15-712: Advanced Operating Systems & Distributed Systems

Active RDMA

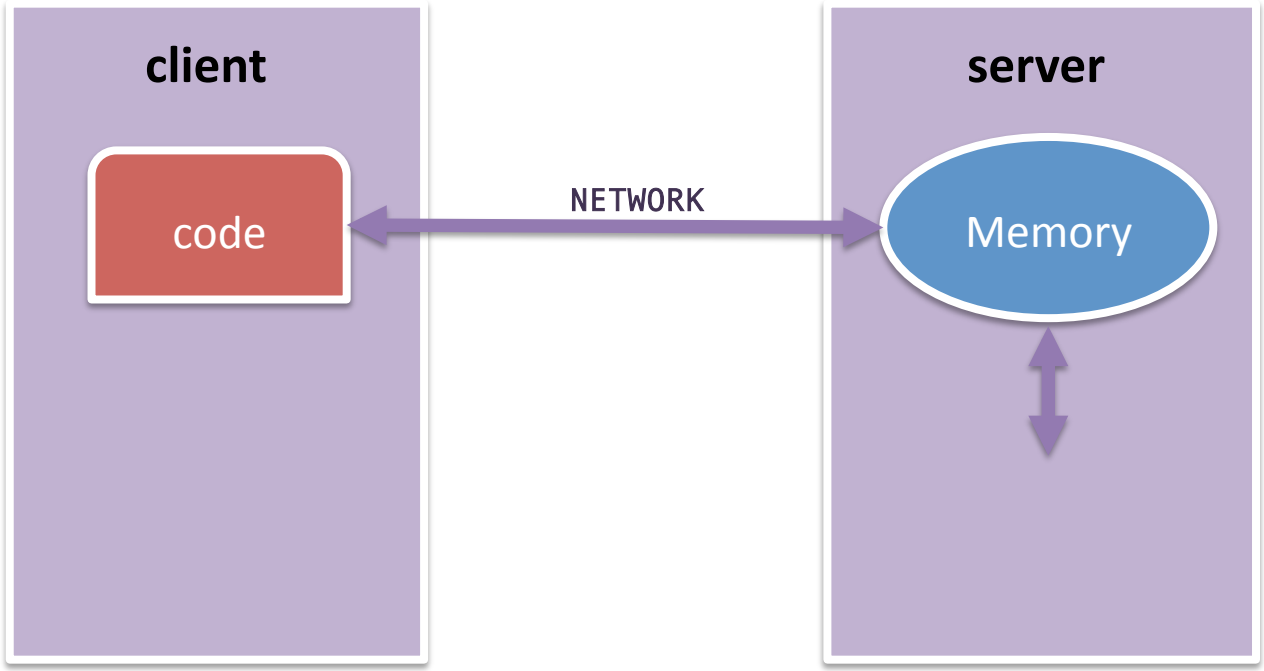
Chris Fallin, Anshul Madan, Filipe Militão

Introduction & Motivation

- Classic *tradeoff* in distributed systems:
  - RPC style semantics:
    - Static interface accessible through RPC calls
    - Calls can be optimized
  - Remote DMA style semantics:
    - Increased flexibility by exposing data structures
    - High separation of data and (running) code

