Arachne:
Core-Aware Thread Management

Henry Qin, Qian Li, Jacqueline Speiser,
Peter Kraft, John Ousterhout

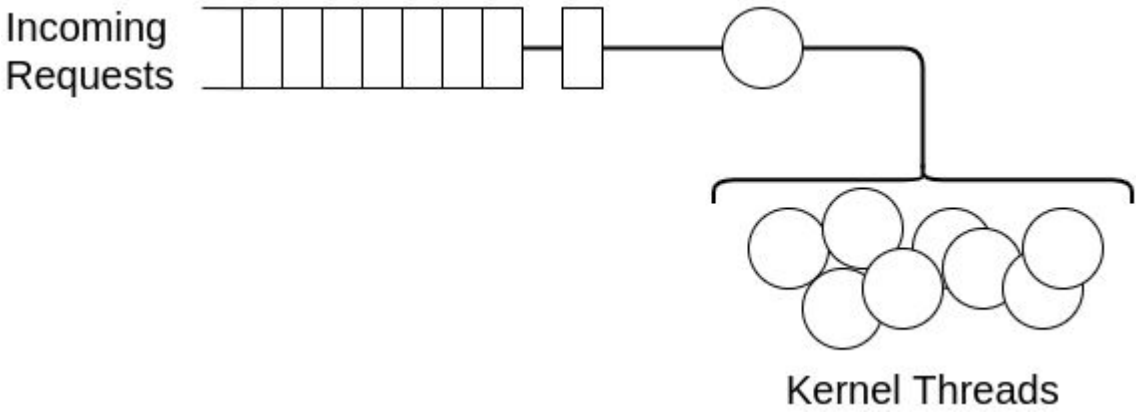
Latency and Throughput

- Low latency services
  - Memcached
  - RAMCloud
  - Redis
- Response times around 5μs or lower

- Low Latency ⇒ Poor Resource Management ⇒ Low Throughput

Problem: Inefficient Kernel Threads

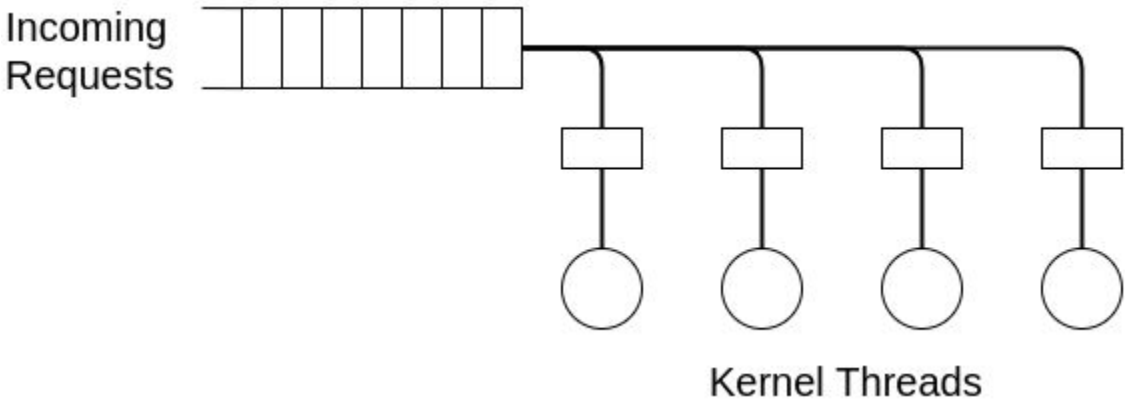
One kernel thread per request. Too slow!



Problem: Matching Parallelism to Resources

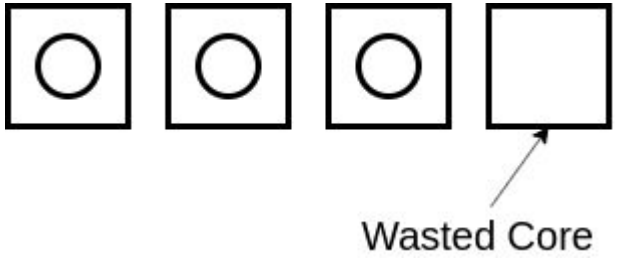
Multiplex requests across long-lived kernel threads.

How many threads? Goal: # of threads = # of cores. Don't know # of allocated cores!

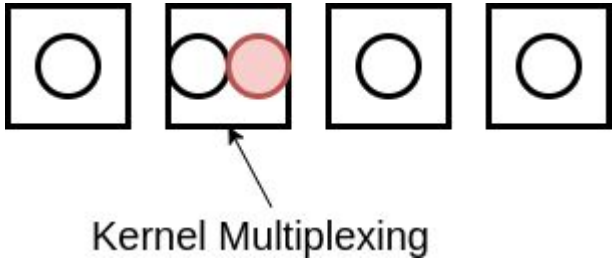


Problem: Matching Parallelism to Resources

Too Few Threads

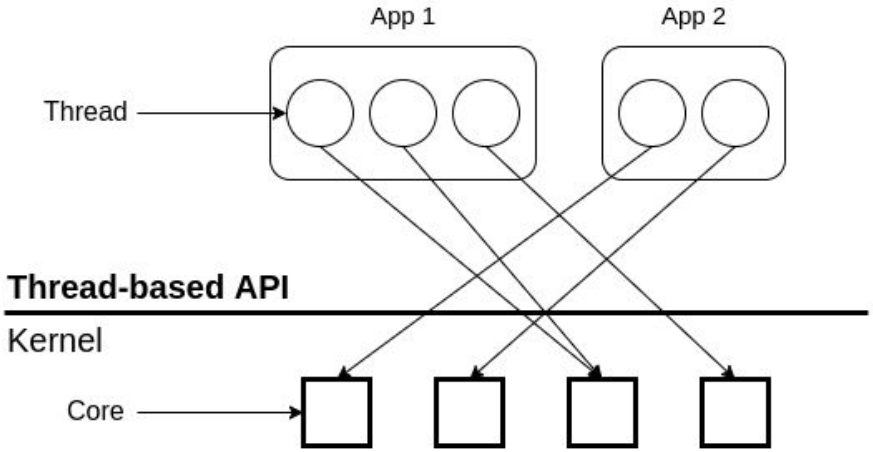


Too Many Threads

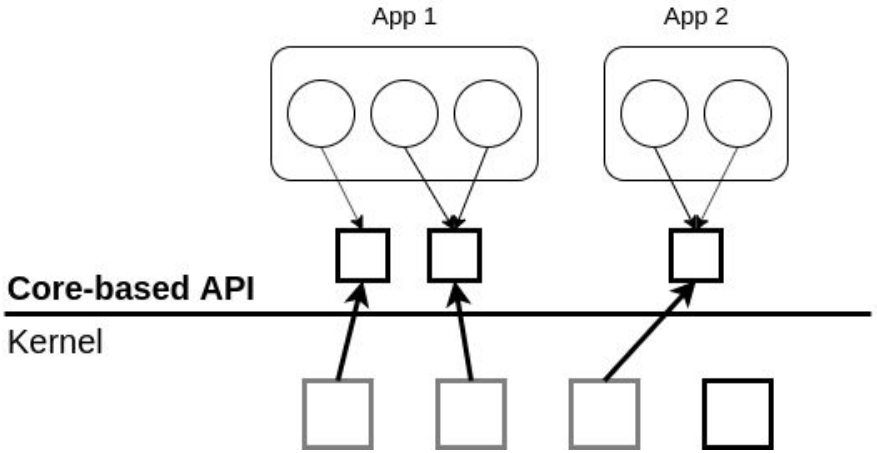


Arachne: Core Aware Thread Management

Now: applications lack control over cores



Arachne: applications are aware of cores



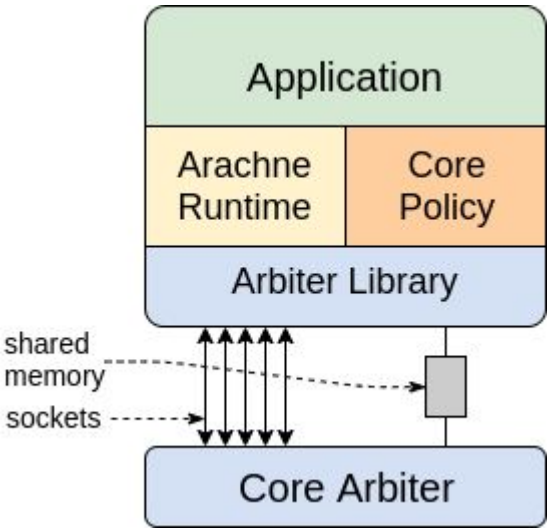
Arachne: Core Aware Thread Management

- Applications have more control over cores
  - Application requests cores, not threads
  - Application knows the exact cores it owns
  - Application has exclusive use of cores

- Thread management is moved to userspace
  - Fast threading primitives: 100 - 300 ns

Architecture Overview

- Core arbiter
  - Allocates cores to applications
- Arachne Runtime
  - Implements threading primitives
- Core Policy
  - Determines thread placement
  - Manages core estimation

