Toward Advocacy-Free Evaluation of Packet Classification Algorithms

Haoyu Song, *Member*, *IEEE*, and Jonathan S. Turner, *Fellow*, *IEEE*

Abstract—Understanding the real performance of a proposed algorithm is a basic requirement for both algorithm designers and implementers. However, this is sometimes difficult to achieve. Each new algorithm published is evaluated from different perspectives and based on different assumptions. Without a common ground, it is almost impossible to compare different algorithms directly. Choosing an incompetent algorithm for an application can incur significant cost. This is especially true for packet classification in network routers, since packet classification is intrinsically a hard problem and all existing algorithms are based on some heuristics and filter set characteristics. The performance of the packet classification subsystem is critical to the overall performance of the network routers. Although numerous algorithms have been proposed so far, a benchmark that can give them consistent evaluation and reveal their comparable performance is still missing. This paper summarizes our efforts toward improving this situation. First, we conduct a high-level survey on the existing algorithms and extract some insights on the general design ideas. Second, we describe an open-source platform dedicated for advocacy-free evaluation of packet classification algorithms. Many representative algorithms are actually implemented under a set of uniform conditions and assumptions. The freely available implementations allow other researchers to easily test them under different scenarios. We also enforce some consistent and fundamental criteria for the algorithm evaluation, so that their performance and potentials are directly comparable, regardless of the actual implementation platforms. This project serves dual purpose: It helps the researchers to accelerate the innovation in the area of packet classification algorithm development by relieving them from the labor of replicating the previous work and by enabling them to quickly compare and evaluate algorithms. Meanwhile, it also helps the system implementers to easily choose the capable algorithm for their particular applications. Aiming to build an open-source library, we encourage external contributions of new algorithm implementations and evaluations under the same framework. We believe the practice will benefit the research and design community as a whole.

Index Terms—Packet classification, algorithm evaluation.

1 Introduction

PACKET classification enables network routers to provide advanced network services in addition to the basic packet forwarding, such as network security, policy-based routing, and quality of service (QoS) assurance. There is increasing industrial and academic interest in high-performance algorithms and systems for packet classification. On the one hand, network security and QoS have become the driving factors requiring large-scale packet classification. Currently, the largest packet filter sets in use contain thousands of filters and each filter involves five or more packet header fields. Tens of thousands of filters in a filter set are expected in the near future. On the other hand, increasing network traffic poses greater challenges than ever for the application of large-scale packet classification. Today, OC-192 and 10 GbE (10 Gbps) line speeds are common in edge networks, while the use of OC-768 (40 Gbps) and 100 GbE (100 Gbps) line speeds starts to emerge in core networks. A fully loaded OC-192 line can see as many as 30 million packets in a second, leaving only 32 ns to classify a packet. This daunting task is even more challenging when higher line speeds are considered. As a result, packet classification is still an open problem demanding continuing investigations.

The function of packet classification is to match packet headers against a set of predefined filters. The relevant packet header fields usually include the source IP address, the destination IP address, the transport layer protocol, the source port, and the destination port. Other header fields (e.g., the MAC layer header, the IP class of service, and the TCP flags) can also be included. Formally, a filter set consists of a finite set of n filters: $R_1, R_2 \ldots R_n$. Each filter is a combination of k header field specifications: $H_1, H_2 \ldots H_k$. Each header field specifies one of four types of matches: exact match, prefix match, range match, or masked-bitmap match. A packet P is said to match a filter R_i if and only if its header fields, $H_1, H_2 \ldots H_k$, match the corresponding fields in R_i as specified. Each filter R_i has an associated action A_i that determines how a matching packet P should be handled. Since filters can overlap, it is possible for a packet to match multiple filters. To resolve the ambiguity, each filter is assigned a priority. For a packet, the filter with the highest priority among all the matching filters is chosen as the best matching filter. Usually, a filter's priority is implied by its position in the ordered filter set: the earlier it appears, the higher its priority is. In case the application requires finding all the matching filters, the order loses its significance.

The most compact storage for a filter set (i.e., $O(n)$ storage for n filters) requires the linear search for packet classification.

- *H. Song is with the Bell Labs, Alcatel-Lucent, HOH R-231, 791 Holmdel-Keyport Rd., Holmdel, NJ 07733. E-mail: haoyu.song@alcatel-lucent.com.*
- *J.S. Turner is with the Department of Computer Science and Engineering, Washington University in St. Louis, Campus Box 1045, 1 Brookings Dr. St. Louis, MO 63130. E-mail: jst@arl.wustl.edu.*

Manuscript received 3 Apr. 2009; revised 10 Sept. 2009; accepted 16 Dec. 2009; published online 6 Dec. 2010.
Recommended for acceptance by V. Leung.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-04-0143.
Digital Object Identifier no. 10.1109/TC.2010.252.

Although simple, the scheme can be prohibitively slow even for moderate-sized filter sets. In most applications, more sophisticated algorithms or Ternary Content Addressable Memory (TCAM) hardware must be used to accelerate the lookup speed. Each approach has its own advantages and disadvantages in terms of throughput, cost, power dissipation, ease of implementation, and scalability. Hybrid architectures that leverage these two approaches are also possible.

In Section 2, we briefly survey the existing algorithms from a very high-level perspective, trying to capture the basic ideas and the inherent links between these algorithms. By studying these algorithms, we found, as a common theme, a better algorithm can often be obtained by relaxing the artificial constraints or by introducing extra degrees of freedom. The literature survey also reveals a fact that the status quo of algorithm evaluation is far from satisfactory. The research community urgently needs more systematic and consistent evaluations for existing algorithms to enable comprehensive and unbiased performance comparison. It is ideal to set up a benchmark and a common framework for impartial algorithm evaluation. As an effort toward this direction, this paper summarizes our project, in which we propose several solid criteria as key performance indicators, provide common filter sets with different characteristics, implement many representative algorithms, and evaluate them under the fundamental and consistent criteria. We describe the project details in Sections 3 and 4. An example is presented in Section 5. Finally, we summarize our contributions and discuss the future work in Section 6.

2 High-Level Review

As mentioned before, packet classification problem can be solved by either algorithms or TCAM hardware.

2.1 Algorithmic Solutions

The algorithmic solutions use commodity memory components (SRAM or DRAM) as the storage media for the algorithm data structure. As a general design rule, an algorithm always seeks to maximize the throughput and minimize the storage.

We can map the packet classification problem to the point location problem in a multidimensional space, where each relevant header field is treated as a dimension. In this space, each filter defines a hyper-cube and a packet corresponds to a point. The goal of packet classification is to determine the highest priority hyper-cube that covers a given point. Point location problem is a classical and fundamental topic in computational geometry. It has proven to be difficult with poor worst-case bound [1]. Assume there are n filters and F dimensions. It was showed that, in order to achieve $O(\log n)$ lookup time, the required storage can be as large as $O(n^F)$; on the other extreme, if the storage is limited to $O(n)$, a lookup may take $O(\log^{F-1} n)$ time to finish. Both cases are unacceptable in practice. Fortunately, the real filter sets often exhibit some characteristics that allow the filters to be organized in more efficient data structures and allow the packets to be classified much faster, using some heuristics.

