Near Space-Optimal Perfect Hashing Algorithm

Nivio Ziviani
Fabiano C. Botelho
Department of Computer Science
Federal University of Minas Gerais, Brazil

International Conference on the Analysis of Algorithms
Maresias, Brazil, April 17, 2008
Objective of the Presentation

Present a perfect hashing algorithm that uses the idea of partitioning the input key set into small buckets:

- Key set fits in the internal memory
  - Internal Random Access memory algorithm (IRA)
- Key set larger than the internal memory
  - External Cache-Aware memory algorithm (ECA)
Objective of the Presentation

Present a perfect hashing algorithm that uses the idea of partitioning the input key set into small buckets:

- Key set fits in the internal memory
  - Internal Random Access memory algorithm (IRA)
- Key set larger than the internal memory
  - External Cache-Aware memory algorithm (ECA)

Theoretically well-founded, time efficient, highly scalable and near space-optimal
Perfect Hash Function

Static key set $S$ of size $n$

Hash Table

$S \subseteq U$, where $|U| = u$
Minimal Perfect Hash Function

Static key set $S$ of size $n$

$$S \subseteq U, \text{ where } |U| = u$$
Where to use a PHF or a MPHF?

- Access items based on the value of a key is ubiquitous in Computer Science
- Work with huge static item sets:
  - In data warehousing applications:
    - On-Line Analytical Processing (OLAP) applications
  - In Web search engines:
    - Large vocabularies
    - Map long URLs in smaller integer numbers that are used as IDs
Mapping URLs to Web Graph Vertices

URLS

URL 1
URL 2
URL 3
URL 4
URL 5
URL 6
URL 7
...
URL n

Web Graph Vertices

0
1
2
3
4
5
6
.. 
n-1
Mapping URLs to Web Graph Vertices
Information Theoretical Lower Bounds for Storage Space

- PHFs \( (m \approx n) \): Storage Space \( \geq \frac{n^2}{m} \log e \)

- MPHFs \( (m = n) \): Storage Space \( \geq n \log e \)

\[ m < 3n \]
\[ \log e \approx 1.4427 \]
Uniform Hashing Versus Universal Hashing

Key universe $U$ of size $u$ \rightarrow \text{Hash function} \rightarrow \text{Range } M \text{ of size } m
Uniform Hashing Versus Universal Hashing

Uniform hashing

- # of functions from U to M?
  \[ m^u \]

- # of bits to encode each function
  \[ u \log m \]

- Independent functions with values uniformly distributed
Uniform Hashing Versus Universal Hashing

**Key universe**
- U of size u

**Hash function**
- $u \log m$

**Range M of size m**

**Uniform hashing**
- # of functions from U to M?
  - $m^u$
- # of bits to encode each function
  - $u \log m$
- Independent functions with values uniformly distributed

**Universal hashing**
- A family of hash functions $\mathcal{H}$ is universal if:
  - for any pair of distinct keys $(x_1, x_2)$ from U and
  - a hash function $h$ chosen uniformly from $\mathcal{H}$ then:
    $$\Pr(h(x_1) = h(x_2)) \leq \frac{1}{m}$$
Intuition Behind Universal Hashing

- We often lose relatively little compared to using a completely random map (uniform hashing)

- If S of size n is hashed to $n^2$ buckets, with probability more than $\frac{1}{2}$, no collisions occur
  
  - Even with complete randomness, we do not expect little $o(n^2)$ buckets to suffice (the birthday paradox)
  
  - So nothing is lost by using a universal family instead!
Related Work

- Theoretical Results
  (use uniform hashing)

- Practical Results
  (assume uniform hashing for free)

- Heuristics
## Theoretical Results

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schmidt and Siegel (1990)</td>
<td>Not analyzed</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Hagerup and Tholey (2001)</td>
<td>$O(n+\log \log u)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
### Theoretical Results – Use Uniform Hashing

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schmidt &amp; Siegel (1990)</td>
<td>Not analyzed</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Hagerup &amp; Tholey (2001)</td>
<td>$O(n+\log \log u)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Theoretic ECA (CIKM 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
### Practical Results – Assume Uniform Hashing For Free

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Pagh (1999)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Botelho, Kohayakawa, &amp; Ziviani (2005)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
Practical Results – Assume Uniform Hashing For Free

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Pagh (1999)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Botelho, Kohayakawa, &amp; Ziviani (2005)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>IRA (WADS 2007)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
# Practical Results – Assume Uniform Hashing For Free

