DRFQ: Multi-Resource Fair Queueing for Packet Processing

Ali Ghodsi\textsuperscript{1,3}, Vyas Sekar\textsuperscript{2}, Matei Zaharia\textsuperscript{1}, Ion Stoica\textsuperscript{1}
\textsuperscript{1}UC Berkeley, \textsuperscript{2}Intel ISTC/Stony Brook, \textsuperscript{3}KTH
Increasing Network Complexity

• Packet processing becoming evermore *sophisticated*
  – Software Defined Networking (SDN)
  – Middleboxes
  – Software Routers (e.g. RouteBricks)
  – Hardware Acceleration (e.g. SSLShader)

• Data plane no longer merely forwarding
  – WAN optimization
  – Caching
  – IDS
  – VPN
Motivation

• Flows increasingly have heterogeneous resource consumption
  – Intrusion detection bottlenecking on CPU
  – Small packets bottleneck memory-bandwidth
  – Unprocessed large packets bottleneck on link bw

Scheduling based on a single resource insufficient
Problem

How to schedule packets from different flows, when packets consume *multiple resources*?

How to generalize fair queueing to multiple resources?
Contribution

<table>
<thead>
<tr>
<th>Resource Fairness</th>
<th>Allocation in Space</th>
<th>Allocation in Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Resource Fairness</td>
<td>Max-Min Fairness</td>
<td>Fair Queueing</td>
</tr>
<tr>
<td>Multi-Resource Fairness</td>
<td>DRF</td>
<td></td>
</tr>
</tbody>
</table>

Generalize Virtual Time to Multiple Resources
Outline

• Analysis of Natural Policies
• DRF allocations in Space
• DRFQ: DRF allocations in Time
• Implementation/Evaluation
Desirable Multi-Resource Properties

• **Share guarantee:**
  – Each flow can get $1/n$ of at least one resource

• **Strategy-proofness:**
  – A flow shouldn’t be able to finish faster by increasing the resources required to process it.
Violation of Share Guarantee

- Example of FQ applied to a
  - Two resources CPU and NIC, *used serially*
  - Two flows with profiles <2 μs, 1 μs> and <1 μs, 1 μs>
  - FQ based on NIC alternates one packet from each flow

Share Guarantee Violated by Single Resource FQ
Violation of Strategy-Proofness

• **Bottleneck fairness**
  – Determine which resource is bottlenecked
  – Apply FQ to that resource

• Example with Bottleneck Fairness
  – 2 resources (CPU, NIC), 3 flows <10,1>, <10,14>, <10,14>
  – *CPU bottlenecked* and split equally
  – Flow 1 changes to <10,7>. *NIC bottlenecked* and split equally

Bottleneck Fairness Violates Strategy-Proofness
Bottleneck Fairness and Multiple Bottleneck

• Example with 2 flows and 2 res. $<\text{CPU, NIC}>$
  – Demands $<1,6>$ and $<7,1>$  $\rightarrow$ bottleneck unclear
  – CPU bottleneck: $7 \times <1,6> + <7,1> = <14,43>$
  – Oscillates to the other resource
  – NIC bottleneck: $<1,6> + 6 \times <7,1> = <43,12>$
Natural Policy

- **Per-Resource Fairness (PRF)**
  - Have a buffer between each resource
  - Apply fair queueing to each resource

- Per-Resource Fairness not strategy-proof
  - 2 resources, 2 flows <4,1>, <1,2>
    - Flow 1 changes demand to <4,2>

![Bar chart showing CPU and NIC usage for Flow 1 and Flow 2]
Problems with Per-Resource Fairness

• PRF *violates* strategy-proofness
  – Can be manipulated by wasting resources

• PRF requires *per-resource queues*
  – Problematic for parallel resource consumption
    e.g. CPU and memory consumption in a module
Why care about strategy-proofness?

