A New Design of Bloom Filter for Packet Inspection Speedup

Yang Chen, Abhishek Kumar, and Jun (Jim) Xu
College of Computing, Georgia Institute of Technology
{yangchen, akumar, jx}@cc.gatech.edu

***Abstract*— Bloom filter is a space-efficient randomized data structure for group membership query. It is widely used in networking applications which involve the packet header/content inspection. To provide fast membership query operation, this data structure resides in the main memory in most of its applications. Each membership query consists hashing for a set of memory addresses and memory accesses at these locations. In this paper, we propose a new design of Bloom filter in which every two memory addresses are squeezed into one I/O block of the main memory. With the burst-type data I/O capability in the contemporary DRAM design, the total number of memory I/O's involved in the membership query is reduced by half. Therefore, the average query delay can be reduced significantly. The cost of using this new design is a negligible increment of false positive rate as shown by both analysis and simulation.**

I. Introduction

Bloom filter has recently been adopted in many network applications from web caching to Peer-to-Peer networks and IP traceback [1], [2]. It is a space efficient randomized data structure for set membership query [3]. The space efficiency is achieved at the cost of a small probability of false positive, that is, an element may be announced as in the set while it is not. By exploiting the randomized nature of this data structure, the probability of false positive, i.e., false positive rate, is usually very small and outweighed by the space saving. However, as the transmission speed over the Internet backbone keeps increasing, how to speed up membership query in packet inspections at the link speed, e.g., per-flow traffic measurement at a Gigabit router [4], is an important yet challenging problem.

In most cases, a Bloom filter resides in the main memory to provide fast response to the query. Each membership query repeats the following procedure with a certain number of hash functions: hashing the query value for a memory address and one bit access. It is possible to speed up the membership query by reducing the number of hash functions. However, the false positive rate increases unless the space-efficiency is sacrificed as a tradeoff [5]. If the memory access is parallel, multiple memory accesses can be possibly finished within one I/O cycle [1]. But parallel memory access requires specific and expensive I/O design [6]. Processing multiple membership queries simultaneously using an array of Bloom filters is discussed in [7]. Note that this is only possible when these Bloom filters contain mutual exclusive subsets and each membership query is pre-determined for one of these subsets.

We propose a new Bloom filter design which introduces pair-wise correlation among those hashed memory addresses by squeezing every two addresses into one memory I/O block. To be specific, once an address is computed, the next address computed must be within a certain offset away from the previous one. Therefore, each pair of hashed bits are accessed in one memory I/O cycle with the burst-type I/O mechanism provided in some contemporary memory designs such as Synchronous DRAM (SDRAM). Compared with one memory access for one hash result in the standard design, the component of average membership query delay due to memory I/O latency is reduced by up to 50% in our new design.

Note that a false positive occurs when each hashed result from a non-member is equal to at least one hashed value from any of the members. Intuitively, the larger the output range of the hash function, the less chance that two hashed results are identical, and correspondingly the smaller false positive rate given a fixed number of hash functions. In the standard Bloom filter design, the output range of those hash functions is the entire memory allocation. However, in our design, half of the hash functions' output range is limited within one memory block size around the outputs of their counterparts. Since the *randomness* level of the hash functions' outputs is reduced, the false positive rate in the new design is understandably larger than the optimal value. We provide an analytical model to quantify the impact of this correlation existing between memory address pair on the false positive rate. Interestingly, both the analytical and simulation results show that if the memory block size is reasonably large, e.g., 32 bits, the difference of false positive rate is negligible.

The rest of this paper is organized as follows. In Section II, background knowledge of Bloom filter is provided with a short review of previous approaches for query speedup. Section III describes the details of the new Bloom filter design. Section IV provides an analytical model for the false positive rate of the new design and derives the average query delay. The performance evaluation of this new design as well as the analytical model's validation are given in Section V. Section VI concludes this paper.

