A Memory-optimized Bloom Filter using An Additional Hashing Function

Mahmood Ahmadi and Stephan Wong
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
{mahmadi, stephan}@ce.et.tudelft.nl

***Abstract*— A Bloom filter is a simple space-efficient randomized data structure for the representation set of items in order to support membership queries. In recent years, Bloom filters have increased in popularity in database and networking applications. In this paper, we introduce a new extension to optimize memory utilization for regular Bloom filters, called Bloom filter with an additional hashing function (BFAH). The regular Bloom filter stores items from a set k times k memory locations that are determined by the k addresses stored in the bit-array structure. Which k addresses to utilize is determined by to which positions in the structure the k (regular) hashing functions are pointing to. Utilizing the additional hashing function, only one out of these k memory addresses is selected to store the item only once. Consequently, there is no longer needed to store the $k-1$ redundant copies. We implemented our approach in a software packet classifier based on tuple space search with the $H3$ class of universal hashing functions. Our results show that our approach is able to reduce the number of collisions when compared to a regular Bloom filter.**

Keywords: Bloom filter, network processor, universal hashing, packet classification.

I. Introduction

Bloom filters are frequently utilized in databases and networking applications, such as distributed databases, packet classification, packet inspection, forwarding, p2p networks, IP route lookup, and distributed web caching [5][7][8]. Most network devices, e.g., routers and firewalls, require the processing of incoming packets (e.g., classification and forwarding) at wire speeds. These devices mostly incorporate special network processors that are comprised of a processor core with several memory interfaces and special co-processors that are optimized for packet processing[1]. The gap between processor and memory performance has been a major source of concern for all of computing; this problem is exacerbated in packet processing systems. Such memory bottlenecks can be overcome by the following mechanisms: hiding of memory latencies through parallel processing and reducing the memory latencies by introducing an special memory architectures[11]. One approach to achieve higher lookup performance is to utilize the Bloom filter data structure that recently is utilized in embedded memory technology in network processors[14]. A version of Bloom filter that is specifically utilized by network processors and packet processing applications is the counting Bloom filter. In the regular and counting Bloom filters, for each input item, k hashing functions are utilized to lookup positions in the bit or counter array (respectively) that also contain the k memory addresses of k memory locations at which the item is stored. Therefore, in the regular and counting Bloom filters k copies of each item are kept, but only one copy is accessed and the other $(k-1)$ copies of items are never accessed. To optimize the memory in the regular Bloom filter the caching policy and for the counting Bloom filter the pruning technique was proposed[7][14]. The caching policy was exploited the Bloom filter properties and only decreased the number of accesses[2][7]. The pruning technique has some limitations as follows: high processing time due to incremental update and reconsidering all items for each inserted item, the pruning technique only works in conjunction with the counting Bloom filter that in turn can only be applied to a limited number of applications. In this paper we introduce an approach to eliminate the redundant items in the regular and counting Bloom filters that operates independently from the counters and bit-array in the counting and regular Bloom filters, respectively. In this approach, we utilize an additional hashing function to select one of the addresses pointed to the k hashing functions in the regular and counting Bloom filters. The utilization of an additional hashing function has the following advantages: decreasing memory redundancy in comparison to regular Bloom filters, increasing the performance (in term of reducing the number of collisions) in comparison to the regular Bloom filters and the simplicity to be implemented in hardware. This paper is organized as follows. Section II presents related work. Section III describes the regular and the (pruned) counting Bloom filters. Section IV describes the concept and architecture of Bloom filter with an additional hashing function (BFAH). Section V presents a software implementation and some benchmarking results. In Section VI, we draw the overall conclusions.

II. Related Work

In this section, we take a brief look at previous works regarding the Bloom filter and its memory organization. In [8][14], an extended version of the Bloom filter is considered. It presents a novel hash table architecture and lookup algorithm and converts a Bloom filter into a counting Bloom filter and associated hash bucket which improves the performance over a standard hash table by reducing the number of mem-

978-1-4244-2324-8/08/$25.00 © 2008 IEEE.

ory accesses needed for the most time-consuming lookups. It only works in conjunction with counting Bloom filters and needs to reconsider all of the items for each inserted item that consequently leads to longer processing time. In [2], a cache architecture for the counting Bloom filter to decrease the number of accesses to memory was proposed. It presented the analysis of the number of accesses and size of different levels of designed cache. Our approach introduces an architecture that decreases utilized memory in regular and counting Bloom filters. In the other techniques (memory optimization in the counting Bloom filter), the searching and insertion criteria are performed based on the value of counters that for each operation k (number of hashing functions) counters should be inspected. In our work, the output addresses of k hashing functions is selected using an additional hashing function that is performed independently from the counters or bit-array in counting or regular Bloom filters, respectively.

