FnSched: An Efficient Scheduler for Serverless Functions

Amoghavarsha Suresh, Anshul Gandhi
PACE Lab, Stony Brook University

Motivation

Serverless computing is becoming popular

Features:

- Providers responsible for resource management
- Pay-for-what-you-use (runtime)

Benefits:

- Easy deployment: Write your code and ship it!
- Increases programmer productivity
- Seemingly *infinite* scalability

AWS Lambda

Azure Functions

Google Functions

Motivation

- Interest from different domains
  - **Edge-Triggered** applications: e.g. Web apps, backends, data preprocessing
  - **Massively Parallel** applications: e.g. MapReduce, Stream Processing
- Serverless offers cost benefits: 20¢ per 1M lambda requests
  - Ex-Camera [NSDI'17] **serverless** video encoding is **60x** faster and **6x** cheaper than VM based (**serverful)** solution.
- Interest in serverless computing will rise. For a viable service:
  - **Efficient** resource usage **@ scale** is important for the provider
  - ***Reasonable*** performance is important for the **user**

Smart scheduling and resource management is critical

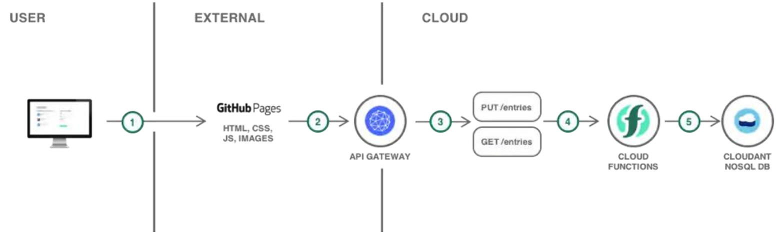
Outline

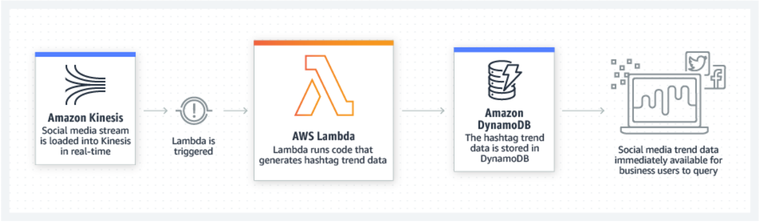
- Motivation
- Scheduling Challenges
- FnSched Design
- Evaluation
- Conclusion & Future work

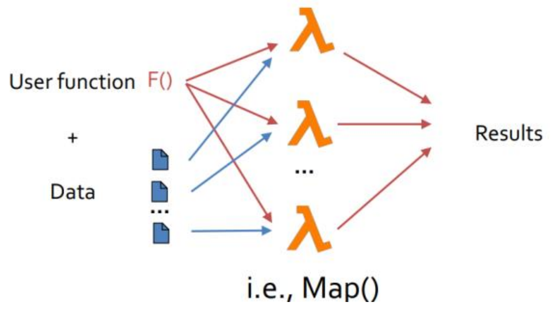
Scheduling challenge 1/3: Application Diversity

Increased Interest -> Application diversity

- Edge-Triggered applications:
    - Short-lived, lightweight
    - e.g. Web apps, backends, data preprocessing
- Massively Parallel applications:
    - Long running, computationally intensive
    - E.g. MapReduce, Stream Processing

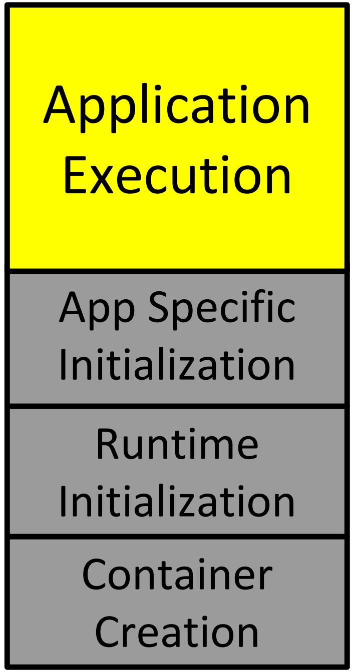






Scheduling challenge 2/3: Containers

- Serverless applications are hosted on containers
  - Absence of running container results in **Cold Start**
  - Cold-Start:
    - Application execution is delayed, e.g. ~3s in our setup
  - Should minimize the number of cold-starts