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Pagh (1999)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Botelho, Kohayakawa, &amp; Ziviani (2005)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>IRA (WADS 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Heuristic ECA (CIKM 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
## Empirical Results

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox, Chen &amp; Heath (1992)</td>
<td>Index data in CD-ROM</td>
<td>Exp.</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Lefebvre &amp; Hoppe (2006)</td>
<td>Sparse spatial data</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Chang, Lin &amp; Chou (2005, 2006)</td>
<td>Data mining</td>
<td>O(n)</td>
<td>O(1)</td>
<td>Not analyzed</td>
</tr>
</tbody>
</table>
Internal Random Access and External Cache-Aware Memory Algorithms

- Near space optimal
- Evaluation in constant time
- Function generation in linear time
- Simple to describe and implement
- Known algorithms with near-optimal space either:
  - Require exponential time for construction and evaluation, or
  - Use near-optimal space only asymptotically, for large $n$
- Acyclic random hypergraphs
  - Used before by Majewski et all (1996): $O(n \log n)$ bits
- We proceed differently: $O(n)$ bits
  (we changed space complexity, close to theoretical lower bound)
Random Hypergraphs (r-graphs)

- 3-graph:

  0  1  2  3  4  5

- 3-graph is induced by three uniform hash functions
Random Hypergraphs (r-graphs)

- 3-graph:

  \[ h_0(jan) = 1 \quad h_1(jan) = 3 \quad h_2(jan) = 5 \]

- 3-graph is induced by three uniform hash functions
Random Hypergraphs (r-graphs)

- 3-graph:

  \[ \begin{align*}
  h_0(\text{jan}) &= 1 & h_1(\text{jan}) &= 3 & h_2(\text{jan}) &= 5 \\
  h_0(\text{feb}) &= 1 & h_1(\text{feb}) &= 2 & h_2(\text{feb}) &= 5
  \end{align*} \]

- 3-graph is induced by three uniform hash functions
Random Hypergraphs (r-graphs)

- 3-graph:

  \[ h_0(\text{jan}) = 1 \quad h_1(\text{jan}) = 3 \quad h_2(\text{jan}) = 5 \]
  \[ h_0(\text{feb}) = 1 \quad h_1(\text{feb}) = 2 \quad h_2(\text{feb}) = 5 \]
  \[ h_0(\text{mar}) = 0 \quad h_1(\text{mar}) = 3 \quad h_2(\text{mar}) = 4 \]

- 3-graph is induced by three uniform hash functions
- Our best result uses 3-graphs
The Internal Random Access memory algorithm ...
Acyclic 2-graph

\[ G_r: \]

\[
\begin{array}{c c c c}
0 & 1 & 2 & 3 \\
\text{mar} & \text{jan} & \text{feb} & \text{apr}
\end{array}
\]

\[
\begin{array}{c c c c}
4 & 5 & 6 & 7 \\
& h_0 & & h_1
\end{array}
\]

L:Ø
Acyclic 2-graph

\[ G_r: \]

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
\end{array}
\]

\[
\text{L: } \{0,5\}
\]
Acyclic 2-graph

\[ G_r : \]

\[ \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\downarrow \text{jan} & \downarrow \text{apr} & \downarrow \\
4 & 5 & 6 & 7 \\
\end{array} \]

\[ h_0 \]

\[ \begin{array}{cc}
0 & 1 \\
\end{array} \]

\[ L : \{0,5\} \{2,6\} \]

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Acyclic 2-graph
Acyclic 2-graph

$G_r$ is acyclic

$G_r$: 

$0 \quad 1 \quad 2 \quad 3 \quad h_0$

$L$: 

\begin{align*}
0 \quad & \{0, 5\} \\
1 \quad & \{2, 6\} \\
2 \quad & \{2, 7\} \\
3 \quad & \{2, 5\}
\end{align*}

$4 \quad 5 \quad 6 \quad 7 \quad h_1$
Internal Random Access Memory Algorithm (r=2)

S

jan
feb
mar
apr
Internal Random Access Memory Algorithm (r=2)
Internal Random Access Memory Algorithm (r=2)

Mapping:
- Jan: 0
- Feb: 1
- Mar: 2
- Apr: 3

Assigning:
- h₀: 4
- h₁: 5

L:
- 0: 6
- 1: 7
- 2: 8
- 3: 9

L: {0,5} {2,6} {2,7} {2,5}
Internal Random Access Memory Algorithm (r=2)

Mapping:
S
jan  feb  mar  apr

Gr:
0  1  2  3 h_0
4  5  6  7 h_1

Assigning:

