Cuckoo Filter: Practically Better Than Bloom

Bin Fan (CMU/Google)
David Andersen (CMU)
Michael Kaminsky (Intel Labs)
Michael Mitzenmacher (Harvard)
What is Bloom Filter? A Compact Data Structure Storing Set-membership

• Bloom Filters answer “is item $x$ in set $Y$” by:
  • “definitely no”, or
  • “probably yes” with probability $\varepsilon$ to be wrong

• Benefit: not always precise but highly compact
  • Typically a few bits per item
  • Achieving lower $\varepsilon$ (more accurate) requires spending more bits per item
Example Use: Safe Browsing

Lookup("www.binfan.com")

It is Good!

Please verify "www.binfan.com"

Probably Yes!

No!

Known Malicious URLs

Stored in Bloom Filter

Scale to millions URLs
Bloom Filter Basics

A Bloom Filter consists of $m$ bits and $k$ hash functions.

Example: $m = 10$, $k = 3$

Insert($x$)  
\[ \text{hash}_1(x) \quad \text{hash}_2(x) \quad \text{hash}_3(x) \]
\[
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
\]

Lookup($y$) = not found  
\[ \text{hash}_1(y) \quad \text{hash}_2(y) \quad \text{hash}_3(y) \]
\[
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
\]
# Succinct Data Structures for Approximate Set-membership Tests

<table>
<thead>
<tr>
<th></th>
<th>High Performance</th>
<th>Low Space Cost</th>
<th>Delete Support</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bloom Filter</strong></td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Counting Bloom Filter</strong></td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Quotient Filter</strong></td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Can we achieve all three in practice?
Outline

• Background

• Cuckoo filter algorithm

• Performance evaluation

• Summary
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**\( (x) \): A hash value of \( x \)
  - Lower false positive rate \( \epsilon \), longer fingerprint
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**(x): A hash value of x
  - Lower false positive rate $\epsilon$, longer fingerprint

- **Insert**(x):
  - add **Fingerprint**(x) to hash table

\[
\begin{array}{cccccccc}
0: & 1: & FP(x) & 2: & FP(a) & 3: & FP(c) & 4: & FP(b) \\
\end{array}
\]
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint** \( (x) \): A hash value of \( x \)
  - Lower false positive rate \( \varepsilon \), longer fingerprint

- **Insert** \( (x) \):
  - Add **Fingerprint** \( (x) \) to hash table

- **Lookup** \( (x) \):
  - Search **Fingerprint** \( (x) \) in hashtable

\[
\begin{array}{c}
0: \\
1: \text{FP}(x) \\
2: \text{FP}(a) \\
3: \\
4: \text{FP}(c) \\
5: \\
6: \text{FP}(b) \\
7: \\
\end{array}
\]

Lookup \( (x) \) = found
Basic Idea: Store Fingerprints in Hash Table

- **Fingerprint**\((x)\): A hash value of \(x\)
  - Lower false positive rate \(\varepsilon\), longer fingerprint
- **Insert**\((x)\):
  - add **Fingerprint**\((x)\) to hash table
- **Lookup**\((x)\):
  - search **Fingerprint**\((x)\) in hashtable
- **Delete**\((x)\):
  - remove **Fingerprint**\((x)\) from hashtable

How to Construct Hashtable? 

FP(a)  
FP(b)  
FP(c)  
FP(x)
Perfect hashing: maps all items with no collisions

\{a, b, c, d, e, f\}

\(f(x)\)

FP(b)
FP(e)
FP(c)
FP(d)
FP(f)
FP(a)

( Minimal) Perfect Hashing:
No Collision but Update is Expensive
• Perfect hashing: maps all items with no collisions

\{a, b, c, d, e, f\}\\\downarrow\\\{a, b, c, d, e, g\}

\text{f}(x) = ?

\text{FP}(a) \quad \text{FP}(b) \quad \text{FP}(c) \quad \text{FP}(d) \quad \text{FP}(e) \quad \text{FP}(f)

• Changing set must recalculate \( f \) ➔ high cost/bad performance of update
Convention Hash Table: High Space Cost

- **Chaining**:
  - Pointers → low space utilization

- **Linear Probing**
  - Making lookups $O(1)$ requires large % table empty → low space utilization
  - Compare multiple fingerprints sequentially → more false positives
Cuckoo Hashing \cite{Pagh2004} Good But ..

