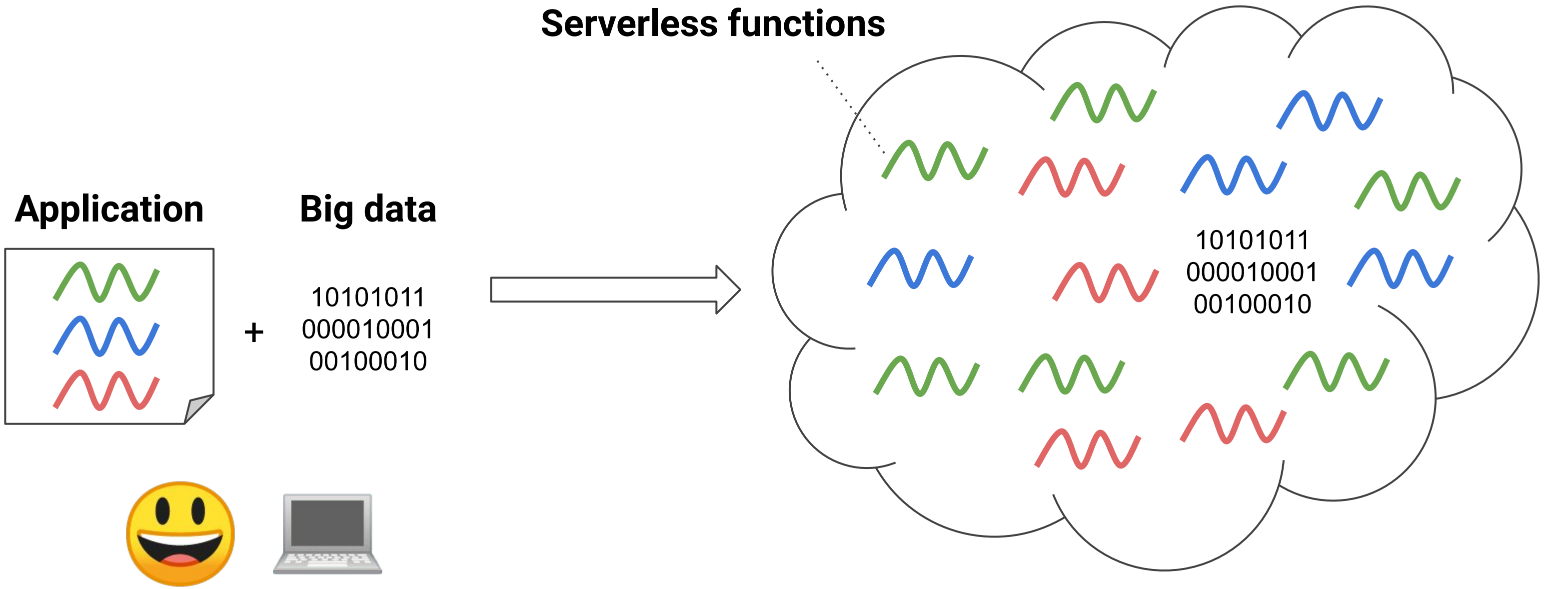
FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing

Simon Shillaker and Peter Pietzuch
Large-scale Data and Systems Group, Imperial College London

LSDS
Large-Scale Data & Systems Group

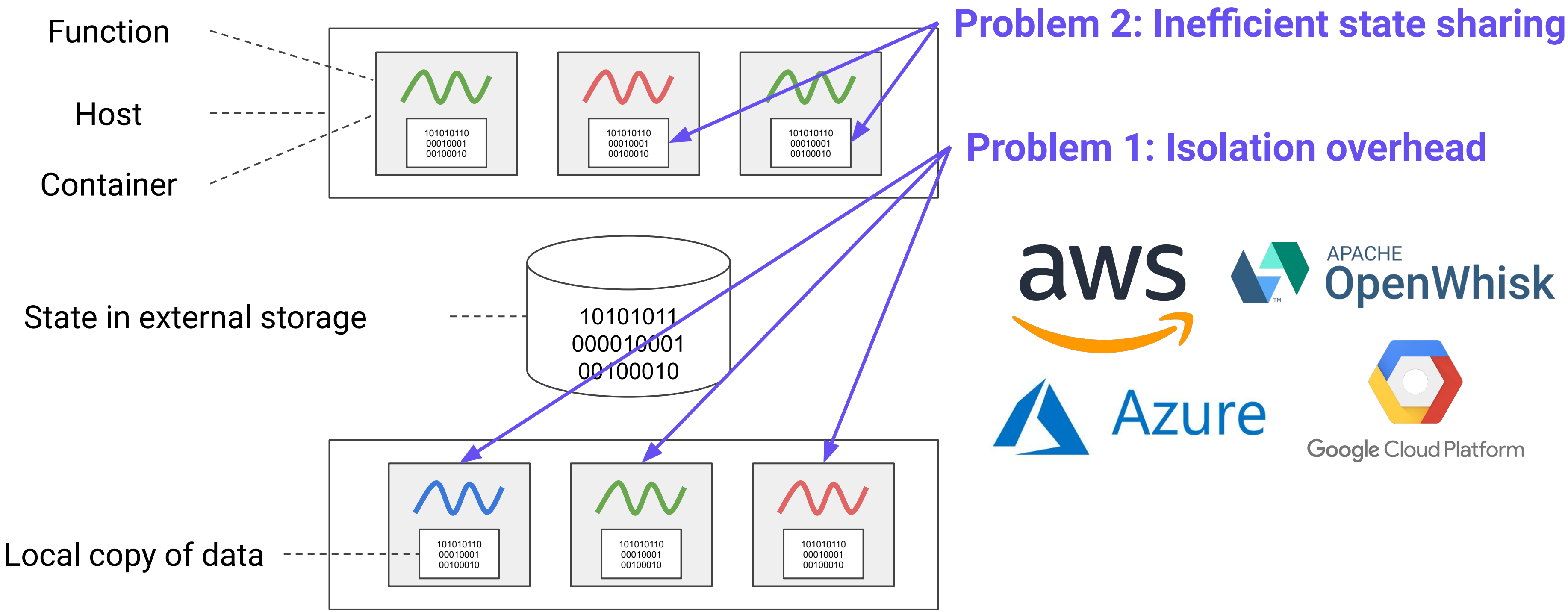
Imperial College London

Serverless Big Data Vision



Cheap, highly scalable big data processing

Serverless Under the Hood



Images: AWS , Azure , GCP , OpenWhisk

Problem 1: Isolation Overhead

Per tenant isolation, i.e. sharing containers

E.g. ***PyWren****, Jonas et al., SoCC '17;* ***Crucial****, Barcelona et al., Middleware '19*

