Some Uses of Hashing in Networking Problems

Michael Mitzenmacher

Two Applications

- Improved Analysis of the Lossy Difference Aggregator
  - With H. Finucane (undergrad!) (Computer Communications Review)
- Carousel: Scalable Logging for Intrusion Prevention Systems
  - With T. Lam and G. Varghese (NSDI 2010)
- Key, simple idea : Partition data into buckets by hashing, then analyze.

Lossy Difference Aggregator

- Motivation : sampling for latency estimates.
  - Packets travel from A to B : average latency.
  - Must cope with occasional packet loss.
  - Cannot add per packet timestamps (too much space, packet headers already designed).
- Applications
  - Interactive multimedia (games, videoconferencing)
  - Trading platforms
  - High-performance systems/data centers
- Proposed in [KLSV, SIGCOMM 2009]

Zero Loss Case

- Assumptions: over a time window, consistent clocks at endpoints
- Sender sums timestamps for packets sent; receiver sums timestamps for packets received.
  - Recall timestamps NOT sent.
- Sender sends control packet with sum.
- Receiver takes difference, divides by number of packets to get average.
- Fails entirely once loss introduced.

Dealing with Loss : Hashing

- Hash packets into logical buckets.
  - Disjoint; not all packets necessarily hashed.
- Need time accumulator + counter for each bucket.
- Sender sums timestamps and counts packets for each bucket; same for receiver.
- Sender sends control packet with sum/count for each bucket.
- Each bucket with no loss gives a useful sample for measuring latency.

Example

Sender		Receiver		
Sum	Ctr	Sum	Ctr	Difference
120	5	180	5	60
234	10	348	9	~~114~~ ~~1~~
15	2	37	2	22
88	1	96	1	8

Estimate = (60 + 22 + 8)/(5 + 2 + 1)

Analysis : Questions

- Optimal sampling rate per bucket given known loss rate.
  - Natural restriction : what if sampling rates are of form 2^{-j} (hash on last j bits).
    - Restricted sampling rates implies simpler hardware.
- What do when loss rates are unknown.
  - Extended analysis based on competitive analysis.
  - Single sampling rate vs. multiple sampling rate optimization and design.
    - Fewer sampling rates implies simpler hardware.

Simple Analysis

- Let *n* be total number of packets, *l* be loss rate, *z*/*n* be sampling rate, and *Z* be packets obtained by bucket.

$$\Pr(Z = i) = \binom{n}{i}\left(\frac{z}{n}\right)^{i}\left(1 - \frac{z}{n}\right)^{n-i}(1-l)^{i}$$

$$E[Z] = z(1-l)(1-lz/n)^{n-1} \rightarrow z(1-l)e^{-lz}$$

- Expectation reaches maximum when $z = 1/l$, number of packets is $(1-l)/(el)$.

Restrictions : Powers of 2

- Suppose instead of choosing sample rate z/n we choose y/n, with expectations Z and Y.

$$E[Y]/E[Z] \to (y/z)e^{l(z-y)}$$

- When $z = 1/l$, $y=xz$ or $y=2xz$ chosen as nearest inverse power of 2, then ratio is

$$\min_{1/2<x\le 1} \max(xe^{1-x}, 2xe^{1-2x}) \ge 0.942$$

- Loss of less than 6%; worst case when $x = \ln 2$.

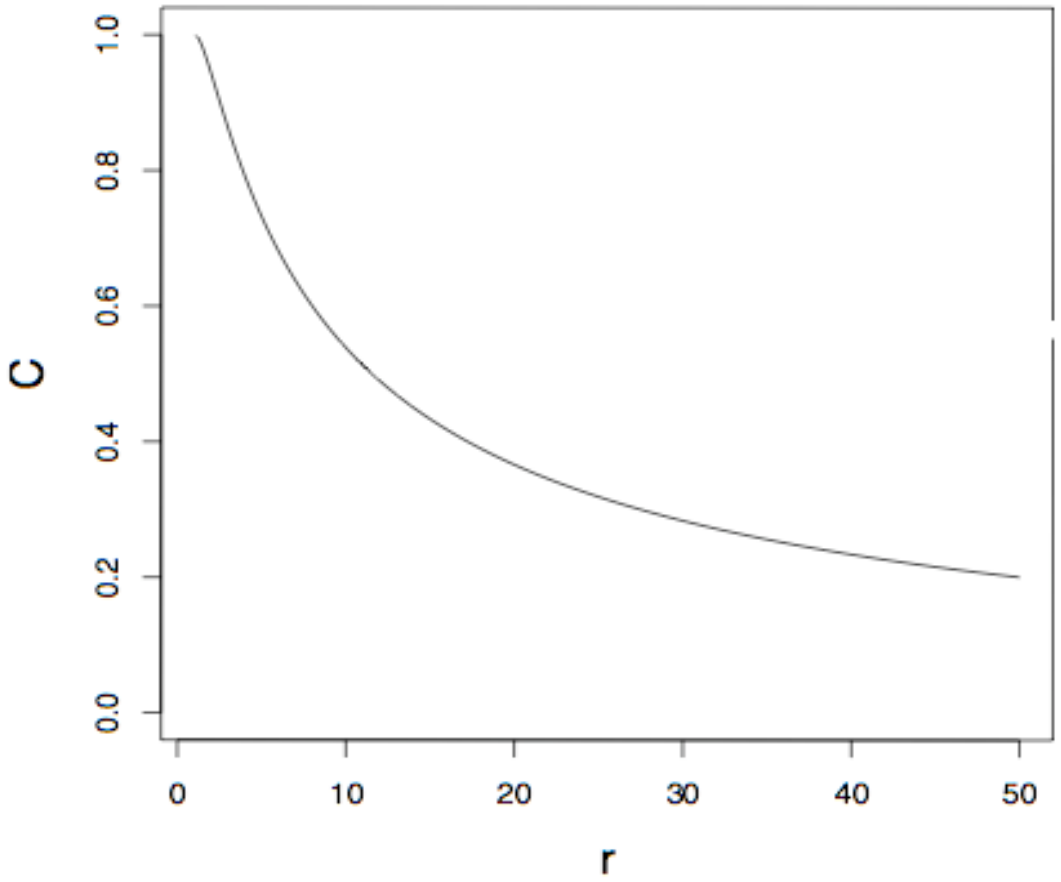
Unknown Loss Rates

- What to optimize for unknown loss rate?
  - Lots of possibilities.
- Our suggestion : optimize competitive ratio, over a range of possible loss rates, between expected number of packets obtained and optimal expected number of packets obtained if loss rate is known.
- Gives a rigorous framework, can be extended to other possibilities.

