High-throughput Traffic Classification on Multi-core Processors*

Da Tong, Yun R. Qu, Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: datong@usc.edu, yunqu@usc.edu, prasanna@usc.edu

Abstract—Traffic classification is a critical mechanism performed for many important network management tasks at Internet routers. Decision-trees are commonly used in Machine Learning (ML)-based traffic classification algorithms to realize efficient search operations. Most of the existing works are hardware-based, while a new trend for network applications is to use software accelerators and virtual machines. Since the decision-tree used for traffic classification is highly imbalanced, it is challenging to achieve high throughput for decision-tree-based traffic classification on multi-core processors. In this paper, we present a high-throughput traffic classifier employing scalable data structures on multi-core processors. We convert decision-trees used in ML-based algorithms into a compact rule set table. Based on this data structure, we develop a divide-and-conquer algorithm by (1) searching all the columns of this table in parallel, and (2) merging the outcomes from all the columns into the final classification result. A high throughput can be sustained using our approach even if the size of the rule set table is scaled up in (1) vertical direction with respect to the number of decision-tree leaves, and (2) horizontal direction with respect to the number of features examined during the classification process. We prototype our design on state-of-the-art multi-core processors. For a typical decision-tree-based traffic classifier consisting of 128 leaf nodes and 6 flow-level features, our traffic classifier achieves a throughput of 90 Million Lookups Per Second (MLPS). Our traffic classifier sustains high throughput even for highly imbalanced decision-trees consisting of a large number of nodes. We achieve 13× throughput compared to the SVM-based traffic classifiers on multi-core processors.

Keywords—traffic classification, multi-core, performance

I. INTRODUCTION

Traffic classification [1] serves as one of the kernel applications at the network routers. It requires the network traffic flows to be categorized into various application classes based on many criteria; this benefits many value-added services as well as the network security. The rapid growth of the Internet requires traffic classifiers to support extremely high throughput of over hundreds of gigabits per second. However, most of the existing traffic classification engines only support a few tens of gigabits per second throughput [2]; this makes traffic classification a performance bottleneck for high-speed routers.

Existing traffic classification approaches can be categorized into 4 classes: (1) port-number-based schemes classify traffic based on transport-layer port numbers. Since many applications today tend to dynamically assign port numbers, port-number-based approaches are no longer reliable. (2) Deep Packet Inspection (DPI) compares the traffic payload with known signatures. DPI-based techniques [3] can achieve the highest accuracy; however, they suffer from long processing latency. (3) heuristic-based techniques [4] classify traffic based on heuristic patterns; compared to other techniques, the classification accuracy of heuristic-based techniques is relatively low. (4) Machine-Learning (ML) techniques, including the well-known C4.5 decision-tree, examine statistical properties of traffic [5], [6]. ML-based techniques have demonstrated high classification accuracy; however, realizing high-performance ML-based traffic classification engine is still challenging.

A new trend in network applications is to use software accelerators and virtual machines [7]–[9]. State-of-the-art multi-core processors [10], [11] exploit caches and instruction level parallelism (ILP) to improve the performance; this makes multi-core processors an attractive platform for high-performance network applications. However, efficient parallel algorithms are still needed on multi-core processors.

In this paper, we present a high-throughput traffic classification approach on multi-core processors. This approach employs a novel divide-and-conquer algorithm based on a compact rule set table. Specifically, our contributions include the following:

- We convert a generic decision-tree to a compact rule set table. We develop a divide-and-conquer algorithm based on the rule set table.
- We propose the rule set table as a scalable data structure. Using our approach, a high traffic classification throughput can be sustained even if the size of the rule set table is scaled up in both the number of decision-tree leaves, and the number of flow-level features.
- We prototype our approach and conduct extensive experiments on both the state-of-the-art AMD and Intel multi-core processors. Our classifier sustains high throughput even for highly imbalanced decision-trees consisting of a large number of nodes.
- We support 90 MLPS throughput with 97% classifica-

