Exact Match Lookup

Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst
Exact Match Lookup

- Often, single header field can serve as lookup key
  - Ethernet addresses, IP multicast addresses
  - Virtual Circuit Identifier in ATM (also, X.25, frame relay)
  - VLAN and Multiprotocol Label Switching (MPLS) tags

- Direct lookup is simplest
  - can be impractical for large keys
  - must be extended for multiple fields
    - e.g. Ethernet VLAN+destination address

- Hash tables are usually better option
  - constant time expected performance
  - extends naturally to exact match on many fields
  - must engineer for “wire-speed” processing
Basic Hash Table with Chaining

- Packet fields combined to form hash index
  - ideal hash function maps items uniformly across buckets
- Largest bucket determines worst-case performance
  - memory access per list item
  - given $n$ items distributed randomly across $b$ buckets
  - $E(\# \text{ buckets with } \geq k \text{ items})$
  $$E(\# \text{ buckets with } \geq k \text{ items}) = b \sum_{i=k}^{n} \binom{n}{i} \left(\frac{1}{b}\right)^i \left(1 - \frac{1}{b}\right)^{n-i}$$

![Graph showing the distribution of buckets with different sizes.](image)
**d-Left Hashing**

- Separate table of entries from hash table
- Divide hash table into $d$ sections of $b$ buckets each
- Each bucket can hold $k$ “items”, each item includes a pointer to an entry and a fingerprint
- To search first section
  - hash packet fields to get index and fingerprint
  - read bucket to find item with matching fingerprint
  - follow pointer to retrieve table entry
- If not in first section, repeat using different hash
Insertion in $d$-Left Hash Tables

- Compute separate hash on entry for each section
- Retrieve bucket from each section
- Insert entry into least-loaded bucket
  » break ties "to the left"
- If no room in any bucket
  » reject insertion, or
  » insert into small overflow table
  » or, implement each bucket as a list with $k$ items per list node
Advantages of $d$-Left Hashing

- Separation of data and use of fingerprints allows checking of several locations per memory access
  - because for off-chip memory, cost of reading a "large" block is only slightly larger than reading a small block
  - for modern processors entire bucket can fit in one cache line
  - can apply this to basic hash table also
- Multiple hash functions significantly reduces collision probability
- Number of memory accesses rarely exceeds $d+1$
  - exception: false match of fingerprint
  - in practice, $d=2$ typically works best
- Drawback is insertion can fail if all buckets full
Theoretical Basis

- Distributing $n$ balls randomly in $n$ bins
  - bin with most balls has $\approx 1 + \ln n / \ln \ln n$ with high probability (5.7 for $n=100K$)

- Placing each ball in least full of $d$ random bins, breaking ties randomly
  - bin with most balls has $\approx 1 + (\ln \ln n) / \ln d$ with high probability (4.6 for $d=2$, $n=100K$)

- Dividing bins into two groups, placing each ball in least full bin; break ties “to left”
  - bin with most balls has $\approx 1 + 1.05 \ln \ln n$ with high probability (3.6 for $n=100K$)
  - with $n$ items in $n/h$ bins, bin with most balls has $\approx h + 1.05 \ln \ln (n/h)$ w.h.p (12.3 for $n=100K$, $h=10$)
Simulation of \( d \)-Left

- Random Insertion
  - \( n \) table entries
  - \( d \) sections
  - \( b \) buckets per section
- Avg of 10 items/bkt
- Max items/bkt
  - 25 for \( d=1 \), 12 for \( d=2 \),
    11 for \( d=4 \)
- Worst-case memory accesses
  - 2 for \( d=1 \), 3 for \( d=2 \), 5 for \( d=4 \) (note: 25,12,11 items/bkt)
- Memory efficiency
  - 40% for \( d=1 \), 83% for \( d=2 \), 91% for \( d=4 \)
Other Simulation Cases

- For $db=n$
  - memory efficiency: 14%, 33%, 50% for $d=1, 2, 4$
- For $db=n/4$
  - memory efficiency: 25%, 57%, 67%
Bloom Filters

- Approximate set membership data structure
  - small false-positive probability
- Compact representation – can often fit “on-chip”
  - can be used to avoid useless off-chip memory accesses
  - e.g. could use BF for each section of d-left hash table
- Uses bit vector and k hash functions
  - to insert item, compute k hashes and set all selected bits
  - to check for membership, compute hash and check bits
- Given n items, target false positive probability f
  - optimal number of hash functions, \( k \approx \lg \frac{1}{f} \)
  - and required bit vector length, \( m \approx 1.44 kn \)
  - so, for \( f = 1/2^{10} \), use \( k = 10, m = 14.4n \)