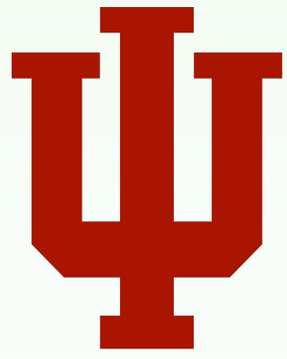
Keeping it Local: Functions as a Service Resource Management

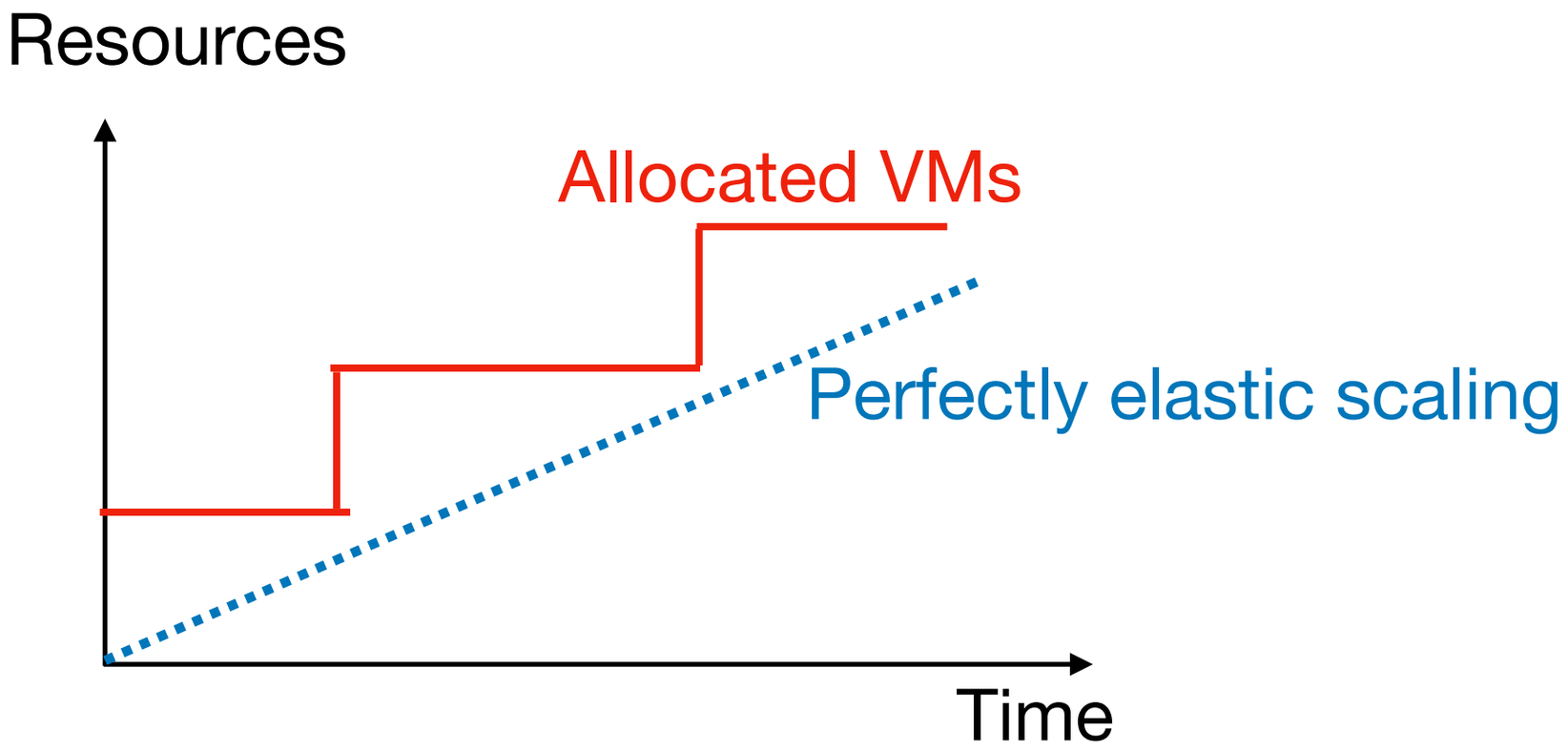
Prateek Sharma
Indiana University Bloomington



Infrastructure as a Service Pitfalls

Infrastructure is now software, but even that is too hard

- Developers configure and manage complete software stack
  - OS and software upgrades, security patches
  - Monitoring and logging
  - Auto-scaling, redundancy, geo-replication



Functions as a Service

Many cloud applications are event-driven

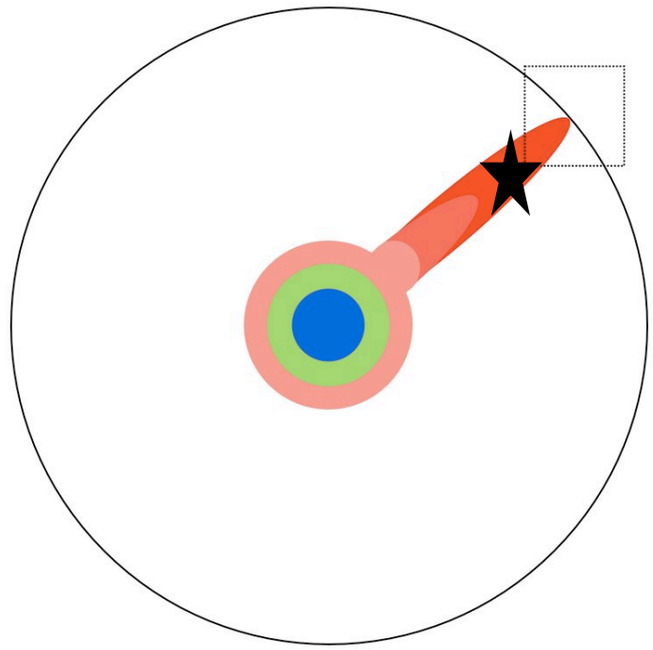
- Events: HTTP requests, storage updates, publish-subscribe
- Event handlers:
  - Web servers: generate html response
  - Machine learning: model inference on input passed via HTTP

FaaS characteristics:

- Developers provide event-handling code as function source-code
- Functions are ‘pure functions’ and stateless
  - All state is stored in cloud storage (S3, etc)
- Each function invocation runs in a sandbox (container/VM)
  - Typically small resource limits (1 cpu core) and duration < 30 minutes
- Pricing: linear “scale-to-zero”. Per-invocation ($ 10e-7)

Overview and Themes

1. Functions as a Service: benefits for users and challenges for providers
2. **Temporal locality** to reduce cold-start overheads
3. Load-balancing
4. Queueing for GPU functions
5. Polymorphic function dispatch



Themes:

1. Look at resource management problems at various scales (server, cluster level ; CPU, GPU, mixed,..)
2. Find similar problems in broader systems, and specialize and apply classic solutions
3. Many interesting open problems “local” to your current knowledge and expertise (the frontier is closer than you think)

FaaS Workflow From User Perspective

Source code uploaded to provider

```
#Initialization code
import numpy as np
import tensorflow as tf

m = download_model('http://model_serve/
img_classify.pb')
session = create_tensorflow graph(m)

def lambda_handler(event):
    #This is called on every function invocation
    picture = event['data']
    prediction_output =
run_inference_on_image(picture)
    return prediction_output
```

HTTP-trigger

PUT https://faas.com/img_recogn?input=photo.png

Provider returns code results

GCloud Functions Demo

```
gcloud functions deploy hw --gen2 --runtime=python312
--region=us-east1 --entry-point=hello_http --trigger-
http --allow-unauthenticated -source=hw
```

Execute the function with https://us-east1-first-220321.cloudfunctions.net/hw/?name=fooo

Hw directory contains main.py

- Some restrictions of the FaaS programming model:
  - No local state (i.e., cannot use local variables, file-system to maintain status, counters, etc.)
  - No direct networking — all communication via cloud storage

FaaS Provider Perspective

- API consists of 2 main end-points:
  - **Register** a function. Bind function name with event-trigger and function-code. Results in creation of container sandbox
  - **Invoke** a function. When triggered (e.g., HTTP), run function code in its sandbox on some server

