Granular Computing Platform

Cluster Scheduling

Low-Latency RPC

Scalable Notifications

Thread/App Mgmt

Hardware Accelerators

Low-Latency Storage
Introduction

- Computation is becoming more granular
  - Faster networks, increased use of DRAM $\rightarrow$ 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)
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.

But we must know the number of available cores.

Want # of threads == # of cores
Thread Pools

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

But colocating other applications causes competition for cores.
Thread Pools ( Desired Behavior )

Reduce number of kernel threads at low load.

Incoming Requests

Cores

Threads from another application.

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

![Diagram showing Core Arbiter Design Overview](image)
- 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

Each thread assigned to a cpuset runs only on cores of that cpuset.

- Core
- Kernel Thread
- Omnipresent
- Limited
- Cpuset: subset of cores
Using cpusets for core allocation

One unmanaged cpuset initially includes all cores.

One cpuset for each core

C0  C1  C2  C3

Blocked && waiting for core
Running on managed core
Running on unmanaged

Allocate Core

Allocated cores are removed from unmanaged cpu set.
Core Preemption

Application

Cores

Give me back one core

Core Arbiter

KT2

KT3

KT0

KT1

Blocked

Running
Core Preemption

Application

Cores

Core Arbiter

KT0

KT1

KT2

KT3

Blocked
Running

Give me back one core

I will block.
Core Preemption

Application

Cores

Give me back one core

I will block.

Blocked
Running
Core Preemption

Application

Cores

Give me back one core

Core Arbiter

KT0

KT1

KT2

KT3

Blocked

Running
Core Preemption (Misbehaving App)

Application

Give me back one core
Wait for 10 ms

Core Arbiter

Cores

Work on my own computation

KT0 KT1 KT2 KT3

Blocked Running
Core Preemption (Misbehaving App)

Application

Cores

KT0

KT1

KT2

KT3

Give me back one core

Core Arbiter

I'll force you off that core

Blocked

Running
Core Preemption (Misbehaving App)

Keep running in unmanaged cpuset

Application

Give me back one core

Core Arbiter

I’ll force you off that core

Blocked
Running

Cores

KT0 KT1 KT2 KT3
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?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arachne</th>
<th>std::thread</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Creation</td>
<td>182 ns</td>
<td>5760 ns</td>
<td>261 ns</td>
</tr>
<tr>
<td>Condition Variable Notify</td>
<td>195 ns</td>
<td>4137 ns</td>
<td>317 ns</td>
</tr>
<tr>
<td>Yield</td>
<td>88 ns</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Null Yield</td>
<td>15 ns</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

- **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 (i.e., 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?
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