SCIENCE CHINA
Information Sciences

• **RESEARCH PAPERS** •

January 2011 Vol. 54 No. 1: 23–37
doi: 10.1007/s11432-010-4132-4

An index-split Bloom filter for deep packet inspection

HUANG Kun[1] & ZHANG DaFang[1,2*]

[1]*School of Computer and Communication, Hunan University, Changsha* 410082*, China;*
[2]*School of Software, Hunan University, Changsha* 410082*, China*

Received February 18, 2009; accepted November 2, 2009; published online November 24, 2010

Abstract Deep packet inspection (DPI) scans both packet headers and payloads to search for predefined signatures. As link rates and traffic volumes of Internet are constantly growing, DPI is facing the high performance challenge of how to achieve line-speed packet processing with limited embedded memory. The recent trie bitmap content analyzer (TriBiCa) suffers from high update overhead and many false positive memory accesses, while the shared-node fast hash table (SFHT) suffers from high update overhead and large memory requirements. This paper presents an index-split Bloom filter (ISBF) to overcome these issues. Given a set of off-chip items, an index of each item is split apart into several groups of constant bits, and each group of bits uses an array of on-chip parallel counting Bloom filters (CBFs) to represent the overall off-chip items. When an item is queried, several groups of on-chip parallel CBFs constitute an index of an off-chip item candidate for a match. Furthermore, we propose a lazy deletion algorithm and vacant insertion algorithm to reduce the update overhead of ISBF, where an on-chip deletion bitmap is used to update on-chip parallel CBFs, not adjusting other related off-chip items. The ISBF is a time/space-efficient data structure, which not only achieves $O(1)$ average memory accesses of insertion, deletion, and query, but also reduces the memory requirements. Experimental results demonstrate that compared with the TriBiCa and SFHT, the ISBF significantly reduces the off-chip memory accesses and processing time of primitive operations, as well as both the on-chip and off-chip memory sizes.

Keywords network security, packet processing, deep packet inspection, hash table, Bloom filter

Citation Huang K, Zhang D F. An index-split Bloom filter for deep packet inspection. Sci China Inf Sci, 2011, 54: 23–37, doi: 10.1007/s11432-010-4132-4

1 Introduction

In recent years, the Internet has been threatened and assaulted by a variety of emerging break-in attacks, such as worms, botnets, and viruses. Network intrusion detection and prevention systems (NIDS/NIPS) [1] are recognized as one of the most promising components to provide protection on the network. Deep packet inspection (DPI) is the core of NIDS/NIPS, which inspects both packet headers and payloads to identify and prevent suspicious attacks. DPI usually performs packet preprocessing on packet headers to classify and search each incoming packet, such as TCP connection and session records [2, 3], and per-flow state lookups [4]. Afterwards, signature matching algorithms [5, 6] are used to perform a pattern matching on packet contents for predefined signatures of an attack. In essence, DPI is one of the dominant content filtering techniques, which has found many applications in network besides NIDS/NIPS, such as Linux layer-7 filter [7], P2P traffic identification [8, 9], and context-based routing and accounting.

*Corresponding author (email: dfzhang@hunu.edu.cn)

As link rates and traffic volumes of Internet are constantly growing, DPI is facing high performance challenges such as how to satisfy the time/space requirements of packet processing. In high-speed routers, DPI is typically deployed in the critical data path, where massive high-speed packets are processed against hundreds of thousands of predefined rules. Since software-based DPI solutions cannot keep up with high-speed packet processing, many hardware-based DPI solutions [10–14] have been recently proposed to achieve 10–40 Gbps packet processing. Modern embedded devices, such as application specific integrated circuit (ASIC), field programmable gate array (FPGA), network processor (NP), and ternary content addressable memory (TCAM), are exploited by these hardware-based solutions to improve the DPI throughput.

Unfortunately, hardware-based DPI solutions suffer from large memory requirements which cannot fit in small embedded memory. Modern embedded devices usually adopt the hierarchical memory architecture, integrating high-rate on-chip memory and high-capacity off-chip memory. The on-chip memory offers fast lookups, e.g. SRAM access time at 1–2 ns, but has a small memory space. In contrast, the off-chip memory has a large memory space but performs slow lookups, e.g. DRAM access time at 60 ns. For instance, the state-of-art Xilinx Vertex-5 FPGA provides an on-chip SRAM of total 10 Mb, which is not sufficient to satisfy ever-increasing memory requirements of DPI. Hence, it is critical and crucial to implement a time/space-efficient packet preprocessing scheme in hardware, which reduces both the off-chip memory accesses and memory requirements, thus accelerating the performance of DPI.

Artan et al. [15] have proposed a trie bitmap content analyzer (TriBiCa) to achieve high throughput and scalability of DPI. The aim of the TriBiCa is to provide minimal perfect hashing functionality and support fast set-membership lookups. The TriBiCa consists of an on-chip bitmap trie and a list of off-chip items. When an item x is stored, x is hashed to a node at each layer in the on-chip bitmap trie, and its unique index in off-chip memory is yielded. When an item y is queried, the on-chip bitmap trie is traversed by hashing y to return an index, and the corresponding item x is accessed for an exact match on y. In essence, the TriBiCa uses the on-chip bitmap trie to filter out most of irrelevant items and yield an index pointing to an off-chip item before signature matching. Hence, the TriBiCa reduces numbers of both on-chip memory accesses and signature matching operations, thus improving the DPI throughput.

However, the TriBiCa suffers from high update overhead and many false positive memory accesses. First, the TriBiCa uses heuristic equal-partitioning algorithms to construct the on-chip bitmap trie, which leads to no support for dynamically changed items. When an item is inserted or deleted, the TriBiCa needs to reconstruct the on-chip bitmap trie, which results in high update overhead. Second, the TriBiCa uses only one hash function at each layer to construct each node in the on-chip bitmap trie, which incurs too many false positive off-chip memory accesses, thus limiting the worst-case lookup performance.

To improve the worst-case performance of DPI, Song et al. [16] have proposed a shared-node fast hash table (SFHT), supporting fast hashing lookups. The SFHT consists of m buckets, each containing an on-chip counter and a linked list of off-chip items, and m counters compose an on-chip counting Bloom filter (CBF) [17]. The SFHT uses hash functions $h_1(), h_2(), \ldots, h_k()$ to map each item to k storage locations, among which one location is selected to store and search the item. When an item x is stored, k buckets indexed by hashing x are made point to a shared linked list. When an item y is queried, one of k buckets with the minimal counter value and the smallest index is selected to search y. In essence, the SFHT uses the on-chip CBF to filter out most of irrelevant items and yield an index pointing to an off-chip item before signature matching. Hence, the SFHT reduces numbers of both on-chip memory accesses and signature matching operations, thus improving the lookup performance.

However, the SFHT suffers from high update overhead and large memory requirements. First, the SFHT has $O(k+nk^2/m)$ and $O(k)$ average off-chip memory accesses of insertion and deletion respectively, where there are n items, m buckets, and k hash functions. Especially, when an item is inserted, the SFHT traverses k linked lists indexed by hashing the item. This traversal induces frequent accesses of off-chip memory, which in turn increases the update overhead. Second, the SFHT has large on-chip/off-chip memory requirements. Each bucket contains one 4-bit counter and one $\lceil \log_2 n \rceil$-bit header pointer pointing to an off-chip linked list, which consumes more scarce and expensive on-chip memory. In addition, to guarantee that the counter value per bucket reflects the length of an associated linked list, the SFHT needs to duplicate 1–3 times shared items, which leads to large off-chip memory requirements.

In this paper, we propose a novel index-split Bloom filter (ISBF) to significantly reduce both the off-chip memory accesses and memory requirements. The ISBF consists of several groups of on-chip parallel CBFs and a list of off-chip items. When a set of items are stored, an index of each item is split apart into B groups, each containing b bits; each group of bits splits the overall items into 2^b subsets, each represented by a CBF, and thus a total of 2^b parallel CBFs per group are constructed in on-chip memory. When an item y is queried, B groups of on-chip parallel CBFs constitute an index of an item candidate x, and a final decision is made by checking whether x matches y. Furthermore, both a lazy deletion algorithm and vacant insertion algorithm are proposed to reduce the update overhead of ISBF. These algorithms use an on-chip deletion bitmap to record states of all off-chip items. When an item is inserted or deleted, these algorithms only update on-chip parallel CBFs, but do not adjust indexes of other off-chip items.