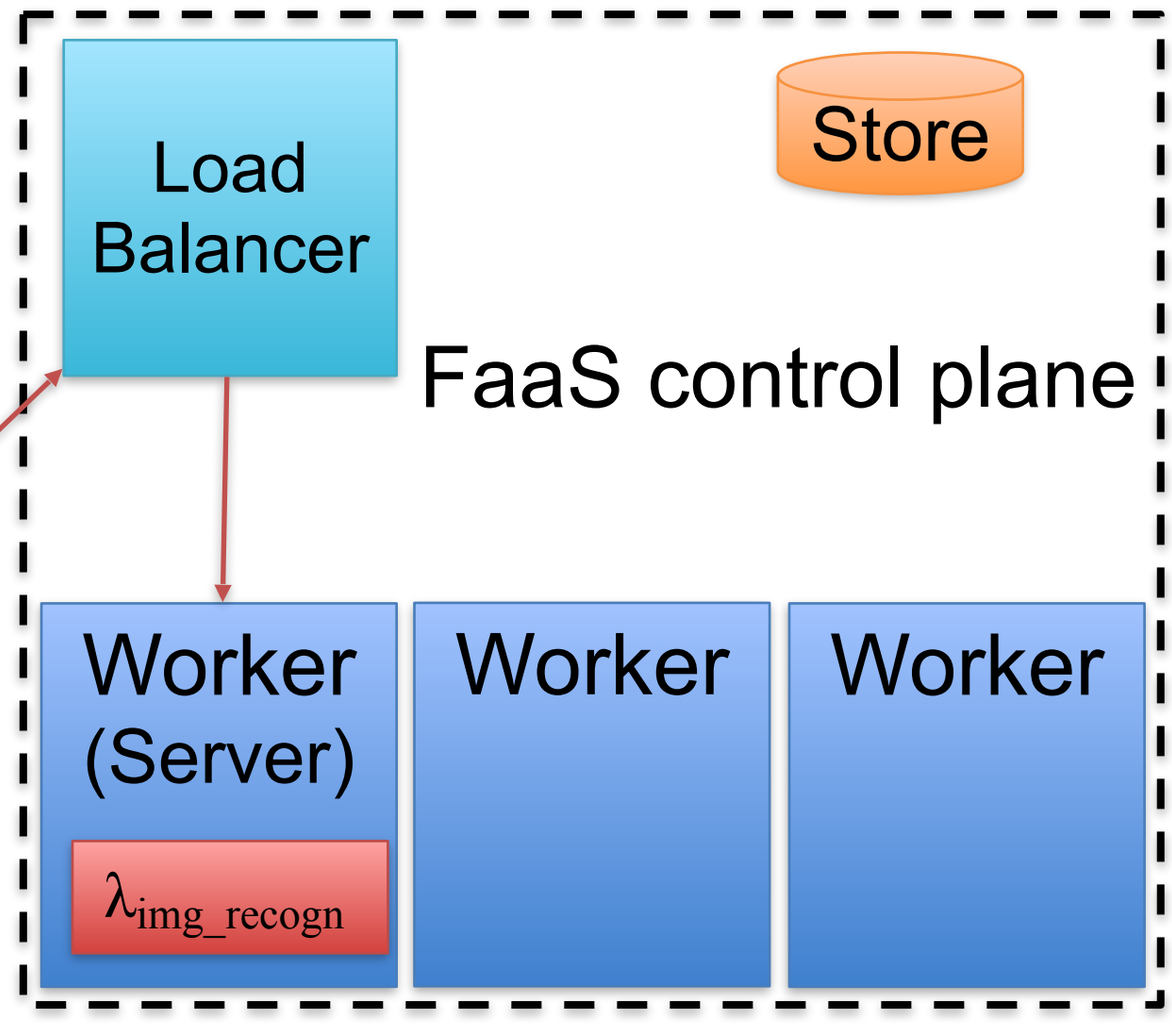
```
#Initialization code
import numpy as np
import tensorflow as tf

m = download_model('http://model_serve/img_classify.pb')
session = create_tensorflow graph(m)

def lambda_handler(event):
    #This is called on every function invocation
    picture = event['data']
    prediction_output = run_inference_on_image(picture)
    return prediction_output
```

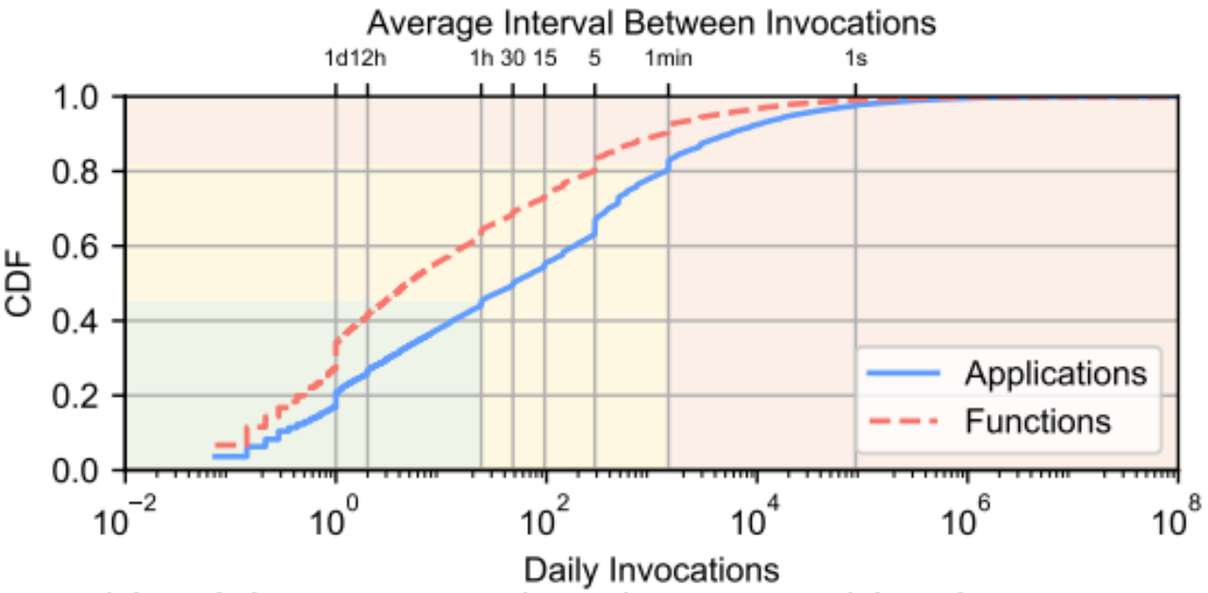
HTTP-based invocation, with inputs

PUT https://faas.com/img_recogn?input=face.png

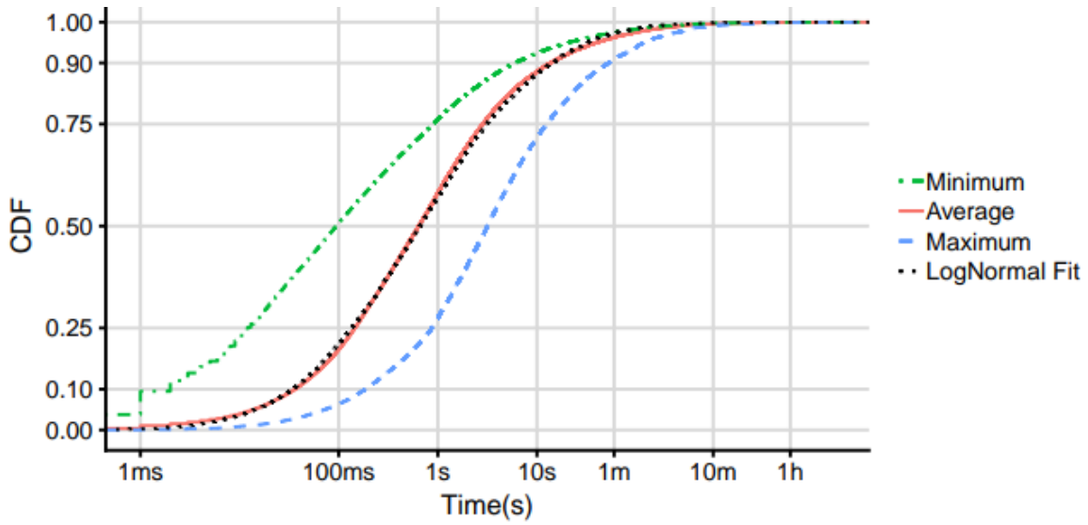


FaaS Workloads

- FaaS is a common abstraction supporting a wide range of applications
- A workload is composed of a mix of many functions, each with different:
  - Inter-arrival-time distribution (popularity): milliseconds to days
  - Container size (cpu cores, memory): 0.1 to 10 GB
  - Execution time: milliseconds to minutes
- Extremely skewed and heavy-tailed



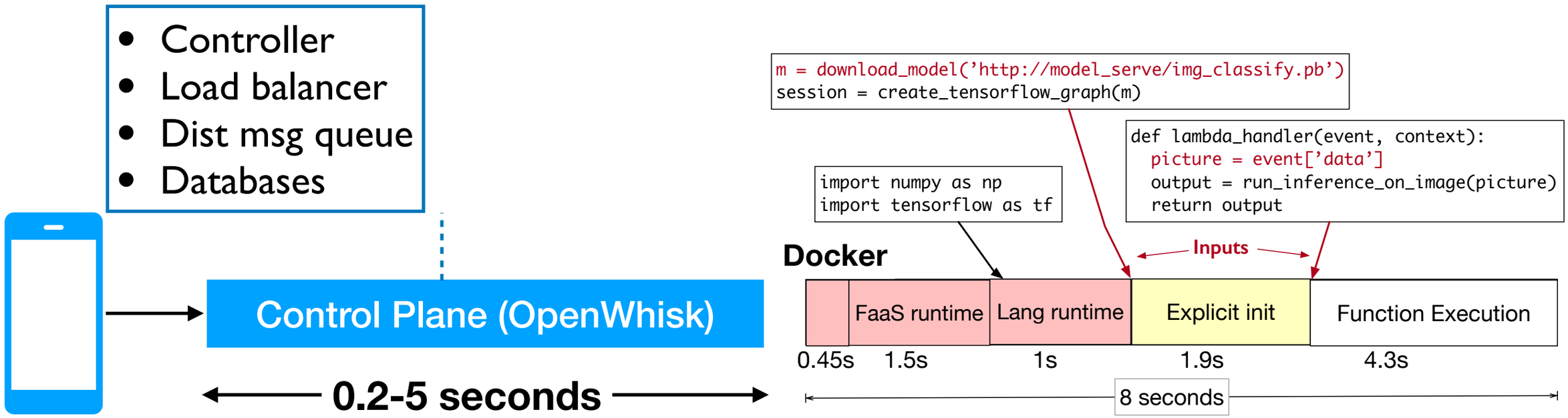
20% of functions (red) are 99% of invocations