2.2 Insights on Packet Classification Algorithms

An excellent survey of the art of packet classification techniques can be found in [2]. In this section, we identify certain high-level characteristics of the algorithmic approaches, with the objective of developing insights that can guide future algorithm improvements and developments.

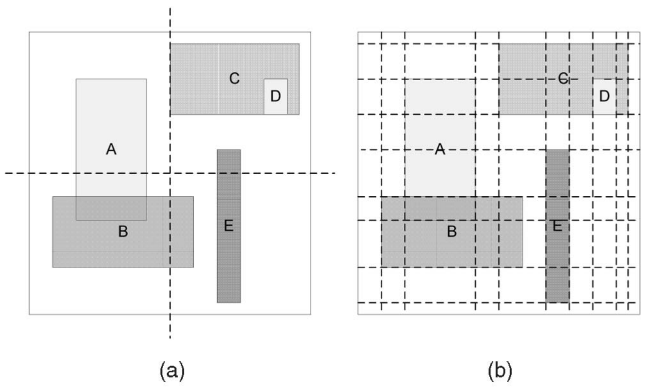


Fig. 1. (a) Cutting and (b) Projecting.

2.2.1 *One Theme—Space and Time Tradeoff*

The well-designed algorithms exhibit a clear tradeoff between storage and throughput. Algorithms without tunable parameters often perform poorly. For example, the naive form of the cross-producting algorithm [3] suffers from poor storage efficiency; so it fails to scale to even moderate-sized filter sets. In contrast, the Recursive Flow Classification (RFC) algorithm [4], a variation of the cross-producting algorithm in essence, trades off the throughput in order to reduce the storage to some extent. While still maintaining the high throughput, the storage is significantly reduced. However, for larger filter sets, the storage efficiency of RFC remains low. This suggests additional tradeoffs need to be realized to obtain better performance.

2.2.2 *Two Strategies—Cutting and Projecting*

The geometric view of packet classification exposes some basic ideas on how to construct the data structures and how to represent packet filters. In the geometric view, many algorithms adopt the strategy of cutting or projecting in the multidimensional space to preprocess the filter sets. Fig. 1 illustrates both strategies on a 2D plane. The cutting strategy cuts a space region into smaller subregions at selected vantage points. Each subregion, therefore, contains fewer hyper-cubes (note that some hyper-cubes are divided into multiple subregions which they overlap). This process recursively narrows down the search scope.

The second strategy projects the hyper-cubes to each dimensional axis. Two adjacent projecting points on an axis define an elementary interval that is fully covered by a unique subset of the hyper-cubes (filters). Identifying the elementary intervals that a packet belongs to helps to narrow down the search scope too.

Projecting has a finer granularity than cutting, so it can differentiate the filters better; however, locating an elementary interval from projecting is more difficult than locating a subregion from cutting.

The decision tree-based algorithms usually apply the cutting strategy and the decomposition-based algorithms often apply the projecting strategy.

2.2.3 Three Techniques—Splitting, Intersecting, and Grouping

The goal of packet classification is to find the best matching filter for a given packet header. The initial filter set is too large to be handled efficiently under limited storage or time, so the basic strategy is nothing more than "divide and conquer." The filter set is partitioned and regrouped, so that a packet can quickly identify a reduced filter set that includes the matching filter. The filter set reduction is achieved by "eliminating" the filters that are not needed for finding the final match.

The first concrete technique is *filter set splitting*. It splits the filter set into some smaller subsets based on the value of a few carefully selected header bits. Some other header bits are then used to continue splitting each subset. A decision tree is formed from this recursive process. Algorithms using this approach include Woo's modular packet classification [5], Hierarchical Intelligent Cuttings (HiCuts) [6], and Multidimensional Cuttings (HyperCuts) [7].

The second technique is *filter set intersecting*. The key idea of this technique is that it is easier to match a partial filter than to match the entire filter all at once. If we split the packet header into a set of substrings, then each substring can match a subset of filters. The intersection of these subsets is exactly the filters matching the entire packet header. Intersection can be implemented with different tradeoffs of storage and throughput. For example, we can intersect all subsets obtained from the substring lookups in a single step. The Bit Vector (BV) algorithm [8] and the Aggregated Bit Vector (ABV) algorithm [9] explicitly represent the subset of filters for each partial match by using bit vectors. In contrast, the cross-producting algorithm [3] implicitly encodes the subsets of filters into indices that are used to form the keys to the cross-product table. These algorithms are very fast and their throughput mainly depends on the speed of partial header lookups; however, they can consume excessive amounts of memory. On the contrary, the RFC algorithm [4] and the Distributed Cross-producting of Field Labels (DCFL) algorithm [10] provide a much nicer tradeoff of storage and throughput by recursively performing parallel set intersections in multiple steps. The Fat Inverted Segment Trees (FIST) algorithm [11] uses a similar approach but performs the intersections in sequence.

The partial header lookup can be done using different methods. The fastest method is to use a direct lookup table, yet it also consumes the most storage. Instead, we can use any well-established single-field lookup techniques, such as binary search and longest prefix matching [12], [13], [14], [15].

The last and least used technique is *filter set grouping*. Filters in a set are regrouped into disjoint subsets according to certain common features. Lookups can be performed in each of these smaller subsets in parallel. The best match is determined from the results of all the lookups. Tuple Space Search (TSS) [16], in which filters are grouped based on a tuple specification, belongs to this category. Lookups in each tuple can be conducted through a simple hash table. The 2D Compressed Tuple Space Search algorithm [17] also uses this basic technique.

Each of these techniques has its own limitations. It appears that in order to achieve the consistently good performance, the algorithm designer needs to combine the best characteristics of different approaches and to make good use of the time-space tradeoff. While attempting to find new algorithms from totally different perspectives seems unlikely, a systematic analysis of the existing algorithms can lead to significant performance improvements. Here, we discuss two possible research directions briefly.

One problem with *filter set splitting* is that some filters are too similar to be efficiently separated. But one can use *filter set grouping* to group the filters based on some sort of "similarity" measurement, so that the filters in a single subset exhibit the maximum dissimilarity. Now applying *filter set splitting* to each subset can be more efficient.

The problem of *filter set intersecting* is its excessive memory consumption, which is partially due to a large number of elementary intervals [11] or equivalence classes [4]. We can reduce the number by aggregating the elementary intervals. This coarser granularity leads to much smaller cross-product tables. It can also speed up the single field lookups at the cost of a small linear search in the final intersection step. For example, in the RFC algorithm [4], the first-level lookup tables take very large storage space. By slightly sacrificing the throughput, one can construct a more efficient data structure to reduce the table size significantly.

