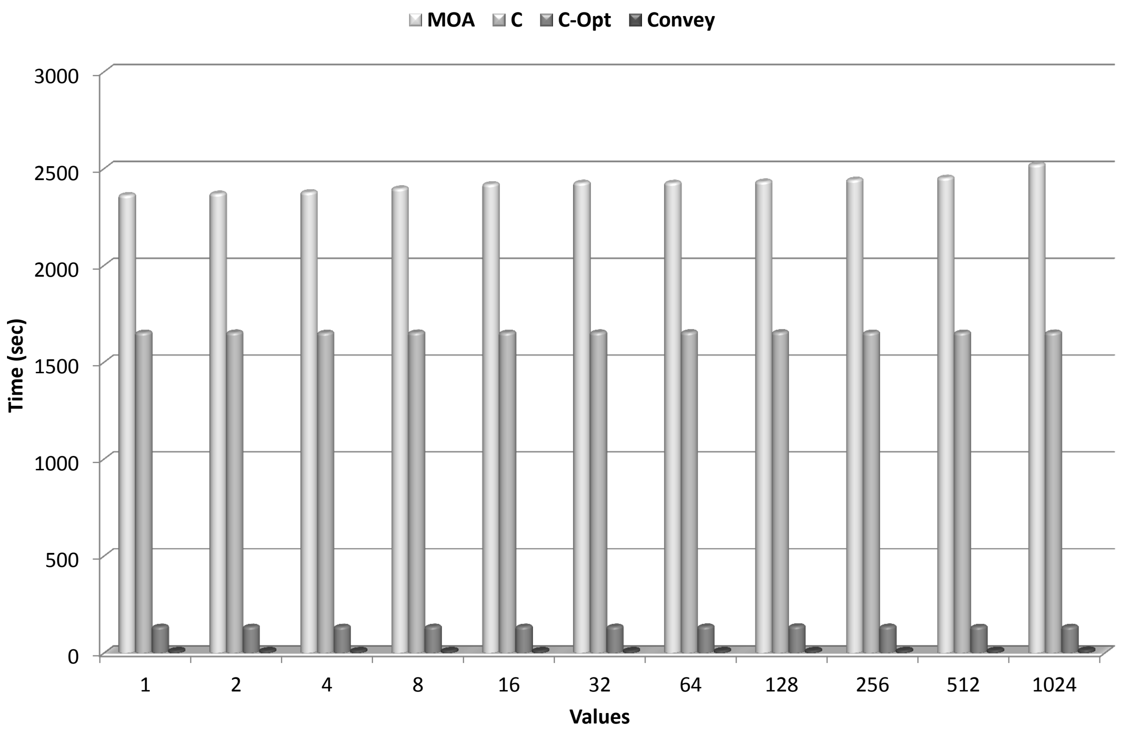

Figure 5.12: Time to process multiple leaves and instances. The number of instances defines the size of the instance packets (1024 leaves, 1024 attributes, 8 classes, 256 values per attribute).