Execution-duration

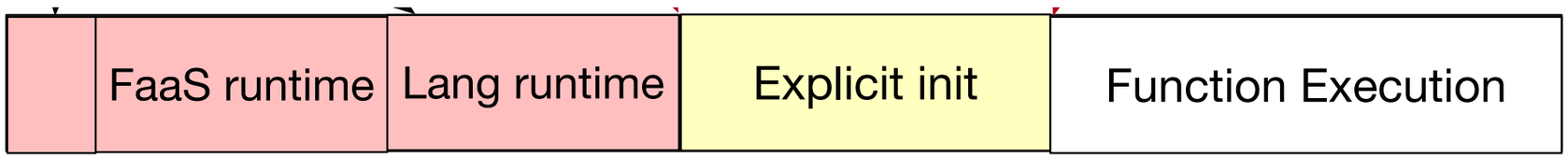
Function Latency

- Function latencies can be significantly higher than conventional client-server architectures
- In figure below, the main function execution is 4.3 seconds, but the end-to-end latency can be 5 + 8 seconds.



Function Keep-Alive

- Each function invocation must run in a sandboxed environment
  - Create and setup new container/VM for each invocation



Application	Mem size	Run time	Init. time
ML Inference (CNN)	512 MB	6.5 s	4.5 s
Video Encoding	500 MB	56 s	3 s
Matrix Multiply	256 MB	2.5 s	2.2 s
Disk-bench (dd)	256 MB	2.2 s	1.8 s
Web-serving	64 MB	2.4 s	2 s
Floating Point	128 MB	2 s	1.7 s

[*]

Keep-alive: Keep container in memory.

Subsequent invocations don't incur cold-start.

Keep-Alive Tradeoffs

- Keeping containers alive in memory can reduce latency by ~10x
- But increases server memory requirements

Keep-alive policy:

Which function containers to keep in memory, and for how long?

- Many tradeoffs:
  - Frequently invoked functions?
  - Small memory footprint functions?
  - Largest (cold - warm) time?

FaaS Keep-alive === Caching

- **Cold-start -> Cache Miss**
- **Warm-start -> Cache hit**
- **Keep-alive policy -> Cache eviction**

- Cache eviction policies, analysis, etc can be used for FaaS resource management!

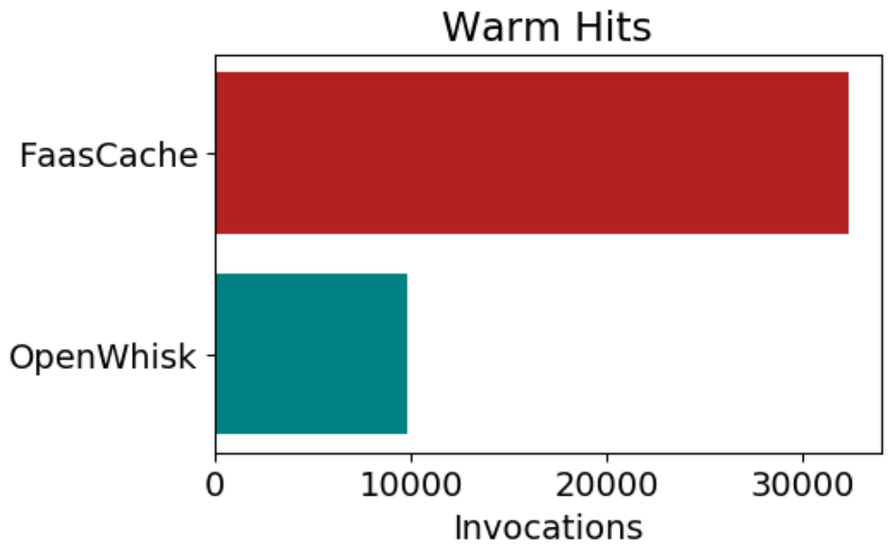
- **Current FaaS keep-alive: fixed Time-to-Live (TTL)**
  - **OpenWhisk: 10 minutes TTL until eviction**

Greedy-Dual Keep-Alive

- For functions, memory size is highly variable and crucial
- Other parameters: frequency, recency, cold-warm time
- Most caching is size-oblivious (LRU)
- Size-aware cache eviction: Greedy-Dual-Size-Frequency

$$KeepAlivePriority = Recency + \frac{Frequency * InitTime}{Memory}$$

FaasCache:
Our Greedy-Dual implementation

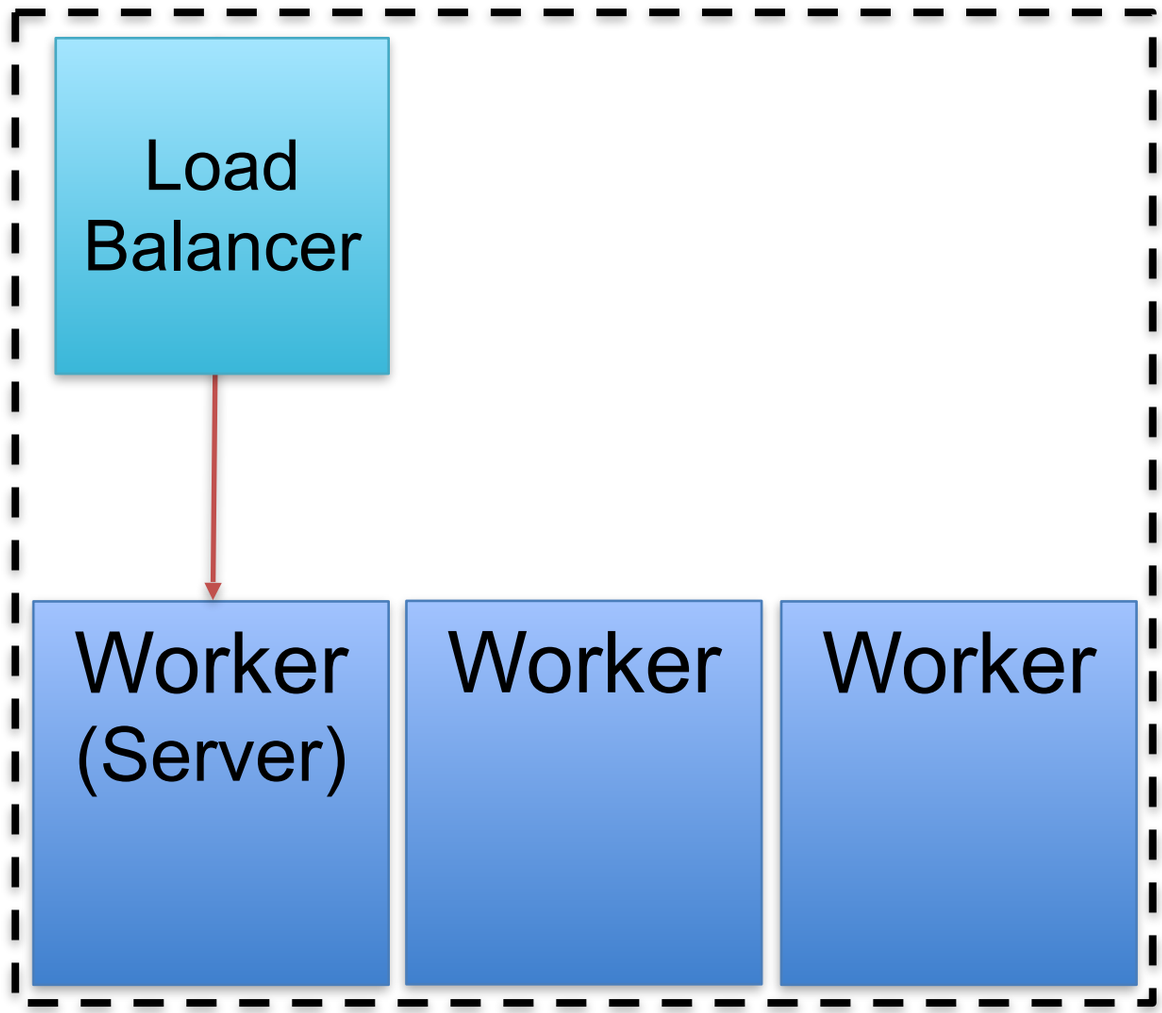


Better warm-hit performance

FaasCache: Keeping Serverless Computing Alive With Greedy-Dual Caching
ASPLOS 2021. A. Fuerst and P. Sharma

