Decentralized Task-Aware Scheduling for Data Center Networks

Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, Ant Rowstron

Presented by Eric Dong (yd2dong)

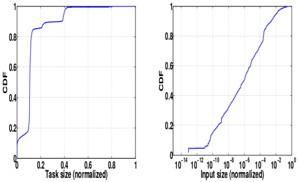
October 30, 2015

Tasks in data centers

- Applications execute rich and complicated *tasks*
- Replying to search queries, gathering information for a news feed, etc
- Each task can involve dozensof flows, *all* of which have to complete for the task to finish

Tasks in data centers

- Two important metrics
- Task size: sum of the sizes of network flows involved
    - All sorts of statistical distributions (ex: search vs. data analytics)



    - Uniform, heavy-tailed, etc
- Flows per task
    - Varies very wildly, from dozens to thousands.
- Scheduling algorithm must work on a wide

Traditional metrics

- Per-flow fair sharing (TCP, DCTCP)
  - Poor average performance when multiple tasks occur at the same time
- Flow-level scheduling metrics (shortest flow first, etc)
  - Considers flows in isolation
  - Example: SFF schedules the shorter flows of *different tasks* first, leaving the longer flows of all the tasks to the end, thus delaying the completion of all the tasks.
- We need something better.
- Unfortunately, this problem is NP-hard :(. But we can use some heuristics!

Task serialization

- The set of policies where an entire task is scheduled before the next. This improves upon fair-sharing because it eliminates contention. One good task serialization algorithm is actually simple: first-in-first-out.

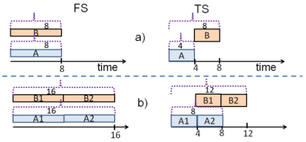


Figure 4: Distilling the Benefits of Task Serialization (TS) over Fair Sharing (FS).

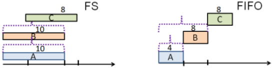


Figure 5: FIFO ordering can reduce tail completion times compared to fair sharing (FS).

- Another example would be shortest-task-first (STF), which improves the average completion time, but leads to high tail latency or even starvation if short tasks keep coming in and preempting long tasks.

Task serialization

- FIFO is great for light-tailed distributions — in fact it's provably optimal for minimizing the tail completion time.
- But it isn't that great for heavy-tailed distributions.
- "Elephant" flows which happen to arrive first end up blocking small flows, increasing latency.

FIFO-LM

- The paper proposes FIFO-LM: first-in-first-out with *limited multiplexing*
- Just like FIFO, but does a limited number of tasks — the degree of multiplexing — at once.
- Hybrid between FIFO (degree $= 1$) and fair-sharing (degree $= \infty$).

- The authors' distributed implementation of FIFO-LM
- No explicit coordination
- Based on globally unique *task-id*s.
  - Lower ID means higher priority
  - Flows inherit the ID of their tasks.
  - Incrementing counter for every point where tasks arrive

Prioritization mechanism

- We have task priorities now. We still need an algorithm that uses them to efficiently schedule tasks.
- We can theoretically use one of the zillions of different existing flow-prioritization algorithms. But they don't have the properties we need.

	Strict Priority	Fair Sharing	Heavy Task Support	Work Conservation	Preemption
DCTCP	No	Yes	No	Yes	No
RCP	No	Yes	No	Yes	No
D^3	Partial	Yes	No	Yes	No
pFabric	Yes	Yes	No	Partial	Yes
PDQ	Yes	No	No	Yes	Yes

Table 2: Desired properties and whether they are supported in existing mechanisms.

- We need a new algorithm.

Smart Priority Class

- Similar to traditional priority queues
  - High-priority flows preempt low-priority flows
  - Flows with the same priority share bandwidth fairly
- Two differences:
  - On-switch classifier: one-to-one mapping between tasks and priorities. Detects heavy tasks on-the-fly, and bump their priority down to that of the next-prioritized class. -LM part in FIFO-LM!
  - Explicit rate control: switches tell senders how quickly to send.
- This moves more work to the end hosts and reduces the overhead of bookkeeping in switches.

