Stretching the edges of SVM traffic classification with FPGA acceleration

Tristan Groléat, Matthieu Arzel, Sandrine Vaton

***Abstract*—Analyzing the composition of Internet traffic has many applications nowadays, like tracking bandwidth-consuming applications or QoS-based traffic engineering. Even though many classification methods, such as Support Vector Machines (SVMs) have demonstrated their accuracy, the ever-increasing data rates encountered in networks are higher than existing implementations can support. As SVM has been proven to provide a high level of accuracy, and is challenging to implement at high speeds, we consider in this paper the design of a real-time SVM traffic classifier at hundreds of Gb/s to allow online detection of categories of applications. We show the limits of software implementation and offer a solution based on the massive parallelism and low-level network interface access of FPGA boards. We also improve this solution by testing algorithmic changes that dramatically simplify hardware implementation. We then find theoretical supported bit rates up to 473 Gb/s for the most challenging trace on a Virtex 5 FPGA, and confirm them through experimental performance results on a Combov2 board with a 10 Gb/s interface.**

***Index Terms*—Network and systems monitoring and measurements, Design and simulation, Machine learning**

I. Introduction

Traffic classification associates network traffic with the application or category of applications that generated this traffic. Operators can benefit from knowing which application packets belong to in order to better engineer the traffic, charge their customers, lawfully intercept illegal traffic, etc. Operators already continuously track the composition of traffic per application category, analyzing trends and tracking the emergence of new bandwidth-consuming applications. Quality of Service (QoS) solutions segregate traffic into classes to provide different service guarantees to different kinds of traffic. These solutions also require the ability to associate traffic with applications, since the applications do not tag their traffic by themselves. Traffic classification is also useful to charge customers differently depending on how they use their connection, and for Service Level Agreement (SLA) verification. In many countries, lawful interception of illegal traffic also makes it mandatory for Internet Service Providers (ISPs) to analyze their customers' traffic and recognize illegal traffic. Although lawful interception requires the detection of very specific packets, traffic classification can be used to keep only interesting applications to ease detection.

The task of classifying traffic is challenging for several reasons. First, operators receive and transmit a huge amount of traffic at a high bit rate, making it difficult to analyze, as reported by the annual CISCO report [1]. Second, traditional techniques for traffic classification have some limitations [2]. The most widespread methods to classify traffic are the analysis of port numbers and pattern-matching. Port-based classification has become less and less reliable, since many applications change their port number dynamically or hide themselves behind well-known ports belonging to other applications (especially port 80 for HTTP, which is reliably open in most networks). Pattern-matching techniques look for specific character strings directly in the packet payload. These techniques have more trouble detecting applications that cipher their traffic. They are also under distress when a large number of signatures have to be recognized in high bit rate traffic. This is why they are mostly used on subsampled traffic with a lower bit rate.

This paper focuses on building a high-speed traffic classifier using only the size of some packets. This way, it is able to classify ciphered traffic as well as normal traffic. Classic approaches involve using probes that centralize statistics about traffic in databases, and then applying some algorithms to get useful data about the traffic. In contrast our approach involves building an active probe directly in the network, able to analyze data in real time and act on the traffic immediately. Packets are received and parsed, data is gathered and analyzed, and then packets are tagged with a label indicating the category of the application which generated the packet. All of this is done without slowing down the traffic on a high-speed link. The goal is to support hundreds of gigabits per second. To reach this goal, we use the high degree of parallelism and low-level access to network interfaces offered by Field Programmable Gate Array (FPGA) boards. This active tool can be used as a brick in a Software-Defined Network (SDN) to then apply different Quality of Service (QoS) levels to the various categories of applications.

There has been a large body of literature on lightweight techniques for traffic classification that do not rely on port numbers or pattern-matching [3]–[7], which is evidence of significant academic interest in this topic. A few surveys are available, for example [8]–[10]. This paper does not focus on designing a new algorithm, but rather on the architecture and implementation of a lightweight traffic classifier using SVM, which is recognized as one of the best performing traffic classification algorithms [8], [11]. The classification accuracy (i.e. percentage of flows which are correctly associated with the generating application) of different classification techniques has been deeply investigated. But despite the plethora of literature about traffic classification, the question of how these methods must be implemented *in practice* to enable online

T. Groléat and S. Vaton are in the Computer Science department at Télécom Bretagne, Brest, France. E-mails: {tristan.groleat, sandrine.vaton}@telecom-bretagne.eu.

M. Arzel is in the Electronic Engineering department at Télécom Bretagne, Brest, France. E-mail: matthieu.arzel@telecom-bretagne.eu.

traffic analysis has received less attention. The classification phase is the one that must be accelerated for fast traffic classification. The learning phase is carried out offline, so its acceleration is outside the scope of this article.

We consider a hardware-accelerated implementation of the detection phase of the SVM algorithm. SVM might be considered too complex for high-speed processing when compared to other algorithms, yet it will be shown to be possible and attractive in terms of accuracy. As the rate of traffic can easily reach hundreds of Gb/s in access and core networks, one needs to accelerate the detection algorithm to perform real-time classification. So we focus on the speed of detection. We improve our previous implementation [12] of an SVM classifier on FPGA, modifying the algorithm to boost performance and adapt it to high-rate capture points. To validate our contributions, we consider a new dataset generated in a real network and conduct extensive experiments on an actual board. Our main contributions are the validation of the use of SVM for highly accurate online traffic classification at high speed thanks to hardware acceleration, a working implementation of the classifier on a board tested on real traces, and a generic hardware-accelerated SVM implementation that may be used for other classification problems.

The remainder of the paper is organized as follows. Section II lists existing algorithms for traffic classification and their implementations. Section III provides a description of the SVM algorithm, its accuracy, and how we will use it. Section IV-A describes the requirements on the traffic classifier to support high bit rates and shows that a software implementation is not enough. The remainder of Section IV studies the SVM classification algorithm and presents its challenges and potential. The hardware architecture and implementation of the classifier are detailed in Section V, along with a variation of the classification algorithm, which is more adapted to hardware. Finally, Section VI presents both theoretical and real stress-test results using a 10 Gb/s traffic generator.

II. Traffic classification

A large number of alternatives exist for traffic classification. We will first list different algorithm specificities and then study how they have been accelerated in the litterature.

A. Algorithms

Traffic classification algorithms can be separated into different types [13]: some classifiers are simply based on well-known UDP or TCP ports. Pattern-matching classifiers are based on rules checked against the content of each packet. Lightweight classifiers work on simple features extracted from packet headers, like packet sizes or inter-packet delays. Active classifiers are based on users tagging their traffic themselves.

Lightweight classifiers work on flows. The word “flow” is defined throughout this article as a set of UDP or TCP packets sharing the same flow identifier (source and destination IP addresses and ports and transport protocol), and sent within a short delay. A flow is considered to have expired after no packet with the same flow identifier has been received for a configurable duration. If a new packet with the same flow identifier is received after the flow has expired, it belongs to a new flow. Specific packets used in TCP to signal the start or end of a flow are ignored, and only the expiration delay is used to separate flows. This delay is measured in number of new flows received. A flow expires after one million new flows have been received. Flows are unidirectional; packets sent from A to B belong to a different flow than packets sent from B to A. This is important because packets do not always use the same path in both directions in core networks.

Techniques based only on well-known ports are now inefficient due to protocols using random ports and web-based applications. Pattern-matching techniques still give very good results, but are very limited when classifying encrypted traffic. This is a problem, because big Internet actors like Google and Facebook now use encrypted connections for most of their applications. Active classifiers require the network operator to trust its users, which is generally not the case. This is why lightweight efficient techniques are a research focus.

Different algorithms have been used for lightweight traffic classification. For example Support Vector Machines (SVMs) give good results [6], [14]. A binary tree-based algorithm called C4.5 seems to provide slightly better accuracy than SVM [10], but we will show in Section III-C that it depends on the features used to describe each flow.

To build classifiers that can be deployed on real networks, it is now important to be ready to support traffic with data rates up to hundreds of Gb/s, corresponding to tens of millions of packets per second and tens of thousands of flows per second. This is why studies have been carried out focusing not on increasing the accuracy of the classification, but rather on the supported speed.

For pattern-matching classifiers, using optimized automata for regular expression matching [15] accelerates the algorithm. Another method is to reduce the number of packets in a flow on which the matching is done [16]. This optimization is a kind of subsampling, and it can be done at packet or flow level. This technique reduces the load for all algorithms, but it also causes a drop in accuracy. The impact of subsampling on lightweight classification [17] can be significant if it makes the considered flow features inaccurate.

For lightweight classifiers, one way to reduce the amount of computation required is feature selection [11]; use fewer features to classify a flow, while maintaining accuracy. Other approaches have tried to improve performance with efficient software implementations. In [18], the authors study the performance of a software version of the SVM algorithm and optimize it in order to deal with links up to 1 Gb/s. In [19] performance is improved by offloading a part of the packet processing to a programmable hardware board. But the classification is still done using software, which limits the number of flows that can be classified each second; using six threads and an SVM model with only two classes and a small number of support vectors, they classify 73.2% of the packets of a trace replayed at 384 000 flows per second. We will show that our solution brings better performance with more classes and more support vectors.