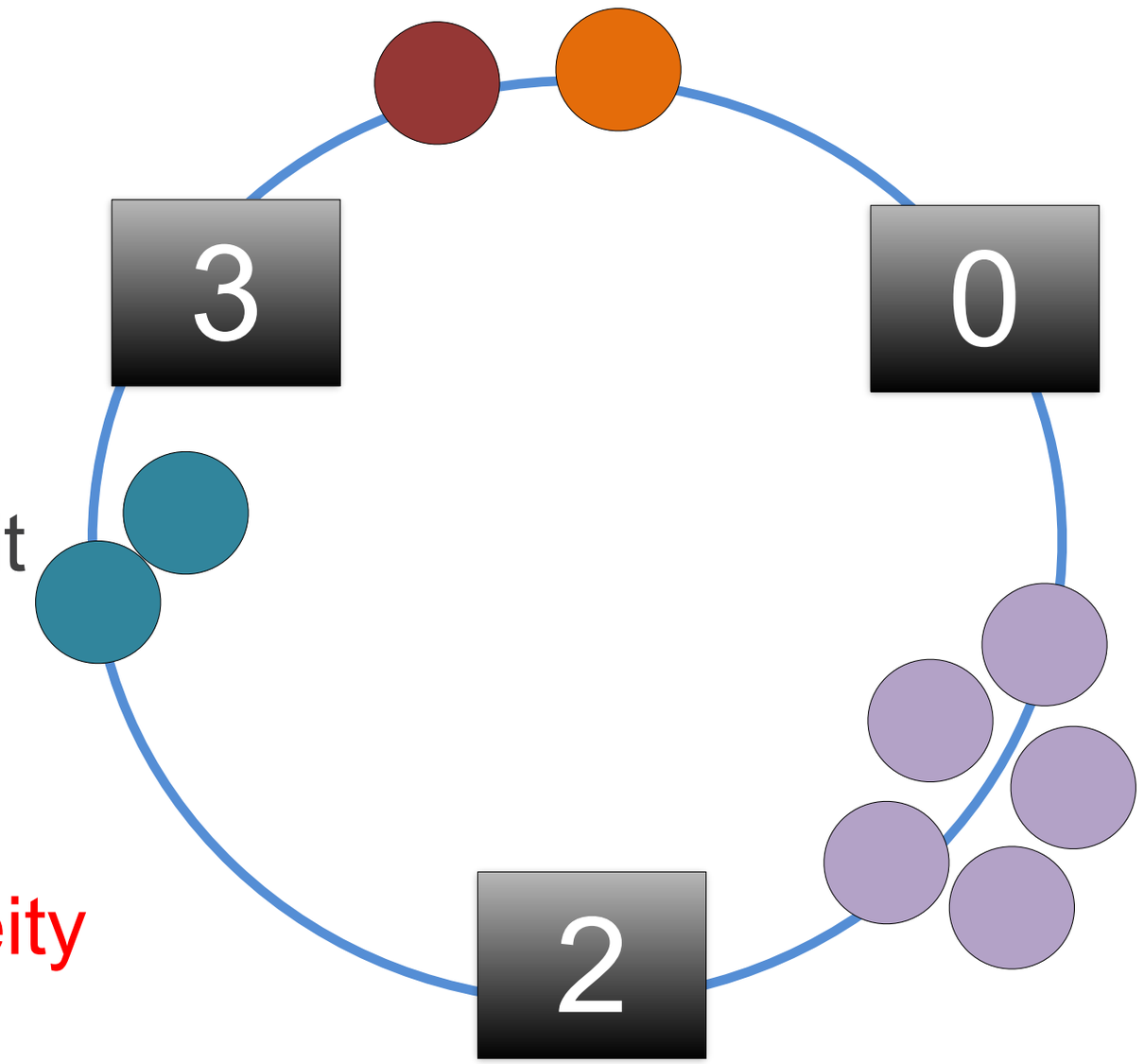
FaaS Load-balancing Challenges

- **Consider both server-load and locality (i.e., “sticky”)**
- Improve cluster utilization and function latency
- Support horizontal scaling (change number of servers to meet traffic needs)



Consistent Hashing with Bounded Loads

1. If server is “full” we forward to the next server
2. When under the load bound: pure locality
3. Forwarding has a high but decaying probability of warm hit



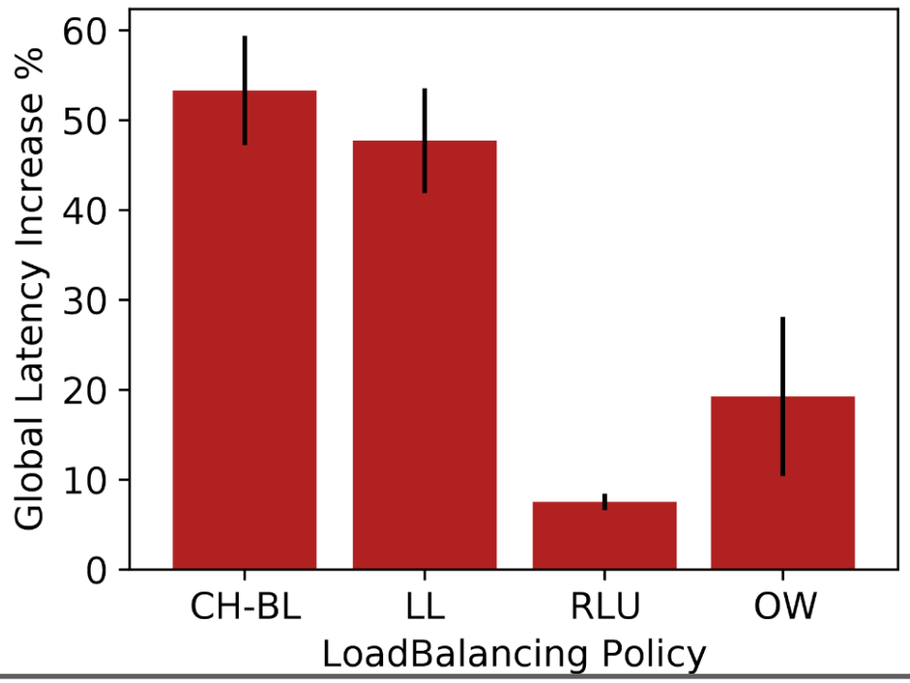
Challenges: Function heterogeneity and stale cluster information

Load bound = 3
(max number of running functions per server)

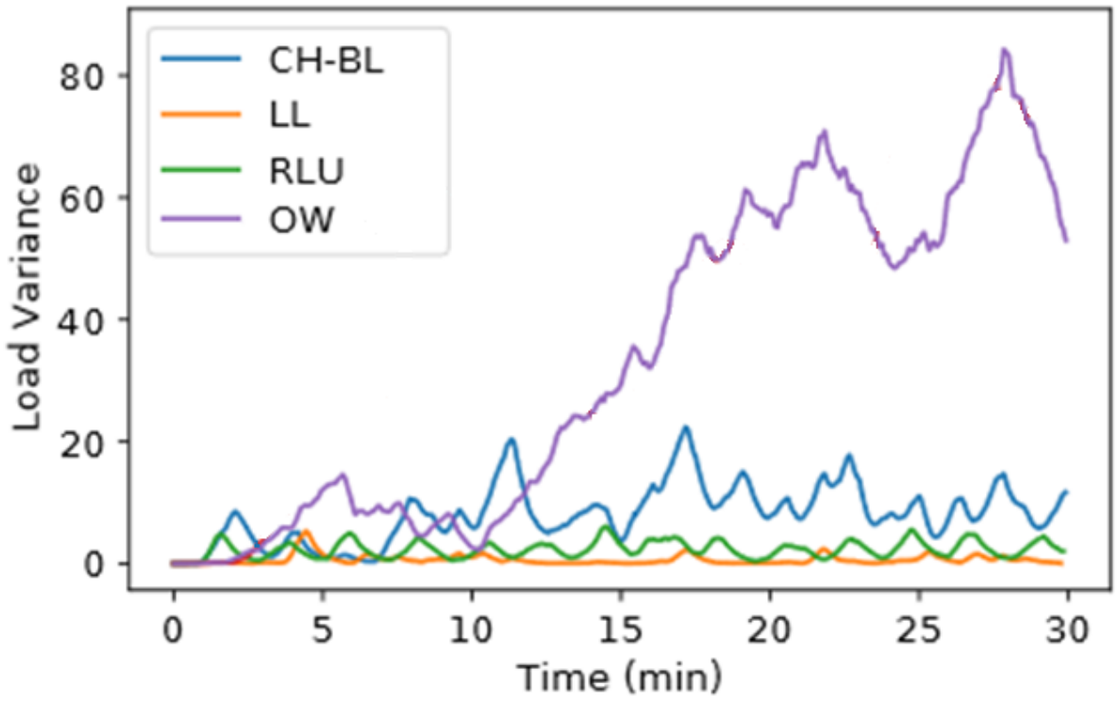
Our approach: CH-RLU
Consistent Hashing with Random Load Updates

- Key refinements to CH-BL:
  - Use system-load information
    - Can be stale/inconsistent
      - Estimate “load in flight” and add this random load to observed (stale) load as a “buffer”
  - Separate handling of highly popular, “bursty” functions
    - Use SHARDS from caching for sampling-based popular function detection
- More aggressive forwarding of bursty functions to prevent overloaded servers

CH-RLU yields good locality, load-balancing, and function latency



- **Increase in function latency compared to best-case latency (warm start under no load)**
- **CH-RLU: < 5% increase, vs. OpenWhisk's 20%**



- **Variance in server loads over time**
- **LL: Least loaded. Not locality aware, but lowest load variance**
- **OpenWhisk: Not load aware, so can result in some servers getting overloaded while others are idle**

Locality-aware Load-Balancing For Serverless Clusters. HPDC 2022, A. Fuerst and P. Sharma

