Algorithms and Data Structures to Accelerate Network Analysis

Reservoir Labs

Jordi Ros-Giralt, Alan Commike, Peter Cullen, Richard Lethin
{giralt, commike, cullen, lethin}@reservoir.com

4th International Workshop on Innovating the Network for Data Intensive Science
November 12, 2017
Roadmap

• Problem definition
• Optimizations
  • Long queue emulation
  • Lockless bimodal queues
  • Tail early dropping
  • LFN tables
  • Multiresolution priority queues
• Benchmarks
• System wide optimization of network components like routers, firewalls, or network analyzers is complex.
• Hundreds of different SW algorithms and data structures interrelated in subtle ways.
• Two inter-related problems:
  • Shifting micro-bottlenecks
  • Nonlinear performance collapse
It’s difficult...
Shifting Micro-Bottlenecks

...to optimize...
Shifting Micro-Bottlenecks

...bottlenecks...
Shifting Micro-Bottlenecks

...that keep moving...
Shifting Micro-Bottlenecks

...every microsecond...
...or so.
Non-linear Performance Collapse

- Net I/O
- PCIE
- CPU
- Disk I/O
- Cache
- Memory

Network Bandwidths:
- Net I/O: 40 Gbps
- PCIE: 64 Gbps
- CPU: 56 GHz
- Disk I/O: 10.4 Gbps
- Cache: 1092 Gbps

Cache Sizes:
- L1-I cache: 896 kB
- L1-D cache: 896 kB
- L2 cache: 7168 kB
- L3 cache: 71680 kB
Non-linear Performance Collapse

Healthy cache regime:
- CPU operates out of cache
- High cache hit ratios

State 1: network is the bottleneck
Non-linear Performance Collapse

Highly inefficient memory regime:
- CPU operates out of RAM
- High cache miss ratios

State 2: network is no longer the bottleneck

Network I/O

PCIE

Disk I/O

Cache

Memory

L1-I cache: 896 kB
L1-D cache: 896 kB
L2 cache: 7168 kB
L3 cache: 71680 kB

10x penalty

10.4 Gbps

64 Gbps

56 GHz

1092 Gbps
Non-linear Performance Collapse

By removing the network bottleneck, system spends more time processing packets that will need to be dropped anyway → net performance degradation (performance collapse)

Highly inefficient memory regime:
- CPU operates out of RAM
- High cache miss ratios

State 2: network is no longer the bottleneck

- Net I/O
  - 40 Gbps
- PCIE
  - 64 Gbps
- CPU
  - 56 GHz
- Disk I/O
  - 10.4 Gbps
- Cache
  - L1-I cache: 896 kB
  - L1-D cache: 896 kB
  - L2 cache: 7168 kB
  - L3 cache: 71680 kB
- Memory
  - 1092 Gbps
The process of performance optimization needs to be a meticulous one involving small but safe steps to avoid the pitfall of pursuing short term gains that can lead to new and bigger bottlenecks down the path.
Long queue emulation: Reduces packet drops due to fixed-size hardware rings

Lockless bimodal queues: Improves packet capturing performance

Tail early dropping: Increases information entropy and extracted metadata

LFN tables: Reduces state sharing overhead

Multiresolution priority queues: Reduces cost of processing timers
Long Queue Emulation

Dispatcher Model:

- Packet read cache penalty.
- Descriptor read cache penalty

Long queue emulation Model:

- Packet drop penalty under certain conditions
Lemma 1. Long queue emulation performance.

λ : average packet arrival rate
λ_{max} : maximum packet arrival rate
μ_{dt} : packet processing rate of the DT model
μ_{lqe} : packet processing rate of the LQE model
s_{lsr} : size of the LSR ring

\[ \lambda_{max} \leq s_{lsr} \cdot \mu_{lqe} ? \]

- Yes: Use LQE
- No: \[ \lambda \geq \mu_{lqe} \]
  - Yes: Use LQE
  - No: Use DT
Long Queue Emulation

<table>
<thead>
<tr>
<th>$\lambda_{\text{max}}$ (Gbps)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{\text{lsr}}/\lambda_{\text{max}}$ (secs)</td>
<td>1.09</td>
<td>0.55</td>
<td>0.27</td>
<td>0.18</td>
<td>0.14</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 1. Maximum packet processing time for a Solarflare SFN7122F NIC

Table 2. Packet processing time distribution.