III. Bloom Filters

In this section, we present the regular and counting Bloom filters concepts, and afterward, describe the pruning procedure in the counting Bloom filters.

A. Regular Bloom Filter

A Bloom filter is a simple-space efficient randomized data structure for representing a set in order to support membership queries. Burton Bloom introduced Bloom filters in the 1970s [4]. A set S $(x_1, x_2, ..., x_n)$ of n items is represented by an array V of m bits that are initially all set to 0. A set of k independent hash functions h_1, h_2, ..., h_k (each with an output range between 1 and m) is utilized to set k bits in array V at positions $h_1(x)$, $h_2(x)$, ..., $h_k(x)$ for all x in set S. More precisely, for each item $x \in S$, the bits at positions $h_i(x)$ are set to 1 for $1 \leq i \leq k$. Moreover, a location can be set to 1 multiple times. To verify whether an item y is a member of the set S, the same set of hash functions is utilized to determine $h_i(y)$ (for $1 < i < k$) indicating the locations in array V to be checked whether their content is a 1. If one of these location yields a 0, y is certainly not a member of the set S. If all locations yield a 1, there is a high probability that y is a member of the set S (*positive*)[10][14]. An example of the regular Bloom filter is depicted in Figure 1 (A).

Figure 1 (A) depicts the creation of a regular and counting Bloom filters for a set of four items $R0$, $R1$, $R2$ and $R3$.

B. (Pruned) Counting Bloom Filter

The previously discussed Bloom filter works fine when the members of the set do not change over time. When they do, adding items requires little effort since it only requires hashing the additional item and setting the corresponding bit locations in the array. On the other hand, removing an item conceptually requires unsetting the ones in the array,

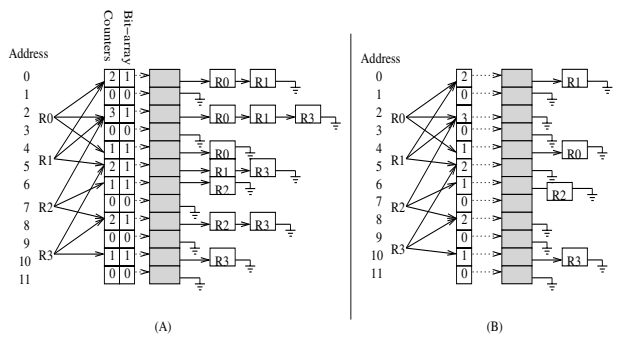

Fig. 1. (A) The hash table architecture using regular Bloom filter with bit-array (counting Bloom filters with counters) for four items. (B) The hash table architecture using pruned counting Bloom filter.

but this could inadvertently lead to removing a '1' that was the result of hashing another item that is still member of the set. To overcome this problem, the counting Bloom filter was introduced[9]. In the counting Bloom filter, each bit in the array is replaced by a small counter. When inserting an item, each counter indexed by the corresponding hash value is incremented, therefore, a counter in this filter essentially give us the number of items hashed to it. When an item is deleted, the corresponding counters are decremented. Based on a counting Bloom filter, we compute k hashing functions $h_1(), ..., h_k()$ over an input item and increment the related k counters indexed by these hash values. Subsequently, we store the item in the lists associated with each of the k buckets hence a single item is stored k times in memory. In the mentioned approach, we need to maintain up to k copies of each item requiring k times more memory compared to a standard hash table. However, in a Bloom filter only one copy is accessed while the other $(k-1)$ copies of item are never accessed, therefore, the memory requirement can be minimized in the mentioned architecture, resulting in the pruned counting Bloom filter. The pruned counting Bloom filter for Figure 1(A) is depicted in Figure 1(B). A method for pruning is to create normal counting Bloom filter and only keeping items with a minimum value in their counter, and for the items with same counter, the item with lower index is selected. In this method, insertion and deletion of redundant items are preformed simultaneously. It must be noted that during the pruned counting Bloom filter creation, the counter values are not changed thereby, after pruning, the counter do not express the number of items in the list and is greater than or equal to the number of items in each bucket. In the pruning procedure, all the other copies of an item except the one which is accessed during the search can be deleted. Therefore, after the pruning procedure we have one copy for each item and the result of this procedure is memory optimization. A limitation of pruning procedure occurs in the sequential insertion since the value of counters after each insertion does not show the number of items in the bucket, and also changes the counters of the other items that were formerly hashed in to the buckets. This limitation is overcome by the searching the buckets that pointed to by hashing functions and then recalculating addresses of items in these buckets. In other words, for inserting one item we

must reconsider all items in those buckets (A memory buffer that are pointed to by the counters storing the items of the set) [3][14].