General FaaS resource management principles

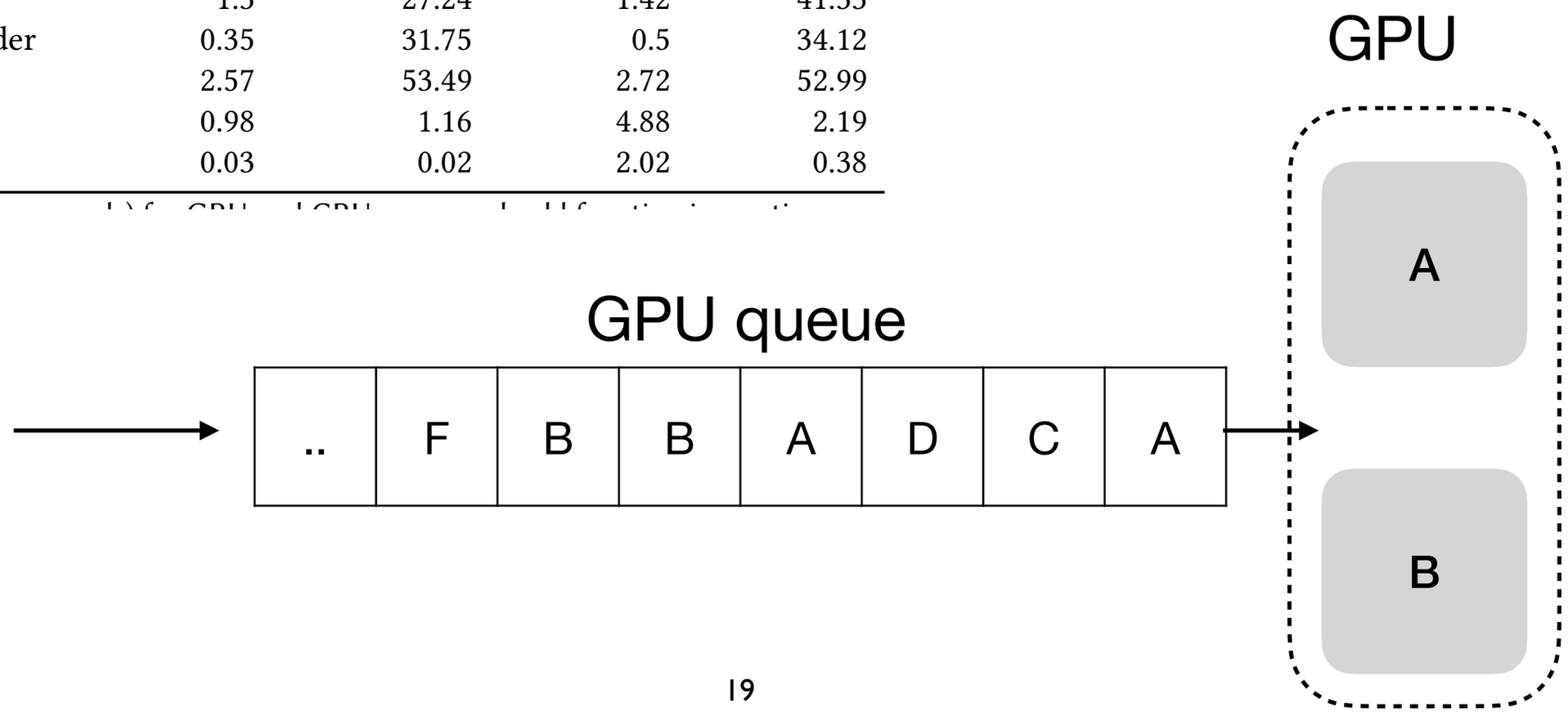
Recap:

- 2 main challenges: heterogeneity and locality (cold-time >> warm)
- Main opportunity: can predict/estimate function characteristics using historical information (arrival rate and cold/warm times)
- Workload: treat as a mixture of individual functions with different characteristics
- Invocation latency: execution (i.e., service) time + queueing time
  - Both depend on server-load and overcommitment-level

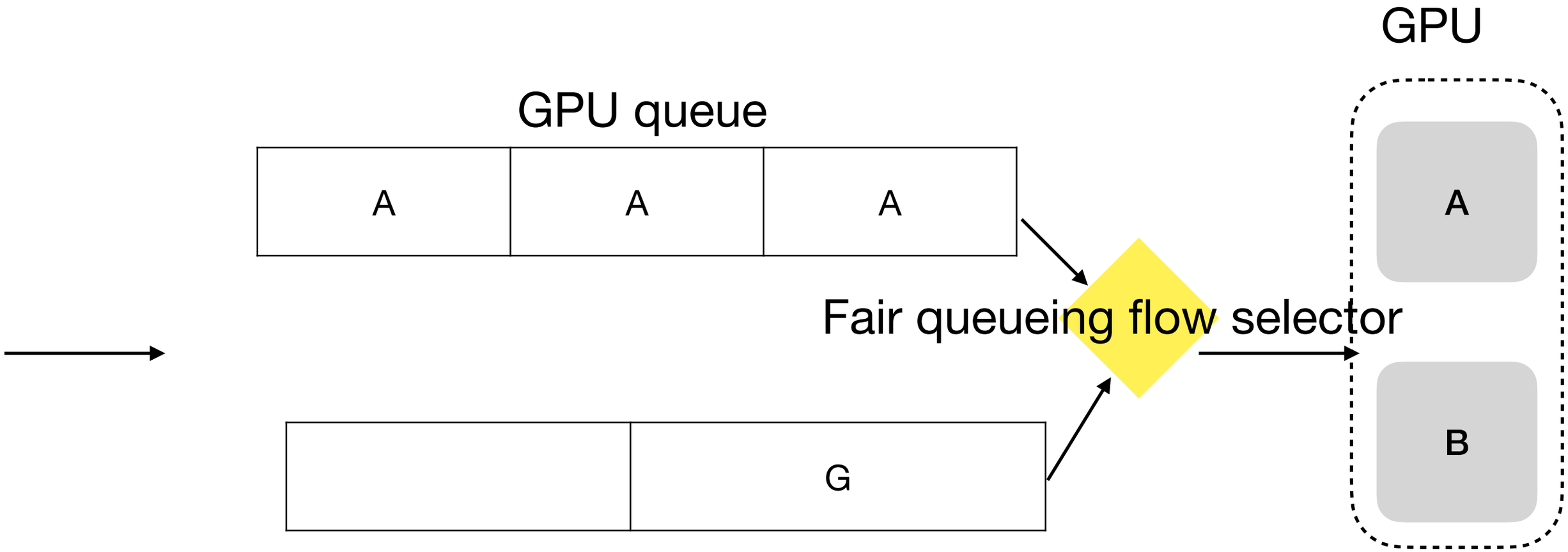
GPU Functions

- GPUs: Very limited concurrent execution capability (2—5)
- GPU containers: high initialization/cold-start overheads

Function	GPU [Warm]	CPU [Warm]	GPU [Cold]	CPU [Cold]
cupy	0.89	11.76	2.12	12.69
imagenet	1.93	4.77	6.71	4.5
onnx-roberta	0.18	4.62	1.08	0.89
pyhpc-eos	0.01	0.04	3.53	0.05
pyhpc-isoneural	0.02	0.52	7.93	0.54
rodinia-lavamd	0.6	17.75	0.75	15.0
rodinia-lud	0.74	22.24	0.93	55.21
rodinia-myocyte	1.67	40.49	2.97	41.21
rodinia-needle	1.3	27.24	1.42	41.33
rodinia-pathfinder	0.35	31.75	0.5	34.12
rodinia-srad	2.57	53.49	2.72	52.99
squeezenet	0.98	1.16	4.88	2.19
torch-rnn	0.03	0.02	2.02	0.38