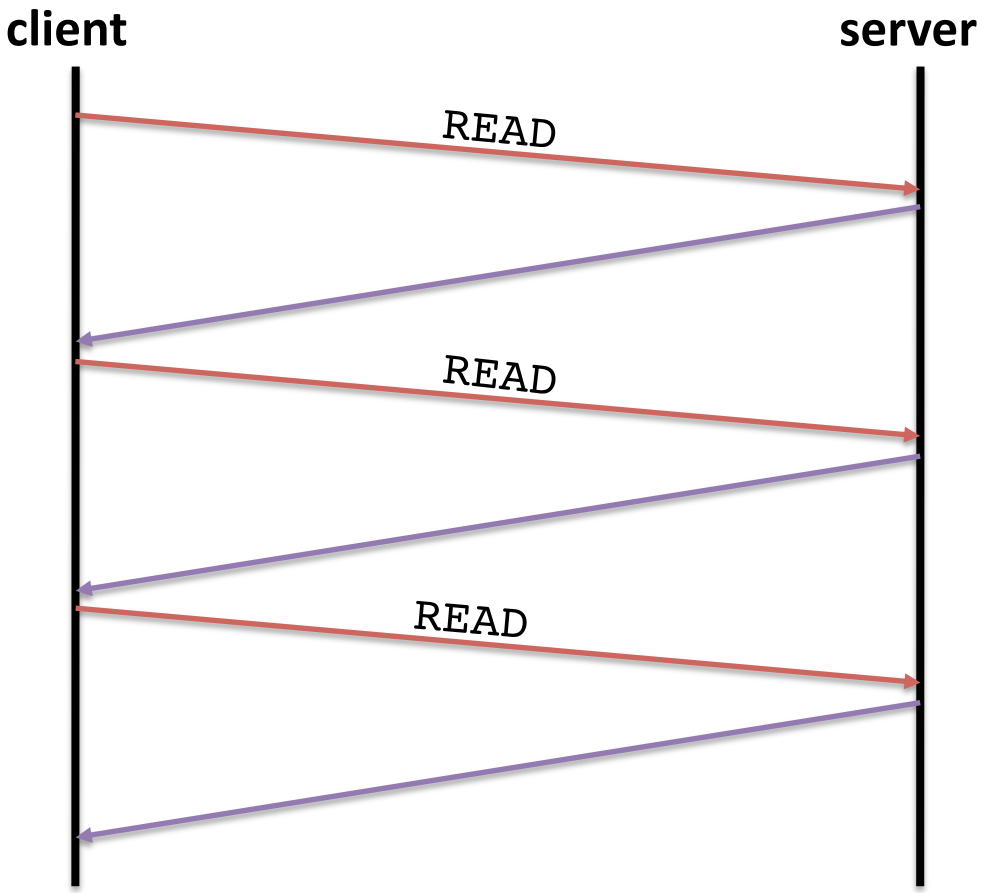
Propose & Prototype

- A middle ground: *Active RDMA*
  - Keep the flexibility of RDMA…
  - … but enable **data-code locality** with code migration (active code).



