Arachne: Core-Aware Thread Management

Henry Qin
Stanford University
hq6@cs.stanford.edu

Qian Li
Stanford University
qianli@cs.stanford.edu

Jacqueline Speiser
Stanford University
jspeiser@cs.stanford.edu

Peter Kraft
Stanford University
kraftp@stanford.edu

John Ousterhout
Stanford University
ouster@cs.stanford.edu

Abstract

Arachne is a new user-level implementation of threads that provides both low latency and high throughput for applications with extremely short-lived threads (only a few microseconds). Arachne is *core-aware*: each application determines how many cores it needs, based on its load; it always know exactly which cores it has been allocated, and it controls the placement of its threads on those cores. A central core arbiter allocates cores between applications. Adding Arachne to memcached improved SLO-compliant throughput by 37%, reduced tail latency by more than 10x, and allowed memcached to coexist with background applications with almost no performance impact. Adding Arachne to the RAMCloud storage system increased its write throughput by more than 2.5x. The Arachne threading library is optimized to minimize cache misses; it can initiate a new user thread on a different core (with load balancing) in 220 ns. Arachne is implemented entirely at user level on Linux; no kernel modifications are needed.

1 Introduction

Advances in networking and storage technologies have made it possible for datacenter services to operate at exceptionally low latencies [5]. As a result, a variety of low-latency services have been developed in recent years, including FaRM [9], Memcached [18], MICA [15], RAMCloud [25], and Redis [28]. They offer end-to-end response times as low as 5 µs for clients within the same datacenter and they have internal request service times as low as 1–2 µs. These systems employ a variety of new techniques to achieve their low latency, including polling instead of interrupts, kernel bypass, and run to completion [6, 26].

However, it is difficult to construct services that provide both 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. Multi-level services, in which servicing one request may require nested requests to other servers (such as for replication), create additional opportunities for resource underutilization, particularly if they use polling to reduce latency. Background activities within a service, such as garbage collection, either require additional reserved (and hence underutilized) resources, or risk interference with foreground request servicing. Ideally, it should be possible to colocate throughput-oriented services such as MapReduce [8] or video processing [17] with low-latency services, such that resources are fully occupied by the throughput-oriented services when not needed by the low-latency services. However, this is rarely attempted in practice because it impacts the performance of the latency-sensitive services.

One of the reasons it is difficult to combine low latency and high throughput is that 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. As a result, applications cannot adjust their internal parallelism to match the resources available to them, and they cannot use application-specific knowledge to optimize their use of resources. This can lead to both under-utilization and over-commitment of cores, which results in poor resource utilization and/or suboptimal performance. The only recourse for applications is to pin threads to cores; this results in under-utilization of cores within the application and does not prevent other applications from being scheduled onto the same cores.

Arachne is a thread management system that solves these problems by giving applications visibility into the physical resources they are using. We call this approach *core-aware thread management*. In Arachne, application threads are managed entirely at user level; they are not visible to the operating system. Applications negotiate with the system over cores, not threads. Cores are allocated for the exclusive use of individual applications and remain allocated to an application for long intervals (tens of milliseconds). 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.

User-level thread management systems have been implemented many times in the past [34, 12, 4] and the basic features of Arachne were prototyped in the early 1990s in the form of scheduler activations [2]. Arachne is novel in the following ways:

- 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.
- 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. Arachne applications can coexist with traditional applications that do not use Arachne.

We have implemented the Arachne runtime and core arbiter in C++ and evaluated them using both synthetic benchmarks and the memcached and RAMCloud storage systems. Arachne can initiate a new thread on a different core (with load balancing) in about 220 ns, and an application can obtain an additional core from the core arbiter in 20–30 μs. When Arachne was added to memcached, it reduced tail latency by more than 10x and allowed 37% higher throughput at low latency. Arachne also improved performance isolation; a background video processing application could be colocated with memcached with almost no performance loss. When Arachne was added to the RAMCloud storage system, it improved write throughput by more than 2.5x.

2 The Threading Problem

Arachne was motivated by the challenges in creating services that process very large numbers of very small requests. These services can be optimized for low latency or for high throughput, but it is difficult to achieve both with traditional threads implemented by the operating system.

As an example, consider memcached [18], a widely used in-memory key-value-store. Memcached processes requests in about 10 μs. Kernel threads are too expensive to create a new one for each incoming request, so memcached uses a fixed-size pool of worker threads. New connections are assigned statically to worker threads in a round-robin fashion by a separate dispatch thread.

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. For best performance, one core must be reserved for each worker thread. If background tasks are run on the machine during periods of low load, they are likely to interfere with the memcached workers, due to the large number of distinct worker threads. Furthermore, 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. Memcached would perform better if it could scale back during periods of low load to use a smaller number of kernel threads (and cores) more intensively.

In addition, memcached's static allocation of connections to workers can result in load imbalances under skewed workloads, with some worker threads overloaded and others idle. This can impact both latency and throughput.

The RAMCloud storage system provides another example [25]. RAMCloud is an in-memory key-value store that processes small read requests in about 2 μs. Like memcached, it is based on kernel threads. A dispatch thread handles all network communication and polls the NIC for incoming packets using kernel bypass. When a request arrives, the dispatch thread delegates it to one of a collection of worker threads for execution; this approach avoids problems with skewed workloads. The number of active worker threads varies based on load. The maximum number of workers is determined at startup, which creates issues similar to memcached.

RAMCloud implements nested requests, which result in poor resource utilization. When a worker thread receives a write request, it sends copies of the new value to backup servers and waits for those requests to return before responding to the original request. All of the replication requests complete within 7-8 μs, so the worker busy-waits for them. If the worker were to sleep, it would take several microseconds to wake it up again; in addition, context-switching overheads are too high to get much useful work done in such a short time. As a result, the worker thread's core is wasted for 70-80% of the time to process a write request; write throughput for a server is only about 150 kops/sec for small writes, compared with about 1 Mops/sec for small reads.

The goal for Arachne is to provide a thread management system that allows a better combination of low latency and high throughput. For example, 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. In addition, Arachne should provide an implementation of user-level threads that is efficient enough to be used for very short-lived threads, and that allows useful work to be done during brief blockages such as those for nested requests.

3 Arachne Overview

Figure 1 shows the overall architecture of Arachne. Three components work together to implement Arachne threads. 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

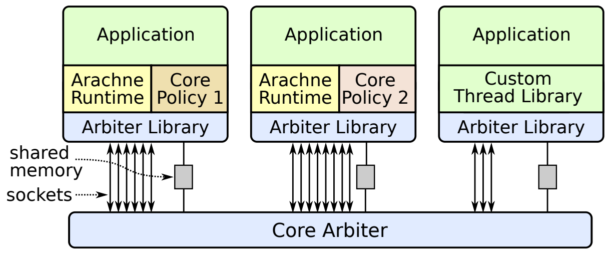


Figure 1: The Arachne architecture. The core arbiter communicates with each application using one socket for each kernel thread in the application, plus one page of shared memory.

for the Arachne runtime and core policy, while still using the core arbiter.