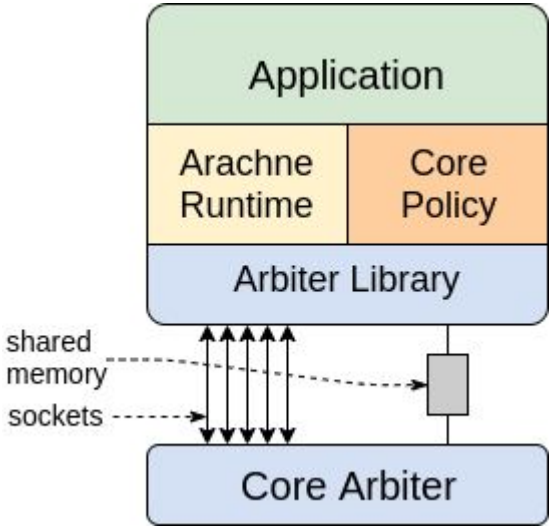


Core Arbiter

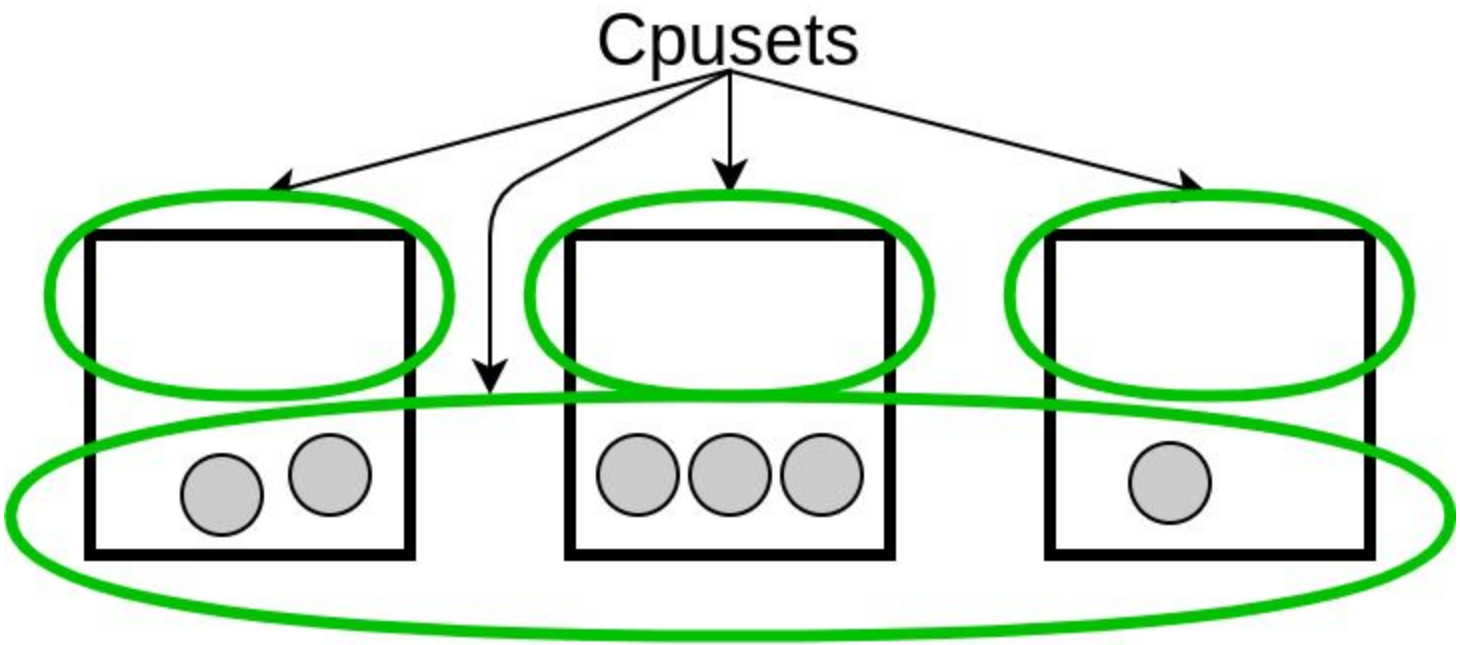
Core Arbiter

- Divides cores into two groups
  - Managed cores
  - Unmanaged cores

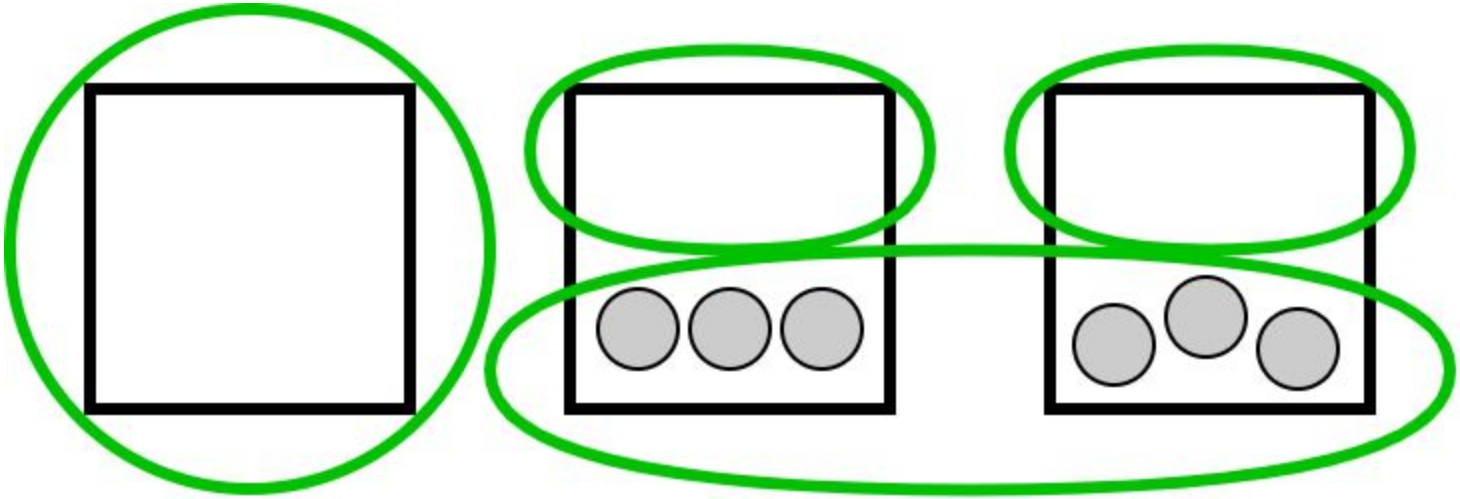
- Uses Linux cpuset mechanism
  - A cpuset is a collection of cores and memory banks
  - Each kernel thread can be assigned to exactly one cpuset
  - Thread executes only on cores in its cpuset