Single Sampling Probability

- Using previous analysis, can show:
  - When range is [*a*,*b*], and $r = b/a$, best competitive ratio is: $\dfrac{e \ln r}{(r-1) r^{1/(r-1)}}$
  - When $r \leq 2$, a *single sampling probability* is sufficient for best competitive ratio.

Competitive Ratio



Design : Multiple Sampling Probabilities

- Must deal with loss rates over several orders of magnitude.
- Choose sampling probabilities to cover geometrically spaced ranges.
  - For large ranges $[a,b]$ with ratio $r = b/a$, and c sampling probabilities, split into c subranges with ratio $r^{1/c}$.
    - Competitive ratio at worst reduced by factor c.
  - Can do better with more analysis, but gives a good initial rule of thumb.

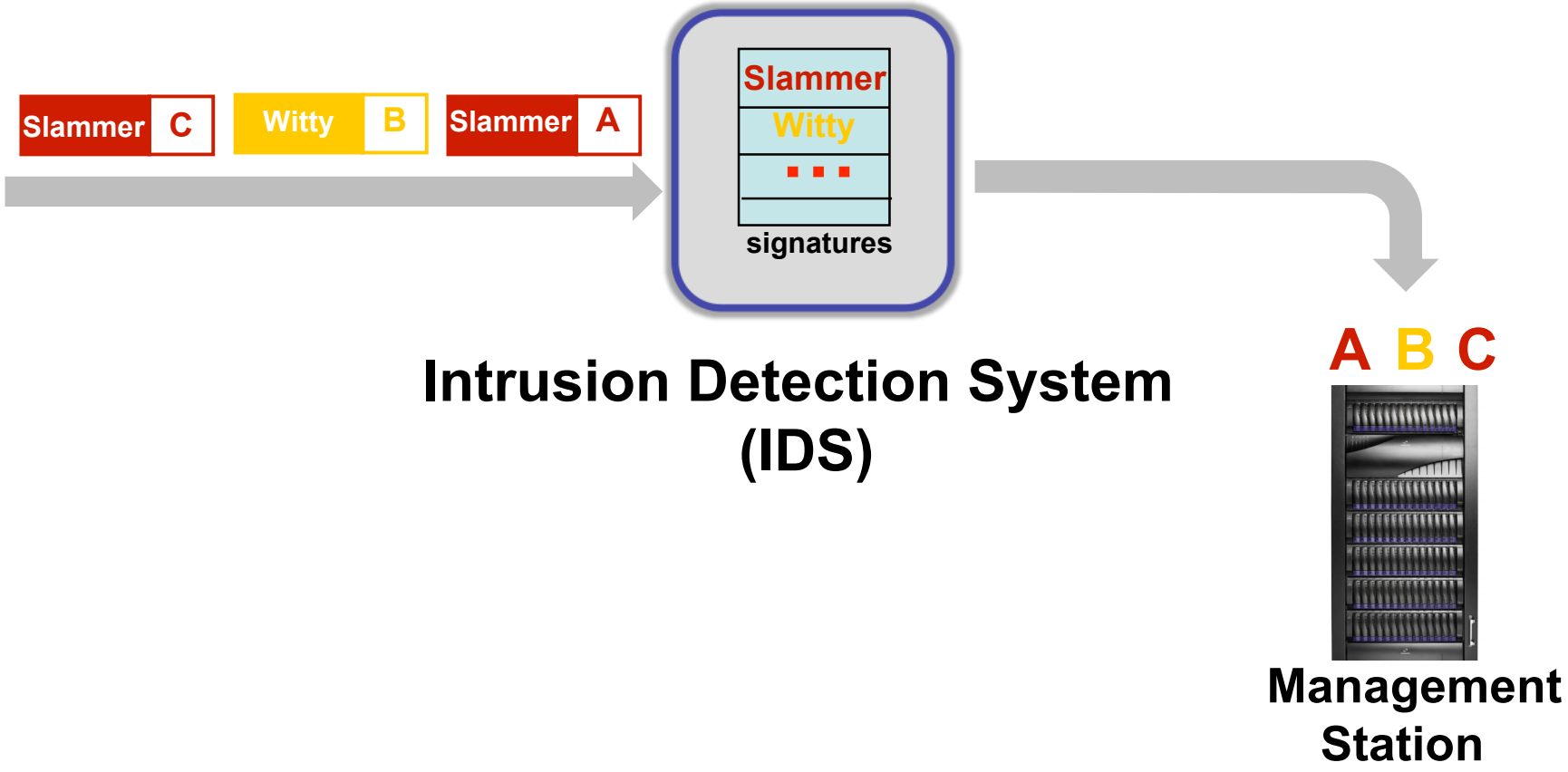
Takeaways

- Can estimate delay across sender-receiver via bucketing
  - Example of “Coordinated streaming”
  - Other applications?
  - More general functions?
    - Can be extended to find sample variance using standard techniques.
- Analysis focuses on practical issues, design rules
  - Competitive analysis for *parameter setting*.

