ARACHNE: IMPROVING LATENCY AND EFFICIENCY THROUGH CORE AWARE THREAD MANAGEMENT

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Henry Qin
July 2019

This dissertation is online at: http://purl.stanford.edu/df501xt7856

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Peter Bailis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Matei Zaharia

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Computation and data are rapidly moving into data centers. Today's faster storage systems and faster networks enable the implementation of low latency applications. Unfortunately, today's thread management infrastructure forces applications to choose between low latency and efficient core usage. Operators of latency-sensitive applications must statically provision CPU cores for them to ensure latency SLOs. This is wasteful when such applications are underloaded. Additionally, the applications themselves must be careful in their thread use to avoid oversubscription of cores.

This dissertation presents Arachne, a new system for thread management. Arachne combines low latency with high utilization for threads with lifetimes of microseconds. Arachne changes the scheduling abstraction for applications from one based on threads to one based on cores. A standalone process called the *core arbiter* allocates cores to applications. Each Arachne application determines how many cores it needs based on its load. Applications always know the exact set of cores they have been granted. Arachne also allows applications to control the placement of their threads onto those cores.

Arachne enables applications to achieve a better combination of latency and throughput. The core arbiter can reallocate a core in 30 μs, allowing applications to rapidly respond to load increases. The Arachne runtime linked into each application is optimized to reduce cache misses; it can initiate a new user thread on a different core in 320 ns. 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.

Acknowledgements

This direct cause of this doctoral journey and resulting dissertation was my father, Dehao Qin, encouraging me to apply to graduate school in my senior year at Duke. Were it not for his insistence that I apply for the PhD while my recommmendations were still recent, I would likely have applied for a Master's degree and returned to industry five years ago. Thus, thanks to my father for encouraging me to apply when I was unsure of whether this was a good idea. My mother, Ru Wei, always supported me in making my own career choices, and did not speak on the matter of whether I should do a PhD. However, her own post-PhD career trajectory gave me faith that it would not be disasterous for my career to do a PhD.

When I was deciding between schools, I received the advice to choose not a university or department, but an advisor. I came to Stanford because I had a strong feeling I could work effectively with John, and I still believe it was a great choice. Thanks to my awesome advisor, John Ousterhout, for teaching me to think clearly and communicate effectively in both verbal and written form. John taught me to hold myself and my work to higher standards, and to think deeply about assumptions implicitly held. Perhaps most importantly, I learned that my first ideas can always be improved upon, and one can only arrive at reasonable ideas through iteration. I am grateful for all the times you pointed out bugs in my data, asked me to zero-base my plots, and returned paper drafts with more red ink than black.

Thanks to my awesome labmates Ankita Kejriwal, Stephen Yang, Collin Lee, Seo Jin Park, Jonathan Ellithorpe, Behnam Montazeri, Yilong Li, Jacqueline Speiser and Qian Li for making the last six years full of hilarity, absurd ideas, and stimulating intellectual discussions! I am happy to count you among my friends as well as my labmates. Through sitting in the same space, impromptu design discussions, lab lunches (127 of them since September 2015 alone!), shared tmux sessions, and the many three-hour dry run feedback sessions, I learned from you and grew with you. Our lab's culture will always be the standard against which I measure future work environments.

Social life is just as important as academic life, and so I want to acknowledge a subset of my

labmates for the sheer number of meals we have shared. My records extend only as far back as September 2015, but here are the folks I have eaten with the most, together with the number of meals I have enjoyed with each person as of July 2019, including research lunches.

- Stephen Yang: 587
- Seo Jin Park: 436
- Collin Lee: 431
- Jonathan Ellithorpe: 329

In addition to having eaten the most meals, I want to thank Stephen Yang for patiently teaching me about the physical world.

There are always challenges and questions when entering a new place and interacting with a new group of people. The latency with which difficulties are resolved and questions are answered strongly impacts one's onboarding experience. I owe a big thank you to the first generation of John's students at Stanford, Ryan Stutsman, Diego Ongaro, and Stephen Rumble, for helping me onboard and answering all my silly questions when I first joined John's group, as well as answering all my emails after they had already graduated and in Stephen's case moved out of the country.

Organizing a defense is one of the most difficult final challenges in the PhD journey, because faculty are incredibly busy people. Thanks to Carol Dweck for agreeing to serve as the chair of my University Oral Exam as well as for her monumental work on the growth mindset. My way of thinking about learning and personal development was heavily influenced by your work. Thanks to David Mazières, Mendel Rosenblum, and Matei Zaharia for making the time to serve on my University Oral Exam committee. Thanks to Matei Zaharia and Peter Bailis for serving on my reading committee, and giving input on my thesis proposal. I am especially grateful to Matei for pointing me to Akaros and Parlib.

Arachne would not exist in its current form without significant contributions from my collaborators. Jacqueline Speiser co-designed and implemented the first version of the Core Arbiter. Qian Li designed and prototyped multiple ways of integrating Arachne and memcached, as well as several important experiments for the OSDI submission. Peter Kraft implemented an initial prototype of the core policies mechanism, fixed bugs in the Arachne runtime, and made YCSB benchmarks run on RAMCloud. I would additionally like to extend special thanks to Collin Lee for participating in countless design discussions in the early days of Arachne, Yilong Li for helping debug RAMCloud

transports on CloudLab during deadlining for the OSDI submission, and Amy Ousterhout for addressing all my questions about Shenango. It was a pleasure working with all of you.

It is difficult to work on any project for three years and still be able to quickly identify its weaknesses. Thanks to the 10 anonymous reviewers and Jon Howell for giving feedback on the two submissions of the Arachne paper. Your feedback was invaluable for improving the paper and helping me fight the curse of knowledge.

Thanks to my undergraduate research advisors Landon Cox and Susan Rodger. Your excellent teaching directly influenced my choice to study computer science at Duke. Doing research with you gave me the knowledge to decide to actually pursue a PhD.

Thanks to the fine folks at Eloquent Labs (especially Gabor Angeli and Keenon Werling) for offering me the opportunity to start making an impact in industry before I finished writing, and then generously endorsing my need to take time off work to finish writing.

When I first moved to the Bay Area from Seattle, I found it difficult to make new friends and settle in. Thanks to the past and present members of the Theory Reading Group for making my early years at Stanford a more pleasant experience. Thanks especially to Remy Xue for keeping reading going in one form or another for all these years.

It is my view that personal health and happiness are incredibly important to sustainably working on long-term projects and pushing them to completion. This was true for the design and implementation of Arachne and it was true for the writing of this dissertation. Thanks to all my classmates, friends and family for making this journey glorious!

On a final note, this dissertation owes much to Rebecca He Zhang, for being a most supportive and patient life partner over the last few years. She not only put up with the many late evenings and weekends I spent writing it, but also listening to dry runs of the defense talk, and served as a rubber ducky whenever I needed someone to bounce ideas off of late at night. I look forward to spending the rest of my life with you.

Preface

Contents

List of tables

List of figures

Chapter 1

Introduction

A key challenge for cloud applications that interact with consumers is low-latency data access. Unlike traditional high performance computing (HPC) applications, today's cloud applications must manipulate many small pieces of data and respond quickly enough to provide interactivity. For example, constructing a Facebook feed requires collecting posts from a nontrivial subset of posts made by one's friends and then filtering them. Similarly, every time a Google search is performed, the service must collect candidate webpages from many sources and rank them by relevance. Such searches often also fetch user data for personalization and ad data for revenue generation. For scalability reasons, these data are spread across multiple machines. Consequently, one external request often triggers many internal requests.

Spawning many internal requests to access data to handle each external request makes it challenging for applications to respond interactively. If one internal request depends on the output of another, applications must issue the two requests serially. In the Facebook example, the news feed application must first fetch all of one's friends before issuing the requests to find their posts. Even if most internal requests can be issued in parallel, latency still suffers from the straggler problem. Suppose a server issues one hundred internal requests and cannot respond to the external client until at least fifty of them have responded. Its response latency is at minimum the response time of the 51th fastest response.

The traditional barriers to low latency data access have been networking and storage, but they have become less of a bottleneck in recent years and are continuing to improve. In the past, it took milliseconds to transmit a small message across a data center network [37] when running a typical data center workload. Similarly, reading from spinning hard disks took several milliseconds. Today's network fabrics can transmit a small message in 2 μs over RDMA [9]. Kernel bypass libraries such

as DPDK [2] remove the overheads caused by kernel buffering and privilege boundary crossing. New network transports such as Homa [37] and TIMELY [35] minimize queuing delays in the network. Together, these developments drive network latencies down towards hardware limits. On the storage front, today's services have almost entirely migrated from hard drives to main memory for storing data requiring low latency access, sidestepping slow disk speeds. Moreover, the recent emergence of non-volatile memory devices is making it possible to access more data with low latency.

As a result of these improvements, it is now possible for applications to service requests in as little as a few microseconds. However, today's kernel-based threading mechanisms cannot support microsecond scale services efficiently. Starting a kernel thread takes several microseconds, making it impractical to create a new thread for each request. Moreover, waking up a blocked kernel thread takes several microseconds, so kernel threads must not block in applications requiring low latency. This means they must busy-wait when waiting for operations taking microseconds. Examples of such operations are remote procedure calls and the taking of contested mutexes. Cores which are busy-waiting cannot perform other useful work; polling reduces useful core utilization.

Another problem in today's threading systems is that applications lack visibility into and control over the use of cores. The kernel completely owns thread management; applications cannot control which of their threads run on which cores. Applications also cannot determine when the kernel will schedule and deschedule threads. This makes reducing latency difficult because a thread handling a request can be descheduled by the kernel at any time.

This dissertation describes and evaluates Arachne, a new system for *core-aware* thread management. Arachne changes the interface between applications and the kernel scheduler from one based on threads to one based on cores. In the traditional model, applications create kernel threads which are multiplexed over cores by the operating system. In the Arachne model, applications request cores from a user-space process, which are then exclusively controlled by the application. Each Arachne application determines how many cores it needs based on its load. An Arachne application always knows the exact set of cores it has been granted. Arachne implements fast threads in user space on top of allocated cores, which do not suffer from the problems plaguing threads implemented by the kernel. Applications using Arachne are able to control the placement of their threads onto cores, enabling them to put threads with different latency requirements on different cores.

Arachne consists of three logical components (depicted in Figure 1.1), which work together to offer a fast threading abstraction on top of allocated cores.