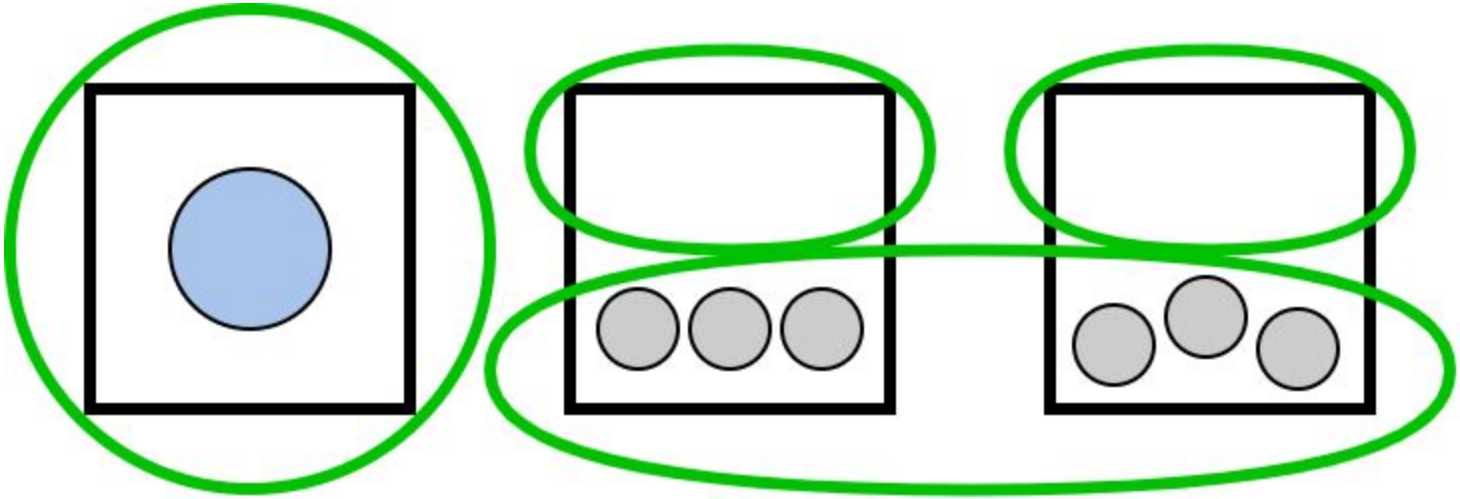
Arbiter Startup



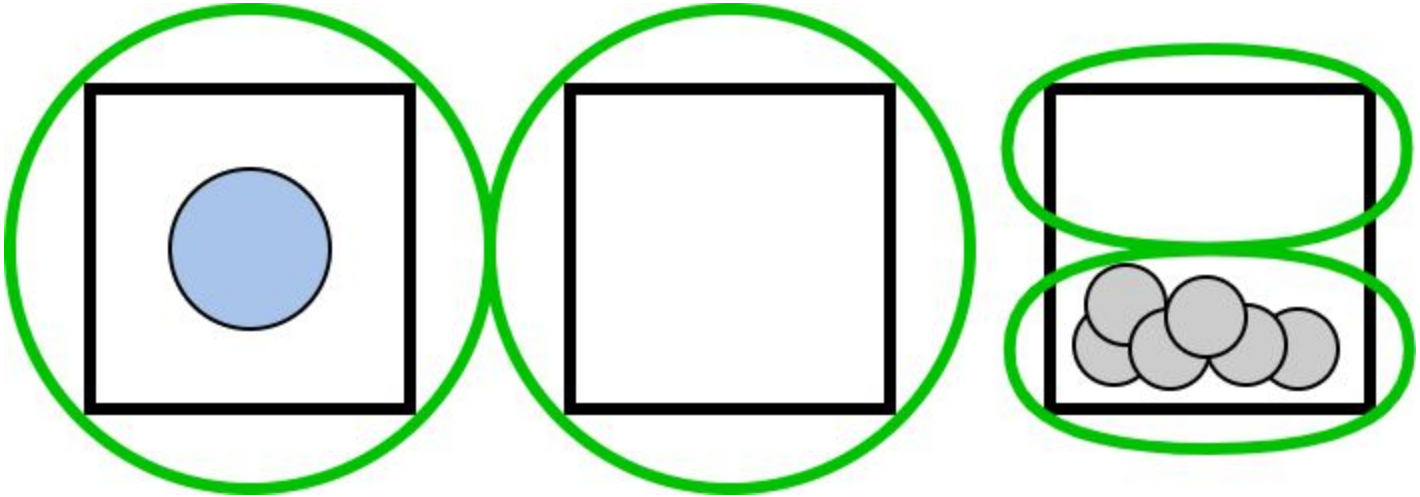
Granting a Core



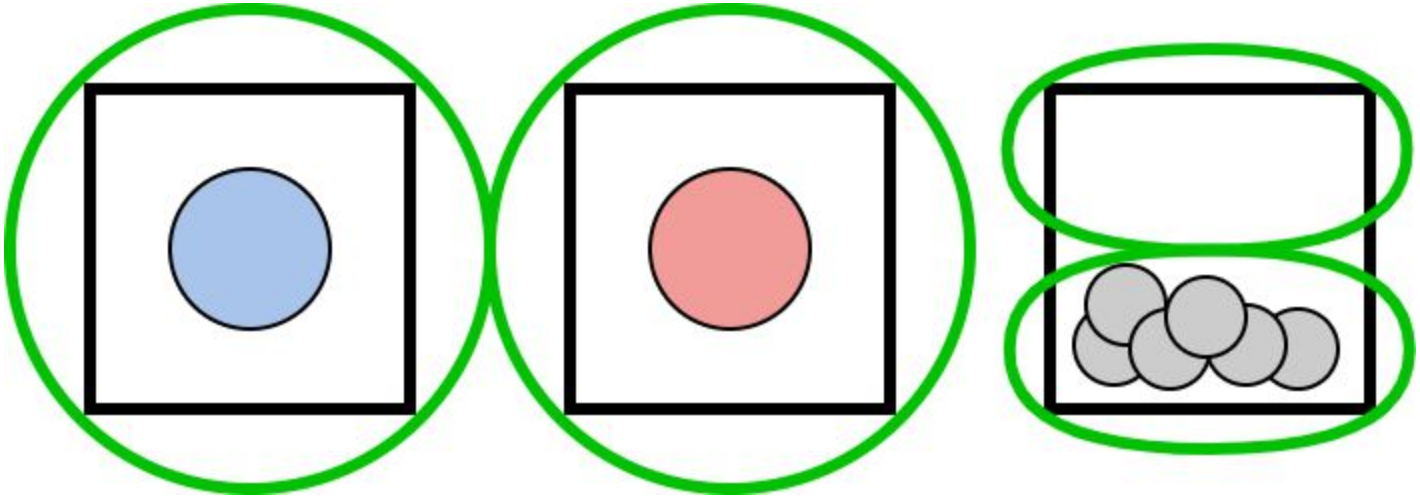
Granting a Core



Granting a Core



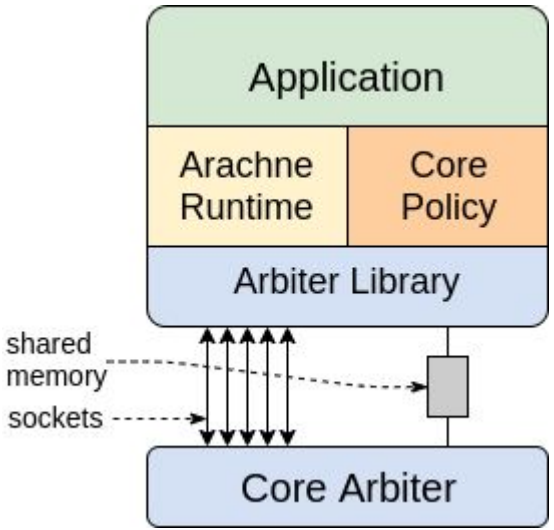
Granting a Core



Communication with Core Arbiter

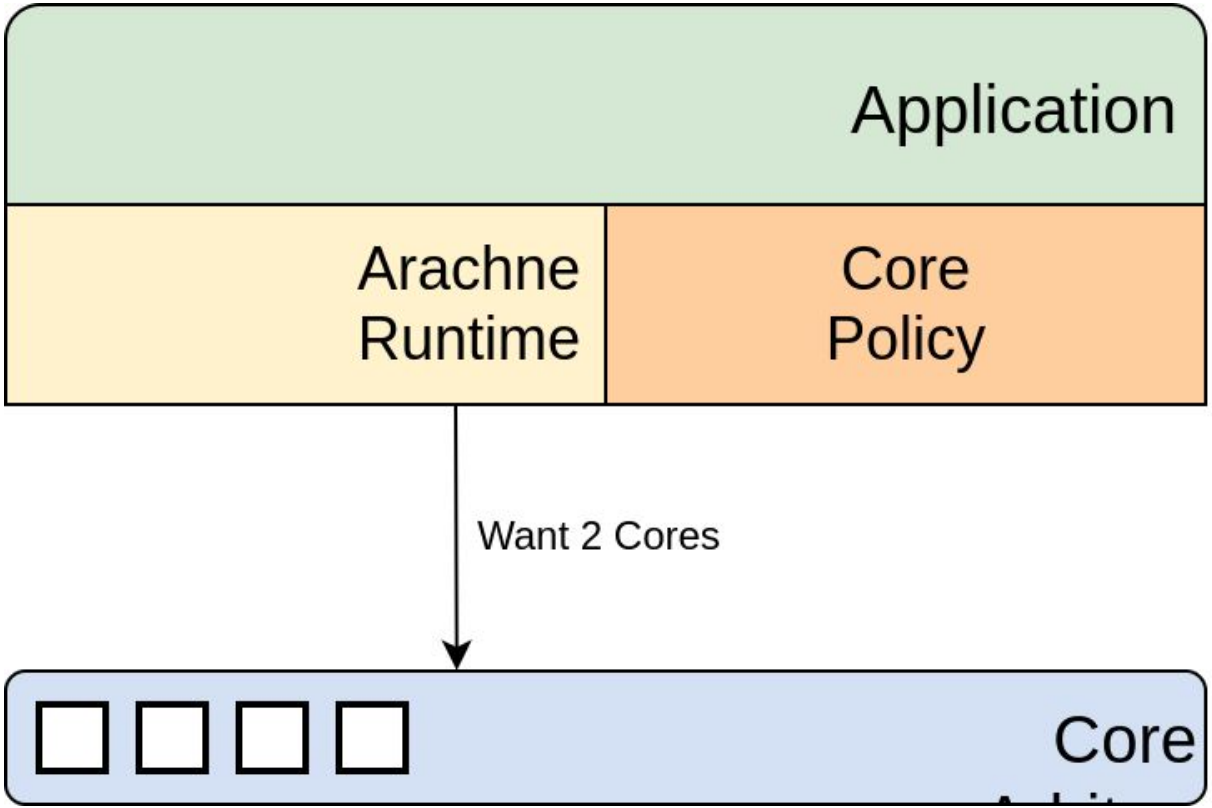
- Arbiter library provides methods for communication with the core arbiter