✅ Spreads isolation overhead
❌ Loses fine-grained scaling

Snapshots and restore

E.g. ***SOCK****, Oakes et al., ATC '18;* ***SEUSS****, Cadden et al., Eurosys '20;* ***Catalyzer****, Du et al., ASPLOS '20*

✅ Low initialisation time
❌ Same memory footprint

Software-based Isolation

E.g. ***"Micro" services****, Boucher et. al, ATC '18;* ***Cloudflare Workers****;* ***Fastly Terrarium***

✅ Low overheads
❌ No resource isolation

Problem 2: Inefficient State Sharing

Add extra services to containers

*E.g. **Cloudburst**, Sreekanti et al., arXiv '20; **SAND**, Akkus et al., ATC '18*

✔️ Reduces network overhead
❌ Still duplicates locally, increases isolation overhead

Execute functions on external storage

*E.g. **Shredder**, Zhang et al., SoCC '19*

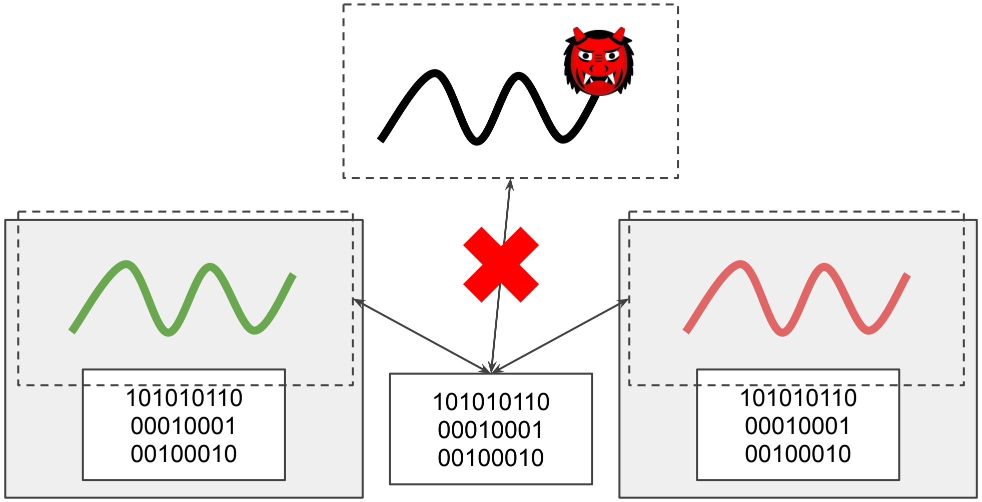
✔️ Moves code to data
❌ Does not replicate across hosts

Make external storage faster

*E.g. **Pocket**, Klimovic et al., OSDI '18*

✔️ Reduces latency
❌ Still not sharing

How Do We Efficiently Share State But Maintain Isolation?



We need an isolation mechanism that gives us fine-grained control over memory

Software-Fault Isolation with WebAssembly

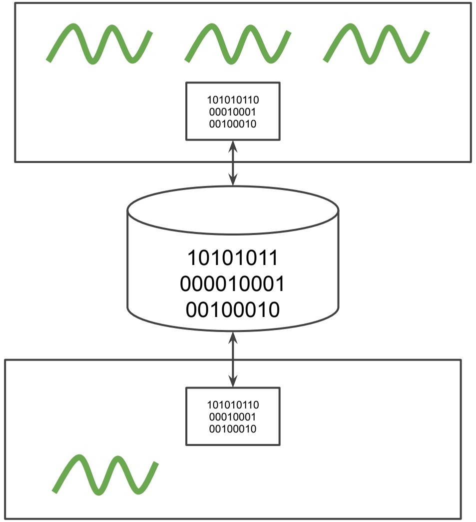
WebAssembly

- Lightweight memory safety
- Used by Fastly, Cloudflare, Krustlet

Challenges:

- *Relax isolation to share memory at runtime*
- *Virtualisation between functions and host resources*

Two-Tier State - Distribution *and* Locally-Shared State



Local tier
Shared memory

Global tier
Cross-host synchronisation

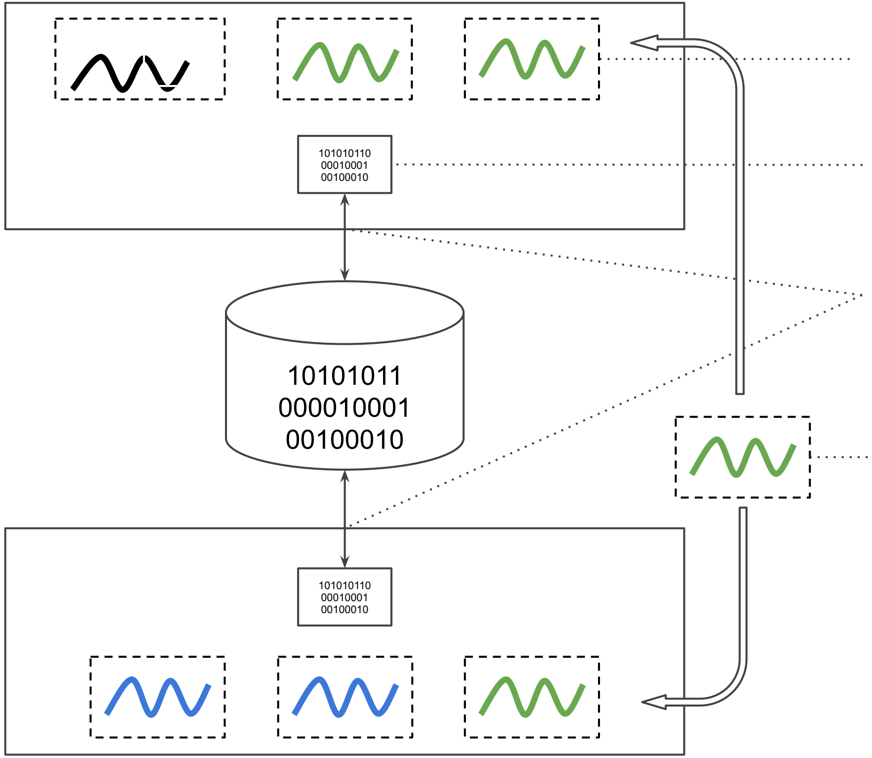
Two-tier state

Challenges:

