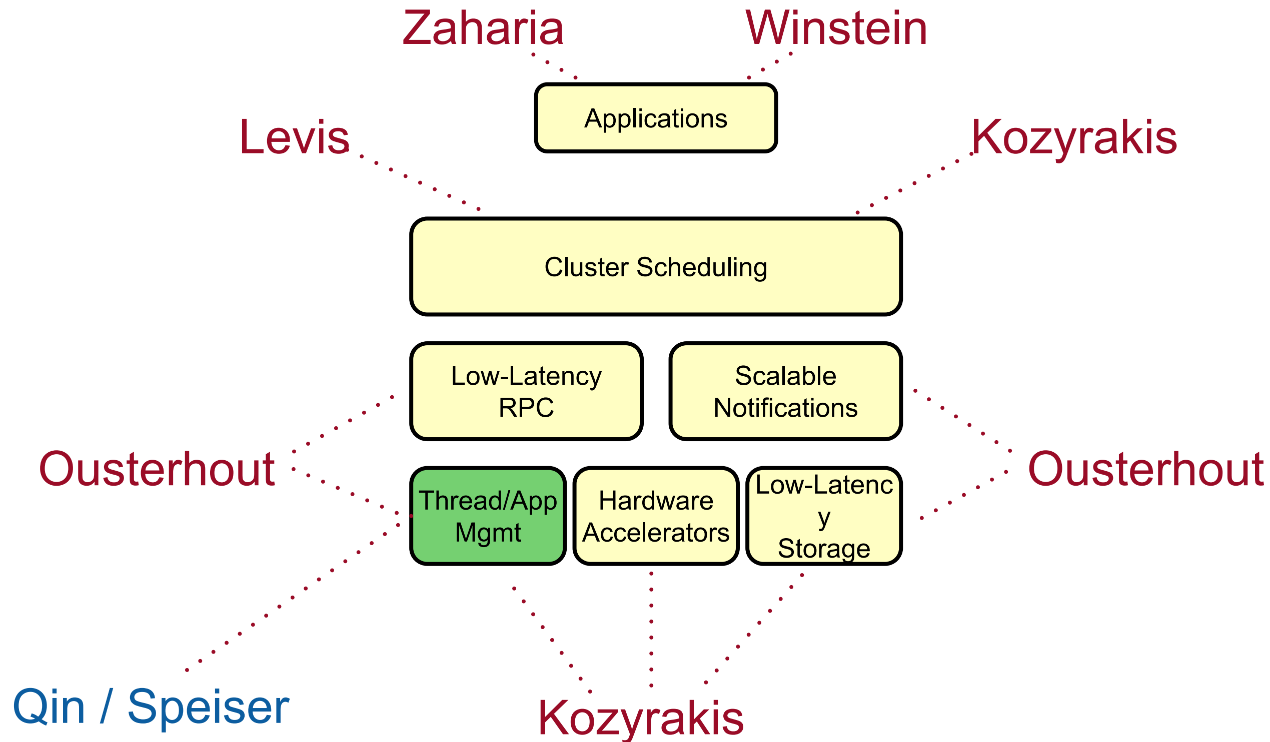
Arachne

Core Aware Thread Management

Henry Qin
Jacqueline Speiser
John Ousterhout

PLATFORMLAB

Granular Computing Platform



Introduction

- Computation is becoming more granular
  - Faster networks, increased use of DRAM → response times in µs's
    - RAMCloud performs reads in 5 µs, service time 2 µs
- **Problem: Hard to achieve both low latency and high throughput**
- **Arachne: Core-Aware Approach to Thread Management**
  - Applications own cores for tens of ms, run userspace threads
  - Each application estimates its core needs
  - Core arbiter allocates cores among applications
- **Results**
  - Efficient user-level threads (< 200 ns thread creation)
  - Efficient core usage (21 µs core reallocation)

Outline

- Problem: Thread management in modern data centers is inefficient
- Arachne overview
- Core allocation
- Arachne runtime
- Performance Benchmarks
- Conclusion

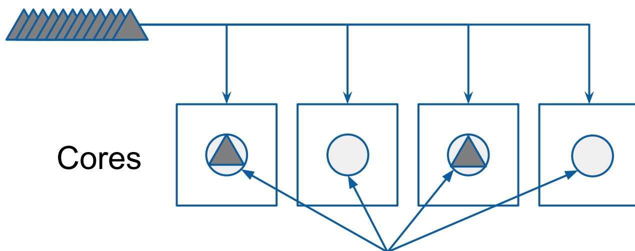
Problems with Thread Management

- **Efficiency**
  - Kernel threading overheads too high
  - 5 μs to create a kernel thread
- **Resource opacity**
  - Applications don’t know how many cores they have
  - Hard to match internal parallelism to available resources
- **Interference**
  - Other applications compete for cores
  - Need exclusive use of cores for low latency
  - Application loads vary over time

Thread Pools

Thread pools solve the kernel thread efficiency problem by reusing kernel threads.

Incoming
Requests

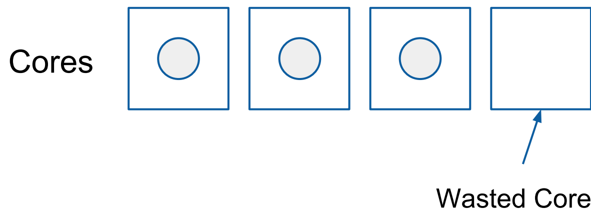


Want # of threads == # of cores

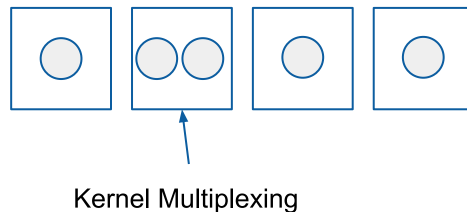
Kernel Threads

But we must know the number of available cores.