To summarize, there are still plenty of opportunities to improve the algorithm performance once the existing algorithms are fully understood.

2.3 TCAM Solution

Ternary Content Addressable Memory chips (TCAMs) are widely deployed in high-end network routers for packet classification because of their unmatched lookup throughput. A TCAM chip is a special memory device, which can store ternary bit strings and perform parallel searches on all of its entries simultaneously. In TCAMs, the packet filters are represented as ternary bit strings and are stored in decreasing priority order. Given a packet header, the search for the best matching filter is performed on all the entries in parallel. The index of the first matching filter is then used as the key to access a table to retrieve the associated data for the matching filter. This elegant architecture allows packets to be classified very fast. A commercial TCAM chip can store more than 100K ternary filters, large enough even for the largest filter set applied today. It can classify 250+ million packets per second, exceeding the throughput demands of all the existing networks today [18]. The room for algorithmic solutions to compete with TCAMs seems very narrow.

While TCAMs remain the most popular choice for high performance packet classification in network routers, the research on algorithmic alternatives for packet classification is still going on because of the following drawbacks of TCAM devices:

- *Low density and high cost.* A TCAM chip uses 16 transistors to store a bit, while an SRAM chip uses just six and a DRAM chips uses just 1. Consequently, the storage density of a TCAM chip is significantly lower than that of commodity memory chips. In addition, the relatively small market for TCAMs makes them very expensive: The cost per bit of TCAMs is roughly 20 times more than that of SRAMs and hundreds of times more than that of DRAMs.

- *High power consumption.* Because TCAMs search all entries in parallel for every packet, power consumption is a big issue. Twenty-five Watts is a fairly typical power budget for a TCAM device in a high performance application. Modern TCAMs allow the entries to be grouped into segments that can be selectively powered up and searched in order to reduce the power consumption. When the filters are partitioned into the segments appropriately, the power consumption can be significantly reduced [19], [20], [21]. However, these hybrid solutions actually lower the system throughput and impair the generality of TCAMs.
- *Poor arbitrary range support.* TCAMs can only search on ternary bit strings. This is not ideal for packet filters that include arbitrary ranges specifications on some fields. The easiest way to solve this problem is to convert each filter with range specifications into a set of filters defined by ternary bit strings only (i.e., each range can be converted to a series of nonoverlapping prefixes). Unfortunately, this conversion can lead to significant space expansion. An original filter can become as many as 900 expanded filters in the worst case [20]. This inefficiency has motivated the development of new algorithms that combine single field searches with encoded range values [22], [23], [24] and proposals for direct hardware support for range lookups [20]. Again, these hybrid solutions tend to lower the system throughput and impair the generality of TCAMs.
- *Poor multiple-match support.* Many recent network security applications, such as network intrusion detection and prevention, require all the matching filters to be reported rather than the highest-priority matching filter. However, the conventional TCAMs can only output the matching filter with the smallest index. It is likely that the future TCAMs, driven by new application requirements, will be designed to support multiple matches. Currently, the temporary remedies dealing with this issue are discussed in [25], [26], [27], [23]. In contrast, almost all the algorithmic solutions naturally support multiple-match applications.

Our research focuses primarily on the algorithmic solutions for packet classification, aiming to promote better design and evaluation standards for packet classification systems and to accelerate the innovation process in this area by providing an open-source platform.

3 Challenges

Although many packet classification algorithms and architectures have been proposed and the research in this area is still active, researchers and technology adopters find it difficult to choose an appropriate algorithm for a particular application and to evaluate new algorithms consistently and objectively. Before exploring new possibilities, it is imperative to understand the existing algorithms under uniform test conditions and a common set of benchmark criteria. Unfortunately, there is no such a platform available today, and all we can do is to read each individual paper. While this step is necessary, we often face the following challenges during this process:

3.1 Incommensurable Evaluation Results

First, the performance evaluations by different authors are not based on the same filter sets. This is partially because the researchers have limited accessibility to real-world filter sets. Sometimes they have to use randomly generated filter sets for evaluations. However, the performance of packet classification algorithms is very sensitive to the structure of the filter sets. Second, the performance evaluations are not based on the common implementation assumption and do not share a common implementation model. Some algorithms assume a software-based implementation and the others assume a hardware-based implementation. The different assumptions on implementation architectures and platforms can lead to very different performance evaluation results. Third, there is an absence of widely accepted tools, benchmarks, and metrics for packet classification algorithm evaluation. Different people have their own understanding and selection of the evaluation criteria. Some criteria are idealistic and make it difficult to determine the actual performance one might expect in practice.

3.2 Irreproducible Implementations and Evaluation Results

Researchers often fail to provide enough details to allow readers to exactly reproduce the work reported in their research papers. Either some key points of the algorithm description are missing or the test conditions, such as parameter settings and the filter sets used, are undisclosed. These situations cause confusion and create unnecessary hurdles for others trying to understand the algorithms and to advance the research. Even if enough details can be acquired from the publications, reproducing the previous work can still be a time-consuming task, impeding the speed of innovation.

3.3 Incomplete Evaluation and Unconvincing Results

The packet classification algorithms often involve some tradeoffs, heuristics, and optimizations. The tunable parameters may have subtle effects on algorithm performance. It is, therefore, important to isolate them and evaluate their behavior carefully in order to clarify their impact on the algorithm performance. However, some evaluations fail to identify the performance impact of individual parameters. The incomplete evaluation hampers the implementer's confidence to adopt the algorithm and also makes the thorough and fair comparisons difficult.

3.4 Inadequate Insights

The research papers discussing new algorithms often focus on the algorithm details. It is up to the readers to figure out the high-level insights that reveal the inherent and intrinsic principles underlying the algorithms. Deeper understanding of the problem is needed to enable more effective algorithm design efforts.

Having listed these challenges, we intend to improve the situation by setting up a platform accessible by all researchers and implementers.

4 Our Approach

Ideally, the evaluation should cover many aspects of the algortihm, including the criteria of throughput, storage,

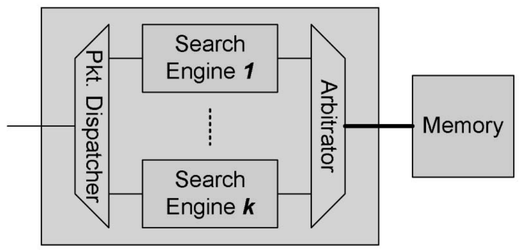


Fig. 2. The model for the implementation of packet classification algorithms.

incremental update support, preprocessing time, scalability to the size of filter sets, adaptability to the structure of filter sets, implementation cost, and power dissipation. More important, all the evaluation results should be normalized in a directly comparable way. It is understandable that in different applications some performance criteria may be more critical than the others, but the evaluation itself should provide information without preference and let readers make their own judgments. While asymptotic analysis of timing and storage complexity is a useful metric, the evaluation should not be limited to it. This is because packet classification algorithms are mostly based on heuristics. Even for a same algorithm, different filter sets with different structures and sizes tend to give very different results. The performance of an algorithm on real filter sets is the decisive factor in any realistic evaluation to deliver useful information.