- *Hide complexity from the user*
- *Minimise synchronisation*
- *Schedule to optimise co-location*

Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing

https://github.com/lsds/Faasm



Faaslet isolation

Shared memory regions

Global synchronisation

Proto-Faaslet snapshots

Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing

Problem 1: Isolation overheads

Faaslets - lightweight isolation based on WebAssembly

Host interface - minimal serverless-specific virtualisation

Proto-Faaslets - 500μs initialisation, 90kB memory

Problem 2: Inefficient state sharing

Faaslet shared regions - shared memory without breaking isolation

Two-tier state - global synchronisation

Problem 1: Isolation overheads

Faaslets - lightweight isolation based on WebAssembly

Host interface - minimal serverless-specific virtualisation

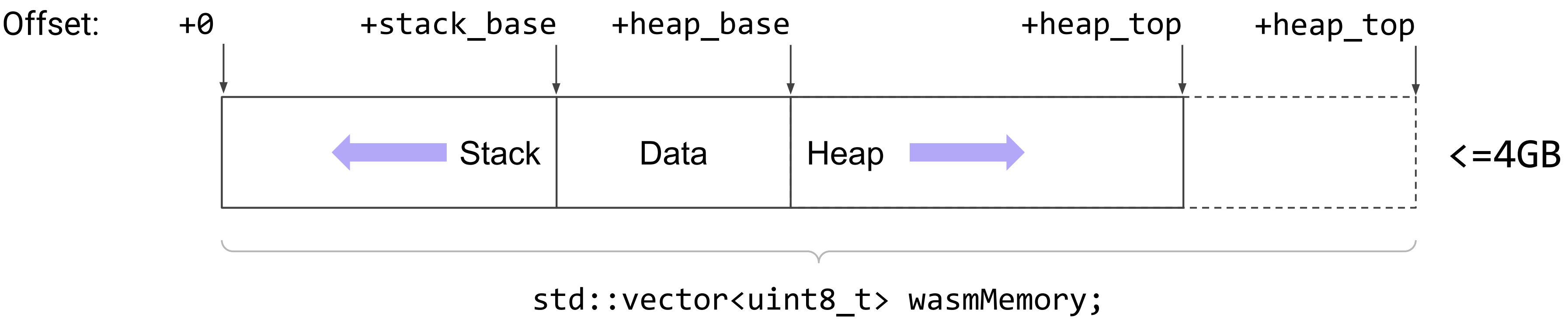
Proto-Faaslets - 500μs initialisation, 90kB memory