- Communication is achieved through
  - Unix sockets
  - Shared memory

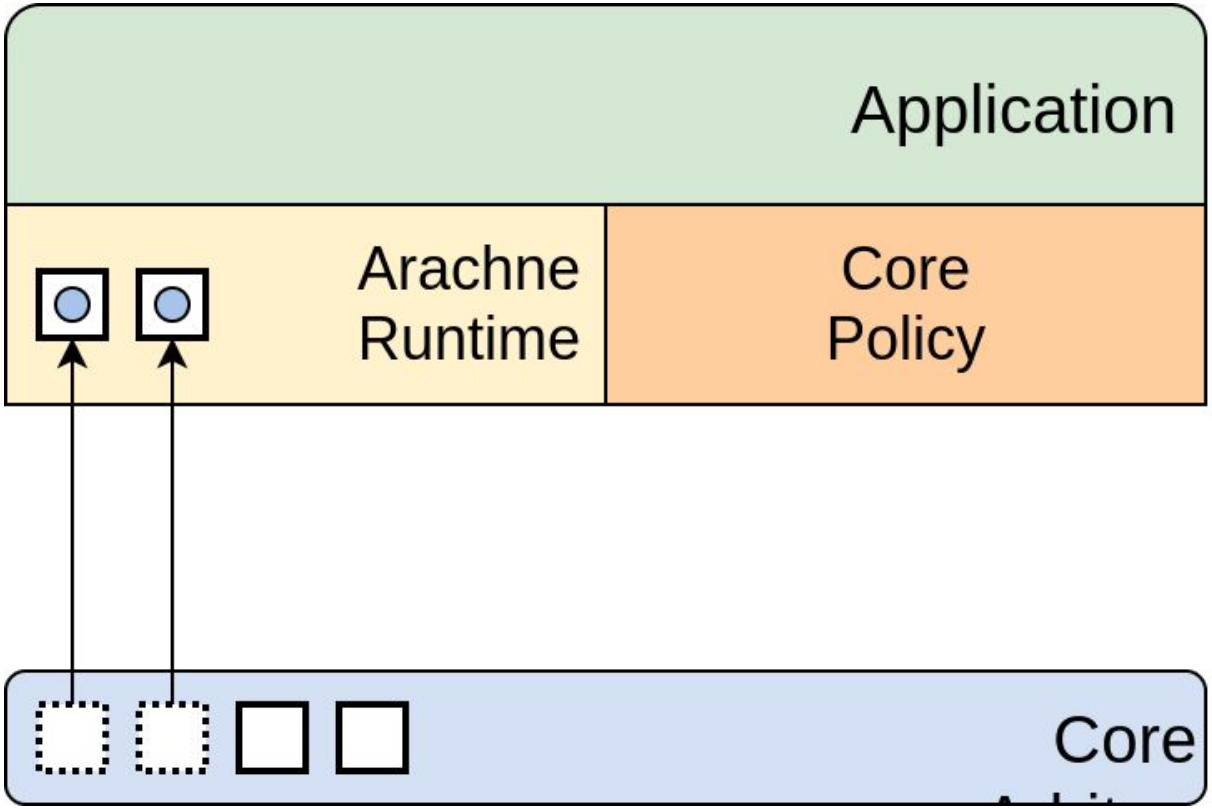


Arachne Application Lifecycle

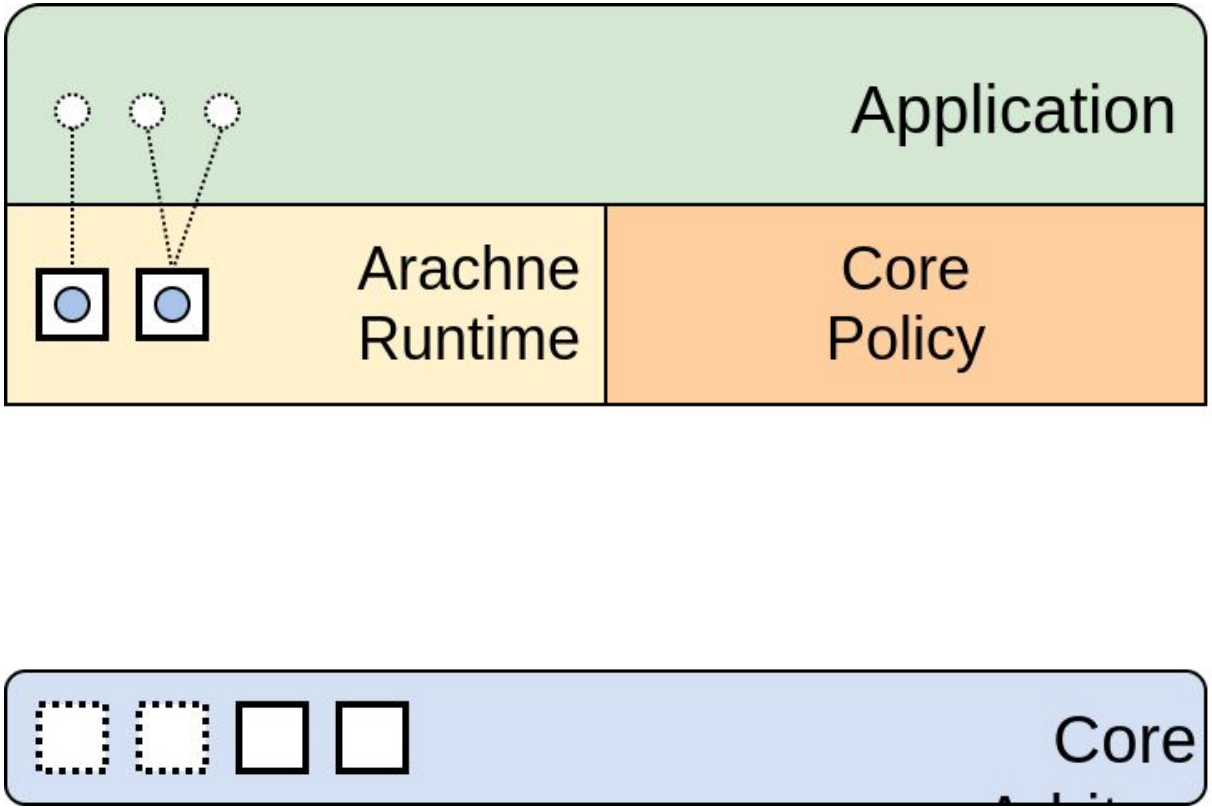
Application Startup



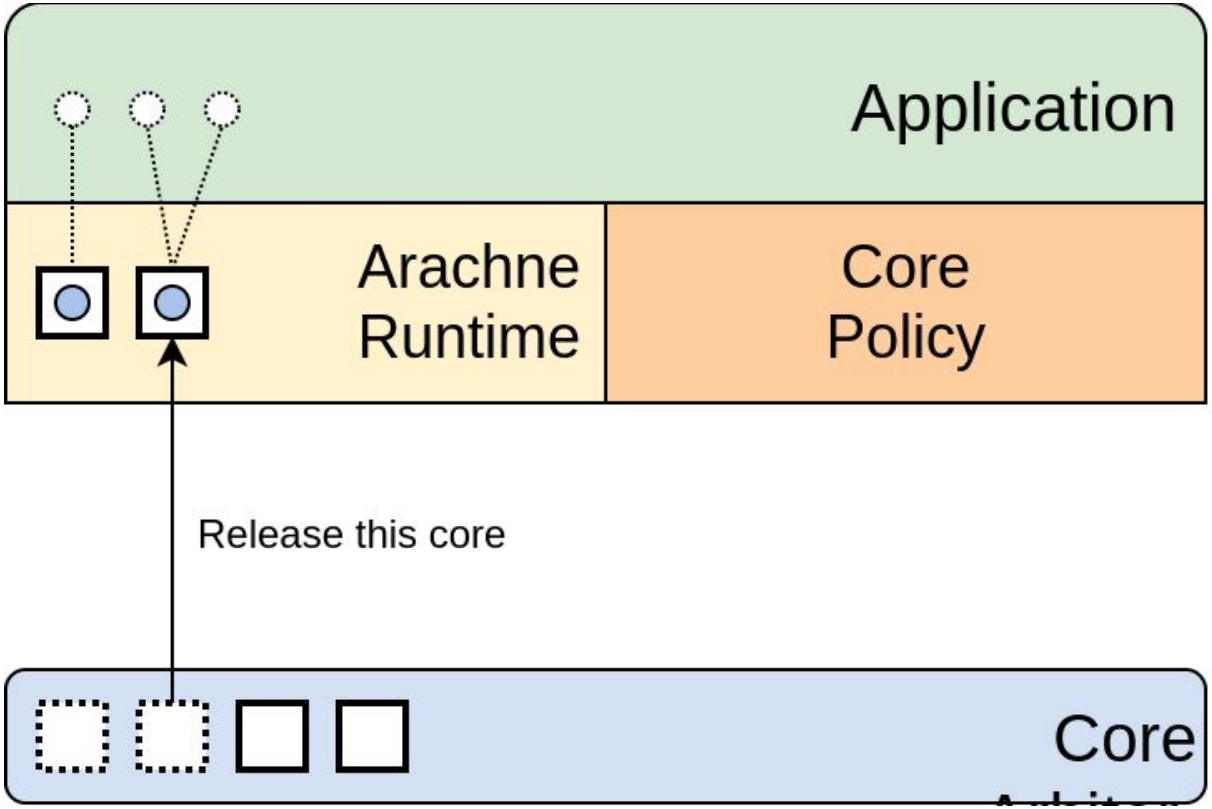
Application Startup



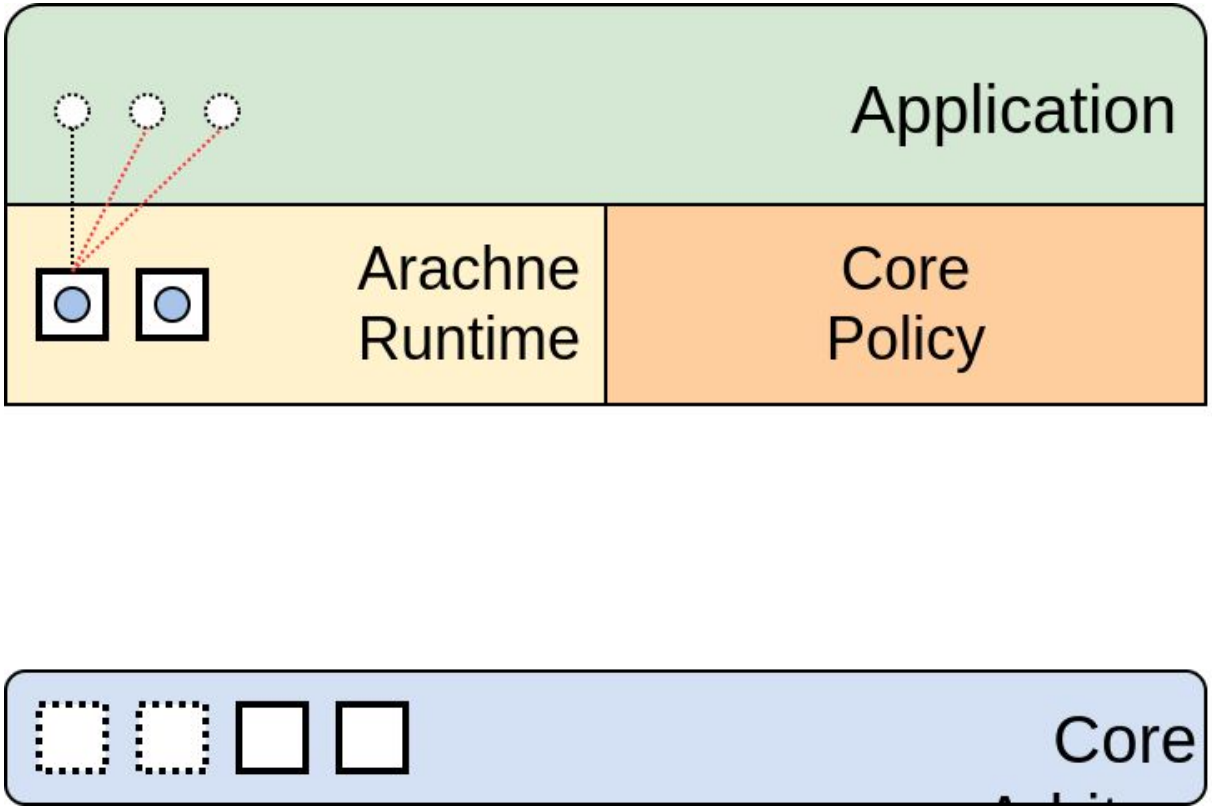
Multiplex User Threads



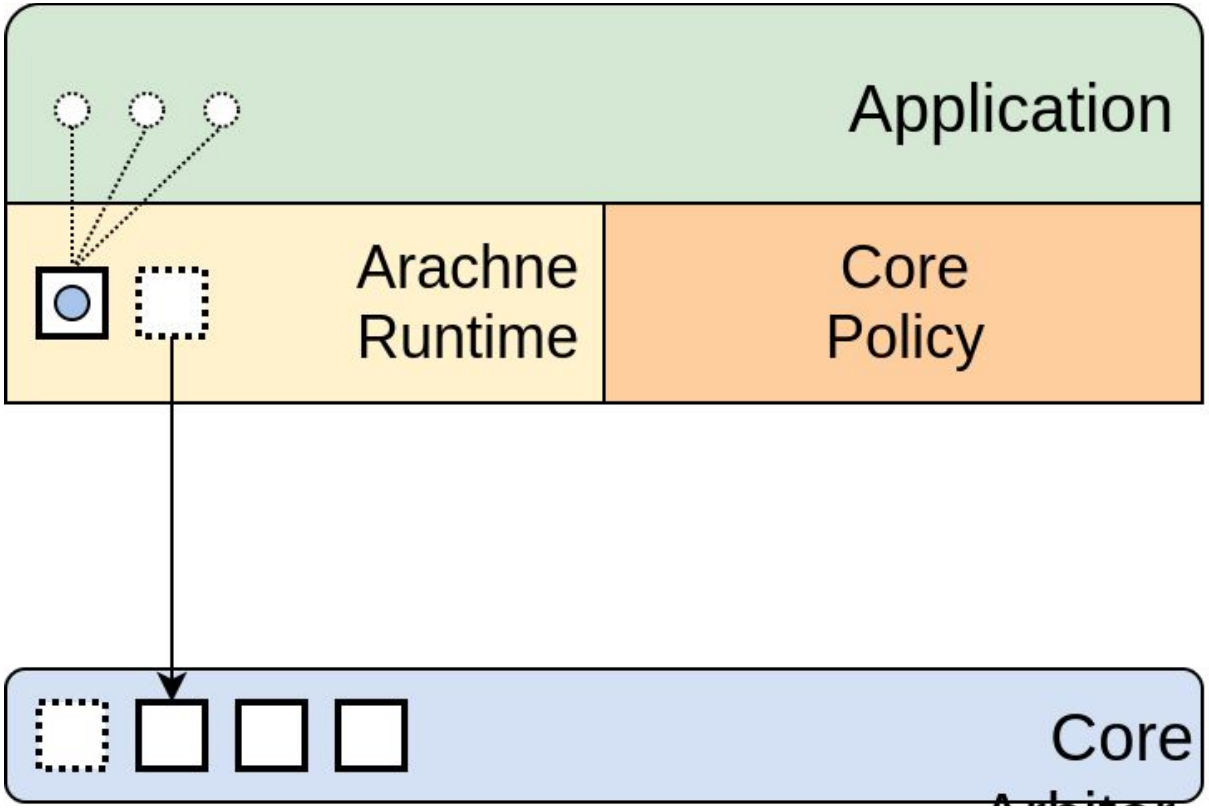
Core Preemption



User Thread Migration



Core Preemption Respected



Arachne Runtime: Cache-Optimized Design

Arachne Runtime: Cache-Optimized Design

- Threading performance dominated by cache operations
- Arachne runtime designed around cache as bottleneck
  - Eliminate cache misses where possible
  - Overlap unavoidable cache misses

- Concurrent misses
- No run queues; dispatcher runs context Runnable flags