Too Few Threads

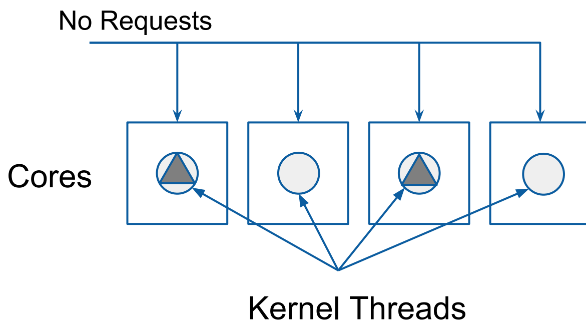


Too Many Threads



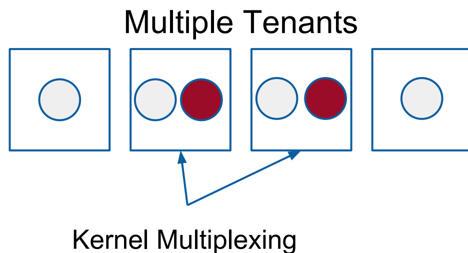
Thread Pools

Dedicated machines enable us to know the number of cores, but they are wasteful under low load.



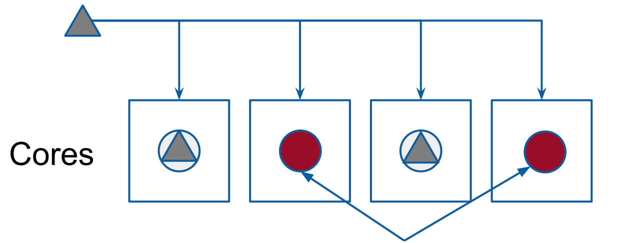
Thread spin with no requests.

But colocating other applications causes competition for cores.

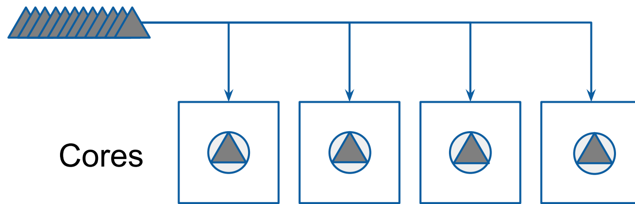


Thread Pools (Desired Behavior)

Reduce number of kernel threads at low load.



Displace other applications at high load.

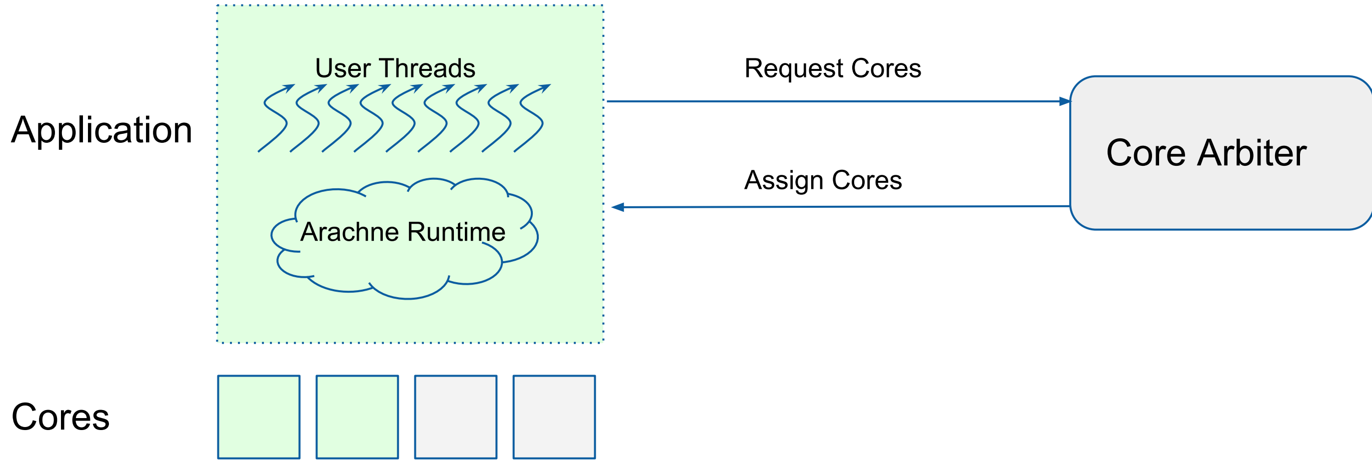


Arachne: Core-Aware Thread Management

- **Cores dedicated to particular applications**
  - Provide isolation → eliminate interference problem
  - Application requests cores, not threads
  - Application knows # of cores it owns
- **Move thread management to userspace**
  - Very fast thread operations (100 - 200 ns)
  - Multiplex short-lived threads on allocated cores
  - Context switch when waiting on µs-scale operations