Problem 2: **Inefficient state sharing**

Faaslet shared regions - shared memory without breaking isolation

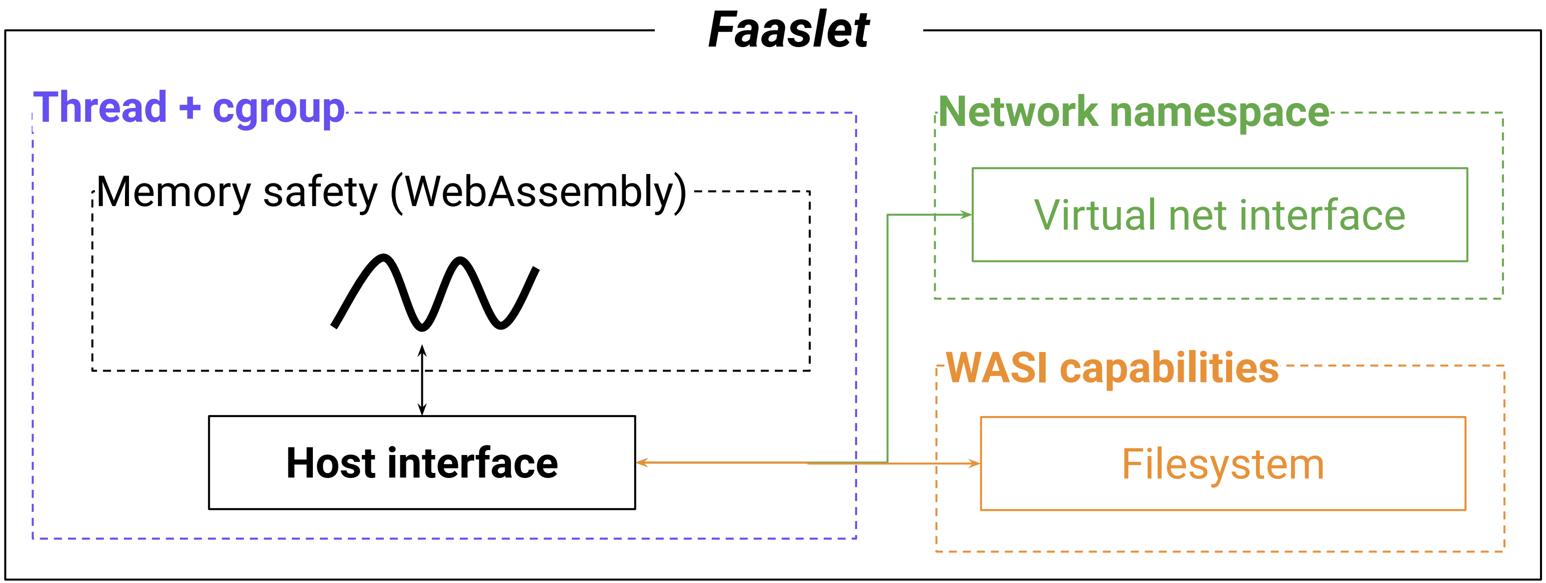
Two-tier state - global synchronisation

WebAssembly - memory safety with fine-grained control



WebAssembly memory model

Memory safety *and* resource isolation



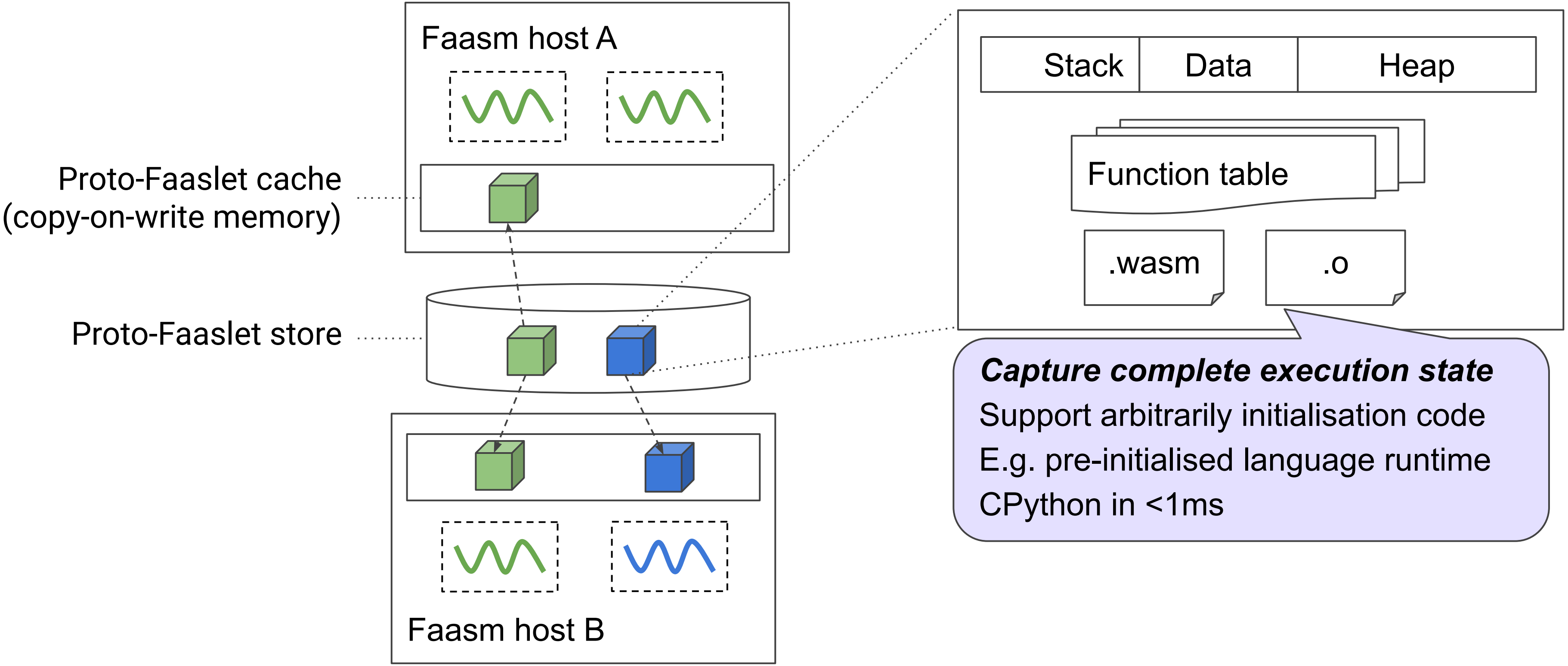
Faaslet multi-tenant isolation

Minimal Virtualisation for Serverless *and* POSIX applications

Category	*Sub-category*	*API*
Serverless	Chaining	`chain_call()`, `await_call()`, ...
	State	`get_state()`, `set_state()`, ...
POSIX	Dynamic Linking	`dlopen()`, `dlsym()`, ...
	Memory	`mmap()`, `brk()`, ...
	Network	`socket()`, `connect()`, `bind()`, ...
	File I/O	`open()`, `close()`, `read()`, ...

The Faaslet Host Interface

Proto-Faaslets - Host-Independence, µs Restore, kBs Memory Footprint



Capture complete execution state

Support arbitrarily initialisation code

E.g. pre-initialised language runtime

CPython in <1ms

Proto-Faaslet snapshot and restore

Problem 1: Isolation overheads

Faaslets - lightweight isolation based on WebAssembly

Host interface - minimal serverless-specific virtualisation

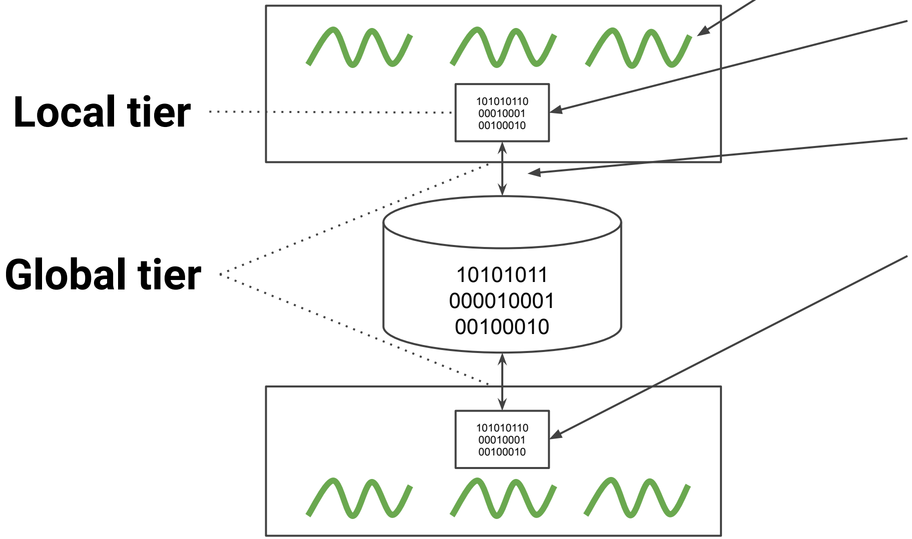
Proto-Faaslets - 500μs initialisation, 90kB memory

Problem 2: **Inefficient state sharing**

Faaslet shared regions - shared memory without breaking isolation

Two-tier state - global synchronisation

Two-Tier State Architecture Top-Down View



1. FAASM programming model
2. Faaslet shared memory regions
3. Two-tier push-pull
4. Serialisation-free data transfer

FAASM Programming Model - Distributed Machine Learning (SGD)