Other work uses a light algorithm called C4.5 [10]: a classification involves making a succession of comparisons to browse a tree. This approach allows more efficient implementations than SVM, but we will show that depending on the selected features, SVM can be more accurate than C4.5.

Although new software optimizations and the evolution of commodity computers keep making pure software implementations more efficient, using more specialized hardware makes it possible to support much higher data rates.

B. Acceleration techniques

To accelerate traffic analysis, the most affordable technique is to optimize the use of commodity hardware. One important limit of software implementation is packet reception, as the simple task of receiving incoming packets is challenging in software at high speeds. [20] uses PacketShader, an improved packet capture engine able to support speeds up to 20 Gb/s with two Network Interface Cards (NICs). Interesting work has been done to process traffic using high-performance NICs. In [21], the authors propose a capture engine called PFQ, which replaces the Linux network stack to manage traffic up to 10 Gb/s using a specific Intel NIC. Another advantage of this technique is that it makes it possible to handle traffic using processes on different cores, which provides parallelism.

To accelerate processing, Graphics Processing Units (GPUs) are readily available tools. They are used in [22] to implement pattern-matching and in [23] for intrusion detection. A crucial property of the GPU is that it has a higher level of parallelism and wider memory bandwidth than the Central Processing unit (CPU). Many algorithms already have implementations on GPU, like SVM [24]. This is one possible way to accelerate traffic classification, but a GPU is optimally used by sending a large set of independent data so that the same operations can be done on all data in parallel. For traffic classification, it would be necessary to batch the packets to get enough independent data, which is not acceptable for a classifier that is supposed to tag packets in real time: the latency would slow down the network. And packets still go through the CPU, so the limits of the CPU to process traffic would still be a problem.

To avoid the limitations of commodity hardware, this article explores the use of FPGA boards. FPGAs are integrated circuits that can be configured using a hardware description language, which involves connecting logic gates (and, or, add, etc.) and memory points (registers) to perform any computation. This approach enables massive parallelism and precise control of the algorithm at each clock cycle. Boards integrating an FPGA with high-speed network interfaces are particularly well-adapted. For example, the NetFPGA 10G [25] board has four 10 Gb/s interfaces, and the Combo LXT board [26] has two 10 Gb/s interfaces. These boards will allow us to process traffic up to 40 Gb/s, and to test the scalability of the implementation to handle traffic of hundreds of gigabits per second. Indeed, a Combo board with an interface at 100 Gb/s is now available.

FPGA-based implementations of traffic classification have been studied for pattern-matching [27], because it is a task requiring many independent computations that can be done in parallel on FPGAs. It brings speed improvements but does not suppress the drawbacks of pattern-matching.

FPGAs have also been used with lightweight classification techniques, like a classifier focused on multimedia traffic [28] which uses the k-Nearest Neighbors classification algorithm. The maximum speed is 250 million packets per second, which is a huge improvement compared to software implementation.

Generic implementations of the SVM classification algorithm exist on FPGA. For example, [29] presents an accelerator made to work with a computer. It provides a speed-up factor of about 500 compared to a pure software implementation. While this is a very good generic hardware implementation of SVM, it does not focus on the requirements of traffic classification. It is limited to two classes and implements the exact same SVM algorithm as on a computer. We will show later that the algorithm can be simplified without losing accuracy for traffic classification, which makes a simpler implementation possible.

Decision tree algorithms like C4.5 are also very easy to implement on FPGA. They are used for routing up to 169 million packets per second [30] or for more generic classification like [31], which is based on C4.5 and claims to be able to classify up to 550 Gb/s of traffic. One difference with our contribution is that they use both packet sizes and ports as flow features, while we only use packet sizes. We also use SVM instead of C4.5. Both algorithms have comparable accuracy results, but Section III-C shows that SVM gives better results with the features we use. Although decision trees are adapted for fast classification and SVM requires more computations to classify a flow, we will show that an implementation of SVM on FPGA can be very fast.

III. Using SVM for traffic classification

We will now explain why we have chosen the SVM algorithm, and how we apply it to traffic classification.

A. Proposed solution

We are studying the implementation of a lightweight classification algorithm with a high-performance, hardware-accelerated solution. We take as a base algorithm the well-known SVM [32]. This algorithm is regarded as a good algorithm for traffic classification [8], [11] and has been adopted by several authors to determine the applications that generate peer-to-peer traffic [7], or the kind of applications that generate TCP traffic [6], or even TCP and UDP traffic [14], [18]. Most applications use the length of packets as features for the classification, but some use different features, like the number of packets exchanged between two hosts [7].

Classification is performed on flows, like in most other works using SVM [6], [14]. This means that a flow builder receives packets and groups them into flows, and the classifier is triggered only once enough data has been gathered about a flow. Each flow is described by simple packet-level features, in this case the size of the Ethernet payload of packets 3, 4 and 5 in the unidirectional flow [14] [6]. The first two packets are ignored because they are often very similar for TCP flows, as they correspond to the TCP handshake. This approach works for TCP and UDP flows. If a flow has fewer than five packets, it is never classified. It is not a big issue if the classification is used for QoS, as small flows have little impact on the QoS.

Like any supervised classification method, the SVM algorithm consists of two main phases: a training phase and a detection phase. During the training phase, the algorithm starts from a learning trace labeled with categories of applications, and computes the classification model. Using this model, the detection phase decides the class of new flows.

We consider a hardware-accelerated implementation of the detection phase of the SVM algorithm. The learning phase is done offline in software using the readily available libSVM library [33]. It is not implemented on the FPGA because it is not subject to real-time requirements. In our experience, getting the optimal SVM model from a trace can require up to one day of computation using a powerful machine. But this is not a problem, because the model is only computed once to configure the online classifier. It is necessary to start the learning phase again only when the classifier is installed on a different network or when new applications appear in the traffic. So the acceleration of the learning phase is outside the scope of this article. The code of the hardware-accelerated classifier is open-source and available online [34].

Section III-B gives some background on the SVM algorithm, and Section III-C shows that the algorithm is interesting for traffic classification in terms of accuracy.

B. Background on Support Vector Machine (SVM)

SVM [32] is a supervised classification algorithm. It transforms a non linear classification problem into a linear one, using a so-called "kernel trick". Taking a set of sample points in a multi-dimensional space, each sample point being associated beforehand with a class, SVM tries to find hyperplanes which separate the points of each class without leaving any point in the wrong class. But it is often impossible to separate sample points from different classes by hyperplanes. The idea of SVM is to use a specific function, called a "kernel", to map training points onto a transformed space, where it is possible to find separating hyperplanes. The output of the training phase is the SVM model. It is made up of the parameters of the kernel and a set of support vectors x_i that define the separating hyperplane. During the detection phase, SVM simply classifies new points according to the subspace they belong to, using the SVM model. Among the several algorithms for SVM-based classification, we chose the original and simplest C-Support Vector Classification algorithm [35].

More formally, let us assume that we have a set of training points $x_i \in \mathbb{R}^n, i = 1, \ldots, l$ in two classes and a set of indicator values $y_i \in \{-1, +1\}$ such that $y_i = +1$ if x_i belongs to class 1 and $y_i = -1$ if x_i belongs to class 2. Let ϕ be a function such that $\phi(x_i)$ maps training point x_i onto a higher dimensional space.

During the training phase, a hyperplane is found that separates points $\phi(x_i)$ belonging to classes 1 and 2. The hyperplane is defined by an offset b and a vector $w = \sum_{i=1}^{l} y_i \alpha_i \phi(x_i)$. The $\phi(x_i)$ corresponding to a non-zero α_i are called support vectors, and they are the only useful vectors for classification.

In the detection phase, any new point x is classified by computing its position compared to the hyperplane using $\text{sign}(w^T \phi(x) + b) = \text{sign}(\sum_{i=1}^{l} y_i \alpha_i K(x_i, x) + b)$ with the kernel function defined as $K(x_i, x) = \phi(x_i)^T \phi(x)$. Depending on the sign, x is placed into class 1 or 2.

In this article we first use the Radial Basis Function kernel as a reference, as it was used in [6]:

$$K(x_i, x_j) = \exp(-\gamma \parallel x_i - x_j \parallel^2),\ \gamma > 0 \qquad (1)$$

γ is a parameter that can be changed to get better classification accuracy by "cross-validation" [33], which simply involves testing different values and selecting the one for which the classification gives the highest accuracy.

Section V-C presents another kernel, more adapted to hardware implementation, and demonstrates that it brings better performance and accuracy for traffic classification.

From this simple two-class SVM problem, one can easily deal with multi-class SVM classification situations. We use the so-called "one versus one" (1 vs 1) approach: $\frac{n(n-1)}{2}$ two-class SVM problems are considered, one for each pair of classes. Training and classification are performed for each two-class problem, thus producing $\frac{n(n-1)}{2}$ decisions. The final decision is taken on the basis of a majority vote, that is to say that the new point is allocated to the class which was chosen the highest number of times.

C. Accuracy of the SVM algorithm

For a given dataset, the accuracy of the SVM-based classifier is measured by the percentage of flows with a known ground truth that are placed in the proper class. In order to assess this accuracy, we have performed validation over three different datasets. The learning and detection phases were performed using the libSVM library [33]. The input of the SVM classifier is the one described in Section III-A.