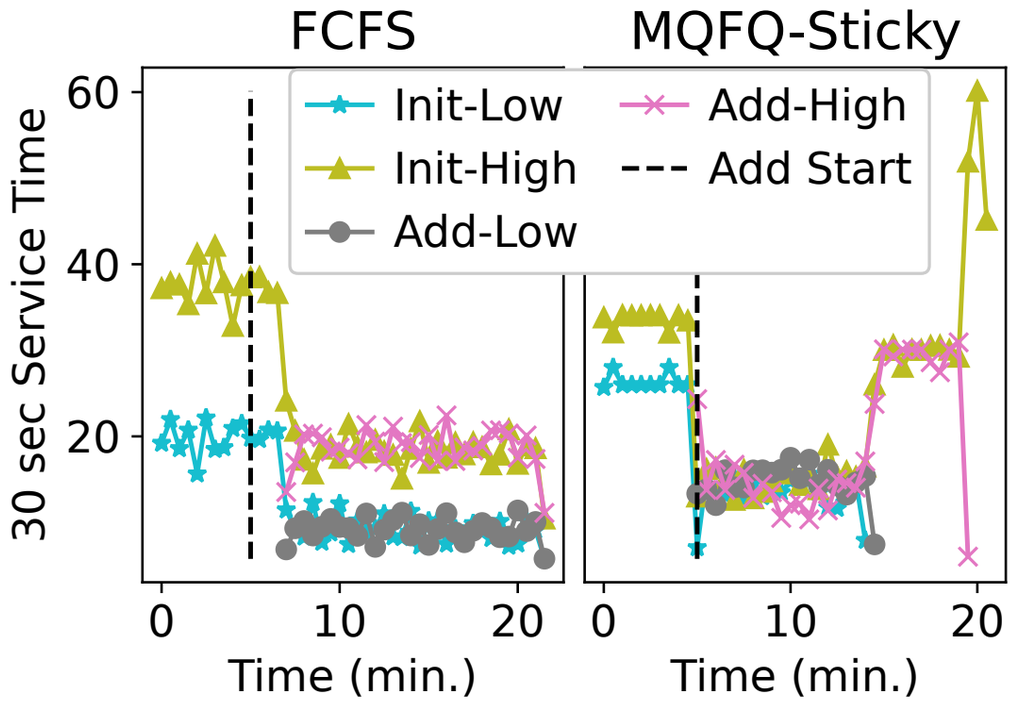


GPU Queueing: Balancing Locality and Waiting

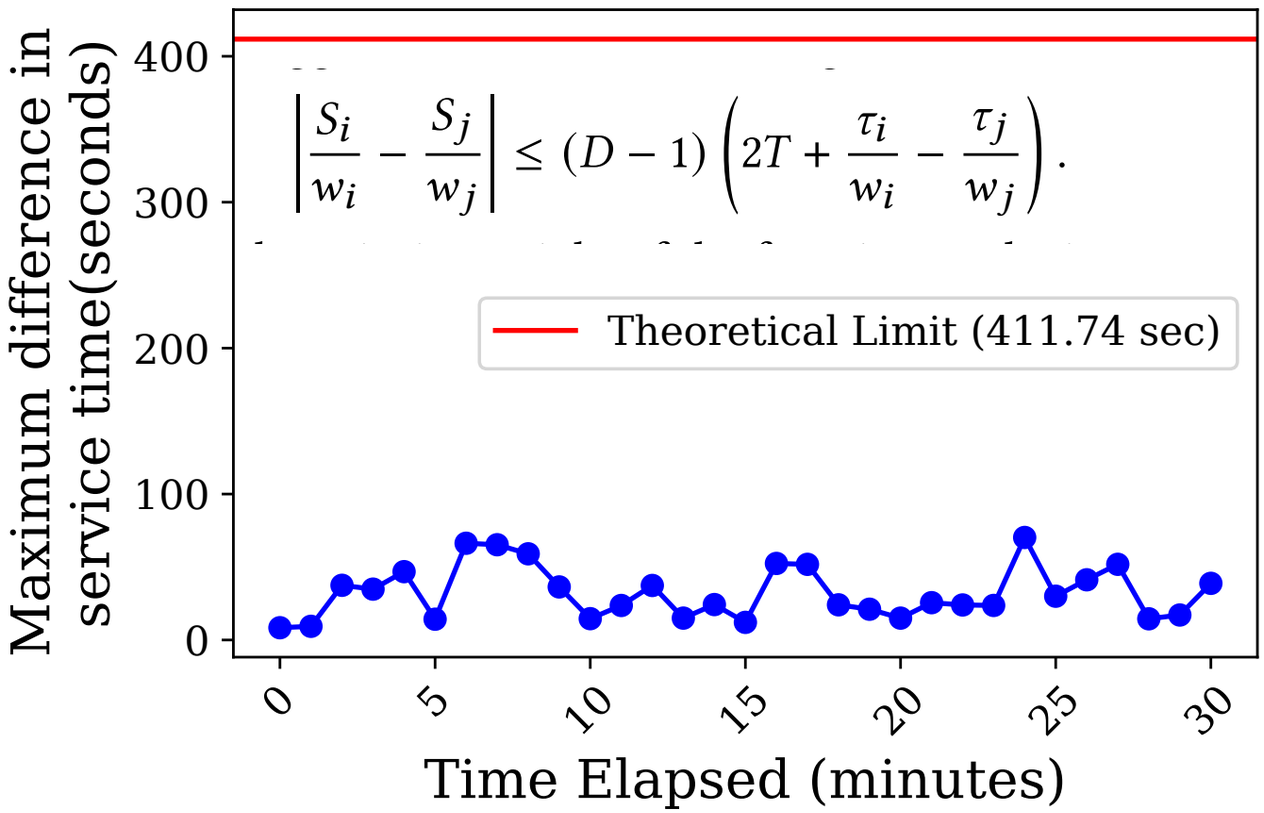


- Multi-Queue Fair Queueing (MQFQ): each function represents a flow
  - Originally from disk scheduling: preserve locality for different application request streams
- Select next flow to dispatch based on function arrival and service time
- Flow run-ahead for locality: looser fairness bound

MQFQ Fairness

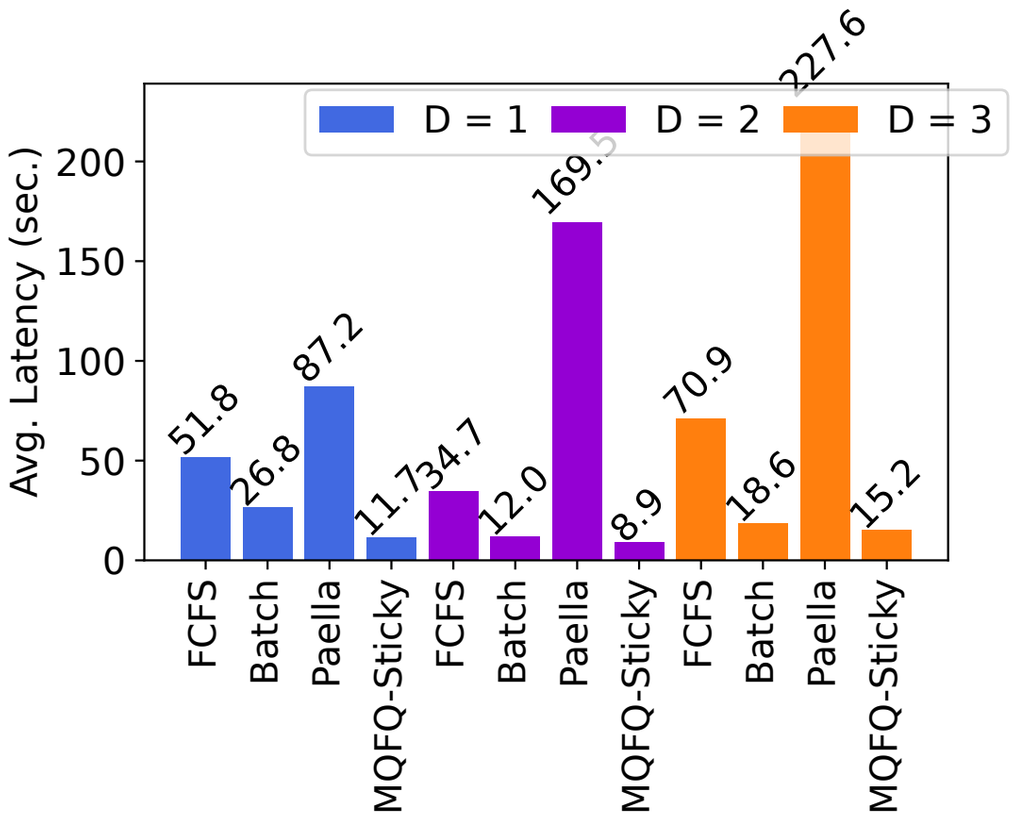


(a) GPU service time as functions are added to the workload at the 5 minute mark. MQFQ is fair, and provides all functions similar service, unlike popular functions dominating with FCFS.

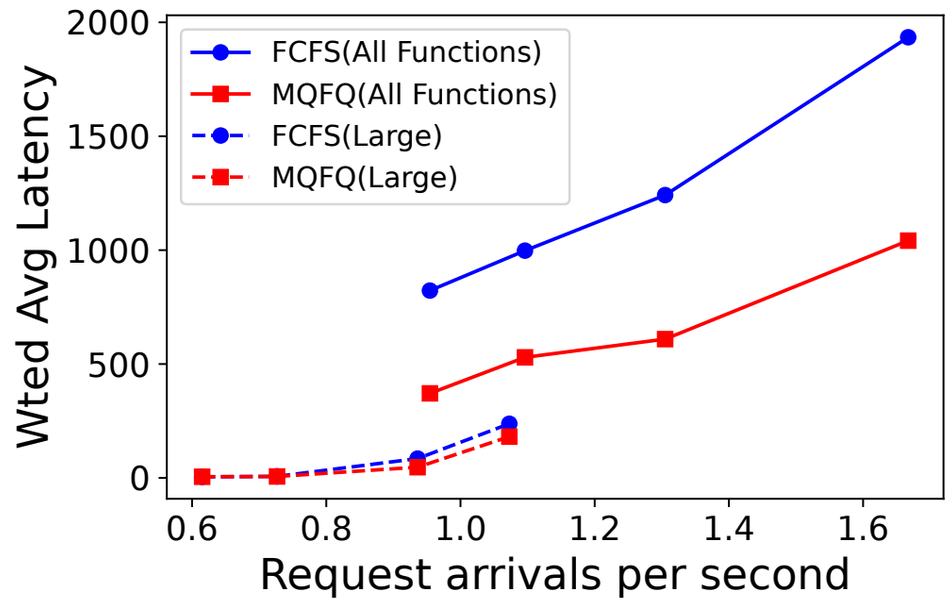


(b) The maximum difference in GPU execution time among all functions is significantly smaller than the theoretical upper-bound.