Core Policy

Core Policy

- User thread placement
  - Each user thread is assigned one *thread class* by application
  - Based on a thread class acceptable cores are selected for a thread

- Core estimation
  - Two statistics are measured for each core
    - Utilization
    - Load factor
  - Based on statistics a number of required cores is calculated

Evaluation

Evaluation

- Configuration
  - CPU: Xeon D-1548 - 8 cores (16 HT) @ 2.0 GHz
  - RAM: 64 GB DDR4-2133 @ 2400 Mhz
  - Disk: Toshiba THNSN5256GPU7 (256 GB)
  - NIC: Dual-port Mellanox ConnectX-3 10 Gb
  - Switches: HPE Moonshot-45XGc

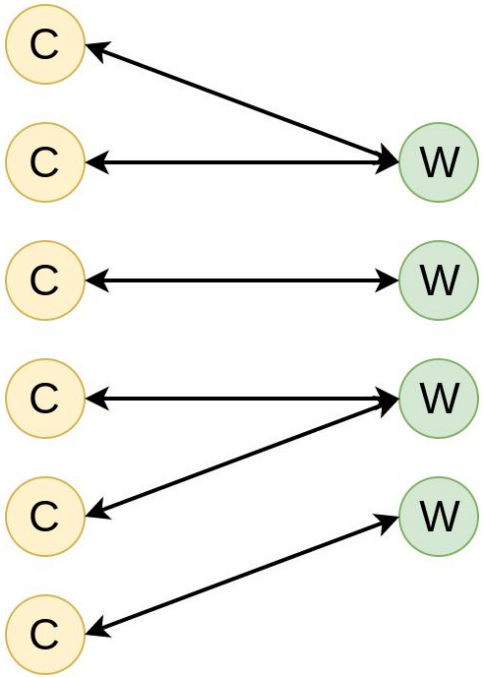
- Experiments
  - Threading primitives
  - Latency vs throughput
  - Changing load and background operations

