# Arachne: Core-Aware Thread Management
# Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft

PRESENTOR :

SHWETHA BALI YADAV

# Arachne

- Arachne is a new user-level implementation of threads that provides both low latency and high throughput for applications with extremely short-lived threads.
- It is core-aware:
  - Each application determines how many cores it needs, based on its load;
  - Always aware exactly which cores it has been allocated, and it controls the placement of its threads on those cores.

# Contents:

# Introduction

- It is difficult to provide services with low latency and high throughput.
- Techniques for achieving low latency, such as reserving cores for peak throughput or using polling instead of interrupts, waste resources.
- Reason:
- Applications must manage their parallelism with a virtual resource (threads); they cannot tell the operating system how many physical resources (cores) they need, and they do not know which cores have been allocated for their use.

# Introduction

- To resolve this problem Arachne is used. It is the thread management system gives, application visibility into the physical resources they are using. Hence it is known as core-aware thread management.
- Application thread are managed at user level, they are not visible to operating system.
- Each application always knows exactly which cores it has been allocated and it decides how to schedule application threads on cores.
- A core arbiter decides how many cores to allocate to each application and adjusts the allocations in response to changing application requirements.

# Introduction

Features of Arachne:

- Arachne contains mechanisms to estimate the number of cores needed by an application as it runs.
- Arachne allows each application to define a core policy, which determines at runtime how many cores the application needs and how threads are placed on the available cores.
- The Arachne runtime was designed to minimize cache misses. It uses a novel representation of scheduling information with no ready queues, which enables low latency and scalable mechanisms for thread creation, scheduling, and synchronization.

# Introduction

- Arachne provides a simpler formulation than scheduler activations, based on the use of one kernel thread per core.
- Arachne runs entirely outside the kernel and needs no kernel modifications; the core arbiter is implemented at user level using the Linux cpuset mechanism.