Aiming to eliminate the inconsistency in the algorithm evaluations for packet classification, we establish a standard procedure of algorithm description and evaluation. In particular, we provide the research community the objective and "advocacy-free" evaluation for a suite of representative packet classification algorithms. A summary of our approach is as follows:

4.1 Documenting the Method

First, we provide a complete description of the key data structures and all the tunable parameters used in the algorithm. Second, we provide a detailed description of the algorithm preprocessing and lookup process along with step-by-step illustrations using an example. Third, we provide the source code for an actual implementation.

After carefully considering the mostly used application scenarios for packet classification algorithms, we assume that a simple hardware-based model based on ASICs, FPGAs, or Network Processors is used in our implementation, which includes one or multiple on-chip lookup engines or threads, a common memory interface, and commodity off-chip memory chips, as shown in Fig. 2. All data structures are stored in the off-chip memory. The lookup for one packet is conducted by a sequence of dependent memory accesses initiated from one of the search engines. The memory bandwidth is shared by multiple independent lookup engines (hardware blocks or software threads). We assume a perfect sharing scheme, so no memory bandwidth is wasted.

The on-chip resource is consumed by the hardware logic to implement the search engines and some other facilitated circuits including the packet dispatcher and the memory interface. Given the current hardware technology, this part is considered very low cost. A $10\ \mathrm{mm} \times 10\ \mathrm{mm}$ die at the standard 65 nm technology offers more than 100 million gates, while a search engine consumes no more than a few thousand gates on the average (Once the data structure is set up, the actual searches are done through a series of very simple operations, such as constructing the search key and accessing the data structure). Memory incurs the most cost of the packet classification system. Moreover, since we can afford multiple parallel search engines at low cost, the throughput performance bottleneck lies on the memory interface which, in turn, is determined by the available memory technology. Memory also consumes the most of the power in the system. For these reasons, we only evaluate the memory consumption, which incurs the most cost, and the memory bandwidth consumption, which determines the overall system throughput while ignoring the cost and impact of the other components in the system. We believe this simplified model captures the essence of the performance evaluation for packet classification algorithms.

We also try to categorize the algorithms based on their high-level ideas and provide insights to help to improve the existing algorithms or design new algorithms.

4.2 Documenting the Filter Sets

Since we cannot access many operational filter sets in production networks, and even if we can, we are not allowed to publish them due to privacy and security concerns, we have to resort to the synthetic filter set. The open-source *ClassBench* [28] is used to generate several synthetic filter sets with different scales and structures. We provide the parameters used for the filter set generation. We also generate a packet header trace using *ClassBench* for each filter set. The packet header traces are used to verify the algorithm implementation and to evaluate the algorithm performance. The size of each trace is about 10 times larger than the corresponding filter set.

We also provide the original filter sets that are used as seeds for reference. The statistics files extracted from these original filter sets can be downloaded from the *ClassBench* website.

All evaluations we have conducted use the same filter sets. This is the first step to guarantee the algorithm commensurability.

4.3 Essential Evaluation Metrics

Based on our memory-centric model, the key performance measure can be directly derived and inferred from the algorithm data structure built for a particular filter set.

We measure the storage consumption of an algorithm using *the average number of bytes consumed per filter*. This measurement is essential to understand the overall memory efficiency. It avoids two main drawbacks of the previous practices: 1) one algorithm may involve multiple data structures and tables, but the evaluation fails to cover all of them; 2) only the overall memory consumption is reported, but the size of the filter set applied is unknown.

We can measure the throughput of an algorithm using the memory bandwidth consumption: *the number of bytes per memory access* and *the number of dependent memory accesses per packet lookup*. However, to make the evaluation results directly comparable, we actually use the product of the above two numbers as the measurement of the bandwidth

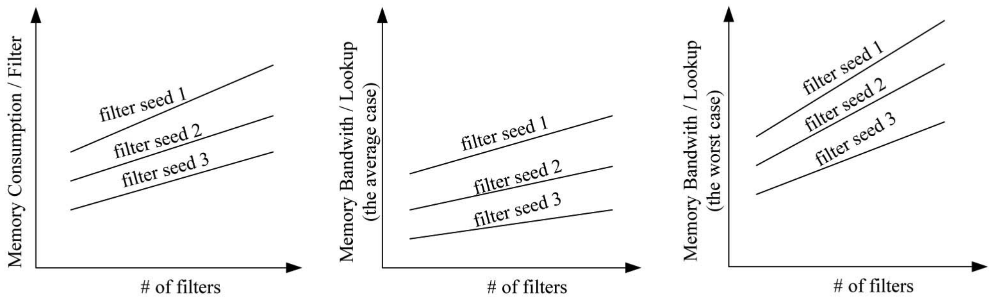


Fig. 3. Presentation of evaluation results.

consumption. This measurement is essential to understand the achievable system throughput. It eliminates the difficulties to directly compare algorithms. For example, some algorithms assume a parallel table lookup architecture (e.g., RFC [4]) and some others assume a sequential table lookup architecture (e.g., HyperCuts [7]), some algorithms use very wide lookup words (e.g., BV [8]) and some others use narrow lookup words (e.g., Tuple Space Search [16]). Our measurement reflects the actual throughput regardless of these differences. The memory bandwidth consumption is evaluated in both the worst case and the average case.

We use three figures like those shown in Fig. 3 to present the results. Given these measurements, the overall storage is simply the product of the memory consumption per filter and the number of filters. The overall throughput can be calculated by simply dividing the total memory bandwidth by the memory bandwidth consumed per packet lookup. Some other related metrics that readers might be interested in (e.g., the tree depth in the decision tree-based algorithms), can be easily obtained by running the algorithm implementations.

Without an accurate power model and the real implementation, it is very difficult to measure the absolute power consumption of an algorithm. Fortunately, since the memory consumes the most of the power, once we know the memory and the memory bandwidth consumption of an algorithm, the relative power consumption can be derived.

Another important aspect of the algorithm evaluation is about the cost and speed of filter set preprocessing and incremental updates. In our evaluations, we found that the preprocessing time varies drastically, depending on the filter set size and structure, the design parameters, and the performance target. The time can range from milliseconds to even minutes. With the algorithm implementations provided, users can easily test the preprocessing time based on their filter sets. The more important issue is about the incremental updates. Some decomposition-based algorithms, such as RFC [4] and BV [8] do not support incremental updates in most cases. This means they only work on the static filter sets. Anytime a new filter needs to be added (or an old filter needs to be removed), the data structures of these algorithms have to be rebuilt from scratch. On the other hand, the decision tree-based algorithms, such as HiCuts [6] and HyperCuts [7], support incremental updates. However, the resulting data structure will become suboptimal after updates, which will eventually lead to performance deterioration. It is necessary to rebuild the data structure once the performance becomes unacceptable. The incremental update speed can also be derived from the implementations provided. Such information is clearly documented, since it might be critical for some applications.