Threading Primitives

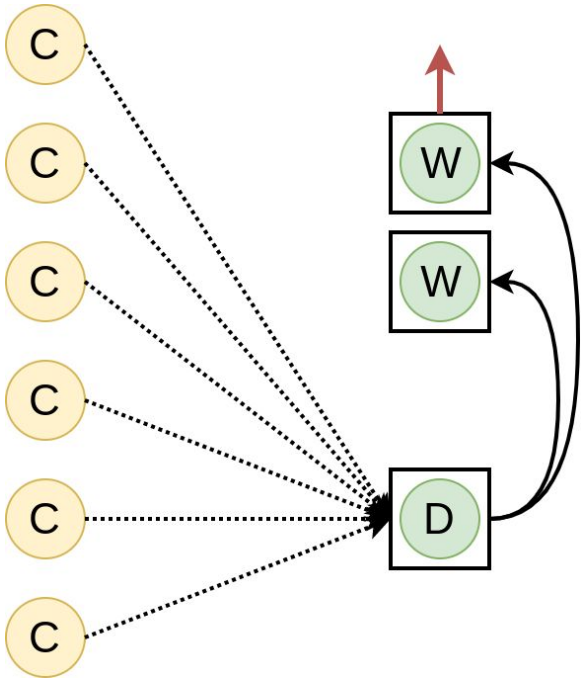
Operation	Arachne	Go	uThreads	std::thread
Thread creation	320 ns	444 ns	6132 ns	13329 ns
Condition variable notify	272 ns	483 ns	4976 ns	4962 ns