In each dataset, each flow is associated with a label identifying the application that has generated the flow, called the "ground truth". It has been obtained using a combination of pattern-matching, with for example L7-filter [36], and the analysis of logs of system calls on the machines generating traffic. The latter technique requires specific software installed on each machine generating traffic. A tool called GT [5] is able to combine pattern-matching and log analysis to create a ground truth on a real network.

The characteristics of the three traffic traces used as benchmarks are listed in Table I. They correspond to three different scenarios: one laboratory environment, one campus network, and one student residential network. As a consequence, the composition of traffic is significantly different from one trace to the other. However, they all contain only IPv4 traffic over Ethernet. All packets that are not UDP or TCP are ignored.

1) The Ericsson dataset corresponds to traffic that has been generated in a laboratory environment at Ericsson Research. The ground truth has been obtained directly from system logs.
2) The Brescia dataset is public [5] and corresponds to traffic captured on a campus network. The ground truth has been obtained with the GT tool.
3) The ResEl dataset is a trace we captured ourselves on the network for student residence halls at Télécom Bretagne. ResEl is the association managing the network, which

Trace	Network of capture	Ground truth source	Bytes	Flows	Flows with ground truth	Capture rate (kb/s)	Average packet size (B)	Average flow size (kB)
Ericsson	Local area network at an Ericsson laboratory	Logs	6 222 962 636	36 718	16 476	323.56	651	169
Brescia	Campus trace generated at University of Brescia, Italy	Logs + pattern matching (GT)	27 117 421 253	146 890	74 779	1 048.0	812	185
ResEl	Campus trace generated at Télécom Bretagne, France	Pattern matching	6 042 647 054	25 499	10 326	85 898	880	138

TABLE I
TRAFFIC TRACES AND THEIR PROPERTIES

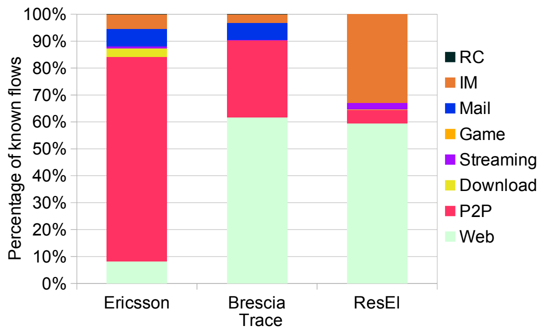

Fig. 1. Percentage of flows in each class according to the ground truth

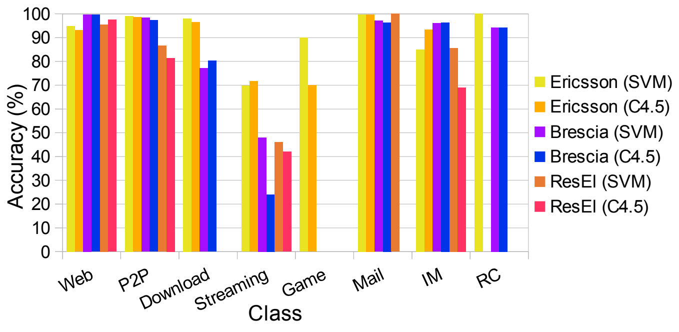

Fig. 2. Accuracy per traffic class using SVM and C4.5

agreed to the capture on the single link between the residence halls and the Internet. It was performed around 1 PM on a workday for a bit more than five minutes with an average data rate of 84 Mb/s. The ground truth was obtained using L7-filter. The GT tool was not used, because it would have required installing probes on the computers of certain students, which was not possible. This makes the ground truth less accurate than using GT, but it is interesting to test the limits of the classifier nonetheless.

As each trace corresponds to a completely different network, the parameters of the SVM algorithm have been set differently, and one different SVM model has been learned for each trace.

The definition of classes is not universal. In order to enable a comparison between traces, we have grouped applications into eight different categories. Other traffic classification studies [3], [18] use a similar number of classes. Unsupervised classifiers produce more classes, but this is not desired. In the repartition shown in Figure 1, IM stands for Instant Messaging and RC stands for Remote Control. Some classes are more present than others in each trace. This makes classification difficult, because few data points are available for the least frequent classes. No games have been detected in the Brescia and ResEl traces.

Using the RBF kernel and the best parameters found in cross-validation, the accuracy, that is to say the overall percentage of flows that are correctly classified is 97.82% for Ericsson, 98.99% for Brescia and 90.60% for ResEl. The accuracy is different for each trace because traces contain different applications, and some applications are easier to identify than others. Accuracy for the ResEl trace is not as good as for the other traces, probably because of the poor quality of the ground truth, simply obtained using pattern-matching. But this represents real-world applications where the ground truth is always difficult to get, so we kept this trace to check if an implementation works on it as well.

Columns labeled "SVM" in Figure 2 provide the accuracy per class, that is to say the percentage of flows of each class that has been accurately classified. The proportion of a class in a trace impacts the ability of the SVM algorithm to detect it. For example, as the "Web" class is present with a good proportion in all three traces, the accuracy of the detection is high. However, the "Streaming" class is almost absent in the three traces, and thus has the worst accuracy.

As discussed in Section II, C4.5 is an algorithm known to give good results for traffic classification, even better than SVM in some situations [10]. So we tried to use C4.5 on the Brescia dataset, using two-thirds of the trace for testing, and a part of the remaining trace for learning. C4.5 always resulted in more classification errors than SVM, with 20% more errors using a third of the trace for learning (16 602 flows), and even 45% more errors using a tenth of the trace (498 flows). This may be due to the fact that we do not use ports as classification features. Figure 2 shows the accuracy of C4.5 for each class. Although C4.5 is slightly more accurate for some classes, it is less accurate for most, especially for classes with few flows.

Article [10] also claims that using discretization makes SVM classification more accurate. Discretization involves grouping the feature values (packet sizes) into intervals. Classification is then done using these intervals. We tried using the entropy-based minimum description length algorithm to discretize the features of the Brescia dataset. We obtained a model with fewer support vectors, which makes classification faster, but with almost three times more classification errors, so we did not use discretization.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2014.2346075, IEEE Transactions on Network and Science Management

IV. SVM classification implementation

In the following, the implementation of online SVM traffic classification is studied.

A. Requirements

In our scenario, probes are located on an operator access or core network to monitor traffic at packet level, reconstruct flows and classify them with an SVM. Only the detection phase of SVM is performed online. The learning phase is performed offline periodically with a ground truth generated by tools such as GT. Using boards such as the NetFPGA 10G [25] or the COMBO LXT [26], algorithms should be able to support up to 40 Gb/s. But the eventual goal for the algorithm is scaling to 100 Gb/s and more.

Two main functions will be required for traffic classification:

- The flow reconstruction reads each packet, identifies which flow it belongs to, and stores the packet lengths required for classification. The processing speed depends on the number of packets per second in the traffic.
- The SVM classification runs the SVM algorithm once for each received flow. The processing speed depends on the number of flows per second in the traffic.

The traces used to test the classification algorithm are described in Table I. To reach a certain bit rate, requirements in terms of packets per second for flow reconstruction and flows per second for classification are different depending on the dataset considered. This is due to different average packet and flow sizes (last columns of Table I). The flow size is the number of bytes received between the reception of two flows with enough data to send them to be classified. It includes possible delays between packets. Average packet and flow sizes make it possible to compute the requirements in terms of flows and packets per second if the trace is replayed at any rate. For example at 10 Gb/s, the ResEl trace would be replayed with an average of $rate/flow\ size = 10 \times 10^6/(138 \times 8) = 9\,058$ flows/second and $rate/packet\ size = 10 \times 10^9/(880 \times 8) = 1\,420\,454$ packets per second. So to support a data rate of only 10 Gb/s, the flow reconstruction speed should range from $1\,420\,454$ (ResEl) to $1\,920\,123$ packets per second (Ericsson) and the SVM classification from $6\,757$ (Brescia) to $9\,058$ flows per second (ResEl). The ResEl trace is easy to handle for flow reconstruction because it contains large packets, but hard to handle for classification because it contains small flows.

We first developed a software version of the classifier that is fed by a trace, to assess the possible performance in software. The classifier is made up of three main modules: (i) reading the trace, (ii) rebuilding flows from the stream of packets, and (iii) classifying flows. Flow reconstruction is handled using the C hash table library, Ut Hash [37]. For the SVM algorithm, the libSVM [33] library (written in C) was chosen. To use all the cores of the processor, openMP [38] for libSVM is enabled.

Table II shows the performance of the software implementation on a machine with two 6-core Xeon X5650 processors at 2.66 GHz with hyper-threading disabled and 12 GB of DDR3 RAM. The flow builder uses only one core, and the classifier uses the eleven remaining cores. Performance values are derived from the time the flow builder and classifier modules work to process the whole trace, divided by the number of packets and flows in the trace. Maximum speeds are computed using the mean values in Table I. It shows that the software implementation is only able to support low classification rates, ranging from $1\,827$ to $5\,975$ flows per second depending on the trace.