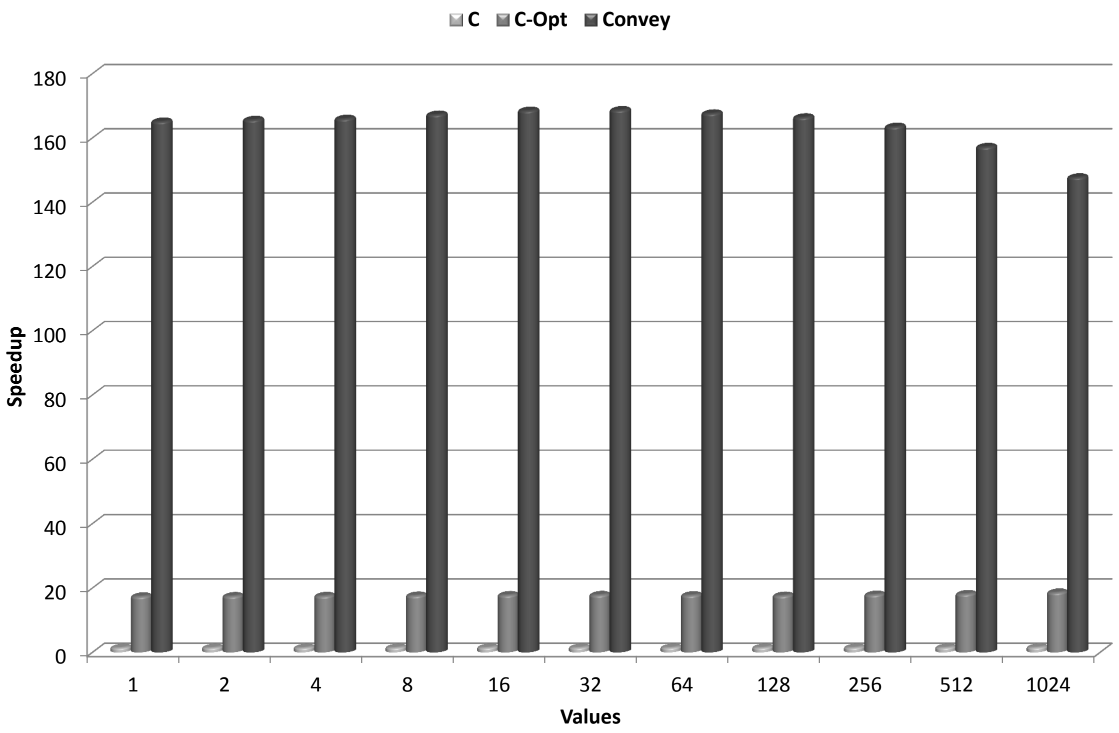

Figure 5.13: Speedup against MOA when processing multiple leaves and instances. The number of instances defines the size of the instance packets (1024 leaves, 1024 attributes, 8 classes, 256 values per attribute).

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

We examined a data stream mining algorithm, the Hoeffding tree, and found, that firstly, there is a lot of parallelism underneath, and secondly, operations that execute on leaf statistics are the most time consuming. The updating of the leaf statistics and the evaluation of the split suggestions for a leaf are good candidates for FPGA implementations. In our case, we found that evaluation, aided by information gain, maps well on FPGA resources.

Convey HC-2$^{ex}$ was found to be an outstanding platform for hardware design and implementation. Although to fully utilize such a system you need to have knowledge in a vast number of areas, it is very flexible with a unique memory organization. At the same time the hardware interfaces are kept clear and simple. Furthermore, HC-2$^{ex}$ is able to provide speedups or throughput increases of two or three orders of magnitude against C-based single threaded implementations. In many cases, it scores even better against implementations that are build upon software frameworks and try to maintain a high level of abstraction.

The paradigm, where an ensemble of modules cooperate in a pipelined manner to process a continuous stream of data, is the best for cases such ours. We implemented many utility modules (accumulators, multiplexers, decoders, multiply-accumulators etc.) for "fine-grained" data streams (streams of independent numbers), and many modules pointing in more "coarse-grained" (packet oriented) streams (streams of attributes). The benefits from a stream-based processing are mainly the partitioned design (modules can be designed independently), the structured, hierarchical result design (the final design will be well structured with a clear hierarchy) and the performance of the final implementation. Moreover, a stream-based pipeline is able to spare FPGA resources, as the designer have the ability to select where in the pipeline is the bottleneck and must provide more resources. Finally, we have a better utilization of the memory subsystem.

## 6.2 Future Work

Ideas for future work include:

- A full implementation of the algorithm in Convey. We currently implemented only the hardware modules, but it would be interesting to how it performs in together with a software implementation of the tree structure in the host.
- Build a new architecture layer upon the currently implemented modules where all memory controller ports in all four FPGAs available will be used. Convey has 16 ports available in each FPGA, and has four FPGAs, giving a total of 64 ports. We currently use only one. The way to easy succeed this, we could exploit the available coarse grained parallelism.
- Try to use the same modules for batch algorithms such as C4.5.
- Convey-based implementation of Hoeffding tree ensembles.
- Implement algorithms that address concept drift.
- Extend the hardware modules to implement more functionality (e.g. multiway splits).
- Provide a full embedded system.

# Appendix A

# Theoretical Bounds

## A.1 About Theoretical Bounds

Theoretical bounds help us to make good split decisions fast and save computational resources. Moreover, we will select the two attributes that currently provide the split suggestions with the highest score. Then, we will apply a theoretical bound to the difference of the two scores (the highest score minus the second highest) to find if it is guaranteed with high probability that it will continue to have a higher score than the second one. Usually, this also guarantees that the attribute with the highest score will continue to have the highest score among all attributes.