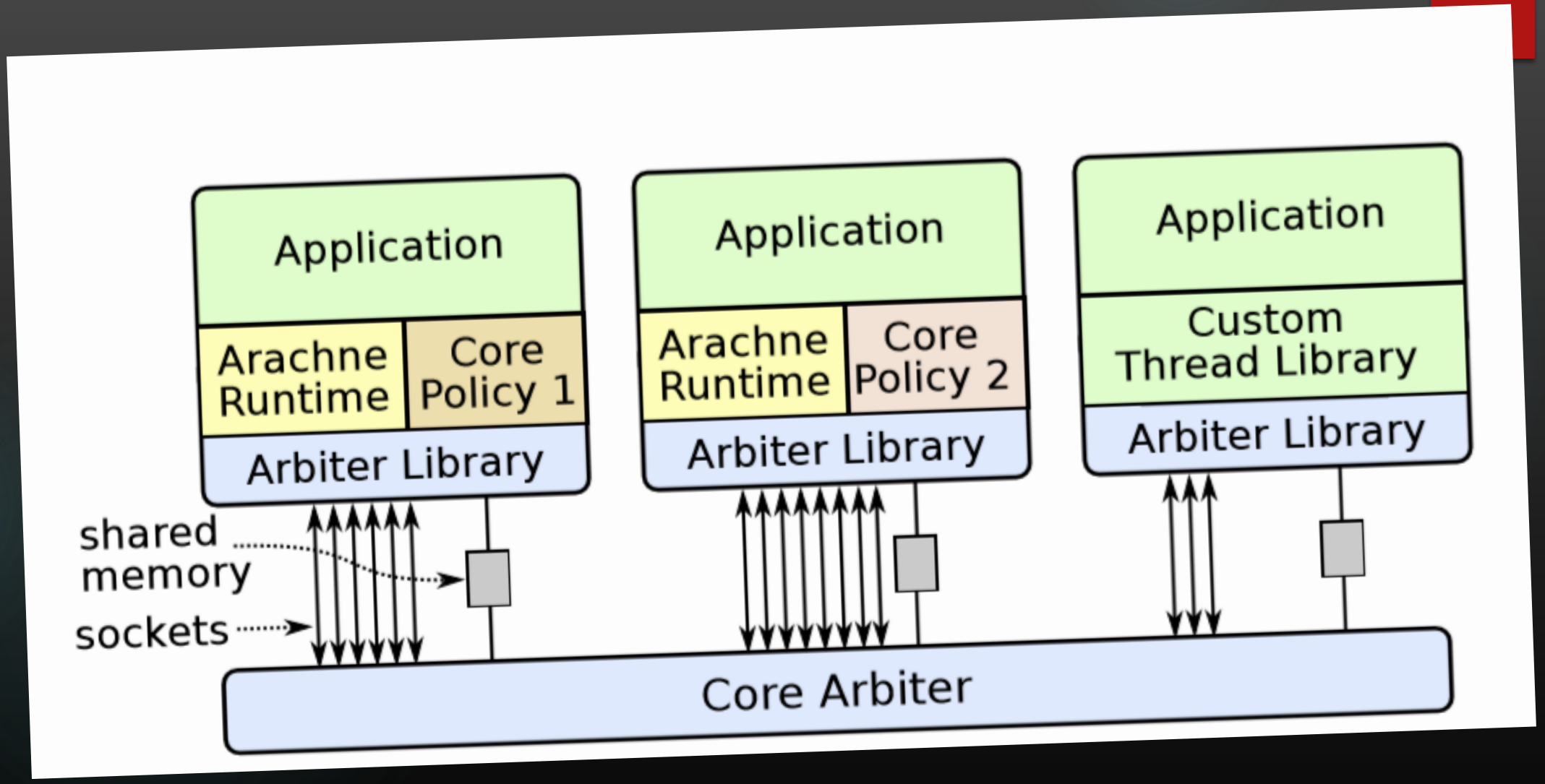
# The Threading Problem

- The number of worker threads is fixed when memcached starts, which results in several inefficiencies. If the number of cores available to memcached is smaller than the number of workers, the operating system will multiplex workers on a single core, resulting in long delays for requests sent to descheduled workers.
- One core must be reserved for each worker thread.
- During periods of low load, each worker thread will be lightly loaded, increasing the risk that its core will enter power-saving states with high latency wakeups.

# Arachne Overview

- The goal for Arachne is to provide a thread management system that allows a better combination of low latency and high throughput.
- Each application should match its workload to available cores, taking only as many cores as needed and dynamically adjusting its internal parallelism to reflect the number of cores allocated to it.

# Arachne Architecture



- The core arbiter consists of a stand-alone user process plus a small library linked into each application.
- The Arachne runtime and core policies are libraries linked into applications.
- Different applications can use different core policies. An application can also substitute its own threading library for the Arachne runtime and core policy, while still using the core arbiter.

# The Core Arbiter:

- It is a user-level process that manages cores and allocates them to applications.
- It collects information from each application about how many cores it needs and uses a simple priority mechanism to divide the available cores among competing applications.
- It adjusts the core allocations as application requirements change.

# Core Arbiter 3 features:

- It implements core management entirely at user level using existing Linux mechanisms; it does not require any kernel changes.
- It coexists with existing applications that don't use Arachne.
- It takes a cooperative approach to core management, both in its priority mechanism.

# Arachne Runtime

The Arachne runtime communicates with the core arbiter using three methods in the arbiter's library package:

- **setRequestedCores:** invoked by the runtime whenever its core needs change; indicates the total number of cores needed by the application at various priority levels.
- **blockUntilCoreAvailable:** invoked by a kernel thread to identify itself to the core arbiter and put the kernel thread to sleep until it is assigned a core.
- **mustReleaseCore:** invoked periodically by the runtime; a true return value means that the calling kernel thread should invoke blockUntilCoreAvailable to return its core to the arbiter.

# Core-optimized design

- The performance of the Arachne runtime is dominated by cache misses.
- For example, to transfer a value from one core to another, it must be written on the source core and read on the destination core.
- This takes about three cache miss times: the write will probably incur a cache miss to first read the data; the write will then invalidate the copy of the data in the destination cache, which takes about the same time as a cache miss; finally, the read will incur a cache miss to fetch the new value of the data.
- Cache misses can take from 50-200 cycles, so even if an operation requires only a single cache miss, the miss is likely to cost more than all of the other computation for the operation.

# Core-optimized design

- The effective cost of a cache miss can be reduced by performing other operations concurrently with the miss.
- For example, if several cache misses occur within a few instructions of each other, they can all be completed for the cost of a single miss.