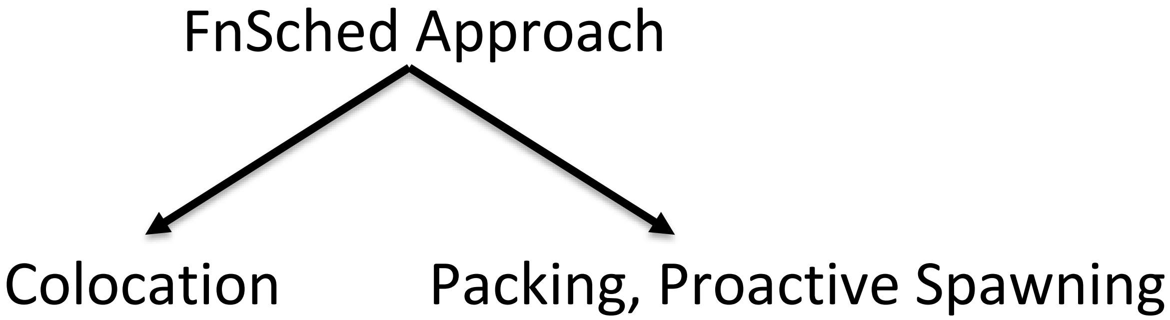


Scheduling challenge 3/3: Allocation & Placement

- Strawman: Allocate a core for each application
- However, provider cost will escalate!!
- Solution: **Effective packing**
  - Where to place a container?
  - Whether to colocate a container?
  - How long should the container be alive?
  - Whether to add new nodes?

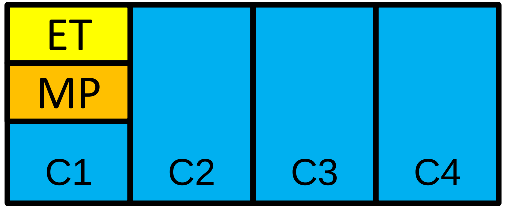
FnSched Approach

- **Goal:** Target a **maximum degradation latency** and **minimize the number of servers/invokers used**.



FnSched: Resource Management

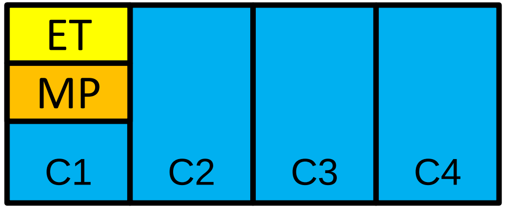
- Popular Serverless platforms tie CPU allocation to memory requirement
- CPU requirement is dependent on the class of applications



- Short running ET apps are severely impacted compared to MP apps
- We need to decouple memory and CPU requirement for effective colocation

FnSched: CPU Shares Algorithm

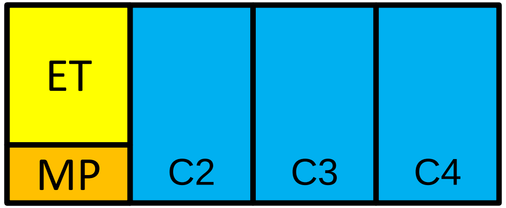
CPU Shares: Soft limit, decides proportion of CPU during contention



Allocate more of CPU time to short running ET during contention!

FnSched: CPU Shares Algorithm

CPU Shares: Soft limit, decides proportion of CPU during contention



- When to increase the cpushares?
- How much to increase?
- How to balance the cpu shares?