L:
{0,5}  {2,6}  {2,7}  {2,5}

G_r:
0  1  2  3 h_0
4  5  6  7 h_1

S
jan  feb  mar  apr

L:
{0,5}  {2,6}  {2,7}  {2,5}

L: 0  1  2  3
g
0 r
1 r
2 0 r
3 r
4 r
5 r
6 r
7 r
Internal Random Access Memory Algorithm (r=2)

Mapping

Assigning

$G_r$: 0, 1, 2, 3

$L$: {0,5} {2,6} {2,7} {2,5}

$g$

0: r
1: r
2: 0
3: r
4: r
5: r
6: r
7: 1
Internal Random Access Memory Algorithm (r=2)

S
jan
feb
mar
apr

Mapping

0 1 2 3
G_r:

assigned
assigned

0 1 2 3
h_0

4 5 6 7
assigned
assigned

Assigning

0 1
0

g

L

0 1 2 3 4 5 6 7
r
r
r
1
1
Internal Random Access Memory Algorithm (r=2)

\[ i = (g[h_0(feb)] + g[h_1(feb)]) \mod r = (g[2] + g[6]) \mod 2 = 1 \]
Internal Random Access Memory Algorithm: PHF

\[ i = (g[h_0(feb)] + g[h_1(feb)]) \mod r = (g[2] + g[6]) \mod 2 = 1 \]

\[ phf(feb) = h_{i=1}(feb) = 6 \]
Internal Random Access Memory Algorithm: MPHF

\[
i = (g[h_0(feb)] + g[h_1(feb)]) \mod r = (g[2] + g[6]) \mod 2 = 1
\]
\[
\text{phf(feb)} = h_{i=1}(feb) = 6
\]
\[
\text{mphf(feb)} = \text{rank(phf(feb))} = \text{rank}(6) = 2
\]
Space to Represent the Function

Mapping

Assigning

2 bits for each entry

\[ G_r : \]

\[ h_0 \]

\[ h_1 \]

\[ g \]

\[ \begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
7 \\
\end{array} \]

\[ \begin{array}{c}
0 \\
r \\
0 \\
r \\
r \\
p \\
1 \\
1 \\
\end{array} \]
Space to Represent the Functions ($r = 3$)

- **PHF $g$:** $[0,m-1] \rightarrow \{0,1,2\}$
  - $m = cn$ bits, $c = 1.23 \rightarrow 2.46n$ bits
  - $(\log 3) cn$ bits, $c = 1.23 \rightarrow 1.95n$ bits (arith. coding)
  - Optimal: $0.89n$ bits

- **MPHF $g$:** $[0,m-1] \rightarrow \{0,1,2,3\}$ (ranking info required)
  - $2m + \varepsilon m = (2 + \varepsilon)cn$ bits
  - For $c = 1.23$ and $\varepsilon = 0.125 \rightarrow 2.62n$ bits
  - Optimal: $1.44n$ bits.
Use of Acyclic Random Hypergraphs

- Sufficient condition to work
- Repeatedly selects $h_0, h_1, \ldots, h_{r-1}$
- For $r = 2$, $m = cn$ and $c \geq 2.09$: $P_{ra} = 0.29$
- For $r = 3$, $m = cn$ and $c \geq 1.23$: $P_{ra}$ tends to 1
- Number of iterations is $1/P_{ra}$
  - $r = 2$: 3.5 iterations
  - $r = 3$: 1.0 iteration
The External Cache-Aware memory algorithm ...
External Cache-Aware Memory Algorithm (ECA)

- First MPHF algorithm for very large key sets (in the order of billions of keys)

- This is possible because
  - Deals with external memory efficiently
  - Generates compact functions (near space-optimal)
  - Uses a little amount of internal memory to scale
  - Works in linear time

- Two implementations:
  - Theoretical well-founded ECA (uses uniform hashing)
  - Heuristic ECA (uses universal hashing)
External Cache-Aware Memory Algorithm (ECA)

MPHF(x) = MPHF_i(x) + offset[i];
Key Set Does Not Fit In Internal Memory

Partitioning

Key Set S of $\beta$ bytes

File 1

$N = \beta / \mu$

$0 \leq 2^b - 1$

File N

Each bucket $\leq 256$

$\mu$ bytes of Internal memory

$\mu$ bytes of Internal memory

$h_0$

$h_0$

0 1 2

$2^b - 1$

$\ldots$

$\ldots$

$\ldots$

$\ldots$

0 1 2

$\ldots$

$\ldots$

$\ldots$

$\ldots$
Important Design Decisions

- We map long URLs to a fingerprint of fixed size using a hash function
- Use our IRA linear time and near space-optimal algorithm to generate the MPHF of each bucket
- How do we obtain a linear time complexity?
  - Using internal radix sorting to form the buckets
  - Using a heap of N entries to drive a N-way merge that reads the buckets from disk in one pass
Use the Internal Random Access Memory Algorithm for Each Bucket