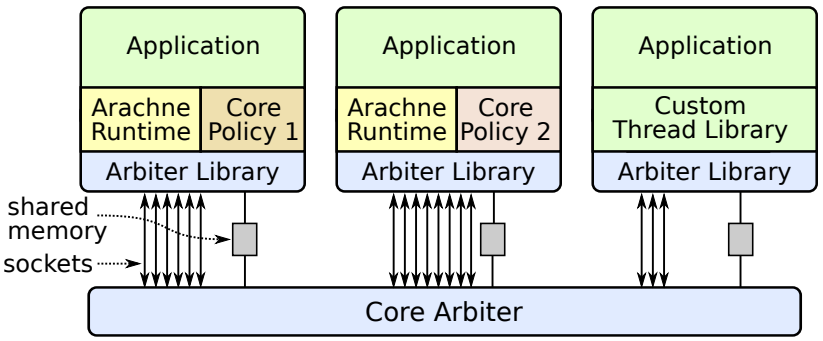


Figure 1.1: The Arachne architecture. The core arbiter communicates with each application using one socket for each kernel thread in the application, plus one page of shared memory.

- A *core arbiter* process allocates cores to applications. It leverages the Linux *cpusets* mechanism and does not require kernel modifications.
- A *runtime library* links into each application and provides a fast user-level thread abstraction.
- A set of *core policies* determine the number of cores needed and define how application threads are mapped onto cores.

An application writer will interact with a different subset of these components depending on their performance needs. For example, applications requiring only a fast user thread abstraction with load estimation can invoke APIs exposed by the Arachne runtime. If a developer wishes to implement her own threads or events on top of the core abstraction, she can link against the core arbiter's client API and request cores. Applications with diverse threading needs and stringent latency requirements may wish to control the scheduling of threads onto cores and perform their own load estimation; these applications will implement new core policies.

User-level thread management systems have been implemented many times in the past [56, 17, 8] and the basic features of Arachne were prototyped in the early 1990s in the form of scheduler activations [4]. Arachne makes the following novel contributions:

- Arachne contains mechanisms to estimate the number of cores needed by an application as it runs. These mechanisms rely on simple metrics collected by the runtime at very low overhead.
- 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.

Arachne enables applications executing short-lived tasks to achieve a better combination of latency and throughput. The Arachne runtime can initiate a new user thread on a different core in 320 ns. 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 remainder of this dissertation motivates Arachne with a more detailed look at the problems with modern threading systems and describes the design and implementation of each of the components of Arachne. It then describes the integration of Arachne into RAMCloud and memcached and evaluates the performance of Arachne.

Chapter 2

Motivation

Arachne was motivated by our experiences building the RAMCloud storage system [42]. In building and benchmarking RAMCloud, my research group discovered the difficulty of building systems that offer low latency while making efficient use of cores with existing tools for thread management. Efficient use of cores consists of two conditions:

- **Not wasting cycles within an application**: The ratio of cycles spent on useful work to cycles spent on overhead is relatively high.
- **Not wasting cycles globally**: When load on a particular application is low, the remainder of the cores are able to do useful work, rather than idling.

Today's tools for thread management have three major problems, which individually and collectively make it difficult to simultaneously achieve low latency and high efficiency when task lifetimes are short.

- Thread primitives are too slow.
- Applications have neither visibility into and nor control over cores.
- It is difficult to isolate applications with dynamically changing workloads to different cores.

RAMCloud suffers from each of the three problems above, and this chapter uses it as an example to illustrate these problems. RAMCloud achieves low latency in spite of slow thread primitives by implementing many of its own thread primitives, but this approach resulted in high overheads, wasting many CPU cycles. RAMCloud avoids interference from other applications by taking over the entire machine. It attempts to avoid self-interference between threads induced by lack of control

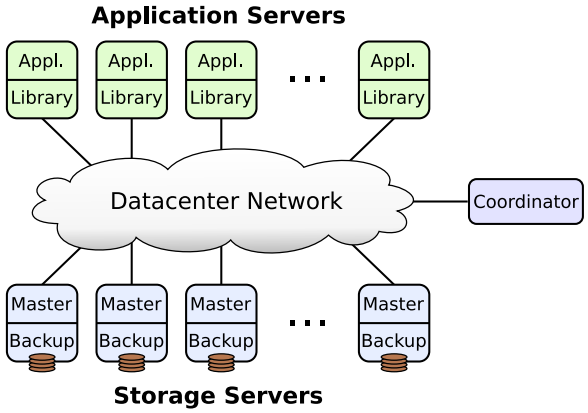


Figure 2.1: The RAMCloud cluster architecture.

over cores by limiting its number of runnable threads. Unfortunately, both of these behaviors can easily waste most of the cores on the system when the system is underloaded. We built Arachne to address the three problems above, and in doing so enable multi-threaded systems to enjoy a better combination of latency and throughput. The following sections introduce RAMCloud, describe its architecture, and elaborate on the problems that motivated Arachne, using RAMCloud as an illustrative example. A final section discusses broader motivations for having a system like Arachne.

2.1 A brief introduction to RAMCloud

2.1.1 RAMCloud overview

RAMCloud [42] is an in-memory storage system that provides low-latency access to large volumes of durably replicated data. To support large capacities, it distributes data across the main memory of thousands of servers. RAMCloud uses a log-structured mechanism [50] to manage DRAM, which results in high performance and efficient memory usage. To support low latency, RAMCloud bypasses the kernel for networking and uses a polling-based approach to communication. Client applications can read small objects from any RAMCloud storage server in less than 5 μs. Durable writes complete in about 13.5 μs.

2.1.2 RAMCloud architecture

RAMCloud is a software package that runs on a collection of commodity servers (see Figure 2.1). A RAMCloud cluster consists of a collection of *storage servers* managed by a single *coordinator*.

Client applications access RAMCloud data over a datacenter network using a thin Rpc library. Each storage server contains two components. A *master* module manages the DRAM of the server to store RAMCloud data, and handles read and write requests from clients. A *backup* module uses local disk or flash memory to store copies of data owned by masters on other servers.

A RAMCloud server performs a variety of client-facing and background tasks during its lifetime. Simple requests, such as reads, are run to completion and returned to the client immediately, because they can be serviced directly from the server's local log. Write requests must be durably replicated to other servers. When a master receives a write Rpc, it writes the data to local memory and then issues replication Rpcs to replicate the data to other servers before responding to the client. Replication Rpcs are an example of *nested Rpcs*, which are Rpcs issued in the context of handling another Rpc. In addition to these client-facing activities, each RAMCloud server must periodically perform other tasks, including the following.

- Log cleaning to reclaim free space from its append-only log.
- Flushing backup segments to secondary storage.
- Writing log messages to a log file or `stdout`.
- Sending probe messages to other servers to detect crashes.

As a result of these periodic tasks and changes in load, the degree of parallelism required by a RAMCloud sever varies over time.

2.1.3 RAMCloud threading model

This section describes how RAMCloud uses threads to implement the functionality described in the previous section; the next section will leverage this threading model to explain the problems with modern thread management tools.

Each RAMCloud server creates a fixed pool of threads at startup. Rpcs are handled by a single *dispatch thread* and a collection of *worker threads* as shown in Figure 2.2. The dispatch thread handles all network communication, including incoming requests and outgoing responses. When a complete Rpc message has been received by the dispatch thread, it selects a worker thread and hands off the request for processing. The worker thread handles the request, generates a response message, and then returns the response to the dispatch thread for transmission.

For example, an incoming read request is received by the dispatch thread and handed off to a worker thread. The worker thread reads the opcode to determine the type of request and looks up

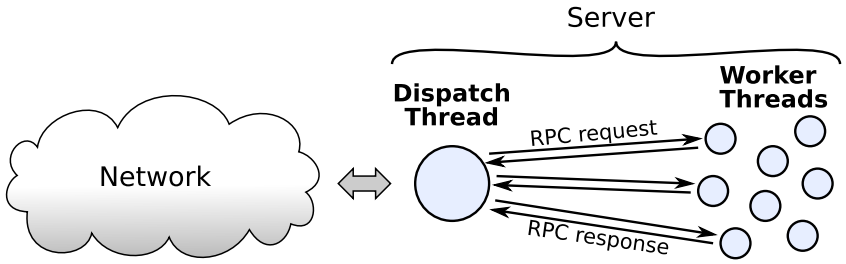


Figure 2.2: The RAMCloud threading architecture. A single dispatch thread handles all network communication; it passes each incoming RPC request to a worker thread for handling. The response message is returned to the dispatch thread for transmission. Each server also contains additional threads for asynchronous tasks such as log cleaning.

the object, copying its contents into a response buffer for the Rpc. It then passes the response buffer to the dispatch thread, which sends the response.

A write request is received and handed off to a worker thread in the same fashion. In the case of a write request, the worker thread writes the data to the local log, sends out replication requests to backups (by passing the requests to the dispatch thread to send), and then must wait until all replication requests return before generating a response.

In addition to the dispatch and worker threads, a RAMCloud server periodically runs a small number of background threads to handle the maintenance tasks described in Section 2.1.2. Whenever the log cleaner runs, it does so in its own thread; this thread sleeps whenever there is no log cleaning to be done. Printing messages to log files is also performed on a separate thread, to avoid paying IO overheads in the threads generating the log messages. Flushing backup segments to disk is also performed on a separate thread to avoid slowing down worker threads which could be used for handling incoming requests.