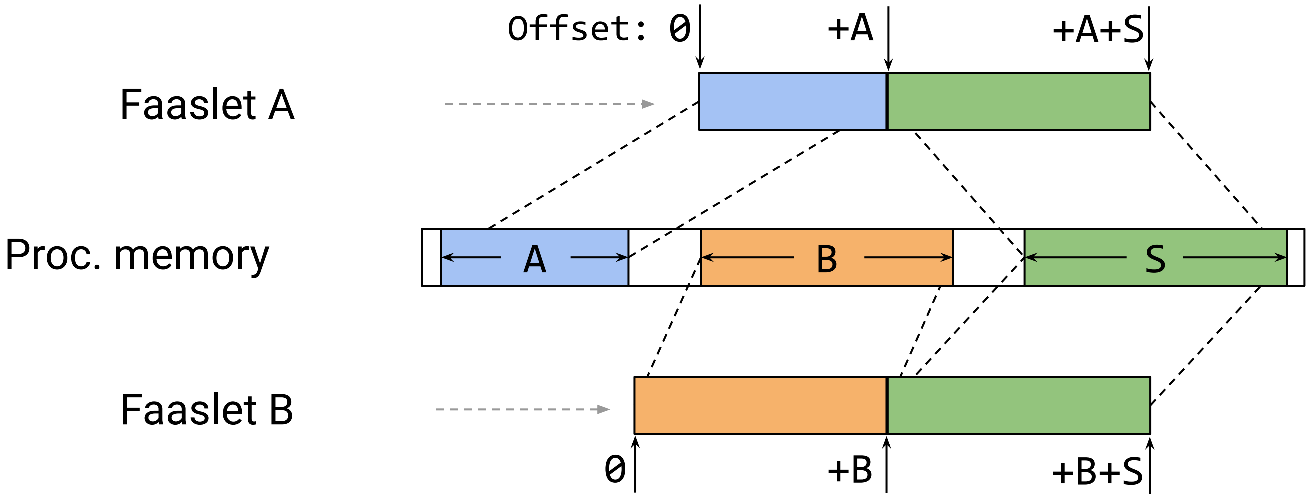
```
t_a = SparseMatrixReadOnly("training_a")
t_b = MatrixReadOnly("training_b")
weights = VectorAsync("weights")

@serverless_func
def weight_update(idx_a , idx_b):

    for col_idx , col_a in t_a.columns[idx_a:idx_b]:
        col_b = t_b.columns[col_idx]
        adj = calc_adjustment(col_a , col_b)

        for val_idx , val in col_a.non_nulls ():
            weights[val_idx] += val * adj

            if iter_count % threshold == 0:
               weights.push()

@serverless_func
def sgd_main(n_workers , n_epochs):
    for e in n_epochs:
        args = divide_problem(n_workers)
        c = chain(weight_update, n_workers, args)
    await_all(c)
```

High-level Object-Oriented abstractions
Read-only matrices
Asynchronous vector
Flexible consistency

Standard Programming constructs
Transparent optimisations
Direct access to shared memory

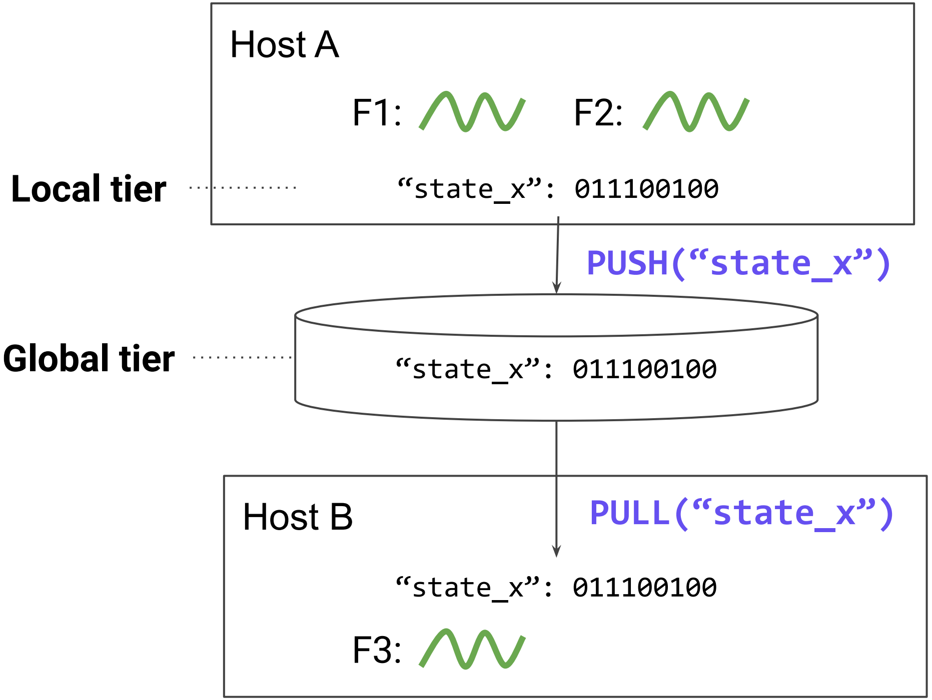
Intuitive mark-up
Function annotation
Fork-join parallelism