The core arbiter 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. The core arbiter adjusts the core allocations as application requirements change. Section 4 describes the core arbiter in detail.

The Arachne runtime creates several kernel threads and uses them to implement user threads, which are used by Arachne applications. The Arachne user thread abstraction contains facilities similar to thread packages based on kernel threads, including thread creation and deletion, locks, and condition variables. However, all operations on user threads are carried out entirely at user level without making kernel calls, so they are an order of magnitude faster than operations on kernel threads. Section 5 describes the implementation of the Arachne runtime in more detail.

The Arachne runtime works together with a core policy, which determines how cores are used by that application. The core policy computes the application's core requirements, using performance information gathered by the Arachne runtime. It also determines which user threads run on which cores. Each application chooses its core policy. Core policies are discussed in Section 6.

Arachne uses kernel threads as a proxy for cores. Each kernel thread created by the runtime executes on a separate core and has exclusive access to that core while it is running. When the arbiter allocates a core to an application, it unblocks one of the application's kernel threads on that core; when the core is removed from an application, the kernel thread running on that core blocks. The Arachne runtime runs a simple dispatcher in each kernel thread, which multiplexes several user threads on the associated core.

Arachne uses a cooperative multithreading model for user threads: the runtime does not preempt a user thread once it has begun executing. If a user thread needs to execute for a long time without blocking, it must occasionally invoke a `yield` method, which allows other threads to run before the calling thread continues. We expect most threads to either block or complete quickly, so it should rarely be necessary to invoke `yield`.

One potential problem with a user-level implementation of threads is that a user thread might cause the underlying kernel thread to block. This could happen, for example, if the user thread invokes a blocking kernel call or incurs a page fault. This prevents the kernel thread from running other user threads until the kernel call or page fault completes. Previous implementations of user-level threads have attempted to work around this inefficiency in a variety of ways, often involving complex kernel modifications.

Arachne does not take any special steps to handle blocking kernel calls or page faults. Most modern operating systems support asynchronous I/O, so I/O can be implemented without blocking the kernel thread. Paging is almost never cost-effective today, given the low cost of memory and the large sizes of memories. Modern servers rarely incur page faults except for initial application loading. Thus, for simplicity, Arachne does not attempt to make use of the time when a kernel thread is blocked for a page fault or kernel call.

Note: we use the term *core* to refer to any hardware mechanism that can support an independent thread of computation. In processors with hyperthreading, we think of each hyperthread as a separate logical core, even though some of them share a single physical core.

4 The Core Arbiter

This section describes how the core arbiter claims control over (most of) the system's cores and allocates them among applications. The core arbiter has three interesting features. First, it implements core management entirely at user level using existing Linux mechanisms; it does not require any kernel changes. Second, it coexists with existing applications that don't use Arachne. And third, it takes a cooperative approach to core management, both in its priority mechanism and in the way it preempts cores from applications.

The core arbiter runs as a user process and uses the Linux *cpuset* mechanism to manage cores. A cpuset is a collection of one or more cores and one or more banks of memory. At any given time, each kernel thread is assigned to exactly one cpuset, and the Linux scheduler ensures that the thread executes only on cores in that cpuset. By default, all threads run in a cpuset containing all cores and all memory banks. The core arbiter uses cpusets to allocate specific cores to specific applications.

The core arbiter divides cores into two groups: *managed cores* and *unmanaged cores*. Managed cores are allocated by the core arbiter; only the kernel threads created by Arachne run on these cores. Unmanaged cores continue to be scheduled by Linux. They are used by processes that do not use Arachne, and also by the core arbiter itself. In addition, if an Arachne application creates new kernel threads outside Arachne, for example, using `std::thread`, these threads will run on the unmanaged cores.

When the core arbiter starts up, it creates one cpuset for unmanaged cores (the *unmanaged cpuset*) and places all of the system's cores into that set. It then assigns every existing kernel thread (including itself) to the unmanaged cpuset; any new threads spawned by these threads will also run on this cpuset. The core arbiter also creates one *managed cpuset* corresponding to each core, which contains that single core but initially has no threads assigned to it. To allocate a core to an Arachne application, the arbiter removes that core from the unmanaged cpuset and assigns an Arachne kernel thread to the managed cpuset for that core. When a core is no longer needed by any Arachne application, the core arbiter adds the core back to the unmanaged cpuset.

This scheme allows Arachne applications to coexist with traditional applications whose threads are managed by the Linux kernel. Arachne applications receive preferential access to cores, except that the core arbiter reserves at least one core for the unmanaged cpuset.

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 (see below for details).
- `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. At that point the kernel thread wakes up and this method returns the identifier of the assigned 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.

Normally, the Arachne runtime handles all communication with the core arbiter, so these methods are invisible to applications. However, an application can implement its own thread and core management by calling the arbiter library package directly.

The methods described above communicate with the core arbiter using a collection of Unix domain sockets and a shared memory page (see Figure 1). The arbiter library opens one socket for each kernel thread. This socket is used to send requests to the core arbiter, and it is also used to put the kernel thread to sleep when it has no assigned core. The shared memory page is used by the core arbiter to pass information to the arbiter library; it is written by the core arbiter and is read-only to the arbiter library.

When the Arachne runtime starts up, it invokes `setRequestedCores` to specify the application's initial core requirements; `setRequestedCores` sends a message to the core arbiter over a socket. Then the runtime creates one kernel thread for each core on the machine; all of these threads invoke `blockUntilCoreAvailable`. `blockUntilCoreAvailable` sends a request to the core arbiter over the socket belonging to that kernel thread and then attempts to read a response from the socket. This has two effects: first, it notifies the core arbiter that the kernel thread is available for it to manage (the request includes the Linux identifier for the thread); second, the socket read puts the kernel thread to sleep.

At this point the core arbiter knows about the application's core requirements and all of its kernel threads, and the kernel threads are all blocked. When the core arbiter decides to allocate a core to the application, it chooses one of the application's blocked kernel threads to run on that core. It assigns that thread to the cpuset corresponding to the allocated core and then sends a response message back over the thread's socket. This causes the thread to wake up, and Linux will schedule the thread on the given core; the `blockUntilCoreAvailable` method returns, with the core identifier as its return value. The kernel thread then invokes the Arachne dispatcher to run user threads.

If the core arbiter wishes to reclaim a core from an application, it asks the application to release the core. The core arbiter does not unilaterally preempt cores, since the core's kernel thread might be in an inconvenient state (e.g. it might have acquired an important spin lock); abruptly stopping it could have significant performance consequences for the application. So, the core arbiter sets a variable in the shared memory page, indicating which core(s) should be released. Then it waits for the application to respond.