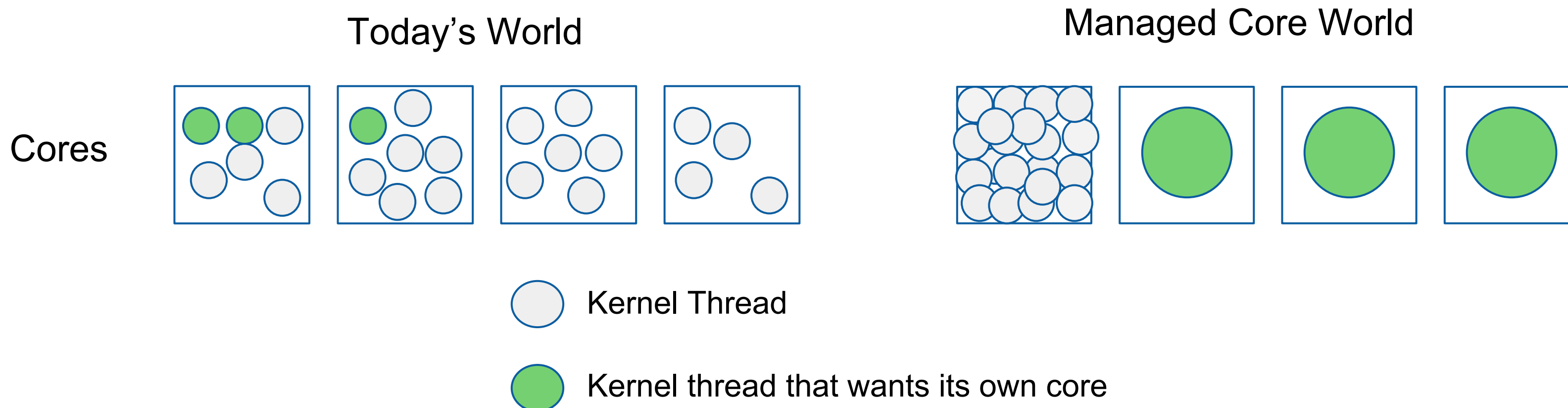
System Overview

- **Core arbiter: an external setuid process shared by all applications**
  - Allocates (managed) cores to applications and arbitrates between applications
- **Runtime component linked into each application**
  - Multiplexes user threads on top of managed cores
  - Estimates number of cores needed



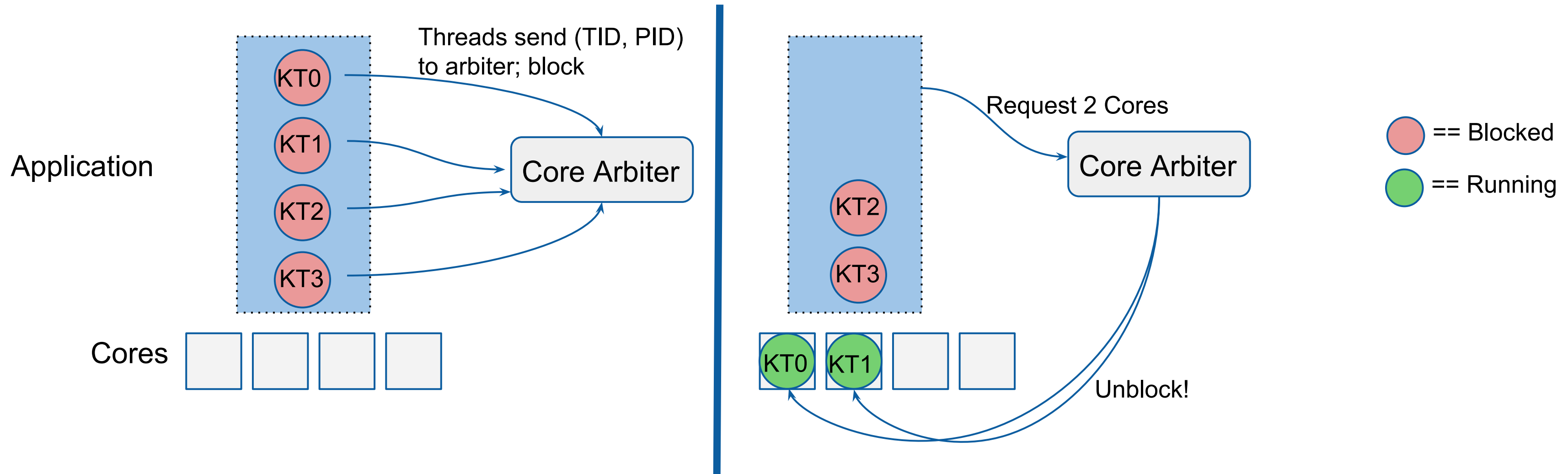
Core Arbiter Design Overview

- **How to “allocate” a core to an application?**
  - Ensure only one kernel thread runs on that core
  - Core pinning is insufficient
- **Applications own cores for long periods (tens of ms)**
  - Adjust allocation as application workloads change