II. Background and Related Work

A. Bloom Filter Theory

A Bloom filter representing a set $S = \{x_1, x_2, \ldots, x_n\}$ usually consists of an array of m bits (initially all set to 0) and k independent hash functions $h_1, \ldots, h_k$ with output range $[1, \ldots, m]$. For each and every item x in the set S,

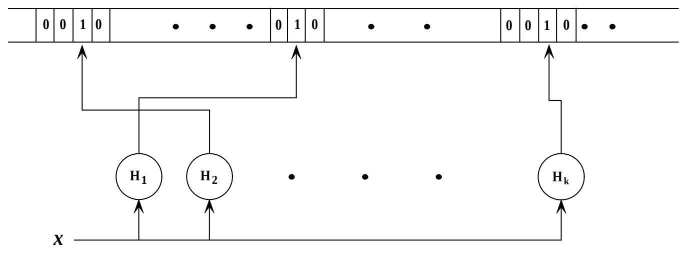

Fig. 1. Add one member x

the $\{h_i(x)|i = 1 \ldots k\}$ bits in the array are set to 1. Figure 1 shows the process of inserting member x's hashing results into the array. To determine whether an element y is a member of set S or not, we hash it k times with the same hash function set and check the corresponding bits in the array. If at least one of those bits is 0, this element cannot be in the set; otherwise, the element is considered to be a set member.

The advantage of using Bloom filter is its storage space efficiency. Only m/n bits is assigned for each element in the set instead of storing the element, which may take up to several hundred bytes in applications such as web caching [8]. But the tradeoff of using Bloom filter is the occurrence of false positive, i.e., k hash functions all return locations with the bit's value as 1 for an item which is actually not in the set. For many applications, false positive is acceptable as long as the false positive rate is sufficiently small and outweighed by the storage space saving.

False positive rate can be calculated in a straight forward approach assuming the hash functions' outputs are perfectly random. After all the elements of S are inserted into the bit array, the probability that a specific bit is still 0 is

$$(1 - \frac{1}{m})^{kn} \approx e^{-kn/m} \tag{1}$$

Let $p = e^{-kn/m}$, the probability of a false positive is

$$(1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - p)^k \tag{2}$$

Note that when $k = (m/n) \cdot \ln 2$, p equals $1/2$, we have a minimum false positive rate for the given m and n values according to the analysis in [1].

B. Speedup Variants

Since one time membership query in Bloom filter repeats address computation and memory access for each of the k hash functions, if k decreases, the time involved in those address computations and memory I/O is reduced correspondingly. However, the false positive rate increases according to equation 2. Therefore, the value p in equation 2 must increase to keep the optimal false positive rate. For an example, given a Bloom filter with 16 hash functions, the optimal false positive rate is $(0.5)^{16}$. It can be shown by simple calculation that for a same size (n) set, more than 20 times larger memory size (m) is required if k is reduced to 2. This tradeoff between number of hash function and space-efficiency is also discussed in [5].

Another approach for query speedup is using a Bloom filter array to process multiple queries simultaneously [7]. Without loss of generality, suppose the original set S can be divided into g equal size and mutual exclusive subset $s_1, s_2, \ldots, s_g$. Hence, we can build g Bloom filters each of which has k hash functions and memory size of m/g. Each Bloom filter contains n/g elements from one subset. This Bloom filter array can process g queries simultaneously given each query is predetermined for one of these subsets. The total memory size does not change while the average query delay can be reduced to $1/g$ of that in a standard Bloom filter [1]. Nevertheless, $(g - 1) * k$ extra hash functions and corresponding control units are required. Furthermore, unlike the application in [7], normally there is no explicit mapping between queries and subsets, i.e., different Bloom filters in the array. Thus, a query must go through all the Bloom filters and there is no saving on the query delay.

With parallel memory I/O support, a straightforward variant of Bloom filter is proposed in [1]. Instead of using an m bits array for all the k hash functions, each hashing function has an output ranging of m/k consecutive bits and disjoint with the outputs of the others. Therefore, the k times one bit memory I/O can be done in parallel within one time memory I/O cycle. But this variant requires a memory type having parallel input/output ports and supporting concurrent memory accesses. Since the manufacturing cost is dominated by the number of I/O ports in the package [6], the current memory products in the market usually do not support a large number of input/output ports which makes the parallel approach not practically feasible in a near future.

In a parallel Bloom filter, the probability that a specific bit is 0 is $(1-k/m)^n \approx e^{-kn/m}$. Since $(1-k/m)^n \leq (1-1/m)^{kn}$, the false positive rate of this parallel variant is slightly higher than that of a standard Bloom filter [1]. This shows how the level of *randomness* of those hashed addresses affect the false positive rate. To be specific, since each hash function's output is limited to a block of size m/k instead of m in a standard Bloom filter, the false positive rate increases. However, as long as m/k is reasonably large, the effect of randomness reduction on false positive rate is not significant. In the next section, we provide a new design of Bloom filter which also has reduced *randomness* in hashed addresses and a negligible false positive rate increment.