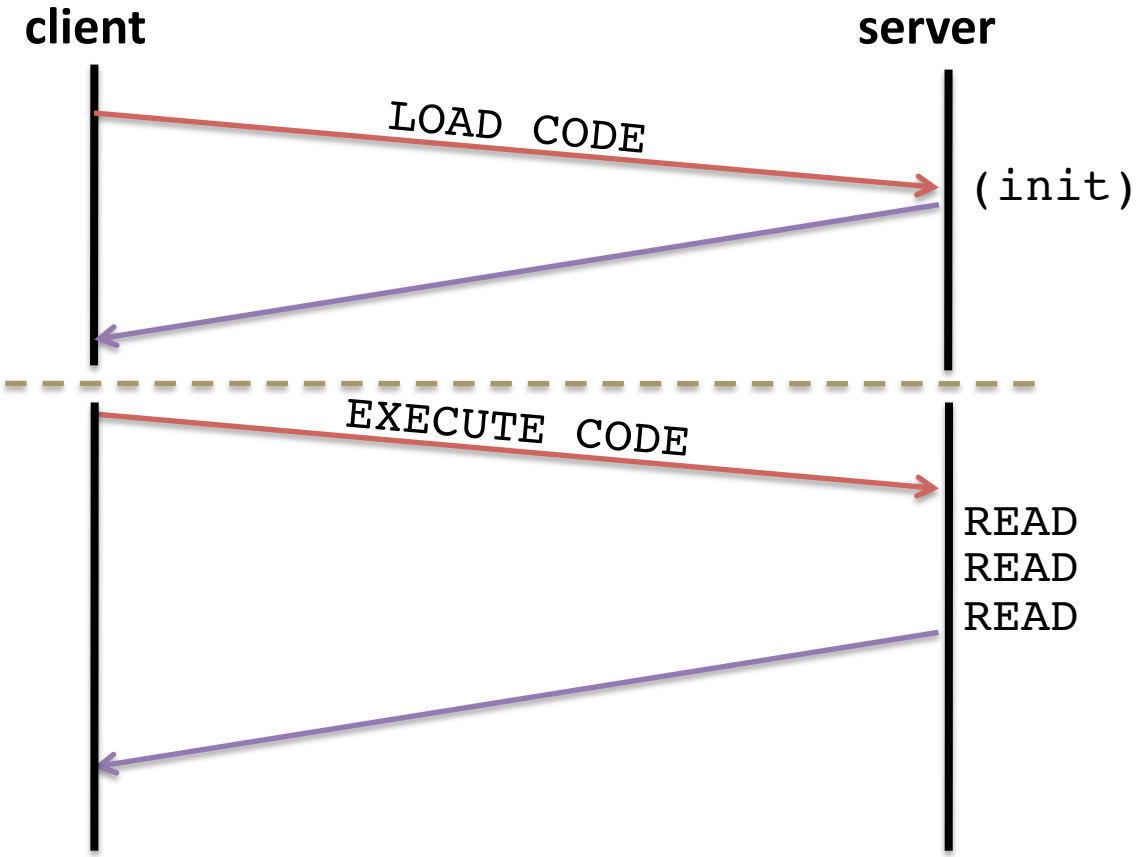
RDMA traffic

Operation: `p->next->next->val`



Active RDMA traffic

Operation: `p->next->next->val`



Contributions

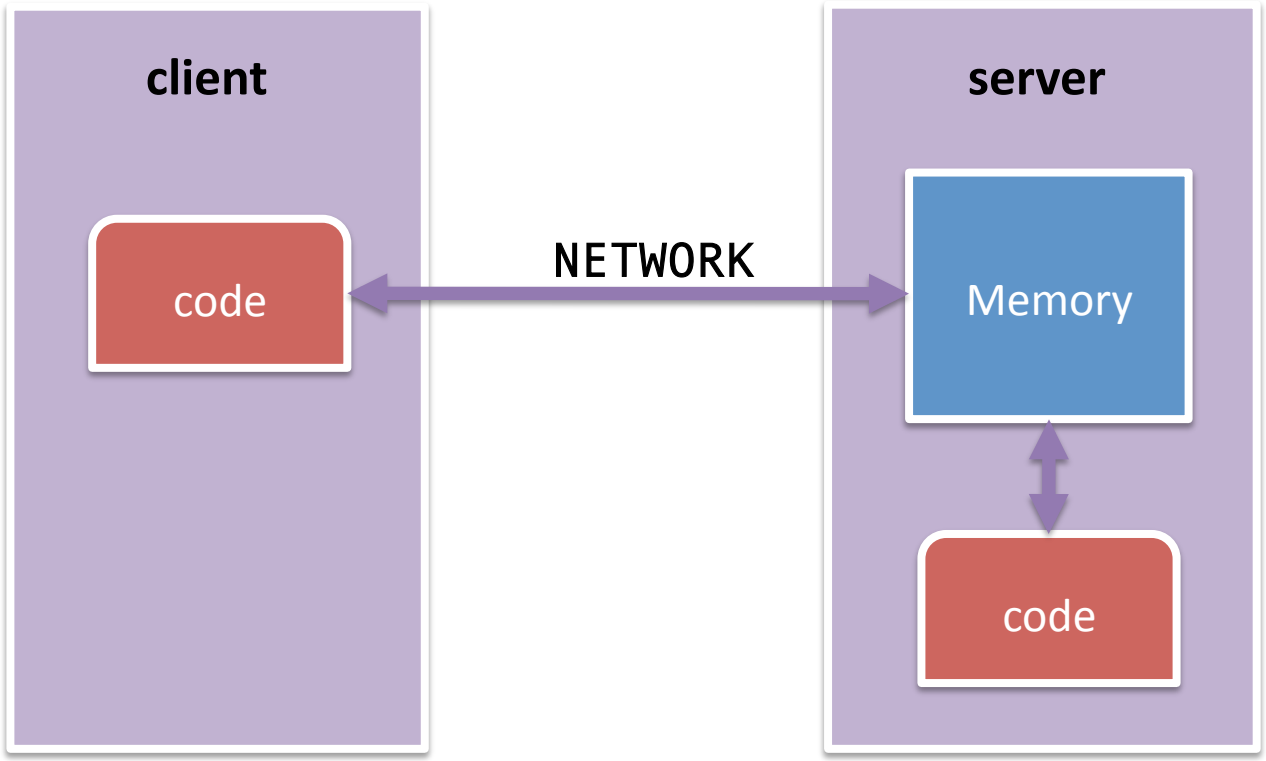
- Run active code directly on the *NIC*
  - low-level interface for remote code
  - reference implementation of ***Active RDMA***
- Simulation infrastructure:

  modified Bochs x86 + Java JVM + timing model
- Initial evaluation of the concept:
  - distributed (in-memory) FS *vs* NFS
  - performance impact: *grep* + *find* + *copy*

Related Work

- *Active Networks*
  - customization at the network/routing level
- *Active Disks*
  - execute code at the disk controller
  - mostly data processing
- *Mobile Code*
  - higher level abstraction
  - sandboxing and security