Each kernel thread is responsible for testing the information in shared memory at regular intervals by invoking `mustReleaseCore`. The Arachne runtime does this in its dispatcher. If `mustReleaseCore` returns true, then the kernel thread cleans up as described in Section 5.4 and invokes `blockUntilCoreAvailable`. This notifies the core arbiter and puts the kernel thread to sleep. At this point, the core arbiter can reallocate the core to a different application.

The communication mechanism between the core arbiter and applications is intentionally asymmetric: requests from applications to the core arbiter use sockets, while requests from the core arbiter to applications use shared memory. The sockets are convenient because they allow the core arbiter to sleep while waiting for requests; they also allow application kernel threads to sleep while waiting for cores to be assigned. Socket communication is relatively expensive (several microseconds in each direction), but it only occurs when application core requirements change, which we expect to be infrequent. The shared memory page is convenient because it allows the Arachne runtime to test efficiently for incoming requests from the core arbiter; these tests are made frequently (every pass through the user thread dispatcher), so it is important that they are fast and do not involve kernel calls.

Applications can delay releasing cores for a short time in order to reach a convenient stopping point, such as a time when no locks are held. The Arachne runtime will not release a core until the dispatcher is invoked on that core, which happens when a user thread blocks, yields, or exits.

If an application fails to release a core within a timeout period (currently 10 ms), then the core arbiter will forcibly reclaim the core. It does this by reassigning the core's kernel thread to the unmanaged cpuset. The kernel thread will be able to continue executing, but it will probably experience degraded performance due to interference from other threads in the unmanaged cpuset.

The core arbiter uses a simple priority mechanism for allocating cores to applications. Arachne applications can request cores on each of eight priority levels. The core arbiter allocates cores from highest priority to lowest, so low-priority applications may receive no cores. If there are not enough cores for all of the requests at a particular level, the core arbiter divides the cores evenly among the requesting applications. The core arbiter repeats this computation whenever application requests change. The arbiter allocates all of the hyperthreads of a particular hardware core to the same application whenever possible. The core arbiter also attempts to keep all of an application's cores on the same socket.

This policy for core allocation assumes that the applications running on a given system will cooperate in their choice of priority levels: a misbehaving application could starve other applications by requesting all of its cores at the highest priority level. Anti-social behavior could be prevented by requiring applications to authenticate with the core arbiter when they first connect, and allowing system administrators to set limits for each application. We leave such a mechanism to future work.

5 The Arachne Runtime

This section discusses how the Arachne runtime implements user threads. The most important goal for the runtime is to provide a fast and scalable implementation of user threads for modern multi-core hardware. We want Arachne to support granular computations, which consist of large numbers of extremely short-lived threads. For example, a low latency server might create a new thread for each incoming request, and the request might take only a microsecond or two to process; the server might process millions of these requests per second.

5.1 Cache-optimized design

The performance of the Arachne runtime is dominated by cache misses. Most threading operations, such as creating a thread, acquiring a lock, or waking a blocked thread, are relatively simple, but they involve communication between cores. Cross-core communication requires 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. On our servers, the cache misses to transfer a value from one core to another in the same socket take 7-8x as long as a context switch between user threads on the same core. Transfers between sockets are even more expensive. Thus, our most important goal in implementing user threads was to minimize cache misses.

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 (modern processors have out-of-order execution engines that can continue executing instructions while waiting for cache misses, and each core has multiple memory channels). Thus, additional cache misses are essentially free. However, modern processors have an out-of-order execution limit of about 100 instructions, so code must be designed to concentrate likely cache misses near each other.

Similarly, a computation that takes tens of nanoseconds in isolation may actually have zero marginal cost if it occurs in the vicinity of a cache miss; it will simply fill the time while the cache miss is being processed. Section 5.3 will show how the Arachne dispatcher uses this technique to hide the cost of seemingly expensive code.

5.2 Thread creation

Many user-level thread packages, such as the one in Go [12], 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. However, work stealing is an expensive operation (it requires cache misses), which is particularly noticeable for short-lived threads. Work stealing also introduces a time lag before a thread is stolen to an unloaded core, which impacts service latency. For Arachne we decided to perform load-balancing at thread creation time; our goal is to get a new thread on an unloaded core as quickly as possible. By optimizing this mechanism based on cache misses, we were able to achieve thread creation times competitive with systems that create child threads on the parent's core.

Cache misses can occur during thread creation for the following reasons:

- **Load balancing**: Arachne must choose a core for the new thread in a way 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. Depending on how this information is managed, it can result in additional cache misses.

We describe below how Arachne can create a new user thread with as few as four cache misses.

In order to minimize cache misses for thread contexts, Arachne binds each thread context to a single core (the context is only used by a single kernel thread). Each user thread is assigned to a thread context when it is created, and the thread executes only on the context's associated core. Most threads live their entire life on a single core. A thread moves to a different core only as part of an explicit migration. This happens only in rare situations such as when the core arbiter reclaims a core. A thread context remains bound to its core after its thread completes, and Arachne reuses recently-used contexts when creating new threads. If threads have short lifetimes, it is likely that the context for a new thread will already be cached.

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. The state consists of a 64-bit `maskAndCount` value for each active core. 56 bits of the value are a bit mask indicating which of the core's thread contexts are currently in use, and the remaining 8 bits are a count of the number of ones in the mask.

When creating new threads, Arachne uses the "power of two choices" approach for load balancing [21]. 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. If the compare-and-swap fails because of a concurrent update, Arachne rereads the `maskAndCount` for the chosen core and repeats the process of allocating a thread context. This creation mechanism is scalable: with a large number of cores, multiple threads can be created simultaneously on different cores.

Once a thread context has been allocated, Arachne copies the address and arguments for the thread's top-level method into the context and schedules the thread for execution by setting a scheduling state variable. In order to minimize cache misses, Arachne uses a single cache line to hold all of this information. This limits argument lists to 6 one-word parameters on machines with 64-byte cache lines; larger parameter lists must be passed by reference, which will result in additional cache misses.

With this mechanism, a new thread can be invoked in four cache miss times in the best case. One cache miss is required to read the `maskAndCount` and three cache miss times are required to transfer the line containing the method address and arguments and the scheduling flag.

5.3 Thread scheduling

The traditional approach to thread scheduling uses one or more ready queues to identify runnable threads (typically one queue per core, to reduce contention), plus a scheduling state variable for each thread, which indicates whether that thread is runnable or blocked. This representation is problematic from the standpoint of cache misses. Adding or removing an entry to/from a ready queue requires updates to multiple variables. Even if the queue is lockless, this is likely to result in multiple cache misses when the queue is shared across cores. Furthermore, we expect sharing to be common: a thread must be added to the ready queue for its core when it is awakened, but the wakeup typically comes from a thread on a different core.