<table>
<thead>
<tr>
<th></th>
<th>[0, 10us)</th>
<th>[10us, 100us)</th>
<th>[100us, 1ms)</th>
<th>[1ms, 10ms)</th>
<th>[10ms, 100ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>305</td>
<td>405493</td>
<td>3387846</td>
<td>127</td>
<td>7</td>
</tr>
<tr>
<td>Total packets:</td>
<td>3793778</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Packet processing time distribution

$\lambda_{\text{max}} \leq s_{\text{lsr}} \cdot \mu_{\text{lqe}}$?

Use LQE
• Optimal LQE size

Packet drops at 10Gbps

<table>
<thead>
<tr>
<th>Size of USQ (buffers)</th>
<th>Drops %</th>
</tr>
</thead>
<tbody>
<tr>
<td>150 buffers</td>
<td>5.769</td>
</tr>
<tr>
<td>750 buffers</td>
<td>3.065</td>
</tr>
<tr>
<td>1500 buffers</td>
<td>1.361</td>
</tr>
<tr>
<td>2048 buffers</td>
<td>1.956</td>
</tr>
<tr>
<td>4096 buffers</td>
<td>2.623</td>
</tr>
</tbody>
</table>
• Goal: move packets from the memory ring to the disk without using locks
• Goal: move packets from the memory ring to the disk without using locks
Lockless bimodal queue without using CAS
(producer must be permanently active to avoid consumer starvation)

1 typedef struct {
2   volatile unsigned int offset_p;
3   volatile unsigned int offset_c;
4   volatile bool req; // owned by consumer
5   volatile bool ack; // owned by producer
6   packet_t* vector[RINGSIZE];
7 } ring_t;

8 void enqueue(ring_t* ring, packet_t* pkt) {
9   if(!ring->req) {
10      if(ring->ack)
11         ring->ack = false;
12      if(ring->offset_p == ring->offset_c)
13         dequeue(ring);
14   }
15   else {
16      if(!ring->ack)
17         ring->ack = true;
18      while(ring->offset_p == ring->offset_c);
19   }
20   ring->vector[ring->offset_p++] = pkt;
21 }

22 packet_t* dequeue(ring_t* ring) {
23   if(ring->offset_p == ring->offset_c)
24      return NULL;
25   ring->offset_c = ring->offset_c + 1 % RINGSIZE;
26   return(vector[ring->offset_c - 1]);
27 }

28 void start_c(ring_t* ring) {
29   ring->req = true;
30   while(!ring->ack);
31 }

32 void stop_c(ring_t* ring) {
33   ring->req = false;
34   while(ring->ack);
35 }

Lockless bimodal queue using CAS
(producer does not need to be permanently active)

1 typedef struct {
2   volatile unsigned int offset_p;
3   volatile unsigned int offset_c;
4   volatile bool trans; // used to transition modes
5   volatile bool state; // the current mode
6   packet_t* vector[RINGSIZE];
7 } ring_t;

8 void enqueue(ring_t* ring, packet_t* pkt) {
9   while(!cas(&ring->lock, false, true));
10   if(!ring->state) {
11      if(ring->offset_p == ring->offset_c)
12         dequeue(ring);
13   }
14   else {
15      while(ring->offset_p == ring->offset_c);
16      ring->trans = false;
17      ring->vector[ring->offset_p++] = pkt;
18   }
19   packet_t* dequeue(ring_t* ring) {
20      if(ring->offset_p == ring->offset_c) return NULL;
21      ring->offset_c = ring->offset_c + 1 % RINGSIZE;
22      return(ring->offset_c - 1);
23   }

24 void start_c(ring_t* ring) {
25      while(!cas(&ring->trans, false, true));
26      ring->state = true;
27      ring->trans = false;
28 }

29 void stop_c(ring_t* ring) {
30      while(!cas(&ring->trans, false, true));
31      ring->state = false;
32      ring->trans = false;
33 }