*This work is supported by U.S. National Science Foundation under grant CCF-1320211.
Table I: Example rule set table ($N = 3$ rules and $M = 6$ features)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>6891, 6901</td>
<td>6891, 6900</td>
<td>(0, 1500)</td>
<td>[1200, 1500)</td>
<td>[0, 250)</td>
<td>MSN</td>
</tr>
<tr>
<td>TCP</td>
<td>any</td>
<td>80</td>
<td>(0, 1500)</td>
<td>[1000, 1500)</td>
<td>[0, 500)</td>
<td>HTTP</td>
</tr>
<tr>
<td>UDP</td>
<td>1542</td>
<td>1963</td>
<td>(0, 1000)</td>
<td>[1200, 1500)</td>
<td>[0, 100)</td>
<td>MSN</td>
</tr>
</tbody>
</table>

B. Performance of Traffic Classification

Many existing works have proposed to use reconfigurable hardware devices such as FPGA for online traffic classification. In [2], an FPGA-based architecture is presented for the C4.5 algorithm; explicit range match is explored and memory accesses are parallelized to improve the performance of their architecture. They use the number of memory accesses instead of throughput as a performance metric; no post-place-and-route results are reported for their implementation on FPGA. In [16], an FPGA-based architecture is proposed for multimedia traffic classification. The classifier is based on $k$-Nearest-Neighbor algorithm and packet-level features.

Their classifier achieves high accuracy for large training data sets. Their approach is restricted to classifiers with a small number of application classes. In [17], a hardware implementation of C4.5 decision-tree is presented for online traffic classifier on FPGA. Their architecture is based on a straightforward mapping from a binary decision-tree onto FPGA. As the tree grows larger, the clock rate and throughput are affected negatively by the increasing wire length; the scalability of their approach is very limited.

In [18], an SVM-based approach is explored on a dual Xeon PC; this machine has a total number of 24 cores running at 2.6 GHz and 48 GB RAM. An SVM-based algorithm is also employed in [19], targeting a dual Xeon PC with 24 GB RAM and a total number of 24 cores each running at 2.6 GHz. However, these approaches can only achieve 7 MLPS throughput. To the best of our knowledge, we are not aware of any prior work realizing high-performance traffic classifier on multi-core processors.

C. Problem Definition

We define a generic decision-tree-based classification problem as: *given an arbitrary (possibly imbalanced) decision-tree, and $M$ input features, find the corresponding application class with high throughput*. To address this problem, we develop a parallel algorithm based on a scalable data structure on multi-core processors.

III. DATA STRUCTURE

A. Classic C4.5 Decision-tree

The classic C4.5 decision-tree-based approach consists of three phases: training, discretization and decision-making. During the training phase, a group of $M$ discretization-trees [20], [21] are built for $M$ features, respectively; each leaf node of a discretization-tree stores a unique number representing a range of feature values. For example, a unique
number of 1000 can represent the range $[1000, 1200)$ for a particular feature. A binary decision-tree $T$ is also built in order to make decisions based on the unique numbers. We show an example of the classic C4.5 decision-tree-based approach in Figure 1; two features, the Destination Port number (DP) and the Average Packet Size (APS) are used for classification. Note $T$ is highly imbalanced, implying a long worst-case search time. Unfortunately, little changes can be made to rearrange $T$ into a balanced tree without losing classification accuracy, since the tree branches are predetermined by the C4.5 algorithm.

During discretization, each feature collected from the traffic is searched first in its corresponding discretization-tree until a leaf node is reached; the outputs of the discretization process are the unique numbers stored in the reached leaves.

After the inputs for all the $M$ features are discretized, the decision-tree $T$ is searched during the decision-making phase. Each non-leaf node in the decision-tree makes a decision based on the $M$ unique numbers from the discretization process, along with the outcome of a “true/false” statement. For example, in Figure 1, the root node in $T$ represents the statement “DP = 80”. Since the discretization process provides a unique number of DP = 40, the left child of the root node is searched next. This process goes on until a leaf node of $T$ is reached, where a final decision on the application class can be made.

The above decision-tree-based approach has the following disadvantages: (1) Two types of trees are built and searched; this increases the search time and degrades the throughput. (2) The worst-case performance of the tree structures depends on the shape of the tree; for unbalanced trees, the worst-case search time is almost linear with respect to the number of tree nodes.