Shared Memory Without Breaking Safety Guarantees



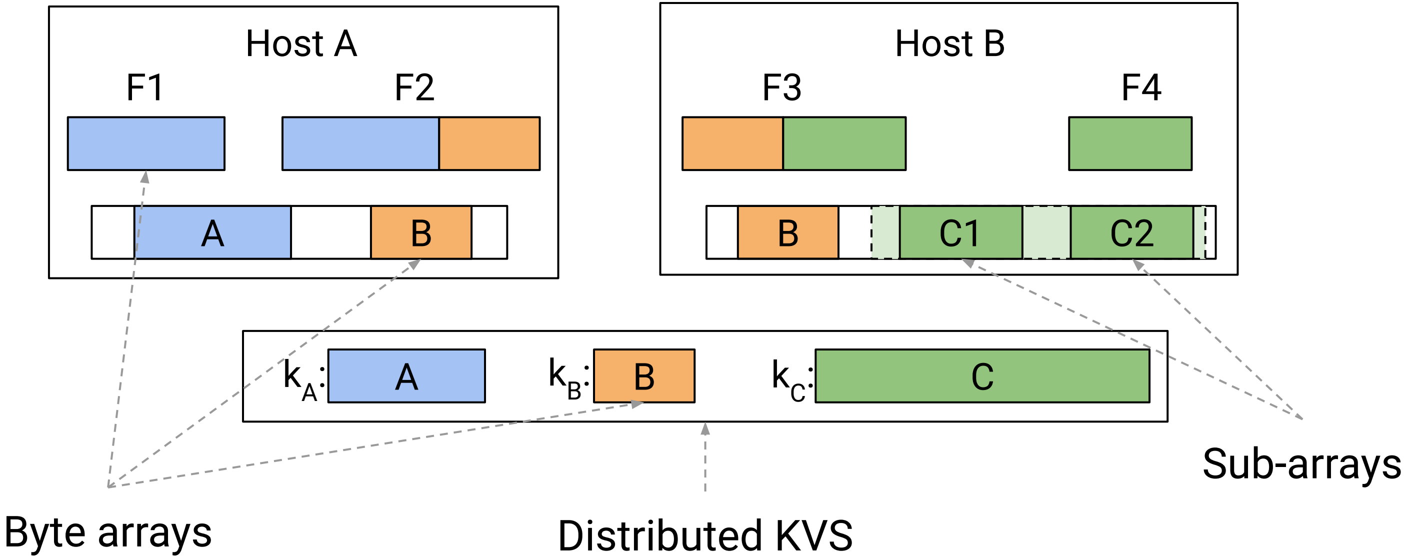
Faaslet Shared Memory Regions

Push-pull - Global Synchronisation with Variable Consistency



Two-Tier Push-Pull

Serialisation-Free Transfer of Arbitrarily Complex Data Structures



Faasm's serialisation-free state

Evaluation

Questions:

1. How do Faaslets compare to containers?
2. Can FAASM improve efficiency and performance of ML training?
3. Can FAASM improve throughput of ML inference?
4. Does Faaslet isolation affect performance of dynamic languages?

Comparison:

- Knative running identical code
- Code compiled natively for Knative
- Code compiled to WebAssembly for FAASM

Image: Knative

How do Faaslet Overheads Compare to Containers?

Lower overheads mean lower latency and lower costs

	Docker (alpine)	Faaslets	Proto-Faaslets	vs. Docker
Initialisation	2.8s	5.2ms	0.5ms	**5.6K x**
CPU cycles	251M	1.4K	650	**385K x**
Memory Footprint	1.3MB	200KB	90KB	**15 x**
Density	~8K	~70K	>100K	**12 x**

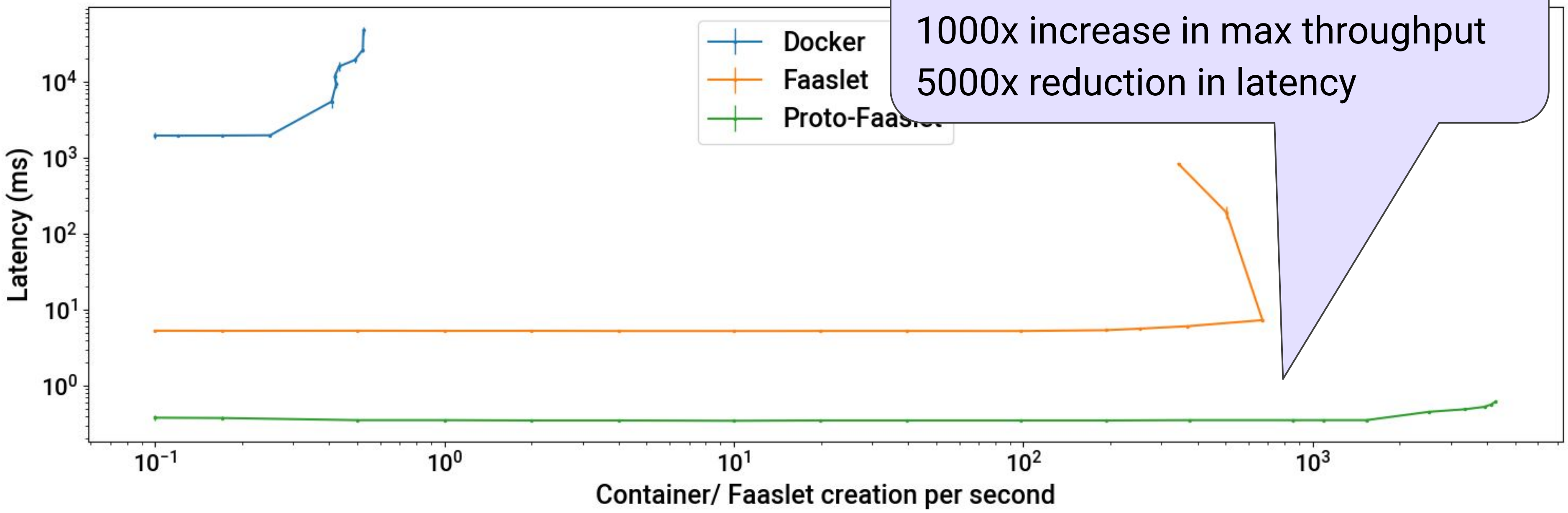
How do Faaslets “Churn” Compared to Containers?

Higher churn means higher utilisation of shared infrastructure

High Churn

1000x increase in max throughput

5000x reduction in latency



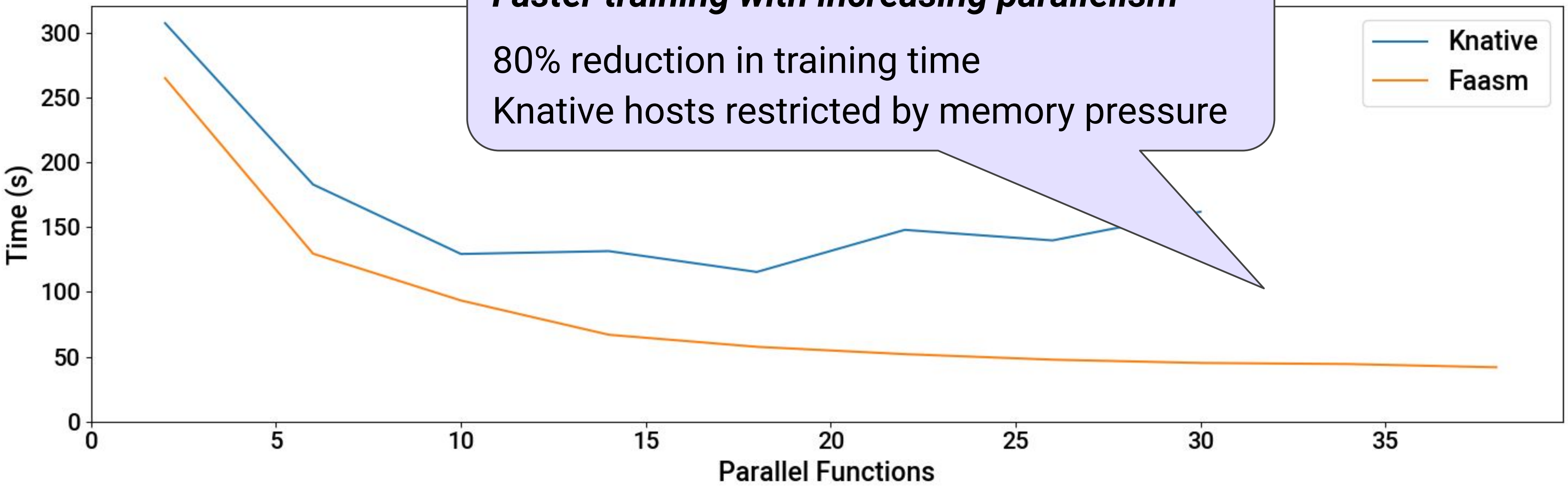
Can Faasm Improve Efficiency and Performance of Parallel ML Training?