4.4 Sensitivity Study

We determine how each individual parameter influences the overall performance quantitatively in our algorithm evaluations. For each tunable parameter, we produce some figures like that shown in Fig. 4. Each figure uses a differently scaled filter set. The sensitivity study clarifies the issues often left unresolved in the original papers. It also helps users to determine the optimal design parameters for a given filter set.

The documentations of the algorithms and the evaluation results are posted on a publicly accessible website: www.arl.wustl.edu/~hs1/PClassEval.html.

Note that our reference implementations are only for the purpose of simulation and evaluation; thus, the source code is not optimized for software execution and the implementations do not directly map to either ASICs/FPGAs or network processors. Our effort will help to understand some representative algorithms better, to promote the standard for impartial and consistent algorithm evaluations, to ease the research curve, and to encourage external contributions from the entire research community.

5 Summary and Example Illustrated

5.1 Evaluated Algorithms

Six representative algorithms have been implemented and evaluated. They are HiCuts [6], HyperCuts [7], Woo's Modular Packet Classification [5], RFC [4], BV [8], and the Tuple Space Search algorithm [16]. These algorithms set up the foundations of this field and cover all the basic techniques, we have discussed in Section 2. New algorithms mainly seek to improve these classical algorithms.

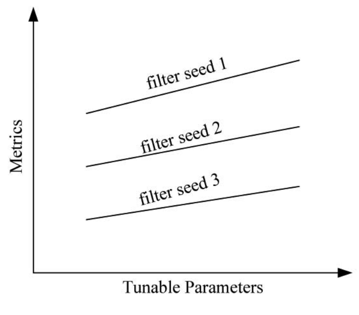


Fig. 4. Effect of parameter settings.

The difficulty we encountered during the algorithm implementations is mainly due to the ambiguous and incomplete algorithm descriptions in the original papers. Since the algorithms depend heavily on heuristics and the performance of the heuristics is very sensitive to the filter set structure, some algorithms describe several possible implementation options or just high-level guidelines for implementations. Therefore, it is up to the users to determine the appropriate implementation details for their applications; so several rounds of trial and error are needed. During this course, we found that sometimes the details are hard to determine; however, getting them right is crucial for the overall performance.

When implementing the HiCuts algorithm [6], we were troubled by the heuristics for choosing the dimension to cut. Although four possible strategies are listed, it is unclear which one is the most effective. Our experiments show that this decision is crucial. Random choice may cause the algorithm to degrade its performance significantly or even to fail to work. For example, the option that choosing the dimension that can maximize the entropy of filter distribution does not work in case all the remaining filters span the entire region on some dimensions. No matter how many cuts are performed on these dimensions, they always return the maximum entropy. However, cutting on any of these dimensions are useless for filter set separation. The only effect is to duplicate the same set of filters into multiple subregions. Similar issue happens in other algorithms such as HyperCuts [7]. While the dimension choosing strategy is clear in this algorithm, the decisions about the number of cuts and their distribution among the chosen dimensions are open-ended. In RFC [4], the issue on how exactly the reduction tree should be shaped is left undetermined. However, a poor decision can negatively impact the memory efficiency.

The ambiguity in algorithm details makes the algorithm evaluation and comparison difficult. In our implementation, we allow users to freely configure the parameters. To compare the algorithms under a particular filter set, we choose the configurations that lead to the best overall performance.

5.2 Filter Sets

In our evaluations, we use three parameter files to generate the synthetic filter sets with variable sizes of 100 to 10K filters. The parameter files are:

acl1: Extracted from a real-world Access Control List (ACL) filter set with 733 filters. In this filter set, the source and destination IP prefix specifications are quite specific. The destination port specification can be exact value (in most cases) arbitrary range, or wild-card. All the source port specifications are wild-card, so essentially the filters have only four dimensions.

ipc1: Extracted form a real-world IP Chain (IPC) filter set with 1,702 filters. The structure of this filter set is similar to that of the ACL filter set, except that the source ports also contain exact values or arbitrary range specifications.

fw1: Extracted from a real-world firewall filter set with 283 filters. In this filter set, a large number of filters contain the wild-card specification in one or more fields.

The detailed characteristics of these parameter files can be found in [29].

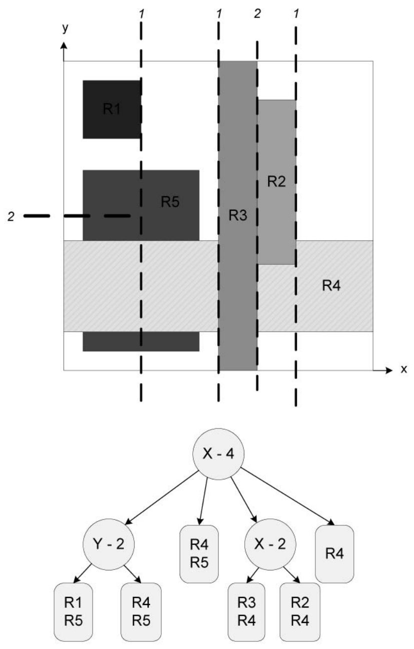


Fig. 5. HiCuts algorithm illustration.

5.3 Example Illustrated—The HiCuts Algorithm

As an example to show what we have achieved, we provide the documenting and evaluation results of HiCuts [6]. Readers can access the project website for more information and the evaluation results of other algorithms.

HiCuts is the first decision tree-based packet classification algorithm. It takes the geometric view of the packet classification problem. We consider a k-dimensional space, where k is the number of header fields involved in packet filters. With such an abstraction, each filter defines a hyper cube and each packet header defines a point in this k-dimensional space. The decision-tree construction algorithm recursively cuts the space into smaller subregions. Only one dimension is cut at each step. Each subregion ends up containing fewer hyper cubes. The cutting continues until each subregion contains few enough filters that allow a low cost linear search to find the best matching filter.

The lookup algorithm is straightforward. Based on the value of the packet header, the algorithm follows the cutting sequence to locate the target subregion (i.e., a leaf node in the decision tree) and then, performs a linear search on the hyper-cubes that overlap this subregion.

Fig. 5 illustrates the decision-tree construction for a 2D filter set. On the plane are five rectangles. Each rectangle represents a filter. At the first step, we cut along the x-axis to generate four subregions. At the following steps, we choose two of these subregions to cut along the y-axis and x-axis, respectively. Now each subregion overlaps ≤ 2 rectangles. If we decide it is affordable to do a linear search on at most two filters, we can stop cutting the space further. The resulting decision tree is also shown in Fig. 5.