- High Space Utilization
  - 4-way set-associative table: >95% entries occupied
- Fast Lookup: $O(1)$

Standard cuckoo hashing doesn’t work with \cite{Pagh2004} Cuckoo hashing.
Standard Cuckoo Requires Storing Each Item

\[ \text{Insert}(x) \]

\[ h_1(x) \]

\[ h_2(x) \]
Standard Cuckoo Requires Storing Each Item

Insert(x)

Rehash a: alternate(a) = 4
Kick a to bucket 4
Standard Cuckoo Requires Storing Each Item

Insert(x)

$h_2(x)$

Rehash c: alternate(c) = 1
Kick c to bucket 1

Rehash a: alternate(a) = 4
Kick a to bucket 4
Standard Cuckoo Requires Storing Each Item

Insert complete
(or fail if MaxSteps reached)

Rehash c: alternate(c) = 1
Kick c to bucket 1

Rehash a: alternate(a) = 4
Kick a to bucket 4

Insert(x)
Challenge: How to Perform Cuckoo?

• Cuckoo hashing requires rehashing and displacing existing items

With only fingerprint, how to calculate item’s alternate bucket?

Kick FP(a) to which bucket?

Kick FP(c) to which bucket?

FP(b)
FP(c)
FP(a)
We Apply Partial-Key Cuckoo

- Standard Cuckoo Hashing: two independent hash functions for two buckets
  
  \[ \text{bucket1} = \text{hash}_1(x) \]
  
  \[ \text{bucket2} = \text{hash}_2(x) \]

- Partial-key Cuckoo Hashing: use one bucket and fingerprint to derive the other \[\text{Fan2013}\]

  \[ \text{bucket1} = \text{hash}(x) \]
  
  \[ \text{bucket2} = \text{bucket1} \oplus \text{hash(FP}(x)) \]

To displace existing fingerprint:

\[ \text{alternate}(x) = \text{current}(x) \oplus \text{hash(FP}(x)) \]

\[\text{Fan2013}\] MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing
Partial Key Cuckoo Hashing

• Perform cuckoo hashing on fingerprints

Can we still achieve high space utilization with partial-key cuckoo hashing?
Fingerprints Must Be “Long” for Space Efficiency

- Fingerprint must be $\Omega(\log n/b)$ bits in theory
- $n$: hash table size, $b$: bucket size
- see more analysis in paper

Table Space Utilization

Table size: $n=128$ million entries

When fingerprint > 5 bits, high table space utilization

- $f$: fingerprint size in bits
Space Efficiency

More Space

More False Positive

bits per item to achieve $\varepsilon$

$\varepsilon$: target false positive rate

Lower bound
Space Efficiency

More Space

Bloom filter

Lower bound

More False Positive

ε: target false positive rate

bits per item to achieve ε

0.001% 0.01% 0.1% 1% 10%

0 20 40 60 80

More Space

25
Space Efficiency

More Space

\[ \varepsilon : \text{target false positive rate} \]

bits per item to achieve \( \varepsilon \)

- Bloom filter
- Cuckoo filter
- Lower bound

More False Positives

\( \varepsilon \): target false positive rate
Space Efficiency

**More Space**

- **Bloom filter**
- **Cuckoo filter**
- **Lower bound**

**Cuckoo filter + semisorting**

More compact than Bloom filter at 3%

ε: target false positive rate

More False Positive
Outline

• Background

• Cuckoo filter algorithm

• Performance evaluation

• Summary
Evaluation

• Compare cuckoo filter with
  • Bloom filter (cannot delete)
  • Blocked Bloom filter [Putze2007] (cannot delete)
  • d-left counting Bloom filter [Bonomi2006]
  • Cuckoo filter + semisorting
  • More in the paper

• C++ implementation, single threaded


Lookup Performance (MOPS)

<table>
<thead>
<tr>
<th>Method</th>
<th>Hit</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cuckoo</td>
<td>11.93</td>
<td>11.92</td>
</tr>
<tr>
<td>Cuckoo + semisort</td>
<td>6.28</td>
<td>6.45</td>
</tr>
<tr>
<td>d-left counting Bloom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocked Bloom (no deletion)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloom (no deletion)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lookup Performance (MOPS)

<table>
<thead>
<tr>
<th>Method</th>
<th>Hit</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cuckoo</td>
<td>11.93</td>
<td>11.92</td>
</tr>
<tr>
<td>Cuckoo + semisort</td>
<td>6.28</td>
<td>6.45</td>
</tr>
<tr>
<td>d-left counting</td>
<td>7.96</td>
<td>8.51</td>
</tr>
<tr>
<td>Bloom (no deletion)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bloom (no deletion)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cuckoo filter is among the fastest regardless.
Cuckoo filter has decreasing insert rate, but overall is only slower than blocked Bloom filter.
Summary