III. Our Design

Membership query in Bloom filter contains a series of hashing and memory I/O. Note that hash functions can be implemented efficiently in hardware [9]. The memory I/O latency can be the major component of membership query delay.

One time memory I/O latency consists of two components: address initiation delay and data I/O delay [10]. The former is the dominant factor and cannot be reduced without some hardware technology advances [11]. In today's memory design, the word size is usually 16 bits (or larger), which means 16

[1]Throughout this paper, we call the Bloom filter design discussed in Section II-A the standard Bloom filter

consecutive bits are read or written in one memory access after one address initiation. In addition, burst mode data I/O, i.e., multiple consecutive words access after one address initiation, is available in the new designs of DRAM such as Synchronized DRAM (SDRAM) [6].

Although we only need to access one bit for each hashed address during a membership query, one or multiple words are accessed in the memory after one address initiation. In the standard Bloom filter, the addresses of all the one bit memory access are uniformly and independently distributed throughout the memory array, which makes the address initiation sharing impossible. But if we introduce correlations between the addresses of every two one-bit I/O's, they can share one address initiation. This is an equivalent to two bits access within one memory I/O. Thus, the total number of memory I/O is now $k/2$ instead of k. The average membership query time can be reduced accordingly. Based on this observation, we introduce a new Bloom filter design which can speed up the membership query over the practically available non-parallel memory types.

The new Bloom filter design has an m bits memory array and k hash functions as in a standard Bloom filter. Therefore, the storage efficiency as well as the hardware cost do not change. The m bit array is divided into blocks of size w. And the k hash functions are evenly divided into two categories: primary and secondary. The output range for primary hash function is from 1 to m while the secondary's output range is 1 to w. In other words, the primary hash function operates the same way as a hash function does in a standard Bloom filter while the secondary hash function only provides a pseudo random offset within a memory block.

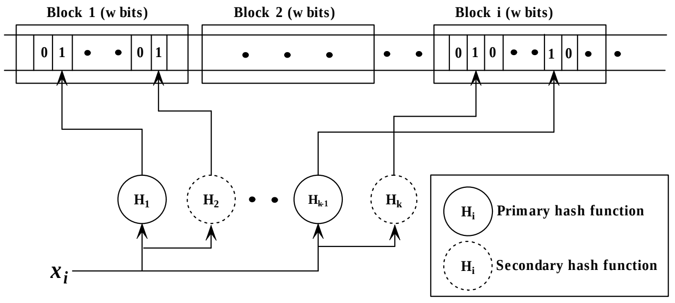

Fig. 2. Construct a Bloom filter of our design

Suppose in one burst mode memory I/O, a total size of w consecutive bits are accessed after one address initiation. Algorithm 1 and Figure 2 show how to construct a Bloom filter of our new design. One member in the set S is hashed by these hash function pairs. Two bits are visited and set after one address initiation. This procedure continues until all the members in S are hashed and stored in the memory.

To query about element y's membership, it is sent to those primary-secondary hash function pairs as well. For each address initiation, two bits within one memory block of size w can be checked. If the k bits are all 1, y is considered as a member of S. Details of this procedure are shown in Algorithm 2.

Algorithm 1 Construct the New Bloom filter

```
$x_i$: value of the $i^{th}$ element in the set
$h_{2j-1}$: $(2j-1)^{th}$ hash function with output range $[1 \dots m]$
$h_{2j}$: $(2j)^{th}$ hash function with output range $[1 \dots w]$
$v_i$: $i^{th}$ bit's value in the bit array
for $i = 1$ to $n$ do
  for $j = 1$ to $k/2$ do
    $addr_1 \Leftarrow h_{2j-1}(x_i)$
    /* $addr_1$ is in the block defined by bits $[L \dots U]$ */
    $addr_2 \Leftarrow h_{2j}(x_i)$
    $v_{addr_1} \Leftarrow 1$
    if $((addr_1 + addr_2) > U)$ then
      $v_{(addr_1+addr_2-w)} \Leftarrow 1$
    else
      $v_{(addr_1+addr_2)} \Leftarrow 1$
    end if
  end for
end for
```