The number of decision tree nodes and the number of stored filters (filters can be duplicated if crossing the cutting boundaries) determine the storage of the algorithm data structure. The depth of the decision tree and the number of filters in the leaf nodes determine the worst-case lookup throughput. It is difficult to find the globally optimal decision tree, so in practice the algorithm uses some heuristics to make optimal local decisions and to trade off storage and throughput. Gupta and McKeown [6] introduce the following configurable parameters to control the tradeoff.

5.3.1 Configurable Parameters

The number of cuts at each node. Intuitively, the more cuts at each node are conducted, the fatter and shorter the resulting decision tree will become. However, the storage penalty due to the high degree of the decision tree can easily overwhelm the throughput gain. Therefore, we choose a suitable number of cuts np at each internal decision tree node r. np is dynamically determined by the local cutting situation and a global configurable space measure factor, *spmf*. We choose the largest possible np as long as the following inequation is satisfied and it does not exceed the design limitation:

$$\begin{aligned} &spmf*(\#\ filters\ at\ r)\\ &\quad \geq \sum (\#\ filters\ at\ each\ child\ of\ r) + np. \end{aligned}$$

For the convenience of implementation, the chosen value of np is always the power of two. Different configurations of $spmf$ need to be tested to determine the best one.

The dimension chosen to cut at each node. In addition to the number of cuts, the dimension to cut at each internal decision tree node r is also critical to the algorithm performance. The algorithm provides four options in this regard. No one is consistently better than the others for all filter sets. Their effectiveness can only be tested through experiments. First, for each of the dimensions, we calculate the largest number of cuts and record the statistics of filter distribution in the child nodes, then we test the following four options to choose the best one:

1. *option 0*: Find the largest number of filters, n_i, in a child node for each field. Choose the dimension that gives the smallest n_i.
2. *option 1*: Assume node r contains n filters and a child node of r contains n_i filters. Let n_i/n be a probability distribution of np elements. Choose the dimension that gives the largest entropy of the distribution.
3. *option 2*: Choose the dimension that results in the smallest

$$\sum \#\ filters\ at\ each\ child\ of\ r + np.$$

4. *option 3*: Choose the dimension that has the largest number of distinct range specifications.

The bucket size at leaf nodes. The bucket size is defined as the maximum number of filters allowed in a leaf node. It is used to determine as to when we can terminate the decision tree construction. A larger bucket size can help to reduce the size and depth of a decision tree, but it can induce a longer linear search time. A smaller bucket size has the counter-effects. Experiments are needed to determine the appropriate bucket size for the best tradeoff of storage and throughput.

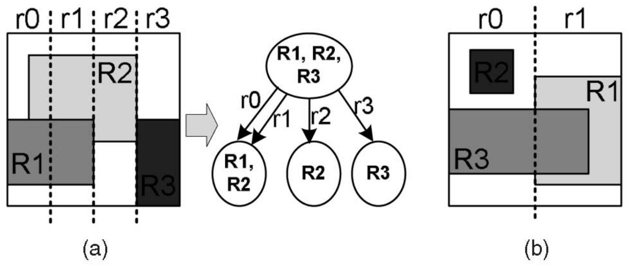


Fig. 6. (a) Child node reuse optimization. (b) Redundancy elimination optimization.

5.3.2 Algorithm Optimizations

Several algorithm optimizations were introduced in [6]:

Child node reuse. After a number of cuts are performed along a dimension, many child nodes may contain an identical set of filters. In such a case, we can avoid storing these child nodes separately. Instead, we use a pointer to point to a common child node, which is shared by all such child nodes. This optimization has two implications. First, a pointer must be maintained for each potential child node. Second, each node must keep its region boundary explicitly. Both can result in a larger node size. Therefore, the optimization must be carefully evaluated to see if it can actually lead to a more compact storage. A misuse can incur the storage penalty instead.

In Fig. 6a, the first two subregions r_0 and r_1 have the same set of filters, $\{R_1, R_2\}$, so only one child node is generated.

Redundancy elimination. After a sequence of cuttings are performed, the portion of a hypercube in a subregion might be fully covered by another hypercube with a higher priority. The corresponding filter at this decision tree node is therefore redundant, so it can be removed to save the storage.

In Fig. 6b, if R_1's priority is higher than R_3's, then in the subregion r_1, R_3 becomes redundant, so it can be removed. However, R_1 and R_3 should coexist in the subregion r_0, since they are only partially overlapped.

5.3.3 Storage Efficiency and Scalability of Storage and Throughput

In our experiments, we set the bucket size to 16, the space measure factor to two, and the dimension choosing option to three. Fig. 7 shows the results. The algorithm consistently demonstrates better performance and scalability on the *acl1* filter sets than on the other types of filter sets. The *fw1* filter sets result in the poorest overall performance. When the number of filters exceeds 1K for *fw1* or 5K for *ipc1*, the storage becomes unacceptable, so the data points are not shown in the figures.

5.3.4 Sensitivity Study

We use the filter sets *acl1-10K*, *ipc1-1K*, and *fw1-100* to evaluate the algorithm sensitivity to the configurable parameters.

Sensitivity to the space measure factor. In this simulation, we set the bucket size to 16 and the dimension choosing option to three. The results are shown in Fig. 8. A larger space measure factor means larger storage and better throughput. *acl1-10K* and *ipc1-5K* show the similar performance while *fw1-100* is much worse in terms of the storage, even there are only about 100 filters in the filter set.

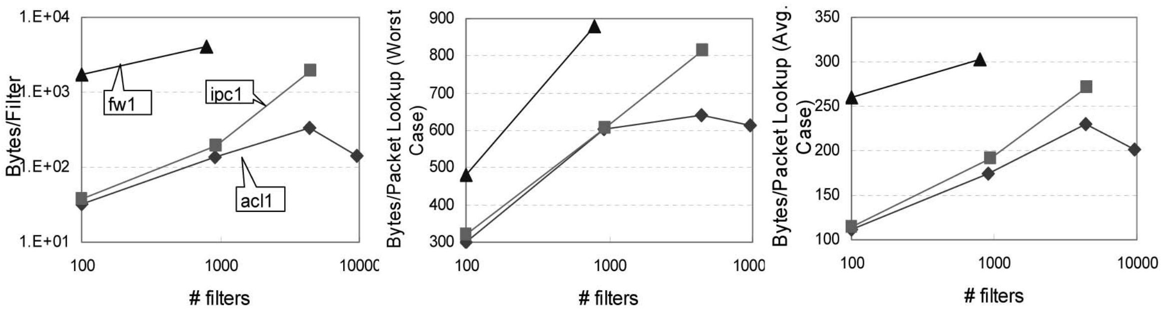


Fig. 7. HiCuts performance evaluation.