2.2 Thread primitives are too slow

The first problem with today's thread management is that thread primitives are too slow. Applications using today's slow thread primitives naively (e.g. creating one thread per request) incur high latency and low throughput. In modern Linux, it takes 13 µs to create a thread and 5 µs to wake up a blocked thread. RAMCloud [42] can service a read request in 2 µs and a write request in 10 µs. Memcached [31] can service a read request in 10 µs. Thus, it is infeasible for these systems to

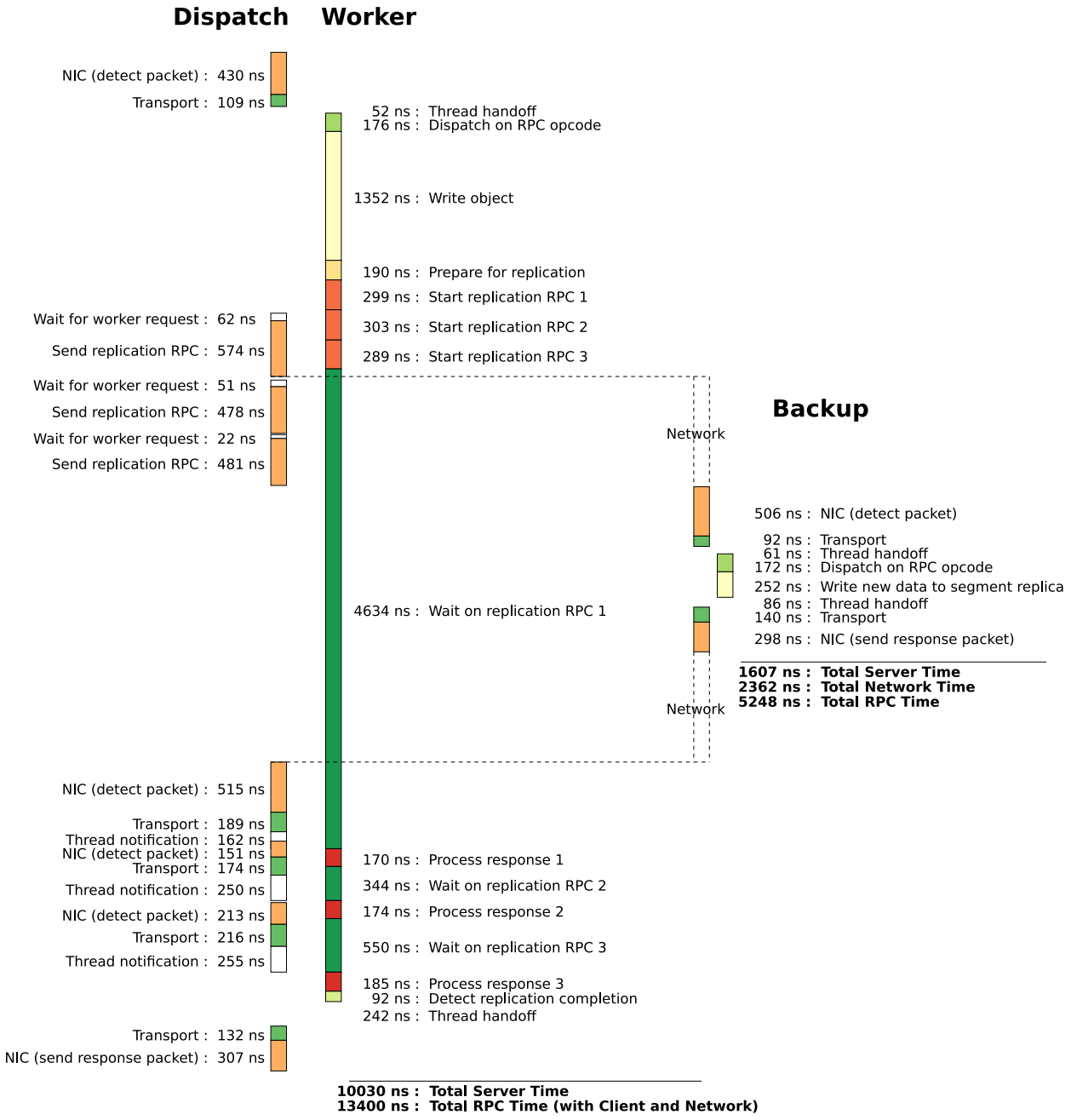


Figure 2.3: Timeline to write a 100B object with 30B key chosen at random from a large table, with a replication factor of three. The figure shows only time on the master and one of the three backups.

create a thread for every request; doing so would at least double latency and cut throughput by half.

Latency-sensitive systems work around slow threading primitives by avoiding their use whenever possible, but workarounds waste CPU cycles, using cores inefficiently. Consider the example of servers issuing nested Rpcs. When waiting for a nested Rpc to return, a server thread has two options. It can either block and allow other threads to execute, or it can busy-wait until the nested Rpcs return. It takes about 5 μs for one thread to sleep and a second thread to start running, so blocking is pointless. By the time the worker thread issuing the nested Rpc had blocked and another thread was running, the Rpc would have already returned. Instead, worker threads send out all the nested Rpcs and poll until they complete. Unfortunately, polling means that the core is unavailable to service other requests until the nested Rpcs have returned. I measured the breakdown of server-side time for a RAMCloud write [42]. The result is shown in Figure 2.3. When handling a write request, a RAMCloud worker thread spends approximately 55% (5.5 μs) of its service time (10 µs) waiting for replication Rpcs to return.

The tradeoff between low latency and efficient core usage exists because threads are too slow. If there existed highly efficient threads that could context switch in nanoseconds, it would be practical for RAMCloud to service other requests while waiting for nested Rpcs to complete. Arachne is such a threading system; integrating it doubles RAMCloud's throughput without substantially increasing latency.

2.3 Applications lack visibility into and control over cores

The second problem with today's threading systems is applications' lack of visibility into cores. Without control over the cores they run on, it is difficult for applications to simultaneously offer low latency and use machines efficiently.

I define an application's *offered parallelism* to be the number of threads an application makes runnable at a given point in time. By runnable, I mean the thread is not sleeping on a mutex, condition variable, or blocking IO. Since the kernel is able to place threads on different cores, this represents the maximum number of parallel tasks that an application can simultaneously run, assuming an infinite number of cores.

I loosely define *number of available cores* to be the number of threads that can simultaneously execute on a given machine. For example, if a machine has four physical cores with two hyperthreads each, I would declare that the machine has eight available cores. Note that the number of cores available to a given application will almost always be less than the number of available cores

on the machine, since the kernel and other applications must execute somewhere.

I define an application's *true parallelism* as the number of threads that are running simultaneously at a given point in time. Suppose a machine has four cores and an application creates three runnable threads. If the three threads are simultaneously executing on three distinct cores, then the application's true parallelism is three. If two of the application's threads have been placed on the same core so that they cannot run simultaneously, then the application's true parallelism is two.

Lack of visibility and control over cores means that applications cannot communicate their true parallelism needs to the operating system kernel. Instead, they can merely attempt to control their offered parallelism by creating threads. Unfortunately, this sort of control forces applications to choose between low latency and high efficiency at best, and grants neither at worst. In the ideal case, using threads should give complete control over offered parallelism; real applications are often not architected around such precise control.

2.3.1 Oversubscription of cores increases latency

A central premise in the discussions below is that oversubscription of cores (which happens when an application or group of applications collectively offers more parallelism than the number of available cores) increases latency for all applications on a system. This premise stems from the belief that oversubscription of cores forces the operating system kernel to multiplex threads.

Why does kernel multiplexing of threads increase latency? When the kernel is tasked with running more threads than available cores, it must decide when to schedule each thread and when to deschedule each thread. Unfortunately, the kernel lacks visibility into application state: it has no idea what the application is doing semantically. Consequently, it cannot pick convenient times to deschedule an application's thread. For example, if a thread is handling a request and gets descheduled immediately before it sends a response, the response will be delayed by the amount of time that the thread was descheduled. Further, if this thread was assigned to handle multiple requests at the time it was descheduled, each of these requests experiences increased latency.

2.3.2 Offered parallelism is not enough

Suppose that an application's architecture enables it to set its offered parallelism to an arbitrary value at any time. For example, an application that computes pure functions as a service might create a thread whenever it wants to increase its offered parallelism and terminate a thread whenever it wants to decrease its offered parallelism. Even such an application must choose between efficiency and

low latency.

The ability to control offered parallelism is not enough because the optimal setting for offered parallelism depends on the number of available cores. The traditional assumption in multi-threaded programming is that there is a single correct number of threads for a given application at a given time: the application creates this many threads and the kernel schedules them. This assumption is broken because there is no performance to be gained from offering more parallelism than the number of available cores. The additional threads will simply be multiplexed by the kernel, increasing latency without increasing throughput. For example, if RAMCloud created 4 worker threads (5 threads including dispatch) on a machine with only 4 cores, requests latency would increase, but throughput would be no better than with 3 worker threads. For best performance, applications need to match their offered parallelism to the number of available cores. It is strictly better (lower latency, equivalent or better throughput) to serialize request processing in a few threads than to create more threads than cores. Before Arachne, applications had no mechanism for determining the number of available cores except to assume ownership of the entire machine.

If an application assumes ownership of the entire machine, it can match application parallelism to available cores and handle the maximum number of requests in parallel by creating one thread for each core. However, this approach is inefficient because most applications do not handle the same load over time. When load is low, entire cores sit idle; this wastes cycles globally. Consequently, an application with tight control over offered parallelism nevertheless cannot simultaneously enjoy low latency and high efficiency.

2.3.3 Real applications lack tight control over offered parallelism

Real applications such as RAMCloud [42] and Memached [31] do not tightly control their offered parallism; application threads wake up whenever there is work to be done without concern about how many other threads are running. This makes it difficult to match offered parallelism to available cores even when the application takes over the whole machine. For example, RAMCloud servers run background threads somewhat sporadically depending on need. Whenever these threads run, the application's offered parallelism increases. Consequently, depending on how many threads RAMCloud is configured to run with, its offered parallelism will either oversubscribe the cores whenever background threads run or undersubscribe cores when the background threads do not run. Therefore, even assuming whole-machine takeover and constant load, RAMCloud must choose between offering high latency some of the time and wasting cores some of the time.

2.4 Lack of isolation between applications

The third problem with today's thread management is the lack of isolation between applications. The operating system considers threads from different applications to be equivalent when making scheduling decisions; it will happily multiplex threads from different applications on the same core. Thus, two applications running on the same machine will compete for cores and interfere with one another's performance. For example, if RAMCloud runs on the same machine as another application like the x264 video encoder [34], RAMCloud's threads can be descheduled by the kernel to run threads of x264. As discussed in Section 2.3.2, threads being descheduled increases latency.

In order to achieve both low latency and high efficiency using today's threading systems, applications must coordinate their use of resources. For example, when load on RAMCloud increases, it must simultaneously increase its offered parallelism and request x264 to decrease its offered parallelism. Otherwise, their combined offered parallelism would exceed the number of cores in the system, forcing the kernel to multiplex their threads and increasing latency. If there are more than two applications, they would have to all be aware of one another and communicate to maintain low latency. It is unreasonable to expect all applications to coordinate with all other applications when deciding how many threads to create. A system in which applications can negotiate over cores rather than simply creating threads would make it easier for multiple applications to coordinate their parallelism.

2.4.1 Thread affinity is not enough

A frequently proposed solution to the problem of thread management is setting thread affinities to pin threads to different cores and thereby avoid kernel multiplexing. Unfortunately, setting affinities only works if an application controls the whole machine. One application pinning its thread to core 0 does not prevent another application from also pinning its thread to core 0. In such a scenario, the threads share the core even if other cores are available. Consequently, applications independently setting affinity can actually hurt performance.