In addition, the scheduling state variable is subject to races. For example, if a thread blocks on a condition variable, but another thread notifies the condition variable before the blocking thread has gone to sleep, a race over the scheduling state variable could cause the wakeup to be lost. This race is typically eliminated with a lock that controls access to the state variable. However, the lock results in additional cache misses, since it is shared across cores.

In order to minimize cache misses, Arachne does not use ready queues. Instead of checking a ready queue, the 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 turns out to be relatively efficient, for two reasons. First, we expect only a few thread contexts to be occupied for a core at a given time (there is no need to keep around blocked threads for intermittent tasks; a new thread can be created for each task). Second, the cost of scanning the active thread contexts is largely hidden by an unavoidable cache miss on the scheduling state variable for the thread that woke up. This variable is typically modified by a different core to wake up the thread, which means the dispatcher will have to take a cache miss to observe the new value. 100 or more cycles elapse between when the previous value of the variable is invalidated in the dispatcher's cache and the new value can be fetched; a large number of thread contexts can be scanned during this time. Section 7.4 evaluates the cost of this approach.

Arachne also uses a new lockless mechanism for scheduling state. The scheduling state of a thread is represented with a 64-bit `wakeupTime` variable in its thread context. The dispatcher considers a thread runnable if its `wakeupTime` is less than or equal to the processor's fine-grain cycle counter. Before transferring control to a thread, the dispatcher sets its `wakeupTime` to the largest possible value. `wakeupTime` doesn't need to be modified when the thread blocks: the large value will prevent the thread from running again. To wake up the thread, `wakeupTime` is set to 0. This approach eliminates the race condition described previously, since `wakeupTime` is not modified when the thread blocks; thus, no synchronization is needed for access to the variable.

The `wakeupTime` variable also supports timer-based wakeups. If a thread wishes to sleep for a given time period, or if it wishes to add a timeout to some other blocking operation such as a condition `wait`, it can set `wakeupTime` to the desired wakeup time before blocking. A single test in the Arachne dispatcher detects both normal unblocks and timer-based unblocks.

Arachne exports the `wakeupTime` mechanism to applications with two methods:

- `block(time)` will block the current user thread. The `time` argument is optional; if it is specified, `wakeupTime` is set to this value (using compare-and-swap to detect concurrent wakups).
- `signal(thread)` will set the given user thread's `wakeupTime` to 0.

These methods make it easy to construct higher-level synchronization and scheduling operators. For example, the `yield` method, which is used in cooperative multithreading to allow other user threads to run, simply invokes `block(0)`.

5.4 Adding and releasing cores

When the core arbiter allocates a new core to an application, it wakes up one of the kernel threads that was blocked in `blockUntilCoreAvailable`. The kernel thread notifies the core policy of the new core as described in Section 6 below, then it enters the Arachne dispatcher loop.

When the core arbiter decides to reclaim a core from an application, `mustReleaseCore` will return true in the Arachne dispatcher running on the core. The kernel thread modifies its `maskAndCount` to prevent any new threads from being placed on it, then it notifies the core policy of the reclamation. If any user threads exist on the core, the Arachne runtime migrates them to other cores (we omit the details of this mechanism, due to space limitations). Once all threads have been migrated away, the kernel thread on the reclaimed core invokes `blockUntilCoreAvail-able`. This notifies the core arbiter that the core is no longer in use and puts the kernel thread to sleep.

6 Core Policies

One of our goals for Arachne is to give applications precise control over their usage of cores. For example, in RAMCloud the central dispatch thread is usually the performance bottleneck. Thus, it makes sense for the dispatch thread to have exclusive use of a core. Furthermore, the other hyperthread on the same physical core should be idle (if both hyperthreads are used simultaneously, they each run about 30% slower than if only one hyperthread is in use). In other applications it might be desirable to colocate particular threads on hyperthreads of the same core or socket, or to force all low-priority background threads to execute on a single core in order to maximize the resources available for foreground request processing.

The Arachne runtime does not implement the policies for core usage. These are provided in a separate *core policy* module. Each application selects a particular core policy at startup. Over time, we expect Arachne to incorporate a few simple core policies that handle the needs of most applications; applications with special requirements can implement custom core policies outside of Arachne.

In order to manage core usage, the core policy must know which cores have been assigned to the application. The Arachne runtime provides this information by invoking a method in the core policy whenever the application gains or loses cores.

When an application creates a new user thread, it specifies an integer *thread class* for the thread. Thread classes are used by core policies to manage user threads; each thread class corresponds to a particular level of service, such as "foreground thread" or "background thread." Each core policy defines its own set of valid thread classes. The Arachne runtime stores thread classes with threads, but has no knowledge of how they are used.

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. The Arachne runtime places the new thread on one of those cores using the "power of two choices" mechanism described in Section 5. If the core policy wishes to place the new thread on a specific core, `getCores` can return a list with a single entry. Arachne also invokes `getCores` to find a new home for a thread when it must be migrated as part of releasing a core.

One of the unusual features of Arachne is that each application is responsible for determining how many cores it needs; we call this *core estimation*, and it is handled by the core policy. The Arachne runtime measures two statistics for each core, which it makes available to the core policy for its use in core estimation. The first statistic is *utilization*, which is the average fraction of time that each Arachne kernel thread spends executing user threads. The second statistic is *load factor*, which is an estimate of the average number of runnable user threads on that core. Both of these are computed with a few simple operations in the Arachne dispatching loop.

6.1 Default core policy

Arachne currently includes one core policy; we used the default policy for all of the performance measurements in Section 7. The default policy supports two thread classes: exclusive and normal. Each exclusive thread runs on a separate core reserved for that particular thread; when an exclusive thread is blocked, its core is idle. 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.

6.2 Core estimation

The default policy requests one core for each exclusive thread, plus additional cores for normal threads. Estimating the cores required for the normal threads requires making a tradeoff between core utilization and fast response time. If we attempt to keep cores busy 100% of the time, fluctuations in load will create a backlog of pending threads, resulting in delays for new threads. On the other hand, we could optimize for fast response time, but this would result in low utilization of cores. The more bursty a workload, the more resources it must waste in order to get fast response.

The default policy uses different mechanisms for scaling up and scaling down. The decision to scale up is based on load factor: when the average load factor across all cores running normal threads reaches a threshold value, the core policy increases its requested number of cores by 1. We chose this approach because load factor is a fairly intuitive proxy for response time; this makes it easier for users to specify a non-default value if needed. In addition, performance measurements showed that load factor works better than utilization for scaling up: a single load factor threshold works for a variety of workloads, whereas the best utilization for scaling up depends on the burstiness and overall volume of the workload.