Explicit rate protocol

- Every RTT, sender transmits a scheduling request message that demands a certain rate.
- Switch tells sender two numbers
  - Actual rate (AR): how much should be sent in the next RTT
  - Nominal rate (NR): maximum possible rate based on the priority

Algorithm 1 Sender – Generating SRQ

MinNR - minimum NR returned by SRX
$Demand_{t+1} \leftarrow min(NIC\_Rate, DataInBuffer \times RTT)$ //if flow already setup
if $MinNR < Demand_t$ **then**
  $Demand_{t+1} \leftarrow min(Demand_{t+1}, MinNR + \delta)$
end if

Algorithm 2 Switch - SRQ Processing

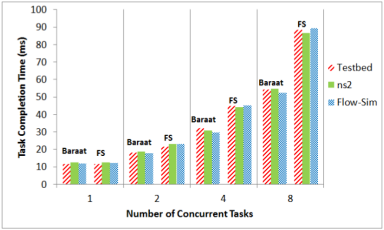
Return Previous Allocation and Demand
$Class = Classifier(TaskID)$
$ClassAvlBW = C - Demand(HigherPrioClasses)$
$AvailShare = ClassAvlBW - Demand(MyClass)$
if $AvailShare > CurrentDemand$ **then**
  $NominalRate(NR) \leftarrow CurrentDemand$
else
  $NR \leftarrow ClassAvlBW / NumFlows(MyClass)$
end if
if $(C - Allocation) > NR$ **then**
  $ActualRate(AR) \leftarrow NR$
else
  $AR \leftarrow (C - Allocation)$

Explicit rate protocol

- We end up implementing FIFO-LM in a distributed way, with no global communication or central controller needed.
- But is it actually a lot better than existing schedulers?
- Experiments!

Evaluation

- The paper evaluates Baraat on three platforms
  - Small scale testbed
  - Huge datacenter simulation
  - Micro-benchmarks



- All show significant improvements compared to other techniques

Small-scale tests

- Storage retrieval scenario: clients read data from storage servers in parallel
- One rack of four nodes running Memcached as the client, four more racks acting as the backend
- One switch connecting everything

	Avg	Min	95^{th} perc.	99^{th} perc.
FS	40ms	11ms	72ms	120ms
Baraat	29ms	11ms	41ms	68ms
Improvement	27%	0	43%	43.3%

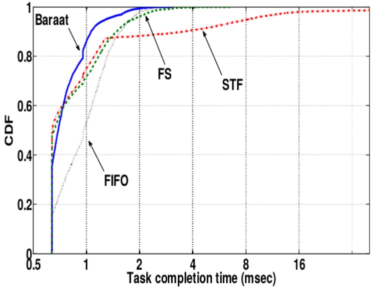
- Very significant improvmeents in task completion time.
- (Nitty-gritty details of setup in paper)

Large datacenter simulation

- Three-level tree topology
- Racks of 40 machines with 1 Gbps links connected to top-of-rack switch and then to aggregator switch
- Three different workloads:
  - Search engine (Bing)
  - Data analytics (Facebook)
  - Homogeneous application: uniformly distributed flow sizes from 2 KB to 50 KB

Bing-like workloads

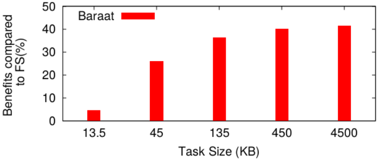
- Policies comparable until the 70th percentile
  - At that point, size-based policies begin starving heavy tasks.
  - Baraat's "limited multiplexing" fixes this problem well.



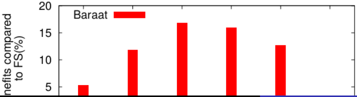
Other two workloads

- Data-analytics workloads are heavy-tailed, and FIFO suffers from head-of-line blocking.
  - Size-based policies reduce completion time relative to fair-sharing here.
    - But still causes starvation issues at the very end of the tail
  - Baraat still much faster
    - 60% faster than fair-sharing
    - 36% faster than size-based policies
- Uniform workloads show benefits too
  - Baraat is 48% faster than fair-sharing
  - Size-based policies have serious starvation issues, and ends up 50% *slower* than fair-sharing.

- The ns-2 network simulator was used to microbenchmark small tasks with tiny flows.
- Still provides significant benefits over fair-sharing due to minimal setup overhead



- We can improve performance even more by breaking our one-task-per-priority-class invariant, and aggregating multiple tiny tasks into a single class.
    - But only up to a point! Otherwise it degenerates into fair-sharing.



Discussion and further work

- Multi-pathing
  - Data centers often have multi-root topologies for path diversity.
  - Existing mechanisms for spreading traffic among paths maintain flow-to-path affinity.
  - So Baraat can be used even in multi-root topologies.
  - Senders can load-balance by sending SRQ packets among different paths
- Non-network resources
  - Baraat doesn't try to schedule non-network resources like CPU
  - This is generally not an issue: Baraat will either saturate the CPU or the network link depending on which is the bottleneck
  - Future work: improve performance even more by coordinating multiple resources.