IV. A Bloom Filter with an Additional Hashing Function (BFAH)

In this section, we present the concept of Bloom filter with an additional hashing function (BFAH).

A. The BFAH architecture and concept

For the counting Bloom filter, a pruning procedure was proposed to minimize the memory utilization. We utilize an additional hashing function in the regular and counting Bloom filters to minimize the memory. The concept of the BFAH is depicted in Figure 2.

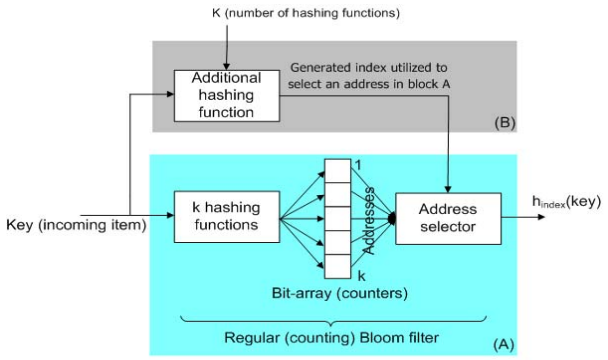

Fig. 2. A Bloom filter architecture with the additional hashing function.

In Figure 2, the incoming item is hashed by k hashing functions and the related bits in the bit-array are set (or counters are incremented in the counting Bloom filter) (see block A). We have to note that the bit-array (or counter array) points to addresses in the memory that physically stores the item. Subsequently, one of the generated addresses by k hashing functions is selected by another hashing function. This address is used to store the item in memory. The additional hashing function receives the incoming key and selects one of k received addresses to store in the memory. The additional hashing function (in block B depicted in Figure 2) outputs an index that is being used to select one of k addresses performed by the address selector in block A. Compared to the pruned counting Bloom filter, pruning criteria based on the counter values are used to determine the storage in the memory but in this approach the memory utilization is performed based on the additional hashing function. The proposed solution has the following features:

- Minimize the memory redundancy.
- The distribution of the incoming items are more randomize in comparison to Bloom filters since in this procedure after the generating of addresses, the selection of generated addresses by hashing function is performed by hashing technique that it assists to distribute incoming items uniformly.
- It can be utilized by regular Bloom filter, since it does not depend on the value of counter that is exploited by other pruning procedures in counting Bloom filter.

In the following, we discuss an example to highlight the BFAH concept. In this example, we assume that that there is a one-to-one relation between the array location index and the memory location address, i.e., array location 0 corresponds to address 0 in the memory. The BFAH for the Bloom filter in Figure 1 is depicted in Figure 3.

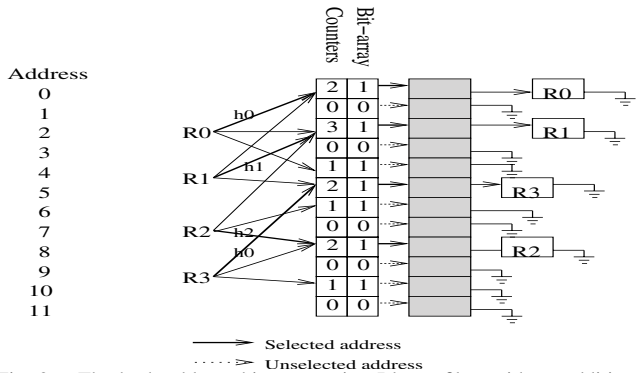

Fig. 3. The hash table architecture using Bloom filter with an additional hashing function.