Active RDMA Architecture



Active RDMA - primitives

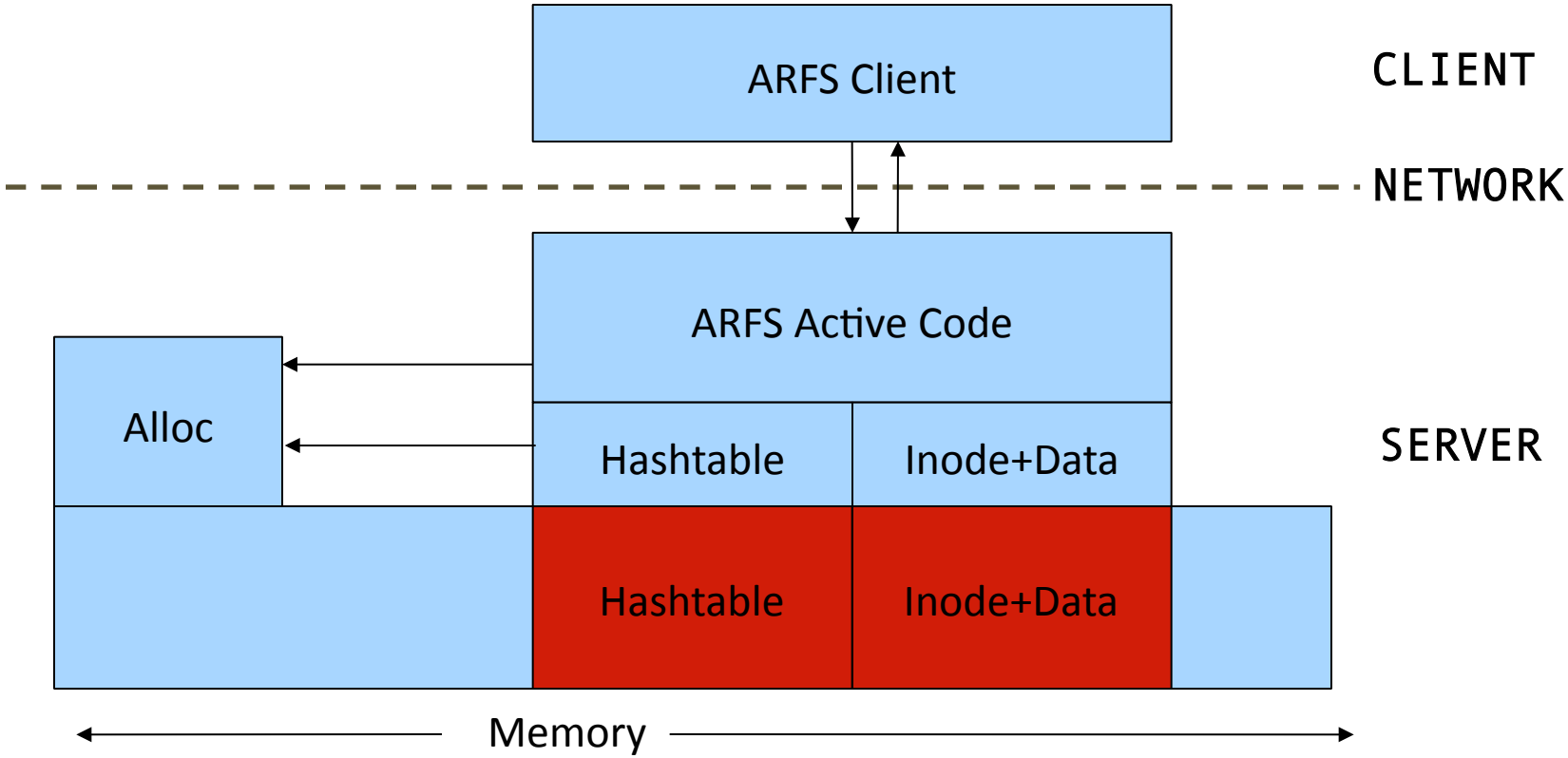
- Existing RDMA primitives like read, write, CAS
- Our extensions:
  - Load (bytecode)
  - Execute(md5, args)

Initial Test Applications

- Linked List
- Locking Service
- Hash table

Algorithm	Operation	Run Time
list - RDMA	put	39.675 ms
list - RDMA	get	19.38 ms
list - Active	put	0.33 ms
list - Active	get	0.245 ms
table - RDMA	put	1.919 ms
table - RDMA	get	1.739 ms
table - Active	put	0.245 ms
table - Active	get	0.239 ms

Active RDMA File System (ARFS)



ARFS Architecture

Uses other Active code as libraries:

- Alloc()
- Hash table : Path → Address
- Inode store : Address → Data blocks
- FS interface :