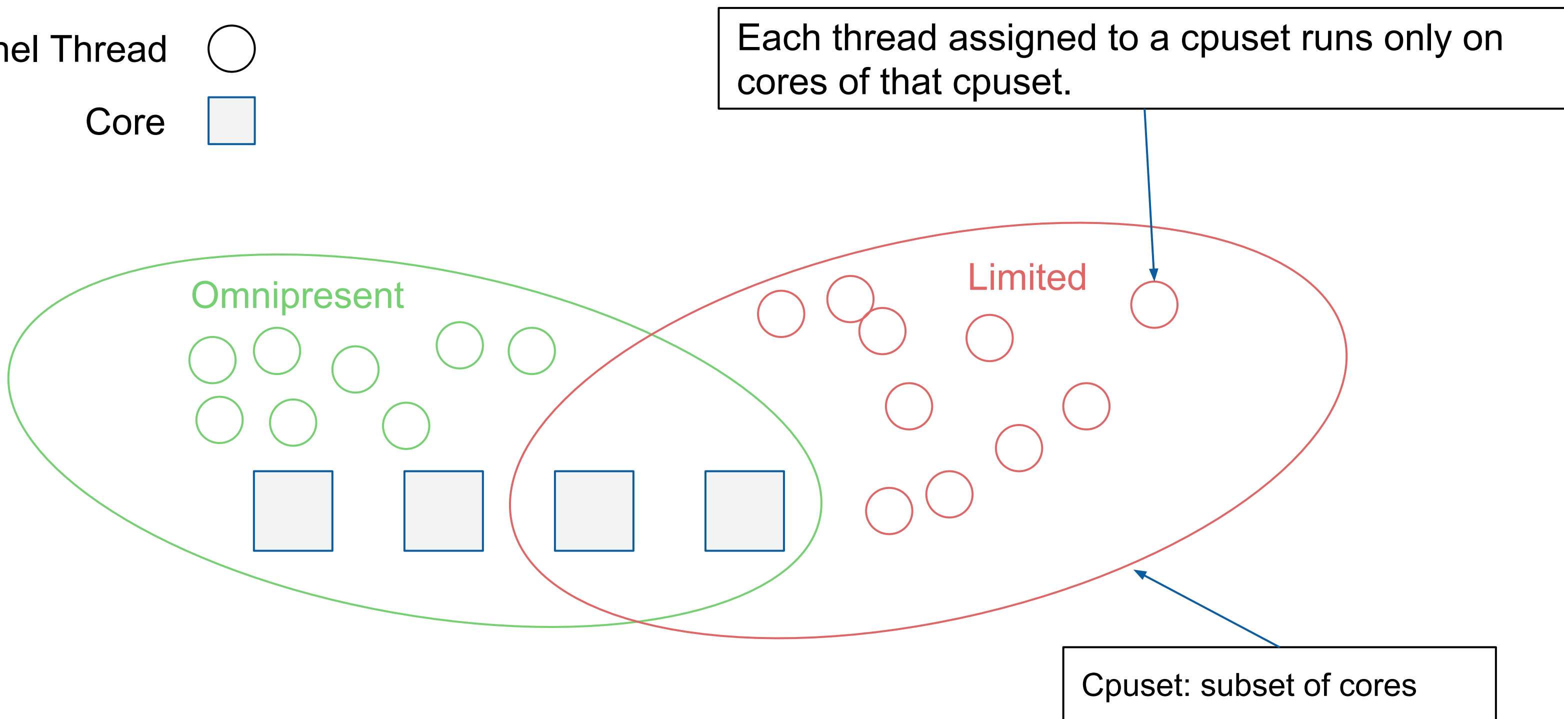
Kernel Threads → Cores

- **Arachne runtime creates a pool of kernel threads; threads block**
- **Application requests X cores**
- **Arbiter moves X kernel threads to managed cores and unblocks them**
  - Unblocked threads own cores