We theoretically show that the ISBF is a time/space-efficient data structure, achieving $O(1)$ average off-chip memory accesses of insertion, deletion, and query; compared with the TriBiCa and SFHT, the ISBF reduces the on-chip memory requirements by $2b$ times and b times respectively, where b is the number of split bits. Experimental results show that compared with the TriBiCa and SFHT, the ISBF reduces the off-chip memory accesses of insertion by 12.5 times and 4.6 times respectively as well as the off-chip memory accesses of deletion by 6 times and 5.8 times respectively, and has almost the same off-chip memory accesses of query as the TriBiCa and SFHT; the ISBF reduces the processing time of insertion by 45.9 times and 89 times respectively as well as the processing times of deletion by 10.4 times and 76.2 times respectively; in addition, the ISBF reduces 41.9% and 54.6% on-chip memory respectively, and has the same off-chip memory as the TriBiCa, while reducing 2.9 times off-chip memory compared with the SFHT.

2 Related work

As proliferation of high-speed network applications increases, software-based DPI solutions cannot keep up with packet processing at line speed. For example, both Snort [5] and Bro [6] only achieve the throughput of about 100 Mbps. In the past few years, hardware-based DPI solutions have been proposed to deal with high-speed large-volume streaming packets. Recent effort to increase the DPI speed focuses on two aspects as follows: 1) Efficient signature matching algorithms are implemented in hardware to increase the speed of unit inspection operation. 2) Efficient packet preprocessing schemes are proposed for hardware implementation to decrease the number of inspection operations.

Multi-pattern signature matching algorithms typically use a deterministic finite automaton (DFA) to represent a set of predefined signatures. Signature matching algorithms are classified into string matching and regular expression matching according to signature representations. To increase the throughput, several hardware-based signature matching algorithms have been proposed by using specific-purpose embedded devices. For instance, FPGA-based solutions [10, 11] can achieve 10 Gbps throughput, and ASIC-based solutions [18–20] can achieve 20 Gbps throughput, while TCAM-based solutions [12, 14, 21] can achieve more than 20 Gbps throughput. However, these algorithms suffer from limited embedded memory, which is not sufficient to satisfy the memory requirements of DFA.

Several interesting string matching algorithms have been proposed to speed up the throughput and reduce the memory requirements. Tuck et al. [22] proposed bitmap compression and path compression methods to reduce the memory requirements of Aho-Corasick algorithm [23] and enhance the worst-case performance. Dharmapurikar et al. [24] and Hua et al. [25] independently proposed fixed-stride and variable-stride string matching algorithms to improve the throughput. Tan et al. [18] proposed a bit-split Aho-Corasick algorithm, Piyachon et al. [13] proposed a bit-byte-level Aho-Corasick algorithm, and Lu et al. [14] proposed a multiple-character parallel Aho-Corasick algorithm, aiming at parallelizing string matching and minimizing the memory requirements. Lunteren et al. [20] and Song et al. [26] proposed the B-FSM and CDFA to reduce the memory requirements of DFA by merging the state transitions respectively.

As regular expressions are more expressive and flexible, several memory-efficient regular expression matching algorithms have been proposed for hardware implementation. Kumar et al. [27, 28] proposed a D^2FA and its improved CD^2FA, where several identical transitions between states were replaced by a default transition to reduce the memory requirements of DFA. Becchi et al. [29] proposed a state-merged DFA, where several non-equivalent states were merged by using transition labeling to compress the memory requirements. Yu et al. [21] proposed an mDFA, where a large set of signatures were partitioned into multiple groups to reduce the overall memory requirements of DFA. Smith et al. [30, 31] proposed an XFA, where the DFA was augmented with auxiliary variables and simple instructions to eliminate the state-space explosion problem. Kumar et al. [32] proposed several heuristic mechanisms to tackle three drawbacks of DFA, i.e. insomnia, amnesia and acalculia, in order to reduce the memory requirements and at the same time speed up the throughput.

In recent years, some novel packet preprocessing schemes such as hardware hash tables and Bloom filters have been proposed to further improve the DPI throughput. Hash tables support fast associate lookups, while the Bloom filter [33] is a space-efficient randomized data structure for fast set-membership queries. The Bloom filter is widely used in many network applications, such as Web caching, P2P collaboration, network measurements and monitoring, and packet processing. For example, Dharmapurikar et al. [34] proposed parallel Bloom filters to significantly improve the performance of DPI. Bonomi et al. [4] proposed an approximate concurrent state machine, where d-left hashing and fingerprinting schemes [35] were used to build a stateful Bloom filter, thus achieving fast per-flow state lookups.

Bloom filters and their variants have been widely adopted to accelerate hardware hash tables. Kirsch et al. [36] proposed alternative simple construction schemes of FHT to reduce the off-chip memory accesses and memory requirements. These schemes use on-chip simple summaries and an off-chip multi-level hash table to implement the functionality of FHT. Since an array of standard Bloom filters are used to comprise the on-chip simple summaries, these schemes require high memory bandwidth and only allow insertion. Kumar et al. [37, 38] proposed a segmented hash table (SHT) and peacock hash table (PHT) to reduce memory bandwidth and space requirements. Both the SHT and PHT are composed of an array of equal-size sub-tables, each maintaining an on-chip Bloom filter to reduce off-chip probes of the table memory. But these hashing schemes suffer from high overhead of sub-table rebalancing and low space utilization. Kirsch et al. [39] proposed multiple-choice hash tables with one move, where both a conservative scheme and second chance scheme were introduced to allow at most one item to be moved during an insertion, which increased the space utilization but did not support deletion.

To reduce the memory requirements, Yu et al. [40] proposed a multi-predicate Bloom-filtered hash table (MBHT). The MBHT comprises two multi-predicate Bloom filters (MBFs) in on-chip memory. In each MBF, an index of each off-chip item is split apart into several groups of bits, and each group uses several parallel standard Bloom filters in on-chip memory to represent the overall off-chip items. The MBHT uses left and right MBFs to support the incremental updates due to the left/right register, but suffers from high update overhead and many false positive memory accesses, without support for non-contiguous off-chip items. Our proposed ISBF has the same functionality as the MBF, but outperforms the MBF in three aspects. First, the ISBF uses on-chip parallel CBFs instead of standard Bloom filters to support both insertion and deletion. Second, both the lazy deletion algorithm and vacant insertion algorithm are proposed to support the incremental updates on non-contiguous items. Finally, the tradeoffs between the on-chip memory requirements and false positive memory accesses are considered, which indicates how to set proper parameters to minimize the false positive probability, thus further improving the performance.

3 Index-split Bloom filter

To achieve line-speed packet processing and reduce the memory requirements, we propose the ISBF to support time/space-efficient lookups. The ISBF consists of several groups of on-chip parallel CBFs and a list of off-chip items. The key idea of ISBF is as follows. Given a set of n off-chip items, an index of each item is split apart into B group, each containing b bits, where $B = \lceil \log_2 n/b \rceil$. Each group of bits splits the overall items into 2^b subsets, each represented by an on-chip CBF, and thus a total of 2^b

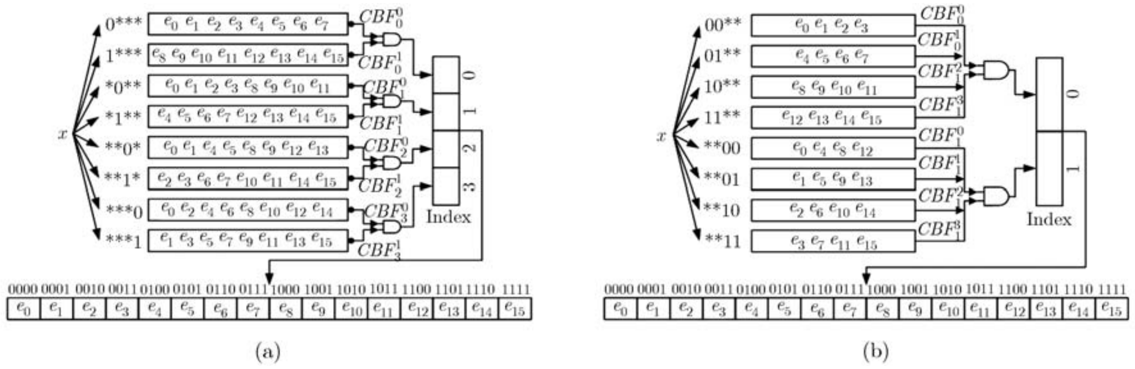

Figure 1 Examples of *b*-bit ISBF. (a) 1-bit; (b) 2-bit.

parallel CBFs per group are constructed in on-chip memory. When an item y is queried, B groups of on-chip CBFs are checked in parallel to constitute an index of an item candidate x, and a final decision is made by checking whether x matches y.