Before items are inserted, the values of the counters and the bits in the bit-array are set to zero. In the first step, item $R0$ is hashed to array locations 0, 2, and 4 using hashing functions $h0$, $h1$, and $h2$. The values of the corresponding counters are incremented (or bits in the bit-array are set) and subsequently, the item/key must be stored in a memory location corresponding to one of the array location. Therefore, an additional hashing function utilized to select one of k addresses. In here, we utilize the hashing function "$key\ mod\ k$" as the additional hashing function where k represents the number of hashing functions. Based on this hashing function, address 0 that generated by hashing function $h0$ is selected and $R0$ is stored in the address 0. After that, $R1$ is hashed to addresses 0, 2, and 5. The value of counters is incremented (bit-array is set) to 2, 2, and 1, respectively. Using the hashing function "$key\ mod\ k$" to select the generated addresses by hashing functions, address 2 that generated by hash function $h1$ is sleeted to store $R1$. $R2$ is hashed to addresses 2, 6, and 8 that the value of their counters are 3, 1, and 1, therefore address 8 that generated by $h2$ is selected and item $R2$ is stored in address 8. For item $R3$ the procedure is similar to previous cases and it is stored in address 5.

V. Implementation and Results

In this section, we briefly present the implementation of a software packet classifier that utilizes of regular, pruned counting Bloom filters and a BFAH in tuple space packet classification and after that present the results.

A. Implementation

We implemented a software packet classifier based on the tuple space search that utilizes of regular, pruned

counting Bloom filters and BFAH. A high-level approach for multiple field search employs tuple spaces with a tuple representing information in each field specified by the rules. Srinivasan, et. al. introduced the tuple space approach and the collection of tuple search algorithms in [15][16]. In multidimensional packet classification, each rule defines prefix specifications on multiple packet header fields, and therefore each rule has more than one prefix length. The vector of prefix lengths of a rule is called a tuple, and the tuple space is the set of distinct tuples in a rule-set. The rules mapped to the same tuple can be searched using one hash operation. Since the number of tuples is generally much smaller than the number of rules. For each tuple, we utilize a Bloom filter (also: Bloom filter with an additional hashing function) with a set of universal hashing functions. The class of universal hashing functions is called H3 hashing functions that we utilize in the software packet classifier[6][12]. Based on tuple space representation for rule-set databases and IP packets, the size of input key is 88 bits (32 bit source IP address, 32 bit destination IP address, 8 bit Range-ID, 8 bit Nesting-Level and 8 bit protocol bit). The concept of Range-ID and Nesting-Level is defined to represent the tuple value of port ranges[15]. In this implementation, based on our observations of rule distribution in the tuples the maximum size of tuple or address space is assumed 2^{16} rules for 16 bit address. Therefore, $Q_{88\times16}$ denotes a set of matrices to define an $H3$ hashing function for tuple space packet classification algorithm.

B. Results

For testing the system, we use different rule-set databases and packet traces that have been utilized by the Applied Research Laboratory in Washington University in St. Louis [13]. The specification of the rule-set databases and packet traces is presented in Table I.

Table I includes seven rule-sets databases and packet traces. The rule-sets FW1, ACL1, IPC1 are extracted from real rule-sets and other generated by the Classbench benchmark[13]. In the following we present the related results. In these results, the graph with label "xx-R. B" shows the results for regular Bloom filter where "xx" shows different rule-set databases, "xx-P. C. B" shows the results for pruned counting Bloom filter and "xx-M. B" shows the results for BFAH. The graph "xx-R. B" is normalized to nk (n is number of items (rules in rule-set database) and k is the number of hashing functions), "xx-P. C. B" and "xx-M. B" are normalized to n. Since in the regular Bloom filter nk items and in the pruned counting Bloom filter and BFAH n items are stored. In the x-axis two sequences of numbers that are labeled by k and m/n, show the number of hashing functions and size of bit-array (number of counters) divide to number of items, respectively. The second sequence is evaluated based on the first sequence that shows the value of m/n and k to optimize the false positive probabilities. The number of collisions of BAHF for real rule-set databases is depicted in Figure 4.

From Figure 4, we can observe that the highest number of collisions occurs with the regular Bloom filter and lowest number of collisions occurs with the pruned counting Bloom filter. In Figure 4 (A), we observe some fluctuations that is due to the number of rules in rule-set database (Fw1 with 266 rules) and number of rules in each tuple. Based on Figure 4, we can observe that when the number of hashing functions and m/n (bit-array size/number of items) is increased the difference in the number of collisions in pruned counting Bloom filter and our solution will be small and our solution converges to pruned counting Bloom filter. The number of collisions for the synthetic rule-set databases is depicted in Figure 5.

Figure 5 depicts the number of collisions for rule-set database Fw1-100, Fw1-1k and Fw1-5k with small, mean and large number of rules, respectively. From the Figures 5(A), 5(B) and 5(C), we can observe that the behavior of different Bloom filters for synthetic rule-set databases is similar to real rule-sets. The average of all of rule-set databases in Table I is depicted in Figure6.