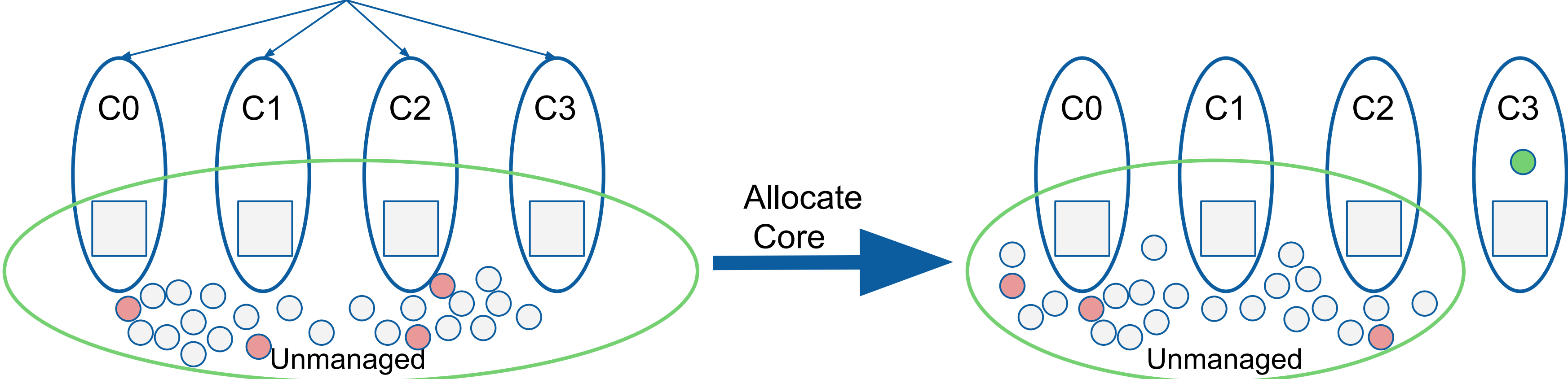


Linux cpusets

- **Idea: Use Linux cpusets to manage cores**



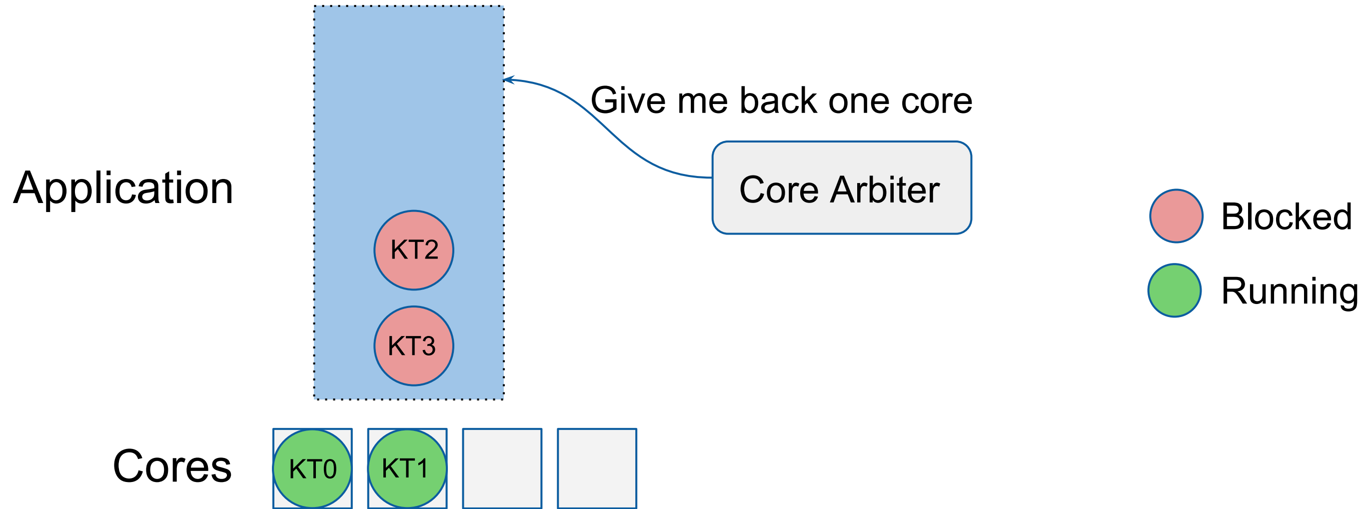
Using cpusets for core allocation



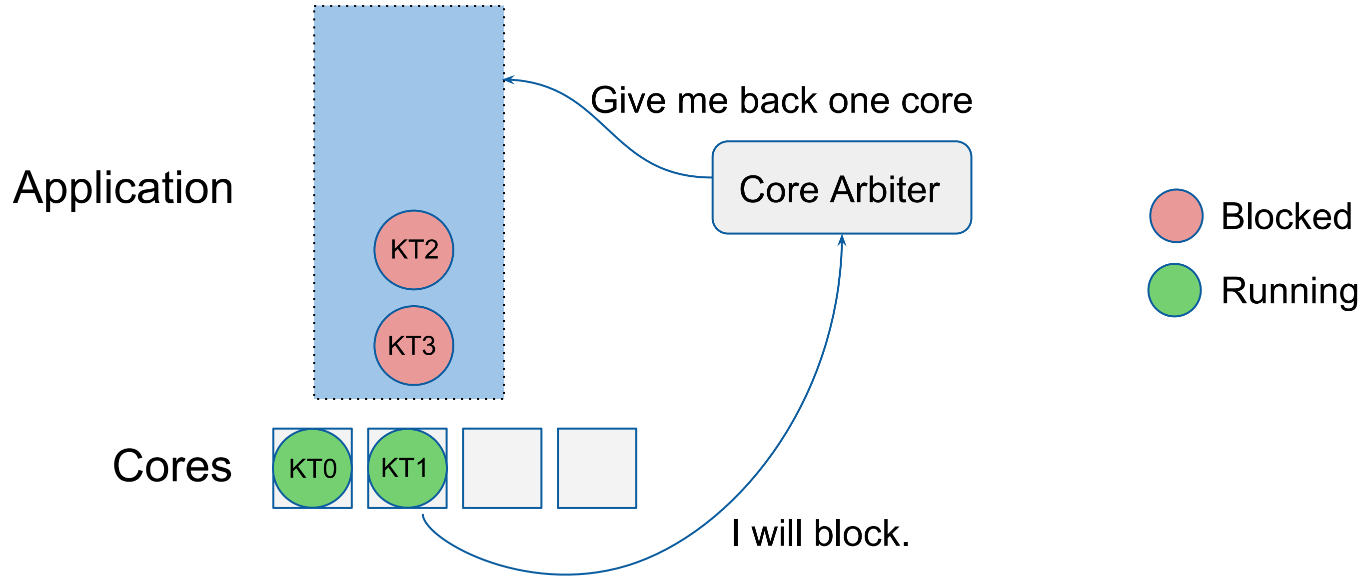
One unmanaged cpuset initially includes all cores.

Allocated cores are removed from unmanaged cpu set.