\section*{B. Compact Rule Set Table}

We define a Rule Set Table (RST) to be a table consisting of $N$ rows, each row having $M$ columns specified for $M$ features, respectively. Each row defines a rule to be satisfied for a specific application class. An example of a RST having $N = 3$ rows and $M = 6$ columns is shown in Table I. Note that each cell of the RST stores a range; also note an application class can specify multiple rules, such as the application “MSN” in the example RST. RST is not explicitly given in any decision-tree-based traffic classification algorithms.

\begin{algorithm}[h]
\caption{Constructing RST}
\begin{algorithmic}[1]
\State \textbf{Input} A $T'$ having $N$ leaves and using $M$ features.
\State \textbf{Output} An array $RST(n, m), n = 0, 1, \ldots, N - 1, m = 0, 1, \ldots, M - 1$; it has $N$ rows (rules) and $M$ columns (features). Each cell $RST(n, m)$ stores a range.
\For{$n = 0$ to $N - 1$} \Comment{RST construction}
\For{$m = 0$ to $(M - 1)$} \Comment{leaf node $n$}
\If{$RST(n, m) = \emptyset$} \Comment{associate a new range}
\State $RST(n, m) \leftarrow [0, 2^K]$; \Comment{initialized as a full range}
\EndIf
\EndFor
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

Our work is motivated by the following observations: (1) The discretization-trees and decision-tree can be combined into one decision-tree. (2) The new decision-tree can be converted to a compact RST; this RST can be searched efficiently using parallel algorithms.

First of all, the discretization-trees and the decision-tree have to be merged into a new decision-tree $T'$. This can be done by simply replacing each “true/false” statement in $T$ by a comparison statement between the input features and an explicit range. For instance, the statement inside the root node of $T$ in Figure 1 can be replaced by the statement “DP $\in [80, 81)$” in $T'$.

Now let us define the following concepts: each leaf $n$ of $T'$ can be reached from the root through a path consisting of statements $S_0^{(n)}, S_1^{(n)}, \ldots, S_{k_n}^{(n)}$, $k_n \leq K$, $K$ denoting the depth of $T'$. At most $M$ features are used along such a path; $m \rightarrow S_i^{(n)}$ denotes the feature $m$ is used in the statement $S_i^{(n)}$. $(S_i^{(n)})$ denotes the range $S_i^{(n)}$ specifies. Let $W_m$ denote the width of the feature $m$.

We convert $T'$ into a RST as shown in Algorithm 1. We construct RST by tracing $T'$ from all the leaf nodes back

\footnote{\text{Without loss of generality, we use half-closed half-open closures; any number in a range can be used as a unique number, as long as (1) the ranges are non-overlapping and (2) the decision-tree is properly tuned.}}

\footnote{\text{This is a very generic representation of a criterion.}}
to the root node. We can easily prove that, if a leaf node \( n \) is reached starting from the root by going through a path consisting of statement \( S_{i}^{(n)} \), \( i = 0, 1, \ldots, (k_n - 1) \) (where \( k_n \leq K, K \) denoting the depth of \( T^M \)), then the rule along this path is \( S_{i_k + 1}^{(n)} \). There is a one-to-one correspondence between a root-to-leaf path and a rule in the RST. Notice along a particular path, some of the features may be examined more than one time; it is also possible that one of the \( M \) features is never checked along this path, leading to an “any” criterion\(^3\) for this feature in the corresponding rule. As shown in Table I, the second rule has “any” for the SP feature; this means the corresponding root-to-leaf path in the decision-tree does not require the SP feature to be examined.

**Theorem 1:** For an arbitrary decision-tree having \( N \) leaves and using \( M \) features, Algorithm 1 produces a compact RST having \( N \) rows and \( M \) columns.

**Proof:** The proof is based on the following observation: if every statement in the decision-tree specifies a range, then the range specified by a child node must be a subset of the range specified by a parent node; otherwise it contradicts the properties of a tree structure. Specifically, along a path \( S_{i_0}^{(n)}, S_{i_1}^{(n)}, \ldots, S_{i_{k_n - 1}}^{(n)} \) from the root to the leaf \( n \), if \( \exists \) a feature \( m \), \( m \rightarrow S_{x_0}^{(n)}, m \rightarrow S_{x_1}^{(n)}, \ldots, m \rightarrow S_{x_{k - 1}}^{(n)} \), where \( x_0 < x_1 < \ldots < x_{k - 1} \) and \( 0 < k \leq k_n \), then this feature \( m \) is used \( k \) times. Hence we have \( \bigcap_{l = x_0}^{x_{k - 1}} S_{l}^{(n)} \subset \bigcap_{l = x_0}^{x_{k - 1}} S_{l}^{(n)} \subset \bigcap_{l = x_0}^{x_{k - 1}} S_{l}^{(n)} \). This implies \( \bigcap_{l = x_0}^{x_{k - 1}} S_{l}^{(n)} \). As a result, no matter how long such a path is, the corresponding rule can specify at most \( M \) ranges by employing all the \( M \) features.