• Cuckoo filter, a Bloom filter replacement:
  • Deletion support
  • High performance
  • Less Space than Bloom filters in practice
  • Easy to implement

• Source code available in C++:
  • https://github.com/efficient/cuckoofilter
Othello Hashing and Its Applications for Network Processing

Chen Qian

Department of Computer Engineering
qian@ucsc.edu
https://users.soe.ucsc.edu/~qian/

• Publications in ICNP’17, SIGMETRICS’17, MECOMM’17 and Bioinformatics
Background

PhD in 2013 from UT Austin
- with Simon Lam

Research:
- Computer networking
- SDN/NFV
- Internet of things
- Security
Motivation

- Network algorithms always prefer small memory and fast processing speed.
  - Fast memory is precious resource on network devices
  - Needs to reach the line rate to avoid being a bottleneck, under large traffic volume

- More important in networks with layer-two semantics
Othello Hashing

- Essentially a **key-value lookup** structure
- Keys can be any names, addresses, identifiers, etc.
- Values should not be too long. At most 64 bit.

For example

- Key: *network address*; Value: link to forward a packet
- Key: *virtual IP*; Value: direct IP
Why Othello is special

- **Minimal query time**: only two memory read operations (cachelines) per query.
- **Minimal memory cost**: 10%-30% of existing hash tables (e.g., Cuckoo).

**Theoretical basis:** **Minimal Perfect Hashing**

- **Support dynamic updates**: can be updated over a million times per second.
Idea of dynamic Othello lookups

Controller Program

Construct Update Lookup

Update via existing API of programmable networks

Optimize memory and query cost
How Othello works

Basic version: Classifies keys to two sets $X$ and $Y$
- Equivalent to key lookups for a 1-bit value

Query result
- $\tau(k) = 0 \iff k \in X$
- $\tau(k) = 1 \iff k \in Y$

Advance version: Classifies keys to $2^l$ sets
- Equivalent to key lookups for a $l$-bit value
Othello Query Structure

Two bitmaps $a, b$ with size $m$ ($m$ in $(1.33n, 2n)$)

$n$ is # of keys

Query is easy. Then how to construct it?

$m$ bits

is in set $Y$
Othello Control Structure: Construct

$G$: acyclic bipartite graph

$h_a \xrightarrow{u_0 \ldots u_7} v_0 \ldots v_7$

$h_b \xrightarrow{u_0 \ldots u_7} v_0 \ldots v_7$

$k \quad h_a(k) \quad h_b(k)$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
Othello Construct

If finding a cycle, use another pair \(<h_a, h_b>\) until an acyclic graph is built.

For \(n\) names, the time to find \(G\) is \(O(n)\).
Compute Bitmap

\[ a \]
\[ u_0 \quad u_1 \quad u_2 \quad u_3 \quad u_4 \quad u_5 \quad u_6 \quad u_7 \]

\[ b \]
\[ v_0 \quad v_1 \quad v_2 \quad v_3 \quad v_4 \quad v_5 \quad v_6 \quad v_7 \]

<table>
<thead>
<tr>
<th>( k )</th>
<th>( h_a(k) )</th>
<th>( h_b(k) )</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>6</td>
<td>5</td>
<td>Y</td>
</tr>
<tr>
<td>( )</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Compute Bitmap

If $G$ is acyclic, easy to find a coloring plan
Name Addition – color flip

If $G$ is acyclic, flipping is trivial
L-Othello functionality

Classifies names into $2^l$ sets:

$Z_0, Z_1, \ldots, Z_{2^l-1}$

$l$ Othellos can classify names to $2^l$ sets

$l < 8$ for network devices
Example

Classify keys in 8 sets: \( Z_0, Z_1, \cdots, Z_7 \)

Orthogonal separation of sets

\[ X_3 = Z_0 \cup Z_1 \cup Z_2 \cup Z_3 ; \]
\[ Y_3 = Z_4 \cup Z_5 \cup Z_6 \cup Z_7 . \]

\[ X_2 = Z_0 \cup Z_1 \cup Z_4 \cup Z_5 ; \]
\[ Y_2 = Z_2 \cup Z_3 \cup Z_6 \cup Z_7 . \]

\[ X_1 = Z_0 \cup Z_2 \cup Z_4 \cup Z_6 ; \]
\[ Y_1 = Z_1 \cup Z_3 \cup Z_5 \cup Z_7 . \]

\[ 6=(110)_2 \quad k \in Y_3 \cap Y_2 \cap X_1 \Rightarrow k \in Z_6 \]

\[ l \quad \text{Othellos : classify keys in } 2^l \text{ sets.} \]
Same $G$, $h_a$, $h_b$.
Different coloring plan and bitmaps

Do we need $2l$ memory reads to query $l$ Othellos?