2.5 The limitations of virtualization

The problems described above result from virtualization, in which an operating system uses a set of physical resources to implement a larger and more diverse set of virtual entities. Virtualization has many benefits, but it only works if there is a balance between the use of virtual objects and the

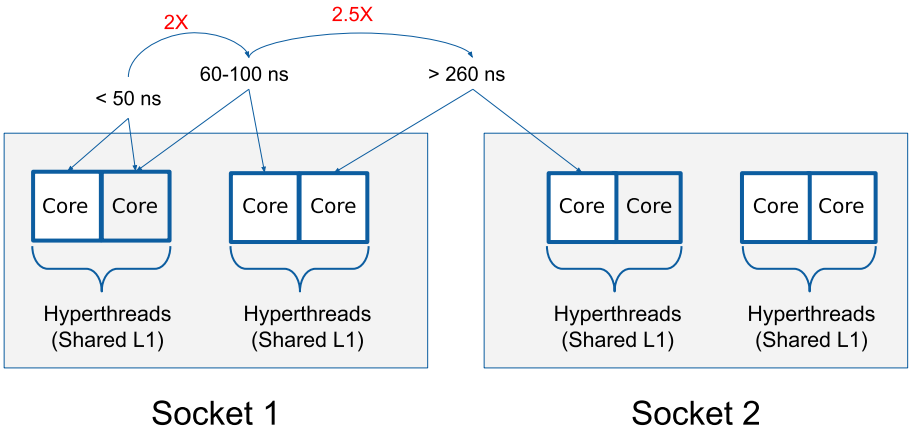


Figure 2.4: Cores are structured as a hierarchy. Communication latencies increases roughly 2x with each additional layer. Latencies in this figure were measured on a dual-socket machine running Intel(R) Xeon(R) CPU E5-2670.

available physical resources. For example, if the usage of virtual memory exceeds available physical memory, the system will collapse under page thrashing. Threads abstract away cores and the challenges of core mangement from applications, but when there are more runnable threads than cores in a system, performance suffers. Applications need visibility into cores to avoid oversubscribing them. Arachne offers applications this kind of visiblity and enables them to run with low latency and high efficiency.

2.6 Problems intensify when cores are not homogenous

The problems induced by virtualization of cores are already severe when all cores are interchangeable, but virtualization exacerbates them on modern architectures because cores are not homogenous. On modern architectures, cores are organized in a hierachy, as illustrated in Figure 2.4. Each physical core typically has multiple (typically two) hyperthreads which run simultaneously and can communicate with one another in tens of nanoseconds. A single CPU socket contains multiple physical cores, and a single machine can contain multiple CPU sockets. Hyperthreads of different cores on the same socket can communicate in approximately one hundred nanoseconds. Hyperthreads of cores on different sockets require hundreds of nanoseconds to communicate. Since the kernel does not expose core topology and core allocations to applications, application threads are subject to

unpredictable communication latencies. Pinning threads based on a known core topology can mitigate some of these problems but this only works if the application takes over the entire machine and is non-portable. Arachne's core arbiter tries to allocate cores to applications in a toplogy-aware fashion, and its core policy framework offers applications the possibility of running across heterogenuous cores in a performant way.

2.7 Summary

This chapter motivated Arachne by describing the difficulty of implementing low latency systems efficiently using today's thread management systems. It pointed out three problems with today's thread abstraction: slow thread primitives, lack of visibility and control into cores, and lack of isolation between applications. Finally, it briefly examined the limitations of virtualization and explained why the performance issues of today's threads are exacerbated by modern architectures.

Chapter 3

From Threads To Cores

Arachne changes the abstraction between applications and the kernel from threads to cores. It is the next stage in a long line of work on thread management. This chapter briefly summarize the costs and benefits of various models for managing threads, culminating in Arachne's offering and how it differs from prior models. It then elaborates on Arachne's model of offering cores, expanding it into a structural overview of the lifecycle of an Arachne application. This structural overview forms a framework on which to hang the remainder of the dissertation.

3.1 User Threads and Kernel Threads

Threads are an abstraction for allowing multiple sequential execution streams to run in the same address space. They can be supported at user level or in an operating system kernel. Anderson et. al [4] argued that each approach has limitations; I describe both types of threads and summarize their arguments below.

Threads implemented by the kernel enable simultaneous execution across multiple physical processors, and are globally work-conserving: cores do not idle when there are threads to run. If one thread is preempted, other threads from some application can be scheduled by the kernel, allowing the application to continue to make progress. Since the kernel implements the page fault handler, it can schedule other threads onto the physical processor if a thread takes a page fault. The story is the same with blocking IO, which the kernel implements inside its system calls.

Unfortunately, kernel threads suffer from the problems described in Chapter 2. As a reminder, they are too heavyweight (13 µs for a thread creation) for use in applications that create short-lived threads. and can be descheduled at any time, causing increased latency for applications. Moreover,

the kernel treats threads of different applications identically, resulting in threads of one application being descheduled to run threads of a different application.

User level threads are typically implemented as a library linked into the application. They use kernel threads as virtual processors. Like a physical processor, the kernel thread executes sequences of instructions defined by the application. Context switching and thread operations (creation, synchronization, blocking, etc) are implemented without going through the operating system kernel. Consequently, user threads can perform thread management operations with an order of magntiude lower latency than kernel threads.

Unfortunately, virtual processors are not equivalent to physical processors. The underlying OS kernel is often tasked with running more threads than there are cores (as a result of applications creating threads without coordination), which means it will preempt and deschedule threads. When the thread serving as a virtual processor is descheduled, user threads assigned to the virtual processor must either be moved to other virtual processors or wait for the thread to be scheduled again before they can run again. Moving user threads to another virtual processor is particularly expensive when virtual processors are descheduled without warning, because other virtual processors must first detect that user threads are piling up on a descheduled virtual processor, since there is no mechanism for notifying them. Both scenarios delay the completion time of user threads. Moreover, the application effectively loses its virtual processor, decrementing the application's offered parallelism.

If a user thread takes a page fault or performs blocking IO, other user threads attached to its processor suffer the same fate as in the descheduling case. That is, if one virtual thread traps to the kernel to do an expensive operation, the other user threads effectively lose their virtual processor. Since user threads do not visibility into kernel state, they cannot always anticipate when page faults will happen or when a system call will block and prepare for it by migrating other user threads attached to the same virtual processor away.

3.2 Definitions

I define *fairness* to mean that each runnable thread is able to run at least once over some user-defined time interval. That is to say, if there are fifty runnable threads and the user specifies a time interval of one hundred miliseconds, then each of those fifty threads will get a chance to run at least once every one hundred milliseconds.

I define *starvation* of a runnable thread to mean that the thread never gets a chance to run. For example, suppose a system has low priority threads and high priority threads. If system policy

dictates that a low-priority thread will never run when there exists a runnable high-priority thread, then a low-priority thread may starve if there are always runnable high-priority threads.

3.3 Threading Models

Over the years, a variety of hybrid threading models have been developed to work around the trade-offs between user threads and kernel threads. The paragraphs below summarize the various threading models briefly.

The most ubiquitous threading model today is the one-to-one model, in which each kernel thread is mapped to a single user thread. This model is semantically equivalent to directly using threads implemented by the kernel, although the standard libraries of several programming languages provide a language-level wrapper around the kernel thread interfaces. This model is simple to reason about and develop against for the application writer because the kernel provides fairness and prevents starvation. The application writer can simply create threads and the kernel ensures that each thread will get a reasonable share of hardware resources and make progress. If a thread runs for a long time, the kernel will preempt it and allow other threads to run. If a thread blocks in the kernel, the kernel will run another thread on the core. The downside of relying on the kernel to schedule threads is that the kernel has no visibility into application behavior, so it cannot differentiate between convenient and inconvenient times to deschedule a thread. For example, a thread might be descheduled immediately before returning a response to a client, increasing the latency observed by that client.

A threading model that was popular when single-core processors were widespread, but no longer makes sense in today's multicore architectures, is the M-to-1 threading model [56, 1]. In this model, an application process runs a single kernel thread and implements many user level threads on it. Such a model allows for efficient context switches between threads because the process need not switch from user mode to kernel mode and back again. Additionally, the application has more control over when each thread runs. Using the example from above, the application can allow a user thread to completely finish handling a request before switching context, because the information about the thread's request-handling state exists in the application's own memory and is represented in a format the application understands. This model offers the application more control over thread management, but also creates more issues for the application to deal with; fairness and starvation are entirely up to the application. For example, one application thread can induce starvation for other application threads by never terminating and never relinquishing control. Additionally, the M-to-1 model suffers from the limitations of user threads described in the Section 3.1. It also cannot take

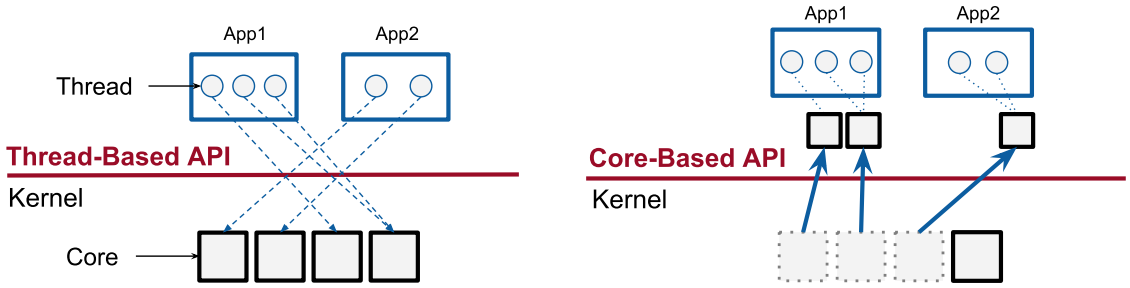


Figure 3.1: An illustration of a thread-based API vs a core-based API.

advantage of multiple physical cores to run threads in parallel.

A threading model that is gaining traction as of this writing is the M-to-N threading model [8, 17]. Under this model, an application spawns multiple kernel threads and multiplexes user threads over them. In typical implementations, the application is developed against the abstraction of user threads; a language runtime or library will determine which user threads run on which kernel threads. With an efficient implementation, applications enjoy fast context switching while being able to take advantage of multiple physical cores. Unfortunately, this model also suffers from the disadvantages of both user threads and kernel threads. If a particular kernel thread is descheduled, then it becomes unuseable for running user threads, decreasing true parallelism. The problem of kernel threads being descheduled at inconvenient times is further exabercated by M-to-N threading. If a user thread acquires a highly contended mutex and gets preempted, other user threads needing this mutex will be unable to run even if there are available cores.

3.4 Getting the best of both worlds: M user threads to N cores

Most of the problems with prior threading models stem from kernel threads not being equivalent to cores; they can be descheduled without warning by the kernel and creating too many or too few results in suboptimal performance as described in Chapter 2. Applications lack visibility into and control over how physical cores are allocated by the kernel, so they must (inefficiently) take over an entire machine to achieve low latency.