MQFQ Performance



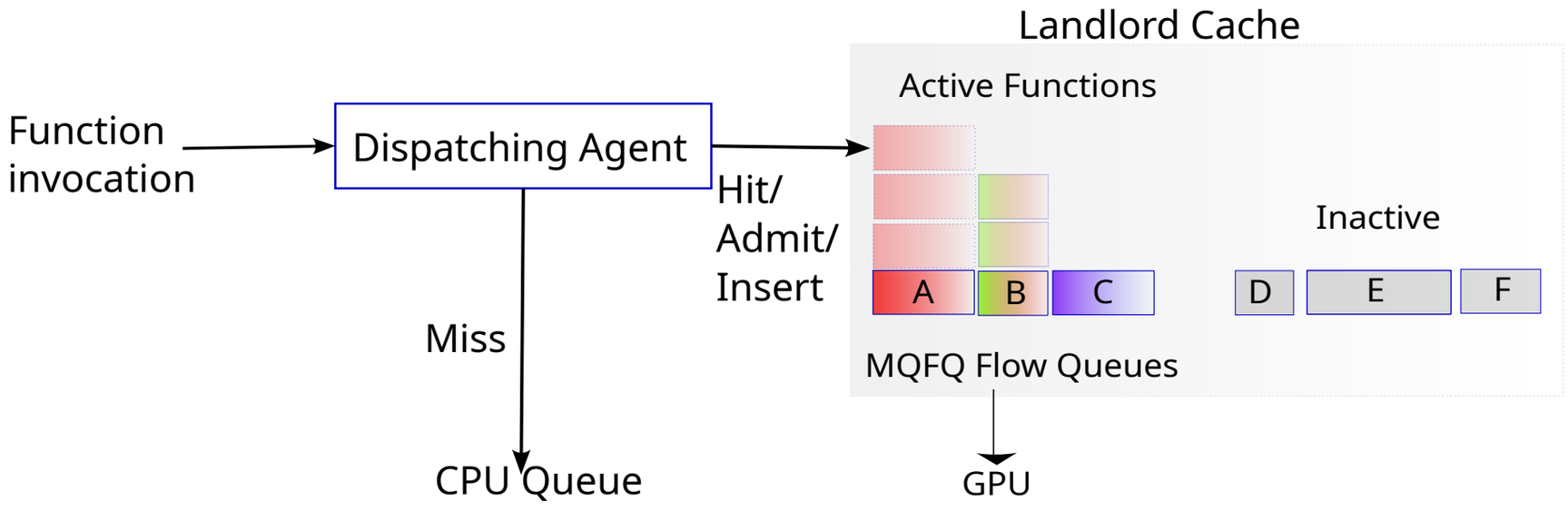
(a) Average latency is $2-5\times$ lower with MQFQ-Sticky for different device-parallelism (D) levels.



- 5x lower latency compared to FCFS (due to cold-start reduction)
- Open question: MQFQ performance modeling
  - Latency depends on flow-queue state (number of functions of different types), interference, etc.
  - Offline optimal queueing policy to minimize expected latency?

Polymorphic Function Dispatch

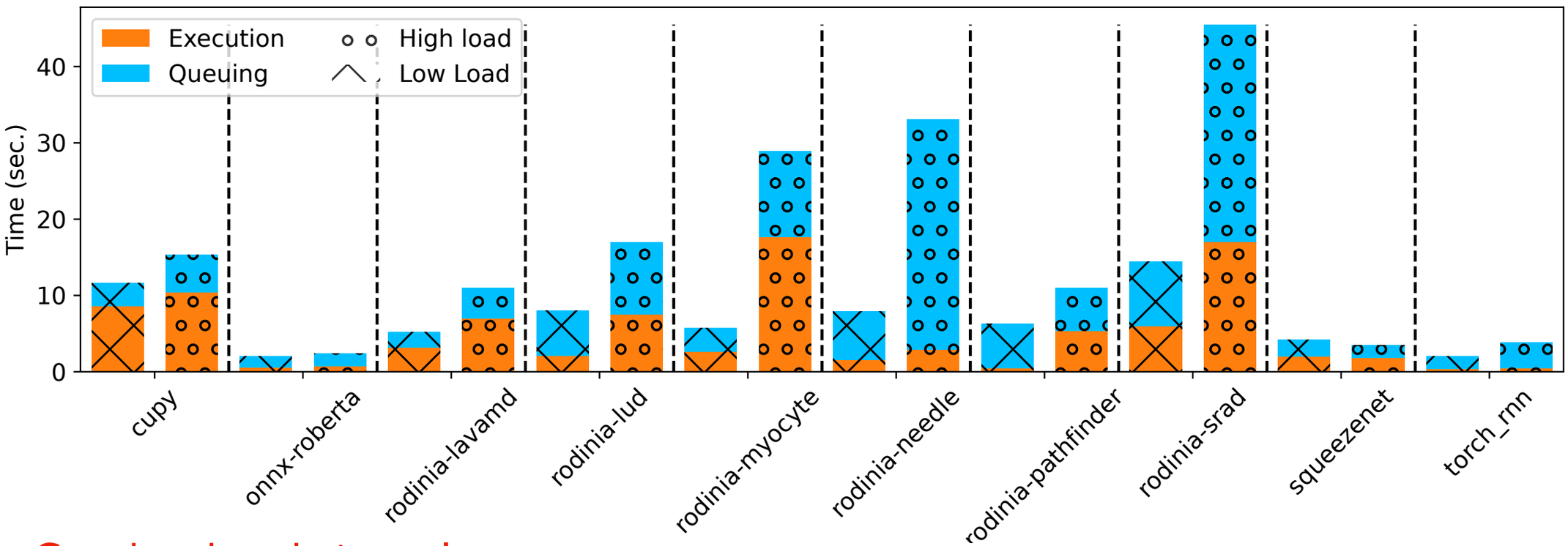
- GPUs are highly congested resource
- **Can we offload some invocations to the CPU?**
- Many functions (ML inference etc) can be made polymorphic
  - Select CPU or GPU container at run-time
- Tradeoffs: some functions see higher GPU speedup; locality; queue wait times
- Can we treat the GPU as a smaller ‘cache’ for functions?



GPU Function Performance Under Load

Function	GPU [Warm]	CPU [Warm]	GPU [Cold]	CPU [Cold]
cupy	0.89	11.76	2.12	12.69
imagenet	1.93	4.77	6.71	4.5
onnx-roberta	0.18	4.62	1.08	0.89
pyhpc-eos	0.01	0.04	3.53	0.05
pyhpc-isoneural	0.02	0.52	7.93	0.54
rodinia-lavamd	0.6	17.75	0.75	15.0
rodinia-lud	0.74	22.24	0.93	55.21
rodinia-myocyte	1.67	40.49	2.97	41.21
rodinia-needle	1.3	27.24	1.42	41.33
rodinia-pathfinder	0.35	31.75	0.5	34.12
rodinia-srad	2.57	53.49	2.72	52.99
squeezenet	0.98	1.16	4.88	2.19
torch-rnn	0.03	0.02	2.02	0.38