  Hash table + Inode store

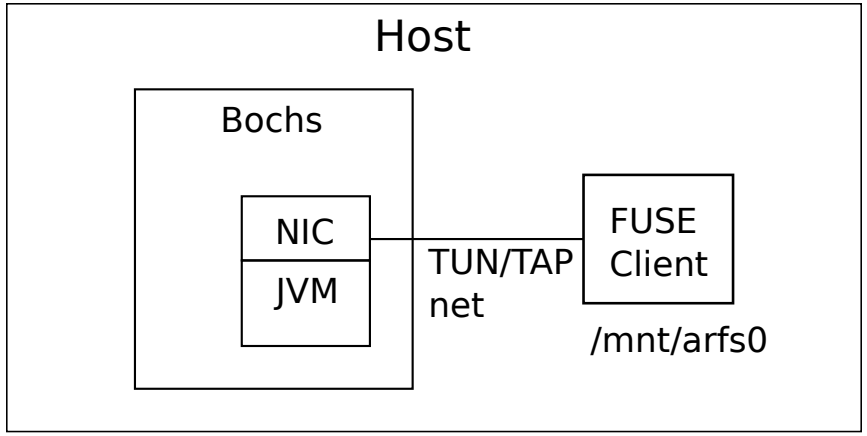
  → hierarchical data store

Evaluation Methodology

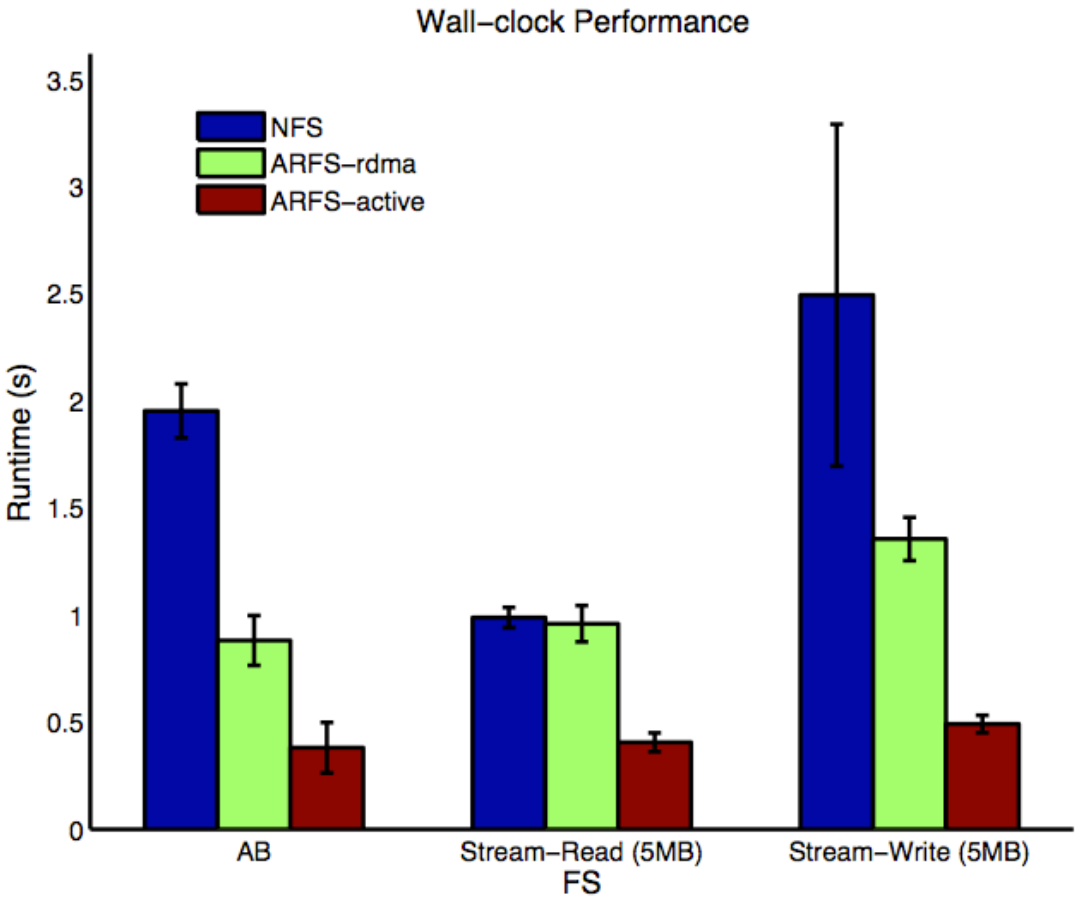
- Active RDMA FS (ARFS) vs. NFS
- ARFS-rdma, ARFS-active variants
- Tests:
  - Andrew Benchmark
  - Stream-Read
  - Stream-Write
- Find, Grep:
  - system utilities
  - active versions