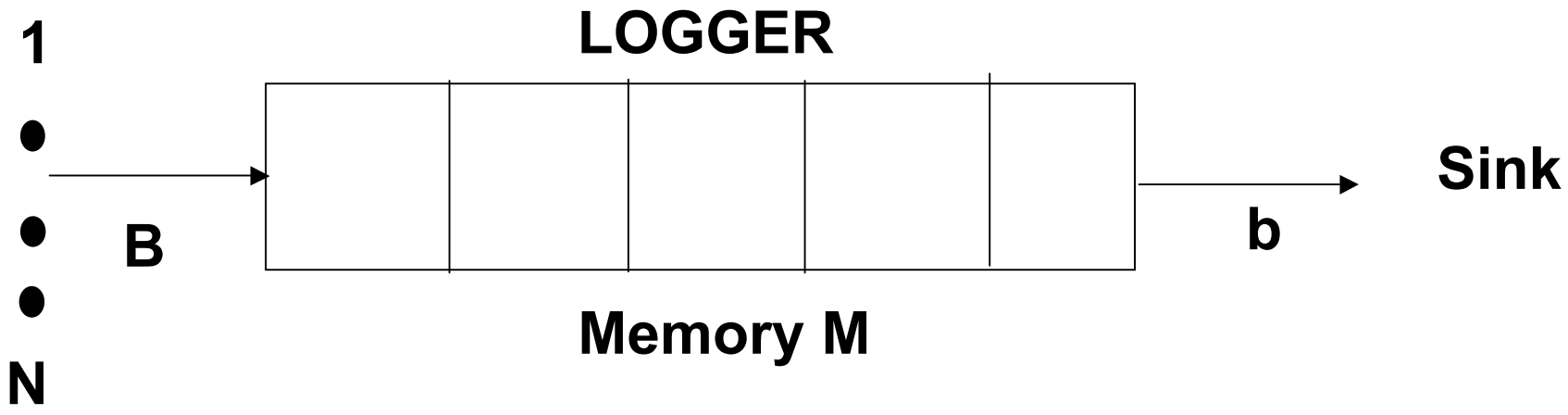
Carousel : Scalable Logging

- Millions of potentially *interesting* events
  - Standard solutions: sampling and summarizing
- What if you want complete collection
  - Remediate infected machines
  - Other examples: Listing IPv6 addresses, MAC addresses in a LAN

Example : Worm Outbreak



Abstract Model



- Challenges
  - Small logging bandwidth b << arrival rate B
    - e.g., b = 1 Mbps; B = 10 Gbps
  - Small memory M << number of sources N
    - e.g., M = 10,000; N=1 Million
- Assumption : persistent sources, keep arriving at the logger.

Naïve Approach

- Just log things into memory as they arrive.
- Problem : repeats
  - Memory too small to track history.
  - Consider random model -- each source is random on [1,N].
  - Repeatedly send same stuff to sink.

Naïve Approach + Bloom Filter

- Bloom filter keeps track of a set, with some false positives.
- Can use to track set of sent items.
- But memory too small to track all sent items, so Bloom filter must reset periodically.
- Same problem -- repeated sends of same info to sink.

Carousel Solution

- Use hashing to bucketize sources.
  - Want 1 bucket to fit into memory (approximately).
- Let $T = M/b$, time to clear memory, be 1 phase.
  - Iterate over buckets, one per phase.
  - Use Bloom filter within phase to prevent duplicates entering memory/being sent.
- Increase or decrease #of buckets as needed to avoid memory overflow.

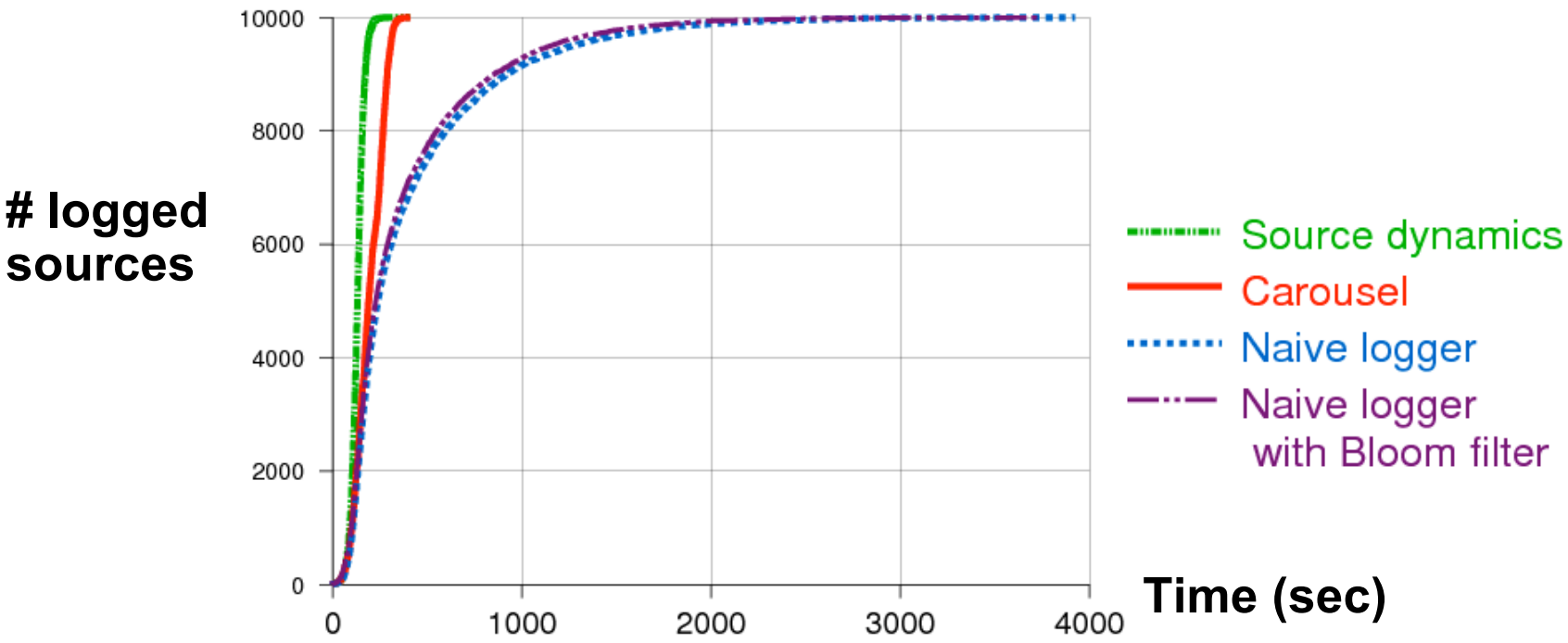
Theoretical Results

- Carousel is “competitive” in that it can collect almost all sources within a factor of (1+epsilon) of optimal with the right number of buckets.
  - Simple application of Chernoff bounds -- you get almost the right number per bucket for all buckets.
  - In practice -- use last k bits of a hash for 2^k buckets, within a factor of (2+epsilon) of optimal.