# Thread Creation

- Create new threads on the same core as the parent; they use work stealing to balance load across cores.
- This avoids cache misses at thread creation time.
- Goal is to get a new thread on an unloaded core as quickly as possible.
- Cache misses can occur during thread creation for the following reasons:

# Thread Creation

- **Load balancing:** Arachne must choose a core for the new thread in away, that balances load across available cores; cache misses are likely to occur while fetching shared state describing current loads.
- **State transfer:** the address and arguments for the thread's top-level method must be transferred from the parent's core to the child's core.
- **Scheduling:** the parent must indicate to the child's core that the child thread is runnable.
- **Thread context:** the context for a thread consists of its call stack, plus metadata used by the Arachne runtime, such as scheduling state and saved execution state when the thread is not running.

# Thread Creation

- To create a new user thread, the Arachne runtime must choose a core for the thread and allocate one of the thread contexts associated with that core.
- Each of these operations will probably result in cache misses, since they manipulate shared state.
- In order to minimize cache misses, Arachne uses the same shared state to perform both operations simultaneously.

# Thread Creation

- When creating new threads, Arachne uses the "power of two choices" approach for load balancing.
- It selects two cores at random, reads their maskAndCount values, and selects the core with the fewest active thread contexts.
- This will likely result in a cache miss for each maskAndCount, but they will be handled concurrently so the total delay is that of a single miss.
- Arachne then scans the mask bits for the chosen core to find an available thread context and uses an atomic compare-and-swap operation to update the maskAndCount for the chosen core.

# Thread Scheduling

- Arachne dispatcher repeatedly scans all of the active user thread contexts associated with the current core until it finds one that is runnable.
- This approach is efficient because of 2 reasons: we expect only a few thread contexts to be occupied for a core at a given time and the cost of scanning the active thread contexts is largely hidden by an unavoidable cache miss.
- The wakeupTime variable is used for Thread Scheduling.

# Core Policies

- The core policy uses thread classes to manage the placement of new threads.
- When a new thread is created, Arachne invokes a method **getCores** in the core policy, passing it the thread's class.
- The **getCores** method uses the thread class to select one or more cores that are acceptable for the thread.
- Each exclusive thread runs on a separate core reserved for that particular thread when an exclusive thread is blocked, its core is idle.
- Exclusive threads are useful for long-running dispatch threads that poll.
- Normal threads share a pool of cores that is disjoint from the cores used for exclusive threads; there can be multiple normal threads assigned to a core at the same time.