Figure 1 illustrates some examples of b-bit ISBF for a set of off-chip items $\{e_0, e_1, \ldots, e_{15}\}$. As shown in Figure 1(a), in the 1-bit ISBF, an index of each item is split into four groups, each containing one bit, and each group splits the set of items into two subsets, each represented by a CBF. Hence, the 1-bit ISBF embodies four groups of total eight parallel CBFs in on-chip memory. As shown in Figure 1(b), in the 2-bit ISBF, an index of each item is split into two groups, each containing two bits, and each group splits the set of items into four subsets, each represented by a CBF. Hence, the 2-bit ISBF embodies two groups of total eight parallel CBFs in on-chip memory.

In the b-bit ISBF, CBF_i^j denotes the jth parallel CBF of the ith group. In Figure 1(a), CBF_0^0 and CBF_0^1 denote the 0th and 1st CBFs of the 0th group respectively. When an item y is queried, CBF_0^0 and CBF_0^1 separately generate the 1-bit y_0^0 and y_0^1, for $y_0^0, y_0^1 \in \{0, 1\}$, and thus the 0th bit y_0 of an index is returned. Similarly, the 1st, 2nd, and 3rd groups of parallel CBFs return the 1st bit y_1, 2nd bit y_2, and 3rd bit y_3 respectively. All the bits y_0, y_1, y_2, and y_3 are combined to form a base-2 index $(y_0y_1y_2y_3)_2$. In Figure 1(b), CBF_0^0, CBF_0^1, CBF_0^2, and CBF_0^3 denote the 0th, 1st, 2nd, and 3rd CBFs of the 0th group respectively. When an item y is queried, the 0th group generates the 2-bit y_0^0, y_0^1, y_0^2 and y_0^3, for $y_0^0, y_0^1, y_0^2, y_0^3 \in \{0, 1, 2, 3\}$, and thus the 1-2th bit y_0 of an index is returned. The 1st group also generates the 2-3th bit y_1. Finally, a base-4 index $(y_0y_1)_4$ is formed.

Figure 2 illustrates a query example in the 1-bit ISBF, where there are n=16 off-chip items, and k=3 hash functions. When an item y is queried, y is hashed to three buckets $\{3, 16, 21\}$ of each on-chip CBF, which are checked in parallel to see whether y is in each CBF. In Figure 2, CBF_0^1, CBF_1^0, CBF_2^0, and CBF_3^1 separately generate the 1-bit 1, 0, 0, and 1, and thus an index $(1001)_2$=9 is returned. Finally, the 9th off-chip item e_9 is accessed and a decision is made by checking whether e_9 matches y. Note that when all k counter values indexed by hashing y are more than 0, CBF_i^j in the b-bit ISBF generates the ith b-bit value j.

3.1 Lazy deletion algorithm

When an item is deleted, the ISBF needs to adjust indexes of other off-chip items and reconstruct all on-chip CBFs, which leads to high deletion overhead, without support for dynamically changed items. In this section, we propose a lazy deletion algorithm to significantly reduce the deletion overhead of ISBF. This algorithm works as follows. An on-chip deletion bitmap is exploited to record states of all off-chip items, where 0 denotes a non-deletion state while 1 denotes a deletion state. When an item x is deleted, the state of x in the deletion bitmap is set at 1, and at the same time x is deleted from each group of on-chip parallel CBFs, while not adjusting indexes of other off-chip items behind x.

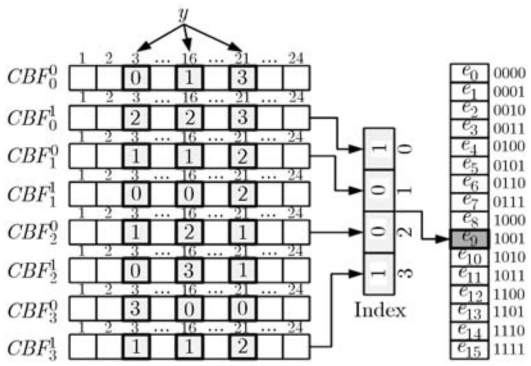

Figure 2 Query example in ISBF.

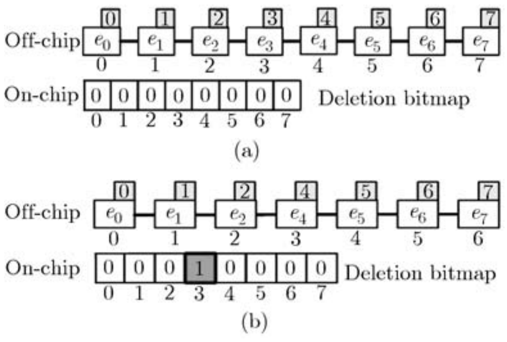

Figure 3 Lazy deletion of item e_3. (a)Before deleting e_3; (b) after deleting e_3.

```
Algorithm 1: Lazy deletion in ISBF
DeleteEntity($x$)
1: index-$GetEntityIndex(x)$;
2: DeletionBitmap[index]=1;
3: for ($i$=1 to $d$) do
4:    $B_i$=$GetBucket(i)$;
5:    $F_i$=$GetBloomFilter(B_i$, index);
6:    for ($p$=1 to $k$) do
7:      if ($h_p(x) \neq h_q(x)$ and  $p < q$) then
8:            $F_i$ Counter$^{hp(x)}$–;
9:        end if
10: end for
11: end for
```

Figure 4 Pseudo-code of lazy deletion algorithm.

```
Algorithm 2: Query in ISBF
QueryEntity($x$)
1: index=0;
2: for ($i$=1 to $d$) do
3:    $B_i$=$GetBucket(i)$;
4:   for ($j$=1 to $f$) do
5:       if (true==$BloomFilterContain(F_j$, x)) then
6:          index[$i$]=index[$i$]—$j$;
7           end if
8:        end for
9: endf or
10: address=index-$Popcount$(index);
11: if (EnfityArray[address]==x) then
12:   return ture;
13: else
14:   return false;
15: end if
```

Figure 5 Pseudo-code of parallel query algorithm.

Figure 3 illustrates an example of lazy deletion, where an item e_3 is deleted from a set of off-chip items $\{e_0, e_1, \ldots, e_7\}$. In Figure 3, each off-chip item has a logical index denoted by an upper number, and a physical address denoted by a lower number. As shown in Figure 3(a), before e_3 is deleted, the on-chip deletion bitmap is initially set at zeros. Figure 3(b) shows that after e_3 is deleted, the third state of e_3 in the on-chip deletion bitmap is set at 1, and other off-chip items $\{e_4, e_5, e_6, e_7\}$ behind e_3 keep their logical indexes invariable, while only adjusting their physical addresses with one left move.

Figure 4 describes the pseudo-codes of lazy deletion algorithm. When an item x is deleted, a function $GetEntityIndex()$ is called to get the index of x in off-chip memory and set the state of x in the on-chip deletion bitmap at 1. In d groups of on-chip parallel CBFs, a function $GetBucket()$ is called to get the ith group of parallel CBFs, and a function $GetBloomFilter()$ is called to get the CBF F_i containing x from the ith group. Finally, k counters indexed by hashing x in each F_i are decremented, so that x is deleted from the corresponding CBFs.

Figure 5 describes the pseudo-codes of parallel query algorithm. When an item x is queried, a function $GetEntityIndex()$ is called to check whether x is in each CBF F_j. Thus, the logical index of x is returned. A function $Popcount()$ is called to count the number of ones before the state of x in the deletion bitmap, and the physical address of x is computed by the logical index minus the number of ones.

Figure 6 illustrates deletion and query examples in the 1-bit ISBF, where an item e_6 is deleted from a set of off-chip items $\{e_0, e_1, \ldots, e_{15}\}$, and an item y is subsequently queried. In Figure 6, as e_6 has a logical index $(0110)_2$=6, the sixth state of e_6 in the deletion bitmap is set at 1, and at the same time e_6 is deleted from an array of on-chip parallel CBFs, including CBF_0^0, CBF_1^1, CBF_2^1, and CBF_3^0. After e_6

is deleted from the set, other off-chip items behind e_6 keep their indexes invariable.