Parallel processing on co-located data reduces training time

Faster training with increasing parallelism

80% reduction in training time

Knative hosts restricted by memory pressure

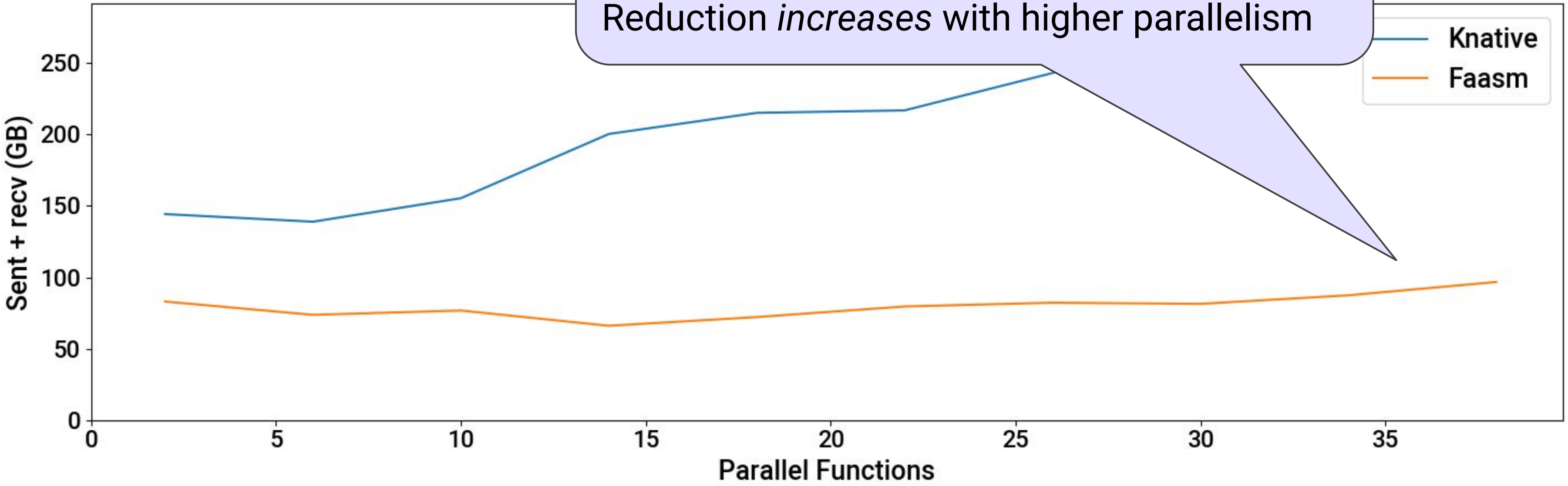


Can Faasm Improve Efficiency and Performance of Parallel ML Training?

Reduced data shipping reduces costs

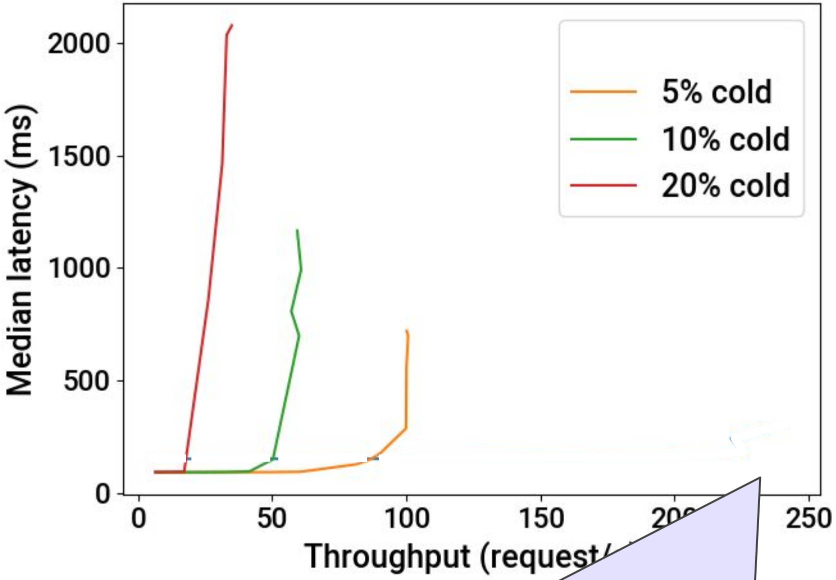
Reduced network transfers

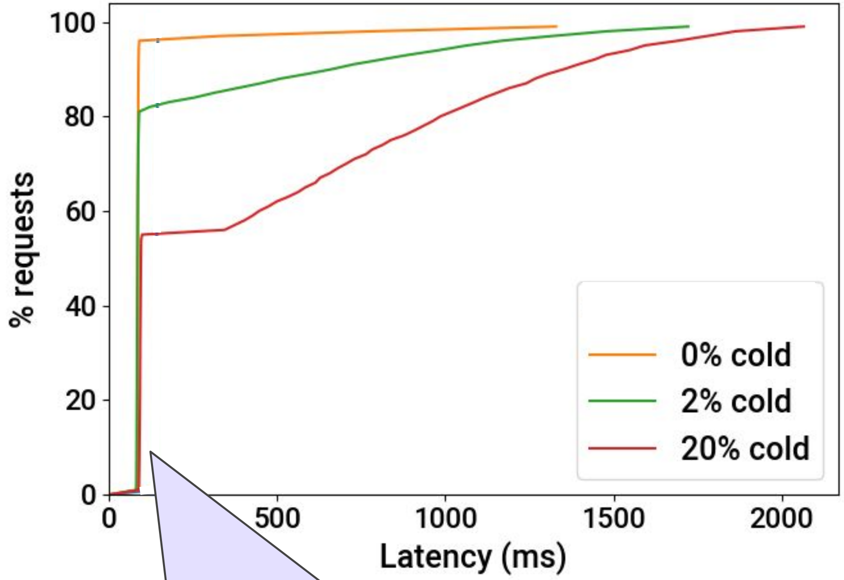
60% reduction in network transfers
Reduction *increases* with higher parallelism



Can Faasm Improve Throughput and Reduce Latency Serving ML Inference?