$S$

Mapping

Assigning

$G_r$: assigned

$h_0$

$h_1$

Assigned

Hash Table

$g$

Ranking

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

0 1 2 3

0 1 2 3

Jan Feb Mar Apr

Mar Jan Feb Apr
Why the ECA Algorithm is Well-Founded?

First Point:

Pool of uniform hash functions on each bucket ≤ 256

Sharing

h_{i0} h_{i1} h_{i2} h_{i0} h_{i1} h_{i2}
Why the ECA Algorithm is Well-Founded?

Second Point:
We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

\[
f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j [y_j(x) \oplus \Delta] + s \sum_{j=k+1}^{2k} t_j [y_{j-k}(x) \oplus \Delta] \right) \mod p
\]

\[
h_{i0}(x) = f(x, s, 0) \mod |B_i|
\]
\[
h_{i1}(x) = f(x, s, 1) \mod |B_i| + |B_i|
\]
\[
h_{i2}(x) = f(x, s, 2) \mod |B_i| + 2|B_i|
\]
Why the ECA Algorithm is Well-Founded?

Second Point:
We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

\[ f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j \left[ y_j(x) \oplus \Delta \right] + s \sum_{j=k+1}^{2k} t_j \left[ y_{j-k}(x) \oplus \Delta \right] \right) \mod p \]

\[ h_{i0}(x) = f(x, s, 0) \mod |B_i| \]
\[ h_{i1}(x) = f(x, s, 1) \mod |B_i| + |B_i| \]
\[ h_{i2}(x) = f(x, s, 2) \mod |B_i| + 2|B_i| \]
Why the ECA Algorithm is Well-Founded?

Second Point:
We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

\[ f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j [y_j(x) \oplus \Delta] + s \sum_{j=k+1}^{2k} t_j [y_{j-k}(x) \oplus \Delta] \right) \mod p \]

- \( h_{i0}(x) = f(x, s, 0) \mod |B_i| \)
- \( h_{i1}(x) = f(x, s, 1) \mod |B_i| + |B_i| \)
- \( h_{i2}(x) = f(x, s, 2) \mod |B_i| + 2|B_i| \)
Why the ECA Algorithm is Well-Founded?

Second Point:

We have shown how to create that pool based on the linear hash functions proposed by Alon et al (JACM 1999)

$$f(x, s, \Delta) = \left( \sum_{j=1}^{k} t_j [y_j(x) \oplus \Delta] + s \sum_{j=k+1}^{2k} t_j [y_{j-k}(x) \oplus \Delta] \right) \mod p$$

$$h_{i0}(x) = f(x, s, 0) \mod |B_i|$$

$$h_{i1}(x) = f(x, s, 1) \mod |B_i| + |B_i|$$

$$h_{i2}(x) = f(x, s, 2) \mod |B_i| + 2|B_i|$$
Why the ECA Algorithm is Well-Founded?

Second Point:

Computing fingerprints of 128 bits with the linear hash functions

\[ h'(x) = 100101011110011011010000111000110 \]
\[ 1101110101001000 \]
\[ 0011000111000110 \]
\[ 0000000111010110 \]
\[ 001111111000110 \]
\[ 111111111000110 \]
\[ 0000000000000110 \]

\[ h_0(x) = h'(x)[96,127] >> (32 - b) \]
\[ y_6(x) = h'(x)[80,95] \]
\[ y_1(x) = h'(x)[0,15] \]
Why the ECA Algorithm is Well-Founded?