Simulation

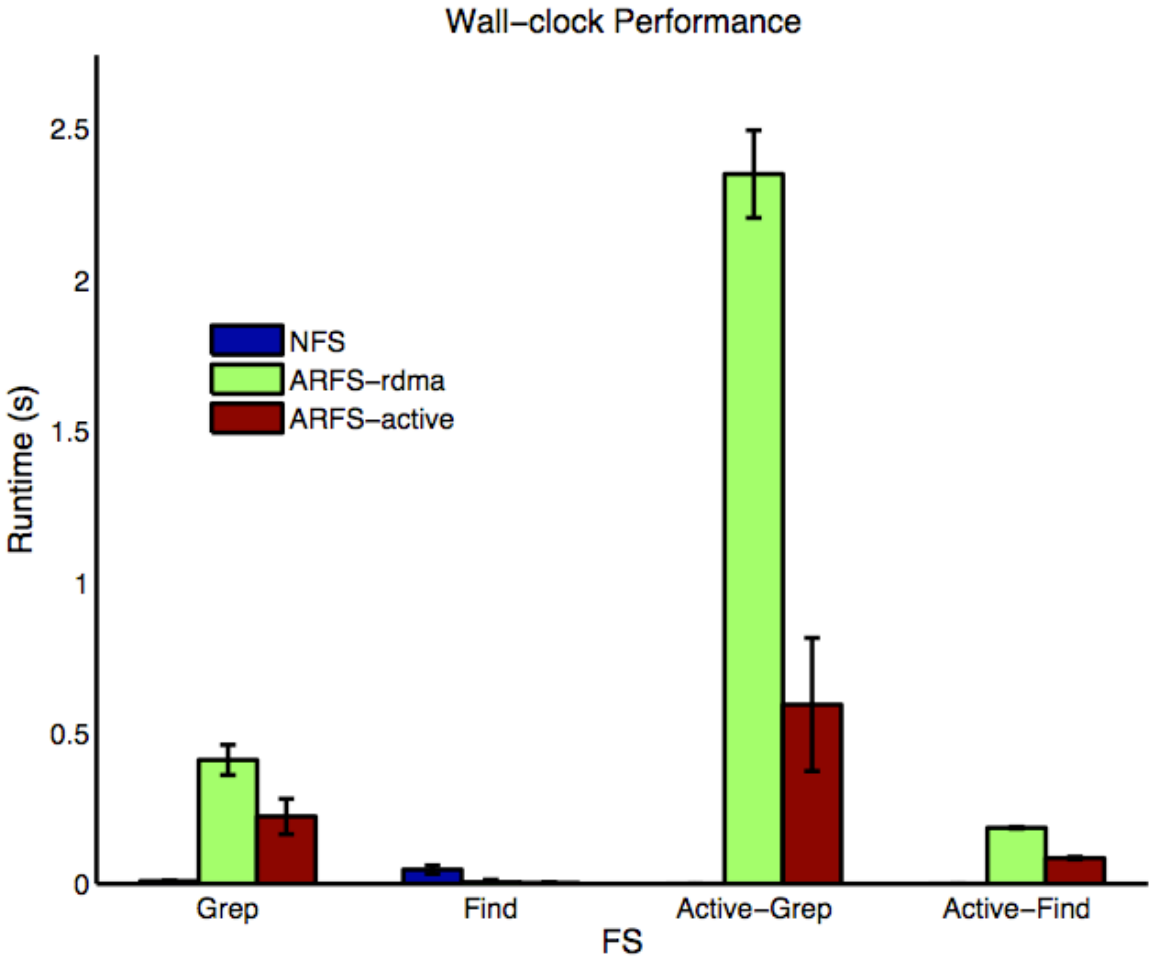
- Bochs full-system simulator
- JVM “coprocessor” in NIC model
- UDP/IP stack in NIC model
- Client: FUSE (userspace)
- Benchmarks run on host
- Wallclock time + synthetic timing statistics