## A.2 Reviewing Hoeffding Bound

Let $X_1, X_2, \dots X_n$ independent random variables,

$$S = X_1 + X_2 + \cdots + X_n,$$

$$\bar{X} = \frac{S}{n},$$

$$\mu = E[\bar{X}] = \frac{E[S]}{n},$$

and

$$a_i \leq X_i \leq b_i (i = 1, 2, \dots, n).$$

Hoeffding(8) states that

$$Pr\{\bar{X} - \mu \geq t\} \leq e^{-2n^2t^2/\sum(b_i - a_i)^2}, \tag{A.1}$$

with

$$\sum (b_i - a_i)^2 = \sum_{i=1}^{n} (b_i - a_i)^2 .$$

We need

$$Pr\{\bar{X} - \mu \geq t\} \leq \delta, \tag{A.2}$$

so

$$\begin{aligned}
e^{-2n^2t^2/\sum(b_i-a_i)^2} &= \delta \\
-2n^2t^2/\sum(b_i-a_i)^2 &= \ln\delta \\
-2n^2t^2 &= \ln\delta\sum(b_i-a_i)^2 \\
2n^2t^2 &= -\ln\delta\sum(b_i-a_i)^2 \\
2n^2t^2 &= \ln(1/\delta)\sum(b_i-a_i)^2 \\
t^2 &= \frac{\ln(1/\delta)\sum(b_i-a_i)^2}{2n^2} \\
t &= \sqrt{\frac{\ln(1/\delta)\sum(b_i-a_i)^2}{2n^2}} \qquad \text{(A.3)}
\end{aligned}$$

If all random variables $X_i$ have the same range $R$ then

$$\sum(b_i-a_i)^2 = \sum_{i=1}^{n}(b_i-a_i)^2 = nR$$

and equation A.3 becomes

$$t = \sqrt{\frac{R^2\ln(1/\delta)}{2n}}. \qquad \text{(A.4)}$$

Equation A.2 equivalently says that with probability greater than $1-\delta$,

$$\bar{X}-\mu \le t.$$

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

# Appendix B

# Multiway Splits

## B.1 Introduction

Multiway splits break the leaf into several new ones. One leaf is created for each attribute value. Although equal/not-equal binary splits could reach the same result through multiple splits on the not-equal branch, most implementations support, and sometimes prefer, multiway splits.

## B.2 Observing the Conditional Entropy

Once again, let us reexamine the conditional entropy (equation (4.4)):

$$H(Y \mid X) = \frac{1}{C} \sum_{x \in X} \left( C_x \log_2 C_x - \sum_{y \in Y} C_{y,x} \log_2 C_{y,x} \right)$$

When computing the conditional entropy to evaluate the binary split for one value indexed with $v$ of the attribute we need to compute:

$$\begin{aligned} H_v(Y \mid X)\, C &= \sum_{x \in X} \left( C_x \log_2 C_x - \sum_{y \in Y} C_{y,x} \log_2 C_{y,x} \right) \\ &= \underbrace{\left( C_x(x_0) \log_2 C_x(x_0) - \sum_{y \in Y} C_{y,x}(y, x_0) \log_2 C_{y,x}(y, x_0) \right)}_{A_v} \\ &\quad + \underbrace{\left( C_x(x_1) \log_2 C_x(x_1) - \sum_{y \in Y} C_{y,x}(y, x_1) \log_2 C_{y,x}(y, x_1) \right)}_{B_v} \\ &= A_v + B_v, \end{aligned} \tag{B.1}$$

where

$$B_v = C_x(x_1) \log_2 C_x(x_1) - \sum_{y \in Y} C_{y,x}(y, x_1) \log_2 C_{y,x}(y, x_1). \tag{B.2}$$

We are interested for the second part of the equation, aliased to $B_v$, where $x = x_1$. From equations (4.2) and (4.5) we have:

$$\begin{aligned} C_x(x_1) &= \sum_y C_{y,x}(y, x_1) \\ &= \sum_y C_{y,z}(y, v) \\ &= C_z(v). \end{aligned} \tag{B.3}$$

So from equations (B.2), (B.3) and (4.5) we have:

$$\begin{aligned} B_v &= C_x(x_1) \log_2 C_x(x_1) - \sum_{y \in Y} C_{y,x}(y, x_1) \log_2 C_{y,x}(y, x_1) \\ &= C_z(v) \log_2 C_z(v) - \sum_{y \in Y} C_{y,z}(y, v) \log_2 C_{y,z}(y, v) \end{aligned} \tag{B.4}$$