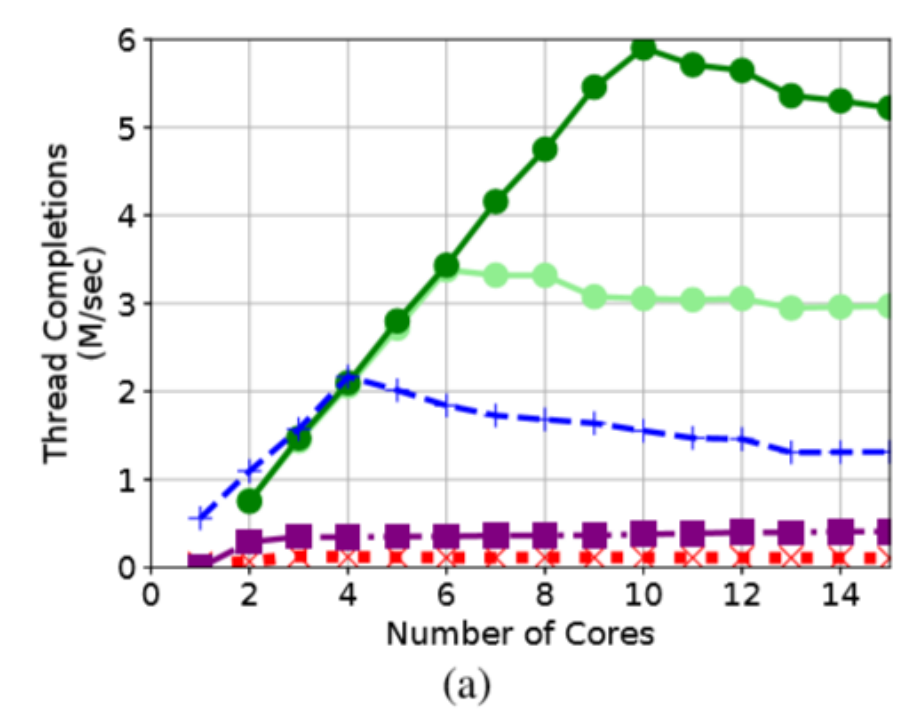
# Evaluation

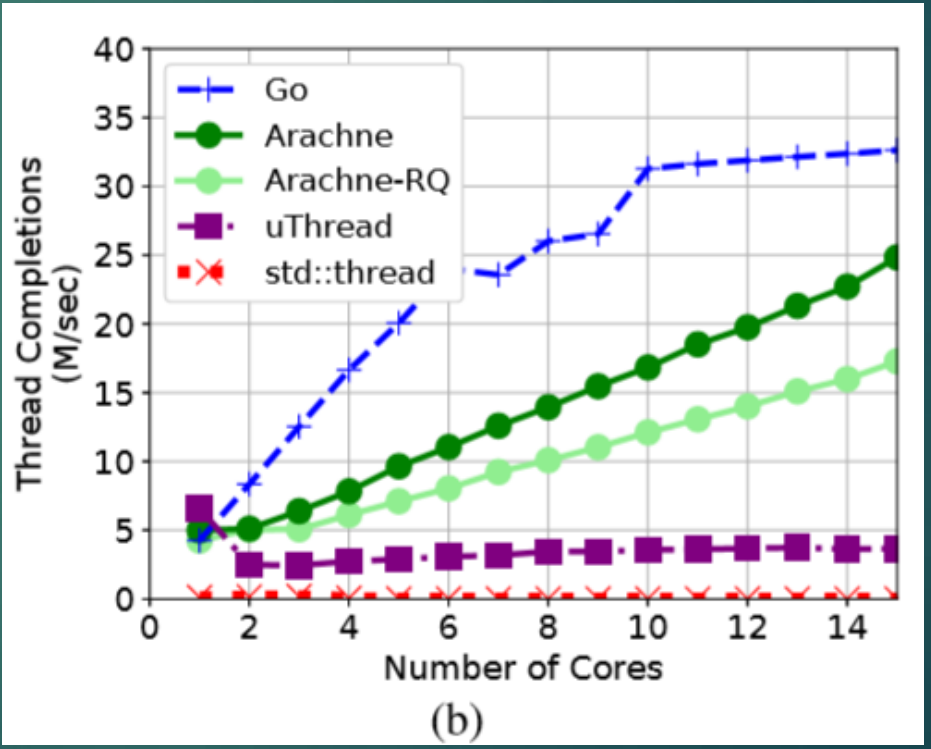
They implemented Arachne in C++ on Linux.

Our evaluation of Arachne addresses the following questions:

- How efficient are the Arachne threading primitives, and how does Arachne compare to other threading systems?
- Does Arachne's core aware approach to threading produce significant benefits for low-latency applications?