Trace	Packets per second (flow reconstruction)	Flows per second (classification)
Ericsson	5 214 218 max. 27 Gb/s	5 975 max. 8.1 Gb/s
Brescia	5 531 895 max. 36 Gb/s	1 827 max. 2.7 Gb/s
ResEl	4 750 621 max. 33 Gb/s	5 619 max. 6.2 Gb/s

TABLE II
Performance of the software implementation

The flow reconstruction speed does not really depend on the trace, as the flow reconstruction algorithm requires a time that is almost constant per packet due to the hash table. The packet capture time is not taken into account here, as traces are processed from a file containing only packet headers.

SVM classification is always more limiting than flow reconstruction (with a maximum speed of 2.7 Gb/s in the worst case). Its speed depends on a variety of factors, including the number of support vectors in each SVM model; Brescia is the trace for which the learned model has the most support vectors ($14\,151$), followed by the ResEl ($5\,055$) and Ericsson ($3\,160$) traces. The number of support vectors represents the complexity of the hyperplane used for classification. The Brescia trace is more difficult to classify accurately than the others. The parameters of the learning phase can be used to alter the trade-off between the number of support vectors and the accuracy. Here accuracy is the priority.

Our software implementation is not able to reach 10 Gb/s, mainly due to its limited ability to parallelize the computation. Although it could be optimized, the limits for packet processing would remain, and even if the classification were made twice as fast, it would still not be enough to support even 10 Gb/s. This justifies the use of hardware acceleration. Different platforms may be used to accelerate network monitoring:

- Graphics processing units can help accelerate the algorithm, but not the packet processing.
- Network processors are programmable in software and provide hardware-accelerated tools for tasks commonly required in network monitoring. But platforms are proprietary, development is not portable, and each commercial solution has very different levels of performance.
- Programmable cards with an integrated Field-Programmable Gate Array (FPGA) are very flexible and provide hardware access to network interfaces. Although development time is long, the code is portable and the results do not depend on a specific vendor.

To be able to fully explore the parallelism possibilities in the SVM classification algorithm, we have chosen to use a board with an FPGA that is more flexible than network processors. Two main vendors provide such boards: (i) NetFPGA with

Algorithm 1 SVM classification algorithm

```
1: $x \leftarrow$ the vector to classify
2: **for all** support vector $x_i$ **do** {Main loop}
3:     $c_i \leftarrow$ the class of $x_i$
4:     $k_i \leftarrow K(x_i, x)$
5:     **for all** class $c_j \neq c_i$ **do** {Sum loop}
6:         $d \leftarrow$ index of the decision between $c_i$ and $c_j$
7:         $S_d \leftarrow S_d + y_{d,i} \times \alpha_{d,i} \times k_i$
8:     **end for**
9: **end for**
10: **for all** decision $d$ between $c_i$ and $c_j$ **do** {Comp. loop}
11:     **if** $S_d - b_d > 0$ **then**
12:         Votes $V_i \leftarrow V_i + 1$
13:     **else**
14:         Votes $V_j \leftarrow V_j + 1$
15:     **end if**
16: **end for**
17: Select class $c_n \leftarrow$ class with the highest votes $V_n$
```

the NetFPGA 10G, which has four interfaces at 10 Gb/s and a Xilinx Virtex-5 XC5VTX240T FPGA, and (ii) INVEA TECH with the Combo, which has two to four interfaces at 10 Gb/s and a Xilinx Virtex-5 XC5VLX155T FPGA. The FPGA provided by the NetFPGA 10G performs better than that on the Combov2 board, so we will perform synthesis on the NetFPGA 10G FPGA. But for now the only 10 Gb/s board we have is a Combo, so stress tests will be made on the COMBO-LXT board with a COMBOI-10G2 extension (two interfaces at 10 Gb/s). This is enough to test the implementation, which will be easy to port later to more powerful boards like the Xilinx VC709 board, offering almost three times the processing power of the NetFPGA 10G.

We did implement flow reconstruction on FPGA, but it is not the focus of this article. We use the same principles as [39] which considers an FPGA implementation of NetFlow. For classification purposes, SVM implementations on FPGA have also been proposed, but they are either focused on the learning phase [40] or not adapted to our classification algorithm [41], as they are restricted to two-class problems or using different kernels. We will therefore now focus on the implementation of the SVM detection phase for traffic classification on an FPGA.

B. The SVM classification algorithm

The classification part of the SVM algorithm takes a vector as input and returns the class of that vector as an output. The main part of the computation is repeated for each support vector. Algorithm 1 describes the steps of this computation. It is the multi-class implementation of the decision making procedure described in Section III-B. This pseudo-code has been written in order to highlight the possibilities for parallelizing the algorithm.

The support vectors and the y, α and b values are parts of the SVM model. Compared to the notations used in Section III-B, index d is added to identify the binary decision problem considered for the model values.

C. Parallelism

The Main loop is where most of the computation time is spent. It iterates many times (from 3 160 to 14 151 support vectors for the traces presented in Table I) and includes elaborate operations (exponential, multiplications). But it can be easily parallelized, as each iteration is independent of the others. The only shared data is in the additive S values. These values have to be duplicated so that iterations are computed in parallel. Then, as S is additive, the duplicated values can be merged by adding them together.

The Sum and Comparison loops have few iterations: one per class. In this article there are eight classes defined in Figure 2, so the loops can be entirely parallelized. The Sum loop iterations are independent, while the Comparison loop iterations share the vote counter, which is additive. So it can be duplicated and then merged.

All loops can be removed by using parallel processing except the main loop, which has too many iterations and would require more area than is available on the Virtex-5. But it is possible to duplicate the Main loop in hardware, so that fewer iterations are required to process one vector.

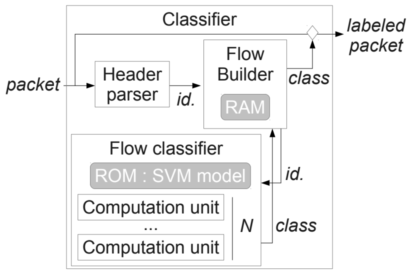

Fig. 3. Architecture of the classifier

V. Adaptation to hardware

We will now see how the SVM implementation can be adapted to get the best performance on FPGA.

A. Architecture

The architecture of a module on a NetFPGA or Combov2 board is similar. It uses two buses, one for input and one for output traffic. The classifier block is described in Figure 3.

The computation units represent the most important part of this architecture; they implement the computation of the main loop described in Algorithm 1. To get the best performance from the FPGA, the operations of the algorithm must be parallelized. As seen in Section IV-C, all loops except the main loop can be totally unrolled by duplicating the hardware for each iteration. The computation unit is duplicated, as much as the FPGA supports.

As the computation in the Main loop is complicated, each iteration will take many clock cycles. To improve the throughput of the loop and reduce its computation time, the iterations can be pipelined; one new support vector is processed by the first operation of the computation unit at each clock cycle, and then forwarded to the next operation. This way all operations work in parallel and each computation unit accepts one support vector at each time step.

As Figure 3 shows, the SVM model is currently stored in Read-Only Memories (ROMs) included on the FPGA. This makes it necessary to configure the FPGA again to change the SVM model used. In future work, ROMs will be converted into RAMs, so that it is possible to change the SVM model faster. This will not require the implementation to be changed as ROMs are simulated on the FPGA as RAMs without write access, but might complicate the placement of the logic on the FPGA, as new connections will have to be created to write into these memories.

B. The RBF kernel

Computation of the RBF kernel function (Equation 1) requires three integer additions (one per vector component), three integer multiplications (to compute the squares), one multiplication by a floating-point constant, and one exponential computation.

Floating-point operations are complex to realize and use too much area on the FPGA. The best solution is to transform the algorithm to use a fixed-point model instead of a floating-point model. This section will present a direct hardware implementation of the classification algorithm using the RBF kernel, and the next section will present an implementation using an adapted kernel, simpler to implement in hardware.

1) Operations: Multiplications are complex to realize in hardware. They could be done on specialized multipliers, but few are available on the chosen FPGA, so parallelism would be limited. They are required to compute the squares in the RBF kernel, but squares are symmetric functions. In the RBF kernel function, they are applied on a difference of two packet sizes. As packet size can vary from 0 to 1518 for Ethernet, the input of squares varies from -1518 to 1518. A ROM with 1519 values is used to emulate squares. On FPGA, this memory is implemented in specialized memory cells (BlockRAM).

To avoid the $y_{d,i} \times \alpha_{d,i} \times k_i$ multiplication, $ln\left(|y_{d,i} \times \alpha_{d,i}|\right)$ (ln is the natural logarithm) is precomputed, and the exponential used to get k_i is executed only after the addition of this term. Delaying the exponential computation transforms the multiplication into an addition. This way only one multiplication by a constant remains in the kernel computation, which is much simpler than a multiplication of two variables. This simplification has a cost, visible in Algorithm 1, as the exponential is moved from the computation of k (line 4) to the computation of S_d (line 7). So instead of computing the exponential once for all classes, it is computed once for each class. As there are eight classes, there are eight times more exponentials to compute with our dataset. These computations are done in parallel, so more time is not required, but it does require more space on the FPGA.