In multiway splits, where we have one branch for each value of the attribute, we have one outcome $x_i$ for each value. $C_{y,x}$ is not like the figure 4.10, but instead it is equal with the attribute statistics $C_{y,z}$. As a consequence, we have an bijective relation between $x$ and $z$ and we can safely define $x_i = v_i$. Thus, $C_{y,x}(y, x) = C_{y,z}(y, x) = C_{y,z}(y, v)$. We have:

$$\begin{aligned} C_x(x) &= \sum_y C_{y,x}(y, x) \\ &= \sum_y C_{y,z}(y, x) \\ &= \sum_y C_{y,z}(y, v) \\ &= C_z(v) \end{aligned} \tag{B.5}$$

When evaluating multiway split suggestions we have:

$$\begin{aligned} H(Y \mid X)\, C &= \sum_{x \in X} \underbrace{\left( C_x \log_2 C_x - \sum_{y \in Y} C_{y,x} \log_2 C_{y,x} \right)}_{S_x} \\ &= \sum_{x \in X} S_x, \end{aligned} \tag{B.6}$$

where:

$$S_x = C_x \log_2 C_x - \sum_{y \in Y} C_{y,x} \log_2 C_{y,x}. \tag{B.7}$$

**Optimization 5.** *We can extend the evaluation of the binary split suggestions to include the multiway split suggestion with the extra cost of a sum.*

*Proof.* From equation (B.7), with the aid of equations (B.5) and (B.4):

$$\begin{aligned} S_x &= C_x \log_2 C_x - \sum_{y \in Y} C_{y,x} \log_2 C_{y,x} \\ &= C_z(v) \log_2 C_z(v) - \sum_{y \in Y} C_{y,z}(y,v) \log_2 C_{y,z}(y,v) \\ &= B_v \end{aligned}$$

■

As consequence, for hardware implementations we need an extra accumulator to compute and store the sum $\sum_{x \in X} S_x = \sum_v B_v$ and some logic to multiplex the result with the binary split suggestions before sending them to the comparator to find the best suggestion.

# Appendix C

# Measurements in Detail

## C.1 Time Tables

| Leaves | MOA | C | C-Opt | Convey |
|---|---|---|---|---|
| 2 | 4.67690 | 3.23600 | 0.27100 | 0.02800 |
| 4 | 9.27975 | 6.46700 | 0.53300 | 0.05600 |
| 8 | 18.46013 | 12.90000 | 1.05000 | 0.11200 |
| 16 | 36.93447 | 25.78700 | 2.08100 | 0.22300 |
| 32 | 73.98996 | 51.69500 | 4.29900 | 0.44700 |
| 64 | 147.68943 | 103.40600 | 8.52400 | 0.89400 |
| 128 | 295.26865 | 206.51401 | 16.80500 | 1.78800 |
| 256 | 591.77270 | 413.08301 | 33.90300 | 3.57700 |
| 512 | 1183.74864 | 826.54401 | 68.11500 | 7.15500 |
| 1024 | 2363.25353 | 1652.02295 | 135.41400 | 14.30900 |

Table C.1: Time to process multiple leaves (1 instance per leaf, 1024 attributes, 8 classes, 256 values per attribute).

| Attributes | MOA | C | C-Opt | Convey |
|---|---|---|---|---|
| 2 | 4.66855 | 3.22900 | 0.26800 | 0.09000 |
| 4 | 9.30391 | 6.45700 | 0.53400 | 0.11700 |
| 8 | 18.54020 | 12.92100 | 1.06600 | 0.16900 |
| 16 | 36.98053 | 25.82500 | 2.12800 | 0.28600 |
| 32 | 73.84332 | 51.64100 | 4.23900 | 0.51000 |
| 64 | 147.89303 | 103.34700 | 8.46700 | 0.95700 |
| 128 | 295.54468 | 206.54900 | 16.93200 | 1.85400 |
| 256 | 590.95062 | 413.15701 | 33.96800 | 3.62500 |
| 512 | 1182.54776 | 825.97699 | 67.68700 | 7.19100 |
| 1024 | 2363.10960 | 1651.90503 | 135.50400 | 14.31000 |