Theoretical Results

- Why almost all, and not all?
- Need a Bloom filter to prevent duplicates from overwhelming memory.
  - But gives false positives.
  - Some items not recorded.
  - In practice: switch hash functions each round, number missed shrinks exponentially with rounds.

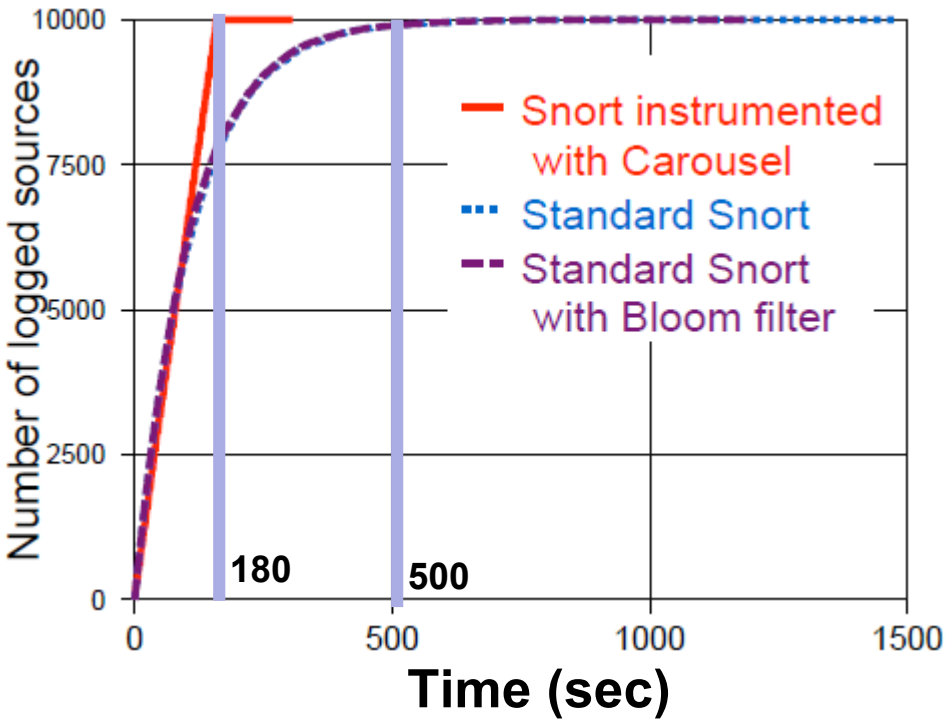
Simulated Worm Outbreak



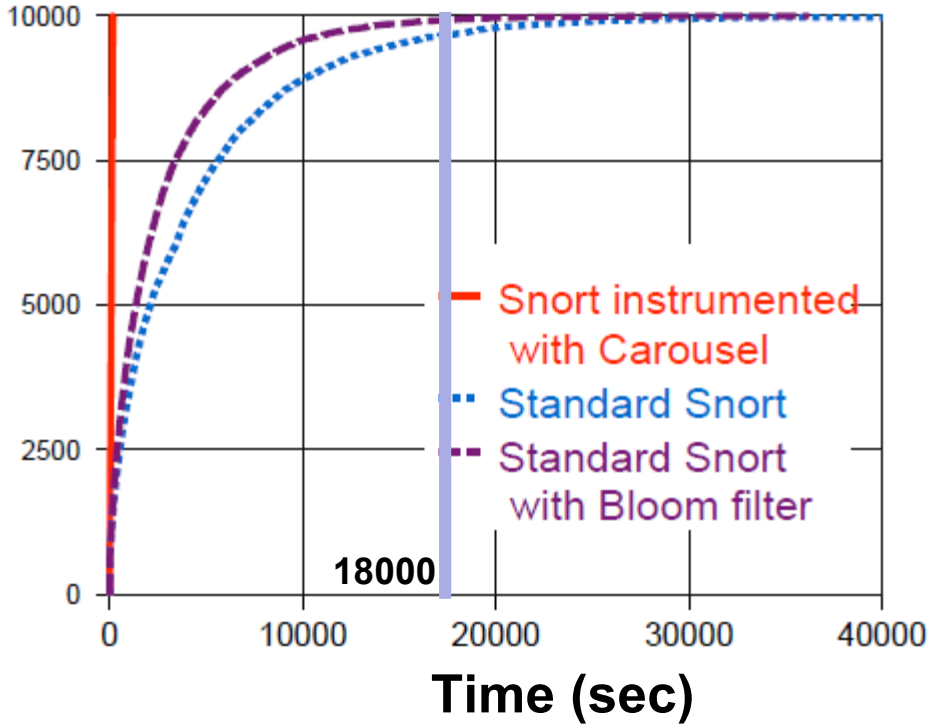
Carousel is nearly ten times faster than naïve collector

- N = 10,000; M = 500; b = 100 items/sec
- Logistic model of worm growth

Snort Results



(a) Random traffic pattern



(b) Periodic traffic pattern

3 times faster with random and 100 times faster with periodic

Hardware Requirements

- Simple is good.
- Requires : hashing, Bloom filter, counters, timers, comparisons.
- Small memory footprint very effective.
- Cheap to add in terms of space/cost.

Open Questions / Issues

- Importance of persistent source assumption?
  - Carousel fails for “one-time events”.
- Most effective dynamic re-sizing?
  - Using last k bits of a hash for 2^k buckets leads to fast re-sizing, at cost of optimality of bucket size.
- Tighter analysis on round times?
- De-duplication at endpoint.
  - Generally easily done.
- *Simplicity is often better than optimality*.

Takeaways

- Simple randomized admission control scheme.
  - Bucketize with hashing.
  - Use Bloom filter or similar structure to avoid duplicates.
  - Dynamically size # of buckets for performance.
- Other uses for this type of admission control?
- Importance of design for key memory/speed bottleneck points.