A ROM is also used to emulate the exponential function. As the function $y = e^x$ tends towards 0 when x tends towards $-\infty$ and y is stored on a fixed number of bits, there is a minimum value x_{min} for which $y_{min} = e^{x_{min}} = 0$. The value x_{max} is determined experimentally by observing the input values of the exponential with different traces. With this technique and the quantization parameters described in next paragraph, only $10\,792$ values have to be stored in memory. To reduce this number even more, we use the property of the exponential $e^{a+b} = e^a \times e^b$: instead of storing values from -4 to 4, we store values from 0 to 4 and we store e^{-4}. This way, $e^{0.8}$ is read directly and $e^{-3.2}$ is computed as $e^{0.8} \times e^{-4}$. This technique costs one comparison and one multiplication by a constant, but divides the memory used by two. In the current implementation, the input domain is divided into eight segments, requiring seven comparisons and seven multiplications by a constant, and reducing the memory used to only $1\,349$ values. Multiplications by constants are converted during synthesis into simple shifts and additions.

Variable	Integer part	Decimal part	Signed
Vector component	11	0	×
$\log_2(\gamma)$	3	0	×
$\ln(\alpha)$	4	10	✓
Exponential input	4	10	✓
Exponential output	12	7	×
Square input	12	0	✓
Square output	22	0	×
b	15	11	✓
S	15	11	✓

TABLE III
QUANTIZATION OF THE MAIN SVM FIXED-POINT MODEL VALUES

2) Quantization: Table III shows the bit widths of different variables used in the SVM fixed-point model.

These quantization parameters have been chosen so that the mathematical operations are carried out on values that are as small as possible, while keeping the same classification accuracy. Experiments show that the accuracy can be as good when using the fixed-point model as when using the floating-point model, but accuracy drops very fast if too small a parameter is chosen. Some parameters are quite large because the classifier is supposed to work for any SVM model. The possible values of the variables have been determined by analyzing SVM models learned using subsets of each trace.

The most critical parameters are the exponential and square input and output widths. This is because they change the width of the memories used, which may take a lot of space on the FPGA, decreasing the possibility of parallelizing the design.

Eleven-bit-wide vector components have been chosen because we assume that the size of a packet will not be larger than 1500 bytes. This is the maximum size of standard Ethernet payloads. Jumbo frames are not supported and none are present in the test traces. Adding jumbo frame support would require using wider vector components.

The γ parameter is chosen as a power of 2 to simplify hardware operations. Its value is determined for each trace by testing different values: first learning is applied on different subsets of the trace, then classification is applied on the whole trace. Results are compared to the ground truth and the parameter which gives the best accuracy is kept. With the traces we use, selected values are always higher than 2^{-7} and lower than 1, so the $-\log_2(\gamma)$ value is between 0 and 7.

The width of $\ln(\alpha)$ is chosen to fit the width of the exponential input, which is made as small as possible to decrease the memory used by the exponential. The square output is kept at 22 bits because it is then multiplied by gamma, which transforms up to seven integer bits to decimal bits. The b and S parameters share the same width because

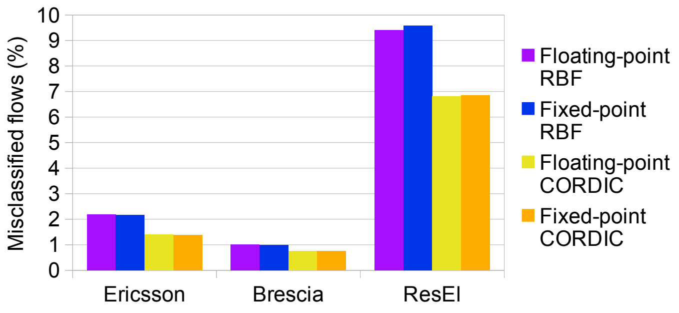

Fig. 4. Errors compared to the ground truth for floating- and fixed-point implementations using the RBF and CORDIC kernels

they are compared with each other.

3) Accuracy: To assess the loss in precision, a software implementation of the classification algorithm with the fixed-point model has been written. Figure 4 displays the percentage of errors compared to the ground truth for the floating- and fixed-point implementations using the RBF kernel. The floating-point results have been described in Section III-C. There are actually fewer errors with fixed-point implementation for most traces. This very small improvement in accuracy is only due to errors which compensate by chance the SVM classification errors. This is not a reliable improvement and changes depending on the trace. But it shows that the transition to fixed-point does not decrease the accuracy of the algorithm. A big percentage of differences is observed between the classification made by the floating- and fixed-point implementations (between 0.32 and 7.6%), but this is mainly on flows with no ground truth, so they cannot be considered errors.

C. The CORDIC algorithm

The RBF kernel is known to be efficient in most situations, and we have seen that it gives good results for traffic classification. But it makes it necessary to compute squares and exponentials, which is done using read-only memories. The routing of these memories increases the critical path on the FPGA, preventing it from working at high frequencies.

D. Anguita *et al.* suggested a different kernel function in [41], that is much more adapted to hardware, because it can be computed using the CORDIC algorithm [42]. This algorithm, described in Section V-C2, is used to approach trigonometric functions using only add and shift operations. These operations are the simplest arithmetical computations that a conventional digital circuit can handle.

Here is the suggested kernel function:

$$K(x_i, x_j) = 2^{-\gamma \|x_i - x_j\|_1} \tag{2}$$

It is very similar to the RBF kernel (Equation 1), except that the exponential has been replaced by a power of 2 and the square function has been replaced by a simple absolute value. The replacement of the exponential by a power of 2 simply corresponds to a scaling of the γ parameter.

1) Floating-point accuracy: The following section explains how using the CORDIC algorithm makes this kernel more adapted to hardware implementation. But before implementing it, the accuracy of the classification with this kernel must be checked. So we first integrated the floating-point version of the kernel in the libSVM library to test the accuracy in software.

Results are shown in Figure 4. Using the floating-point kernel, classification errors represent 1.4% of all known flows for the Ericsson trace, 0.75% for the Brescia trace and 6.8% for the ResEl trace. This is smaller than the 2.2%, 1.0% and 9.5% found with the RBF kernel. So this kernel is even more adapted to traffic classification than the RBF kernel. This is due to the use of an l^1-norm instead of the l^2-norm; it differentiates more vectors with multiple different components, that is to say flows that have more than one of the three packets with different sizes. These flows are more likely to be in different classes than a flow with two packets of the same size, and one packet of a very different size.

2) Algorithm: The CORDIC algorithm is used to compute trigonometric functions using only addition and shift operators. It has also been extended to compute the exponential function, using the relation $e^x = \sinh(x) + \cosh(x)$. For kernel computation, it will be used to get the value of Equation 2, combined with the multiplication by $\alpha_{d,i} \times k_i$ (line 7 of Algorithm 1). To do so, we will compute something of the form $B_0 2^{E_0}$, with input values B_0 and E_0 in $[0; 1[$ to ensure convergence. The way to get to this form from the equations is explained later, in Section V-C3. The CORDIC algorithm works by defining two sequences:

$$E_0, B_0 = \text{initial values} \tag{3a}$$
$$B_n = B_{n-1}\left(1 + D_n 2^{-n}\right) \tag{3b}$$
$$E_n = E_{n-1} - \log_2\left(1 + D_n 2^{-n}\right) \tag{3c}$$

Each D_n is chosen equal to 0 or 1. The goal is to get E_n as near as possible to 0. $l_n = \log_2(1 + 2^{-n})$ is a positive, strictly decreasing sequence. If $l_n > E_{n-1}$, D_n is chosen equal to 0, and $E_n = E_{n-1}$, otherwise $D_n = 1$ and $E_n < E_{n-1}$.

The interesting thing about these equations is that they can be computed using only shift and addition operations once l_n has been pre-computed for each n. Indeed, at each step, $E_n = E_{n-1}$ and $B_n = B_{n-1}$, or $E_n = E_{n-1} - l_n$ and $B_n = B_{n-1} + B_{n-1} \times 2^{-n}$. Multiplying by 2^{-n} is equivalent to a right shift of n bits.

The way D_n is chosen, it is proved [42] that $E_n \to 0$, so that with a big enough N, we have $E_N \approx 0$:

$$E_N = E_0 - \sum_{n=1}^{N-1} \log_2\left(1 + D_n 2^{-n}\right) \approx 0 \tag{4}$$

Using this approximate E_N value, we obtain for B_N:

$$B_N = B_0 \prod_{n=1}^{N-1} (1 + D_n 2^{-n}) \tag{5a}$$
$$= B_0 2^{\sum_{n=1}^{N-1} \log_2\left(1 + D_n 2^{-n}\right)} \approx B_0 2^{E_0} \tag{5b}$$

It is proved in [41] that if $E_0 < 1$ and $B_0 < 1$, in order to get an output accuracy of N bits (in fixed-point

Algorithm 2 CORDIC algorithm to compute $y_{d,i} \times \alpha_{d,i} \times 2^D$

```
1: $E_0 \leftarrow D$
2: $B_0 \leftarrow y_{d,i} \times \alpha_{d,i}$
3: $m \leftarrow 1$
4: for all iteration $n$ from 1 to $N$ (some iterations are duplicated) do {CORDIC loop}
5:     if $E_m \geq l_n$ then
6:         $E_m \leftarrow E_{m-1} - l_n$
7:         $B_m \leftarrow B_{m-1} - (B_{m-1} >> n)$
8:     else
9:         $E_m \leftarrow E_{m-1}$
10:        $B_m \leftarrow B_{m-1}$
11:    end if
12:    $m \leftarrow m + 1$
13: end for
14: The result is $B_m$
```

representation), N iterations of the algorithm are necessary, and some iterations have to be repeated to guarantee the convergence. For example, to guarantee an accuracy of 8 bits, $N = 8$ iterations are necessary and iterations $n = 2$, 4 and 8 have to be duplicated, so a total of 11 iterations have to be implemented. The way to choose which steps should be repeated depending on N is described in [40]. To avoid decreasing accuracy, intermediate E_n and B_n results should be stored on $N + \log_2(N) = 11$ bits in this example.