On the other hand, scaling down is based on utilization. Load factor is hard to use for scaling down because the metric of interest is not the current load factor, but rather the load factor that will occur with one fewer core; this is hard to estimate. Instead, the default core policy records the total utilization (sum of the utilizations of all cores running normal threads) each time it increases its requested number of cores. When the utilization returns to a level slightly less than this, the runtime reduces its requested number of cores by 1 (the "slightly less" factor provides hysteresis to prevent oscillations). A separate scale-down utilization is recorded for each distinct number of requested cores.

Overall, three parameters control the core estimation mechanism: the load factor for scaling up, the interval over which statistics are averaged for core estimation, and the hysteresis factor for scaling down. The default core policy currently uses a load factor threshold of 2.25, an averaging interval of 50 ms, and a hysteresis factor of 5% utilization.

CloudLab m510[30]	
CPU	Xeon D-1548 (8 x 2.0 GHz cores)
RAM	64 GB DDR4-2133 at 2400 MHz
Disk	Toshiba THNSN5256GPU7 (256 GB)
NIC	Dual-port Mellanox ConnectX-3 10 Gb
Switches	HPE Moonshot-45XGc

Table 1: Hardware configuration used for benchmarking. All nodes ran Linux 4.4.0. C-States were enabled and Meltdown mitigations were disabled. Hyperthreads were enabled (2 hyperthreads per core).

Benchmark	Arachne	std::thread	Go	uThreads
Thread Creation	217 ns	13329 ns	444 ns	6132 ns
One-Way Yield	93 ns	–	–	79 ns
Null Yield	12 ns	–	–	6 ns
Condition Notify	281 ns	4962 ns	483 ns	4976 ns
Signal	282 ns	–	–	–

Table 2: Median cost of scheduling primitives. Arachne creates all threads on a different core from the parent. Go always creates Goroutines on the parent's core. uThreads uses a round-robin approach to assign threads to cores; when it chooses the parent's core, the median cost drops to 250 ns. Creation, notification, and signaling are measured from initiation in one thread until the target thread wakes up and begins execution. In "One-Way Yield", control passes from the yielding thread to another runnable thread on the same core. In "Null Yield", there are no other runnable threads, so control returns to the yielding thread.

7 Evaluation

We implemented Arachne in C++ on Linux. The core arbiter contains 4200 lines of code, the runtime contains 3400 lines, and the default core policy contains 270 lines.

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?
- How do Arachne's internal mechanisms, such as its queue-less approach to thread scheduling and its mechanisms for core estimation and core allocation, contribute to performance?

Table 1 describes the configuration of the machines used for benchmarking.

7.1 Threading Primitives

Table 2 compares the cost of basic thread operations in Arachne with C++ `std::thread`, Go, and uThreads [4]. `std::thread` is based on kernel threads; Go implements threads at user level in the language runtime; and uThreads uses kernel threads to multiplex user threads, like Arachne. uThreads is a highly rated C++ user threading library on

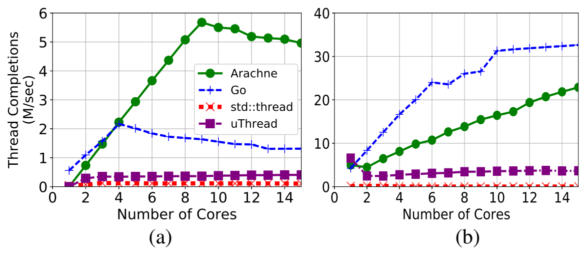


Figure 2: Thread creation throughput as a function of available cores. In (a) a single thread creates new threads as quickly as possible; each child consumes 1 µs of execution time and then exits. In (b) 3 initial threads are created for each core; each thread creates one child and then exits.

GitHub and claims high performance. The measurements use small microbenchmarks, so they represent best-case performance.

Arachne's thread operations are considerably faster than any of the other systems, except that yields are faster in uThreads. Arachne's cache-optimized design performs thread creation twice as fast as Go, even though Arachne places new threads on a different core from the parent while Go creates new threads on the parent's core.

We designed Arachne's thread creation mechanism not just to minimize latency, but also to provide high throughput. We ran two experiments to measure thread creation throughput. In the first experiment (Figure 2(a)), a single "dispatch" thread creates new threads as quickly as possible (this situation might occur, for example, if a single thread is polling a network interface for incoming requests). A single Arachne thread can spawn more than 5 million new threads per second, which is 2.5x the rate of Go and at least 10x the rate of `std::thread` or uThreads. This experiment demonstrates the benefits of performing load balancing at thread creation time. Go's work stealing approach creates significant additional overhead, especially when threads are short-lived, and the parent's work queue can become a bottleneck.

The second experiment measures thread creation throughput using a distributed approach, where each of many existing threads creates one child thread and then exits (Figure 2(b)). In this experiment both Arachne and Go scaled in their throughput as the number of available cores increased. Neither uThreads nor `std::thread` had scalable throughput; uThreads had 10x less throughput than Arachne or Go and `std::thread` had 100x less. Go's approach to thread creation worked well in this experiment. It eliminated cache coherence overheads and contention between creators, and there was no need for work stealing since the load naturally balanced itself. As a result, Go's throughput was 1.5–2.5x that of Arachne.

7.2 Arachne's benefits for memcached

We modified memcached [18] version 1.5.6 to use Arachne. In the modified version ("memcached-A"), the pool of worker threads is replaced by a single dispatch thread, which waits for incoming requests on all connections. When a request arrives, the dispatch thread creates a new Arachne thread, which lives only long enough to handle all available requests on the connection. Memcached-A uses the default core policy; the dispatch thread is "exclusive" and workers are "normal" threads.

Memcached-A provides two benefits. First, it reduces performance interference, both between kernel threads (there is no multiplexing) and between applications (cores are dedicated to applications). Second, memcached-A provides finer-grain load-balancing (at the level of individual requests rather than connections).