Cuckoo Hashing

The Beginnings

Cuckoo Hashing

Rasmus Pagh* and Flemming Friche Rodler

BRICS**
Department of Computer Science
University of Aarhus, Denmark
{pagh,ffr}@brics.dk

Abstract. We present a simple and efficient dictionary with worst case constant lookup time, equaling the theoretical performance of the classic dynamic perfect hashing scheme of Dietzfelbinger et al. The space usage is similar to that of binary search trees, i.e., three words per key on average. The practicality of the scheme is backed by extensive experiments and comparisons with known methods, showing it to be quite competitive also in the average case.

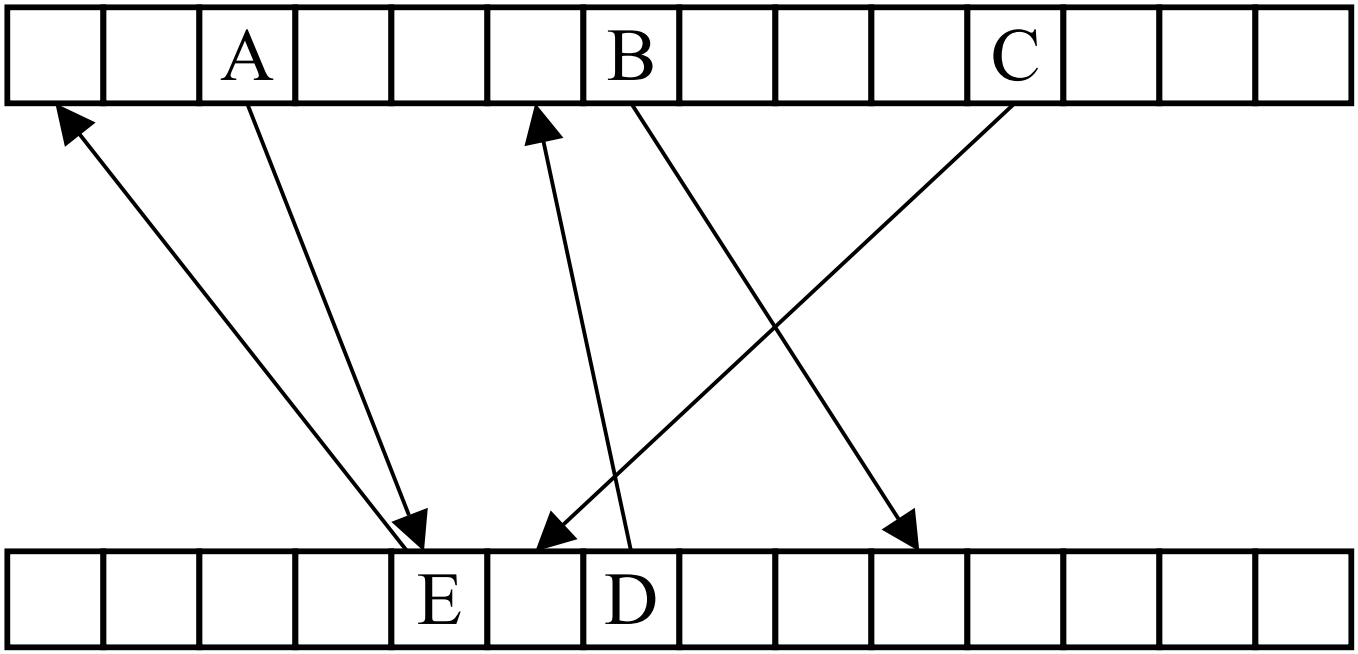
Why Do We Care About Cuckoo Hashing?

- Hash tables a fundamental data structure.
- Multiple-choice hashing yields tables with
  - High memory utilization.
  - Constant time look-ups.
  - Simplicity – easily coded, parallelized.
- Cuckoo hashing expands on this, combining multiple choices with ability to move elements.
- Practical potential, and theoretically interesting!