• Lack of strategy-proofness encourages *wastage*
  – Decreasing goodput of the system

• Networking applications especially savvy
  – Peer-to-peer apps manipulate to get more resources

• Trivially *guaranteed* for single resource fairness
  – But not for multi-resource fairness
# Summary of Policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Share Guarantee</th>
<th>Strategy-Proofness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fair Queueing a Single Resource</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bottleneck Fairness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Per-Resource Fairness</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Dominant Resource Fairness</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Outline

• Analysis of Natural Policies

• DRF allocations in Space

• DRFQ: DRF allocations in Time

• Implementation/Evaluation
Dominant Resource Fairness

• DRF originally in the cloud computing context
  – Satisfies share guarantee
  – Satisfies strategy-proofness
DRF Allocations

• *Dominant resource* of a user is the resource she is allocated most of
  – *Dominant share* is the user’s share of her dominant resource

• *DRF*: apply max-min fairness to dominant shares
  – “Equalize” the dominant share of all users

Total resources: <16 CPUs, 16 GB mem>
User 1 demand: <3 CPU, 1 GB mem> dom res: CPU
User 2 demand: <1 CPU, 4 GB mem> dom res: mem
Allocations in Space vs Time

• DRF provides allocations *in space*
  – Given 1000 CPUs and 1 TB mem, how much to allocate to each user

• DRFQ provides DRF allocations *in time*
  – Multiplex packets to achieve DRF allocations over time
Outline

• Analysis of Natural Policies
• DRF allocations in Space
  • DRFQ: DRF allocations in Time
• Implementation/Evaluation
Determining Packet Resource Consumption

• A-priori packet link usage known in FQ
  – Packet size divided by throughput of link

• Packet processing time a-priori unknown for multi-resources
  – Depends on the modules that process it

• Leverage Start-time Fair Queueing (SFQ)
  – Schedules based on virtual start time of packets
  – Start time of packet \( p \) independent of resource consumption of packet \( p \)
Memoryless Requirement

• Virtual Clock simulates flows with dedicated 1/n link
  – Attach *start* and *finish tags* according to dedicated link
  – Serve packet with *smallest* finish tag (work conserving)

• *Problem*
  – During light load a flow might get more than 1/n
  – That flow experiences long delays when new flows start

• Requirement: *memoryless scheduling*
  – A flow’s share of resources should be independent of its share in the past
Memoryless through Virtual Time

• *Virtual time* to track amount service received
  – A unit of virtual time always corresponds to same amount of service

• Example with 2 flows
  – Time 0: one backlogged flow
  – Time 20: two backlogged flows

• Schedule the packets according to V(t)
  – Assign virtual start/finish time when packet arrives
Dove-tailing Requirement

• FQ: flow size should determine service, not packet size
  – Flow with 10 1kb packets gets same service as 5 2kb packets

• Want flow processing time, not packet processing time
  – Example: give same service to these flows:
    Flow 1: \( p_1 <1,2>, p_2 <2,1>, p_3 <1,2>, p_4 <2,1>, \ldots \)
    Flow 2: \( p_1 <3,3>, p_2 <3,3>, p_3 <3,3>, p_4 <3,3>, \ldots \)

• Requirement: dove-tailing
  – Packet processing times should be independent of how resource consumption is distributed in a flow
Tradeoff

• Dovetailing and memoryless property *at odds*
  - Dovetailing needs to remember past consumption

• DRFQ developed in three steps
  - *Memoryless DRFQ*: uses a single virtual time
  - *Dovetailing DRFQ*: use virtual time per resource
  - *DRFQ*: generalizes both, tradeoff between memoryless and dovetailing
Memoryless DRFQ

• Attach a *virtual start* and *finish time* to every packet
  – $S(p)$ and $F(p)$ of packet $p$

• Computing virtual finish time $F(P)$
  1. $F(p) = S(p) + \max_i\{ p_{\text{time}}(p, i) \}$
    \[
p_{\text{time}}(p, i) = \{ \text{processing time of } p \text{ on resource } i \}\]

• Computing virtual start time, $S(p)$
  2. $S(p) = \max( F(p^{-1}), C(t) )$
    \[
    C(t) = \{ \text{max start time of currently serviced packet} \}\]

• Service the packet with *minimum* virtual start time
Memoryless DRFQ

- Attach a *virtual start* and *finish time* to every packet

- Computing virtual finish time
  1. finish time = start time + packet-max-processing-time

- Computing virtual start time
  2. Start time of the first packet in a burst equals the start time of the packet currently serviced (zero if none)
  3. For a backlogged flow, the start time of a packet is equal to finish time of previous packet

- Service the packet with *minimum* virtual start time
Memoryless DRFQ example

- Two flows become backlogged at time 0
  - Flow 1 alternates <1,2> and <2,1> packet processing
  - Flow 2 uses <3,3> packet processing time