In Figure 6, when y is queried to check and see whether y is in the set, CBF_0^1, CBF_1^0, CBF_2^0, and CBF_3^1 separately generate 1, 0, 0, and 1 to form a logical index $(1001)_2$=9. Thus, the number of ones before the 9th state in the deletion bitmap is counted as 1, and the physical address of y is computed as $9-1$=8. Finally, we access the 8th off-chip item e_9 to check whether e_9 matches y.

3.2 Vacant insertion algorithm

When an item is inserted, the ISBF also needs to adjust indexes of other off-chip items and reconstruct all on-chip CBFs, which leads to high insertion overhead, without support for dynamically changed items. In this section, we propose a vacant insertion algorithm to significantly reduce the insertion overhead of ISBF. This algorithm works as follows. The on-chip deletion bitmap aforementioned is exploited to record states of all off-chip items, and states of vacant locations are ones. When an item x is inserted, one of vacant locations is randomly selected from the deletion bitmap for an insertion, and its state is reset at 0. The index of the vacant location is allocated to the logical index of x. The number of ones before the state of x in the deletion bitmap is counted to compute the physical address of x, and thus x is inserted into the physical address in off-chip memory, while keeping indexes of other off-chip items behind x invariable.

Figure 7 depicts an example of vacant insertion, where an item e_8 is inserted into a set of off-chip items $\{e_0, e_1, \ldots, e_7\}$. In Figure 7, each off-chip item has a logical index and physical address. As shown in Figure 7(a), before e_8 is inserted, the deletion bitmap has two vacant locations $\{3, 6\}$, whose states are ones. As shown in Figure 7(b), after e_8 is inserted, the 3rd vacant location is selected as the logical index of e_8, and its state is reset at 0. The number of ones before the 3rd state in the deletion bitmap is counted as 0, and thus the physical address of e_8 is computed as 3. Finally, e_8 is inserted into the 3rd physical address, and other off-chip items $\{e_4, e_5, e_7\}$ keep their logical indexes invariant.

Figure 8 describes the pseudo-codes of vacant insertion algorithm. When an item x is inserted, a function $RandomVacantIndex()$ is called to randomly select a vacant location from the deletion bitmap as the logical index of x. A function $Popcount()$ is called to count the number of ones before the vacant location in the deletion bitmap, so that the physical address of x is computed. Finally, x is inserted into an on-chip parallel CBF F_i of each group, and into the corresponding physical address in off-chip memory.

Figure 9 illustrates an insertion example in the 1-bit ISBF, where an item z is inserted into a set of off-chip items $\{e_0, e_1, \ldots, e_{15}\}$, from which e_2 and e_6 have been deleted. As shown in Figure 9, when z is inserted, the 2nd vacant location is selected from the deletion bitmap as the logical index of z, and its state is reset at 0. The number of ones before the 2nd index in the deletion bitmap is counted to compute the physical address of z as 2. Finally, we insert z into an array of on-chip parallel CBFs, including CBF_0^0, CBF_1^0, CBF_2^1, and CBF_3^0, and into the 2nd physical address in off-chip memory, while other off-chip items $\{e_7, e_8, \ldots, e_{15}\}$ behind z keep their logical indexes invariable.

4 Algorithmic analysis

In this section, we analyze the time and space complexity of ISBF, and explore the tradeoffs between the on-chip memory requirements and false positive off-chip memory accesses to optimize the performance.

The time complexity of ISBF embodies both the on-chip and off-chip memory accesses. The ISBF has $O(1)$ average on-chip memory accesses, due to performing lookups in several groups of parallel CBFs. In practice, the on-chip query and update operations are dominated by a function $Popcount()$. Fortunately, as modern microprocessors and network processors usually support the *Popcount* instruction [41], these on-chip operations run very fast.

Moreover, the speed ratio of on-chip memory access to off-chip memory access varies from 30 to 60 times [42]. Hence, we neglect the on-chip processing times, and focus on the off-chip memory accesses of primitive operations, which are one of the key performance metrics of ISBF.

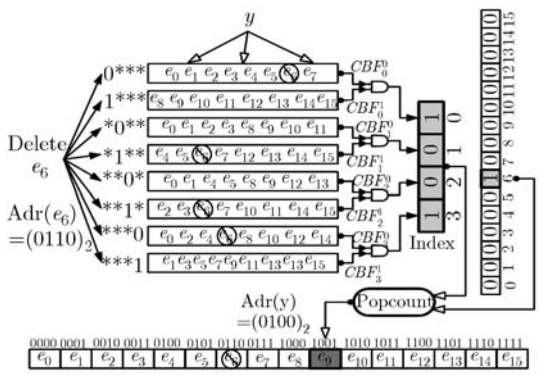

Figure 6 Deletion and query examples in ISBF.

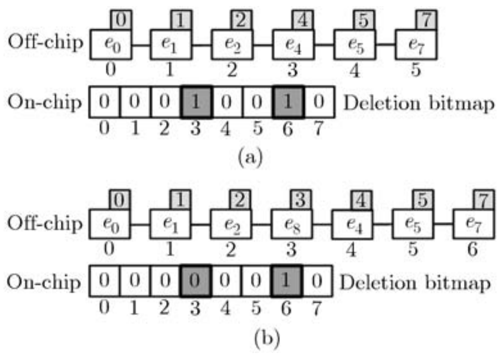

Figure 7 Vacant insertion of item e_8. (a) Before inserting e_8; (b) after inserting e_8.

Algorithm 3: Vacant insertion in ISBF

```
InsertEntity($x$)
1: index=$RandomVacantIndex$(DeletionBitmap);
2: DeletionBitmap[index]=0;
3: address=index–$Popcount$(DeletionBitmap, index);
4: InsertEnityArray($x$, address, index);
5: for ($i$=1 to $d$) do
6:     $B_i = GetBucket(i)$;
7:     $F_i = GetBloomFilter(B_i$, index);
8:     for ($p$=1 to $k$) do
9:         if ($h_p(x) \neq h_q(x)$ and $p < q$) then
10:            $F_i$ Counter$^{hp(x)++}$;
11:        end if
12:    end for
13: end for
```

Figure 8 Pseudo-codes of vacant insertion algorithm.

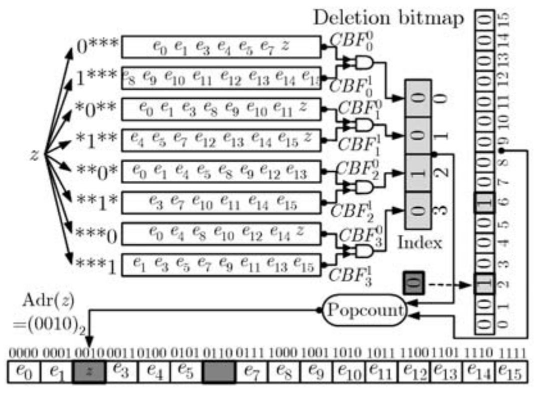

Figure 9 ISBF.

Table 1 Average off-chip memory accesses

	Insert	Delete	Query
TriBiCa	$O(n\log n)$	$O(n\log n)$	$O(1)$
SFHT	$O(nk^2/m+k)$	$O(k)$	$O(1)$
ISBF	$O(1)$	$O(1)$	$O(1)$

Table 1 summarizes the average off-chip memory accesses of insertion, deletion, and query. When an item is inserted or deleted, the TriBiCa needs to reconstruct the on-chip bitmap trie, which leads to $O(n\log n)$ average off-chip memory accesses of insertion and deletion. The SFHT has $O(nk^2/m+k)$ and $O(k)$ average off-chip memory accesses of insertion and deletion respectively, due to traversing off-chip linked lists and duplicating off-chip items. As both the lazy deletion algorithm and vacant insertion algorithm are used to support the incremental updates, without adjusting other related off-chip items, the ISBF has $O(1)$ average off-chip memory accesses of insertion and deletion. In addition, the ISBF has the same $O(1)$ average off-chip memory accesses of query as the TriBiCa and SFHT. Hence, the ISBF not only supports the incremental updates but also provides a guarantee for good lookup performance, due to achieving $O(1)$ average off-chip memory accesses of insertion, deletion, and query.