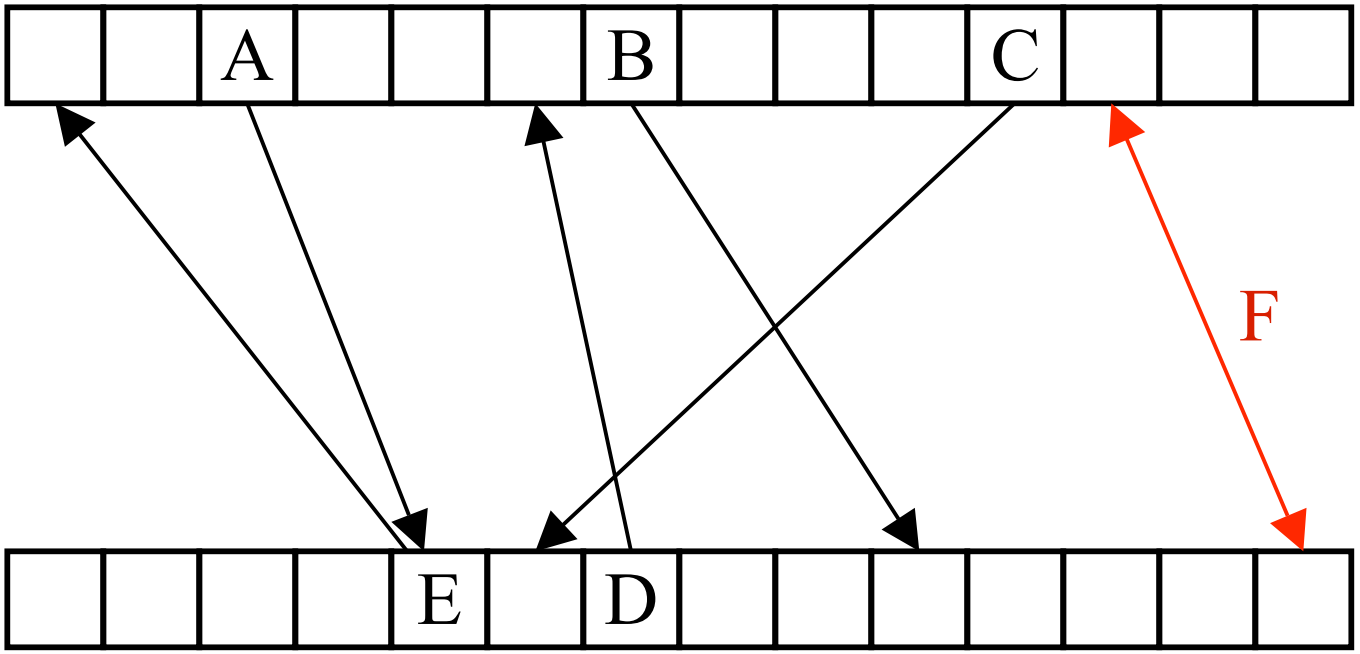
Cuckoo Hashing

- Basic scheme: each element gets two possible locations (uniformly at random).
- To insert *x*, check both locations for *x*. If one is empty, insert.
- If both are full, *x* kicks out an old element *y*. Then *y* moves to its other location.
- If that location is full, *y* kicks out *z*, and so on, until an empty slot is found.