Arachne introduces the notion of cores as an allocatable entity, as illustrated in Figure 3.1. Rather than requesting kernel threads from the operating system kernel, applications request cores. Unlike kernel threads, Arachne's cores correlate one-to-one with hardware cores. Once a core is

allocated to an application, it belongs to the application and will not be preempted from the application without warning.

Based on this new scheduling entity, Arachne introduces a new hybrid threading model: user threads multiplexed over cores. This threading model offers all the benefits of the M-to-N threading models but eliminates the problems associated with kernel threads. Since Arachne threads are implemented in user space, the overheads for thread primitives are low and threads can feasibly be created for short-lived tasks. Applications integrating Arachne run on multiple physical cores, and enjoy fast user threading without suffering from being descheduled at inconvenient times. Since complete cores are allocated to applications, their threads are isolated from the threads of other applications. Under Arachne, applications always have visibility into cores they own; the kernel always asks before preempting cores, giving the application time to either finishing executing or migrate existing user threads.

3.5 Arachne Application Lifecycle

What does life look like for an application integrating Arachne? Here's a high-level summary of the lifecycle for an Arachne application; later chapters dive into the details of each component.

- *Initialization*. When an Arachne application first starts, it is unknown to the Arachne system. The application requests some initial number of cores N from the core arbiter. If cores are available, the core arbiter grants N cores to the application. Otherwise, the core arbiter either takes cores away from other applications or gives this application fewer than N cores, depending on the priority of the application's request relative to the requests of other applications. At this point, only the Arachne application can execute threads on these cores.

- *Steady State*. After initialization, the application enters normal execution, which is characterized by the following recurring activities.

    1. The application creates Arachne threads using the Arachne runtime's API and the Arachne runtime asks the core policy to chose cores for them. For example, the core policy might arrange for a polling-based dispatch thread to have exclusive access to a single core, while worker threads share a pool of cores.
    2. The Arachne runtime collects per-core statistics, such as utilization and number of runnable threads.

3. The active core policy uses these statistics to estimate the ideal number of cores for the current load. If more or fewer cores are needed, the core policy makes a request to the core arbiter.

- *Gaining Cores*. Both at initialization and any time the arbiter grants a core to an application, the newly granted core registers itself with the core policy; the core policy will use this core for future thread creations.

- *Losing Cores*. If the core arbiter needs to reclaim a core, for example to give to a higher priority application, then it does so in a cooperative fashion, by making a request to the application and then waiting some time for the application to respond. The Arachne runtime code running on the core detects the request, runs to a clean stopping point and de-registers the core with the core policy before it returns the core to the arbiter.

3.6 Summary

This chapter introduced various schemes for implementing threads and describes their benefits and limitations. It then offered Arachne's M user threads to N cores model as a remedy to the limitations of prior models. Finally, it elucidated this model by presenting a high-level overview of the lifecycle of an Arachne application.

Chapter 4

Core Arbiter

Chapter 2 highlighted the difficulty of choosing the right number of kernel threads for an application. When multiple applications independently create kernel threads to offer parallelism, the result is descheduling of threads without notice and high latencies. Multicore systems need centralized coordination of cores to be performant and efficient. Arachne's core arbiter offers this means of centralized coordination.

This chapter 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 allows Arachne applications to coexist with existing applications that do not 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.

4.1 Classes of Cores

The core arbiter divides cores into two groups: *managed cores* and *unmanaged cores*. Managed cores are allocated to Arachne applications 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.

4.2 Allocating cores using Linux cpusets

Arachne uses kernel threads to control cores. A kernel thread that is the only runnable kernel thread on a given core owns the core. On initialization, Arachne applications create a collection of blocked kernel threads. The core arbiter allocates a core to an application by unblocking one of the these kernel threads and ensuring that it is the only runnable kernel thread on the core being allocated.

The core arbiter is implemented without kernel modifications. It runs as a user process with root privilege 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.

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 core arbiter moves cores back into the unmanaged cpuset if they have not been allocated to an application for a few milliseconds. In steady state, each core is either in the unmanaged cpuset or allocated to an Arachne application.

4.3 Interaction with the 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 (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 1.1). 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 the thread's 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. The initial thread that invoked `setRequestedCores` enters a loop where it joins each of the kernel threads it spawned one at a time; when the application terminates these kernel threads will exit and the initial thread will unblock and exit. 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.

4.3.1 Cooperatively reclaiming cores from applications

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. If the core arbiter wishes to reclaim a core from an application, it asks the application to release the core. It does so by setting 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 5.5 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.

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.

4.3.2 The benefits of asymetric communication

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.

4.4 Core allocation

4.4.1 Core request structure

For the purposes of core allocation, the core arbiter treats a machine thread (also known as a hyperthread) as a unit of allocation. I use the term *core* to refer to a hyperthread. This work uses the term *hypertwin* with respect to a given core *A* to refer to the core *B* that *A* shares a physical core with.

The core arbiter uses a priority-with-constraints mechanism for allocating cores to applications. Arachne applications can request cores on different priority levels (the current implementation supports eight) and specify constraints on the set of cores they are granted. A core request consists of a vector of integers and two boolean flags, described below.

- `vector<int> coresRequestedByPriority`: Each integer corresponds to the number of cores requested at the priority level of its index in the vector.
- `bool noSharedCores`: A value of 1 means that the application is unwilling to run on a hypertwin of a core that is running a thread belonging to a different application.
- `bool singleNUMAOnly`: A value of 1 means that the application is unwilling to run on cores that span multiple NUMA nodes.

4.4.2 Allocation algorithm

The core allocation algorithm grants cores to applications from highest priority to lowest, subject to hypertwin and socket constraints. 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 approximately divides the cores evenly among the requesting applications. In additional to respecting the constraints above, the arbiter allocates all of the hyperthreads of a particular hardware core to the same application whenever possible, and attempts to keep all of an application's cores on the same socket. These core placement heuristics strongly affect application performance because core topology affects both communication latency and shared cache behavior. Cores on different sockets pay at least a 2.5x performance penalty to communicate compared to cores on the same socket. Hyperthreads of the

same physical core share an L1 and L2 cache; threads from the same application running on two hyperthreads of the same physical core can help fill each other's cache, while threads from different applications will compete for cache space.

The algorithm determines which applications get which cores; this is particularly complicated when there are more cores requested than cores available.

To help illustrate the algorithm, consider a situation with three Arachne applications on a system that made the following core requests.

- Application *A*: coresRequestedByPriority: [5,0,0,0,0,0,0,0], noSharedCores: true, singleNUMAOnly: true
- Application *B*: coresRequestedByPriority: [3,0,0,0,0,0,0,0], noSharedCores: false, singleNUMAOnly: true
- Application *C*: coresRequestedByPriority: [0,1,0,0,0,0,0,0], noSharedCores: false, singleNUMAOnly: true

Assume that these applications are running on a single socket machine with four physical cores that have two hyperthreads each. Assume further that the hyperthreads are numbered 0 to 7, and consecutive even-odd pairs are hypertwins of one another. That is, Core 0 is the hypertwin of Core 1.

The algorithm that the core arbiter runs to allocate cores among applications is described below. The primary challenge in designing this algorithm is balancing application-specified constraints, performance considerations of machine topology, and core request priorities in a way that balances performance and fairness; the algorithm additionally aims to be work-conserving. Conceptually, the algorithm allocates cores to applications in request priority order and reallocates cores from higher-priority applications to lower-priority applications until all constraints are satisfied. This, a higher-priority application with more strict constraints may sometimes yield cores to a lower-priority application with less strict constraints, in the case where the former's constraints cannot be satisfied. To do this, the algorithm computes an ideal ordering for granting cores to applications if all cores were constructed equal and optimistically grants cores in this order, revoking the most recently granted core (and granting it to the next process in the order) whenever constraints are violated.

1. **Construct global priority list**. Compute a priority-ordered list of candidate core grants assuming a machine with infinite cores; each entry in the list represents the grant of a core to

an application. This list is constructed by iterating over the priority levels from highest (0) to lowest (7) and adding grants for applications in a round robin fashion. If an application requested an odd number of cores and set `noSharedCores`, increment its requested core count temporarily before building this list. In our example, the list is A,B,A,B,A,B,A,A,A,C, where the name of the application represents a candidate grant to that application.

2. **Create initial core assignment**. Iterate over this list and assign (generic, not specific) cores to applications up to the limit of the total cores available in the system. In our example, the assignments might look like $A:5,B:3,C:0$. Note that the ordering of these assignments is not significant here.

3. **Adjust for hypertwin constraints**. If any application with `noSharedCores` set is assigned an odd number, decrement its core grant and offer the core to the next application on the list. In our example, this happens twice. First, the grant to A at the 8th position in the global priority list is revoked, and a core is granted to A because it appears at the 9th position. Then, the grant to A at the 9th position is revoked, and a core is granted to C. The subsequent assignments are $A:4,B:3,C:1$.

4. **Assign applications to sockets**. Assign applications with `singleNUMAOnly` set to sockets in descending order of cores granted. In our example, that means the assignment order is ABC. If an application does not fit in any socket, unassign the last assigned core, make a core grant to the next process on the global priority list, and go back to Step 3.

5. **Assign specific cores to applications**. Iterate over the applications and assign specific cores from the assigned socket to each application. This assignment prioritizes granting cores to an application that were previously granted to it, and attempts to minimize sharing of hypertwins among applications. In our example, one possible assignment is $A:[0,1,2,3],B:[4,5,6],C:[7]$.

The core arbiter recomputes the core allocation whenever application requests change.

4.5 Central Event Loop

The core arbiter server is implemented as a single-threaded, non-terminating `epoll` loop which wakes up to handle any of the following events.

- **New connection**. A new Arachne application thread is informing the arbiter of its existence. The arbiter sets up state to track the thread.
- **Core request**. An application has invoked `setRequestedCores`. The arbiter recomputes core allocations and either unblocks threads or requests preemptions as appropriate to satisfy the newly computed allocation.
- **Thread blocking**. A thread has invoked `blockUntilCoreAvailable` and will be going to sleep. The core arbiter updates its metadata about the thread and eventually removes it from the core. If the thread blocked in response to a preemption, the core arbiter might unblock a thread from a higher priority application.
- **Preemption timer expired**. When the arbiter requests a core back from an application as a result of core reallocation, it sets a timer. When this timer expires, the arbiter is awoken to check whether the application has complied with the request.
- **Closed connection**. When an Arachne application exits, its threads will close their socket connections, triggering the arbiter to wake up and clean up state for the thread and process.