Reservoir Labs
Tail Early Dropping

Information/Entropy

Connection bits in sequence of arrival

$\lambda_{ted}$
Tail Early Dropping

\[ I(b_n) \]

\[ \lambda_{ted} \]

Connection bits in sequence of arrival

---

**TED**

1. Upon receiving a packet, do:
   2. `conn = lookup_connection_table(packet)`
   3. if `conn.shunt` or `conn.packet_rec > ted_thr`:
      4. drop the packet
   5. else:
      6. forward the packet
   7. Periodically, do:
      8. if system is congested:
         9. `ted_thr = \min(ted_thr / 2, ted_min)`;
      10. else:
         11. `ted_thr += 1;`
Tail Early Dropping

**HTTP events**

<table>
<thead>
<tr>
<th>Throughput</th>
<th>% captured events</th>
<th>w/o TED</th>
<th>w/ TED</th>
</tr>
</thead>
<tbody>
<tr>
<td>500Mbps</td>
<td>99.88</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>5Gbps</td>
<td>36.11</td>
<td>90.52</td>
<td></td>
</tr>
</tbody>
</table>

**File events**

<table>
<thead>
<tr>
<th>Throughput</th>
<th>% captured events</th>
<th>w/o TED</th>
<th>w/ TED</th>
</tr>
</thead>
<tbody>
<tr>
<td>500Mbps</td>
<td>99.25</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>5Gbps</td>
<td>26.4</td>
<td>83.69</td>
<td></td>
</tr>
</tbody>
</table>

*Throughput*
LFN Tables

\[ \text{put}(o,s) \rightarrow (o,s) \]

- **False negative**: \( s' \neq s \) and \( s' \neq \text{null} \)
- **False positive**: \( s' \neq s \) and \( s' \neq \text{null} \)

True

\[ \text{get}(o) \rightarrow (o,s) \]
LFN Tables

Initial state: $T[e] = \text{NULL}$ for all $e$ such that $0 \leq e < n$

Parameters:

- $n$: size of the table
- $l$: processor’s integer space size (typically $2^{32}$ or $2^{64}$)
- $h(x, k)$: the hash value of $k$ modulo $x$
- $\text{cat}(x, y)$: concatenates the bytes from and $x$ and $y$

```
put(k, v)
1   T[h(n, k)].value = v
2   T[h(n, k)].hash = h[l, cat(k, v)]

get(k)
3   if T[h(n, k)].hash == h(l, cat(k, T[h(n, k)].value)):
4       return T[h(n, k)].value
5   else:
6       return NULL
```
LFN Tables

![Graph 1: % of false negatives versus true states vs. number of objects (k)]

- $n = 10^6$
- $n = 10^5$
- $n = 10^4$

![Graph 2: % of false positives versus true states vs. number of objects (k)]

- $l = 2^{32}$
- $l = 2^{24}$
Multiresolution Priority Queues

- Priority queue: element at the front of the queue is the $greatest$ of all the elements it contains, according to some total ordering defined by their $priority$.

- Found at the core of important computer science problems:
  - Shortest path problem
  - Packet scheduling in Internet routers
  - Event driven engines
  - Huffman compression codes
  - Operating systems
  - Bayesian spam filtering
  - Discrete optimization
  - Simulation of colliding particles
  - Artificial intelligence
### Multiresolution Priority Queues

<table>
<thead>
<tr>
<th>Year</th>
<th>Author</th>
<th>Data structure</th>
<th>Insert</th>
<th>Extract</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1964</td>
<td>Williams [3]</td>
<td>Binary heap</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
<td>Simple to implement.</td>
</tr>
<tr>
<td>1984</td>
<td>Fredman et al. [4]</td>
<td>Fibonacci Heaps</td>
<td>O(1)</td>
<td>O(log(n))</td>
<td>More complex to implement.</td>
</tr>
<tr>
<td>1988</td>
<td>Brown [8]</td>
<td>Calendar queues</td>
<td>O(1)</td>
<td>O(c)</td>
<td>Need to be balanced and resolution cannot be tuned.</td>
</tr>
<tr>
<td>2008</td>
<td>Mehlhorn et al. [7]</td>
<td>Bucket queues</td>
<td>O(1)</td>
<td>O(c)</td>
<td>Priorities must be small integers and resolution cannot be tuned.</td>
</tr>
<tr>
<td>2017</td>
<td>Ros-Giralt et al. (this work)</td>
<td>Multiresolution priority queue</td>
<td>O(1), O(r) or O(log(r))</td>
<td>O(1)</td>
<td>Tunable/bounded resolution error. Error is zero if priority space is multi-resolutive.</td>
</tr>
</tbody>
</table>