Table C.2: Time to process multiple leaves for different number of attributes (1 instance per leaf, 1024 leaves, 8 classes, 256 values per attribute).

| Classes | MOA | C | C-Opt | Convey |
|---|---|---|---|---|
| 2 | 638.11318 | 429.28201 | 53.49800 | 3.61200 |
| 3 | 924.74295 | 633.24597 | 67.87500 | 5.39800 |
| 4 | 1211.45568 | 838.03302 | 81.75000 | 7.17500 |
| 5 | 1494.96879 | 1040.40100 | 95.39100 | 8.96100 |
| 6 | 1785.15165 | 1244.40796 | 108.91500 | 10.74600 |
| 7 | 2074.16439 | 1449.35706 | 122.52200 | 12.52800 |
| 8 | 2361.91050 | 1652.38098 | 135.75200 | 14.30900 |

Table C.3: Time to process multiple leaves for different number of classes (1 instance per leaf, 1024 leaves, 1024 attributes, 256 values per attribute).

| Values | MOA | C | C-Opt | Convey |
|---|---|---|---|---|
| 16 | 29.79839 | 12.50300 | 9.29700 | 0.92500 |
| 32 | 79.02875 | 35.76600 | 18.29100 | 1.82700 |
| 48 | 141.89211 | 68.87300 | 27.07700 | 2.71800 |
| 64 | 219.62047 | 116.20600 | 35.85400 | 3.60800 |
| 80 | 309.18411 | 175.78799 | 44.37000 | 4.50000 |
| 96 | 415.64881 | 247.99100 | 53.12200 | 5.39400 |
| 112 | 536.78852 | 332.24600 | 61.34700 | 6.28600 |
| 128 | 673.72857 | 429.12601 | 69.75800 | 7.17400 |
| 144 | 824.15234 | 538.36200 | 78.13000 | 8.06900 |
| 160 | 1556.51527 | 660.13800 | 86.41400 | 8.96000 |
| 176 | 1854.79646 | 794.24103 | 95.00300 | 9.85100 |
| 192 | 1400.65963 | 940.59100 | 105.87900 | 10.74300 |
| 208 | 1615.52049 | 1099.98303 | 111.10000 | 11.63500 |
| 224 | 1848.98722 | 1271.00903 | 119.29900 | 12.53100 |
| 240 | 2096.33919 | 1454.57605 | 127.13000 | 13.41900 |
| 256 | 2361.44315 | 1653.39795 | 135.54500 | 14.31000 |

Table C.4: Time to process multiple leaves for different number of values per attribute (1 instance per leaf, 1024 leaves, 1024 attributes, 8 classes).

| Instances | MOA | C | C-Opt | Convey |
|---|---|---|---|---|
| 1 | 2362.97992 | 1652.68298 | 135.69400 | 14.31000 |
| 2 | 2370.68491 | 1653.52600 | 135.45900 | 14.31200 |
| 4 | 2378.03762 | 1652.03296 | 135.41499 | 14.32700 |
| 8 | 2397.51143 | 1653.33398 | 135.67400 | 14.33200 |
| 16 | 2419.31607 | 1652.64404 | 135.70900 | 14.35700 |
| 32 | 2426.74096 | 1653.70703 | 135.81799 | 14.38800 |
| 64 | 2426.58614 | 1654.75501 | 136.52200 | 14.47400 |
| 128 | 2432.93303 | 1654.81897 | 137.89700 | 14.61900 |
| 256 | 2442.50410 | 1652.75903 | 136.03200 | 14.93800 |
| 512 | 2452.99948 | 1652.89099 | 135.08600 | 15.60200 |
| 1024 | 2521.25308 | 1653.19898 | 135.25301 | 17.06400 |

Table C.5: Time to process multiple leaves and instances. The number of instances defines the size of the instance packets (1024 leaves, 1024 attributes, 8 classes, 256 values per attribute).