4.6 Limitations

The core arbiter offers the useability benefit of requiring no kernel modifications, but this benefit comes at a cost of being unable to fully remove interference on cores. The cpuset mechanism allows the arbiter to move threads belonging to other user processes to unmanaged cores, but it cannot move threads belonging to the kernel itself. Chapter 7's results show that this limitation does not appear to impact our experimental results, but it is possible that more kernel-intensive background workloads will cause a performance impact.

4.7 Summary

This chapter described how the core arbiter implements centralized control of cores using the Linux cpuset mechanism. The core arbiter enables Arachne applications to request cores rather than threads while coexisting with traditional Linux applications.

Chapter 5

The Arachne Runtime

This chapter 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. This chapter describes general design considerations, explains why performance in Arachne's world is all about cache misses, and presents Arachne's runtime design, which minimizes cache misses with one unified mechanism for thread creation, thread blocking, thread sleeping, and thread migration. It wraps by discussing other functions of the runtime, such as metrics collection and (de-)initialization of cores.

5.1 Design considerations

The Arachne runtime is designed to make efficient use of cores even if tasks have lifetimes of only a few microseconds. 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 focus on short tasks necessitates that the overheads of thread primitives must be as low as possible, lest they become a significant fraction of task execution time, as discussed in Chapter 2.

An additional and non-obvious consequence of designing for short-lived tasks is that load balancing of threads among cores must be done at thread creation time. The common approach to load-balancing today is based on work stealing [17, 22, 12]: child threads are placed on the creator's core, and cores steal from one another when they run out of work. Work-stealing has several benefits. It is work-conserving, in that idle cores will find threads to execute if there are threads waiting to run. It avoids cross-core communication as long as each core remains busy. When stealing, a core can steal multiple threads, amortizing the cost of cross-core communication for each thread.

However, relying on work-stealing to load-balance does not make sense for short-lived tasks for several reasons. First, work-stealing adds latency to thread execution, because threads sit in the creator's core while waiting to be stolen. Executing the child thread with low latency on the creator's core implies that the creator either yielded or exited very soon after creating the thread. This cannot be the common case, because such a creator could simply invoke the child thread's top level function rather than creating a new thread; the creation of the new thread represents a desire to execute code in parallel. Second, a thread that has started running on one core and then migrates to another core will often need to move more state (such as stack memory) compared to a thread that was created on the second core to begin with. Moving additional state incurs cache misses, which increases running time by a roughly one hundred nanoseconds for each cache miss. This increase represents a larger fraction of the running time of short-lived tasks than long-lived tasks.

Thus, placing a short-lived child thread on the creator's core by default is not useful, and such placement should only occur if all other cores are more heavily loaded than the creator's core. Instead, Arachne performs load-balancing at thread creation time, with the goal of getting a new thread running on an unloaded core as quickly as possible.

5.2 Minimizing cache traffic for high performance

Cache coherence dominates the performance of threading primitives. Since we expect threads to run on different cores in the common case, cross-core communication is necessary for thread creation and other thread operations. Operations such as creating a new thread or waking up a sleeping thread do not require many instructions to complete (14 instructions for a context switch on x86), but they require data to be transferred from one core to another. Moving data results in cache invalidations and cache misses, and a single cache miss costs a hundred or more cycles, so the cost of cache operations easily overwhelms the cost of instructions.

Arachne achieves high performance by eliminating cache misses where possible, and overlapping the remaining cache misses to the degree possible. As an example, consider the problem of designing a load-balanced thread creation mechanism. Here is the mininum set of information that must be transferred between cores.

- **Load statistics**: A thread runtime must choose a core for the new thread in a way that balances load across available cores. Load state is shared by definition because it must be updated by the target core and read by the creator's core. Moreover, to load balance effectively in a system running short-lived threads, the thread creator must collect the most up-to-date load;

this virtually ensures a cache miss.

- **Race avoidance synchronization**: The thread performing the creation must synchronize with other creators to ensure that management data structures are not written by more than one creator at a time.
- **Execution state**: The address and arguments for the thread's top-level function 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.

Arachne's thread creation attempts to take no cache misses beyond what is necessary to transfer this set of data.

5.2.1 Overlapping cache misses and hiding expensive computation

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

5.3 A cache-optimized thread management mechanism

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:

Arachne's Key Data Structures	
`ThreadContext`	A data structure associated with each thread; includes thread stack, scheduling state, saved execution state, and pointer to top-level function.
`MaskAndCount`	A data structure associated with each core; it consists of a 64-bit value divided into a 56-bit mask and an 8-bit count of set bits in the mask. Each bit in the mask corresponds to an element in an associated array of `ThreadContext`'s; a set bit indicates that the corresponding context holds a thread. The mask field is named `occupied` and the count field is named `numOccupied`.

Table 5.1: Each core has an associated variable of type `MaskAndCount` and array of `ThreadContext`'s. The `MaskAndCount` variable is named `maskAndCount` in the text and the code.

a thread must be added to the ready queue for its core when it is awakened, and 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 can be 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 associating a ready queue with each core, Arachne binds an array of `ThreadContext`'s to each core (see Table 5.1). 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 (discussed in Section 5.3.3). A thread context remains bound to its core after its thread completes. Arachne reuses recently-used contexts when creating new threads to take advantage of these contexts already being cached.

In lieu 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, I expect only a few thread contexts to be occupied for a core at a given time (many occupied contexts imply that the application is not keeping up with its load). 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

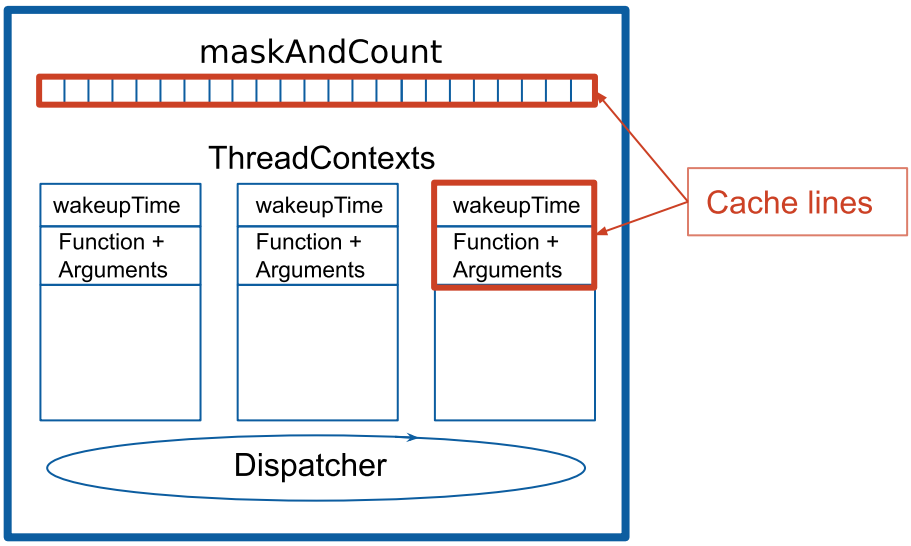


Figure 5.1: Data structures used for queueless thread scheduling. Wakeup time and top level function and arguments for a thread live in one cache line, while the `maskAndCount` variable for each core lives in its own cache line.

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. The cost of this approach is evaluated in in Chapter 7.

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. This variable stores time because it generalizes to handling thread sleep, as discussed in Section 5.3.2. 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.

5.3.1 Thread creation

To help with thread creation, the Arachne runtime associates a `maskAndCount` variable with each core (see Table 5.1). This variable serves two purposes: it is used to track load for load balancing, and to provide mutual exclusion on each thread context associated with the core.

When creating new threads, Arachne uses the "power of two choices" approach for load balancing [36]. 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 approach should work well on systems with many cores: multiple thread creations can occur simultaneously, and the creators will pick different cores with high probability, so there will be few collisions.

Once a thread context has been allocated, Arachne copies the address and arguments for the thread's top-level function into the context and schedules the thread for execution by setting its `wakeupTime` to 0. In order to minimize cache misses, Arachne places `wakeupTime` in the same cache line as the function and arguments. Thus, this costs two cache miss times for reading and invalidation on the creator's core, and then one additional cache miss for the dispatcher on the target core to read `wakeupTime`. Figure 5.1 shows the cache lines boundaries around the data structures.

With this mechanism, a new thread can be invoked on a different core in five cache miss times. Two cache misses times are required to read and perform a compare-and-swap on the target core's `maskAndCount`, and three cache miss times are required to transfer the line containing the function address and arguments and the scheduling flag.

Thread creation failure

Thread creations in Arachne can fail and this ability to fail imposes a cost on application developers: everywhere in their code where they create threads, they must check for and reason about the correct behavior in the event of thread creation failure.

A substantial benefit of thread creation failure is automatic load throttling. When an application integrating Arachne is offered more load than the sytem's cores can keep up with, the 56 contexts on each core fill up with threads, and attempts to create threads will fail until cores clear up the backlog of work. By failing thread creations at this point, Arachne effectively signals to the application that the system is in overload and that creating more threads will simply increase the size of the backlog. An application can use this signal to exert backpressure on its clients to decrease load.

Thread creation failure also simplifies edge cases related to core management. A core that is about to be returned to the core arbiter can reject attempted thread creations by failing them. If

thread creations were defined to not fail, they would need to implement retry logic with back-off to handle cores going offline.

Consequently, application code that creates threads must always check for and handle thread creation failures. This imposes a complexity cost of application developers, but I believe the cost is outweighted by the benefits described above, and applications are quite capable handle these failures. When thread creations fail, applications have multiple options for handling the failure. One option is to repeatedly retry the thread creation until it succeeds; this works reasonably well when overload is not expected and failures are likely the result of core management operations. A second option is to schedule a retry after an exponential backoff. If the application is a service, a third option is to inform the client that triggered the thread creation that the server is busy; the client can then make the request again later. Additionally, the thread creator could choose to run the top level function of the thread itself.

5.3.2 Thread Sleep and Wakeup

Arachne's representation of scheduling state using `wakeupTime` and an array of `threadContexts` enables an simple, efficient implementation of the synchronization primitives `sleep` and `signal`. A thread can block itself by invoking `sleep`. If an optional time argument is passed, then `wakeupTime` is set to the current time plus the time argument. Otherwise, `wakeupTime` remains the maximum representable value. No other memory is accessed, so `sleep` costs either zero or one cache miss time. One thread can wake up another by calling `signal` with the target thread's context. This function simply sets `wakeupTime` to 0, incurring two cache miss times: one to invalidate for writing, and another for the target core's dispatcher to read. It is worth noting that no additional data structures are used for keeping track of sleeping threads, reducing the surface area for potential cache misses.