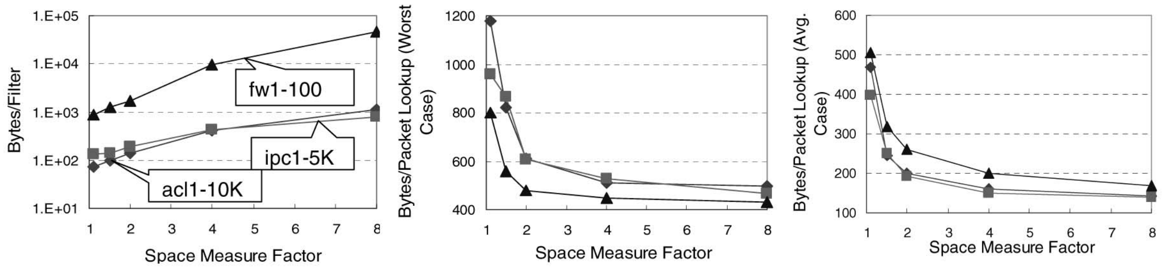


Fig. 8. Sensitivity to space measure factor.

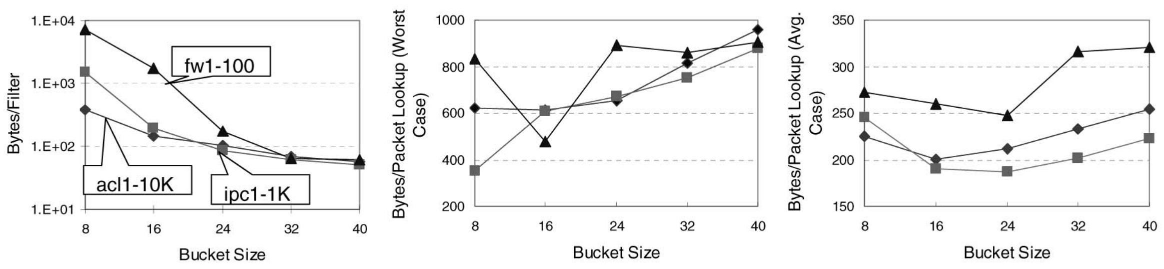


Fig. 9. Sensitivity to bucket size.

Sensitivity to the bucket size. In this simulation, we set the space measure factor to two and the dimension choosing option to three. The results are shown in Fig. 9. The storage decreases monotonously as the bucket size increases. Generally, a larger bucket size means a worse lookup throughput but this does not always hold.

5.3.5 *Preprocessing and Incremental Updates*

The preprocessing time is collected on a PC with a 2.93 GHz Intel Q6800 CPU and 3 GB RAM. Fig. 10a shows the preprocessing time in seconds for different types of filter sets and for different number of filters. In this simulation, the bucket size is fixed to 16 and the dimension choosing option is set to three. The time to process the firewall filter set increases abruptly as the number of filters grows. Fig. 10b shows the preprocessing time in seconds for the ACL filter sets with different bucket size and different number of filters. Smaller bucket size requires significantly longer time to process.

The HiCuts algorithm supports incremental updates to some extent. To insert (or remove) a filter, we can just walk the decision tree similar to the lookup process using the filter specification. However, since the hyper cube specified by a filter may overlap multiple subregions defined by the decision tree leave nodes, the process requires to update all the affected leave nodes. This can be time-consuming for less-specified filters. Moreover, such updates lead to a suboptimal decision tree with deteriorate performance. The algorithm implementers need evaluate the impact of the preprocessing and incremental update cost for their particular applications.

5.4 Observations

The overall evaluation results are consistent with our expectations. However, some interesting behaviors do stand out, which are not discussed in the original papers.

First, the performance of packet classification algorithms on the firewall filter sets is generally poor, even if the size is just moderate. This is especially true for the decision tree-based algorithms. For example, given a firewall filter set with

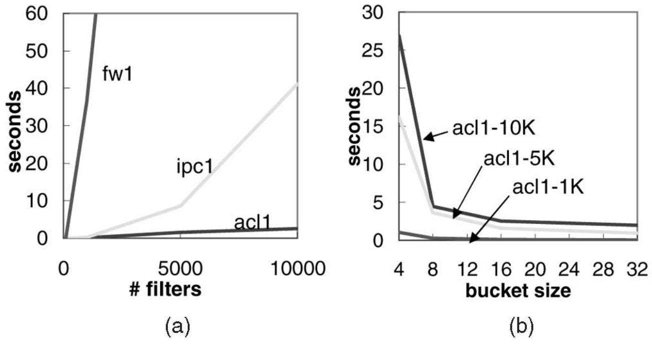


Fig. 10. Preprocessing time.

only 100 filters, the HiCuts algorithm [6] uses more than 1,000 bytes to store a filter on the average and needs to access about 500 bytes to classify a packet in the worst case. The main reason is that each field of the filter tends to cover a wide range of values. In consequence the filters are heavily overlapped and less distinguishable. Fortunately, the number of unique ranges or prefixes on each field is not too large so the decomposition-based algorithms, such as RFC [4] and BV [8] work relatively better on this type of filter sets.

Second, although the decomposition-based algorithms can be very fast, their memory efficiency is relatively poor. We observe excessive memory consumption for some moderate sized filter sets. For example, given an IPC filter set with about 1,000 filters, the RFC algorithm [4] needs 40,000 bytes to store a filter on the average. The preprocessing algorithm can exhaust the system memory when working on the filter sets with just a few thousands of filters.

Third, although the decision tree-based algorithms allow a nice tradeoff between storage and throughput, the overall performance is not very promising. More study needs to be done in at least two areas: 1) find more systematical ways to fine-tune the configurable parameters, and 2) find better adaptive decision-tree construction procedures to fit the target filter set structure.

5.5 Algorithm Comparisons

When our evaluation metrics are applied, comparing different algorithms becomes straightforward. For example, it was difficult to compare the HiCuts algorithm [6] and the Tuple Space Search algorithm [16] before because they are based on very different design paradigms. However, from our evaluation results published on the website, www.arl.wustl.edu~hs1/PClassEval.html, we can easily draw the following conclusions quantitatively: 1) TSS is much more memory efficient than HiCuts; while larger filter sets typically result in worse storage efficiency for HiCuts, the trend is opposite for TSS in many cases. 2) The worst-case throughput and the average-case throughput of TSS are very similar; while HiCuts's worst-case throughput is worse than TSS, its average-case throughput is better than TSS. 3) HiCuts handles the firewall filter set better than TSS in terms of throughput. However, the storage of HiCuts does not scale to large firewall filter sets. Based on these observations, algorithm implementers can easily decide the algorithm that best suits their specific applications.

6 Concluding Remarks

The project we described, focuses on the evaluation issues for high performance packet classification algorithms, which are needed to enable the routers, and switches to meet the QoS and security challenges in high-speed environments.