Third Point:
How to keep maximum bucket size smaller than $l = 256$?

\[
b \leq \log(n) - \log\left(\frac{l}{\log l}\right) + O(1)
\]
\[
l \geq \log n \log \log \log n
\]
The Heuristic ECA Algorithm

- Uses a universal pseudo random hash function proposed by Jenkins (1997):
  - Faster to compute
  - Requires just one random integer number as seed
Experimental Results

- **Metrics:**
  - Generation time
  - Storage space
  - Evaluation time

- **Collection:**
  - 1.024 billions of URLs collected from the web
  - 64 bytes long on average

- **Experiments**
  - Commodity PC with a cache of 4 Mbytes
  - 1.86 GHz, 1 GB, Linux, 64 bits architecture
## Generation Time of MPHFs (in Minutes)

<table>
<thead>
<tr>
<th>n (millions )</th>
<th>32</th>
<th>128</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretic ECA</td>
<td>1.3 ± 0.002</td>
<td>6.2 ± 0.02</td>
<td>27.6 ± 0.09</td>
<td>57.4 ± 0.06</td>
</tr>
<tr>
<td>Heuristic ECA</td>
<td>0.95 ± 0.02</td>
<td>5.1 ± 0.01</td>
<td>22.0 ± 0.13</td>
<td>46.2 ± 0.06</td>
</tr>
</tbody>
</table>
Related Algorithms

- Botelho, Kohayakawa, Ziviani (2005) - BKZ
- Fox, Chen and Heath (1992) – FCH
- Czech, Havas and Majewski (1992) – CHM
- Pagh (1999) - PAGH

All algorithms coded in the same framework
# Generation Time

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRA (r = 3)</td>
<td>6.7 ± 0</td>
</tr>
<tr>
<td>Theoretic ECA</td>
<td>9.0 ± 0.3</td>
</tr>
<tr>
<td>Heuristic ECA</td>
<td>6.4 ± 0.3</td>
</tr>
<tr>
<td>BKZ</td>
<td>12.8 ± 1.6</td>
</tr>
<tr>
<td>CHM</td>
<td>17.0 ± 3.2</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
</tr>
<tr>
<td>PAGH</td>
<td>42.8 ± 2.4</td>
</tr>
</tbody>
</table>

3,541,615 URLs
## Generation Time and Storage Space

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
<th>Space (bits/key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRA (r = 3)</td>
<td>6.7 ± 0</td>
<td>2.6</td>
</tr>
<tr>
<td>Theoretic ECA</td>
<td>9.0 ± 0.3</td>
<td>3.3</td>
</tr>
<tr>
<td>Heuristic ECA</td>
<td>6.4 ± 0.3</td>
<td>3.1</td>
</tr>
<tr>
<td>BKZ</td>
<td>12.8 ± 1.6</td>
<td>21.8</td>
</tr>
<tr>
<td>CHM</td>
<td>17.0 ± 3.2</td>
<td>45.5</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
<td>4.2</td>
</tr>
<tr>
<td>PAGH</td>
<td>42.8 ± 2.4</td>
<td>44.2</td>
</tr>
</tbody>
</table>

3,541,615 URLs
# Generation Time, Storage Space and Evaluation Time

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
<th>Space (bits/key)</th>
<th>Evaluation time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRA (r = 3)</td>
<td>6.7 ± 0</td>
<td>2.6</td>
<td>2.1</td>
</tr>
<tr>
<td>Theoretic ECA</td>
<td>9.0 ± 0.3</td>
<td>3.3</td>
<td>4.9</td>
</tr>
<tr>
<td>Heuristic ECA</td>
<td>6.4 ± 0.3</td>
<td>3.1</td>
<td>2.7</td>
</tr>
<tr>
<td>BKZ</td>
<td>12.8 ± 1.6</td>
<td>21.8</td>
<td>2.3</td>
</tr>
<tr>
<td>CHM</td>
<td>17.0 ± 3.2</td>
<td>45.5</td>
<td>2.3</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
<td>4.2</td>
<td>1.7</td>
</tr>
<tr>
<td>PAGH</td>
<td>42.8 ± 2.4</td>
<td>44.2</td>
<td>2.3</td>
</tr>
</tbody>
</table>

3,541,615 URLs
Key length = 64 bytes
Minimal Perfect Hashing Library

Why to build a library?
- Lack of similar libraries in the free software community
- Test the applicability of our algorithm out there

Feedbacks:
- 1,883 downloads (until Apr 15th, 2008)
- Incorporated by Debian

Library address: http://cmph.sourceforge.net
Conclusions

- Three implementations were developed:
  - Theoretic ECA (external memory)
  - Heuristic ECA (external memory)
  - IRA (internal memory, used in ECA algorithm)

- Near space-optimal functions in linear time

- Function evaluation in time $O(1)$

- First theoretically well-founded algorithm that is practical and will work for every key set from $U$ with high probability
Parallel Version of the ECA Algorithm

<table>
<thead>
<tr>
<th>PCs</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.8</td>
<td>3.5</td>
<td>7.0</td>
<td>8.7</td>
<td>12.2</td>
</tr>
</tbody>
</table>

1 billion URLs using 14 PCs in 5 minutes