The space complexity of ISBF comprises both the on-chip and off-chip memory sizes. The on-chip memory size refers to the size of on-chip CBFs or on-chip bitmap trie, while the off-chip memory size refers to the number of off-chip items. As on-chip memory is small and expensive, the on-chip memory size is the key to the performance of ISBF. Note that unlike the CBF, the ISBF supports both set-membership queries whether an item is in a set, and item-address queries where an item is in a set. Hence, we mainly compare the on-chip memory size of the ISBF with that of the TriBiCa and SFHT.

In the ISBF, we assume that there are n items, k hash functions, and $m_{ISBF}=\alpha n$ buckets in each parallel CBF, for ($\alpha \geqslant 1$). As the b-bit ISBF consists of $\lceil \log_2 n/b \rceil$ groups of parallel CBFs, the on-chip memory size of ISBF is computed as follows:

$$M_{\text{ISBF}} = \lceil \log_2 n/b \rceil \times \alpha n \approx \frac{\alpha}{b} \cdot n \lceil \log_2 n \rceil. \tag{1}$$

Eq. (1) shows that the on-chip memory size of ISBF grows with the increase of split bits b.

In the TriBiCa, the height of bitmap trie is $\lceil \log_2 n \rceil$, and the size of nodes at each layer is $m_{\text{TriBiCa}}=2\beta n$. The on-chip memory size of TriBiCa is computed as follows:

$$M_{\text{TriBiCa}} = \lceil \log_2 n \rceil \times 2\beta n = 2\beta \cdot n \lceil \log_2 n \rceil. \tag{2}$$

In the SFHT, the number of buckets in an on-chip CBF is $m_{\text{SFHT}}=\gamma n$, and each bucket contains at least $\lceil \log_2 n \rceil + 4$ bits. The on-chip memory size of SFHT is computed as follows:

$$M_{\text{SFHT}} = \gamma n \times (\lceil \log_2 n \rceil + 4) = \gamma \cdot n \lceil \log_2 n \rceil + 4\gamma \cdot n. \tag{3}$$

Hence, the on-chip memory size ratio of the ISBF to the TriBiCa is computed as follows:

$$R_{I,T} = M_{\text{ISBF}}/M_{\text{TriBiCa}} \approx \frac{\alpha}{2b\beta}. \tag{4}$$

The on-chip memory size ratio of the ISBF to the SFHT is computed as follows:

$$R_{I,S} = M_{\text{ISBF}}/M_{\text{SFHT}} \approx \frac{\alpha}{b\gamma}. \tag{5}$$

When $\alpha=\beta=\gamma$, $R_{I,T}=1/2b$ and $R_{I,S}=1/b$, meaning that compared with the TriBiCa and SFHT, the ISBF reduces the on-chip memory size by $2b$ times and b times respectively. So the ISBF achieves space efficiency, thus significantly reducing the memory requirements.

As the CBF yields false positives, the ISBF may generate several possible indexes, which leads to increasing numbers of both the off-chip memory accesses and signature matching operations. In essence, the number of false positive indexes is dominated by the multiplication of the false positive bits generated by each parallel CBFs. For instance, in the 2-bit ISBF, we assume that the 0th group of parallel CBFs generates three bits, among which there are two false positives, while the 1st group generates two bits. Thus, total six indexes are returned, which means that there are six off-chip memory accesses for querying an item.

Assuming that each CBF is independent with the same false positive probability f, the average number of false positive indexes is computed as follows:

$$\begin{aligned} F_{\text{ISBF}} &= E\Bigg[\prod_{i=0}^{\lceil \log_2 n/b \rceil} X_i^f \Bigg] + E\Bigg[\prod_{i=0}^{\lceil \log_2 n/b \rceil} (X_i^t + 1) - 1 \Bigg] \\ &= (2^b f)^{\lceil \log_2 n/b \rceil} + ((2^b - 1)f + 1)^{\lceil \log_2 n/b \rceil} - 1l, \end{aligned} \tag{6}$$

where X_i^f is the number of false positive bits generated by the ith parallel CBF if all parallel CBFs generate false positives, and X_i^t is the number of false positive bits generated by the ith parallel CBF if only one of parallel CBFs generates a true negative. Note that both X_i^f and X_i^t obey the binomial distribution. When each parallel CBF yields the same false positive probability $f=(1-\mathrm{e}^{-kn/m})^k$ and has $m=\alpha n$ buckets, the average number of false positive indexes is computed as follows:

$$F_{\text{ISBF}} = (2^b(1-\mathrm{e}^{-k/\alpha})^k)^{\lceil \log_2 n/b \rceil} + ((2^b-1)(1-\mathrm{e}^{-k/\alpha})^k + 1)^{\lceil \log_2 n/b \rceil} - 1. \tag{7}$$

When the on-chip memory size of ISBF is kept constant, that is $c=\alpha/b$, then the average number of false positive indexes is computed as follows:

$$F_{\text{ISBF}} = (2^b(1-\mathrm{e}^{-k/(cb)})^k)^{\lceil \log_2 n/b \rceil} + ((2^b-1)(1-\mathrm{e}^{-k/(cb)})^k + 1)^{\lceil \log_2 n/b \rceil} - 1. \tag{8}$$

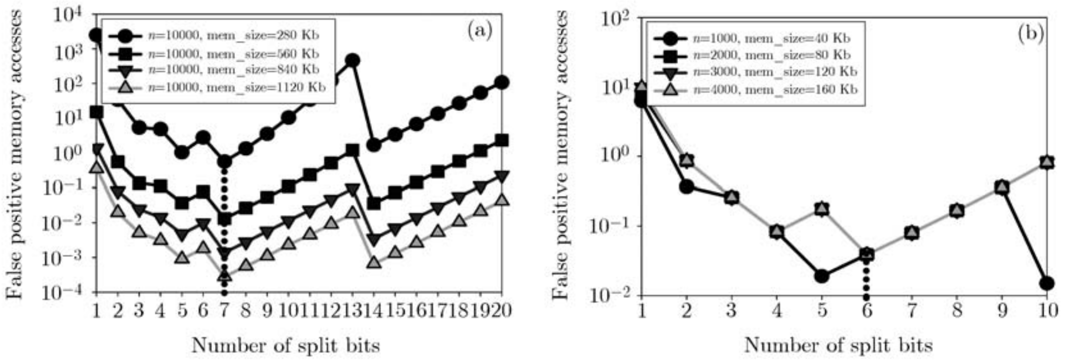

Figure 10 False positive off-chip memory accesses in ISBF. (a) 10000 items; (b) 1000–40000 items.

Eq. (8) shows that given that there are n items and k hash functions, the average number of false positive indexes of ISBF is the function of split bits b.

When the on-chip memory size M_{ISBF} is kept constant, we explore the relationship between the average number F_{ISBF} of false positive indexes and the number b of split bits by using MATLAB. Figure 10 depicts the false positive off-chip memory accesses of ISBF under varied items n, where there are k=6 hash functions for each parallel CBF. As seen in Figure 10(a), when n=10000 and b=7, the false positive off-chip memory accesses is minimized. As seen in Figure 10(b), when n=1000–4000 and b=6, the false positive off-chip memory accesses is nearly minimized.

Hence, we can select proper parameters parameters to optimize the performance of ISBF. In the following experiments, when there are n=1000–4000 items, the number of split bits are set at b=6, and the ratio of the number of buckets to the number of items is set at α=20, thus minimizing the false positive off-chip memory accesses of ISBF.

5 Experimental evaluation

We design and implement the TriBiCa, SFHT, and ISBF with C/C++, and run these programs on a machine with Intel Core2 Duo 2.13 GHz CPU and 2 GB main memory. In this section, extensive simulation experiments on both synthetic and real rule sets are conducted to compare the ISBF with the TriBiCa and SFHT. We measure the performance metrics, including the off-chip memory accesses and processing times of insertion, deletion, and query, as well as both the on-chip and off-chip memory sizes.

5.1 On synthetic rule set

In the experiments on synthetic rule set, a string stream generator is implemented to synthesize an evaluation set, including signature strings and test strings. From the alphabet {‘A’–‘J’}, the generator randomly generates 200–4000 signature strings and 10000 test strings, each containing 4 bytes. The simulation experiments are conducted as follows. First, 200–4000 signature strings are stored into the TriBiCa, SFHT, and ISBF respectively. Second, 100 continuous operations are performed in the TriBiCa, SFHT, and ISBF respectively, in each of which 100–2000 signature strings are deleted and inserted successively, and 10000 test strings are subsequently queried.