Memcached Integration

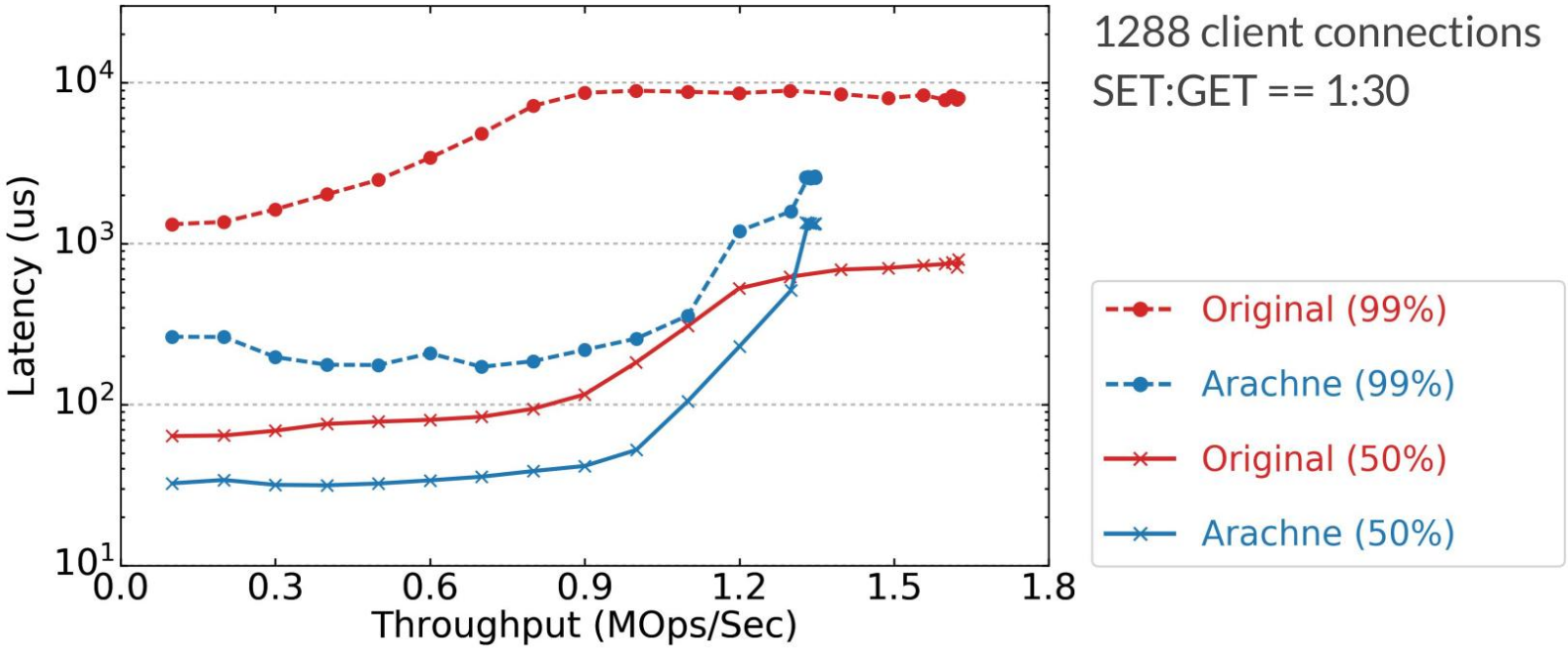
Before: fixed pool of threads



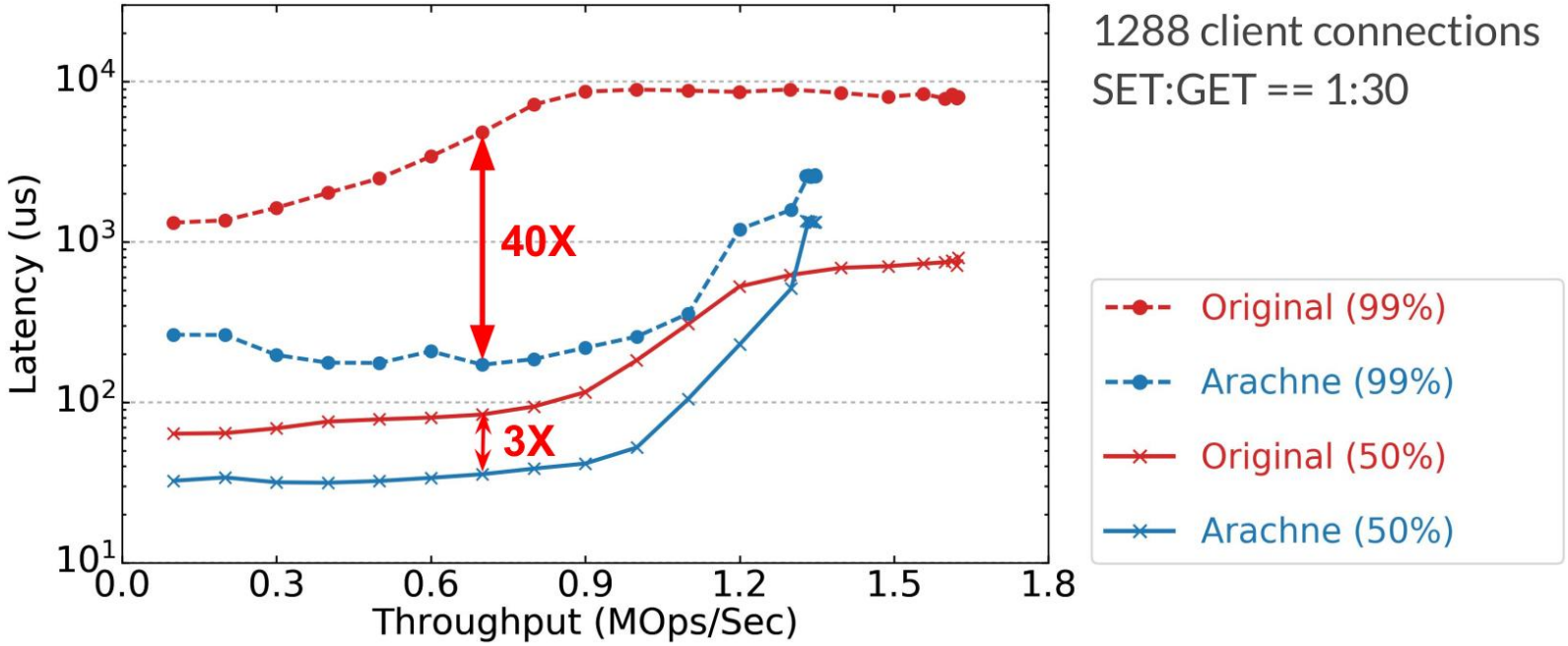
After: one thread per request



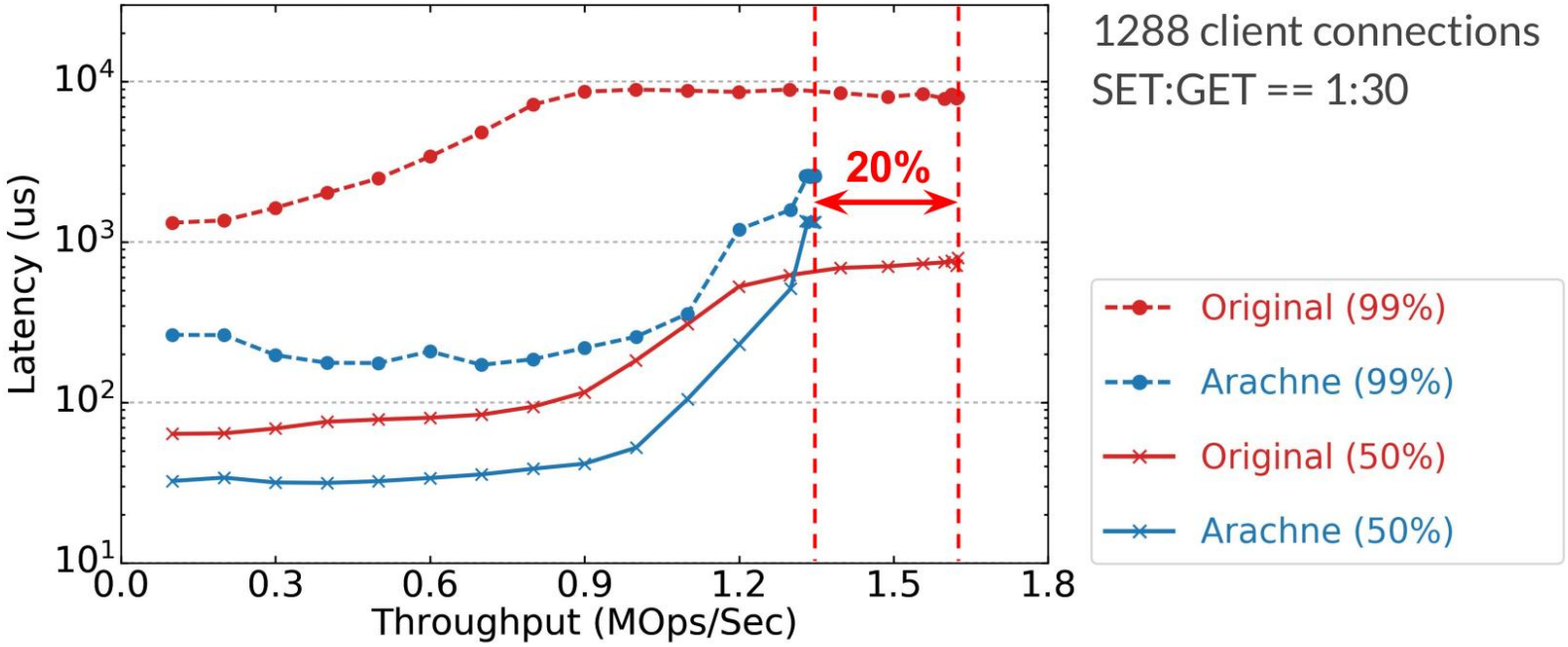
Memcached: Facebook ETC trace



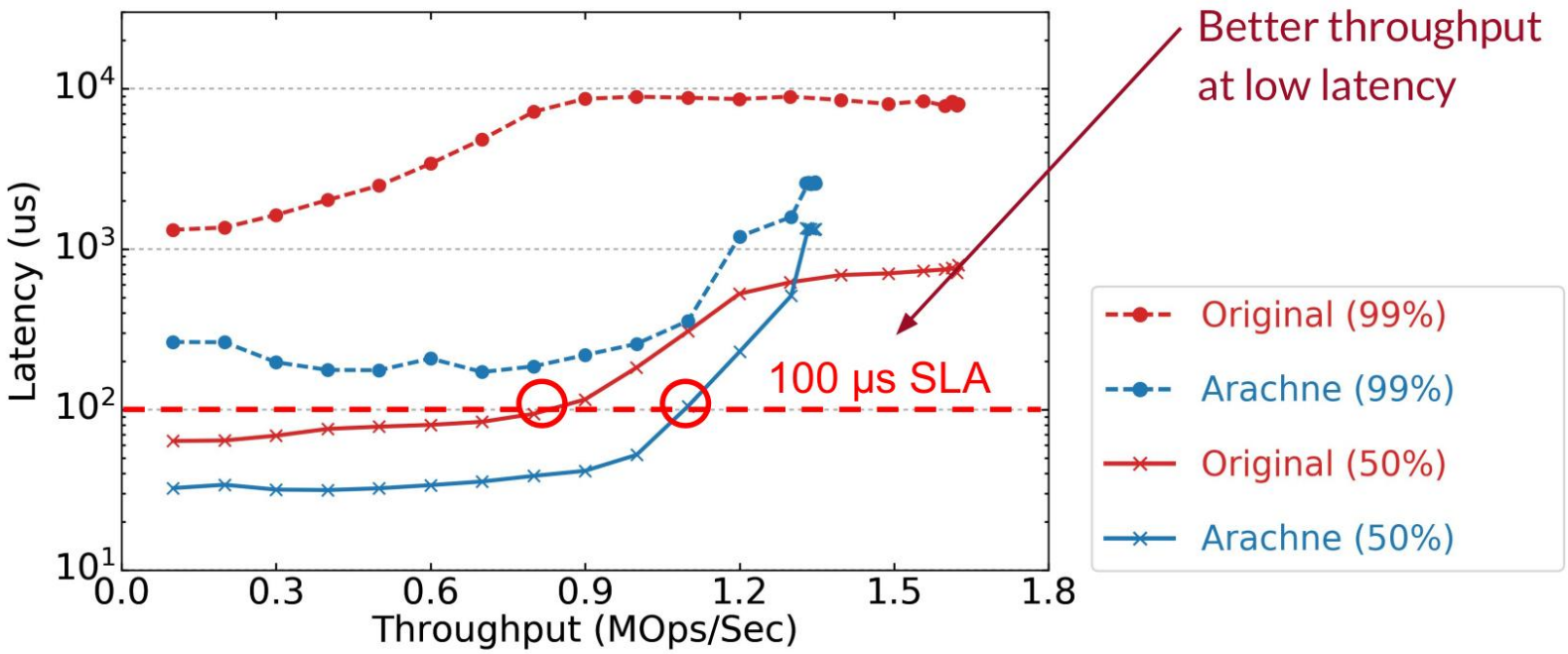
Memcached: Facebook ETC trace



Memcached: Facebook ETC trace



Memcached: Facebook ETC trace



Changing Load and Colocation

Does Arachne scale well with changing load?

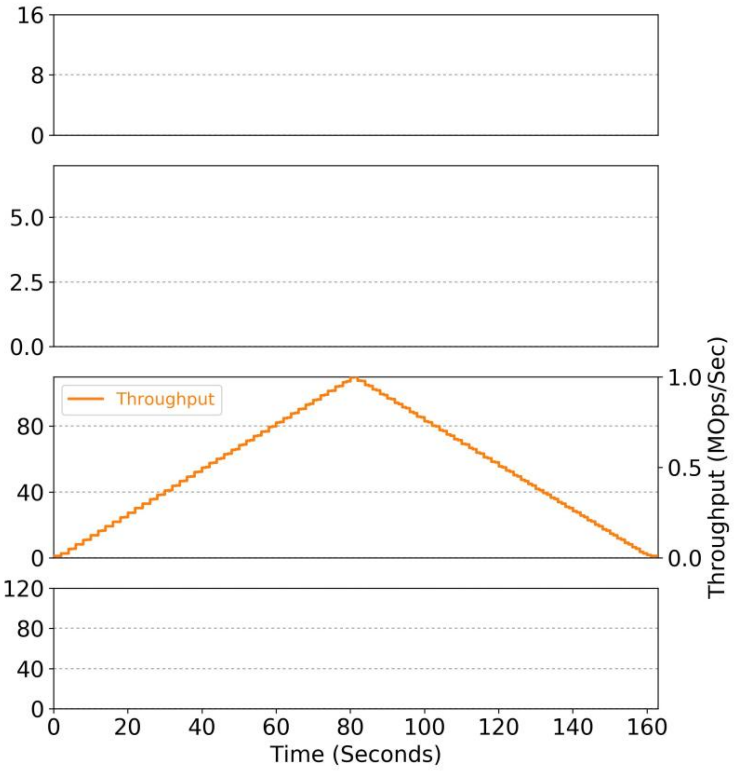
Does Arachne enable high core utilization?