3) Implementation: An implementation of the CORDIC kernel computation already exists [43], but it is not optimized enough for traffic classification. Indeed, it computes values for each support vector one after the other. We implement a version that is able to start computation on one new support vector at each clock cycle, using the same pipelined architecture as for the RBF kernel.

Compared to Algorithm 1, using the CORDIC kernel changes the computation of k_i (line 4). But the $y_{d,i} \times \alpha_{d,i} \times k_i$ multiplication (line 7) is avoided by integrating $\alpha_{d,i} \times k_i$ into the computation of the power of 2 ($y_{d,i} = +-1$ so it is a simple inversion). So the CORDIC implementation described below changes line 7 of Algorithm 1, integrating the multiplication and the kernel function.

The CORDIC is used to compute $\alpha_{d,i} \times 2^{-\gamma \| x_i - x_j \|_1}$. As explained in Section V-C2, to be sure that the algorithm will converge, its input value E_0 must be in $[0;1[$, so $-\gamma \parallel x_i - x_j \parallel_1$ is separated into a negative integer part I and a decimal part D. Only D, in $[0;1[$, is used as input value E_0. A scaling of the α and b parameters by a fixed parameter is also computed beforehand to ensure that input value $B_0 = \alpha_{d,i}$ is always in $[0;1[$. This scaling by a fixed factor of the two elements of the inequation line 11 of Algorithm 1 does not change the classification results.

Algorithm 2 describes the CORDIC implementation. The loop is computed on the original N iterations, and the repeated steps. n represents the number of the iteration without repetition, m includes the repetitions.

As only the decimal part of $\gamma \parallel x_i - x_j \parallel_1$ has been used, the result must then be multiplied by 2^I, which is equivalent to a left shift of $-I$ bits (I is negative by construction).

4) Quantization: To develop a fixed-point version of this new kernel, the same quantization process is used as for the RBF kernel in Section V-B2, in order to get as good a classification accuracy as with the floating-point model. The most important parameter to set is the number of steps in the CORDIC algorithm. In theory, the number of steps sets the number of bits of precision of the output. But by testing many options and comparing their accuracy, the complexity was reduced and the parameters in Table IV were selected.

Variable	Integer part	Decimal part	Signed
CORDIC output width	16	0	×
l_n	17	0	×
Scaled α	0	16	×
Scaled b	0	17	✓
Scaled S	8	16	✓

TABLE IV
QUANTIZATION OF THE MAIN CORDIC VALUES

The number of steps in the CORDIC is 15, and step 3 is repeated once. It is important to decrease this number of steps as much as possible, because the CORDIC is implemented as a pipeline. This means that each step of the CORDIC is implemented on its own hardware, making it possible to start computing a new kernel value at each clock cycle. But this also means that each step has a cost in FPGA resources used.

Although the S parameter is only used to compare it to the b parameter, its integer part width is much bigger. This is important because S accumulates positive or negative values at each clock cycle, so it can grow and then decrease afterwards. If big values are cut, the final result will be changed.

5) Fixed-point accuracy: Figure 4 shows that, like for the RBF kernel, fixed-point implementation has the same accuracy as floating-point implementation. It is even a bit better on some traces, but this is not significant. A point not visible on this figure is that more differences are observed between the floating-point classification and the fixed-point classification than with the RBF kernel; for Ericsson there are 1.3% of flows classified in different classes instead of 0.92% and for Brescia 1.6% instead of 0.33%. The ResEl dataset is an exception with 0.96% of differences instead of 7.6%. As these flows in different classes are mainly flows with no ground truth, they cannot be qualified as errors. So the accuracy of SVM with the CORDIC kernel is a bit better overall than the accuracy of SVM with the RBF kernel for our traces.

Another unexpected difference is that models with the CORDIC kernel have less support vectors than with the RBF kernel: $1\,745$ (-45%) for the Ericsson trace, $8\,007$ (-43%) for the Brescia trace and $4\,838$ (-4.3%) for the ResEl trace. This effect is useful, as the classification time of one flow depends linearly on the number of support vectors. The decrease is probably due to the use of the l^1-norm by the CORDIC kernel, which is more adapted to traffic classification.

D. Comparing the two kernels

We have seen that in terms of accuracy, the CORDIC and RBF kernels are very similar. In terms of processing speed, the CORDIC models for some traces use almost half as many support vectors as the RBF models. This means that the main loop has to be iterated less. But the processing speed also depends on the implementation complexity of this loop. The next section will allow us to conclude on the fastest kernel.

VI. Performance of the hardware-accelerated traffic classifier

We will now assess the performance of the classifier, first using estimates of the implemented circuit performance, and then measuring actual values on a prototype.

A. Synthesis results

1) Synthesis parameters: The proper behavior of the hardware implementation has been tested by checking that its results are exactly identical to the software implementation of the fixed-point model, first in simulation, and then implemented on a NetFPGA 1G and on a Combo board. So the classification results of the hardware implementation are those in Figure 4. This section focuses on performance in terms of the number of flows classified per second.

To assess the performance of the hardware implementation and compare it with the software implementation, it has been synthesized (code converted into a circuit) on a Virtex-5 XC5VTX240, which is present on NetFPGA 10G boards. Only the SVM flow classifier is synthesized. The header parser and flow builder visible in Figure 3 are not included in the results. This choice has been made to speed up the synthesis to be able to compare results using different parameters. To get exactly the same results with the full design, using a bigger FPGA, like the one from the Xilinx VC709 board, would be necessary. A different model has been tested for all three traces, using the RBF and the CORDIC kernel. The number of processing units has been changed as well to exploit the maximum parallelism on the FPGA while keeping a high working frequency. Table V presents the results of these syntheses using the RBF kernel with 2, 4 or 8 computation units in parallel. Table VI presents the results using the CORDIC kernel with 2, 4 or 16 computation units.

The number of occupied slices, registers and look-up tables (LUTs), as well as the maximum frequency, are given by the synthesis tool. They are an indication of the complexity of the implementation. The number of cycles required per flow has been determined by analyzing the implemented computation. It depends on the number of support vectors in the model and on the number of parallel computation units.

Increasing the number of computation units makes it possible to classify a flow using fewer clock cycles, but it requires more resources on the FPGA. If the design contains too many computation units, it does not fit on the FPGA. So the synthesis fails or, if it succeeds, the resulting working frequency is very low because routing the logic on the FPGA is too complicated. This is why results for the RBF kernel use only up to 8 computation units while up to 16 computation units are used for the CORDIC kernel; the RBF kernel uses more space on the FPGA, so the working frequencies with 16 computation units are too low to be usable.

2) Kernel comparison: Thanks to massive parallelism, hardware implementations all have better performance in terms of flows per second than software implementations. It can be observed that the RBF kernel uses more area on the FPGA in terms of slices than the CORDIC kernel. This is logical, as the CORDIC kernel has been specifically designed to be easily implemented on hardware. The RBF kernel uses a lot of memory as look-up tables, while the CORDIC kernel computes simple operations directly. The memory usage of the RBF kernel has a cost in terms of routing; the memory is not always located in the same place where computations occur, so long wires have to be used to fetch data from these memories. The delay induced by these wires lowers the maximum frequency at which the design can work. Tables V and VI show that CORDIC implementations work at higher frequencies than the RBF implementations.

Particularly with the RBF kernel, the Brescia trace gives poor results because of its low working frequency. The particularity of this trace is that the SVM model contains more support vectors than the others. They use too much space on the FPGA, which makes the delays caused by the look-up tables worse. Long and slow routes are created in the design and decrease its maximum frequency. The Ericsson and ResEl traces' SVM models have fewer support vectors.

Another important advantage of the CORDIC kernel is that models have fewer support vectors. This means that less memory has to be used on the FPGA to store these vectors, and less time has to be spent classifying one flow. With n computation units, each support vector costs $1/n$ clock cycle.

Both the higher frequencies and lower number of support vectors give an important advantage to the CORDIC kernel, which is visible in the much higher bit rate supported for each trace. The area used is also smaller for the CORDIC kernel, which allows a higher level of parallelism.

3) Overall performance: The lowest supported classification rate is 17 707 flows per second for the Brescia trace using the RBF kernel with 2 computation units. With the CORDIC kernel and 16 computation units, all traces can be handled with a classification rate of more than 300 000 flows per second. The Brescia trace is the most challenging; fewer flows can be classified per second by one computation unit, because the model has the most support vectors. But parallelization makes it possible to reach high bit rates: with 16 computation units, the trace could be processed at 320 075 flows per second. This would correspond to replaying the trace at 473 Gb/s.