To optimize the lookup performance, we set proper parameters of the TriBiCa, SFHT, and ISBF respectively. In the TriBiCa, we set the ratio of the size of data bitmap to the number of items at c=m/n=10, and set one hash function at each layer of the on-chip bitmap trie. In the SFHT, we set the ratio of the number of buckets to the number of items in the on-chip CBF at c=m/n=10, and set the number of hash functions at k=$\ln 2(m/n)$=6 to minimize the false positive probability. In the ISBF, we set the ratio of the number of buckets to the number items in each on-chip parallel CBF at c=m/n=20, set the number of hash functions at k=6, and set the number of split bits at b=6, thus minimizing the false positive off-chip memory accesses.

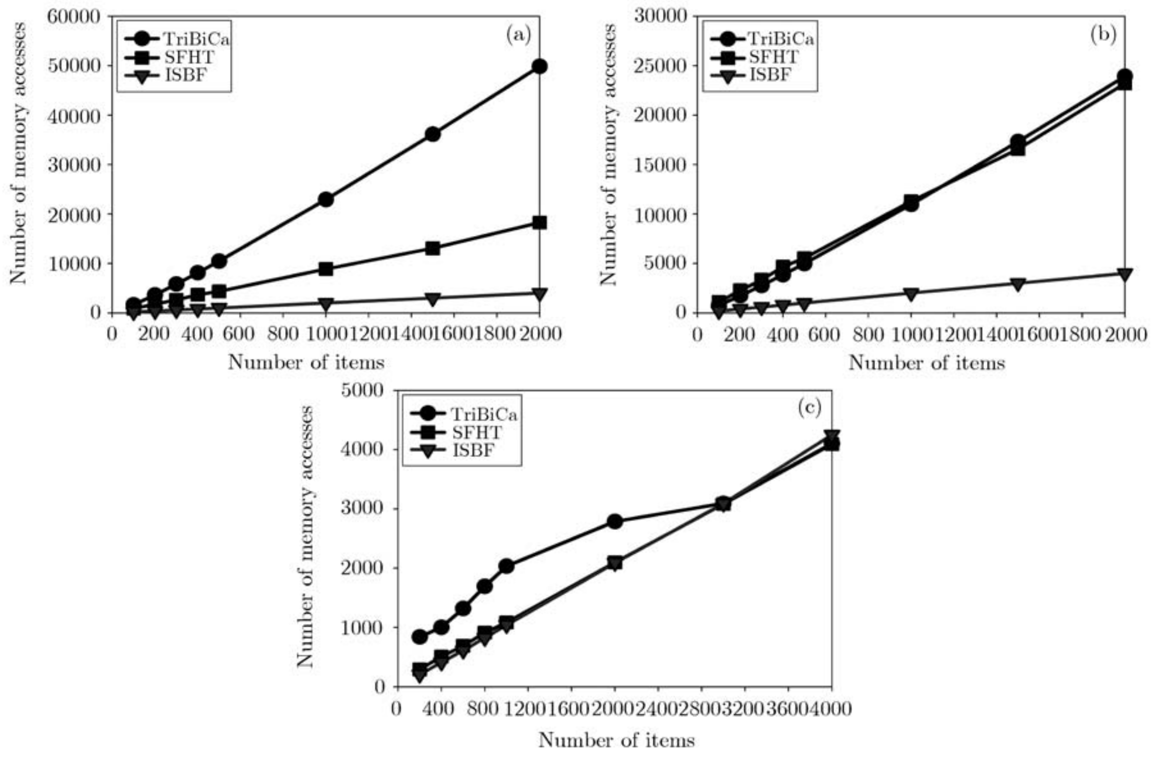

Figure 11 Off-chip memory accesses of primitive operations. (a) Insertion; (b) deletion; (c) query.

Figure 11 depicts the off-chip memory accesses of insertion, deletion, and query. As seen in Figure 11(a), when 100–2000 signature strings are inserted, the TriBiCa has 1629–49863 off-chip memory accesses, and the SFHT has 877–18287 off-chip memory accesses, while the ISBF has only 200–4000 off-chip memory accesses. Figure 11(a) shows that compared with the TriBiCa and SFHT, the ISBF reduces the off-chip memory accesses of insertion by 8.1–12.5 times and 4.4–4.6 times respectively.

As seen in Figure 11(b), when 100–2000 signature strings are deleted, the TriBiCa has 764–23932 off-chip memory accesses, and the SFHT has 1113–23212 off-chip memory accesses, while the ISBF has only 200–4000 off-chip memory accesses. Figure 11(b) shows that compared with the TriBiCa and SFHT, the ISBF reduces the off-chip memory accesses of deletion by 3.8–6 times and 5.5–5.8 times respectively.

In Figure 11(c), 200–4000 signature strings are stored, and 10000 test strings are queried. Figure 11(c) shows that when 200–2000 signature strings are stored, the ISBF reduces the off-chip memory accesses of query by 25%–75.5% and 0.4%–30% in comparison with the TriBiCa and SFHT respectively. When more than 3000 signature strings are stored, the ISBF has almost the same off-chip memory accesses of query as the TriBiCa and SFHT.

Figure 12 depicts the processing times of insertion and deletion. As seen in Figure 12(a), when 100–2000 signature strings are inserted, the TriBiCa has 93–26894 ms processing times, and the SFHT has 145–52129 ms processing times, while the ISBF has only 14–585 ms processing times. Figure 12(a) shows that compared with the TriBiCa and SFHT, the ISBF reduces the processing times of insertion by 6.5–45.9 times and 10.1–89 times respectively.

As seen in Figure 12(b), when 100–2000 signature strings are deleted, the TriBiCa has 32–7083 ms processing times, and the SFHT has 147–52102 ms processing times, while the ISBF has only 15–684 ms processing times. Figure 12(b) shows that compared with the TriBiCa and SFHT, the ISBF reduces the processing times of deletion by 2.2–10.4 times and 10.1–76.2 times respectively.

Figure 13 depicts both the on-chip and off-chip memory sizes. As seen in Figure 13(a), when 200–4000 signature strings are stored, the TriBiCa has the on-chip memory size of 40960–983040 bits, and the SFHT has the on-chip memory size of 72000–1440000 bits, while the ISBF has only the on-chip memory size of 36680–654240 bits. Figure 13(a) shows that compared with the TriBiCa and SFHT, the ISBF reduces the on-chip memory size by 10.4%–41.9% and 39.3%–54.6% respectively.

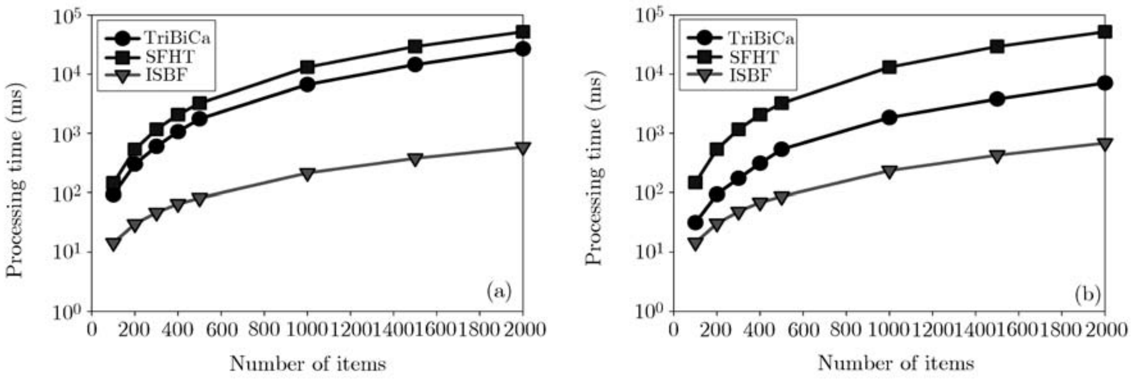

Figure 12 Processing times of insertion and deletion. (a) Insertion; (b) deletion.