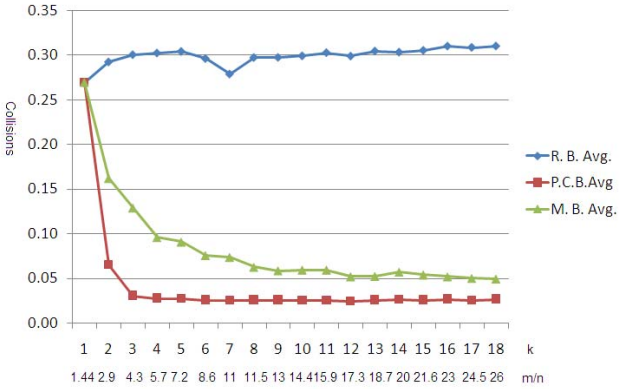

Fig. 6. Average number of collisions for all rule-set databases that normalized to n (number of rules in rule-set database) for pruned counting Bloom filter and BFAH and normalized to nk (number of rules multiply by number of hashing functions) for regular Bloom filter.

In Figure 6, "R. B. Avg.", "P. C. B. Avg." and "M. B. Avg." represent the average of all rule-set databases for regular, pruned counting and BFAH, respectively. Based on this figure, we can observe that the number of collisions for regular and pruned counting Bloom filters remain at constant level, and number of collisions for BFAH converges to pruned counting Bloom filter when the value of m/n is increased.

VI. Overall Conclusions

In this paper, we presented a new approach to decrease the memory utilization in the regular Bloom filter. We utilize an additional hashing function to select a generated address by hashing functions in the Bloom filter. Utilization of an additional hashing function increases the performance of the Bloom filter (in term of number of collisions). This architecture can easily be implement in hardware for counting and regular Bloom filters. We expect this approach to be useful in

Rule database	Fw1-100	Fw1-1k	Fw1-5k	Fw1-10k	Fw1	Acl1	Ipc1
Number of rules	92	971	4653	9311	266	752	1550
Number of tuples	26	42	52	57	36	44	179
Packet trace	**Fw1-100**	**Fw1-1k**	**Fw1-5k**	**Fw1-10k**	**Fw1**	**Acl1**	**Ipc1**
Number of packets	920	8050	46700	93250	2830	8140	17020

TABLE I

RULE-SET DATABASES AND PACKET TRACES SPECIFICATION.

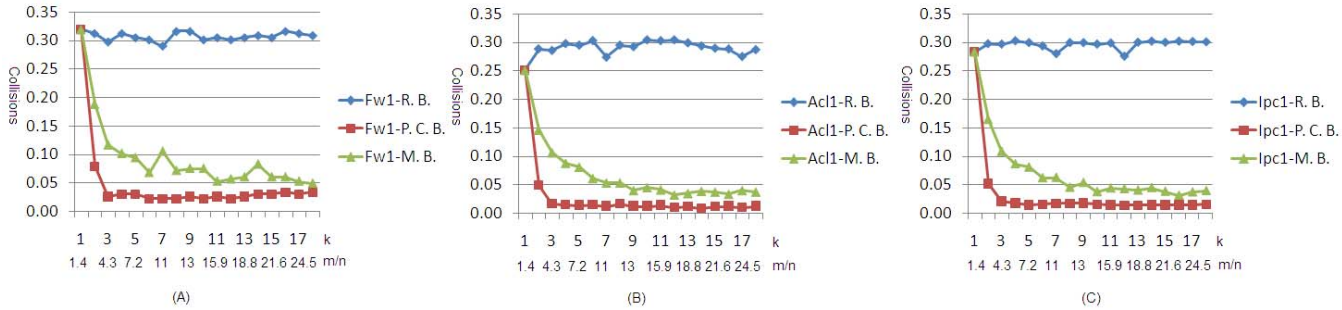

Fig. 4. Number of collisions for real rule-set databases that normalize to n (number of rules in rule-set database) for pruned counting Bloom filter and BFAH and normalize to nk (number of rules multiply by number of hashing functions) for regular Bloom filter. (A) Number of collisions for Fw1. (B) Number of collisions for Acl1. (C) Number of collisions for Ipc1.