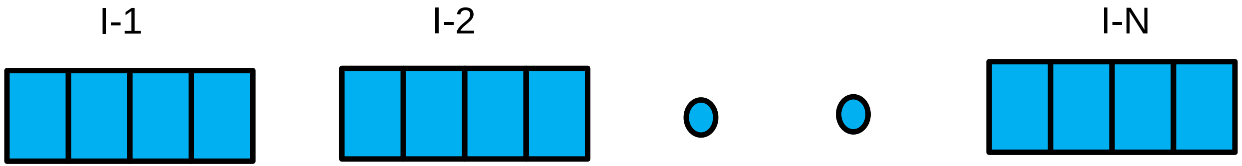
FnSched: CPU Shares Algorithm

```
numUpdates+=1;
latencyRatio = latency/isoLatency;
if latencyRatio > updateLatencyThd then
    if numUpdates > numUpdatesThd then
        if curShares < perContainerMax then
            toAddShares = cpuSharesStep * numConts;
            if (totShares+toAddShares) < maxCpuShares then
                curShares = curShares + cpuSharesStep ;
                totShares = totShares + toAddShares
            else
                toReduceShares =
                 (toAddShares/numOtherConts);
                rebalanceCpuShares(toReduceShares);
                deltaShares = (maxCpuShares - totShares) /
                 numConts ;
                curShares = curShares + deltaShares ;
                totShares = maxCpuShares
            end
        end
    end
end
```

- numUpdatesThd → When to increase the cpushares?
- cpuSharesStep → How much to increase?
- maxCpuShares → How to balance the cpu shares?

Multi Node Placement: Packing

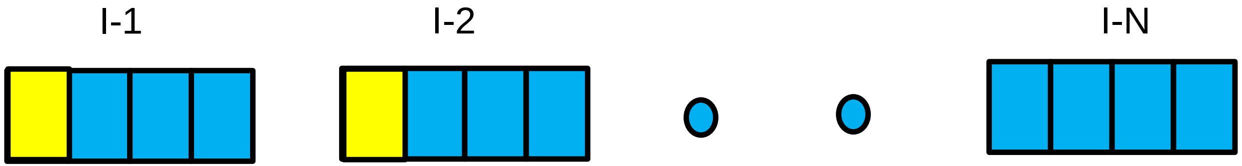
- Packaging: Greedy algorithm based on data center power management policy.



- Allocate request in the smallest index invoker
- Helps to packing requests in as few invokers as possible
- With effective packing, higher index invokers can be turned off

Multi Node Placement: Proactive Spawning

- Packaging: Greedy algorithm based on data center power management policy.



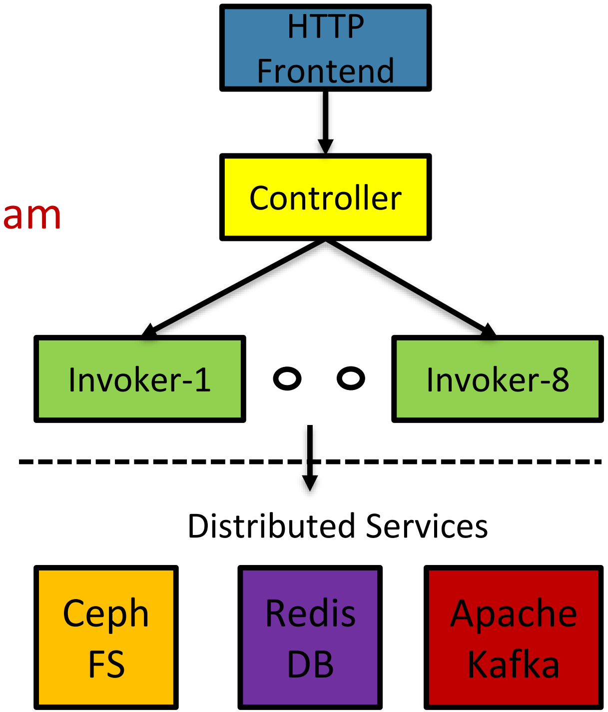
- **Cold Starts:** Scheduling on ***invoker k*** is followed by **proactively** spawning an application container on ***invoker k+1***

Outline

- Motivation
- Scheduling Challenges
- FnSched Design
- Evaluation
- Conclusion & Future work