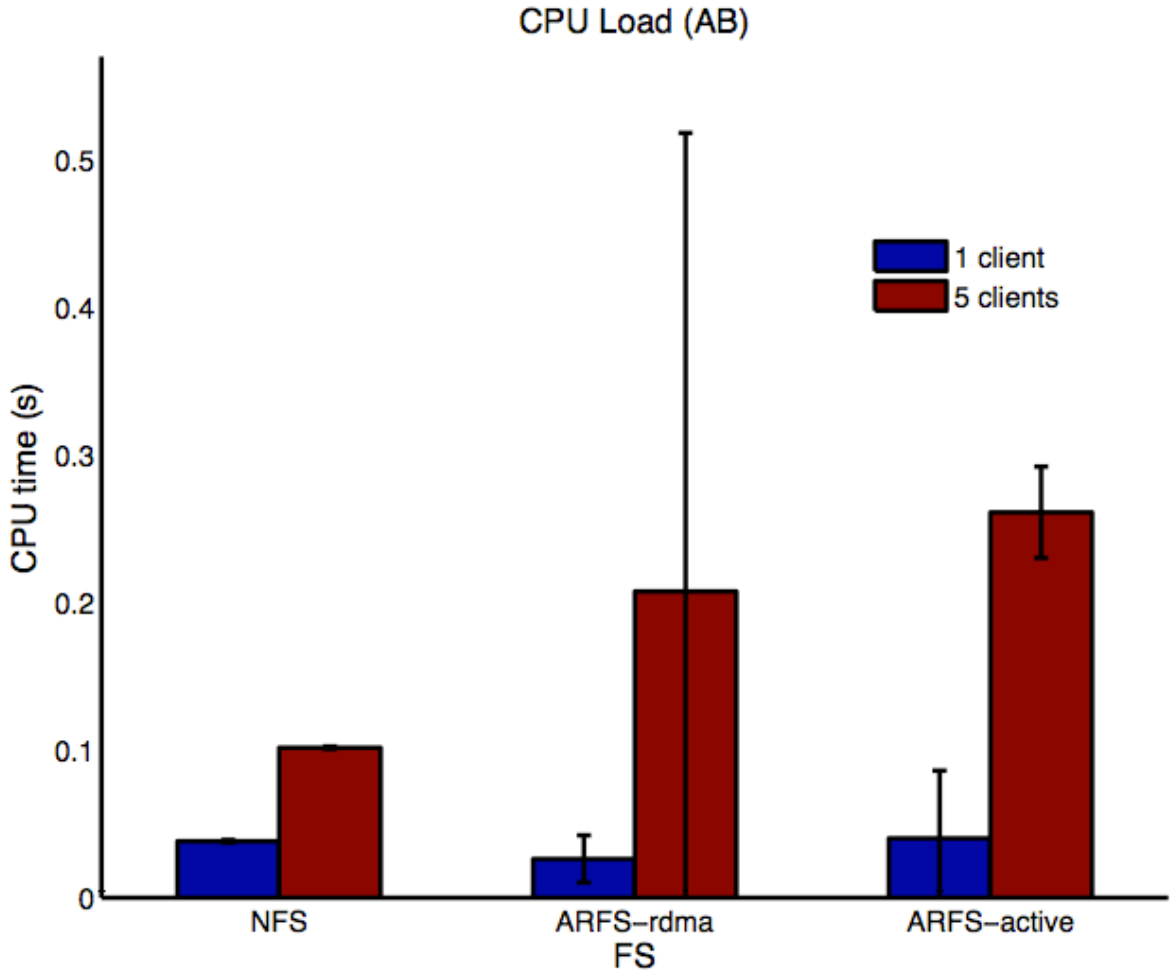
Basic Benchmarks



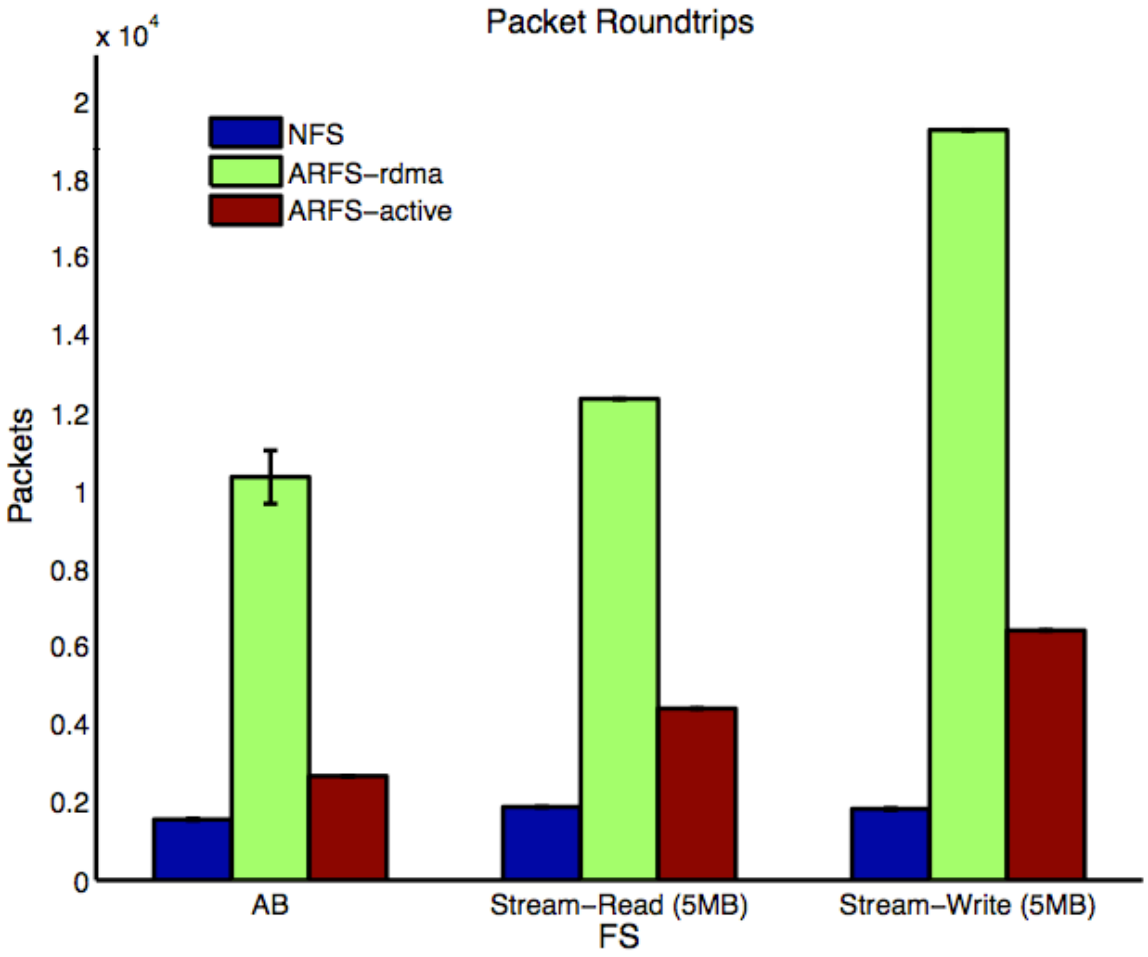
Find and Grep



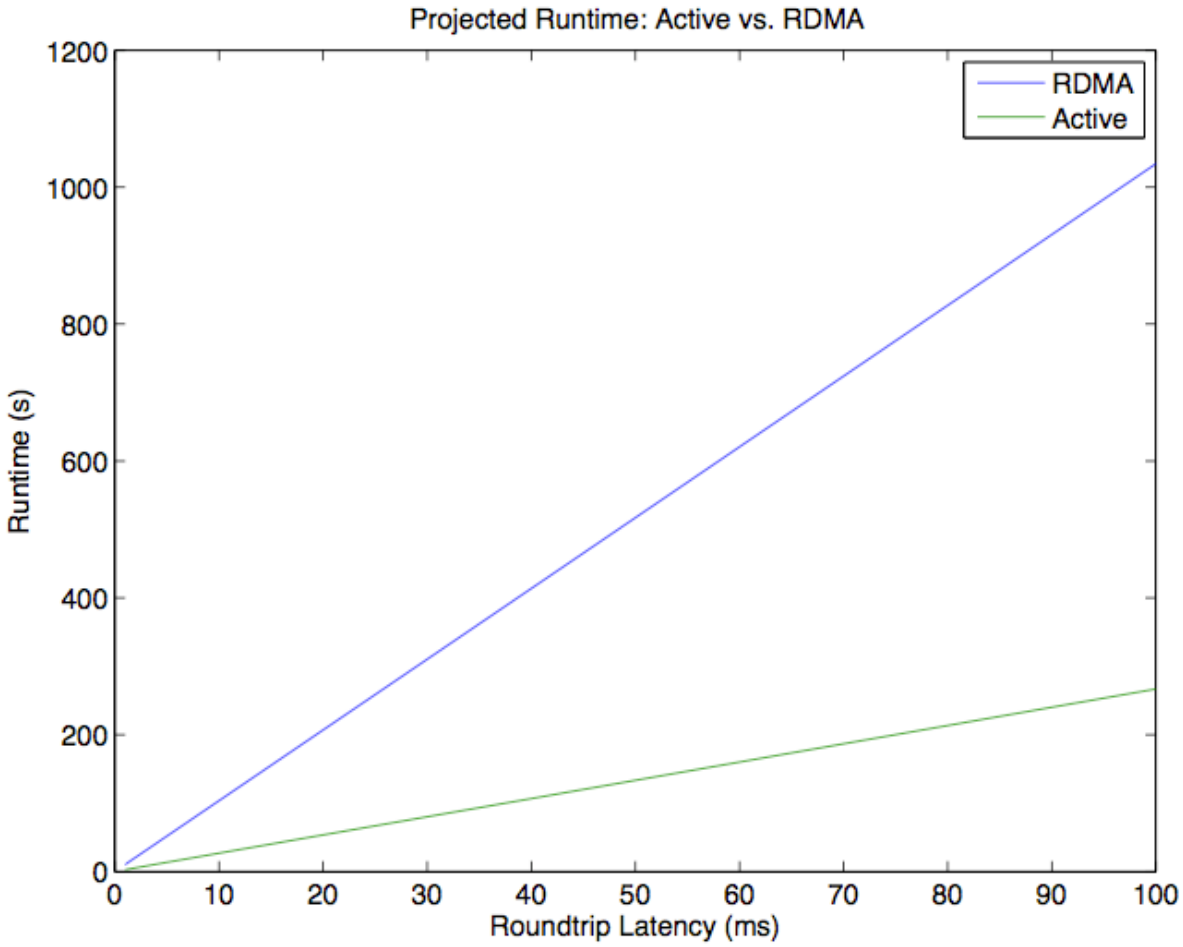
CPU Load Scaling Behavior



Packet Roundtrips



Synthetic Runtime: RDMA vs. Active



Future Work

- Better methodology
  - Acceptable but tractable: pace JVM appropriately, model network delay, run client in separate Bochs
  - Better: Extend cycle-accurate simulated domain to clients and network, build cycle-accurate model of active code coprocessor (real ISA, cache coherence with main CPU, …)
  - Best: prototype in a real system
    - Active code (i) in server kernel or (ii) on real NIC (FPGA?)
    - Clients with in-kernel filesystem

Future Work II

- Implementation: more robust (optimized IO path, robust error handling, …)
- Server-side architecture:
  - Sync with main host CPU (at least for on-disk FS)
  - consider tradeoffs between coprocessor and main processor
- Dynamic active code generation
  - Grep: regex compiler? (after basic optimizations)
  - More complex data queries, custom adapters, …

Summary & Conclusion

- Designed Active RDMA interface and presented reference implementation
- Built proof-of-concept in-memory network filesystem
- Evaluated performance in simulation
- Showed a few active code-specific applications

- Lessons learned:
  - Getting API right is hard (underwent a few changes)
  - Define basic services, build in layers
  - Getting accurate simulation/timing for client/server setups is *HARD*
- Underwhelming prototype performance, but interesting possibilities for future