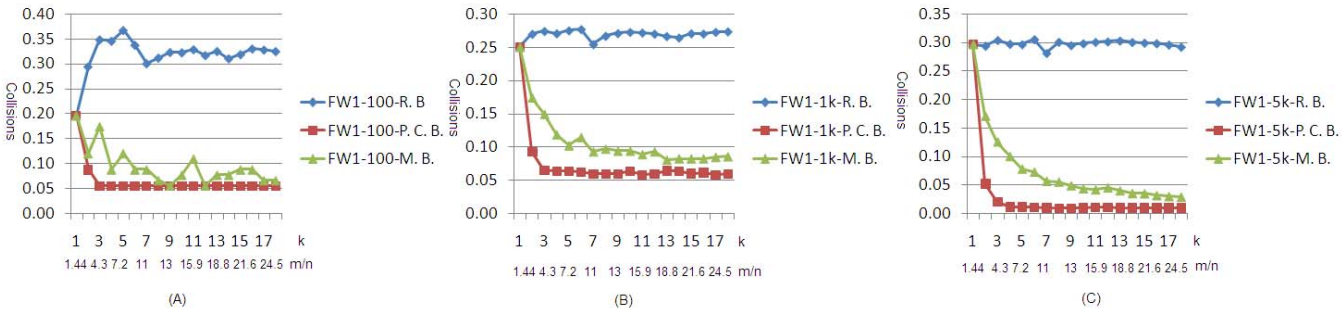

Fig. 5. Number of collisions for synthetic rule-set databases that normalize to n (number of rules in rule-set database) for pruned counting Bloom filter and BFAH and normalize to nk (number of rules multiply by number of hashing functions) for regular Bloom filter. (A) Number of collisions for Fw1-100. (B) Number of collisions for Fw1-1k. (C) Number of collisions for Fw1-5k.

the design of high performance memory architectures utilized in the network processors and related applications.

REFERENCES

[1] M. Ahmadi and S. Wong. "Network Processors: Challenges and Trends". In *Proc. of the 17th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2006*, pages 223–232, November 2006.

[2] M. Ahmadi and S. Wong. "A Cache Architecture for Counting Bloom Filters". In *Proceedings of 15th IEEE International Conference on Networks (ICON2007)*, pages 218–213, November 2007.

[3] M. Ahmadi and S. Wong. "Modified Collision Packet Classification Using Counting Bloom Filter in Tuple Space". In *Proc. of the 25'th IASTED Int. Conf. on Parallel and Distributed Computing and Networks (PDCN 2007)*, pages 70–76, February 2007.

[4] B. H. Bloom. "Space /Time Trade-offs in Hash Coding with Allowable Errors". *Communication of the ACM*, 13(7):422–426, July 1970.

[5] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In *Proc. 14'th Annual Allerton Conf. on Communication, Control, and Computing*, pages 636–646, October 2002.

[6] J. Lawrence Carter and Mark N. Wegman. "Universal Classes of Hash Functions". In *Proceedings of the 9th annual ACM symposium on Theory of computing*, pages 106–112. ACM Press, 1977.

[7] F. Chang, F. Wu-chang, and L. Kang. "Approximate Caches for Packet Classification". In *23'th Annual Conf. of the IEEE, INFOCOM*, pages 2196–2207, March 2004.

[8] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. "Fast Packet Classification Using Bloom Filters". Technical Report 27, Department of Computer Science And Engineering, Washington University in St. Louis, May 2006.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: A Scalable Wide-Area (WEB) Cache Sharing Protocol". *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[10] S. Kumar and P. Crowley. "Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems". In *Proc. Symp. on Architecture for Networking and Communications Systems (ANCS05)*, pages 91–103, October 2005.

[11] J. Mudigonda, H. M. Vin, and R. Yavatkar. "Overcoming the Memory Wall in Packet Processing: Hammers or Ladders?". In *Proc. of Symp. on Architecture for Networking and Communications systems (ANCS-05)*, pages 1–10. ACM Press, October 2005.

[12] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. "Efficient Hardware Hashing Functions for High Performance Computers". *IEEE Trans. Computer.*, 46(12):1378–1381, 1997.

[13] H. Song. "Evaluation of Packet Classification Algorithms". http:// www.arl.wustl.edu/ hs1/PClassEval.html, 2006.

[14] H. Song, J. Turner, S. Dharmapurikar, and J. Lockwood. "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing". In *Proc. of Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 181–192, August 2005.

[15] V. Srinivasan, S. Suri, and G. Varghese. "Packet Classification Using Tuple Space Search". In *Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 135–146, 1999.

[16] D. E. Taylor. *"Models, Algorithms, and Architectures for Scalable Packet Classification"*. PhD thesis, Department of Computer Science and Engineering Washington University, August 2004.