6.5810: FAASM

Adam Belay <abelay@mit.edu>

Logistics

- Sign up to meet + discuss your final project (happening tomorrow)

Recap

- Serverless isolation mechanisms
- So far: GVisor (library OS), Firecracker (virtualization), and UKL (Unikernels)
- Today: FAASM (software fault isolation)
  - Some similarities to CloudFlare Workers (recently open sourced)
  - https://github.com/cloudflare/workers.cloudflare.com

FAASM's plan

- Rely on Web Assembly for isolation
  - Key advantage: Functions can share memory directly
- Everything else done the standard way:
  - Cgroups: memory and CPU quotas
  - Network namespaces + traffic shaping: Bandwidth limits
  - Etc.

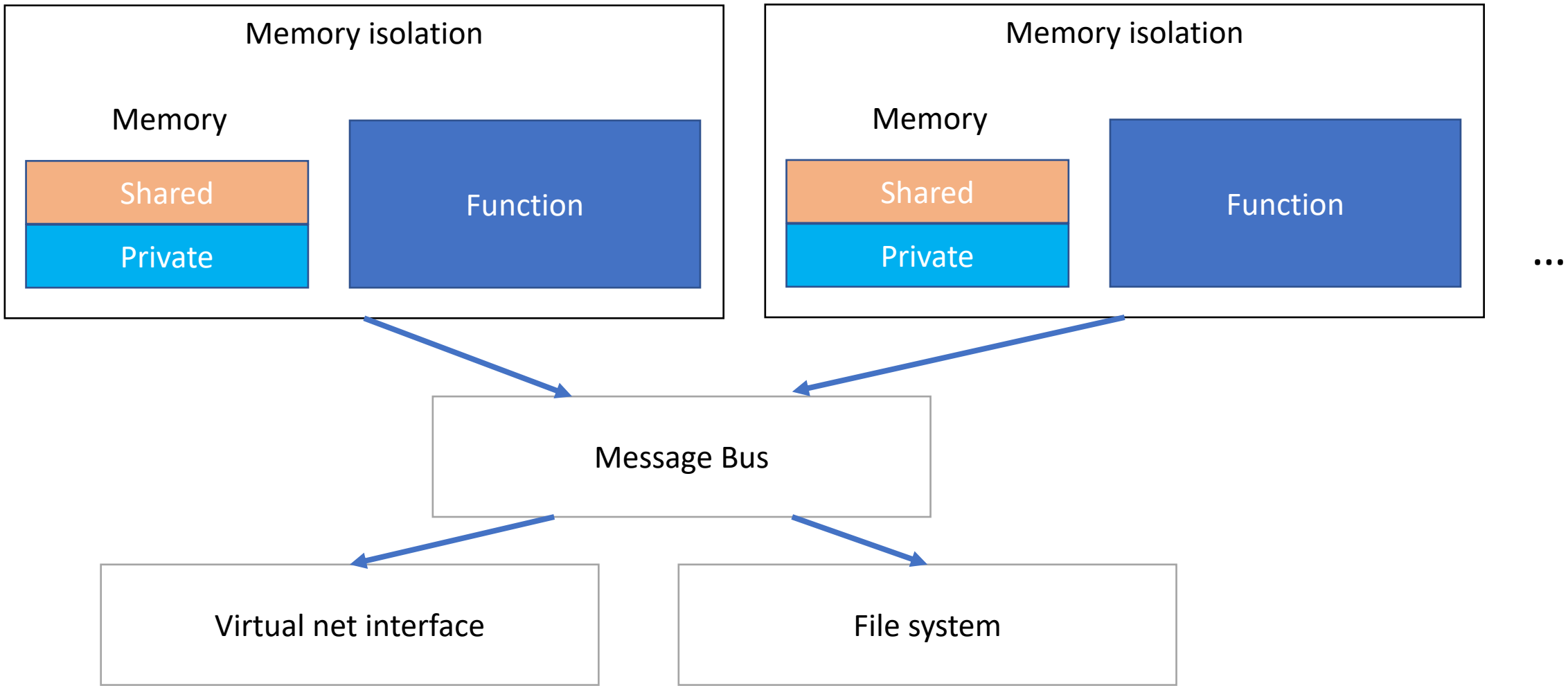
Quick aside: Software fault isolation

- Restricts a function to accesses of its own memory
- Works by inserting checks before memory and control transfer instructions
- Provides a logical isolation domain inside a process (despite sharing the same page table and memory)
- In the past: Binary translation has been used to achieve SFI
- This work targets a special instruction set architecture instead

Quick aside: Web Assembly (WASM)

- History: Highly optimized Javascript JITs already existed (e.g., V8)
- Can we leverage this existing infrastructure to run native code?
- Plan: Define a “web assembly” language for portability
  - This means you must compile to web assembly
  - But can support C, C++, Rust, etc. (any LLVM front end)
- JIT translates web assembly to native assembly
- Checks inserted to preserve memory safety

FAASM Design



Faaslet

- Isolation
  - Access constrained to a contiguous memory region
  - By default, all memory is private, but shared mappings are possible
  - Control flow integrity ensure function can’t jump out of its code
- Faaslets can be snapshotted to reduce startup times
- Each Faaslet runs in a single thread
- Standard mechanism (e.g., cgroups) enforce resource limits

Host interface

Class	Function	Action	Standard
Calls	`byte* read_call_input()`	Read input data to function as byte array	*none*
	`void write_call_output(out_data)`	Write output data for function	
	`int chain_call(name, args)`	Call function and return the `call_id`	
	`int await_call(call_id)`	Await the completion of `call_id`	
	`byte* get_call_output(call_id)`	Load the output data of `call_id`	
State	`byte* get_state(key, flags)`	Get pointer to state value for `key`	
	`byte* get_state_offset(key, off, flags)`	Get pointer to state value for `key` at offset	
	`void set_state(key, val)`	Set state value for `key`	
	`void set_state_offset(key, val, len, off)`	Set `len` bytes of state value at offset for `key`	
	`void push/pull_state(key)`	Push/pull global state value for `key`	
	`void push/pull_state_offset(key, off)`	Push/pull global state value for `key` at offset	
	`void append_state(key, val)`	Append data to state value for `key`	
	`void lock_state_read/write(key)`	Lock local copy of state value for `key`	
	`void lock_state_global_read/write(key)`	Lock state value for `key` globally	
Dynlink	`void* dlopen/dlsym(...)`	Dynamic linking of libraries	POSIX
	`int dlclose(...)`	*As above*	
Memory	`void* mmap(...)`, `int munmap(...)`	Memory grow/shrink only	
	`int brk(...)`, `void* sbrk(...)`	Memory grow/shrink	
Network	`int socket/connect/bind(...)`	Client-side networking only	WASI
	`size_t send/recv(...)`	Send/recv via virtual interface	
File I/O	`int open/close/dup/stat(...)`	Per-user virtual filesystem access	
	`size_t read/write(...)`	*As above*	
Misc	`int gettime(...)`	Per-user monotonic clock only	
	`size_t getrandom(...)`	Uses underlying host `/dev/urandom`	

Table 2: Faaslet host interface (The final column indicates whether functions are defined as part of POSIX or WASI [57].)

Host interface enables faster communication

- Normally: All intermediate results placed in a key value store over the network (e.g., S3)
- Here, functions can pass byte arrays directly to one another inside the same process using shared memory
- But some benefit lost if functions don't fit on same machine

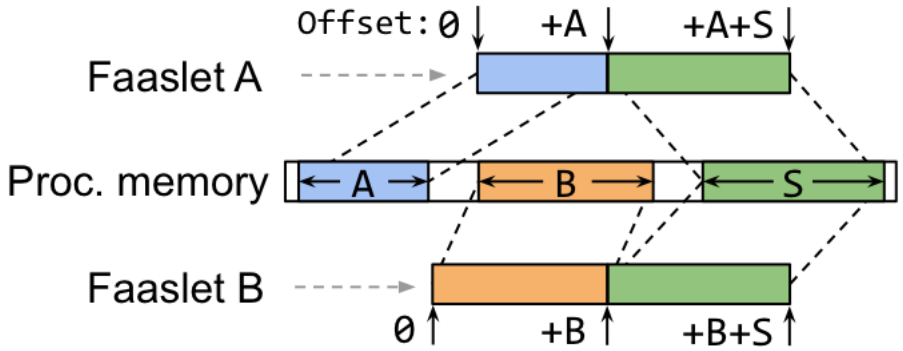


Figure 2: Faaslet shared memory region mapping

State programming model

- Build on distributed data objects (DDOs)
  - Key -> value abstraction
- Two tiers; local tier provides shared memory access, global tier transfers over the network
- Push() and Pull() operations explicitly transfer data from local to global tier

Faasm scheduling

- Distributed scheduling
- Goal: Place calls on instances with warm functions and shared data

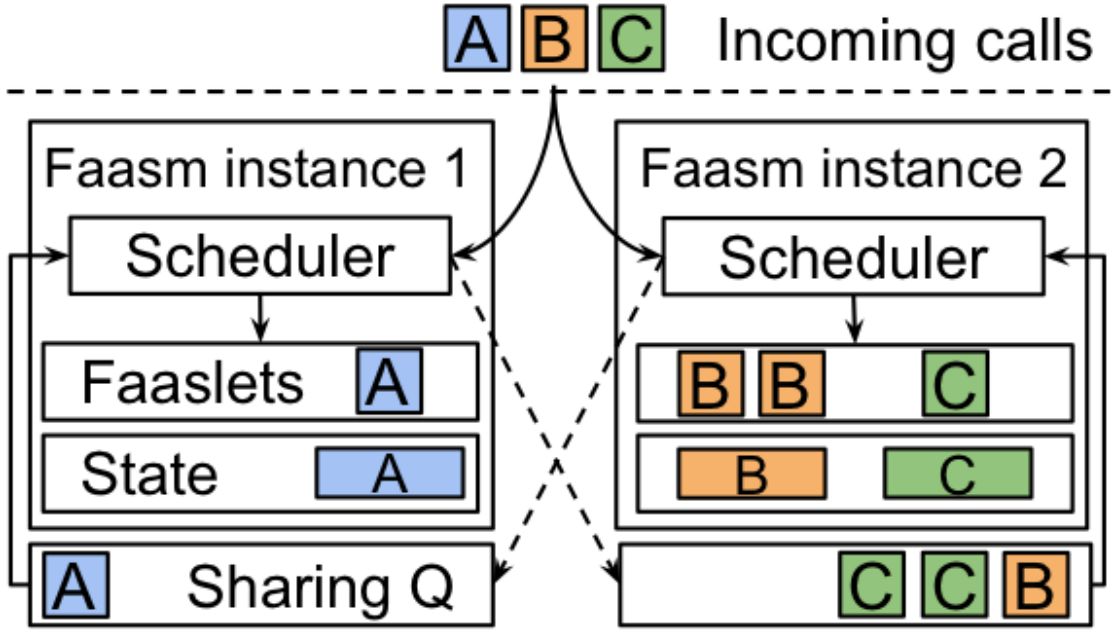
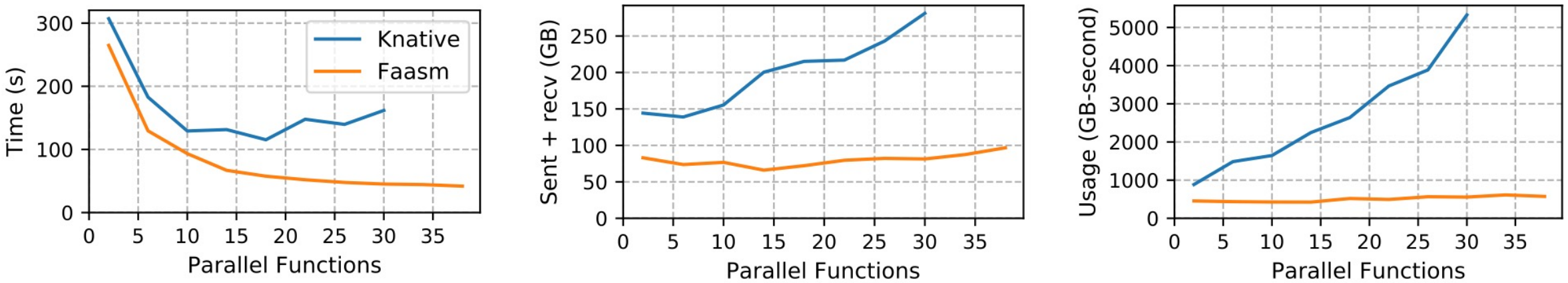


Figure 5: FAASM system architecture

Faasm snapshotting

- Web assembly represents memory as a simple array of bytes
- FAASM's plan: Fully load a function, then copy the byte array and store it for future execution in a shared object store
- Eliminates code generation cost and other initialization
- Copy-on-write mappings can be used to restore a clean copy efficiently

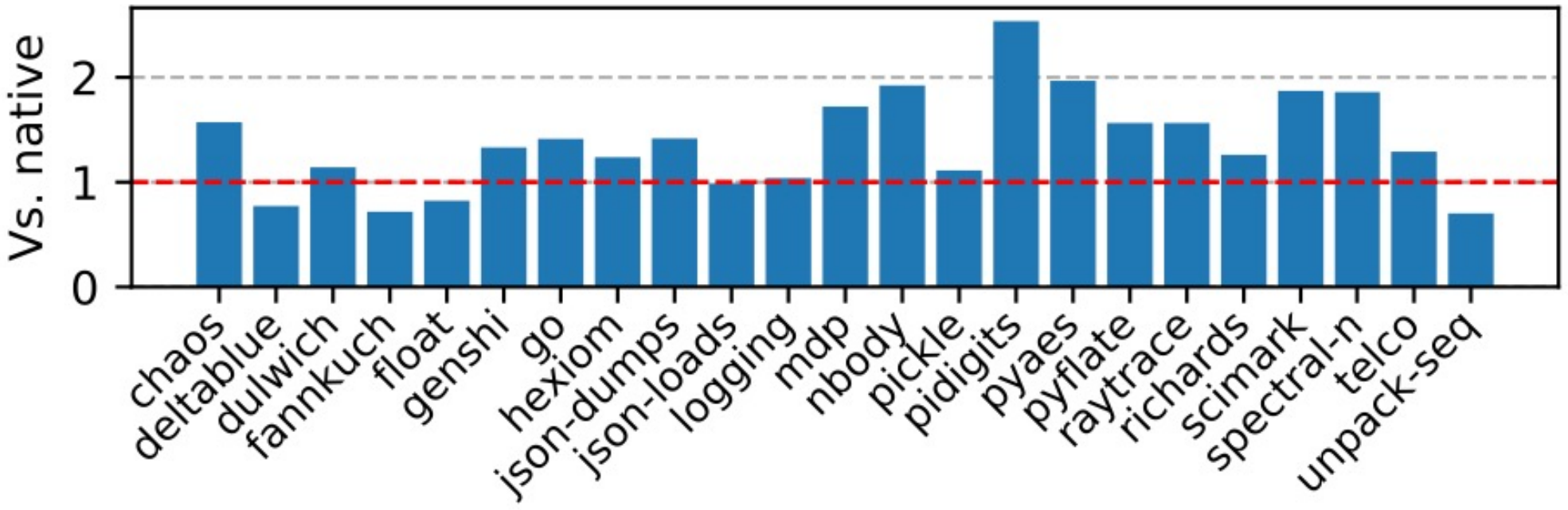
Evaluation: SGD (machine learning)



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

Figure 6: Machine learning training with SGD with Faaslets (FAASM) and containers (Knative)

Evaluation: Python compute overhead



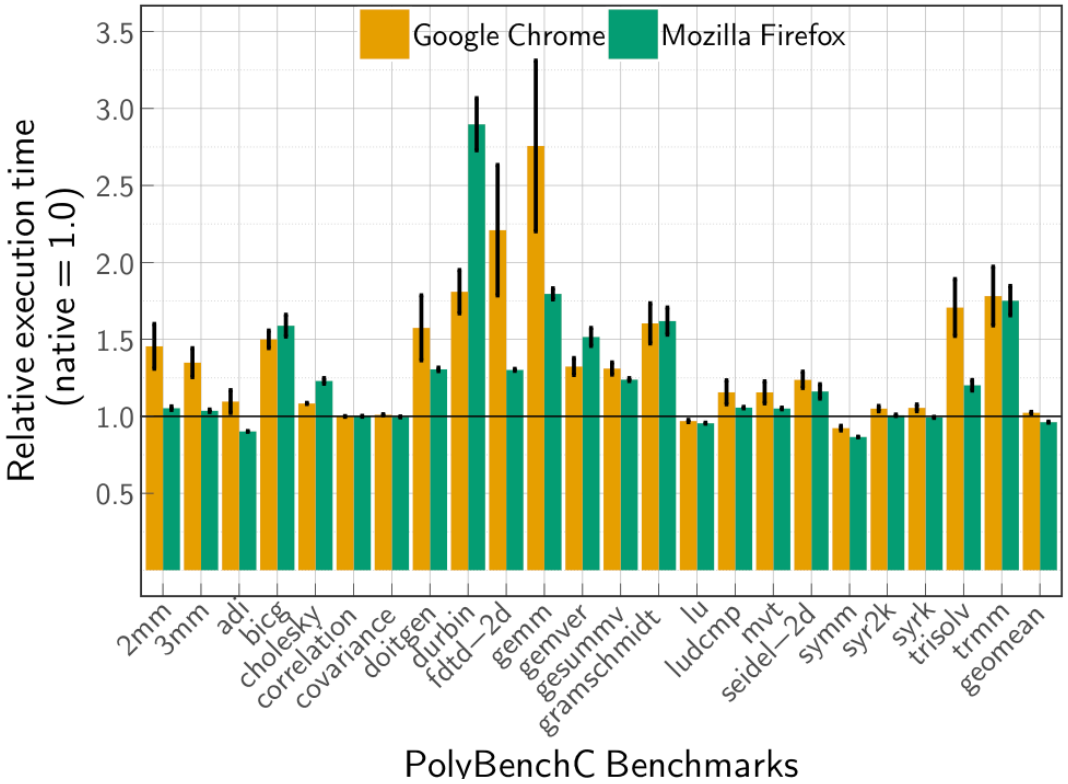
(b) Python Performance Benchmark

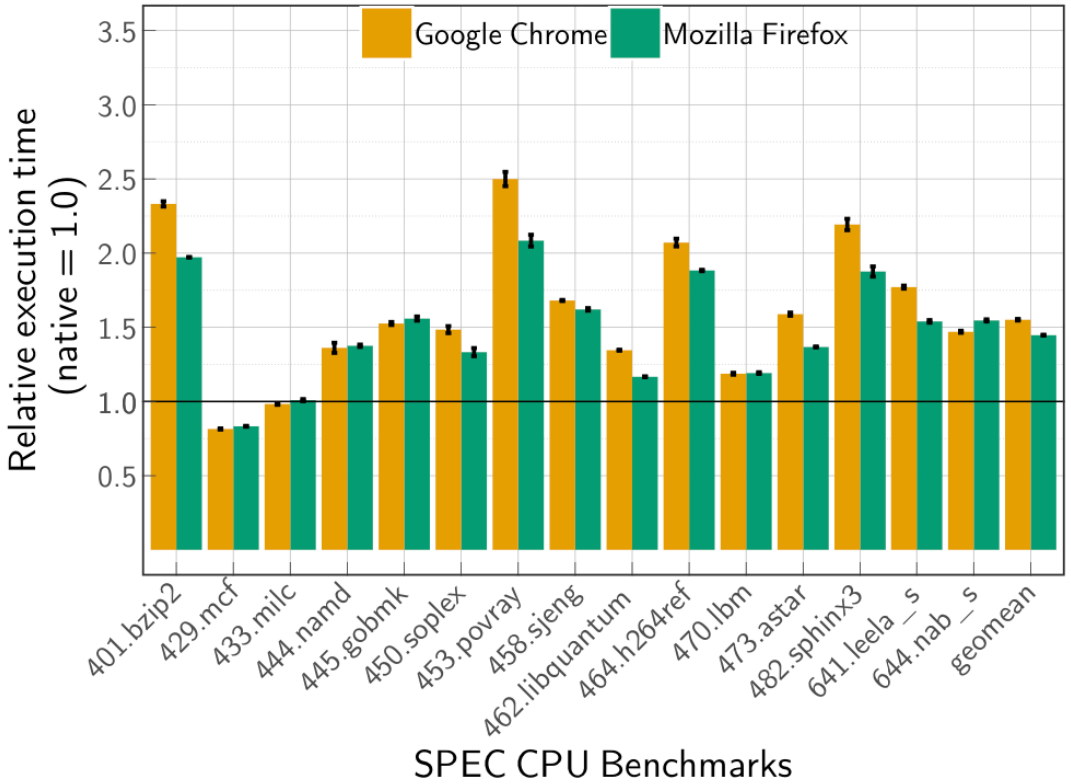
Startup times and initial memory use

	Docker	Faaslets	Proto-Faaslets	vs. Docker
Initialisation	2.8 s	5.2 ms	0.5 ms	**5.6K**×
CPU cycles	251M	1.4K	650	**385K**×
PSS memory	1.3 MB	200 KB	90 KB	**15**×
RSS memory	5.0 MB	200 KB	90 KB	**57**×
Capacity	~8 K	~70 K	>100 K	**12**×

Table 3: Comparison of Faaslets vs. container cold starts (no-op function)

Another perspective on compute overhead





Mean slowdown ~1.5x

Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. Jangda et. Al. ATC'19

Discuss: Does FAASM achieve Amazon's goals?

1. **Isolation**
2. **Density**
3. **Performance**
4. **Compatibility**
5. **Fast switching**
6. **Soft allocation**

Conclusion

- Web assembly + FAASM enables very fast context switching, startup times, snapshotting, and data sharing
- But CPU throughput remains an issue
  - Can it be optimized further?