It should be noted that using two computation units, the CORDIC kernel is 70% faster in terms of flows for the ResEl trace than for the Brescia trace, but with only a 27% increase in bit rate. This is because there are more flows to classify per second for the ResEl trace (Table I). Improvements compared to the software implementation are significant. For the Brescia trace, using the CORDIC kernel with 16 computation units, the bit rate is multiplied by 173 when compared to the software implementation (Table II), which is mostly due to the massive parallelism of the FPGA implementation.

One performance result that is not visible in the tables is the delay that the classifier adds to the packets if it is used directly on a network link to tag packets with their class

Trace	**Ericsson**			**Brescia**			**ResEl**		
Computation units	**2**	**4**	**8**	**2**	**4**	**8**	**2**	**4**	**8**
Occupied slices	6 223	11 362	22 431	11 007	15 716	24 311	5 575	8 962	17 225
Occupied slice registers	8 395	21 010	44 302	7 362	17 644	38 644	5 826	13 399	30 983
Occupied slice LUTs	19 451	36 345	79 602	38 242	51 914	83 968	17 881	28 356	58 568
FPGA usage (% of slices)	16.6	30.3	59.9	29.4	42.0	64.9	14.9	23.9	46.0
Maximum frequency (MHz)	153	165	171	126	71.0	77.6	175	181	154
Cycles per flow	1 602	814	421	7 098	3 562	1 795	2 550	1 288	658
Flows per second	95 636	203 091	405 894	17 707	19 944	43 256	68 463	140 194	233 844
Max. bit rate (Gb/s)	130	275	550	26.1	29.5	63.9	75.6	155	258

TABLE V
SYNTHESIS RESULTS OF SVM TRAFFIC CLASSIFICATION WITH AN RBF KERNEL ON A VIRTEX-5 XC5VTX240 FPGA

Trace	**Ericsson**			**Brescia**			**ResEl**		
Computation units	**2**	**4**	**16**	**2**	**4**	**16**	**2**	**4**	**16**
Occupied slices	4 767	8 092	24 325	6 131	9 108	25 444	4 730	6 834	24 694
Occupied slice registers	8 966	17 168	59 347	8 048	16 462	60 945	6 436	12 624	47 847
Occupied slice LUTs	14 880	24 973	74 664	20 440	29 137	80 762	15 664	22 419	78 681
FPGA usage (% of slices)	12.7	21.6	65.0	16.4	24.3	68.0	12.6	18.2	66.0
Maximum frequency (MHz)	201	199	183	176	201	172	182	181	125
Cycles per flow	903	469	146	4 034	2 034	537	2 449	1 242	339
Flows per second	223 090	424 148	1 254 223	43 704	98 822	320 075	74 201	145 808	369 794
Max. bit rate (Gb/s)	302	575	1 700	64.5	146	473	82.0	161	408

TABLE VI
SYNTHESIS RESULTS OF SVM TRAFFIC CLASSIFICATION WITH A CORDIC KERNEL ON A VIRTEX-5 XC5VTX240 FPGA

number (and not just as a passive probe set up in parallel on the link). The current implementation sends the classification to the host computer instead of tagging the packets, but it could be modified without overhead. For now, 10 clock cycles are required between the arrival of the packet in the classifier and the time when the class of the packet is known. This delay is constant because a packet is considered unknown if the flow has not yet been classified (its class is not in RAM), so it does not depend on the classification time, but only on the flow reconstruction time. On the Combo board, packets are processed with a frequency of 187.5 MHz, which gives an induced delay of 53.3 ns. This delay is acceptable as latency in IP networks is expressed in tens of milliseconds [44].

To improve the supported speed for all traces, many directions are possible. Critical paths in the design may be improved to achieve higher frequencies, by adding registers that cut these paths. Algorithmic changes might also allow a reduction in the number of support vectors, as the use of the CORDIC kernel unexpectedly does. It is also possible to use more powerful FPGAs, or to use multiple FPGAs in parallel to reach higher speeds by having more parallel units. For example, we tested a synthesis using the CORDIC kernel on the Brescia trace with 32 computation units on a Virtex-7 FPGA, which corresponds to the Xilinx VC709 board. With this configuration, $661\,987$ flows per second are classified, a 107% improvement compared to the NetFPGA 10G.

4) Sensitivity analysis: Different factors can affect the performance results presented above, both in terms of computation complexity and supported speed.

The number of support vectors affects both the on-chip memory requirements and the computation time. Each computation unit is able to start computation for one new support vector at each clock cycle. So with n computation units, each support vector costs $1/n$ clock cycle.

The number of classes affects FPGA usage. With n classes, $n(n-1)/2$ binary classifications have to be made in parallel, so FPGA usage increases in n^2. For example, to support twice as many classes, four times fewer computation units could be implemented on the same FPGA, dividing the supported speed by four.

The number of flow features affects FPGA usage. Only the first stage of the kernel computation is affected by an increase in the number of flow features. For the CORDIC kernel, one subtraction and one addition is required for each feature, and these operations are duplicated linearly when the number of flow features increases.

These parameters can all be changed in our generic SVM implementation simply by changing a variable value.

B. Implementation validation

1) Experimental setup: Previous results were obtained using only a synthesis tool. In this section we present results obtained directly by sending traffic using a XenaCompact 10G generator [45] to a Combo board on which the classifier is implemented. Packets are received through an optical fiber at 10 Gb/s by the Combo board, which processes them, adds a tag with their class number, and sends them to the computer hosting the card. A program runs on the computer and logs all received classifications. The tag is currently in a Combo-specific header that is not part of the packet, but it could also be put in the IP header, using the DSCP field for example.

2) Packet processing speed: The first thing to test is that the classifier handles traffic of 10 Gb/s without dropping packets. The most challenging packets to handle are the smallest ones, because handling time is constant for all packets, regardless

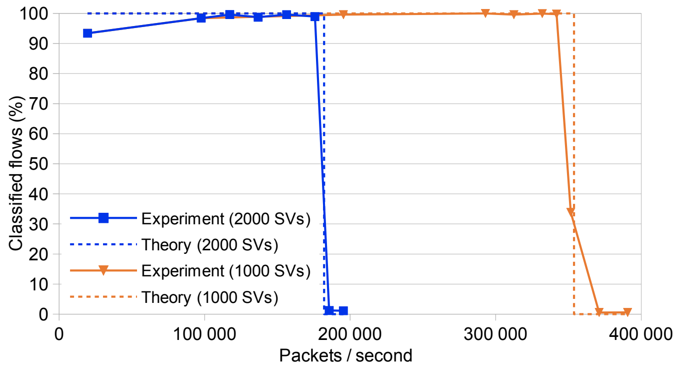

Fig. 5. Classified flows depending on the packet rate for an SVM model with a CORDIC kernel for 1 000 and 2 000 support vectors.

of size. The Ethernet standard defines the smallest Ethernet frames as being 64 bytes long. So in this experiment, 64-byte packets are sent with a configured inter-frame gap and the number of dropped packets is monitored. Results are simple: no packets are dropped for an inter-frame gap between 200 and 16 ns. With an inter-frame gap of 13 ns (the lowest possible gap with the generator), 0.5% of the packets are dropped. The lowest time authorized by Ethernet is the delay required to send 20 bytes, which is 16 ns at 10 Gb/s. So the classifier supports 10 Gb/s even with the smallest packets.

3) Flow processing speed: The second thing to test is the speed of the classifier itself. All the packets are handled properly, but how long does it take to classify one flow? To answer this question, 10 000 UDP flows of six 64-byte packets each are sent to the classifier. The rest of the link is filled with ICMP packets that are ignored by the classifier. Flows could be UDP or TCP, as they are handled exactly the same way, but they have to contain at least six packets; otherwise no packet will be classified. Indeed, the first two packets are ignored, the sizes of the next three are used for classification, and the 6^{th} one is tagged with the class that is found. This way, if the classification time is longer than the delay between the 5^{th} and 6^{th} packets, no packet of the flow is classified.

Figure 5 shows the percentage of classified flows depending on the packet rate for an SVM model with a CORDIC kernel on two computation units. The orange triangles are for a model with 1 000 support vectors and the blue squares for a model with 2 000 support vectors. The packet rate is constant throughout each experiment. The clock used for the flow classifier is the same as that for the packet processing unit, so its frequency is 187.5 MHz on the Combo board.

With 1 000 support vectors, the classification percentage is around 100% up to about 350 000 packets per second. With 2 000 support vectors, it starts failing at about 180 000 packets per second. This is actually the maximum number of flows that can be classified per second, as the delay available to classify a flow is the delay between 2 packets. Using the same method as in Section VI-A, the theoretical maximum is 353 773 flows per second for 1 000 support vectors, and 182 039 flows per second for 2 000 support vectors, as illustrated by the dotted line. Differences near the maximum supported speed are only due to the small number of tested packet rates.

It can be seen that some flows are not classified even with a low number of packets per second. This is due to the generator, which sometimes does not respect the configured inter-packet time. By logging sent packets in a file, we noticed that at low speeds, the generator tends to send packets in bursts, so some flows are not classified when the packet rate is too high. This is a flaw of the traffic generator.

The concordance of the theoretical and measured results validates the implementation. It also validates the method used to compute theoretical performance values in Section VI-A.