Othello 1  Same $X$ $UY$  Othello 2
\[ \tau(k) = 01 \oplus 10 = (11)_2 \]

\( k \) is in set \( Z_3 \)

CPUs can read \( l \) bits at one time
Alien keys

What is $\tau(k) = a[h_a(k)] \oplus b[h_b(k)]$ when $k$ is not in $S$?

- An arbitrary value

$\tau(k)$ return 1 with when

- $a[i] = 1 \land b[j] = 0$, or
- $a[i] = 0 \land b[j] = 1$
Applications of Othello

1. Forwarding Information Base (FIB)
2. Software load balancer
3. Data placement and lookup
4. Private queries
5. Genomic sequencing search

And more...
A Concise FIB

- Resolving FIB explosion is crucial
  - For layer-two interconnected data centers
  - For OpenFlow-like fine-grained flow control

Concise using /-Othello is a portable solution
  - In hardware devices
  - Or software switches

A Fast, Small, and Dynamic Forwarding Information Base, In ACM SIGMETRICS 2017
Network-wide updating

- If all devices share a same set of network names/addresses
  - Such as in layer-two Ethernet-based data centers
  - All Othellos will share a same $G$.
  - Hence network-wide updating is very efficient!

- Update consistency also provided
Implementation of three prototypes

1. Memory mode
   - Query and control structures running on different threads.

2. CLICK modular router

3. Intel Data Plane Development Kit (DPDK)
Comparison:

- **Buffalo**
  - Yu, Fabrikant, Rexford, *in CoNEXT’09*

- **Cuckoo hashing**
  - Zhou, Fan, Lim, Kaminsky, Andersen, *in CoNEXT’13 and SIGCOMM’15*
## Comparison: Memory size

<table>
<thead>
<tr>
<th>FIB Example</th>
<th># Names</th>
<th># Actions</th>
<th>Concise</th>
<th>Cuckoo</th>
<th>Buffalo</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC (48 bits)</td>
<td>7*10^5</td>
<td>16</td>
<td>1M</td>
<td>5.62M</td>
<td>2.64M</td>
</tr>
<tr>
<td>MAC (48 bits)</td>
<td>5*10^6</td>
<td>256</td>
<td>16M</td>
<td>40.15M</td>
<td>27.70M</td>
</tr>
<tr>
<td>MAC (48 bits)</td>
<td>3*10^7</td>
<td>256</td>
<td>128M</td>
<td>321.23M</td>
<td>166.23M</td>
</tr>
<tr>
<td>IPv4 (32 bits)</td>
<td>1*10^6</td>
<td>16</td>
<td>2M</td>
<td>4.27M</td>
<td>3.77M</td>
</tr>
<tr>
<td>IPv6 (128 bits)</td>
<td>2*10^6</td>
<td>256</td>
<td>8M</td>
<td>34.13M</td>
<td>11.08M</td>
</tr>
<tr>
<td>OpenFlow (356 bits)</td>
<td>3*10^5</td>
<td>256</td>
<td>1M</td>
<td>14.46M</td>
<td>1.67M</td>
</tr>
<tr>
<td>OpenFlow (356 bits)</td>
<td>1.4*10^6</td>
<td>65536</td>
<td>8M</td>
<td>67.46M</td>
<td>18.21M</td>
</tr>
<tr>
<td>File name (varied)</td>
<td>359194</td>
<td>16</td>
<td>512K</td>
<td>19.32M</td>
<td>1.35M</td>
</tr>
</tbody>
</table>
Query speed

2x to 4x speed advantage
Update speed

![Graph showing update speed against the number of names before update.]
For unknown network names

1. For data centers with most internal traffic
   - Such situation is rare

2. For networks with much incoming traffic
   - A filter can be installed at a firewall

3. Concise may include an $r$-bit checksum.
   - A lookup still requires 2 memory accesses in total, as long as $l + r \leq 64$. 
Thank You

Chen Qian

cqian12@ucsc.edu

https://users.soe.ucsc.edu/~qian/