5.3.3 Thread Migration for Core Cleanup

When an Arachne application is underloaded, it scales down its number of cores, as described in Section 5.5. To facilitate this scale-down, the runtime reuses the same thread management mechanism for emptying a core of its threads. To satisfy Arachne's performance goals, it is important for migration to be as efficient as possible with respect to minimizing cache misses. The key insight for efficiently migrating a thread from core X to core Y is to exchange an occupied `ThreadContext` on core X with an empty `ThreadContext` on core Y. Using this technique, migration can happen

while preserving the number of thread contexts associated with each core, and there is no need to re-balance thread contexts among cores. To satisfy correctness, migration must eventually terminate, so the migration procedure must also prevent new thread creations from re-filling the core as threads are being migrated. This is done by making the core appear to be filled with threads. The full migration function, described below, is executed inside a user thread on the target core.

1. Block all future creations by setting `numOccupied` to 56.
2. Wait out pending creations that began before creations were blocked. A creation is pending on a `ThreadContext` if its `wakeupTime` is set to the special sentinel value `UNOCCUPIED`, but the corresponding bit in `maskAndCount` is set. Scan over the array until this combination of set bit and `wakeupTime` no longer exists.
3. For each user thread on the core, perform the following procedures.
   (a) Use the power of two choices to select a target core. This costs one cache miss time to read the `maskAndCount`s.
   (b) Reserve a slot by setting an occupied bit, just as in thread creation. This costs one cache miss time to write the `maskAndCount`.
   (c) Swap `ThreadContext` pointers with the target thread, so that the thread being migrated is now bound to the target core, and an empty `ThreadContext` from the target core is bound to the current core. This costs 2 cache miss times to read and then write the `ThreadContext` pointer.

When the target core discovers the migrated thread, it will incur one cache miss to read the new `ThreadContext` pointer (this miss is only occurs during migrations), and one cache miss to read the `wakeupTime`, function and arguments. Thus, the above mechanism costs 6 cache miss times for each migrated thread to start running. If the target core was busy running another thread at the time the migration started, it only pays the cost of the last two cache misses to start the migrated thread.

Why thread migration occurs inside a user thread context

Thread migration must occur inside a user thread context on the source core because the runtime code that detects a core preemption request from the arbiter is executing on a user stack borrowed from a user thread context. (Arachne's runtime is designed this way to avoid a context switch during

thread creations, making them faster.) If the thread context that owns the current stack is occupied, the migration code cannot migrate it, because it should not migrate the stack it is executing on to another core; such migration could result in two cores executing on the same stack simultaneously, which leads to stack memory corruption. Not being able to migrate the thread occupying the current active context is a problem because there is no guarantee that the thread will terminate after it regains control of the core, so the core may never be completely cleared. If the thread context that owns the current stack is unoccupied, then a thread creation currently in progress can create a thread on it before the algorithm starts running, creating the same problem. Performing migration inside a separate thread ensures that the one thread which cannot be migrated (the one that is performing the migration) will naturally terminate at the completion of migration, resulting in a core that is clear of threads.

5.4 Collecting metrics

As the Arachne runtime performs threading operations on behalf of applications, it gathers computationally cheap runtime metrics on each core. These metrics are useful for benchmarking, debugging and optimization. They are recorded as monotonically increasing counters, and consequently add very little overhead to the critical path of the Arachne runtime. The full list of metrics is given below.

- *Idle time*: The number of cycles spent in the Arachne dispatcher looks for threads to run, rather than executing application code.
- *Total time core owned*: The number of cycles the runtime has owned the current core, since the last time this application acquired it.
- *Threads created*: The number of threads this core has created. The number of threads created onto each core is possibly more useful, but it is expensive because it adds an extra cache miss to each thread creation.
- *Threads completed*: The number of threads that have exited on this core. By summing this number and the number of threads created across all cores, it is easy to tell whether threads were lost due to bugs in the runtime.
- *Contended creations*: The number of times the compare-and-swap operation failed at least once during a thread creation. This can occur in any of the following scenarios.
  - Two or more thread creators simultaneously attempt to create a thread on the same core.

  - A thread creation targets a core at the same time a thread is exiting on the same core.
  - One or more thread creators or thread migrators simultaneously target a core.

- *Threads run in the current dispatcher pass*: The number of threads the dispatcher found to be runnable in the most recent scan through all the contexts on the core. This is used for core estimation to estimate the number of runnable threads.
- *Weighted loaded cycles*: The number of threads run in the current dispatcher cycle multipled by the number of CPU cycles it took to finish the pass.
- *Core increments*: The number of times the runtime has gained a core from the arbiter.
- *Core decrements*: The number of times the runtime has returned a core to the arbiter.

The metrics are stored in a simple data structure called `PerfStats`, which contains one value for each metric. Each core maintains its own instance of PerfStats, and they are updated only by that core, so updates incur no cache misses. Arachne's runtime provides functions for aggregating the `PerfStats` across arbitrary subsets of cores. These functions require cache misses to fetch the values, but aggregation occurs relatively infrequently.

5.5 Gaining and losing cores

When the Arachne runtime first obtains a core from the arbiter, it owns the core but the core is not yet ready for use by the runtime. The runtime is responsible for initializing cores with core-local state when they are granted by the core arbiter and de-initializing cores before returning them to the core arbiter. Here are the procedures that the runtime uses to initialize and deinitialize cores. As discussed in Chapter 4, Arachne uses a kernel thread running exclusively on a core to control it; the steps below are executed by this kernel thread.

Initializing Cores

1. Store pointers to the `maskAndCount` and `ThreadContext` array bound to the core that the kernel thread woke up on in thread local variables. This optimization is valuable because the runtime's dispatcher constantly accesses these data structures between periods when user threads are running.
2. Inform the Core Policy that the new core is available.

3. Finally, switch execution context to the first user context and enter the Arachne dispatcher. By executing on the first user context, the Arachne dispatcher can execute the first thread created on this core without having to swap context.

Deinitializing Cores When the Arachne dispatcher on a core is notified that the core arbiter wants the core back, it takes the following steps to prepare a core for return to the core arbiter.

1. Inform the core policy that the core is no longer available.

2. Create a new user thread (call this the *cleaner* thread) on the core, which will run the algorithm described in Section 5.3.3 to clear all threads off the core, and then exit. It is important that the core cleaning algorithm runs in a dedicated user thread, because this ensures that the core can be completely cleared, for reasons described in Section 5.3.3.

3. Immediately before the cleaner thread exits, it sets a thread-local flag to inform the dispatcher that the core is ready to be returned to the core arbiter.

4. When the dispatcher sees this flag, it detaches the thread local variable from the `PerfStats` for the core, and returns control of the core to the core arbiter.

5.6 Limitations

The `maskAndCount` variable limits Arachne to 56 threads per core at a given time. As a result, programming models that rely on large numbers of blocked threads are unsuitable for use with Arachne. Since Arachne assumes threads are created for physical parallelism rather than logical concurrency, there is very little performance gain from creating the 57th thread on a core before the first one finishes executing.

Additionally, Arachne applications are likely to have fewer blocked threads. Rather than a long-lived thread that periodically wakes up to do work, an Arachne application can create a new thread each time work needs to be done; a single long-lived thread can wake up at fixed intervals to spawn such threads. Arachne's primitives are fast enough to make this cheap, and this approach also allows the new threads to be load balanced rather than being stuck on a possibly overloaded core if they were long-lived.

The colocation of `wakeupTime` with the function and arguments 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.

5.7 Summary

This chapter discussed the importance of using threads for parallel execution and its implications for system design. It then explained the importance of minimizing cache traffic, and explained why Arachne's runtime takes fewer cache misses than approaches based on ready queues. Finally, it described how the Arachne runtime achieves a fast and scalable implementation of user threads using a cache-conscious design.

Chapter 6

Core Policies

One of Arachne's goals is to give applications precise control over their usage of cores. Core policies are Arachne's mechanism for applications to precisely control how many cores to request from the core arbiter and how threads are scheduled onto cores. This chapter motivates core policies, discusses how they enable applications to address the thread placement and core estimation problems, describes the interactions between the runtime and core policies, and discusses Arachne's default core policy.

6.1 Introduction

Arachne's core arbiter and runtime solve the three problems associated with thread management described in Chapter 2 by allocating dedicated cores to applications and implementing fast threading primitives. However, control over cores introduces new problems for applications. They must decide which threads to place on which cores, and how many cores they need. In the pre-Arachne world, these decisions were made (poorly) by the operating system kernel. In Arachne's new core-aware world, they represent opportunities for applications to reduce latency and improve efficiency. I believe logic that runs in the application's address space and observes the application's scheduling behavior has a better chance of making these decisions efficiently. A core policy encapsulates logic for making these decisions in a reusable way.

Since applications have diverse threading needs, a single core policy is unlikely to address the needs of all applications. For example, a latency-sensitive application like RAMCloud [42] may utilize a long-lived dispatch thread to communicate over the network and dispatch requests. It would be best if such a thread did not share its core with request-executing threads, since it is often a

bottleneck in dispatch-based thread structures. As another example, many applications [31, 42] have maintenance threads that run periodically but execute for a long time. An application can ensure that request-handling threads are not blocked by maintenance threads by placing all maintenance threads on a single core, and not placing request-handling threads on the same core as a maintenance thread. As a third example, a service application may serve data to and perform computation on behalf of both user-facing and batch consumers. A core policy can run the threads executing requests on behalf of different types of consumers on different cores to prevent batch requests from increasing latency for user-facing requests.

Core policies can allow applications to more effectively handle diverse hardware topologies. If an application cannot fit on a single NUMA node, a core policy can be used to shard request-handling threads across NUMA nodes based on the data they access. That is, it could place threads that need to access different shards of data on different NUMA nodes to minimize communication overheads.

Additionally, applications may have application-specific metrics for determining how intensively to use cores. One application may have a 100 μs SLO at the 99th percentile and be willing to tolerate wasting 60% of its cores to achieve that SLO. Another application may be willing to tolerate 1 ms response times at the 50th percentile but wants to run at 80% core utilization or higher. A third application may care only about how quickly it can service read requests, and only increase its core demands when there are insufficient cores for handling read traffic.

Lacking the experience to anticipate the needs of future applications, I hypothesize that a single core policy cannot address the needs of all applications. Consequently, Arachne offers a framework that provides interfaces and support functions for writing new core policies. Writing high-performance core policies is likely to be challenging, particularly for policies that deal with NUMA issues and hyperthreads. I hope that a small collection of reusable policies can be developed to meet the needs of most applications, so that it will rarely be necessary for an application developer to implement a custom core policy. Applications with unusual threading needs may still need to write their own core policies.