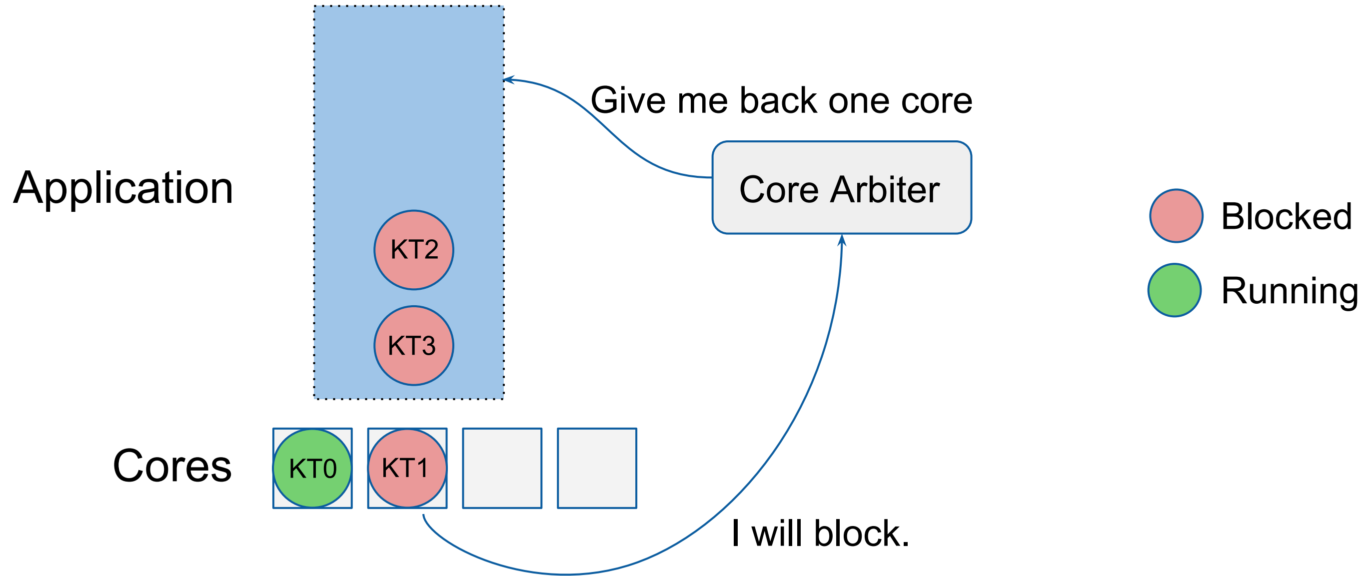
Core Preemption



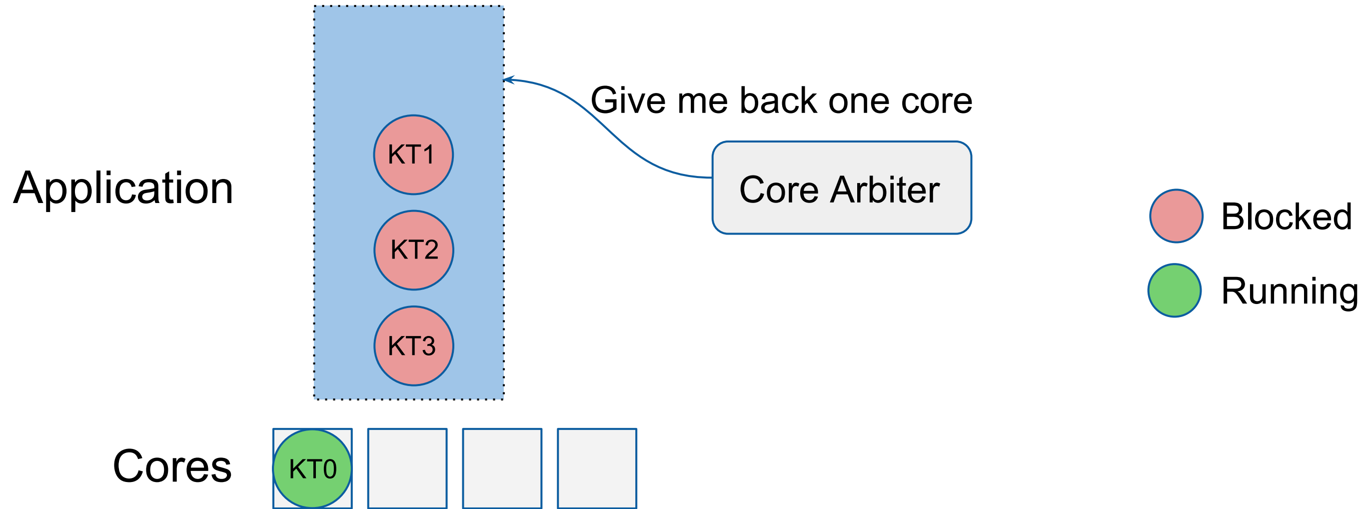
Core Preemption



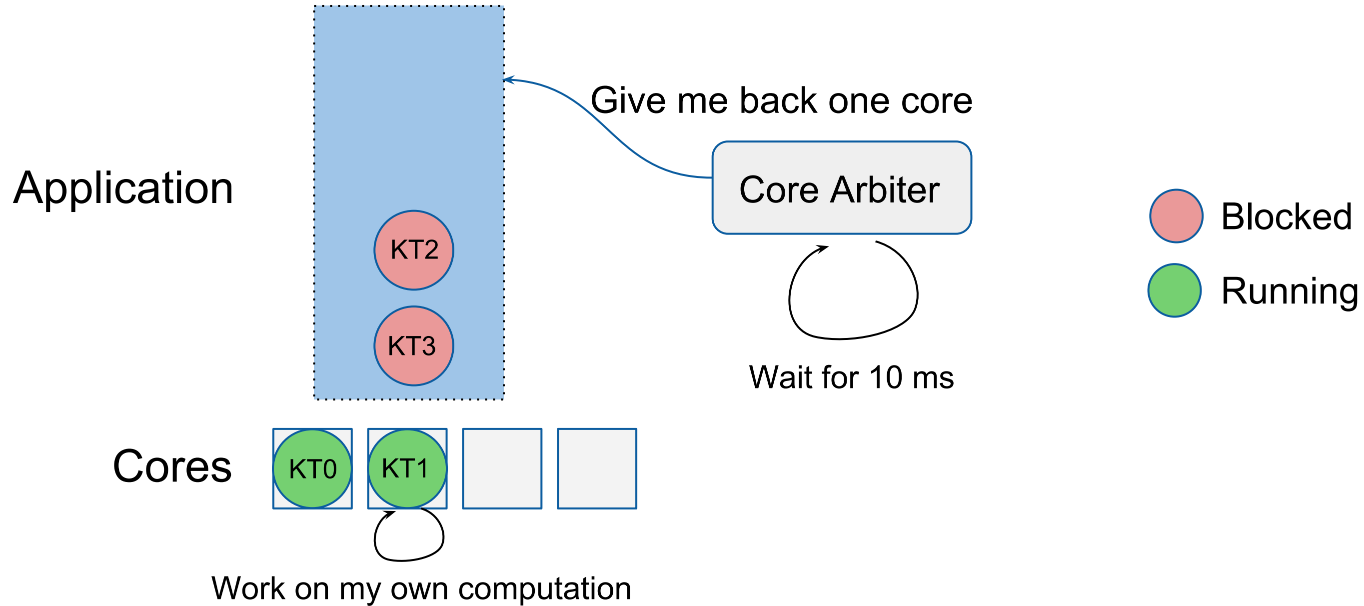
