Arachne Update

Core Policies and Memcached Integration
Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft
John Ousterhout

PLATFORMLAB
Introduction

Arachne: Core-Aware Approach to Thread Management

- Applications own cores for tens of ms, run threads in userspace
- Core arbiter allocates cores among applications
- Application determines how its cores are used; and how many it needs

Benefits of Arachne

- High performance threads → one thread per request is feasible
- Better performance isolation → improved tail latency, especially under load and in competition with background workloads

Today’s Talk

- CorePolicy: Give applications more control over core usage
- Memcached-A: Demonstrate the latency benefits of Arachne
  - Provides 3-40X lower 99% latency on Facebook ETC Workload
  - Improved performance isolation
 Outline

- Brief Arachne overview
- Introduction to CorePolicy
- Memcached thread structure overview
- Introduction to Memcached-A
- Performance comparison: Memcached vs Memcached-A
- Conclusion
Arachne: Core-Aware Thread Management

- Cores dedicated to particular applications
  - Provide isolation, eliminating interference between kernel threads
  - 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
Introduction to CorePolicy
CorePolicy hardcoded into the runtime

- All user threads are equal
- All user threads are load balanced across all cores
- Load estimation is performed on all cores

Problem: Different applications may want to use cores in different ways

- Exclusive on hypertwin (dispatch thread)
- Exclusive on core (heavily loaded dispatch thread)
- Foreground (request processing) vs background (garbage collection)
Mapping from threads to cores is outside the Arachne runtime: CorePolicy

- Responsible for determining how many cores app needs
- When a new thread is created, CorePolicy determines which subset of the cores it can be placed on; Arachne LBs across this subset
- Arachne notifies CorePolicy when cores are added or removed.

Each application picks a CorePolicy on startup

We plan to build a few commonly-used ones

Applications with special requirements can build their own
Applications can implement a CorePolicy in three simple functions.

class CorePolicy {
    virtual void coreAvailable(int myCoreId) = 0;
    virtual void coreUnavailable(int coreId) = 0;
    virtual CoreList getCoreList(int threadClass) = 0;
}
Memcached Integration
Memcached Threading Model

Memcached

- Open source, distributed in-memory key-value cache
- Typically deployed in front of databases

Memcached worker threads model:

- A **fixed-size** pool of worker threads. Connections are assigned statically in a round-robin way (*Only consider TCP here*)
- Requests are delivered to threads via libevent (typically epoll implementation)
Memcached Threading Model

Connection Dispatcher assigns clients to workers.

Request initial assignment
Memcached Threading Problems

Problems with kernel threading

- Poor performance isolation with colocated background jobs
- Unwanted multiplexing even without background competition

Problems with static assignment

- Low throughput under skewed loads
- Sparse usage of kernel threads at low load introduce high latency
  - We believe this is due to C States, but we haven’t yet instrumented this
Design of Memcached-A

We modified memcached version 1.5.6 to use Arachne ("memcached-A")

- **Single** request dispatch thread waits for requests from all connections
- When a request arrives, the dispatch thread creates a short-lived new Arachne thread to handle one or more requests on its connection
- If dispatcher fails to create an Arachne thread (worker threads not keeping up), it handles requests from that connection in place
- Arachne's load estimator automatically adjusts the number of allocated cores

*Preserve the connection dispatcher thread: naturally support multiple dispatch threads*
Memcached-A Threading Model

Clients

Request Dispatcher

Workers
Memcached-A Improves Resource Utilization

**Memcached**

Fixed number of kernel threads, determined at startup.

![Fixed number of kernel threads](image)

**Memcached-A**

Variable number of dedicated cores, determined by load.

![Variable number of dedicated cores](image)

*Better resource utilization; Reduce interference*

Available to other applications
Memcached-A Improves Load Balancing

Memcached

Heavy client load

Memcached-A

Hotspots cannot form

Heavy client load
Memcached Evaluation
Realistic Workload: Facebook ETC trace

1288 client connections
SET:GET == 1:30

16 worker threads, 16 cores
Realistic Workload: Facebook ETC trace

1288 client connections
SET:GET == 1:30

16 worker threads, 16 cores

16 worker threads, 8 cores
Changing Load and Colocation

Modified memtier generates read requests at varying Poisson arrival rate

Offered load starts at 10K QPS and increases to 1M

30B Keys, 200B Values

Some experiments feature a colocated video encoder
Changing Load and Colocation

Memcached-A is able to scale its number of used cores based on load.

Memcached-A maintains nearly constant median and tail latency.

Under background interference, memcached-A scales faster to compensate.
Changing Load and Colocation

Video throughput drops in both systems as memcached load increases

Under high load, video processing throughput drops 3x more
Throughput vs Increasing Skew

An increasing fraction of offered load is directed to connections serviced by one worker thread.
Conclusion

- We built Arachne, a thread management system for granular tasks on multi-core systems.
  - Applications own cores for tens of ms, run userspace threads
  - Core arbiter allocates cores among applications
  - Application determines how its cores are used; and how many it needs

- CorePolicies offers additional flexibility for applications
  - We intend to do future experiments with new CorePolicies
  - Please come to the poster if you have ideas for useful CorePolicies!

- Benefits demonstrated in Memcached-A
  - Threading that is fast enough to be used on a per-request basis
  - Excellent performance isolation between tasks → Low tail latency
  - Good load balancing across cores
Questions?

github.com/PlatformLab/Arachne

github.com/PlatformLab/memcached-A