Algorithm 2 Query with the New Bloom filter

```
$y$: value of the element to be tested
$temp = 0$
for $j = 1$ to $k/2$ do
  $addr_1 \Leftarrow h_{2j-1}(y)$
  /* $addr_1$ is in the block defined by bits $[L \dots U]$ */
  $addr_2 \Leftarrow h_{2j}(y)$
  $temp \Leftarrow temp + v_{addr_1}$
  if $((addr_1 + addr_2) > U)$ then
    $temp \Leftarrow temp + v_{(addr_1+addr_2-w)}$
  else
    $temp \Leftarrow temp + v_{(addr_1+addr_2)}$
  end if
end for
if $(temp == k)$ then
  $y$ is in the set
else
  $y$ is not in the set
end if
```

The level of *randomness* of the memory addresses is reduced because the secondary hash function's output range is limited within a block of size w instead of the entire array m. In the parallel Bloom filter, the false positive rate increases due to the reduced output range of hash functions. A similar increment on the false positive rate is expected in this new design compared with standard Bloom filter. But with a reasonably large memory block size, e.g., 32, this increment is not significant. To quantitatively study the relation between block size and false positive rate, we provide an analytical model in the next section.

Another interesting observation is that our design is *orthogonal* to other Bloom filter variants for membership query speedup. In other words, it can be used together with those approaches such as Bloom filter array and take the advantage of any improvement from them for additional membership query speedup. Even for the parallel Bloom filter which does not allow further I/O delay reduction, our design can be used to reduce the number of I/O ports and associated packaging cost by half because one output contains bits from two hashed addresses instead of one as in standard design.

IV. Performance Analysis

A. False Positive Rate

In the query of an element y's membership, denote e_i as the event that the bit is 1 at the address generated by the i^{th} hash function. Then the false positive rate can be written as the following equation similarly to the analysis in [1]:

$$P(e_1 \wedge e_2 \dots e_{k-1} \wedge e_k) = \prod_{i=1}^{k/2} P(e_{2i-1} \wedge e_{2i})$$

where no correlation exists among results from different primary/secondary hash function pairs.

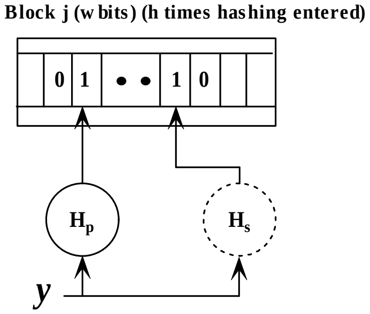

Fig. 3. False positive rate model for $P(e_p \wedge e_s)$

Assume odd order of hash functions are primary ones and a uniform randomness is achieved through good hashing implementation, the above equation equals $P(e_p \wedge e_s)^{k/2}$ where e_p and e_s means that the bit is set to be 1 at the address from the primary and secondary hash function respectively. For each $P(e_p \wedge e_s)$, the problem is simplified as shown in Figure 3 and this probability is rewritten using total probability theorem:

$$P(e_p \wedge e_s) = \sum_{i=1}^{nk/2} P(e_p \wedge e_s | h = 2i) P(h = 2i) \qquad (3)$$

$P(h = 2i)$ is the probability that in constructing the Bloom filter, $2i$ hashing results are the addresses within this particular block. Since a primary/secondary hash function pair sets two bits in the same block, h is always an even number. Also note that to insert n members, there are total nk times hashing. Thus, the i has an upper bound as $nk/2$. On the other hand, $P(e_p \wedge e_s | h = 2i)$ is the probability that in a w bit block, a pair of bits located by the primary/secondary hash function pair are all 1 given $h = 2i$ in the same block. The computation of equation 3 can be terminated once $P(h = 2i)$ is less than a certain threshold, e.g., $1.0e^{-10}$.

