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 $\Rightarrow$ Poor Resource Management $\Rightarrow$ 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

![Diagram showing wasted core due to too few threads]

Wasted Core

Too Many Threads

![Diagram showing kernel multiplexing due to too many threads]

Kernel Multiplexing
Arachne: Core Aware Thread Management

Now: applications lack control over cores

Arachne: applications are aware of cores

Thread-based API

Core

Core-based API

Kernel
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

Want 2 Cores

Core

Arachne Runtime

Core Policy
Application Startup

Application

Arachne Runtime

Core Policy

Core Unit
Multiplex User Threads
Core Preemption

Release this core
User Thread Migration

![Diagram showing 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arachne</th>
<th>Go</th>
<th>uThreads</th>
<th>std::thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread creation</td>
<td>320 ns</td>
<td>444 ns</td>
<td>6132 ns</td>
<td>13329 ns</td>
</tr>
<tr>
<td>Condition variable notify</td>
<td>272 ns</td>
<td>483 ns</td>
<td>4976 ns</td>
<td>4962 ns</td>
</tr>
</tbody>
</table>
Memcached Integration

Before: fixed pool of threads

After: one thread per request
Memcached: Facebook ETC trace

1288 client connections
SET:GET == 1:30
Memcached: Facebook ETC trace

1288 client connections
SET:GET == 1:30

Latency (us)

Throughput (MOps/Sec)

Original (99%)
Arachne (99%)
Original (50%)
Arachne (50%)
Memcached: Facebook ETC trace

1288 client connections
SET:GET == 1:30

Latency (us)

10^4

10^3

10^2

10^1

Throughput (MOps/Sec)

0.0 0.3 0.6 0.9 1.2 1.5 1.8

Original (99%)
Arachne (99%)
Original (50%)
Arachne (50%)

20%
Memcached: Facebook ETC trace

Better throughput at low latency

Latency (μs)

Throughput (MOps/Sec)

- Original (99%)
- Arachne (99%)
- Original (50%)
- Arachne (50%)

100 μs SLA
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

Graph showing throughput over time with a peak at around 80 MOPS/sec.
Changing Load

Cores scale with load

Nearly constant median and tail latency

264 - 597 us
Changing Load

Tail latency increases with load

9x higher than Arachne at load
Colocated with x264 Video Encoder

Arachne latency unchanged.

Memcached latency rises
Colocated with x264 Video Encoder

x264 throughput drops at high memcached load
Conclusion

Benefits

- Better combination of latency and throughput
- Efficient thread primitives