Note the conversion from the decision-tree to the RST shown above does not depend on any characteristic of the tree itself, such as the shape, the depth, or the degree of the tree: (1) The number of rows in the RST only depends on the number of tree leaves; (2) the number of columns only depends on the number of features used for classification. The methodology used in this work is not restricted to binary trees or trees used exclusively for network applications. Algorithm 1 can be applied to any generic decision-tree.

**C. Unique range, Subrange, Range-tree, and Bit Vector**

To search the RST efficiently, we preprocess all the \( M \) columns of the RST to obtain \( M \) sets of *unique ranges*, respectively. We define unique ranges as the projections of rules on a specific feature \( m \); we denote the number of unique ranges for the feature \( m \) as \( q(m) \) \( (q(m) \leq N) \). For example, in Table I, the RST only has 2 unique ranges: \([0, 1500]\) and \([0, 1000]\) for the APS feature. Then we construct a range-tree \( T_m \) as shown in Figure 2:

- We flatten all the unique ranges into non-overlapping *subranges*. In Figure 2, we have two non-overlapping subranges \([0, 1000]\) and \([1000, 1500]\).

\(^3\)Expressed numerically as a full range, \([0, 2^W - m]\).

- For a specific feature, we use the subrange boundaries as non-leaf nodes to construct a balanced range-tree [21]. In the range-tree in Figure 2, each of the three non-leaf nodes stores a subrange boundary.
- Each leaf node of a range-tree stores a *Bit Vector* (BV) [22]. The length of each BV is \( N \) for \( N \) rules; each bit in the BV is set to “1” only if the input satisfies the corresponding rule. In Figure 2, the BV “110” indicates that the input located in subrange \([1000, 1500] \) satisfies the first two rules.

In the range-tree \( T_m \), the subrange boundaries stored in the non-leaf nodes are used for binary search; the BVs stored in the leaf nodes represent the search outcomes. Note the total number of nodes in a range-tree \( T_m \) is much less than the total number of nodes in the original decision tree \( T \); this implies faster lookup time. Also, the range-trees can be easily balanced, while the original decision-tree \( T \) does not have this property.

**IV. ALGORITHM**

Based on the data structures discussed in Section III, we develop a divide-and-conquer algorithm for efficient traffic classification. We present our parallel algorithm in 2 phases:

1. **Search:** search the range-trees in all the columns individually in parallel.
2. **Merge:** merge all the outcomes after the search phase into a final classification result.

**A. Search**

Since we have \( M \) individual range-trees, we exploit parallel search for all the features to improve the throughput. We show an example in Figure 3. In this example, the RST specifies \( N = 3 \) rules using \( M = 2 \) features: 2 BVs each of 3 bits are extracted as the search outcomes. We use \( BV_m \) to denote the BV extracted for the feature \( m \).

\(^4\)There are also other representations that can be used to store search outcomes [22]. However, BV is more efficient with respect to the operation used for merging, memory footprint, and performance.
INPUT: DP = 80, APS = 1000

<table>
<thead>
<tr>
<th>DP</th>
<th>APS</th>
<th>App. Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>[6891, 6900)</td>
<td>[0, 1500)</td>
<td>MSN</td>
</tr>
<tr>
<td>[80, 81)</td>
<td>[0, 1500)</td>
<td>HTTP</td>
</tr>
<tr>
<td>[0, 1964)</td>
<td>[0, 1000)</td>
<td>P2PTV</td>
</tr>
</tbody>
</table>

Figure 3: Divide-and-conquer algorithm

Algorithm 2 Search and Merge

Input: Range-trees $T_m$, $m = 0, 1, \ldots, M - 1$; M input feature values. “&” denotes the bitwise AND operation.