Changing Load

Modified memtier

Poisson arrival rate

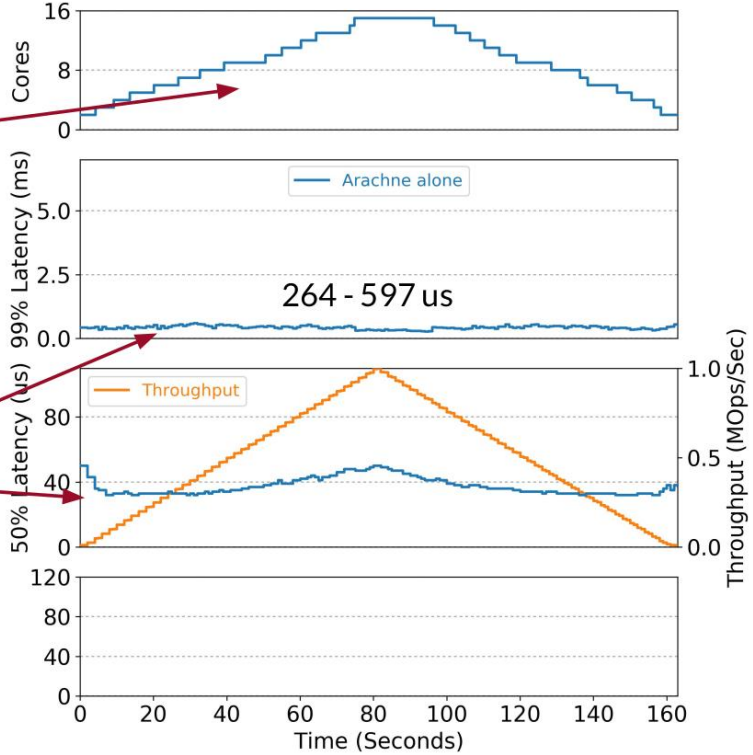
30B Keys, 200B values reads



Changing Load

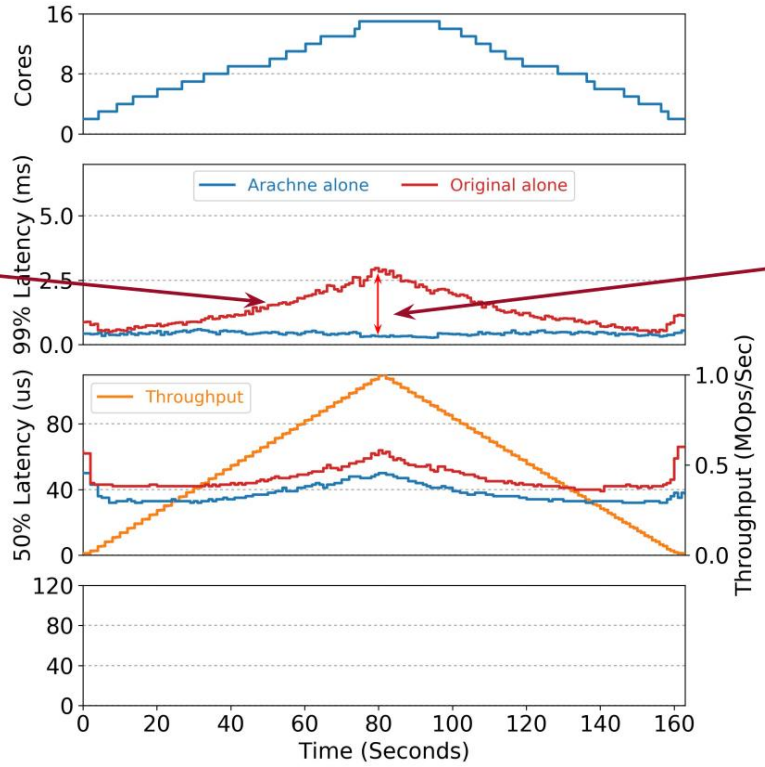
Cores scale with load

Nearly constant median and tail latency



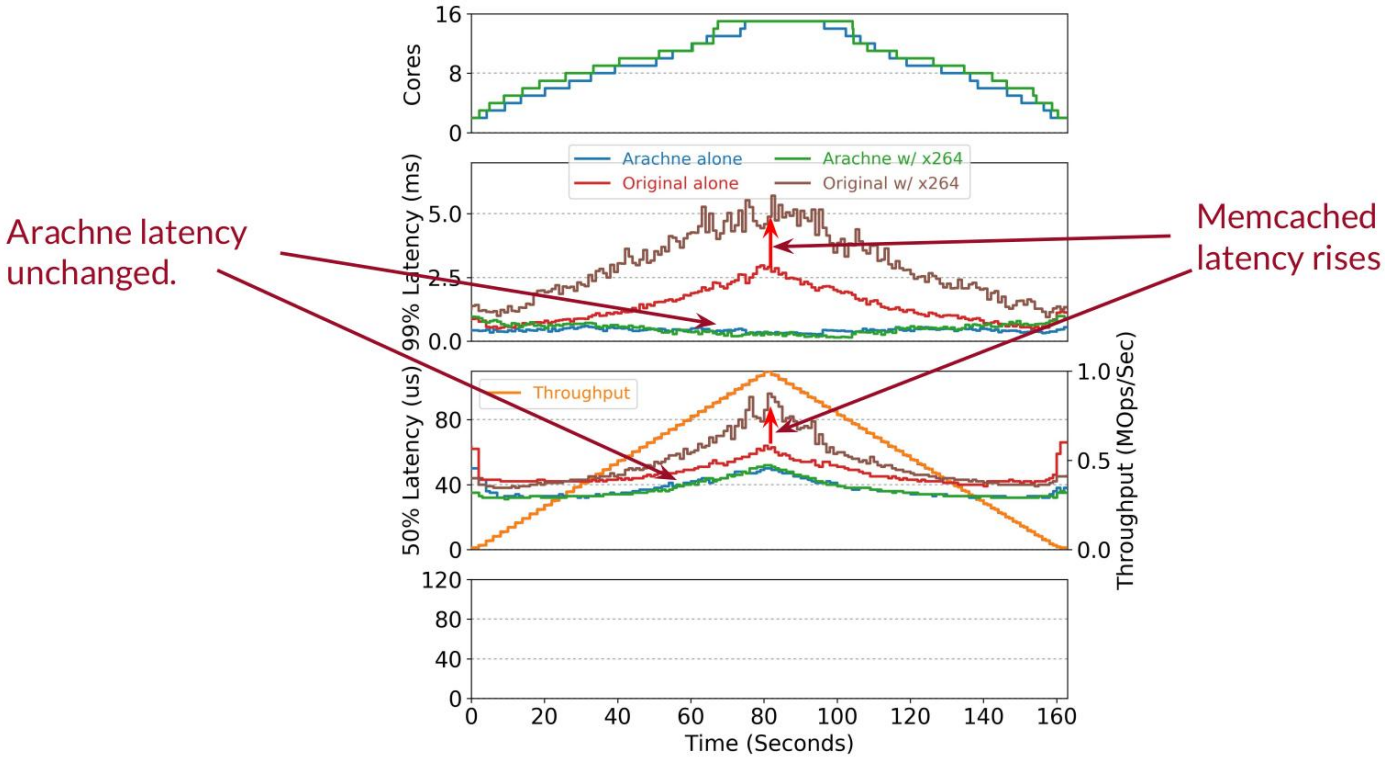
Changing Load

Tail latency increases with load



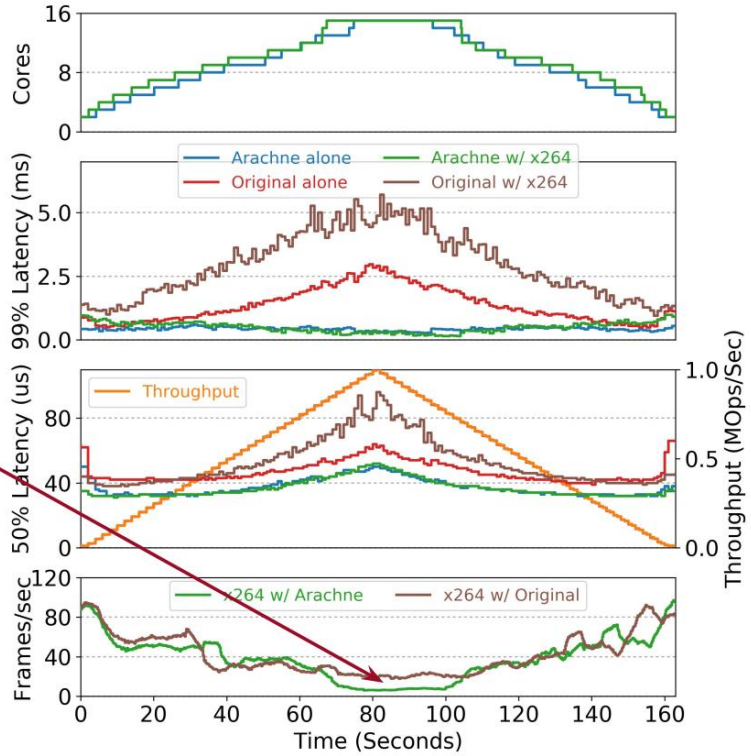
9x higher than Arachne at load

Colocated with x264 Video Encoder



Colocated with x264 Video Encoder

x264 throughput drops at high memcached load



Conclusion

Benefits

- Better combination of latency and throughput
- Efficient thread primitives