- Significant queueing delays
- Execution time also increases



Cupy benchmark time: <1 s

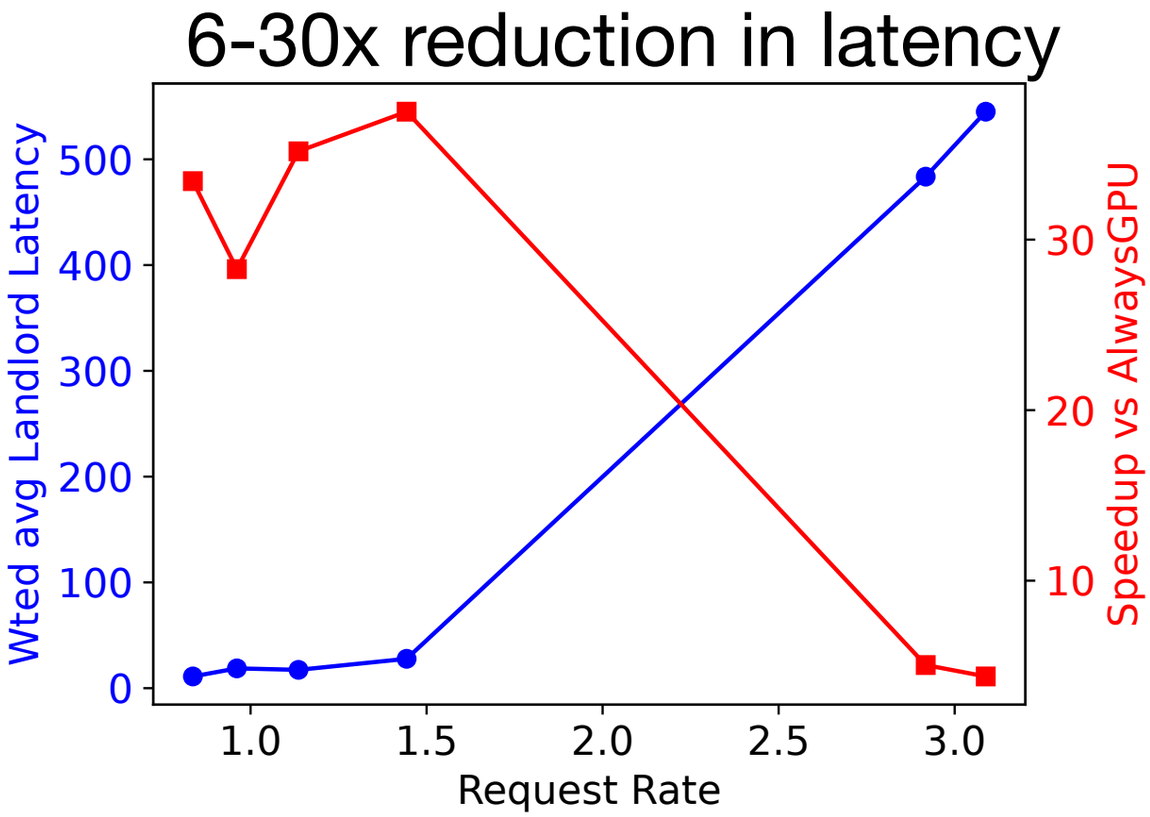
GPULandlord Dispatch

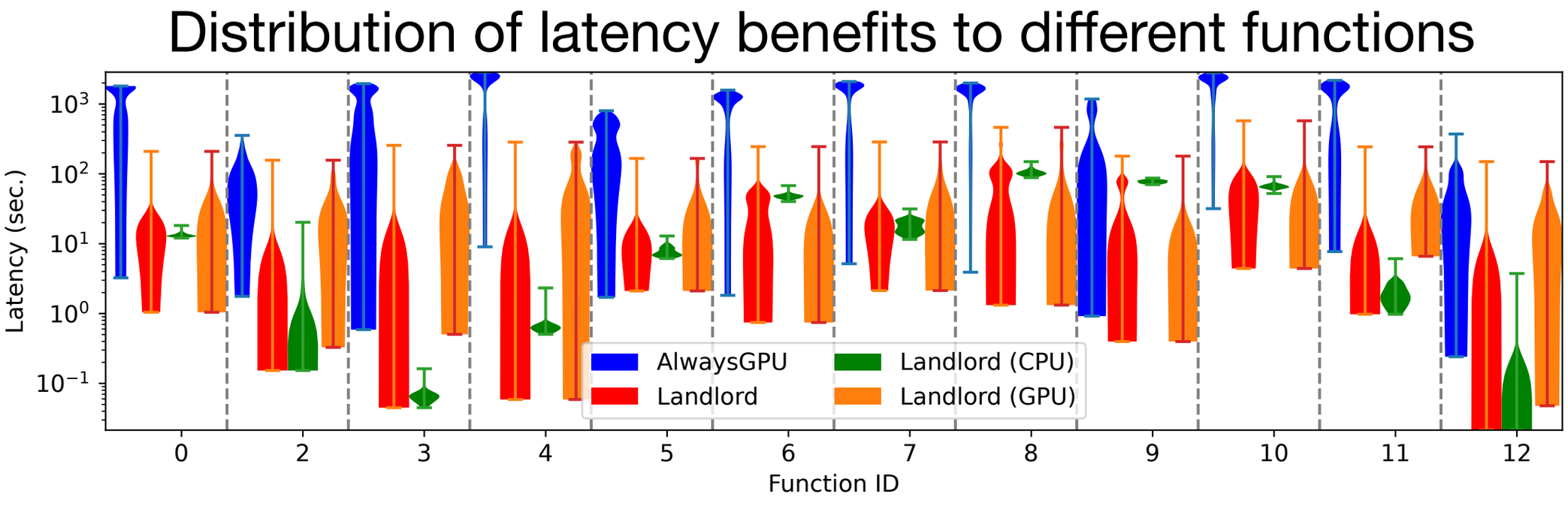
- Landlord caching: a meta-algorithm for size-aware caching
  - Credit associated with each item
  - Upon a hit, the item gets a credit equal to cost of cache miss
  - For a miss, rents are charged from all residents (proportional to their size). Lowest credit items evicted until room for new item
- GPULandlord intuition:
  - To increase GPU warm starts, a small number of functions are ‘resident’
  - Credits and rent proportional to opportunity cost.
    $\pi_i = E[T_{CPU}] - E[T_{GPU}]$
  - E[T] is expected latency (queueing + service)
  - Rent charged in proportion to size and popularity (enqueued items)

GPULandlord Algorithm

```
1:  function GpuLandlord(item)        ▷ Item is function metadata:{fname, credits, N_G, T_G, T_C, ...}
2:      if present(item) then
3:          new_credit = OpportunityCost(item)                          ▷ T_C − T_G
4:          if new_credit < 0 then
5:              return Miss(item)                                       ▷ M1: Negative credit miss
6:          else
7:              return Hit(item)
8:      if AdmitFilter(item) then
9:          Hit(item)
10:     else
11:         Miss(item)
12: function Hit(item)
13:     item.credit += OpportunityCost(item)
14:     return GPU
15: function Miss(item)
16:     ChargeRents(OpportunityCost(item))
17:     item.credit += OpportunityCost(item)                            ▷ Accumulate credit
18:     return CPU
19: function AdmitFilter(item)
20:     return GPU with probability 1/1+item.N_G                        ▷ A1: New function lottery
21:     if GpuLoad( ) < α then
22:         return OpportunityCost(item) > 0                            ▷ A2: Positive credit criteria
23:     item.credits += OpportunityCost(item)                           ▷ Accumulate credits
24:     if T_G/S_G > β then
25:         return false                                                ▷ M2: Overload miss
26:     return item.credits > victim.credits                            ▷ A3: Cumulative opportunity cost displacement
```

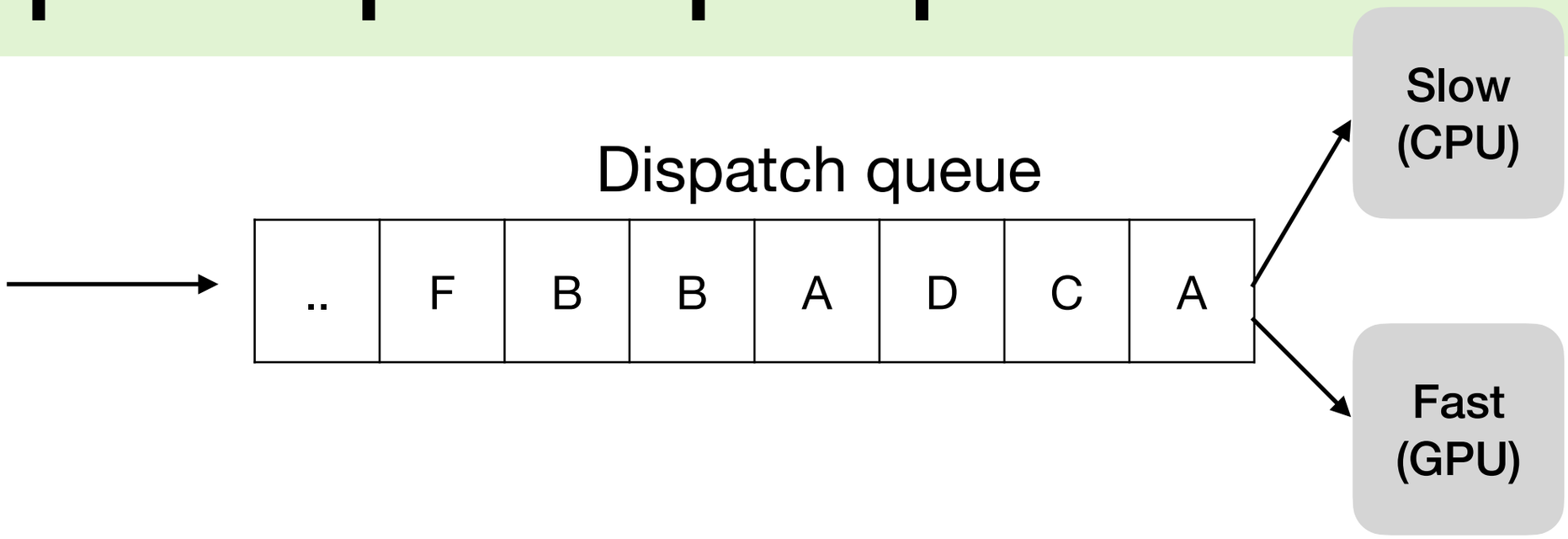
Polymorphic Function Performance





(b) High arrival rate.

Polymorphic dispatch open questions



- Classic [Lin+Kumar ‘84]: threshold-based dispatch to fast server (wait for certain time for the fast server, based on the queue backlog).
- [Hyytia+Righter ’22]: ICE: faster server gets to choose the smaller jobs (skim the icing on the cake)
  - Different speedups of functions => these classic policies cannot be easily applied
- Reinforcement Learning?
- What to optimize? Weighted avg latency? Throughput?
- What is fairness in this context?
  - Skewed workloads => specialized heuristics can work great
    - E.g., high frequency small functions always on CPU

Conclusions

- FaaS: de-facto programming model for modern cloud applications
- Locality and heterogeneity two central challenges for providers
- Keep-alive highly effective at combating cold-starts
- Load-balancing: Consistent hashing with bounded loads
- GPU functions: fair queueing principles to handle contention
  - Modeling MQFQ performance? Lot of non-linearity
  - Load balancing for GPU clusters? CH-BL induces “too much” locality and queueing for the popular functions. New techniques needed
- Polymorphic functions: run on either CPU or GPU.
  - Dispatching policies: how to choose? Treat GPU as a cache? Queueing-based approaches? Reinforcement Learning?

References et.al.

https://cgi.luddy.indiana.edu/~prateeks/