Experimental Setup

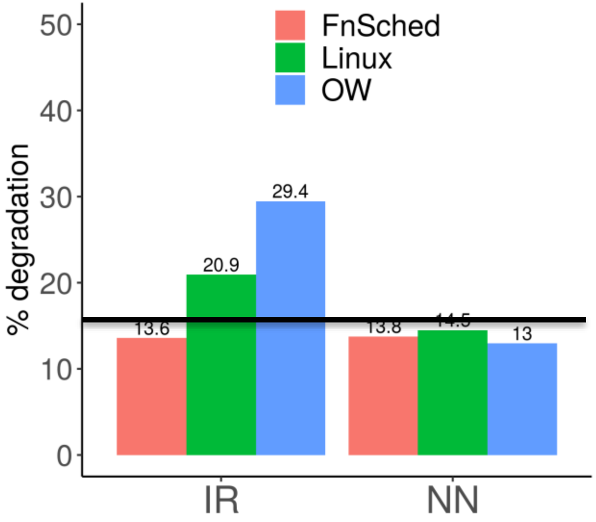
- OpenWhisk Cluster: 10 VMs
  - Front-end + control plane: 2 VMs
  - Invokers: 8 VMs
- Distributed services: Storage: CephFS, Database: Redis, Stream Processing: Apache Kafka
- Applications:
  - **Edge-Triggered:**
    - Image Resizing (**IR**),
    - Streaming Analytics (**SA**)
  - **Massively Parallel:**
    - Nearest Neighbors (**NN**)
    - Computational Fluid Dynamic (**CFD**) solver
- ***latencyThd:*** **1.15** i.e. maximum of 15% performance degradation

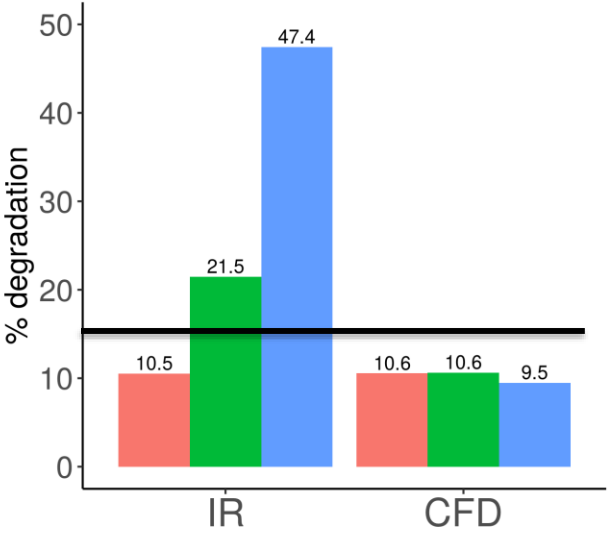


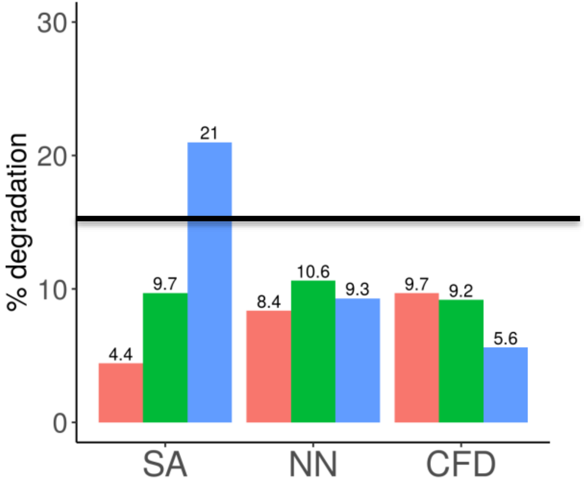
Single Node Evaluation

- **FnSched:** Single node resource allocation
- **Linux:** CPU shares 1024
- **OpenWhisk:** CPU shares proportional to memory

