Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing

Jordan Overbo
Zoe Sadeghi

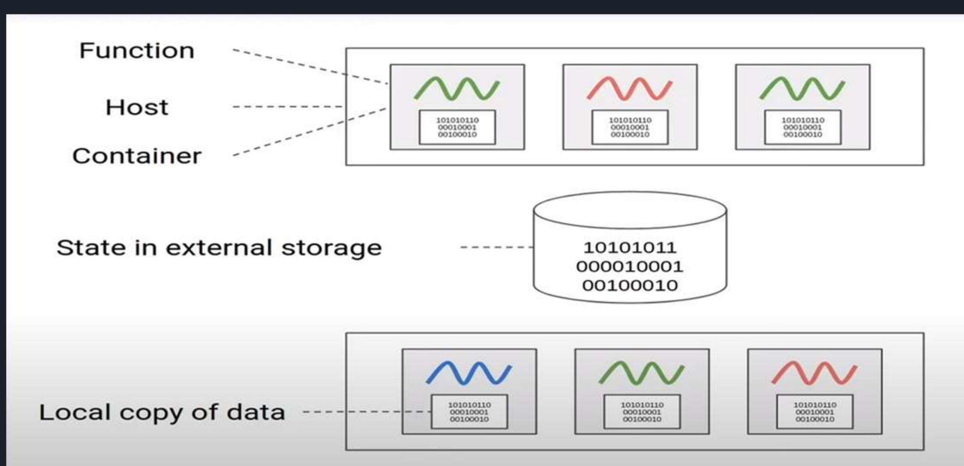
Outline

- Introduction
- Explanation of Problems
- Related Works
- Overview of Faasm
- Key Contributions From Author
- Author's Evaluation
- Author's Conclusions
- Critiques: Strengths/Weaknesses/Evaluation
- Gaps in Research

Introduction (FaaS)

- Function as a Service allows easy development, testing , and running of applications
- Very popular with data-intensive applications
- Decomposing computation can exploit the inherent cloud parallelism
- Many companies provide FaaS
- Functions are isolated in ephemeral, stateless containers
- Problems:
  - Data Access Overhead
  - Container Resource Footprint

Problems



Importance

- State must be maintained externally, incurring costs
- Has resulted in an inefficient model of bringing data to the function
- Repeatedly paying overhead penalties with each function call
- The large container memory footprint reduces scalability
  - Typically only a few thousand containers per 16GB of RAM
- Current solutions have solved problems individually

Related Work For Data Access Overhead

- PyWren introduced to reduce user overhead
- Idea is to share containers between tenants
- Pros
  - This spreads the data access overhead
- Cons
  - Results in the loss of fine-grained parallelism
  - Further increases container size

Related Work for Container Resource Footprint

- Cloudburst: a stateful FaaS platform
- Adds extra services to containers
- Pros
  - Provides a low latency mutable state for communication
  - Also maintains autoscaling benefits of serverless computing
- Cons
  - Duplicates locally
  - Increases the isolation overhead

Overview: Faaslets

- Isolation mechanism for data-intensive applications
- Strong memory and resource isolation guarantees
- Provides sought after efficient shared in-memory
- Supports lightweight virtualization through host interface
- Maintains memory footprint below 200KB with cold starts less than 10ms

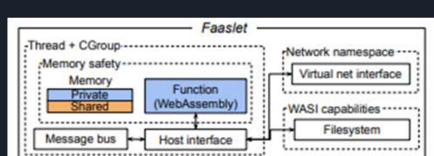


Figure 1: Faaslet abstraction with isolation

Overview: Host Interface

- Targets minimal virtualization in order to minimize overhead
- Low-level API built to support high-performing serverless applications and offers:
    - Chained serverless function invocation
    - Interaction with shared memory states
    - Range of POSIX-style functions
- Results, inputs, and state for functions represented as byte arrays

Overview: Shared Memory Regions

- Adds new concept of shared regions to existing WebAssembly model
- Offers functions concurrent access to disjoint sections of shared memory
- No extra overhead as shared memory is supported with OS virtual memory
- Maintains security guarantee offered by current WebAssembly model

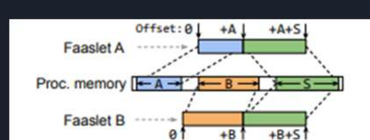


Figure 2: Faaslet shared memory region mapping

Overview: Local and Global State

- Stateful serverless applications offered through distributed data objects
- DDOs represent a single state value
- Represent state through key/value abstractions
- Local consistency ensured by local read and write locks between Faaslets
- Global consistency varies, strong provided with global read and write locks

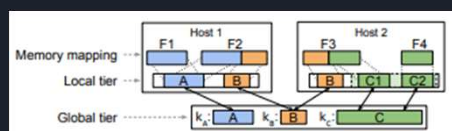


Figure 4: Faaslet two-tier state architecture

Overview: Faasm Runtime

- Serverless runtime that operates using Faaslets to provide stateful applications
- Distributed shared state scheduler to ensure as as many functions as possible are executed with warm faaslets
- Proto-Faaslets included to further reduce cold start latency
- Proto-Faaslets are Faaslets that contain a snapshot with the function's stack, heap, function table, stack pointer and data
- Further reduces cold start initialization to the hundreds of microseconds

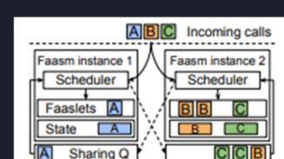


Figure 5: FAASM system architecture

Key Contributions

- Lightweight Isolation of serverless functions
  - Compiled using WebAssembly
  - CPU cycles constrained using Linux cgroups
- Support of efficient local and global state access
  - Faaslets share the same address space
  - Two-tier state architecture
- Fast initialization times
  - Reducing the cold start issue of FaaS
- Flexible host interface
  - Balance between virtualization and overhead

Author Evaluation: Setup

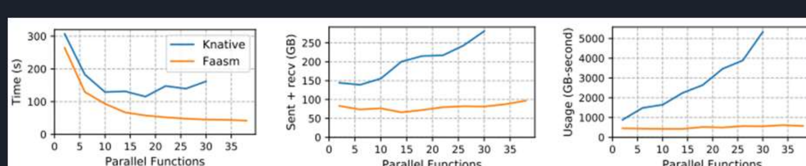
- Match Faasm against high-end serverless platform
  - Knative, which is a container-based platform built using Kubernetes
- Tested using the same code
  - Knative-specific implementation due to inability to share between functions
- Faasm Integration
  - Replicate Faasm runtime instances with Knative through the default autoscaler
- Faasm and Knative both ran on the same Kubernetes cluster
- Metrics include execution time, throughput, latency, as well as billable memory

Author Evaluation: Methods

- Machine learning training
  - Text classification using the HOGWILD! Algorithm
  - Knative and Faasm both ran using an increasing number of parallel functions
  - Reduced training size to determine performance and resource overheads
- Machine learning inference
  - Tests the initialization times on cold starts
  - Inference serving application using TensorFlow Lite
- Language Runtime Performance
  - Matrix multiplication using Python and Numpy
- Efficiency of Faaslets vs. containers
  - Footprint and cold start initialization latency of containers and Faaslets

Author Evaluation: Results

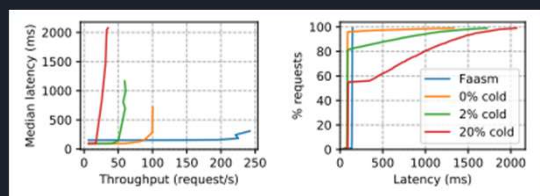
- Machine learning training
  - 10% improvement in runtime with low parallelism, 60% with 15 parallel functions
  - At 38 parallel functions, improvement reaches 80%
  - Both show increasing network transfers, but Knative starts higher and increases faster
  - Billable memory increases much slower for Faasm



(a) Training time (b) Network transfers (c) Memory usage

Author Evaluation: Results

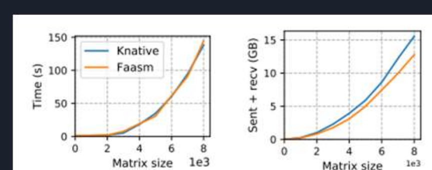
- Machine learning inference
  - All cold start ratios for Knative result in increasing median latency by 100 r/sec while Faasm maintains a latency of 120 ms through 200 r/sec
  - Faasm also maintains tail latency of under 150ms for all cold start ratios whereas Knative has a tail latency of over 2 seconds for 35% of calls



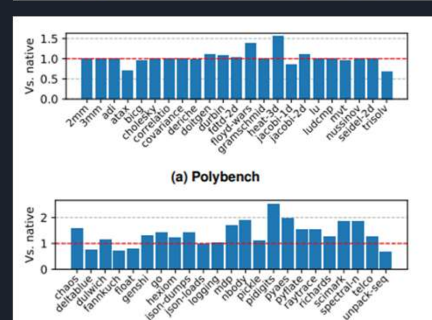
(a) Throughput vs. latency (b) Latency CDF

Author Evaluation: Results

- Language Runtime Performance
  - Faasm and Knative duration increase at almost identical rates
  - Faasm achieves 13% less network traffic across matrix sizes
  - Polybench shows comparable overhead in all but 2 benchmarks
  - Python sees a few benchmarks reach 50-60% extra overhead vs native



(a) Duration (b) Network transfer

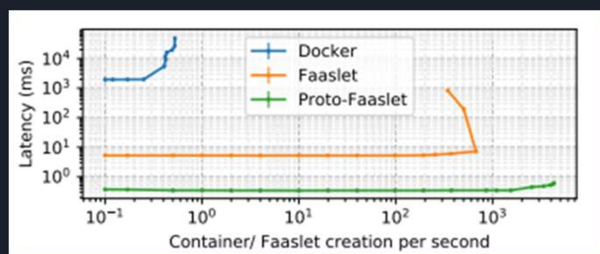


(a) Polybench

(b) Python Performance Benchmark

Author Evaluation: Results

- Efficiency of Faaslets vs. containers
  - Docker begins at ~2 sec initialization and increases after 3 executions/sec
  - Faaslets begin at ~5 ms initialization and maintain that until around 600 executions/sec
  - Proto-Faaslets begin at ~0.5 ms initialization and manage that until about 4000 executions/sec



Author Evaluation: Conclusions

- Their Faasm runtime is able to provide high performance state without compromising isolation
- Faaslets execute functions which allow memory sharing while maintaining memory safety
- Initialization times have been addressed through Proto-Faaslet snapshots
- Faasm's two-tier architecture givers users parallel in memory processing while still allowing host-to-host sharing
- Faaslets also support different language runtimes

Critique: Strengths

- Performance increased across nearly all tests
- Faasm manages to solve both problems mentioned
- Solution also maintains scalable nature of the cloud
- Would reduce costs by limiting data access overhead

Critique: Weaknesses

- The sets of techniques introduced are limited to FaaS delivery model and do not combine well with other types of delivery
- The techniques rely on low-level access to kernel functions
- All the Faaslets need to be deployed manually and fine-tuned manually

Critique: Evaluation

- Proposes a new set of techniques for a different level of abstraction
- Delivers promising results
- Fails to talk about shortcomings
- The graphs are sometimes peculiar without explanation (e.g. see Efficiency of Faaslets vs. containers)

Gaps

- Fails to talk about complexities of shared state (e.g. inconsistency when writing, locking when reading)
- Fails to talk about integration with commercial providers (e.g. AWS)
- Fails to establish a clear benefit of this method over, say, a distributed cache
- Fails to talk about costs of implementation and maintenance
- Fails to talk about how this method completely ignores immutability of data which is the main benefit of functional design

Questions?