## C.2 Speedup Tables

## C.3 Time Tables

| Leaves | C | C-Opt | Convey |
|---|---|---|---|
| 2 | 1.45 | 17.26 | 167.03 |
| 4 | 1.43 | 17.41 | 165.71 |
| 8 | 1.43 | 17.58 | 164.82 |
| 16 | 1.43 | 17.75 | 165.63 |
| 32 | 1.43 | 17.21 | 165.53 |
| 64 | 1.43 | 17.33 | 165.20 |
| 128 | 1.43 | 17.57 | 165.14 |
| 256 | 1.43 | 17.45 | 165.44 |
| 512 | 1.43 | 17.38 | 165.44 |
| 1024 | 1.43 | 17.45 | 165.16 |

Table C.6: Speedup against MOA when processing multiple leaves (1 instance per leaf, 1024 attributes, 8 classes, 256 values per attribute.

| Attributes | C | C-Opt | Convey |
|---|---|---|---|
| 2 | 1.45 | 17.42 | 51.87 |
| 4 | 1.44 | 17.42 | 79.52 |
| 8 | 1.43 | 17.39 | 109.71 |
| 16 | 1.43 | 17.38 | 129.30 |
| 32 | 1.43 | 17.42 | 144.79 |
| 64 | 1.43 | 17.47 | 154.54 |
| 128 | 1.43 | 17.45 | 159.41 |
| 256 | 1.43 | 17.40 | 163.02 |
| 512 | 1.43 | 17.47 | 164.45 |
| 1024 | 1.43 | 17.44 | 165.14 |

Table C.7: Speedup against MOA when processing multiple leaves for different number of attributes (1 instance per leaf, 1024 leaves, 8 classes, 256 values per attribute).

| Classes | C | C-Opt | Convey |
|---|---|---|---|
| 2 | 1.49 | 11.93 | 176.66 |
| 3 | 1.46 | 13.62 | 171.31 |
| 4 | 1.45 | 14.82 | 168.84 |
| 5 | 1.44 | 15.67 | 166.83 |
| 6 | 1.43 | 16.39 | 166.12 |
| 7 | 1.43 | 16.93 | 165.56 |
| 8 | 1.43 | 17.40 | 165.06 |

Table C.8: Speedup against MOA when processing multiple leaves for different number of classes (1 instance per leaf, 1024 leaves, 1024 attributes, 256 values per attribute).

| Values | C | C-Opt | Convey |
|---|---|---|---|
| 16 | 2.38 | 3.21 | 32.21 |
| 32 | 2.21 | 4.32 | 43.26 |
| 48 | 2.06 | 5.24 | 52.20 |
| 64 | 1.89 | 6.13 | 60.87 |
| 80 | 1.76 | 6.97 | 68.71 |
| 96 | 1.68 | 7.82 | 77.06 |
| 112 | 1.62 | 8.75 | 85.39 |
| 128 | 1.57 | 9.66 | 93.91 |
| 144 | 1.53 | 10.55 | 102.14 |
| 160 | 2.36 | 18.01 | 173.72 |
| 176 | 2.34 | 19.52 | 188.29 |
| 192 | 1.49 | 13.23 | 130.38 |
| 208 | 1.47 | 14.54 | 138.85 |
| 224 | 1.45 | 15.50 | 147.55 |
| 240 | 1.44 | 16.49 | 156.22 |
| 256 | 1.43 | 17.42 | 165.02 |

Table C.9: Speedup against MOA when processing multiple leaves for different number of values per attribute (1 instance per leaf, 1024 leaves, 1024 attributes, 8 classes).

| Instances | C | C-Opt | Convey |
|---|---|---|---|
| 1 | 1.43 | 17.41 | 165.13 |
| 2 | 1.43 | 17.50 | 165.64 |
| 4 | 1.44 | 17.56 | 165.98 |
| 8 | 1.45 | 17.67 | 167.28 |
| 16 | 1.46 | 17.83 | 168.51 |
| 32 | 1.47 | 17.87 | 168.66 |
| 64 | 1.47 | 17.77 | 167.65 |
| 128 | 1.47 | 17.64 | 166.42 |
| 256 | 1.48 | 17.96 | 163.51 |
| 512 | 1.48 | 18.16 | 157.22 |
| 1024 | 1.53 | 18.64 | 147.75 |

Table C.10: Speedup against MOA when processing multiple leaves and instances. The number of instances defines the size of the instance packets (1024 leaves, 1024 attributes, 8 classes, 256 values per attribute).

# Bibliography

(1) 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*, ser. KDD '00. New York, NY, USA: ACM, 2000, pp. 71–80.

(2) R. Kirkby, "Improving hoeffding trees," Ph.D. dissertation, University of Waikato, 2007.

(3) A. Bifet, G. Holmes, R. Kirby, and B. Pfahringer, "Data stream mining: A practical approach," May 2011, university of Waikato.