VII. Conclusion and Future Work

This article describes the practical implementation of SVM-based traffic classification. Simpler classification algorithms like C4.5 exist, but we show that with the chosen flow features, which do not include source and destination ports, SVM provides slightly better accuracy. The proposed SVM implementation is generic and can be adapted to other problems.

We use traffic-processing boards based on FPGAs like the NetFPGA 10G, taking advantage of very low-level access to network interfaces and massive parallelism.

This article focuses on the flow classification process using the SVM algorithm, which is implemented in a fully parallel way thanks to a pipeline computing classification data on one support vector each clock cycle. To accelerate this process, multiple processing units can work in parallel, dividing the required time to handle flows faster.

Two different kernels were tested, namely the well-known and generic RBF kernel, and a kernel more adapted to hardware, called the CORDIC kernel. They both give very similar levels of classification accuracy, but the CORDIC implementation supports higher working frequencies and uses less area on the FPGA, making it possible to put more processing units in parallel. An unexpected improvement is that SVM models obtained with the CORDIC kernel have fewer support vectors than with the RBF kernel, which accelerates processing.

Thanks to these optimizations, classification can be performed at 320 075 flows per second for a model with 8 007 support vectors, which would allow a real-time classification of the most realistic Brescia trace at 320 075 flows per second.

Thanks to this very fast implementation of the classification algorithm, the new bottleneck to get a full classification chain working at hundreds of gigabits per second will be the flow reconstruction process. External memories working at fixed frequencies have to be accessed to store data about flows. Higher speeds also mean more concurrent flows to store, creating more significant memory requirements. So in future work we will focus on scaling the flow reconstruction process. To do so, we will have to use efficient storage algorithms with small memory requirements and a limited number of accesses to the memory. Principles used by the Count Min Sketch algorithm [46] to store a large number of counters in a limited memory can be adapted to store flow data.

This way we will be able to test this algorithm on boards with 100 Gb/s interfaces as soon as they become available. We will also make the current classifier more flexible by storing the SVM model in RAMs instead of ROMs, so that a new synthesis is not necessary at each model update.

This hardware classifier is another step towards flexible and accurate online traffic classification without subsampling on high bit rate links. The generic SVM implementation on hardware can also be used for other classification problems.

Acknowledgment

This work has been partly funded by PRACOM, by the EU FP7 DEMONS (257315) project and by the VIPEER project of the French National Research Agency (ANR-09-VERS-014).

References

[1] Cisco Systems, "Cisco Visual Networking Index: Forecast and Methodology, 2010-2015."

[2] Y. Xue, D. Wang, and L. Zhang, "Traffic classification: Issues and challenges," in *Computing, Networking and Communications (ICNC), 2013 International Conference on*, 2013, pp. 545–549.

[3] P. Haffner, S. Sen, O. Spatscheck, and D. Wang, "ACAS: Automated construction of application signatures," in *SIGCOMM 2005 MineNet Workshop*, 2005.

[4] A. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2005, pp. 50–60.

[5] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. Claffy, "GT: picking up the truth from the ground for Internet traffic," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 5, pp. 13–18, 2009.

[6] A. Este, F. Gringoli, and L. Salgarelli, "Support vector machines for TCP traffic classification," *Computer Networks*, vol. 53, no. 14, pp. 2476 – 2490, 2009.

[7] P. Bermolen, M. Mellia, M. Meo, D. Rossi, and S. Valenti, "Abacus: Accurate behavioral classification of P2P traffic," *Elsevier Computer Networks*, vol. 55, no. 6, pp. 1394–1411, 2011.

[8] H. Kim, D. Barman, M. Faloutsos, M. Fomenkov, and K. Lee, "Internet traffic classification demystified: The myths, caveats and best practices," in *Proc. ACM CoNEXT*, 2008.

[9] T. Nguyen and G. Armitage, "A survey of techniques for Internet traffic classification," *IEEE Communications Surveys and Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[10] Y.-s. Lim, H.-c. Kim, J. Jeong, C.-k. Kim, T. T. Kwon, and Y. Choi, "Internet traffic classification demystified: on the sources of the discriminative power," in *Proceedings of the 6th International Conference*, ser. Co-NEXT '10. ACM, 2010, pp. 9:1–9:12.

[11] N. Williams, S. Zander, and G. Armitage., "A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification," *ACM SIGCOMM Computer Communication Review*, 2006.

[12] T. Groleat, M. Arzel, and S. Vaton, "Hardware acceleration of SVM-based traffic classification on FPGA," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2012 8th International*, 2012, pp. 443–449.

[13] J. a. V. Gomes, P. R. M. Inácio, M. Pereira, M. M. Freire, and P. P. Monteiro, "Detection and classification of peer-to-peer traffic: A survey," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 30:1–30:40, Jul. 2013.

[14] G.Gomez and P.Belzarena, "Early Traffic Classification using Support Vector Machines," in *Fifth International Latin American Networking Conference (LANC'09)*, 2009.

[15] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, Aug. 2008.

[16] N. Cascarano, L. Ciminiera, and F. Risso, "Optimizing deep packet inspection for high-speed traffic analysis," *J. Netw. Syst. Manage.*, vol. 19, no. 1, pp. 7–31, Mar. 2011.

[17] D. Rossi and S. Valenti, "Fine-grained traffic classification with Netflow data," in *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, 2010, pp. 479–483.

[18] A. Este and F. Gringoli, "On-line SVM traffic classification," in *Proceedings of the 7th IWCMC Conference (IWCMC TRAC'2011)*, 2011.

[19] F. Gringoli, L. Nava, A. Este, and L. Salgarelli, "MTCLASS: enabling statistical traffic classification of multi-gigabit aggregates on inexpensive hardware," in *Proceedings of the 8th IWCMC Conference (IWCMC TRAC'2012)*, 2012.

[20] P. M. Santiago del Rio, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil, "Wire-speed statistical classification of network traffic on commodity hardware," in *Proceedings of the 2012 ACM conference on Internet Measurement Conference*, 2012, pp. 65–72.

[21] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, "Comparing and improving current packet capturing solutions based on commodity hardware," in *Proceedings of the 10th annual conference on Internet measurement*. ACM, 2010, pp. 206–217.

[22] G. Szabó, I. Gódor, A. Veres, S. Malomsoky, and S. Molnár, "Traffic classification over Gbit speed with commodity hardware," *IEEE J. Communications Software and Systems*, vol. 5, 2010.

[23] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Midea: a multi-parallel intrusion detection architecture," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 297–308.

[24] A. Carpenter, "CUSVM: A cuda implementation of support vector classification and regression," 2009.

[25] NetFPGA, "NetFPGA: a line-rate, flexible, and open platform for research, and classroom experimentation," http://netfpga.org/.

[26] CESNET, "Our hardware," http://www.liberouter.org/hardware.php?flag=U, Nov. 2011.

[27] J. Mu, S. Sezer, G. Douglas, D. Burns, E. Garcia, M. Hutton, and K. Cackovic, "Accelerating pattern matching for DPI," in *SOC Conference, 2007 IEEE International*, 2007, pp. 83–86.

[28] W. Jiang and M. Gokhale, "Real-Time Classification of Multimedia Traffic Using FPGA," in *Field-Programmable Logic and Applications*, 2010, pp. 56–63.

[29] M. Papadonikolakis and C. Bouganis, "A novel FPGA-based SVM classifier," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 283–286.

[30] A. Kennedy, Z. Liu, X. Wang, and B. Liu, "Multi-engine packet classification hardware accelerator," in *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, 2009, pp. 1–6.

[31] 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, 2013, pp. 255–264.

[32] C. Cortes and V. Vapnik, "Support-vector networks," in *Machine Learning*, 1995, pp. 273–297.

[33] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[34] Groléat, T., "Hardware traffic classifier," https://github.com/tristan-TB/hardware-traffic-classifier, 2014.

[35] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, 1992, pp. 144–152.

[36] Clear Foundation, "l7-filter: application layer packet classifier for Linux," http://l7-filter.clearfoundation.com/.

[37] T. D. Hanson, "Ut hash," http://troydhanson.github.com/uthash/, 2013, [Online; accessed 1-March-2013].

[38] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46 –55, jan-mar 1998.

[39] M. Žádník and L. Lhotka, "Hardware-accelerated netflow probe," Technical Report 32/2005, CESNET, Praha, Tech. Rep., 2005.

[40] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: theory, algorithm, and FPGA implementation," *Neural Networks, IEEE Transactions on*, vol. 14, no. 5, pp. 993 – 1009, sept. 2003.

[41] D. Anguita, S. Pischiutta, S. Ridella, and D. Sterpi, "Feed-forward support vector machine without multipliers," *Neural Networks, IEEE Transactions on*, vol. 17, no. 5, pp. 1328 –1331, sept. 2006.

[42] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field Programmable Gate Arrays*, 1998, pp. 191–200.

[43] J. Gimeno Sarciada, H. Lamel Rivera, and M. Jiménez, "CORDIC algorithms for SVM FPGA implementation," 2010.

[44] AT&T, "Global IP network latency," http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html, 2013, [Online; accessed 21-March-2013].

[45] "XenaCompact," http://www.xenanetworks.com/html/xenacompact.html, 2012, [Online; accessed 6-February-2013].

[46] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.