Core Preemption



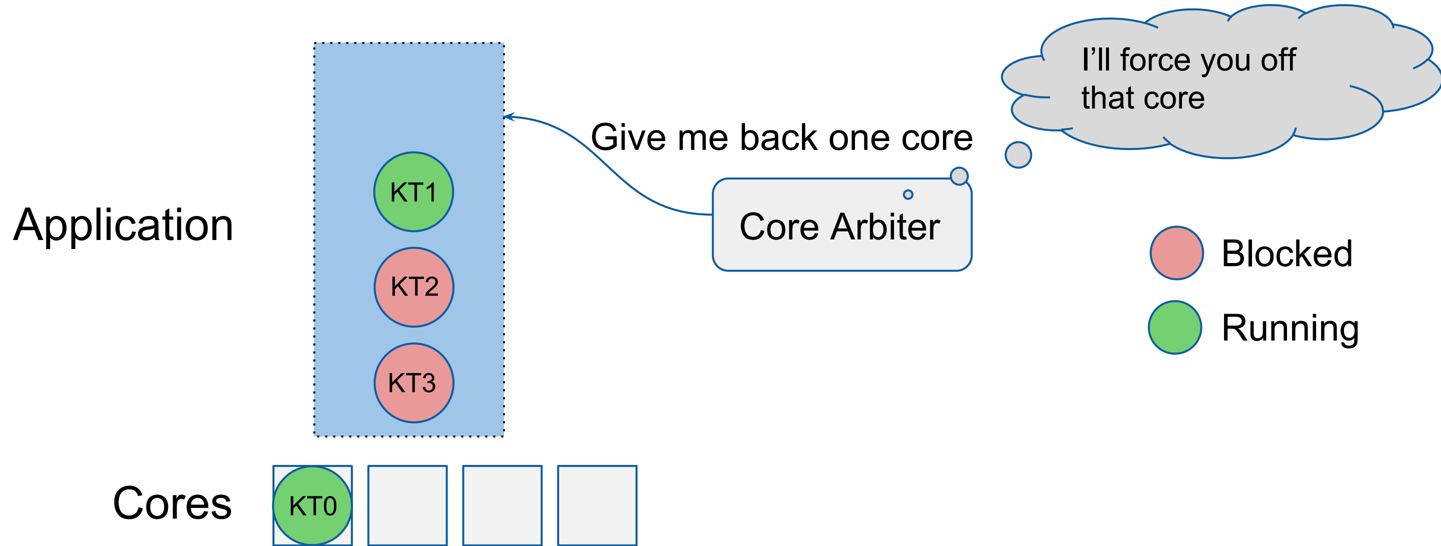
Core Preemption



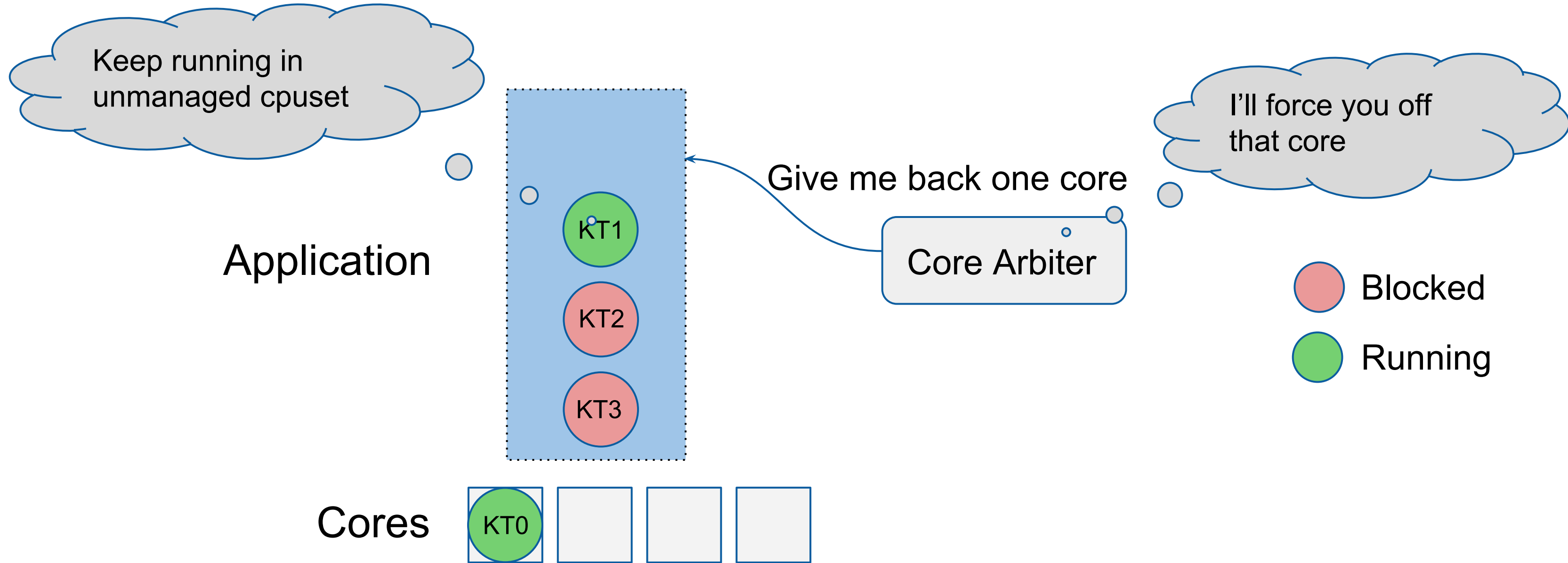
Core Preemption (Misbehaving App)