Output: The application class of the packet or packet flow being classified

1: for $m = 0$ to $M - 1$ par do
2: search the range-tree $T_m$ until a leaf node is reached
3: extract $BV_m$ in the reached leaf node
4: end for
5: $BV_f \leftarrow m^{M-1}BV_m$
6: check RST using $BV_f$ for the application class

B. Merge

In the merge phase, the $M$ BVs are combined by bitwise AND operation. This results in a final BV; let $BV_f$ denote this BV. A simple check using $BV_f$ against the RST produces the application class. In Figure 2, ANDing the 2 BVs gives a $BV_f$ “010”; this indicates the input only matches the second rule with “HTTP” as the application class. Notice in Figure 3, each column of the RST is processed, stored, and searched separately.

We show the complete algorithm for the search and merge phases in Algorithm 2. The search phase requires $O(M \cdot N)$-bit memory to store the search outcomes, while the $BV_f$ in the merge phase occupies an $O(N)$-bit memory. The parallel time complexity for the search phase is $O(\log(\max_m q^{(m)}))$; note $q^{(m)}$ is usually much smaller than $N$. The merge time complexity is $O(N)$.

V. PERFORMANCE EVALUATION

A. Experimental Setup

We conducted the experiments on a 2× AMD Opteron 6278 platform [10] and a 2× Intel Xeon E5-2470 platform [11]. The dual-socket AMD platform has 16 physical cores, each running at 2.4 GHz. Each core is integrated with a 16 KB L1 data cache and a 2 MB L2 cache. A 6 MB L3 cache is shared among all 16 cores. The processor has access to 64 GB DDR3-1600 main memory. The dual-socket Intel platform also has a total number of 16 cores, each running at 2.3 GHz. Each core has a 32 KB L1 data cache and a 256 KB L2 cache. All 16 cores share a 20 MB L3 cache. The processor has access to 48 GB DDR3-1600 main memory.

We implemented our classifier using Pthreads and OpenMP on openSUSE 12.2. Our classifier involves two threads: $P_0$ performs parallel search operations, and $P_1$ merges all the search outcomes. For $P_0$, we use OpenMP to assign $M$ subthreads for the $M$ features, respectively. A buffer is allocated between $P_0$ and $P_1$; Pthreads mutex variables are used to synchronize $P_0$ and $P_1$. Throughout this section, we use Pthreads and threads interchangeably.

We use throughput as the performance metric; we define throughput as the total number of classifications performed per second. As a baseline, we also implemented the classic C4.5 decision-tree on the two multi-core platforms; throughout this section, we denote classic C4.5 decision-tree as the implementation of the binary tree generated from the C4.5 algorithm. We target $M = 6$ flow-level features, as shown in Table I; they demonstrate high classification accuracy [17].

We classified network traffic into 8 categories [17], including HTTP, MSN, P2PTV, QQ_IM, Skype, Skype_IM, Thunder, and Yahoo_IM. We use the data set provided by Tstat [23], a publicly available traffic trace. Let classification accuracy denote the average percentage of correctly classified packet flows; we use WEKA [24], an existing ML software, to measure the classification accuracy.

B. Classification Performance

We have conducted thorough experiments to generate high-accuracy decision-trees [17]. Using the 6 flow-level features, the best accuracy achieved is 97.92%. To figure out the typical number of leaf nodes and number of unique ranges per feature, we collect around 100 decision trees whose accuracy is above 95%. We count their number of leaf nodes and number of unique ranges per feature. Their distributions are shown in Figure 4. As shown in the figure, most of the decision-trees have around 100 leaf nodes. Hence we use decision-trees consisting of $N = 128$ leaf nodes in our implementations. Notice the number of unique ranges distributes almost evenly among all the possible values. In our prototypes, we choose the RSTs having 64 and 128 unique ranges for each feature, although our approach does not constrain the number of unique ranges for each feature.

Each non-leaf decision-tree node has two subtrees. Let $\alpha_{left}$ and $\alpha_{right}$ denote the number of nodes of the two subtrees, respectively. We define balance factor $B = \frac{\max\{\alpha_{left}, \alpha_{right}\}}{\alpha_{left} + \alpha_{right}}$. For example, as shown in Figure 8, a perfectly balanced tree has $B = 0.50$; $B = 0.63$ indicates an imbalanced tree. Note $0.50 \leq B \leq 1.00$.