n: number of elements in the queue  
c: maximum integer priority value  
r: number of resolution groups supported by the multiresolution priority queue
A multiresolution priority queue is a container data structure that at all times maintains the following invariant:

Property 1. Multiresolution Priority Queue (MR-PQ) Invariant. Let $e_i$ and $e_j$ be two arbitrary elements with priorities $p_i$ and $p_j$, respectively, where $p_{\text{min}} \leq p_i < p_{\max}$ and $p_{\text{min}} \leq p_j < p_{\max}$. Then for all possible states, a multiresolution priority queue ensures that element $e_i$ is dequeued before element $e_j$ if the following condition is true:

$$\left[\frac{(p_i - p_{\text{min}})}{p_{\Delta}}\right] < \left[\frac{(p_j - p_{\text{min}})}{p_{\Delta}}\right]$$

(1)

Intuitively:

1. Discretize the priority space into a sequence of slots or resolution groups $[p_{\text{min}}, p_{\text{min}} + p_{\Delta})$, $[p_{\text{min}} + p_{\Delta}, p_{\text{min}} + 2 \cdot p_{\Delta})$, $\ldots$, $[p_{\text{max}} - p_{\Delta}, p_{\max})$
2. Prioritize elements according to the slot in which they belong.
3. Elements belonging to lower slots are given higher priority.
4. Within a slot, ordering is not guaranteed. This enables a mechanism to control the trade-off accuracy versus performance.
Multiresolution Priority Queues

- The larger the parameter $p_\Delta \rightarrow$ the lower the resolution of the queue $\rightarrow$ the higher the error $\rightarrow$ the higher the performance (and vice versa)

- Instead of ordering the space of elements, an MR-PQ orders the space of priorities.

- The information theoretic barriers of the problem are broken by introducing error in a way that entropy is reduced:
  - In many real world problems, the space of priorities has much lower entropy than the space of keys.
  - Example:
    - Space of keys is the set of real numbers ($S_k$)
    - Space of priorities is the set of distances between any two US cities ($S_p$)
    - $\text{Entropy}(S_k) >> \text{Entropy}(S_p)$
• How it works through an example.

Let a multiresolution priority queue have parameters $p_\Delta = 3$, $p_{\text{min}} = 7$ and $p_{\text{max}} = 31$, and assume we insert seven elements with priorities 19, 11, 17, 12, 24, 22 and 29 (inserted in this order). Then:
**BUILD(q)**

1. sentinel = alloc_element();
2. queue = sentinel;
3. queue->next = queue;
4. queue->prev = queue;
5. qltsize = (pmax-pmin)/pdelta + 1;
6. for i in [1, qltsize):
   7.     qlt[i] = NULL;
8.    qlt[0] = queue;
9. queue->prio = pmin - pdelta;
Multiresolution Priority Queues: Base Algorithm

QLTSLOT(e)

24  slot = (int)((e->prio - queue->prio)/pdelta);
25  return slot;

INSERT(e)

10  slot = slot_iter = QLTSLOT(e);
11  while qlt[slot_iter] == NULL:
12       slot_iter--;
13  if slot_iter == slot: // Add to the left
14      e->next = qlt[slot];
15      e->prev = qlt[slot]->prev;
16      qlt[slot]->prev->next = e;
17      qlt[slot]->prev = e;
18  else: // Add to the right
19      e->next = qlt[slot_iter]->next;
20      e->prev = qlt[slot_iter];
21      qlt[slot_iter]->next->prev = e;
22      qlt[slot_iter]->next = e;
23      qlt[slot] = e;
Multiresolution Priority Queues: Base Algorithm

QLTREPAIR(e)