Core Preemption (Misbehaving App)



Core Preemption (Misbehaving App)



Arachne Runtime Overview

- **Arachne is cooperative - threads on a given core must terminate, yield, or block before other threads run**
- **Each kernel thread (aka core) perpetually looks for user threads and runs them**
    - The search continues when threads relinquish control
- **Thread API**
    - `createThread()` - Spawn a thread with given function and arguments
    - `yield()` - Give other threads on this core a chance to run
    - `sleep()` - Sleep for a minimum duration of time
    - `join()` - Sleep until another thread completes
    - `SleepLock{}`
    - `SpinLock{}`
    - `ConditionVariable{}`
    - `Semaphore{}`

Designing a highly efficient runtime

- **Threading performance is dominated by cache operations**
  - Basic operations are not compute heavy
    - Context switch is only 14 instructions
  - Require communication between cores
  - Inter-core communication requires cache operations. E.g. 100 cycles
- **Arachne runtime minimize cache traffic (data transferred across cores)**

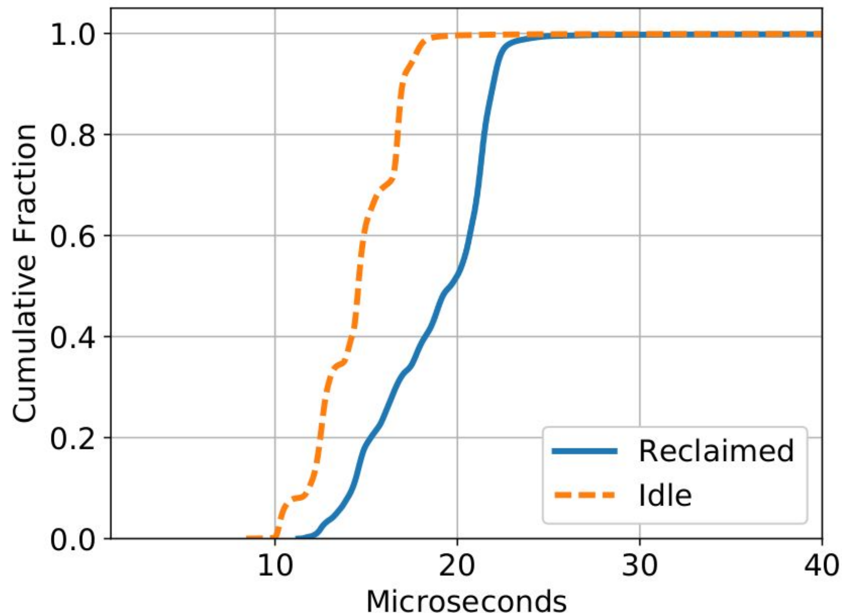
Results

- **Configuration**
  - 4-Core Xeon X3470 @ 2.93 Ghz
  - 24 GB DDR3 @ 800 Mhz
- **Benchmarks**
  - Cost of core allocation
  - Cost of scheduling primitives

What is the cost of core allocation?

- **Plot shows interval from time core requested to time core acquired**
  - Case 1: Arbiter has an idle core (orange line)
  - Case 2: Arbiter must reclaim core from another app (blue line)

- **Median Cost: 21 µs**



What is cost of scheduling primitives?

Operation	Arachne	std::thread	Go
Thread Creation	182 ns	5760 ns	261 ns
Condition Variable Notify	195 ns	4137 ns	317 ns
Yield	88 ns	N/A	N/A
Null Yield	15 ns	N/A	N/A

- **Yield: Time to transfer control from one thread to another on same core**
- **Null Yield: Time for one thread to yield when no other threads are on core**
- **Golang creates threads on same core; higher cost even without cache misses**

Future Work

- **What are the right abstractions for providing more direct control over cores to applications?**
  - Should applications specify different thread profiles (ie latency-sensitive, background, long-running)?
  - Give applications ability to pick cores to run and co-locate specific threads?
- **What are the right policies for allocating hyperthreaded cores?**
  - Always allocate one thread of each pair first?
  - Ensure that hyper-twins always go to the same application?
- **Can Arachne interact with cluster-level schedulers for increased cluster-wide efficiency?**
- **How can Arachne play nicely with today’s containerized world?**

Conclusion

- **We built Arachne, a thread management system for granular tasks on multi-core systems.**
  - Applications own cores for tens of milliseconds and run userspace threads
  - Each application estimates its core needs
  - Core arbiter allocates cores among applications
- **Solves the problems of efficiency, interference, and resource opacity**
- **Enables granular tasks to combine low latency with high throughput**

Questions?

github.com/PlatformLab/Arachne