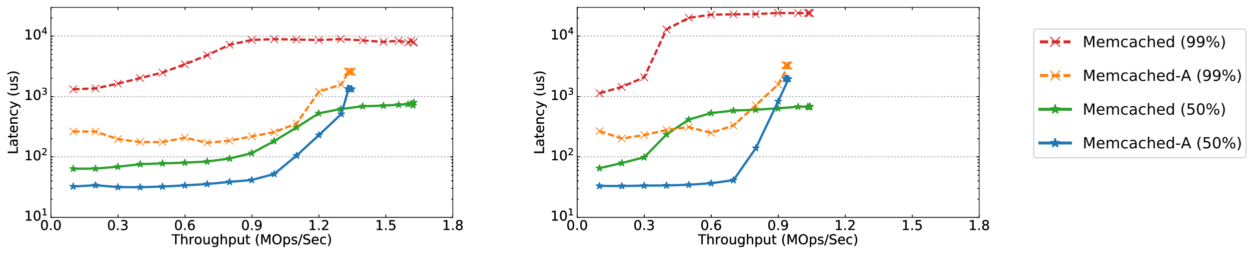
We performed three experiments with memcached; their configurations are summarized in Table 3. The first experiment, Realistic, measures latency as a function of load under realistic conditions; it uses the Mutilate benchmark [14, 22] to recreate the Facebook ETC workload [3]. Figure 3(a) shows the results. The maximum throughput of memcached-A is about 20% lower than memcached (memcached-A has two fewer cores, since one core is reserved for unmanaged threads and one for the dispatcher; in addition, Arachne's thread creation mechanism adds overhead). However, memcached-A has significantly lower latency, so it provides higher usable throughput for applications with service-level requirements. For example, if an application requires a median latency less than 100 µs, memcached-A can support 37.5% higher throughput than memcached (1.1 Mops/sec vs. 800 Kops/sec). At the 99th percentile, memcached-A's latency ranges from 3–40x lower than memcached. We found that Linux migrates memcached threads between cores frequently: at high load, each thread migrates about 10 times per second; at low load, threads migrate about every third request. Migration adds overhead and increases the likelihood of multiplexing.

One of our goals for Arachne is to adapt automatically to application load and the number of available cores, so administrators do not need to specify configuration options or reserve cores. Figure 3(b) shows memcached's behavior when it is given fewer cores than it would like. For memcached, the 16 worker threads were multiplexed on only 8 cores; memcached-A was limited to at most 8 cores. Maximum throughput dropped for both systems, as expected. Arachne continue to operate efficiently: latency was about the same as in Figure 3(a). In contrast, memcached experienced significant increases in both median and tail latency, presumably due to additional multiplexing; with a latency limit of 100 µs, memcached could only handle 300 Kops/sec, whereas memcached-A handled 780 Kops/sec.

The second experiment, Colocation, varied the load dynamically to evaluate how well Arachne's core estimator responded. It also measured memcached and memcached-A performance when colocated with a compute-intensive application (the x264 video encoder [20]). The results are in Figure 4. The top graph shows that memcached-A used

Experiment	Program	Keys	Values	Items	PUTs	Clients	Threads	Conns	Pipeline	IR Dist
Realistic	Mutilate [22]	ETC	ETC	1M	.03	20+1	16+8	1280+8	1+1	GPareto
Colocation	Memtier [19]	30B	200B	8M	0	1+1	16+8	320+8	10+1	Poisson
Skew	Memtier	30B	200B	8M	0	1	16	512	100	Poisson

Table 3: Configurations of memcached experiments. Program is the benchmark program used to generate the workload (our version of Memtier is modified from the original). Keys and Values give sizes of keys and values in the dataset (ETC recreates the Facebook ETC workload [3], which models actual usage of memcached). Items is the total number of objects in the dataset. PUTs is the fraction of all requests that were PUTs (the others were GETs). Clients is the total number of clients (20+1 means 20 clients generated an intensive workload, and 1 additional client measured latency using a lighter workload). Threads is the number of threads per client. Conns is the total number of connections per client. Pipeline is the maximum number of outstanding requests allowed per connection before shedding workload. IR Dist is the inter-request time distribution. Unless otherwise indicated, memcached was configured with 16 worker threads and memcached-A scaled automatically between 2 and 15 cores.



(a) memcached: 16 worker threads, 16 cores

(b) memcached: 16 workers; both: 8 cores

Figure 3: Median and 99th-percentile request latency as a function of achieved throughput for both memcached and memcached-A, under the Realistic benchmark. Each measurement ran for 30 seconds after a 5-second warmup. Y-axes use a log scale.

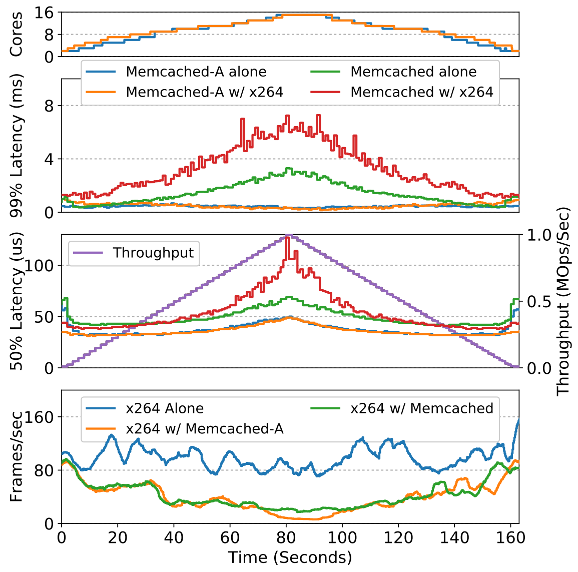


Figure 4: Memcached performance in the Colocation experiment. Throughput increased gradually from 10 Kops/sec to 1 Mops/sec and then decreased back to 10 Kops/sec. In some experiments the x264 video encoder [20] ran concurrently, using the raw video file (sintel-1280.y4m) from Xiph.org [16]. Top graph: number of cores allocated to memcached-A over time. Middle graphs: 99th percentile and median tail latency for memcached and memcached-A. Bottom graph: throughput of the video decoder (averaged over trailing 4 seconds) when running by itself or with memcached or memcached-A.

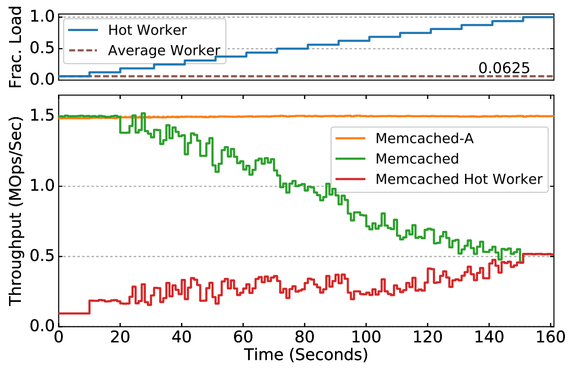


Figure 5: The impact of workload skew on memcached performance with a target load of 1.5 Mops/sec. Initially, the load was spread evenly over all connections; over time, an increasing fraction of the total workload was directed to connections serviced by a single "hot" memcached worker thread (top graph). The bottom graph shows the overall throughput, as well as the throughput of the overloaded worker thread in memcached.

only 2 cores at low load (dispatch and one worker) and ramped up to use all available cores as the load increased. Memcached-A maintained near-constant median and tail latency as the load increased, which indicates that the core estimator chose good points at which to change its core requests. Memcached's latency was higher than memcached-A and it varied more with load; even when running without the background application, 99th-percentile latency was 10x higher for memcached than for memcached-A.