34    slot = QLT SLOT(e);
35    if qlt[slot] != e:
36        return; // Nothing to fix
37    if slot == QLT SLOT(e->prev):
38        qlt[slot] = e->prev; // Fix the slot
39    else:
40        qlt[slot] = NULL; // No elements left in slot

EXTRACT(e)

30    e->prev->next = e->next;
31    e->next->prev = e->prev;
32    QLTREPAIR(e);
33    return e;
Multiresolution Priority Queues: Base Algorithm

**PEEK()**

26   e = q->next;
27   if e == q:
28       return NULL;  // Queue is empty
29   return e;

**EXTRACTMIN()**

41   e = PEEK();
42   EXTRACT(e);
43   return e;
Lemma 1. Correctness of the MR-PQ algorithm. The $\text{INSERT}$ and $\text{REMOVE}$ routines preserve the MR-PQ invariant (Property 1).

Lemma 2. Complexity of the MR-PQ algorithm. The worst case complexity of the MR-PQ algorithm for the $\text{INSERT}$ routine is $O(r)$, where $r = (p_{\text{max}} - p_{\text{min}})/p_{\Delta}$ is the number of resolution groups supported by the queue. The complexity of the $\text{INSERT}$ routine becomes $O(1)$ if there is at least one element in each slot. The complexity of the $\text{PEEK}$, $\text{EXTRACTMIN}$ and $\text{EXTRACT}$ routines is $O(1)$. 
Multi-resolutive Priority Spaces

Definition. Multi-resolutive priority space. Let $\mathcal{P}$ be the set of possible priorities that the elements in a priority queue can take and assume that $\{p_1(t), p_2(t), \ldots, p_n(t)\}$ is the set of priorities of the elements stored in the queue at an arbitrary time $t$. We will say that $\mathcal{P}$ is multi-resolutive with resolution $r$ if there exists a set $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_r\}$ such that the following three conditions are true

1. $\bigcup_{i=1}^{r} \mathcal{P}_i = \mathcal{P}$ and $\mathcal{P}_i \cap \mathcal{P}_j = \emptyset$ for $i \neq j$ (i.e., $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_r\}$ is an $r$-part partition of $\mathcal{P}$)

2. if $p_i(t) \in \mathcal{P}_k$, then $p_j(t) \notin \mathcal{P}_k$, $\forall j \neq i$ and $\forall t$

3. and $r$ is minimal.

- Example:

Consider the priority set of real numbers, $\mathcal{P} = \mathbb{R}$ and assume that at any arbitrary time, any two elements $e_i$ and $e_j$ in the queue are guaranteed to not have priorities closer than $a$ nor further than $b$: $|p_i - p_j| > a$ and $|p_i - p_j| < b$. Then we can say that $\mathcal{P}$ is multi-resolutive with resolution $r = \lceil b/a \rceil$. 
Multi-resolutive Priority Spaces

- Problems with multi-resolutive priority spaces can be resolved by a multi-resolution priority queue at a faster speed and without adding any additional error. In this case, we achieve better performance at no cost.
- If condition (2) does not hold, then an error is introduced but the entropy of the problem stays constant. In this case, we also achieve better performance but at the cost of losing some accuracy.
• We can apply the rules of multi-resolutive priority spaces to optimize the performance of problems involving priority queues.

• Example. Consider the classic shortest path problem, known to have a complexity of $O((v+e)\log(v))$, for a graph with $v$ vertices and $e$ edges (See Section 24.3 of [2]). By using a multiresolution priority queue we have:

  - If the graph is such that the edge weights define a multi-resolutive priority space, then using MR-PQ we can find the exact shortest path with a cost $O(v+e)$.
  - Otherwise, we can find the *approximate* shortest path with a cost $O(v+e)$ and with a controllable error given by the parameter $r$. 
• The base MR-PQ algorithm assumes priorities are in the set $[p_{\text{min}}, p_{\text{max}})$.

• Data structure can be generalized to support priorities in the set $[p_{\text{min}} + d(t), p_{\text{max}} + d(t))$, where $d(t)$ is any monotonically increasing function of a parameter $t$.