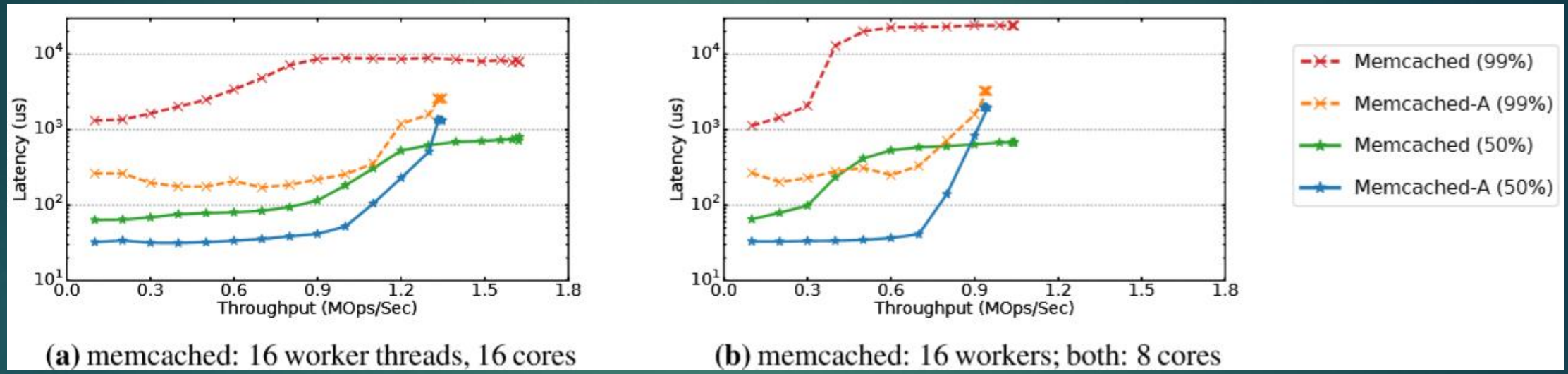
# Thread Primitives



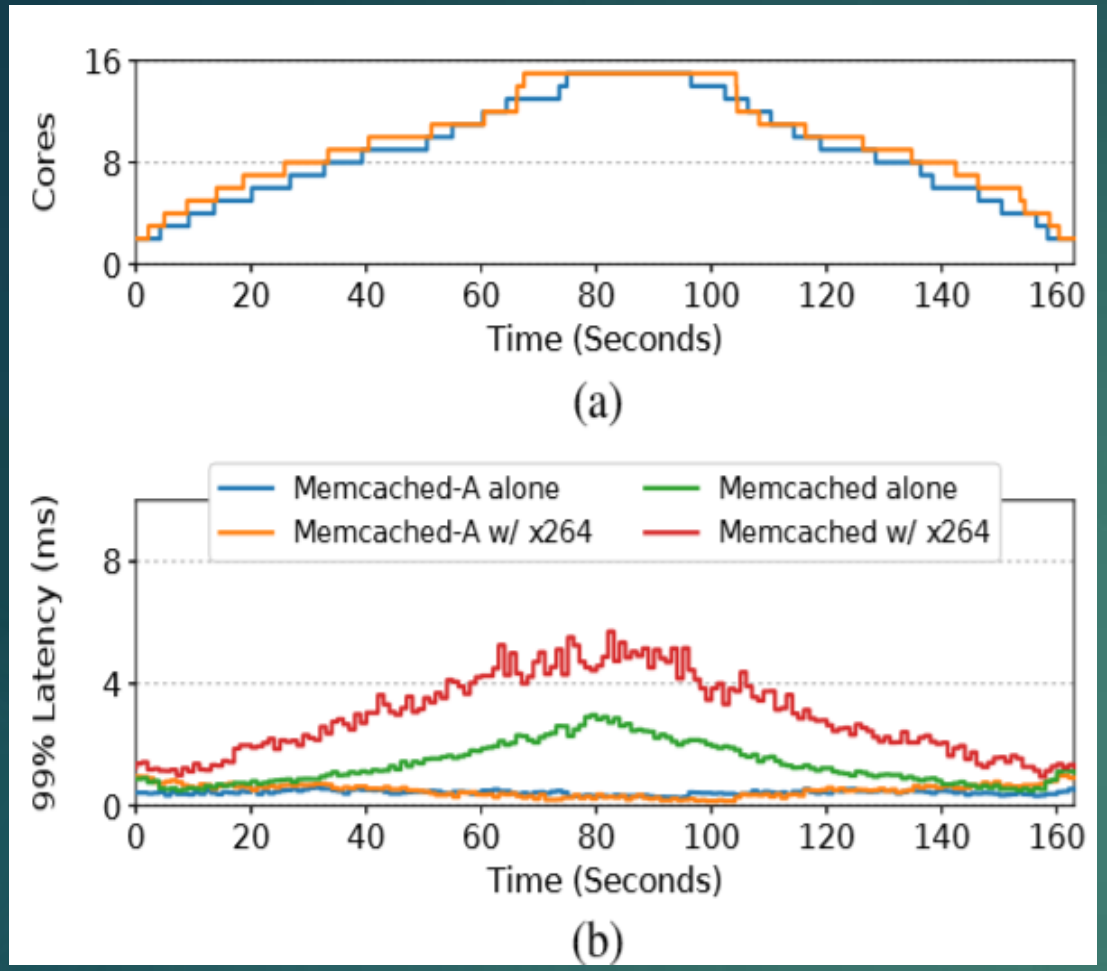



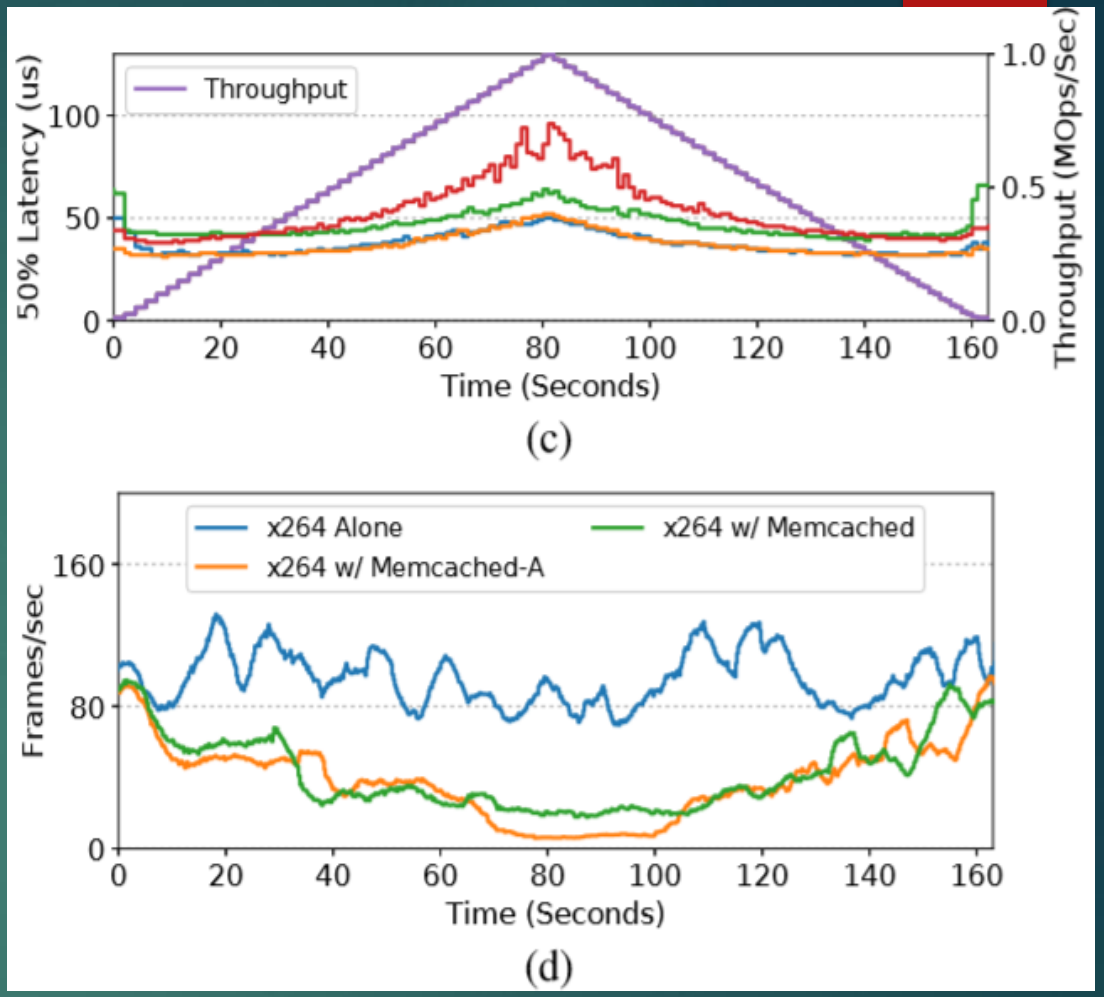

A single thread creates new threads as quickly as possible; each child consumes 1 µs of execution time and then exits.

3 initial threads are created for each core; each thread creates one child and then exits.




Median and 99th-percentile request latency as a function of achieved throughput for both memcached and memcachedA, under the Realistic benchmark. Each measurement ran for 30 seconds after a 5-second warmup. Y-axes use a log scale.




(a) showsthenumberof coresallocatedtomemcached-A;
(b) shows 99th percentile tail latency for memcached and memcached-A;
(c) shows median latency, plus the rate of requests;
(d) shows the throughput of the video decoder (averaged over trailing 4 seconds) when running by itself or with memcached or memcached-A.

# Conclusion

- Arachne provides a mechanism to balance the usage of virtual threads against the availability of physical cores.
- Overall, Arachne's core-aware approach to thread management enables granular applications that combine both low latency and high throughput.