When memcached was colocated with the video application, its latency doubled, both at the median and at the

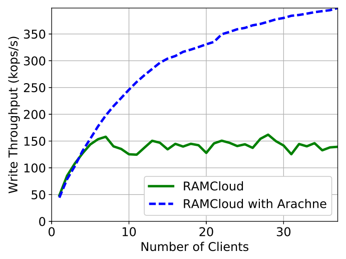


Figure 6: Throughput of a single RAMCloud server when many clients perform continuous back-to-back write RPCs of 100-byte objects. Throughput is measured as the number of completed writes per second.

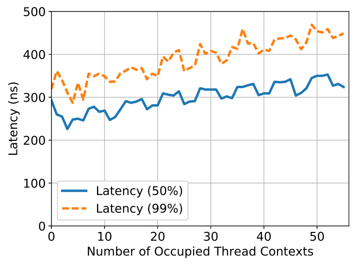


Figure 7: Cost of signaling a blocked thread as the number of threads on the target core increases. Latency is measured from immediately before signaling on one core until the target thread resumes execution on a different core.

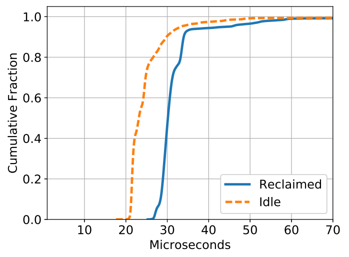


Figure 8: Cumulative distribution of latency from core request to core acquisition (a) when the core arbiter has a free core available and (b) when it must reclaim a core from a competing application.

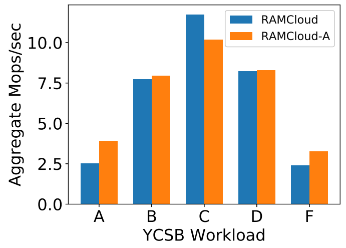


Figure 9: Comparison between RAMCloud and RAMCloud-A on a modified YCSB benchmark [7] using 100-byte objects. Both were run with 12 storage servers. Y-values represent aggregate throughput across 30 independent client machines, each running with 8 threads.

99th percentile. In contrast, memcached-A was almost completely unaffected by the video application. This indicates that Arachne's core-aware approach improves performance isolation between applications. Figure 4 shows that memcached-A ramped up its core usage more quickly when colocated with the video application. This suggests that there was some performance interference from the video application, but that the core estimator detected this and allocated cores more aggressively to compensate.

The bottom graph in Figure 4 shows the throughput of the video application. At high load, its throughput when colocated with memcached-A was less than half its throughput when colocated with memcached. This is because memcached-A confined the video application to a single unmanaged core. With memcached, Linux allowed the video application to consume more resources, which degraded the performance of memcached.

The final experiment for memcached is Skew, shown in Figure 5. If the workload for memcached doesn't match the static partitioning of its connections among workers, overall throughput suffers. In contrast, skew has no impact on memcached-A.

7.3 Arachne's Benefits for RAMCloud

We also modified RAMCloud [25] to use Arachne. In the modified version ("RAMCloud-A"), the long-running pool of worker threads is eliminated, and the dispatch thread creates a new worker thread for each request. Threads that are busy-waiting on nested RPCs yield after each iteration of their polling loop. This allows other requests to be processed during the waiting time, so that the core isn't wasted. Figure 6 shows that RAMCloud-A has 2.5x higher write throughput than RAMCloud. On the YCSB benchmark [7] (Figure 9), RAMCloud-A provided 54% higher throughput than RAMCloud for the write-heavy YCSB-A workload. On the read-only YCSB-C workload, RAMCloud-A's throughput was 15% less than RAMCloud, due to the overhead of Arachne's thread invocation mechanism. These experiments demonstrate that Arachne makes it practical to schedule other work during blockages as short as a few microseconds.

7.4 Arachne Internal Mechanisms

This section evaluates several of the internal mechanisms that are key to Arachne's performance. As mentioned in Section 5.3, Arachne forgoes the use of ready queues as part of its cache-optimized design. Consequently, as cores fill with threads, each core must iterate over more and more wakeup flags in its dispatcher loop. To evaluate the cost of scanning these flags, we measured the cost of signaling a particular blocked thread while varying the number of additional blocked threads on the target core; Figure 7 shows the results. Even in the worst case where all 56 thread contexts are occupied, the average cost of waking up a thread increased by less than 100 ns, which is equivalent to about one cache coherency miss. This means that an alternative implementation that avoids scanning all the active contexts must do so without introducing any new cache misses; otherwise its performance will be worse than Arachne.

Figures 8 and 10 show the performance of Arachne's core allocation mechanism. Figure 8 shows the distribution of allocation times, measured end-to-end from when a thread calls `setRequestedCores` until a kernel thread wakes up on the newly-allocated core. In the first scenario, there is an idle core available to the core arbiter, and the cost is merely that of moving a kernel thread to the core and unblocking it. In the second scenario, a core must be reclaimed from a lower priority application so the cost additionally involves signaling another process and waiting

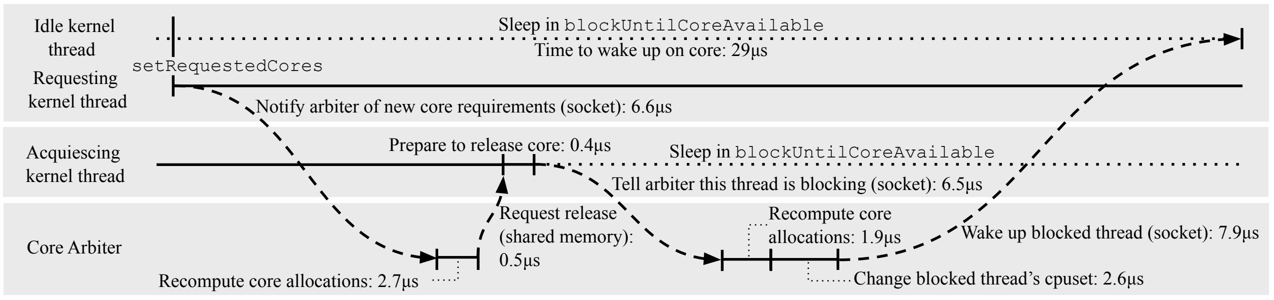

Idle kernel thread
Sleep in blockUntilCoreAvailable
Time to wake up on core: 29µs
setRequestedCores
Requesting kernel thread
Notify arbiter of new core requirements (socket): 6.6µs
Acquiescing kernel thread
Prepare to release core: 0.4µs
Sleep in blockUntilCoreAvailable
Tell arbiter this thread is blocking (socket): 6.5µs
Core Arbiter
Request release (shared memory): 0.5µs
Recompute core allocations: 1.9µs
Wake up blocked thread (socket): 7.9µs
Recompute core allocations: 2.7µs
Change blocked thread's cpuset: 2.6µs

Figure 10: Timeline of a core request to the core arbiter. There are two applications. Both applications begin with a single dedicated core, and the top application also begins with a thread waiting to be placed on a core. The top application has higher priority than the bottom application, so when the top application requests an additional core the bottom application is asked to release its core.

for it to release a core. Figure 8 shows that Arachne can reallocate cores in about 30 µs, even if the core must be reclaimed from another application. This makes it practical for Arachne to adapt to changes in load at the granularity of milliseconds.