$P(h = 2i)$ is calculated using basic urn model given in [12]: the probability that there are precise i balls (hashing pairs) in the urn (block) is:

$$P(h = 2i) = \left(\begin{array}{c} \frac{nk}{2} \\ i \end{array} \right) (\frac{w}{m})^i (1 - \frac{w}{m})^{\frac{m}{w} - i}$$

when $wnk/2m$ is large, which is always the case if k is chosen to be around $(m/n) \cdot \ln 2$ and the block size is reasonably large, e.g., 32 bits, Poisson approximation can be applied:

$$P(h = 2i) = \frac{e^{-\frac{w \ln 2}{2}} (-\frac{w \ln 2}{2})^i}{i!} \qquad (4)$$

Since the order of insertions does not matter, $P(e_p \wedge e_s | h = 2i)$ is equal to the solution of the following problem: given two random and different locations, a^{th} bit and b^{th} bit in the w bit memory block, what is the probability that both bits are set to 1 after setting the bits designated by the $h = 2i$ hashing results. Denote v_a and v_b as the value for the a^{th} bit and b^{th} bit respectively, we have:

$$\begin{aligned} P(e_p \wedge e_s | h = 2i) &= P(v_a = 1 \wedge v_b = 1 | h = 2i) \\ &= 1 - P(v_a = 0 | h = 2i) - P(v_b = 0 | h = 2i) \\ &+ P(v_a = 0 \wedge v_b = 0 | h = 2i) \end{aligned} \qquad (5)$$

$$P(v_a = 0 | h = 2i) = [(\frac{w-1}{w})(\frac{w-2}{w-1})]^i \qquad (6)$$

In equation 6, $(w-1)/w$ means the probability that one of the primary hash result is an address other than a^{th} bit. Suppose it is c^{th} bit. Then the associated secondary hash result must be in an address other than the a^{th} and c^{th} bit. Therefore, the probability is $(w-2)/(w-1)$. $P(v_b = 0 | h = 2i)$ has the same form and similarly:

$$P(v_a = 0 \wedge v_b = 0 | h = 2i) = [(\frac{w-2}{w})(\frac{w-3}{w-1})]^i \qquad (7)$$

B. Average Query Delay

Without loss of generality, we assume the total number of hash functions k is even, the memory access time is normalized as 1 unit and the dominant factor of query time is memory access delay.

To declare a value is a set member, all the hash functions are involved in the query and the total number of memory access is k in a standard Bloom filter and $k/2$ in our design. Therefore, the saving on query time is 50%.

If any of the k hash functions returns a "0" bit, the query is terminated and the value is claimed to be a non-member. In this case, the percentage of time saving is not the same as in the previous case. For an example, suppose a Bloom filter has two hash functions. The probability of any bit in the array to be 0 is 0.5 at the optimal parameter setting as shown in previous section. For the standard design, a non-member is detected after the first hashing and the second hashing with the probability of 0.5 and $(0.5)^2$. The average query time for a non-member is $1 \times 0.5 + 2 \times (0.5)^2 = 1$. On the other hand, all the decisions are made after only one time memory access in our design. So the average query time for a non-members is $1 \times 0.5 + 1 \times (0.5)^2 = 0.75$.

For k hash functions, the average query time of a non-member for a standard Bloom filter is:

$$T_s = \sum_{i=1}^{k} i * (\frac{1}{2})^i = 2[1 - (\frac{k}{2} + 1)(\frac{1}{2})^k] \qquad (8)$$

while the average query time for our design is:

$$T_n = 2\sum_{i=1}^{k/2} i*(\frac{1}{2})^{2i} + \sum_{i=1}^{k/2} i*(\frac{1}{2})^{2i} = \frac{4}{3}[1-(\frac{3k}{8}+1)(\frac{1}{4})^{k/2}] \quad (9)$$

The time saving percentage, i.e., $(1 - T_n/T_s)$ converges to the 33% with the increment of the number of hash functions as shown in Figure 4.

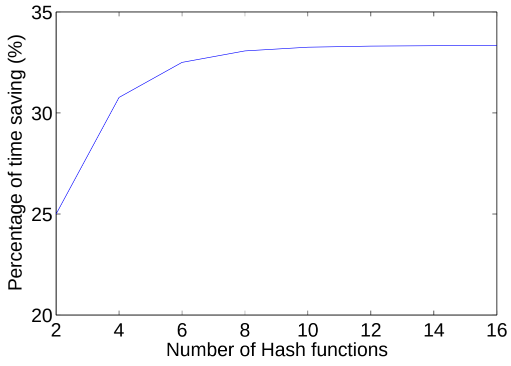

Fig. 4. Average query time saving percentage

In summary, depending on the distribution of the query results, i.e., the percentage of the query with a "hit" or not, the average query time decreases by 30% to 50% in our new design with an adequate number of hash functions.