Can safely co-locate



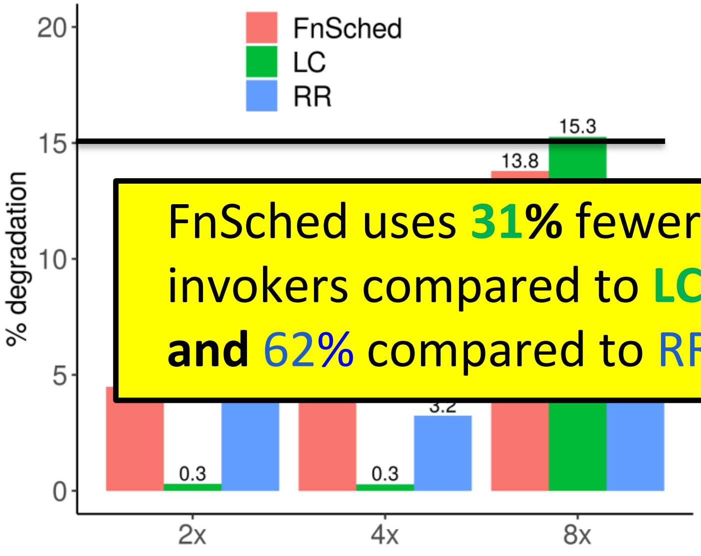




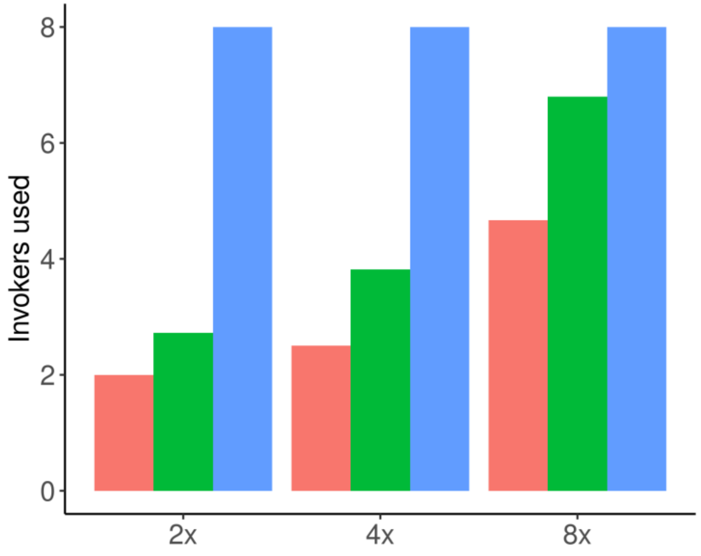
Multinode Evaluation: Scaling

- **FnSched:** Single node resource allocation
- **LeastConnections (LC):** Choose the invoker with least o
- **RoundRobin (RR):** Send successive requests to different … r

Packing can scale and maintain performance



FnSched uses **31%** fewer invokers compared to **LC**, **and** 62% compared to RR



Multi Node Evaluation: Traces

Load:

- IR
- NN

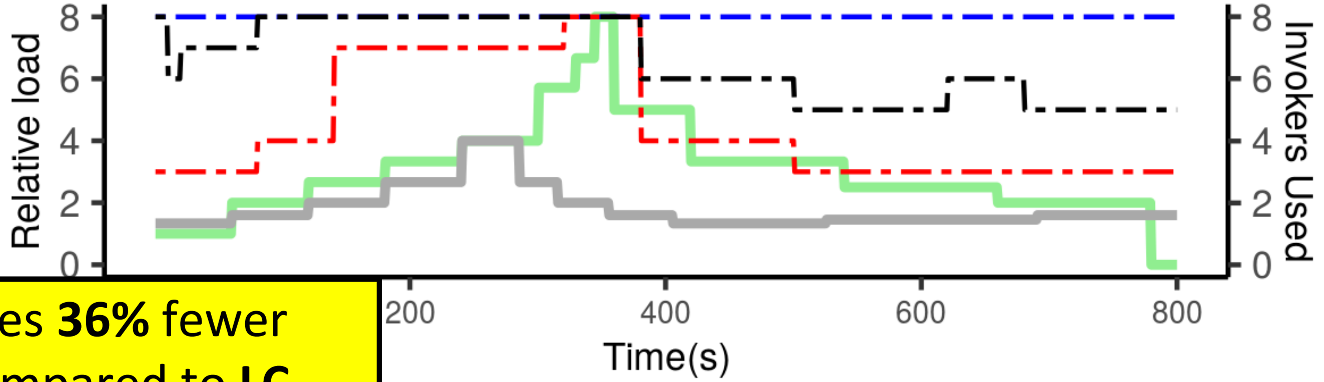
Scheduling:

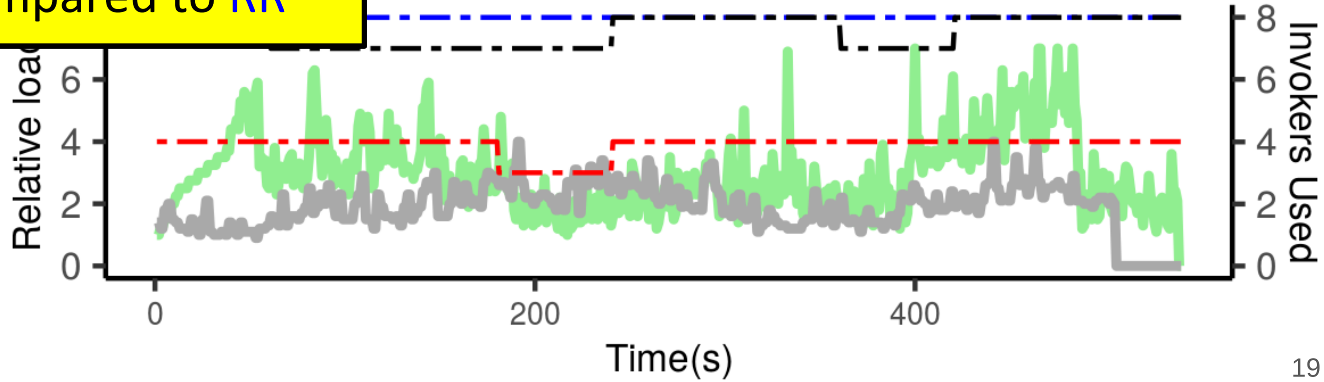
- FnSc
- Leas
- Roun

Traces:

- Synthetic (top)
- WITS (bottom)

FnSched uses **36%** fewer invokers compared to **LC,** **and** 55% compared to RR





Conclusion

- Presented a work-in-progress serverless scheduling algorithm based on **colocation + packing**
- CPU Shares algorithm: Reduces degradation compared to SoA
- Packing + Proactive Spawning: Maintains acceptable performance,
  - While reducing invoker usage by 36% compared to LC, 55% compared to RR

Q&A

Backup Slides

Future Work

- Proactive Spawning : Figure out ~***exact*** number of containers required
- Evaluation: Scenarios where colocation opportunities are fewer
  - Multiple ET applications
  - ET:MP ratio is > 1
- Compare against Knative

FnSched Approach

- **Goal:** Target a **maximum degradation latency** and **minimize the number of servers/invokers used**.

Challenges	FnSched Approach
Application Diversity/ Resource management	Application class based colocation, resource management
Cold-Start	Proactive Spawning
Allocation & Placement	Packing based on data center power management policy

Sensitivity Analysis

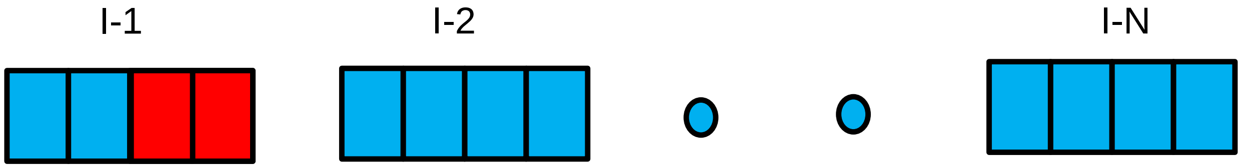
Choose parameters for single node resource allocation algorithm. Parameters vary for application class

- *numUpdatesThd:* Minimum iterations required before updating cpu-shares
- *maxCpuShares:* Ceiling of the cpu-shares per container, maximum of 1024
- cpuSharesStep: Per iteration increment of cpu-shares
- *updateLatencyThd:* Minimum degradation before updating cpu-shares **1.10**

Appln Class	***numUpdatesThd***	***maxCpuShares***	***cpuSharesStep***
ET	5	768	128
MP	3	256	64

Multi Node Placement: Latency monitoring

- Packaging: Greedy algorithm based on data center power management policy.



- Monitor average latency
- Based on threshold latency, mark invoker to be in ***safe, warning, unsafe*** zone
- Capacity of invoker varies by the zone