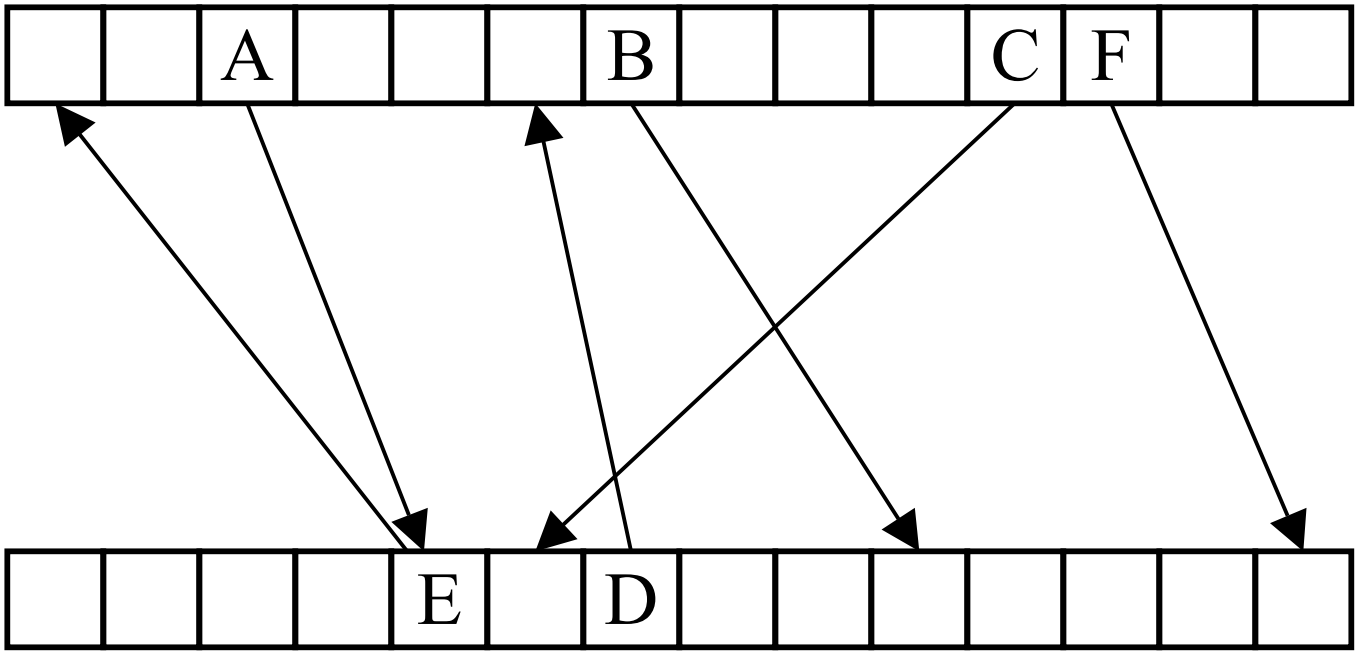
Cuckoo Hashing Examples



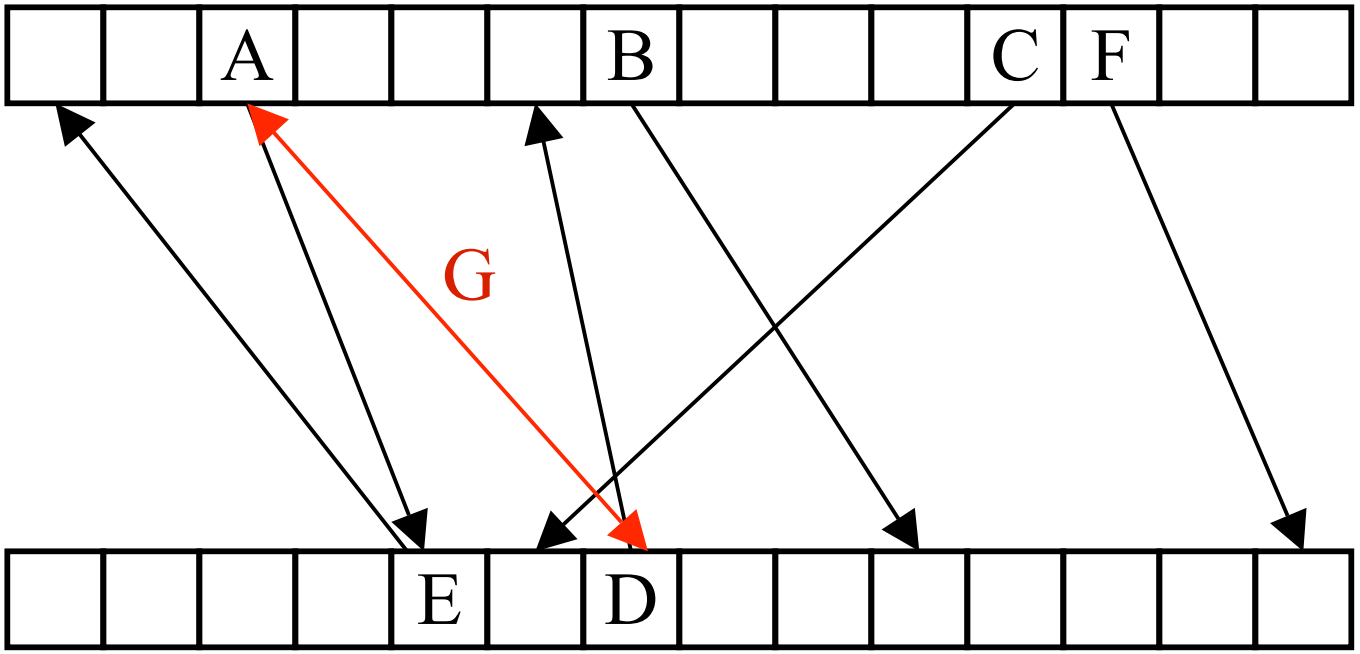
Cuckoo Hashing Examples



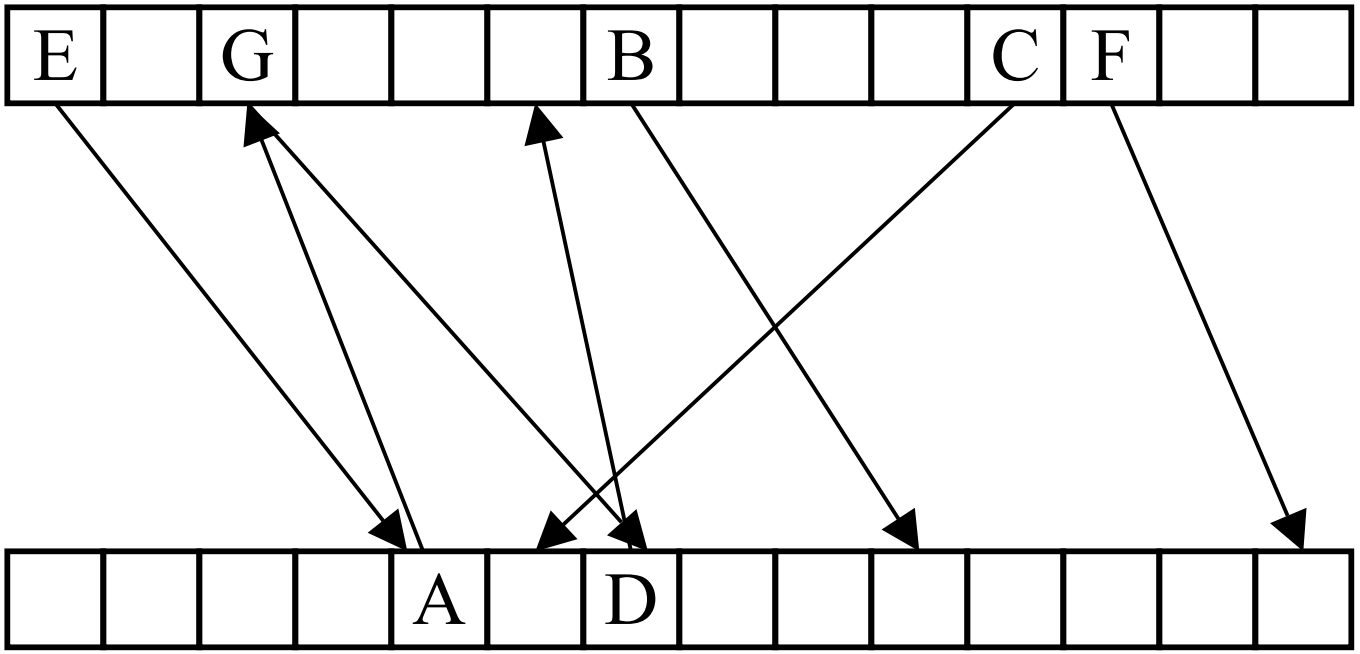
Cuckoo Hashing Examples



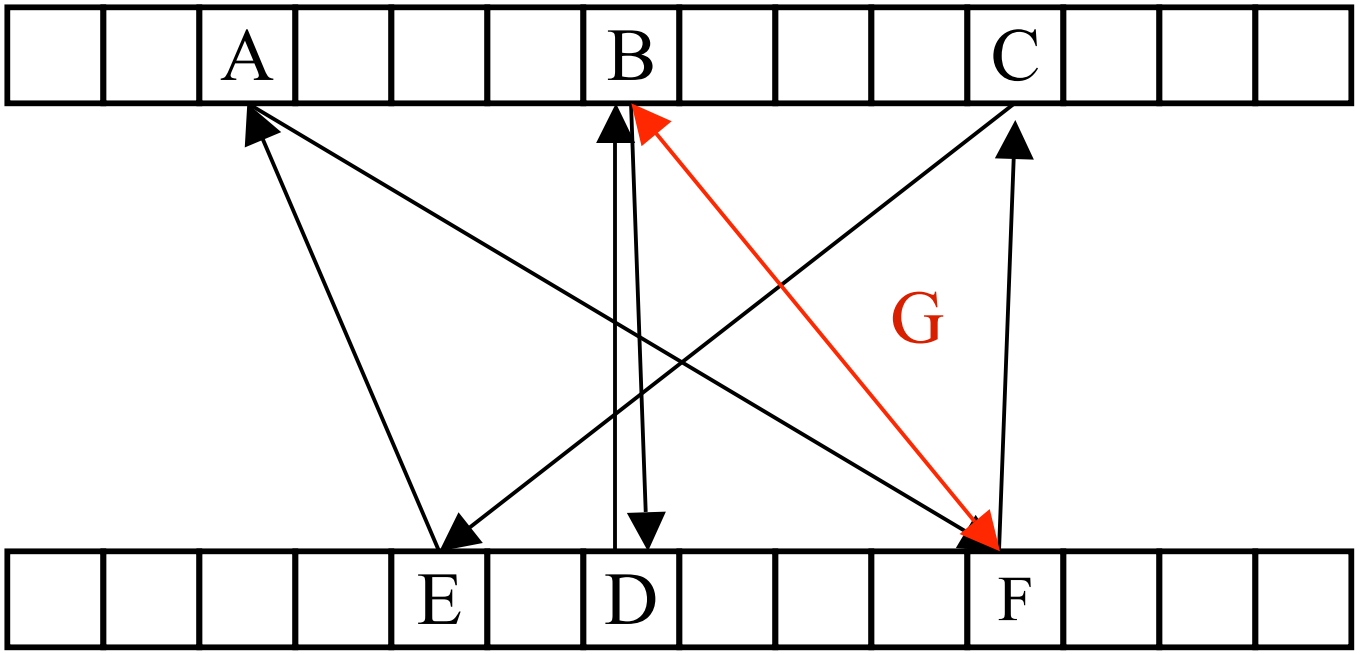
Cuckoo Hashing Examples



Cuckoo Hashing Examples



Cuckoo Hashing Examples



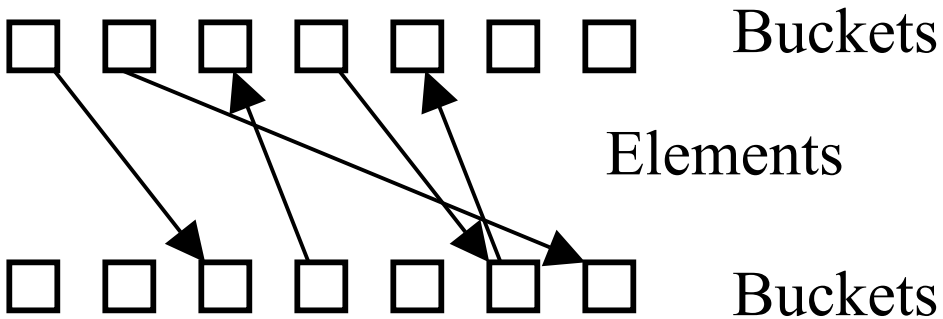
Good Properties of Cuckoo Hashing

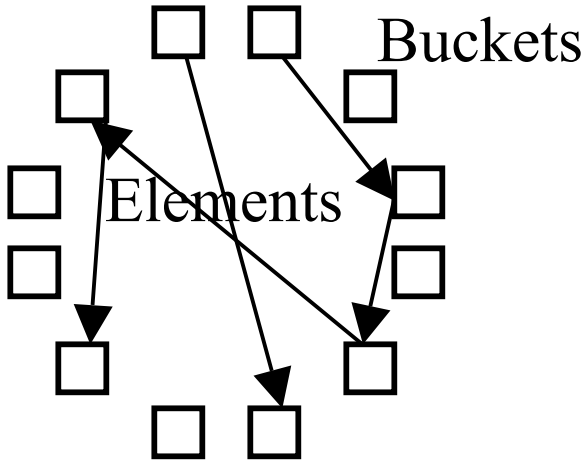
- *Worst case constant lookup time.*
- High memory utilizations possible.
- Simple to build, design.

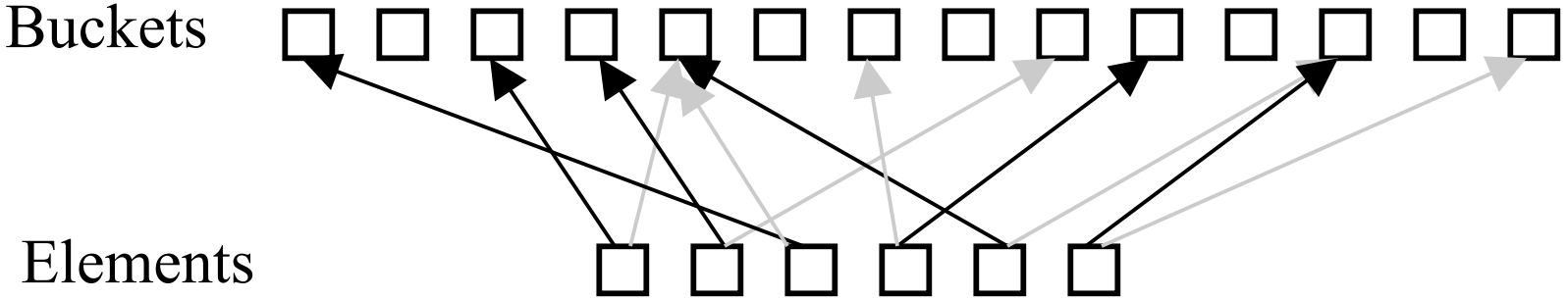
Cuckoo Hashing Failures

- Bad case 1: inserted element runs into cycles.
- Bad case 2: inserted element has very long path before insertion completes.
  - Could be on a long cycle.
- Bad cases occur with very small probability when load is sufficiently low.
- Theoretical solution: re-hash everything if a failure occurs.

Various Representations







Basic Performance

- For 2 choices, load less than 50%, n elements gives failure rate of $\Theta(1/n)$; maximum insert time $O(\log n)$.
- Related to random graph representation.
  - Each element is an edge, buckets are vertices.
  - Edge corresponds to two random choices of an element.
  - Small load implies small acyclic or unicyclic components, of size at most $O(\log n)$.

Natural Extensions

- More than 2 choices per element.
  - Very different : hypergraphs instead of graphs.
  - D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis.
  - Space efficient hash tables with worst case constant access time.
  - More than 2 choices is important.
  - Much higher memory utilizations; 3 choices : 90%+, 4 choices : about 97%.
- More than 1 element per bucket.
  - M. Dietzfelbinger and C. Weidling.
  - Balanced allocation and dictionaries with tightly packed constant size bins.

Recent Work: Parallel Architectures

- Multicores, Graphics Processor Units (GPUs), other parallel architectures possibly the next wave.
- Multiple-choice hashing and cuckoo hashing seem naturally parallelizable.
- Theory and practice?

Related Work

- Plenty on parallel hashing/load balancing schemes.
  - PRAM emulation, related work in the 1990s.
- Technical improvements of last decade suggest more is possible.
- In Amenta et al., we designed new implementation for GPUs based on cuckoo hashing.
  - New theory, practical implementations possible?