V. Numerical Results

In this section, we first use simulation to compare the false positive rate of this new design with that of a standard Bloom filter. Then we validate our analytical model. Throughout the simulation, m is 2^{16}. k and n follow $k = (m/n) \cdot \ln 2$. For each scenario (represented by the unique k and w values), we run 1,000,000 membership queries.

	Ideal*	w=32	w=64	w=128	w=256
k=2	0.25	0.255597	0.252681	0.251539	0.250745
k=4	0.0625	0.065374	0.063910	0.063172	0.062816
k=6	0.015625	0.016720	0.016137	0.015899	0.015790
k=8	0.003906	0.004270	0.004083	0.003988	0.003939
k=10	0.000977	0.001091	0.001033	0.001004	0.000994
k=12	0.000244	0.000277	0.000263	0.000253	0.000248
k=14	0.000061	0.000072	0.000066	0.000063	0.000062
k=16	0.000015	0.000019	0.000016	0.000017	0.000015
*Theoretically optimal performance of the standard Bloom filter design					

TABLE I
False positive rate comparison

Comparison of the false positive rate is given in Table I with k varying from 2 to 16. We test the new design with different block size from 16 to 256. Although some slight increment on false positive rate can be observed when the block size is small, e.g., w is 32, the difference keeps decreasing as we enlarge the block's size. With w equals to 128, i.e., 4 to 8 consecutive words access within one burst memory I/O (common in SDRAM operation), the difference is almost negligible.

Table II shows that except when w is relatively small (Poisson approximation has large deviation), our analytical model is fairly accurate when block size is moderately large.

	k=4		k=8		k=16	
w	Analysis	Simulation	Analysis	Simulation	Analysis	Simulation
8	7.84e-02	7.91e-02	6.15e-03	6.26e-03	3.78e-05	4.12e-05
16	6.93e-02	7.05e-02	4.81e-03	4.98e-03	2.31e-05	2.58e-05
32	6.56e-02	6.64e-02	4.30e-03	4.42e-03	1.85e-05	1.84e-05
64	6.39e-02	6.45e-02	4.09e-03	4.16e-03	1.67e-05	1.82e-05
128	6.32e-02	6.36e-02	3.99e-03	3.99e-03	1.59e-05	1.64e-05
256	6.28e-02	6.36e-02	3.95e-03	3.99e-03	1.56e-05	1.64e-05

TABLE II
Validation of the analytical model

VI. Conclusion

In this paper, we propose a new design of the Bloom filter in which every two bits from hashing must reside close enough to each other in the memory. The burst type I/O in contemporary DRAM design is exploited to reduce the number of memory I/O's in membership query by half and expedite the whole procedure. In this design, the memory access addresses are pairwisely correlated. To quantify its effect on the false positive rate, we provide an analytical model. Both the simulation and analysis show that there is only a negligible increment on the false positive rate in the new design. Furthermore, our design is *orthogonal* to other speed up variants so that it can takes the advantage of any improvement from those approaches for additional membership query speedup.

References

[1] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.
[2] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, 2006.
[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.
[4] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-code bloom filter for efficient per-flow traffic measurement," in *Proc. of IEEE INFOCOM*, vol. 3, 2004, pp. 1762–1773.
[5] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604–612, October 2002.
[6] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary dram architectures," in *Proceedings of the 26th International Symposium on Computer Architecture*, 1999, pp. 222–233.
[7] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *Proceedings of High Performance Interconnects*, 2003, pp. 44–51.
[8] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *Networking, IEEE/ACM Transactions on*, vol. 8, no. 3, pp. 281–293, 2000.
[9] K. Jarvinen, M. Tommiska, and J. Skytta, "Hardware implementation analysis of the md5 hash algorithm," in *Proceedings HICSS'05 - Track 9*. IEEE Computer Society, 2005, p. 298.1.
[10] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design*, 2nd ed. Morgan Kaufmann, 1997.
[11] *MT48LCxM4A2 serie SDRAM data sheet*, Micron INC.
[12] M. V. Ramakrishna, "Hashing practice: analysis of hashing and universal hashing," in *Proceedings of the ACM SIGMOD*, 1988, pp. 191–199.