We first summarize the previous work from a high-level perspective and try to categorize the algorithms according to their basic approaches. This method has a clear advantage to help the researchers and designers to avoid digging into the algorithm details and to gain a clear sight of the big picture. Understanding the problem from a high-level perspective also provides insights that can lead to further improvements to the state of the art.

It is also important to understand the technical merit of each algorithm. We are often in a situation to ask as to which algorithm is indeed better from certain perspective, or if we can use one for a particular application with reasonable confidence. Unfortunately, such questions are difficult to answer just based on the published papers. In this paper, we describe an open-source project to mitigate this issue. Most of the representative algorithms are actually implemented based on the same model and assumptions. The freely available implementations allow others to easily reuse them for different purposes. We also enforce more consistent and fundamental performance measurement criteria for algorithm evaluation so that the algorithms can be directly compared. This project relieves researchers and designers from duplicating the previous work and helps them to quickly evaluate algorithms for any applications. By keeping this platform open, we also encourage external contributions of new algorithm implementations and evaluations. We hope they can be incrementally added to the library under the same framework. We believe this project will benefit the research and design community as a whole.

References

[1] M. Overmars and A. van der Stappen, "Range Searching and Point Location among Fat Objects," *J. Algorithms,* vol. 21, no. 3, pp. 629-656, 1996.

[2] D.E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," Technical Report, WUCSE-2004, Washington Univ., 2004.

[3] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching," *Proc. ACM SIGCOMM,* 1998.

[4] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. ACM SIGCOMM,* 1999.

[5] T.Y.C. Woo, "A Modular Approach to Packet Classification," *Proc. IEEE INFOCOM,* 2000.

[6] P. Gupta and N. McKeown, "Packet Classification Using Hierarchical Intelligent Cuttings," *Proc. IEEE Symp. High Performance Interconnects (HotI),* 1999.

[7] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," *Proc. ACM SIGCOMM,* 2003.

[8] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," *Proc. ACM SIGCOMM,* 1998.

[9] F. Baboescu and G. Varghese, "Scalable Packet Classification," *ACM SIGCOMM,* 2001.

[10] D. Taylor and J. Turner, "Scalable Packet Classification Using Distributed Crossproducting of Field Labels," *Proc. IEEE INFOCOM,* July 2005.

[11] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. IEEE INFOCOM,* 2000.

[12] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM,* 1997.

[13] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems,* vol. 17, pp. 1-40, Feb. 1999.

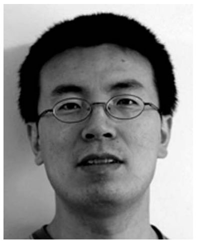
[14] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *ACM SIGCOMM Computer Comm. Rev.,* vol. 34, no. 2, pp. 97-122, Apr. 2004.

[15] H. Song, J. Turner, and J. Lockwood, "Shape Shifting Tries for Faster IP Lookup," *Proc. IEEE Int'l Conf. Network Protocols (ICNP '05),* 2005.

[16] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," *Proc. ACM SIGCOMM,* citeseer.ist.psu.edu/srinivasan99packet.html, 1999.

[17] H. Song, J. Turner, and S. Dharmapurikar, "Packet Classification Using Coarse-Grained Tuple Spaces," *Proc. ACM/IEEE Symp. Architecture for Networking and Comm. Systems (ANCS '06),* pp. 41-50, 2006.

[18] IDT Network Search Engines, http://www.idt.com/products, 2009.
[19] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," *Proc. IEEE INFOCOM,* 2003.
[20] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," *Proc. IEEE Int'l Conf. Network Protocols (ICNP),* 2003.
[21] K. Zheng, H. Che, Z. Wang, and B. Liu, "TCAM-Based Distributed Parallel Packet Classification Algorithm with Range-Matching Solutin," *Proc. IEEE INFOCOM,* 2005.
[22] H. Liu, "Efficient Mapping of Range Classifier into Ternary-CAM," *Proc. IEEE Symp. High Performance Interconnects (HotI),* Aug. 2002.
[23] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," *ACM SIGCOMM,* 2005.
[24] J. Lunteren and T. Engbersen, "Fast and Scalable Packet Classification," *IEEE J. Selected Areas in Comm.,* vol. 21, no. 4, pp. 560-571, May 2003.
[25] F. Yu and R.H. Katz, "Efficient Multi-Match Packet Classification with TCAM," *Proc. Hot Interconnects,* 2004.
[26] F. Yu, T. Lakshman, M.A. Motoyama, and R.H. Katz, "SSA: A Power and Memory Efficient Scheme to Multi-Match Packet Classification," *Proc. Symp. Architectures for Networking and Comm. Systems,* 2005.
[27] H. Song and J. Lockwood, "Efficient Packet Classification for Network Intrusion Detection Using FPGA," *Proc. Int'l Symp. Field-Programmable Gate Arrays (FPGA),* 2005.
[28] D.E. Taylor and J.S. Turner, "Classbench: A Packet Classification Benchmark," *Proc. IEEE INFOCOM,* 2005.
[29] ClassBench, http://www.arl.wustl.edu/~det3/ClassBench, 2005.



Haoyu Song received the BE degree in electronics engineering from Tsinghua University, in 1997, and the MS and DSc degrees in computer engineering from Washington University in St. Louis, in 2003 and 2006, respectively. He is currently a member of Technical Staff researcher at Bell Labs, Alcatel-Lucent. From 2002 to 2006, he was a research assistant at Applied Research Laboratory, Washington University. His research interests include network virtualization and cloud computing, high performance networked systems, algorithms for network packet processing and network intrusion detection, and ASIC/FPGA design and verification. He has published more than 20 peer-reviewed papers and has filed eight patents for his work on network packet processing and network virtualization. He is a member of the IEEE.

Jonathan S. Turner received the MS and PhD degrees in computer science from the Northwestern University, in 1979 and 1982, respectively. He holds the Barbara and Jerome Cox chair of Computer Science at Washington University, and also, he is the director of the Applied Research Laboratory. His research focuses on the design and analysis of high performance networks, and he has led a series of major systems projects over the years, that have demonstrated important innovations in high performance switching, scalable multicast, extensible routers, and network virtualization. He has graduated 21 PhD students and has served as chair of the Department of Computer Science and Engineering (1992-1997, 2007-2008). He was a member of the Technical Staff at Bell Labs (1977-1983), where he provided the technical leadership on an early project seeking to integrate voice and data communication using packet switching. He was a cofounder and the chief scientist of Growth Networks, a startup company that developed scalable switching components for Internet routers and ATM switches, before being acquired by Cisco Systems in early 2000. He received the Koji Kobayashi Computers and Communications Award from the IEEE, in 1994, and the IEEE Millenium Medal, in 2000. He has been awarded 30 patents for his work on switching systems and has many widely cited publications. He is a fellow of both the ACM and the IEEE, a member of the National Academy of Engineering, and a member of the board of the Computing Research Association.