We implemented our approach ($N = 128$, $M = 6$, $q^{(m)} = 128$ unique ranges per feature, 2 Pthreads, and $B =$
0.9) on both the AMD and Intel platforms. Figure 6 shows the performances of the classic C4.5 decision-tree, SVM-based approaches [18], [19], and our approach. Our approach achieves 98 MLPS on the AMD platform and 90 MLPS on the Intel platform. In contrast, the classic C4.5 decision-tree only achieves 65 MLPS and 60 MLPS on the AMD and Intel platforms, respectively. Our approach achieves 1.5× higher throughput than the classic C4.5 decision-tree on both the AMD and the Intel platforms. We also achieve 13× throughput compared to the SVM-based approaches.

Figure 7 compares the throughputs of our approach and the classic C4.5 decision-tree when they are used to implement typical decision trees of various \((N, q^{(m)})\) values. The throughputs shown in the figure are the average throughputs of the implementations on the Intel and AMD platforms. Our approach shows constant 1.5× speedup compared to the classic C4.5 decision-tree over all the \((N, q^{(m)})\) values.

C. Varying Balance Factor

One major draw back of the classic C4.5 decision-tree is that the throughput depends highly on the shape of the tree. Figure 8 shows the throughput of both algorithms with respect to various balance factors achieved on the AMD platform. We implemented typical decision trees \((N = 128, M = 6, q^{(m)} = 128\) unique ranges per feature) using various balance factors. For an imbalanced decision-tree, the classifier has to travel a large number of levels to reach a leaf node in the worst case; this results in poor performance with respect to throughput. The performance of our approach does not depend on the shape of the tree. We converted the decision-trees in Figure 8 and implemented the classifier using our approach. As can be observed, the throughput of our approach varies little with respect to various values of \(B\). We also see consistent performance on the Intel platform.

D. Varying the Number of Threads

The other major draw back is, the classic C4.5 decision-tree offers smaller chance for multi-threading speedup. Figure 9a shows the throughput of the classic C4.5 decision-tree using various number of threads (Pthreads) on the AMD platform, with respect to various combinations of \((N, q^{(m)})\). If we use more than 2 threads, the reason is:

- The decision-tree requires the order of operations to be strictly maintained. To search a tree, the input must start from the root and traverse down the tree level-by-level. This offers little chance for the hardware/software scheduler to improve throughput within each thread.

- Concurrent threads may compete for the limited resources. As long as the tree is large, it is also difficult to use cache efficiently among various threads.

In contrast, our approach employs parallel search and efficient merge operations. As shown in Figure 9b, for all the

Note the Intel platform has a slower clock frequency.
(N, q^{(m)}) pairs in our experiments, we observe significant speedup as we increase the number of threads.

E. Scalability of Our Approach

Figure 10 shows the throughput of our approach using various number of cores.

Figure 11 and Figure 12 show the throughput of our approach on the AMD platform; we constructed synthetic RSTs and vary the number of features (M) and the number of unique ranges per feature (q^{(m)}). In Figure 11, for each value of M, we show the performance for various values of N (the number of leaf nodes in the decision-tree, ranging from 128 up to 2048); similarly, for each value of q^{(m)}, we show the performance with respect to various values of N in Figure 12. Note:

- The throughput varies almost linearly with respect to both M and N. Larger values of M or N require wider BVs to be merged and larger memory footprint to store the data structures in Section III; this results in a throughput degradation.
- The throughput varies little with respect to q^{(m)}. Note the time required to search each balanced range-tree \( T_m \) is \( \log(q^{(m)}) \).

VI. CONCLUSION

In this paper, we proposed a scalable data structure and an efficient parallel algorithm to accelerate decision-tree-based traffic classification on multi-core processors. Our algorithm achieves 90 MLPS when accelerating a typical decision-tree used for traffic classification. This is at least 13× better compared to existing traffic classifiers on multi-core processors. To the best of our knowledge, this is the first high-throughput traffic classifier on multi-core processors. Our parallel algorithm and design methodology can be extended to accelerate any generic decision-tree. In the future, we plan to explore more optimization techniques to further enhance the performance of the traffic classifiers.

REFERENCES