1. \( F(p) = S(p) + \max_i\{ p\text{-time}(p, i) \} \)
2. \( S(p) = \max( F(p^{-1}), C(t) ) \)

Flow 1 gets worse service than Flow 2
Dovetailing DRFQ

• Keep track of start and finish time per resource
  – Use max start time per resource for scheduling
  – Dovetail by keeping track of all resource usage
Dovetailing DRFQ example

- Two flows become backlogged at time 0
  - Flow 1 alternates <1,2> and <2,1> packet processing
  - Flow 2 uses <3,3> per packet

Dovetailing ensures both flows get same service
DRFQ algorithm

• DRFQ *bounds* dovetailing to $\Delta$ processing time
  – Dovetail up to $\Delta$ processing time units
  – Memoryless beyond $\Delta$

• DRFQ is a *generalization*
  – When $\Delta=0$ then DRFQ=memoryless DRFQ
  – When $\Delta=\infty$ then DRFQ=dovetailing DRFQ

• Set $\Delta$ to a few packets worth of processing
Outline

• Analysis of Natural Policies
• DRF allocations in Space
• DRFQ: DRF allocations in Time
• Implementation/Evaluation
Isolation Experiment

• DRFQ Implementation in Click
  – 2 elephants: 40K/sec basic, 40K/sec IPSec
  – 2 mice: 1/sec basic, 0.5/sec basic

Non-backlogged flows isolated from backlogged flows
Simulating Bottleneck Fairness

• 2 flows and 2 res. <CPU, NIC>
  – Demands <1,6> and <7,1> → bottleneck unclear

• Especially bad for TCP and video/audio traffic
Summary

• Packet processing becoming evermore sophisticated
  – Consume multiple resources

• Natural policies not suitable
  – Per-Resource Fairness (PRF) not strategy-proof
  – Bottleneck Fairness doesn’t provide isolation

• Proposed *Dominant Resource Fair Queueing (DRFQ)*
  – *Generalization* of FQ to multiple resources
  – Generalizes virtual time to multiple resources
  – Provides tradeoff between memoryless and dovetailing
  – Provides share-guarantee (isolation) and strategy-proofness
Overhead

• 350 MB trace run through our Click implementation

• Evaluate overhead of two modules
  – Intrusion Detection, 2% overhead
  – Flow monitoring, 4% overhead
Determining Resource Consumption

• Resource consumption obvious in routers
  – Packet size divided by link rate

• Generalize consumption to processing time
  – Normalized time a resource takes to process packet

• Normalized processing time
  – e.g. 1 core takes 20μs to service a packet,
    on a quad-core the packet processing time is 5μs
  – Packet processing time ≠ packet service time
Module Consumption Estimation

- Linear estimation of processing time
  - For module m and resource r as function of packet size

  \[ R^2 > 0.90 \] for most modules
Simulating Bottleneck Fairness

• 2 flows and 2 res. <CPU, NIC>
  – Demands <1,6> and <7,1> \(\rightarrow\) bottleneck unclear

  – CPU bottleneck: \(7 \times <1,6> + <7,1> = <14,43>\)
  – NIC bottleneck: \(<1,6> + 6 \times <7,1> = <43,12>\)
  – *Periodically oscillates* the bottleneck
TCP and oscillations

- Implemented Bottleneck Fairness in Click
  - 20 ms artificial link delay added to simulate WAN
  - Bottleneck determined every 300 ms
  - 1 BW-bound flow and 1 CPU-bound flow

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Flow 1 (BW-bound)</th>
<th>Flow 2 (CPU-bound)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running alone</td>
<td>191 Mbps</td>
<td>33 Mbps</td>
</tr>
<tr>
<td>Bottleneck</td>
<td>75 Mbps</td>
<td>32 Mbps</td>
</tr>
<tr>
<td>DRFQ</td>
<td>160 Mbps</td>
<td>28 Mbps</td>
</tr>
</tbody>
</table>

Oscillations in Bottleneck degrade performance of TCP
Multi-Resource Consumption Contexts

• Different modules within a middlebox
  – E.g. Bro modules for HTTP, FTP, telnet

• Different apps on a consolidated middlebox
  – Different applications consume different resources

• Other contexts
  – VM scheduling in hypervisors
  – Requests to a shared service (e.g. HDFS)