(4) J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

(5) L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *CART: Classification and Regression Trees*. Belmont, California: Wadsworth Publishing Company, 1983.

(6) M. Mehta, R. Agrawal, and J. Rissanen, "SLIQ: A fast scalable classifier for data mining," in *Proceedings of the 5th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '96. London, UK, UK: Springer-Verlag, 1996, pp. 18–32.

(7) J. C. Shafer, R. Agrawal, and M. Mehta, "SPRINT: A scalable parallel classifier for data mining," in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 544–555.

(8) W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, pp. 13–30, Mar. 1963.

(9) L. Rutkowski, L. Pietruczuk, P. Duda, and M. Jaworski, "Decision trees for mining data streams based on the mcdiarmid's bound," *IEEE Trans. on Knowl. and Data Eng.*, vol. 25, no. 6, pp. 1272–1279, Jun. 2013.

(10) L. Rutkowski, M. Jaworski, L. Pietruczuk, and P. Duda, "Decision trees for mining data streams based on the gaussian approximation," *IEEE Trans. on Knowl. and Data Eng.*, vol. 26, no. 1, pp. 108–119, Jan. 2014.

[11] ——, "The cart decision tree for mining data streams," *Information Sciences*, vol. 266, pp. 1–15, May 2014.

[12] G. Hulten and P. Domingos, "VFML – a toolkit for mining high-speed time-changing data streams," 2003. [Online]. Available: http://www.cs.washington.edu/dm/vfml/

[13] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl, "MOA: Massive online analysis, a framework for stream classification and clustering," in *Journal of Machine Learning Research (JMLR) Workshop and Conference Proceedings*, vol. 11. JMLR.org, 2010, pp. 44–50.

[14] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[15] G. De Francisci Morales, "Samoa: A platform for mining big data streams," in *Proceedings of the 22Nd International Conference on World Wide Web Companion*, ser. WWW '13 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2013, pp. 777–778.

[16] A. Murdopo, A. Severien, G. De Francisci Morales, and A. Bifet, *SAMOA Developer's Guide*, Yahoo Labs Barcelona.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ser. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177.

[18] "Storm: Distributed and fault-tolerant realtime computation system." [Online]. Available: https://storm.incubator.apache.org/

[19] A. Srivastava, E.-H. Han, V. Singh, and V. Kumar, "Parallel formulations of decision-tree classification algorithms," in *Parallel Processing, 1998. Proceedings. 1998 International Conference on*, Aug 1998, pp. 237–244.

[20] N. Amado, J. Gama, and F. M. A. Silva, "Parallel implementation of decision tree learning algorithms," in *Proceedings of the10th Portuguese Conference on Artificial Intelligence on Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving*, ser. EPIA '01. London, UK, UK: Springer-Verlag, 2001, pp. 6–13.

[21] G. Chrysos, P. Dagritzikos, I. Papaefstathiou, and A. Dollas, "HC-CART: A parallel system implementation of data mining Classification and Regression Tree (CART) algorithm on a multi-fpga system," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 47:1–47:25, Jan. 2013.

[22] Convey Computer Corporation. [Online]. Available: http://www.conveycomputer.com/

[23] "The r project for statistical computing." [Online]. Available: http://www.r-project.org/

[24] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "Interactive presentation: An fpga implementation of decision tree classification," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '07. San Jose, CA, USA: EDA Consortium, 2007, pp. 189–194.

[25] D. Tong, L. Sun, K. Matam, and V. Prasanna, "High throughput and programmable online traffic classifier on fpga," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 255–264.

[26] S. Kestur, D. Dantara, and V. Narayanan, "Sharc: A streaming model for fpga accelerators and its application to saliency," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[27] Convey Computer Corporation, "Convey hc-2 computer - architectural overview," 2012. [Online]. Available: http://www.conveycomputer.com/index.php/download_file/view/143/142/

[28] ——, "Convey reference manual (version 1.1)," May 2012.

[29] ——, "Convey personality development kit - reference manual (version 5.2)," Apr. 2012.

[30] ——, "Convey programmers guide (version 2.0)," Jun. 2012.

[31] Xilinx Inc., "Virtex-6 family overview (v2.4)," Jan. 2012. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[32] ——, "Virtex-5 family overview (v5.0)," Feb. 2009. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf