IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel Jon Howell Manos Kapritsos Jacob R. Lorch Bryan Parno
Michael L. Roberts Srinath Setty Brian Zill

Presenter: Oreoluwa Alebiosu

Iron Fleet

- Build complex, efficient distributed systems whose implementations are provably **safe** and **live**.
    - Implementations are correct, not just abstract protocols
    - Proofs are machine checked
- First work to produce mechanical proof of liveness of non-distributed protocol and implementation
    - Proofs are not absolute and assume correctness of some things
    - Work is on proving correctness of your code

Iron Fleet

- Toolset modification
- Methodology
  - **Two-level refinement**
  - Concurrency control via reduction
  - **Always-enabled actions (Liveness)**
  - Invariant quantifier hiding
  - Automated proofs of temporal logic
- Libraries

Iron Fleet

Builds upon..

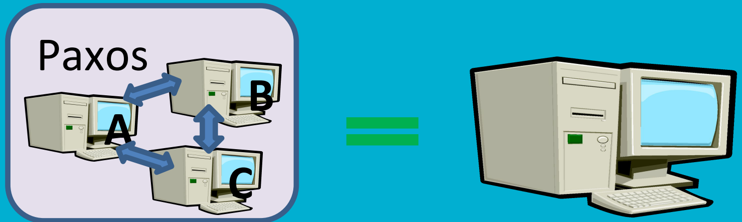
- Single-machine system verification (sel4)
- SMT Solvers
- Dafny

https://docs.google.com/document/d/1KaoFQt8rQCfw39WW4p8uUeYbMx2xtNHgadWBb4OFDVA/edit?usp=sharing

Implementation

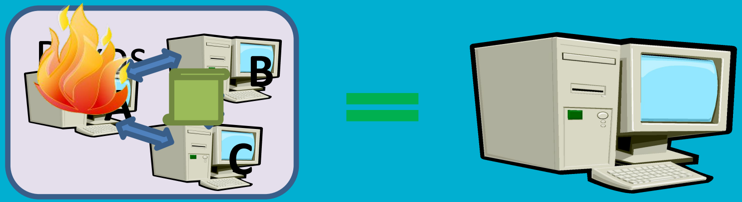
- IronRSL: Replicated state library
  - **Complex with many features:**
    - state transfer
    - log truncation
    - dynamic view-change timeouts
    - batching
    - reply cache
- IronKV: Sharded key-value store

IronRSL



- Safety property: Equivalence to single machine

IronRSL



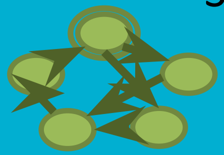
- Safety property: Equivalence to single machine
- Liveness property: Clients eventually get replies

Specification approach: Rule out *all* bugs by construction

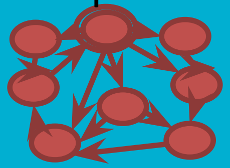
Invariant violations	Parsing errors
Race conditions	Marshalling errors
Integer overflow	Deadlock
Buffer overflow	Livelock

Refinement

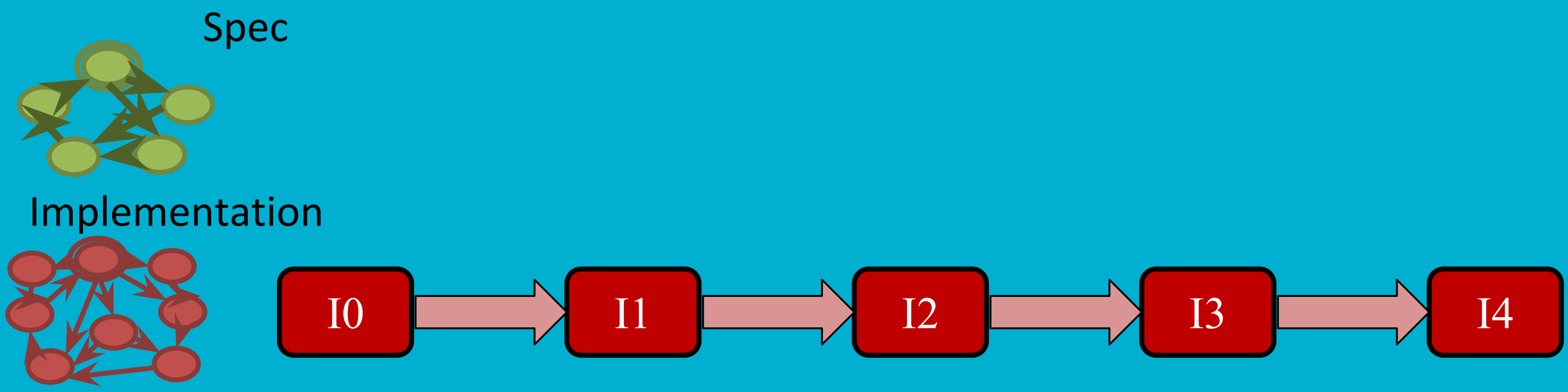
Spec



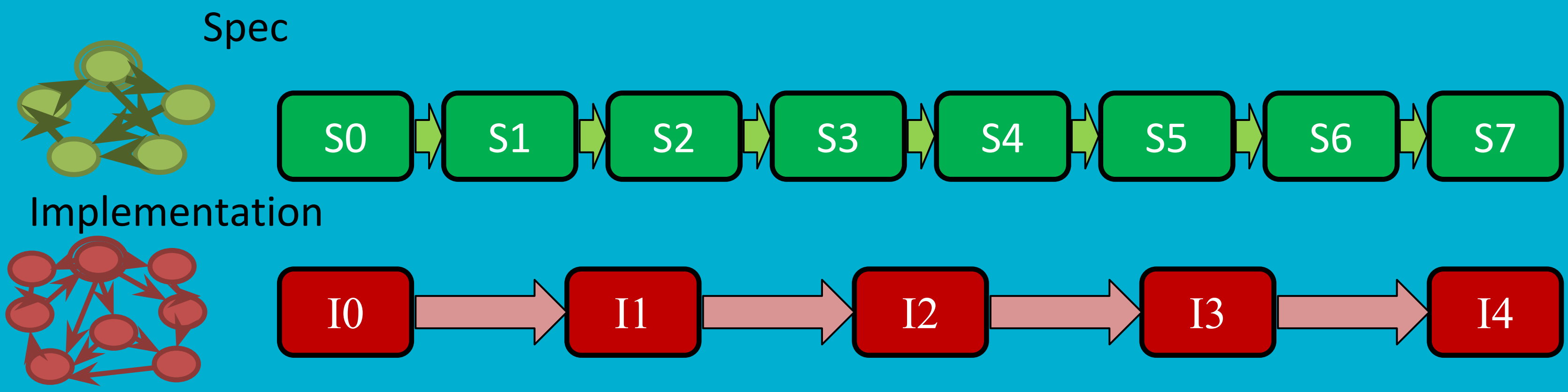
Implementation



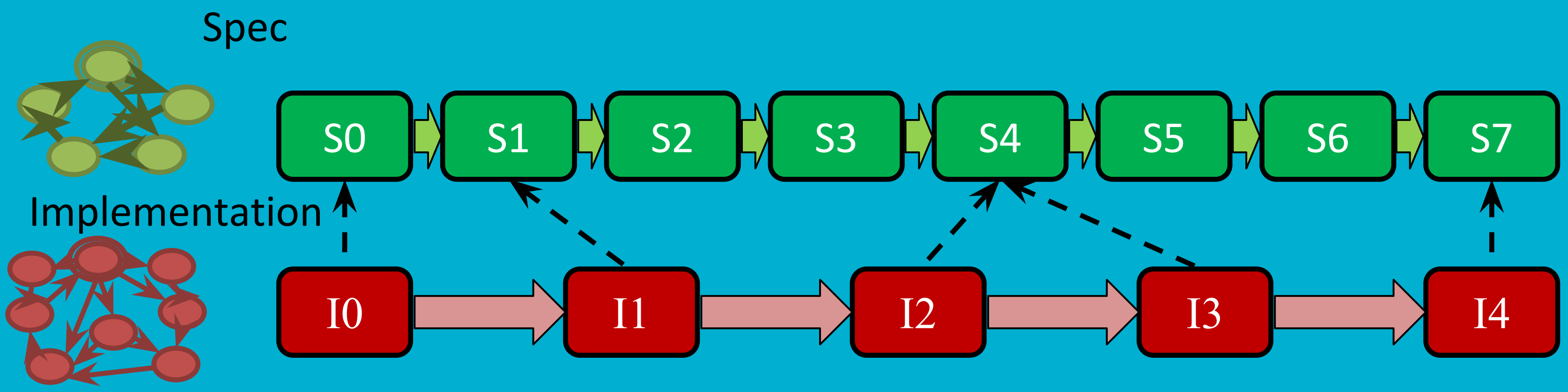
Refinement



Refinement



Refinement



Proving correctness is hard

Subtleties of distributed protocols

Maintaining global invariants

Dealing with hosts acting concurrently

Ensuring progress

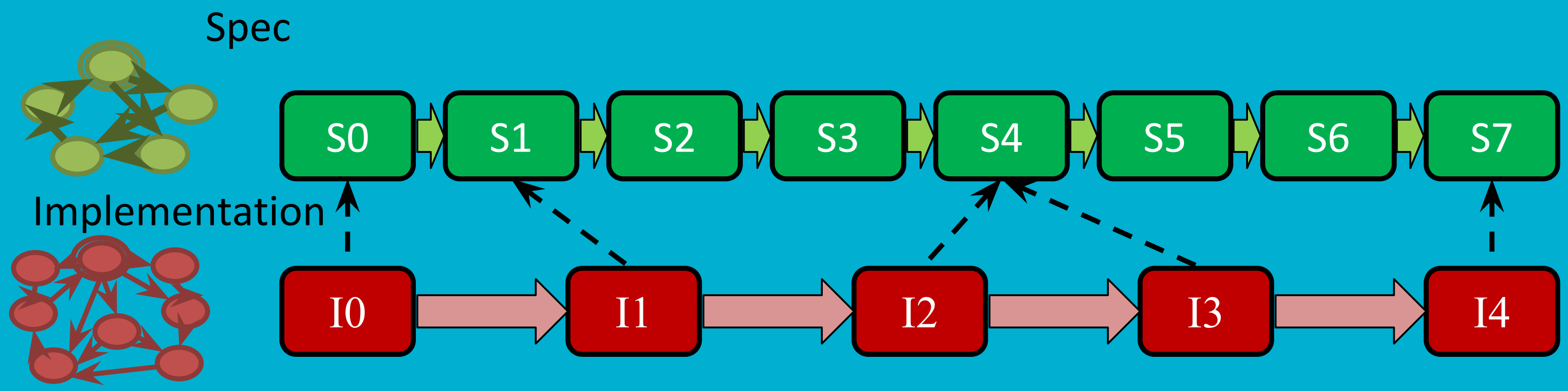
Complexities of implementation

Using efficient data structures

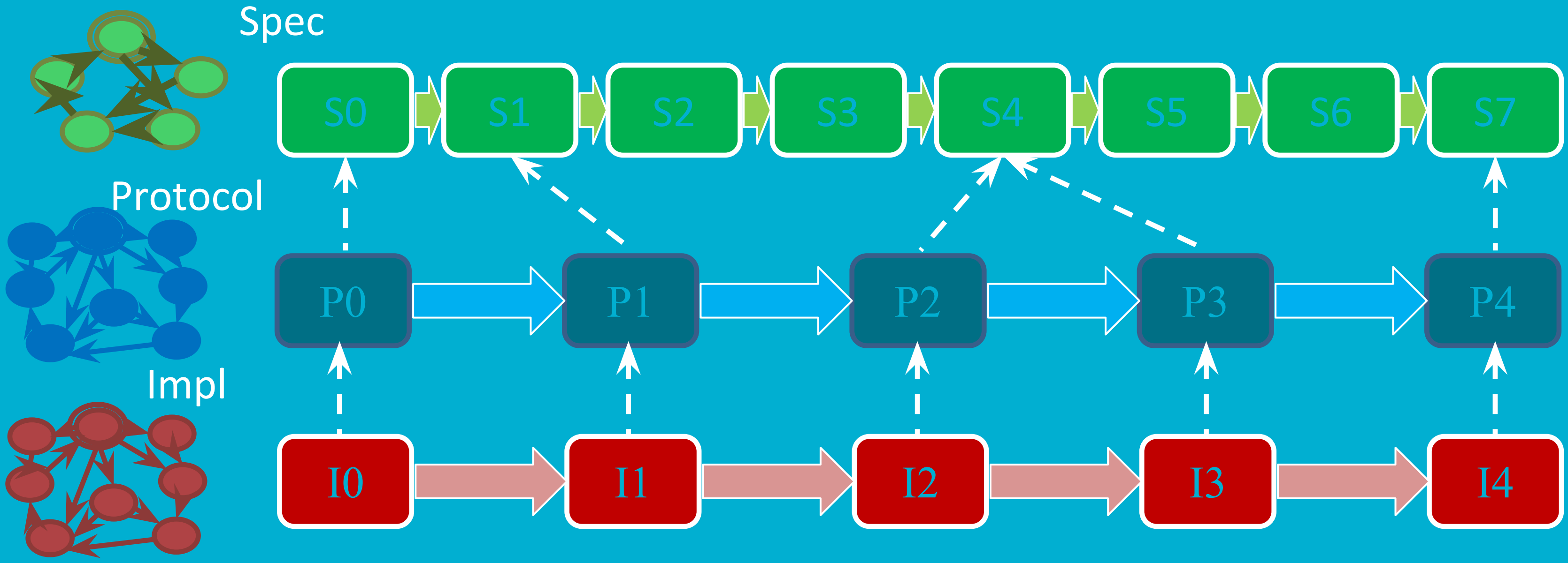
Memory management

Avoiding integer overflow

Refinement



Two Level Refinement



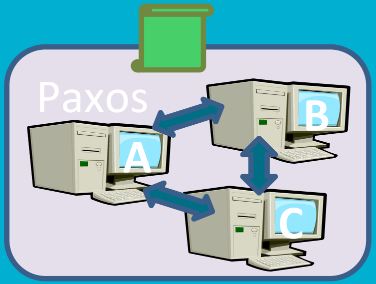
One constructs a liveness proof by finding a chain of conditions

$$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow \ldots \rightarrow C_n$$

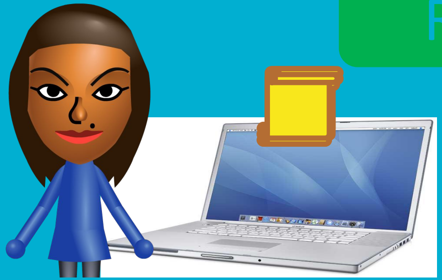
Assumed starting condition

Ultimate goal

Simplified example



- Client sends request
- Replica receives request
- Replica suspects leader
- Leader election starts



Some links can be proven from assumptions about the network

Client sends request

Replica receives request

Replica suspects leader

Leader election starts

Network eventually delivers packets in bounded time

Most links involve reasoning about host actions

Client sends request

Replica has request

Replica suspects leader

Leader election starts

One action that event handler can perform is “become suspicious”

Lamport provides a rule for proving links

$C_i \rightarrow C_{i+1}$

Action

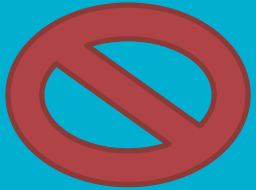
Enablement poses difficulty for automated theorem proving

Tricky things to pro

- *Action* is enabled (can be done) whenever C_i holds
- If *Action* is always enabled it's eventually performed

Always-enabled actions

Handle a client request



If you have a request to handle, handle it; otherwise, do nothing

Always-enabled actions allow a simpler form of Lamport’s rule

$C_i \Rightarrow C_{i+1}$

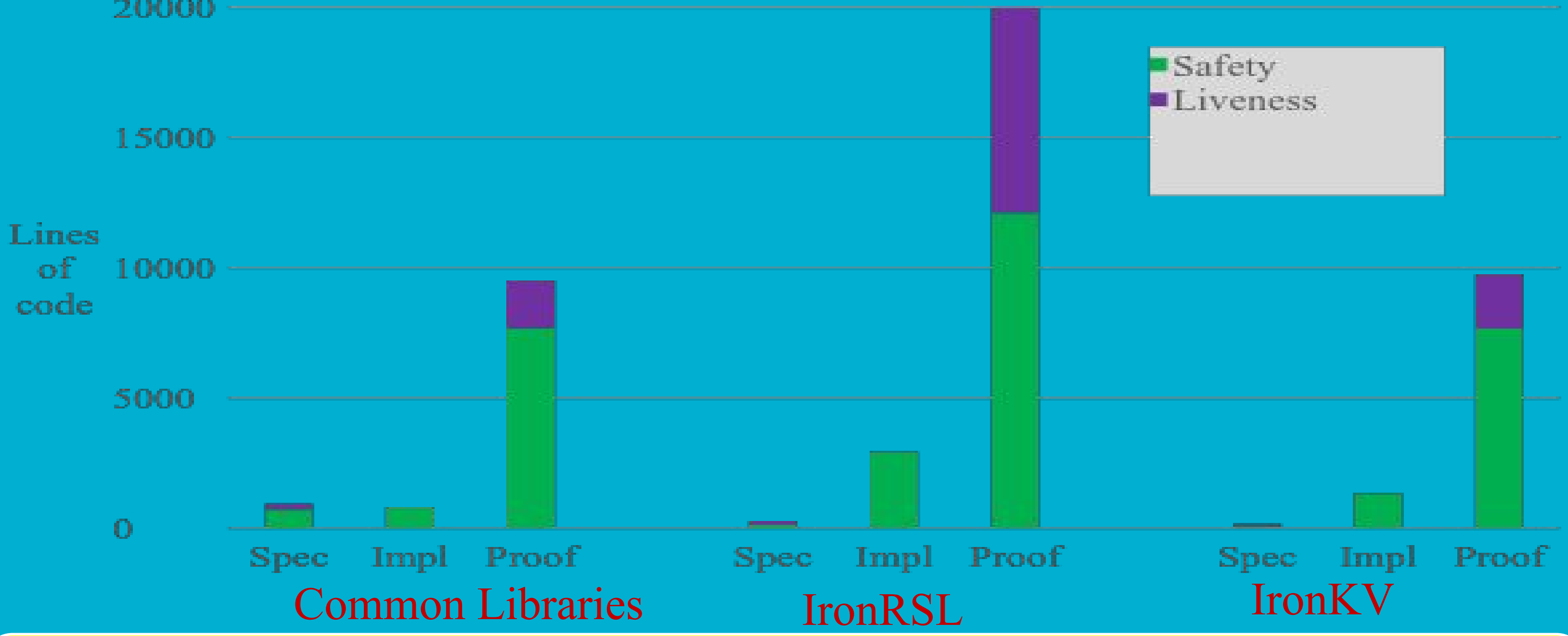
Action

- *Action* is performed infinitely often

Much more in the paper!

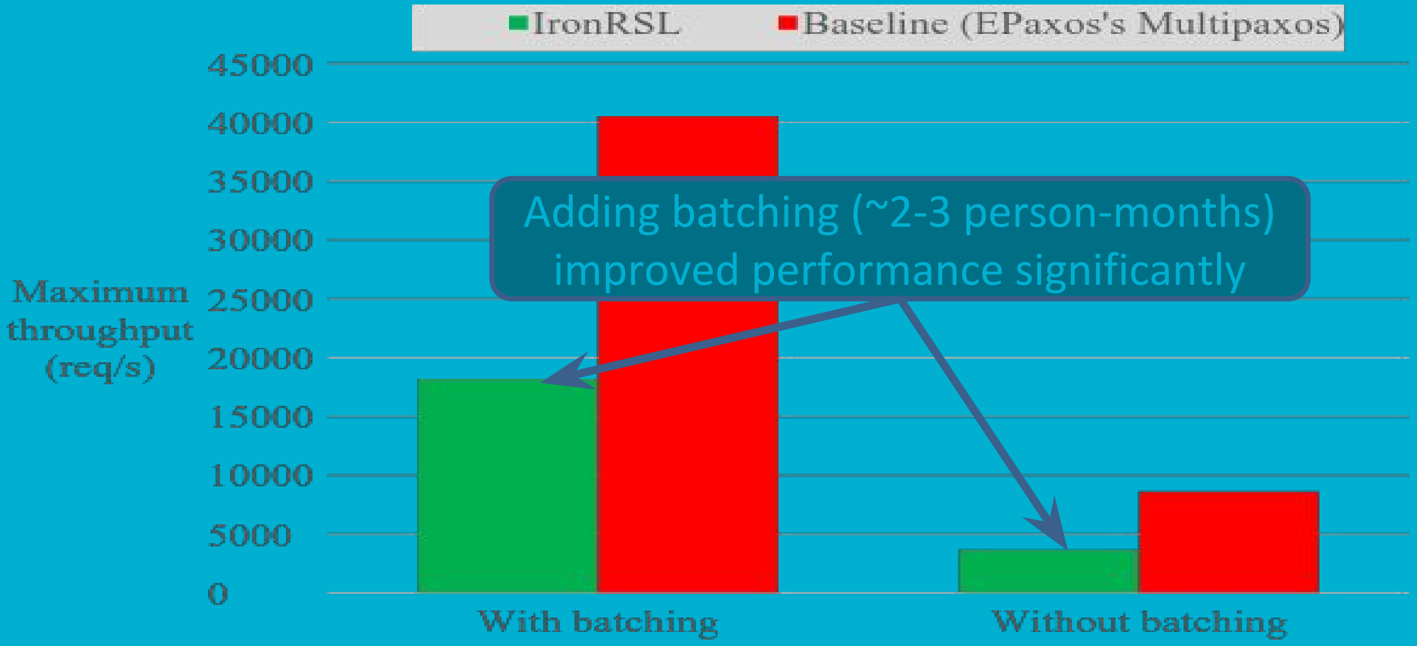
- General Purpose verifying libraries
- Invariant quantifier hiding
- Embedding temporal logic in Dafny
- Reasoning about time
- Strategies for writing imperative code
- Tool improvements

Line counts

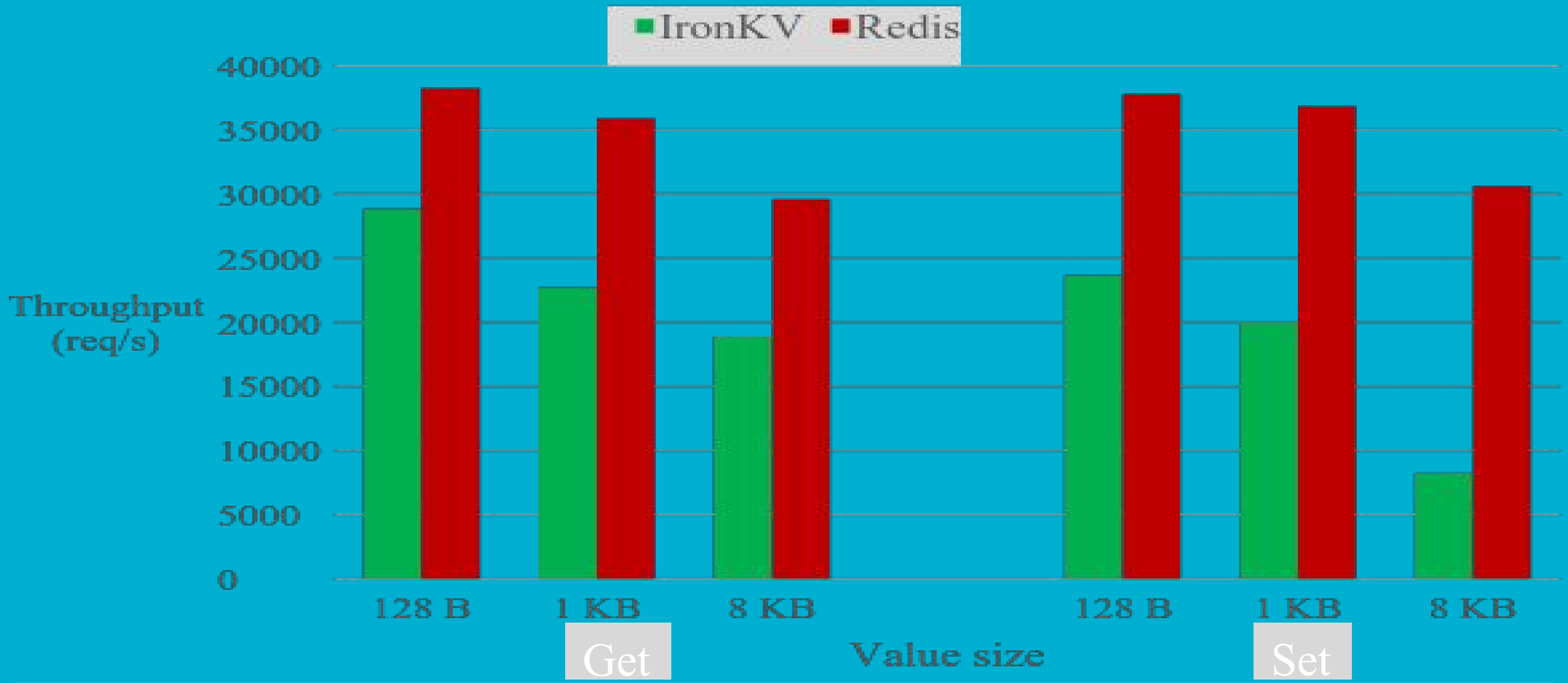


Safety proof-to-code ratio is 5:1
Liveness proof-to-code ratio is 8:1

IronRSL performance



IronKV performance



Throughput 25%-75% of Redis

Conclusions

It’s now possible to build provably correct distributed systems...

...including both safety and liveness properties

...despite implementation complexity necessary for features and performance