Proto-Faaslets increase max throughput and reduce latency





Increased Throughput

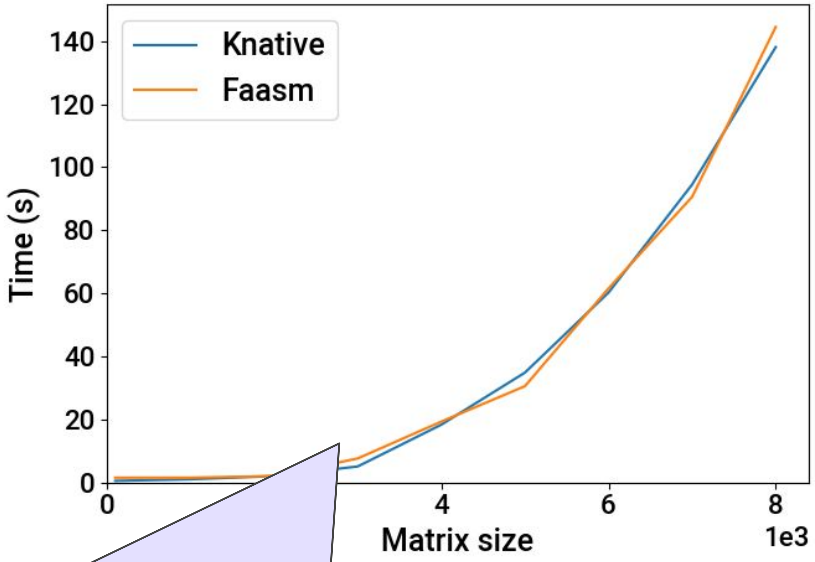
Negligible cold starts with Proto-Faaslets
120% increase in max throughput with 5% cold starts

Decreased tail latency

90% reduction in tail latency

Does Faaslet Isolation Affect Performance of Dynamic Languages?

Faaslet isolation has negligible impact on a distributed Python application

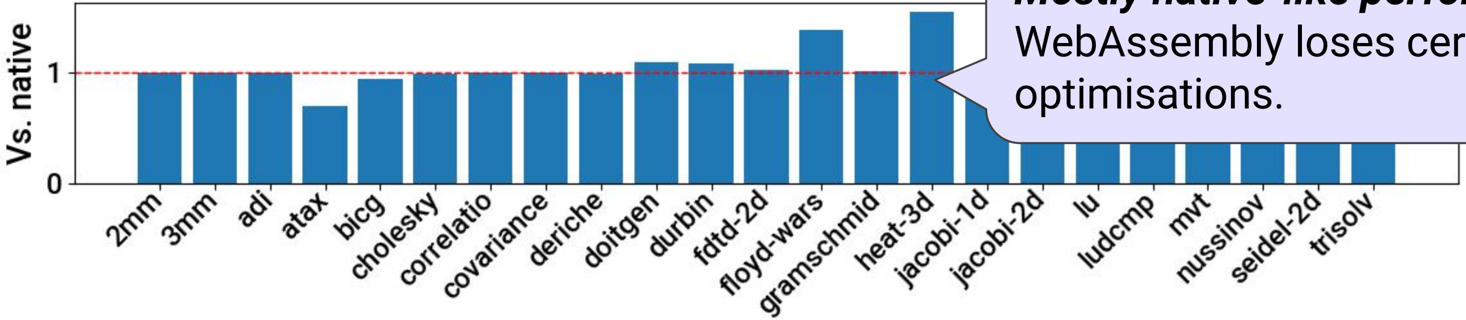


Comparable performance

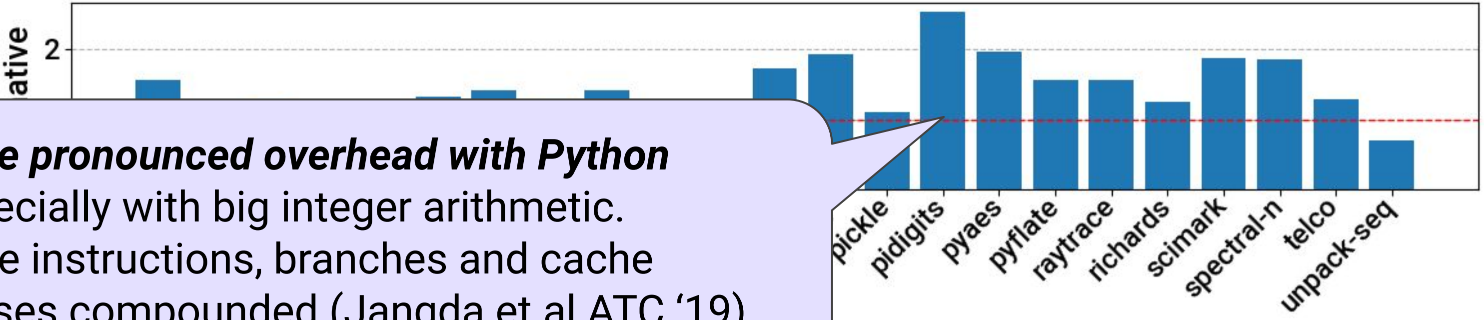
Faaslet isolation shows no significant overhead
Effect persists with increasing matrix size

Does Faaslet Isolation Affect Performance of Dynamic Languages?

Performance overheads increase as applications become more complex



Mostly native-like performance in C
WebAssembly loses certain loop optimisations.



More pronounced overhead with Python
Especially with big integer arithmetic.
More instructions, branches and cache misses compounded (Jangda et.al ATC ‘19).

Conclusions

FAASM makes serverless faster and cheaper:

- Current systems exhibit isolation overhead and inefficient state sharing
- FAASM reduces overheads with Faaslets and Proto-Faaslets
- FAASM supports efficient locally shared and globally synchronised state
- Future work: serverless HPC, trusted hardware, unikernel-based runtime

https://github.com/lsds/Faasm

Thank you