• The case of sliding priority sets is particularly relevant to applications that run event-driven engines. (See for example Section 6.5 of [2].)
Multiresolution Priority Queues: Support for Sliding Priorities

- Sliding priorities can be supported with a few additional lines of code:

```c
QLTSLIDE(prio)

44 shift = (prio - queue->prio)/pdelta - 1;
45 if shift < 1:
46    return;
47 for i in [1, qltsize):
48    if i < qltsize - shift:
49        qlt[i] = qlt[i + shift];
50    else
51        qlt[i] = 0;
52 queue->prio = prio - pdelta;
```

```c
EXTRACTMIN()

53 e = PEEK();
54 EXTRACT(e);
55 QLTSLIDE(e->prio);    // Added to support sliding
                      // priorities
56 return e;
```

Lemma 3. Correctness of the MR-PQ algorithm with sliding priorities. The modified `EXTRACTMIN` routine preserves the MR-PQ invariant (Property 1).
• When not all the slots in the QLT table are filled in, performance can be improved by implementing the QLT table using a binary heap.

• Let a multiresolution priority queue have parameters $p_\Delta = 3$, $p_{\min} = 7$ and $p_{\max} = 31$, and assume we insert seven elements with priorities 19, 11, 17, 12, 24, 22 and 29 (inserted in this order). Then:
### Table 1. Computational cost.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>INSERT</th>
<th>PEEK</th>
<th>EXTRACT_MIN</th>
<th>EXTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH-PQ</td>
<td>$\log(n)$</td>
<td>$O(1)$</td>
<td>$\log(n)$</td>
<td>$\log(n)$</td>
</tr>
<tr>
<td>MR-PQ</td>
<td>$O(r)$ or $O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>BT-MR-PQ</td>
<td>$O(\log(r))$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
• We use MR-PQ to resolve a real world HPC problem.

• Problem statement: when running the Bro network analyzer [12] against very high speed traffic consisting of many short lived connections, the BubbleDown operation in the (binary heap based) priority queue used to manage Bro timers becomes a main system bottleneck.

• Top functions in Bro according to their computational cost:

Total: 63724 samples

<table>
<thead>
<tr>
<th>Function</th>
<th>Samples</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue::BubbleDown</td>
<td>4139</td>
<td>6.5%</td>
</tr>
<tr>
<td>SLL_Pop</td>
<td>2500</td>
<td>3.9%</td>
</tr>
<tr>
<td>Ref</td>
<td>1899</td>
<td>3.0%</td>
</tr>
<tr>
<td>Unref</td>
<td>1829</td>
<td>2.9%</td>
</tr>
<tr>
<td>PackedCache::KeyMatch</td>
<td>1701</td>
<td>2.7%</td>
</tr>
<tr>
<td>Attributes::FindAttr</td>
<td>1537</td>
<td>2.4%</td>
</tr>
<tr>
<td>Dictionary::Lookup</td>
<td>1249</td>
<td>2.0%</td>
</tr>
<tr>
<td>NameExpr::Eval</td>
<td>1184</td>
<td>1.9%</td>
</tr>
</tbody>
</table>
• Call graph showing the BubbleDown function as the main bottleneck.
BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers
BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers
BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers
BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers
BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers
R-Scope Network Security Sensor: System Wide Benchmarks

Events (Millions)

Throughput (Mbps)

- Stock Bro Myri
- R-Scope Myri/mCore
- R-Scope SF/mCore+

For various throughput levels (2000, 4000, 6000, 8000, 10000 Mbps):
- Stock Bro Myri: 26, 28, 28, 29, 17
- R-Scope Myri/mCore: 33, 42, 45, 45, 43
- R-Scope SF/mCore+: 36, 57, 75, 84, 87

Graph showing system wide benchmarks for different throughput levels.
Connections (Millions)

Throughput (Mbps)
DENVER TO BECOME EPICENTER FOR FASTEST INTERNET AND COMPUTERS IN THE WORLD

Denver to Become Epicenter for Fastest Internet and Computers in the World

Massive 1 Terabit Network to Support High Performance Computing Demonstrations at SC13 Conference
You can visit our demo at the SCinet NOC
Thank You

**Reservoir Labs**

632 Broadway  
Suite 803  
New York, NY 10012

812 SW Washington St.  
Suite 1200  
Portland, OR 97205