Figure 10 shows the timing of each step of a core request that requires the preemption of another process's core. About 80% of the time is spent in socket communication.

8 Related Work

Numerous user-level threading packages have been developed over the last several decades. We have already compared Arachne with Go [12] and uThreads [4]. Boost fibers [1], Folly [11], and Seastar [31] implement user-level threads but do not multiplex user threads across multiple cores. Capriccio [34] solved the problem of blocking system calls by replacing them with asynchronous system calls, but it does not scale to multiple cores. Wikipedia[35] lists 21 C++ threading libraries as of this writing. Of these, 10 offer only kernel threads, 3 offer compiler-based automatic parallelization, 3 are commercial packages without any published performance numbers, and 5 appear to be defunct. None of the systems listed above supports load balancing at thread creation time, the ability to compute core requirements and conform to core allocations, or a mechanism for implementing application-specific core policies.

Scheduler activations [2] are similar to Arachne in that they allocate processors to applications to implement user-level threads efficiently. A major focus of the scheduler activations work was allowing processor preemption during blocking kernel calls; this resulted in significant kernel modifications. Arachne focuses on other issues, such as minimizing cache misses in thread operations, estimating core requirements, and enabling application-specific core policies.

Akaros [29] and Parlib [13] follow in the tradition of scheduler activations. Akaros is a new operating system that allocates dedicated cores to applications and makes all blocking system calls asynchronous; Parlib is a framework for building user schedulers on top of dedicated cores. Akaros offers functionality analogous to the Arachne core arbiter, but it does not appear to have reached a level of maturity that can support meaningful performance measurements.

The traditional approach for managing multi-threaded applications on multi-core machines has been gang scheduling [10, 24]. In gang scheduling, each application unilaterally determines its threading requirements; the operating system then attempts to schedule all of an application's threads simultaneously on different cores. Tucker and Gupta pointed out that gang scheduling results in inefficient multiplexing when the system is overloaded [33]. They argued that it is more efficient to divide the cores so that each application has exclusive use of a few cores; the application can then adjust its degree of parallelism to match the available cores. Arachne implements this approach.

Event-based applications such as Redis [28] and nginx [23] represent an alternative to user threads for achieving high throughput and low latency. Behren et al. [34] argued that event-based approaches are a form of application-specific optimization and such optimization is due to the lack of efficient thread runtimes; Arachne offers efficient threading as a more convenient alternative to events.

Several recent systems, such as IX [6], Zygos [27], and Shenango [32], have combined thread schedulers with high-performance network stacks. These systems share Arachne's goal of combining low latency with efficient resource usage, but they take a more special-purpose approach than Arachne by coupling the threading mechanism to the network stack. Arachne is a general-purpose mechanism; it can be used with high-performance network stacks, such as in RAMCloud, but also in other situations.

9 Conclusion

One of the most fundamental principles in operating systems is virtualization, in which the system uses a set of physical resources to implement a larger and more diverse set of virtual entities. However, virtualization only works if there is a balance between the use of virtual objects and the available physical resources. For example, if the usage of virtual memory exceeds available physical memory, the system will collapse under page thrashing.

Arachne provides a mechanism to balance the usage of virtual threads against the availability of physical cores. Each application computes its core requirements dynamically and conveys that to a central core arbiter, which then allocates cores among competing applications. The core arbiter dedicates cores to applications and tells each application which cores it has received. The application can then use that information to manage its threads. Arachne also provides an exceptionally fast implementation of threads at user level, which makes it practical to use threads even for very short-lived tasks. Overall, Arachne's core-aware approach to thread management enables granular applications that combine both low latency and high throughput.

References

[1] Boost fibers. `http://www.boost.org/doc/libs/1_64_0/libs/fiber/doc/html/fiber/overview.html`, 2013.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[4] S. Barghi. uthreads: Concurrent user threads in c++. `https://github.com/samanbarghi/uThreads`, 2014.

[5] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, Mar. 2017.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51:107–113, January 2008.

[9] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[10] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing*, 16(4):306–318, 1992.

[11] Folly: Facebook open-source library. `https://github.com/facebook/folly`, 2012.

[12] The Go Programming Language. `https://golang.org/`.

[13] K. A. Klues. *OS and Runtime Support for Efficiently Managing Cores in Parallel Applications*. PhD thesis, University of California, Berkeley, 2015.

[14] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, New York, NY, USA, 2014. ACM.

[15] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[16] M. Lora. Xiph. org:: Test media. *World Wide Web electronic publication*, 2008, 1994.

[17] A. Lottarini, A. Ramirez, J. Coburn, M. A. Kim, P. Ranganathan, D. Stodolsky, and M. Wachsler. vbench: Benchmarking video transcoding in the cloud. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 797–809. ACM, 2018.

[18] memcached: a Distributed Memory Object Caching System. `http://www.memcached.org/`.

[19] Memtier benchmark. `https://github.com/RedisLabs/memtier_benchmark`, 2013.

[20] L. Merritt and R. Vanam. x264: A high performance H.264/AVC encoder. `http://neuron2.net/library/avc/overview_x264_v8_5.pdf`, 2006.

[21] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on*

Parallel and Distributed Systems, 12(10):1094–1104, 2001.

[22] Mutilate: high-performance memcached load generator. `https://github.com/leverich/mutilate`, 2015.

[23] Nginx. `https://nginx.org/en/`.

[24] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. 3rd International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[25] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[26] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, CO, 2014. USENIX Association.

[27] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proc. of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.

[28] Redis. http://redis.io.

[29] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving per-node efficiency in the datacenter with new os abstractions. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 25. ACM, 2011.

[30] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), dec 2014.

[31] Seastar. `http://www.seastar-project.org/`, 2014.

[32] Shenango: CPU-Efficient Microsecond-Level Tail Latency for Datacenter Workloads. Submitted for publication (author names omitted to preserve blindness for reviewing), 2018.

[33] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-memory Multiprocessors. In *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 159–166, New York, NY, USA, 1989. ACM.

[34] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 268–281. ACM, 2003.

[35] Wikipedia. List of c++ multi-threading libraries — wikipedia, the free encyclopedia, 2017. [Online; accessed 27-September-2017].