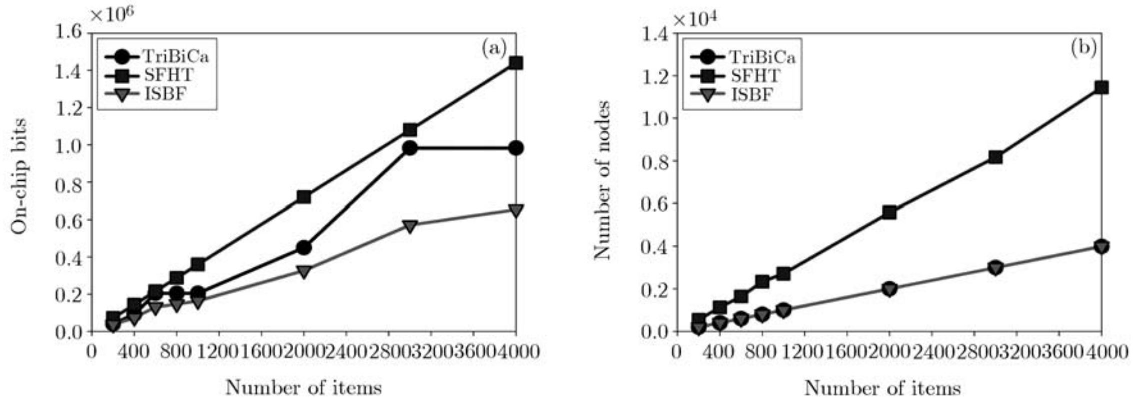

Figure 13 On-chip and off-chip memory sizes. (a) On-chip memory; (b) off-chip memory.

As seen in Figure 13(b), when 200–4000 signature strings are stored, the ISBF has the same 200–400 off-chip items as the TriBiCa, while the SFHT has 546–11447 off-chip items. Figure 13(b) shows that the ISBF has the same off-chip memory size as the TriBiCa, and reduces 2.7–2.9 times off-chip memory size in comparison as the SFHT.

5.2 On real rule set

In the experiments on real rule set, we use the Snort 2.7 rule set of 4077 signature strings. The Snort rule set is partitioned into four subsets: Rule-1, Rule-2, Rule-3, and Rule-4. Note that Rule-1 has 956 signature strings, Rule-2 has 1170 signature strings, Rule-3 has 945 signature strings, and Rule-4 has 1006 signature strings. In the simulation experiments, we repeat 100 times of queries, each containing all 4077 signature strings. Similarly, we select proper parameters of the TriBiCa, SFHT, and ISBF to optimize the lookup performance.

Figure 14 depicts both the off-chip memory accesses and processing times of query in Snort. Figure 14(a) shows that compared with the TriBiCa and SFHT, the ISBF reduces the off-chip memory accesses of query by 8.4%–15.8% and 1%–7.2% respectively. Figure 14(b) shows that compared with the TriBiCa and SFHT, the ISBF reduce the processing times of query by 1.7 times and 11–13.6 times respectively.

Figure 15 depicts the memory sizes and false positive off-chip memory accesses in Snort. Figure 15(a) shows that compared with the TriBiCa and SFHT, the ISBF reduces the on-chip memory size by 20.1%–42.4% and 38.3%–54.8% respectively. Figure 15(b) shows that the ISBF has the same off-chip memory size as the TriBiCa, but reduces 1.8–1.9 times off-chip memory size in comparison with the SFHT. In addition, Figure 15(c) shows that compared with the TriBiCa and SFHT, the ISBF reduces the false positive off-chip memory accesses by 4.5–9.1 times and 1.2–4.5 times respectively.

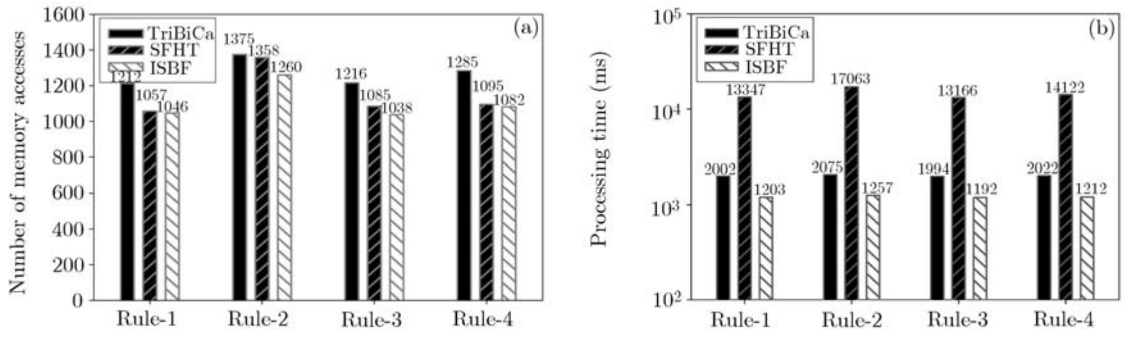

Figure 14 Off-chip memory accesses and processing times of query. (a) Off-chip memory accesses; (b) processing times.

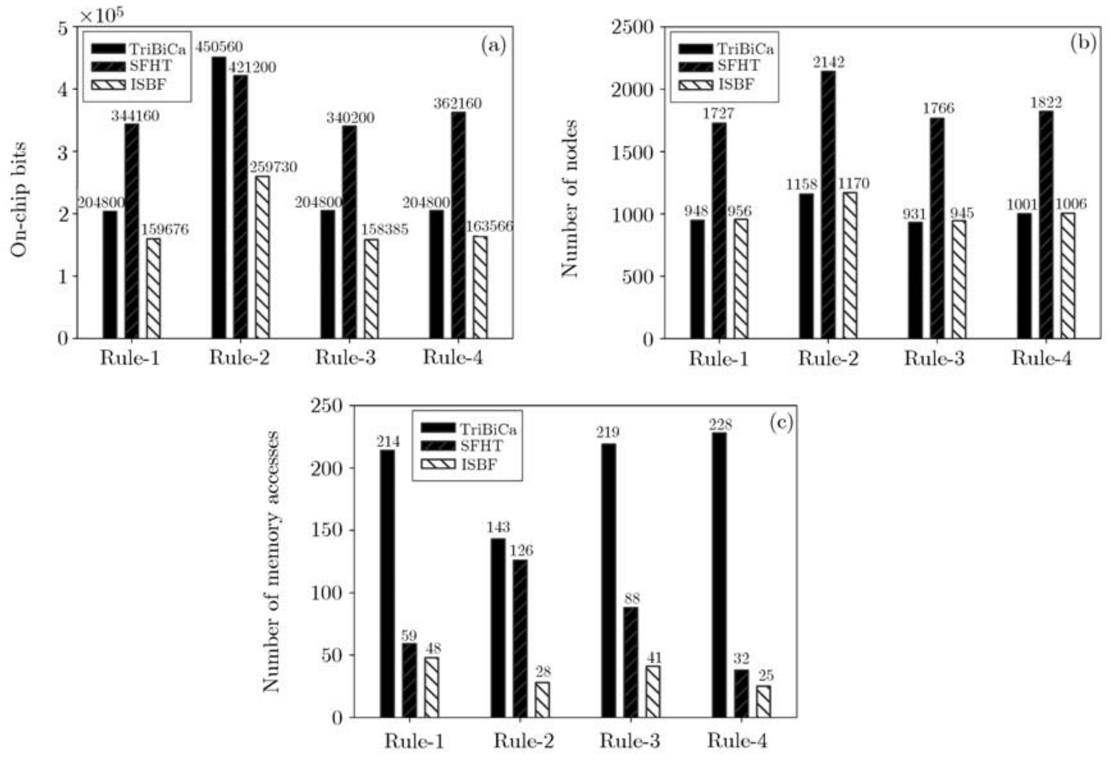

Figure 15 Memory sizes and false positive off-chip memory accesses. (a) On-chip memory; (b) off-chip memory; (c) false positive memory accesses.

6 Conclusions

In this paper, we propose the ISBF to achieve line-speed packet processing and reduce the memory requirements. The key idea of ISBF is as follows. An index of each off-chip item is split apart into B groups, each containing b bits; each group of bits splits the overall items into 2^b subsets, each represented by a CBF, and thus a total of 2^b parallel CBFs per group are constructed in on-chip memory. When an item y is queried, B groups of on-chip parallel CBFs constitute an index of an item candidate x, and a final decision is made by checking whether x matches y. Furthermore, we propose the lazy deletion algorithm and vacant insertion algorithm to reduce the update overhead of the ISBF. These algorithms use an on-chip deletion bitmap to update on-chip parallel CBFs, without adjusting other related off-chip items.

We theoretically show that the ISBF achieves $O(1)$ average off-chip memory accesses of insertion, deletion, and query; compared with the TriBiCa and SFHT, the ISBF reduces the on-chip memory requirements by $2b$ times and b times respectively, where b is the number of split bits. In addition, the ISBF is tuned to achieve the time/space tradeoffs between the on-chip memory size and false positive off-chip memory accesses.

Experimental results show that compared with the TriBiCa and SFHT, the ISBF reduces the off-chip memory accesses of insertion by 12.5 times and 4.6 times respectively as well as those of deletion by 6 times and 5.8 times respectively, and has almost the same off-chip memory accesses of query as the TriBiCa and SFHT; the ISBF reduces the processing time of insertion by 45.9 times and 89 times respectively as well as those of deletion by 10.4 times and 76.2 times respectively; the ISBF reduces 41.9% and 54.6% on-chip memory size respectively, and has the same off-chip memory size as the TriBiCa, while reducing 2.9 times off-chip memory in comparison with the SFHT. Therefore, the ISBF is a time/space data structure, which is suitable for many Internet applications, such as IP lookups, packet classification, and traffic monitoring and accounting.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant Nos. 90718008, 60673155), and the National Basic Research Program of China (Grant No. 2007CB310702).

References

1 Paxson V, Asanovic K, Dharmapurikar S, et al. Rethinking hardware support for network analysis and intrusion prevention. In: Proceedings of USENIX Workshop on Hot Topics in Security 2006. Vancouver: USENIX Press, 2006

2 Estan C, Varghese G. New directions in traffic measurement and accounting. In: Proceedings of ACM SIGCOMM 2001. San Diego: ACM Press, 2001

3 Lakshminarayanan K, Rangarajan A, Venkatachary S. Algorithms for advanced packet classification with ternary CAMs. In: Proceedings of ACM SIGCOMM 2005. Philadelphia: ACM Press, 2005

4 Bonomi F, Mitzenmacher M, Panigrapy R, et al. Beyond Bloom filters: from approximate membership checks to approximate state machines. In: Proceedings of ACM SIGCOMM 2006. Pisa: ACM Press, 2006

5 Roesch M. Snort c lightweight intrusion detection for networks. In: Proceedings of LISA 1999. Seattle: USENIX Press, 1999

6 Paxon V. Bro: A system for detecting network intruders in real-time. Comput Networks, 1999, 31: 2435–2463

7 Levandoski J, Sommer E, Strait M. Application layer packet classifier for Linux. http://l7-filter.sourceforge.net, 2008

8 Sen S, Spatscheck O, Wang D. Accurate, scalable in-network identification of P2P traffic using application signatures. In: Proceedings of WWW 2004. Manhattan: ACM Press, 2004

9 Karagiannis T, Broido A, Faloutsos M, et al. Transport layer identification of p2p traffic. In: Proceedings of IMC 2004. Taormina: ACM Press, 2004

10 Clark C R, Schimmel D E. Scalable pattern matching on high-speed networks. In: Proceedings of IEEE FCCM 2004. Napa: IEEE Press, 2004

11 Sourdis I, Pnevmatikatos D. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In: Proceedings of IEEE FCCM 2004. Napa: IEEE Press, 2004

12 Yu F, Katz R, Lakshman T V. Gigabit rate packet pattern-matching using TCAM. In: Proceedings of IEEE ICNP 2004. Berlin: IEEE Press, 2004

13 Piyachon P, Luo Y. Efficient memory utilization on network processors for deep packet inspection. In: Proceedings of ACM/IEEE ANCS 2006. San Jose: ACM Press, 2006

14 Lu H, Zheng K, Liu B, et al. A memory-efficient parallel string matching architecture for high-speed intrusion detection. IEEE J Select Areas Commun, 2006, 34: 1793–1804

15 Artan N S, Chao H J. TriBiCa: trie bitmap content analyzer for high-speed network intrusion detection. In: Proceedings of IEEE INFOCOM 2007. Anchorage: IEEE Press, 2007

16 Song H, Dharmapurikar S, Turner J, et al. Fast hash table lookup using extended Bloom filter: an aid to network processing. In: Proceedings of ACM SIGCOMM 2005. Philadelphia: ACM Press, 2005

17 Fan L, Cao P, Almeida J, et al. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans Network, 2000, 8: 281–293

18 Tan L, Brotherton B, Sherwood T. Bit-split string-matching engines for intrusion detection and prevention. ACM Trans Architect Code Opt, 2006, 3: 3–34

19 Brodie B C, Cytron R K, Taylor D E. A scalable architecture for high-throughput regular-expression pattern matching.

In: Proceedings of ISCA 2006. Boston: IEEE Press, 2006
20 Lunteren J. High performance pattern-matching for intrusion detection. In: Proceedings of IEEE INFOCOM 2006. Barcelona: IEEE Press, 2006
21 Yu F, Chen Z, Diao Y, et al. Fast and memory-efficient regular expression matching for deep packet inspection. In: Proceedings of ACM/IEEE ANCS 2006. San Jose: ACM Press, 2006
22 Tuck N, Sherwood T, Calder B, et al. Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of IEEE INFOCOM 2004. Hong Kong: IEEE Press, 2004
23 Aho A V, Corasick M J. Efficient string matching: an aid to bibliographic search. Commun ACM, 1975, 18: 333–340
24 Dharmapurikar S, Lockwood J. Fast and scalable pattern matching for content filtering. In: Proceedings of ACM/IEEE ANCS 2005. Princeton: ACM Press, 2005
25 Hua N, Song H, Lakshman T V. Variable-stride multi-pattern matching for scalable deep packet inspection. In: Proceedings of IEEE INFOCOM 2009. Rio de Janeiro: IEEE Press, 2009
26 Song T, Zhang W, Wang D, et al. A memory efficient multiple pattern matching architecture for network security. In: Proceedings of IEEE INFOCOM 2008. Phoenix: IEEE Press, 2008
27 Kumar S, Dharmapurikar S, Yu F, et al. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: Proceedings of ACM SIGCOMM 2006. Pisa: ACM Press, 2006
28 Kumar S, Turner J, Williams J. Advanced algorithms for fast and scalable deep packet inspection. In: Proceedings of ACM/IEEE ANCS 2006. San Jose: ACM Press, 2006
29 Becchi M, Cadambi S. Memory-efficient regular expression search using state merging. In: Proceedings of IEEE INFOCOM 2007. Anchorage: IEEE Press, 2007
30 Smith R, Estan C, Jha S. XFA: Faster signature matching with extended automata. In: Proceedings of IEEE Symposium on Security and Privacy 2008. Oakland: IEEE Press, 2008
31 Smith R, Estan C, Jha S, et al. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In: Proceedings of ACM SIGCOMM 2008. Seattle: ACM Press, 2008
32 Kumar S, Chandrasekaran B, Turner J, et al. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: Proceedings of ACM/IEEE ANCS 2007. Orlando: ACM Press, 2007
33 Broder A, Mitzenmacher M. Network applications of Bloom filters: A survey. Internet Math, 2004, 1: 485–509
34 Dharmapurikar S, Krishnamurthy P, Sproull T S, et al. Deep packet inspection using parallel Bloom filters. IEEE Micro, 2004, 24: 52–61
35 Broder A, Mitzenmacher M. Using multiple hash functions to improve IP lookups. In: Proceedings of IEEE INFOCOM 2001. Anchorage: IEEE Press, 2001
36 Kirsch A, Mitzenmacher M. Simple summaries for hashing with choices. IEEE/ACM Trans Network, 2008, 16: 218–231
37 Kumar S, Crowley P. Segmented hash: an efficient hash table implementation for high performance networking subsystems. In: Proceedings of ACM/IEEE ANCS 2005. Princeton: ACM Press, 2005
38 Kumar S, Turner J, Crowley P. Peacock hashing: deterministic and updatable hashing for high performance networking. In: Proceedings of IEEE INFOCOM 2008. Phoenix: IEEE Press, 2008
39 Kirsch A, Mitzenmacher M. The power of one move: hashing schemes for hardware. In: Proceedings of IEEE INFOCOM 2008. Phoenix: IEEE Press, 2008
40 Yu H, Mahapatra R. A memory-efficient hashing by multi-predicate Bloom filters for packet classification. In: Proceedings of IEEE INFOCOM 2008. Phoenix: IEEE Press, 2008
41 Hua N, Lin B, Xu J. Rank-indexed hashing: a compact construction of Bloom filters and variants. In: Proceedings of IEEE ICNP 2008. Orlando: IEEE Press, 2008
42 Varghese G. Network algorithms: an interdisciplinary approach to designing fast network devices. San Fransisco, CA: Morgan Kaufmann Publishers, 2004