6.1.1 What is a core policy?

A core policy is a pluggable software module that interfaces with Arachne's runtime. It consists of two components that help applications address the two questions arising from application ownership of cores:

- A *thread placement* component determines which core each thread runs on.
- A *core estimation* component determines how many cores an application needs and makes requests to the core arbiter whenever this number changes.

6.2 Thread placement

The first component of core policies addresses the placement of threads onto cores. For example, a particular core policy may decide that threads performing expensive write operations can only run on the first acquired core. A different core policy could run a high-priority thread on Core X and idle the hypertwin of Core X to improve its single-threaded performance.

6.2.1 Design considerations

The core policy framework is designed to simplify the development of core policies and enable third-party researchers to rapidly experiment with new core policies, without needing a deep understanding Arachne internals. That is, implementors of new core policies should only need to specify where threads live, and not have to implement or understand the plumbing needed to place them there. To be more explicit, a well-designed framework should allow a core policy to specify that thread T should run on core X without needing to implement the logic for actually migrating the function and the arguments there. As another example, a core policy should be able to specify that a thread A can run on any of cores X, Y, or Z without having to implement load balancing between them. Thus, a well-designed core policy framework will delegate as many of the details of thread creation to the Arachne runtime as possible, while leaving core policies complete control over which cores threads were placed on.

Consequently, both the runtime and the core policy must participate in thread creation (and migration). I considered two possible designs for how they interact.

In the chosen design, the runtime offers thread creation APIs to the application and invokes the core policy to request a list of acceptable cores. To implement this design, the threads of an Arachne application are divided into groups called *thread classes*. Threads in the same class have similar behaviors, such as short-lived worker threads, low-priority background threads, or long-running dispatch threads. The thread class of every Arachne thread is specified by the application when the thread is created.

Each core policy defines its own thread classes and uses them in arbitrary ways to manage

threads. An application using Arachne selects a core policy at implementation time and may only create threads of the thread classes that are defined by that core policy; thread classes have no meaning outside the context of the core policy that defines them.

Core policies control thread placement by restricting the set of cores that threads of a particular class can run on. The Arachne runtime enforces the core policy's control over thread placement by asking the core policy for a list of acceptable cores each time a thread is created or migrated. A list is returned rather than a single core because it allows load-balancing (described in Section 5.3.1) to be implemented in the Arachne runtime and shared across all core policies. In cases where a core policy desires to deterministically control the exact core a new thread is placed on, it can return a list containing a single core.

An alternate design is for the core policy to offer thread creation APIs and call into the runtime to create threads on cores of its choosing. Under this design, there are no thread classes. Instead, each core policy offers a set of distinct API calls for creating different types of threads. For example, the default core policy (presented in Section 6.5) would offer the functions `createNormalThread` and `createExclusiveThread` rather than offering normal and exclusive thread classes. This alternate design offers more flexibility because such thread creation methods could accept custom arguments (in addition to the function and arguments for the new thread). For example, a thread could be scheduled for deferred creation until a time that is passed in by the application.

Despite the greater expressiveness, Arachne does not use this alternative design for the interface between the core policy and the runtime. First, the alternate design offers increased flexibility at the cost of increased surface area (complexity) in the core policy interface. The scenarios in which the marginal increase in flexibility would be beneficial all seem contrived and farfetched, while the arbitrary increase in core policy surface area (e.g., complexity) imposes cognitive load on application developers. Second, the alternate design places a burden of API design on the creators of new core policies, while the chosen design only requires them to define thread classes. By imposing an API on core policies, Arachne simplifies their design. Third, the alternate design enables and encourages core policy implementors to keep state around about special threads, such as long-running threads, because they have access to the thread handles as result of performing the thread creation. This increases complexity in core policy implementations and causes information about thread lifetimes to be shared between the runtime, the core policy and the application logic. Systems with less shared ownership are easier to manage and reason about.

6.2.2 API

The thread placement component of a core policy is implemented by providing implementations for the three methods of the pure abstract class `CorePolicy`.

- `getCores(int threadClass)`: This method is invoked by the Arachne runtime on every thread creation (and migration) to obtain a list of cores that the thread can be created on (migrated to). A core policy implementation can choose a specific core for the thread by returning a list containing a single core.
- `coreAvailable(int coreId)`: This method is invoked by the Arachne runtime to notify the core policy that a core has been granted by the core arbiter and is now available for the core policy's use.
- `coreUnavailable(int coreId)`: This method is invoked by the Arachne runtime to notify the core policy that a core is about to be returned to the core arbiter, and should no longer be used for running new threads.

6.2.3 Memory management for core lists

The return value semantics of the `getCores` method of the API involves a subtle memory management problem. The Arachne runtime calls this method to request a list of cores from the active core policy on every thread creation, so it is important that these lists are returned from the core policy to the runtime in a way that minimizes overhead in this critical path. There are two types of semantics for returning these lists: value semantics and reference semantics. In this context, value semantics means that the full contents of the core list are copied from the core policy into the runtime on every thread creation. Reference semantics means that a pointer to the list is copied, and changes to the original list will be observed by the caller. If a core list is passed with value semantics, it can quickly become unwieldy on machines with large numbers of cores, because the memory footprint of each copy of the core list scales linearly with the number of cores. Thus, it makes sense for core lists to use reference semantics.

Unfortunately, reference semantics lead to a memory management problem because core lists are sometimes static and sometimes dynamically generated. Static core lists commonly hold cores serving frequently used thread classes; a core policy might maintain a long-lived collection of cores dedicated to runnning threads of such classes, and grow or shrink the collection with size. Dynamically generated core lists are used when returning cores for thread classes that are infrequently used

and do not have a long-lived collection of cores dedicated to them. These two cases have different memory management semantics. The underlying storage for a statically allocated core list is allocated by the core policy when it first initializes and persists until program termination, so the runtime should not free that storage. If the core policy returns a dynamically generated core list, the underlying storage must eventually be freed to avoid leaking memory, but neither the runtime nor the core arbiter has sufficient knowledge to free it. Only the runtime knows when it is done using a core list, and only the core policy knows whether the core list was generated statically or dynamically; neither has complete information about both when and whether it is safe to free the storage.

There exist known complex and expensive solutions to handle this problem, such as manual reference counting and shared pointers, but Arachne uses a shockingly simple and specific solution. The core list consists of a flat array of core IDs (returned by reference) and a single bit indicating whether the underlying storage should be freed or not. This bit is set by the core policy on dynamically generated core lists, and used in the core list's destructor (which is executed by the runtime) to determine whether to free the underlying storage.

6.2.4 Arachne runtime support for thread placement

Core policies control core use, but all the thread control structures belong to the Arachne runtime. Since core policy implementations are intended to be decoupled from the runtime's implementation, it is best if they do not directly read and manipulate runtime data structures. Thus, the Arachne runtime offers a small set of support functions to help core policies manipulate threads and cores.

Core policies are comparatively new to Arachne, so the current implementation's runtime support is heavily tailored to the current default core policy's needs. The remainder of this section describes an idealized set of general-purpose primitives for Arachne's runtime to provide; Section 6.5 will discuss the set of primitives actually implemented.

Since a core policy's primary core manipulation task is to dynamically determine which cores to use for which thread classes, it may need to change the way it is using particular cores. For example, suppose that one set of cores is used for handling write requests and another set is used for handling read requests. If the ratio of read requests to write requests suddenly increased, the core policy may choose to allocate more cores to reads and fewer to writes. If radical and fast changes in core use are needed, a core policy may need to migrate threads off of cores. Thread migration is implemented by the runtime and touches runtime data structures, so it makes sense for the runtime to expose this functionality as an API to core policies. Additionally, thread creations already in progress may

attempt to create a thread onto the core that has already been repurposed to run threads of a different class. Hence, the runtime should also offer APIs for disabling and re-enabling creations to a core.

More explicitly, Arachne's runtime should offer the following three API calls.

- `void migrateThreadsFromCore(int coreId)`: When this function returns, the core with the given id should be empty of threads. The core policy does not provide a list of destination cores for the migration because threads of different classes can coexist on a single core. Instead, the runtime will ask the core policy where to migrate each thread to based on its thread class.

- `void disableCreationsOnCore(int coreId)`: When this function returns, subsequent creations to the core will be disallowed. This is used to prevent thread creations already in progress (those that have already obtained a stale core list from the core policy) from proceeding; the core policy can prevent creations that are not yet in progress by returning a core list that does not contain the core.

- `void enableCreationsOnCore(int coreId)`: When this function returns, subsequent creations to the core will be permitted.

6.3 Core estimation

The second responsibility of core policies is to compute how many cores the application needs at each point in time. Core estimation is directly tied to Arachne's goals of offering a better combination of latency and throughout. By helping applications determine an appropriate number of cores for handling their current load, a good core estimator maintains low latency for applications without using more cores than necessary to achieve a given latency target.

Both the mechanism and semantics of core estimation are entirely up to the implementer of the core policy. Core estimation does not appear in the core policy API, because Arachne's core policy framework does not dictate how, when, or even whether it happens. A particular core policy implementation could spawn a thread that periodically wakes up and checks whether the current core utilization is above or below some threshold; another core policy might define a method for applications to call when an application-observed latency increases significantly. The following are examples of possible core estimation schemes; they are by no means exhaustive and are only intended to give a flavor of what is possible.

- *Fixed core count*. A core policy could choose not to implement core estimation at all if the applications using it have a constant workload. Instead, applications using such a core policy would have the same number of cores throughout their lifetime.
- *Utilization-based estimation*: A core policy can periodically collect metrics from the runtime to compute utilization and adjust cores based on some set of thresholds.
- *Application metrics*: A core policy can perform estimation based on application metrics by invoking an application-defined function to report normalized load as observed by the application and put a threshold on that.

6.3.1 Runtime support for core estimation

As mentioned in Section 5.4, the Arachne runtime gathers basic runtime metrics and makes them available through an API that aggregates them across caller-specified subsets of cores. A core policy can utilize these metrics to estimate load.

6.3.2 Reuseable core estimation module

A core policy with multiple thread classes and a distinct set of cores for each thread class must determine both how many cores to use for running threads of each thread class and whether more cores are needed globally. If the load generated by threads of different thread classes changes over the lifetime of an application, a core policy must re-evaluate and adjust the allocation of cores used for different classes dynamically. For example, a core policy that offers thread classes for background, foreground, and long-running threads might wish to periodically re-evaluate the allocation of cores for background and foreground threads.

Arachne is packaged with a general purpose core estimator that any core policy can use to estimate whether the number of cores currently allocated to running a particular thread class is appropriate for its current load. This core estimator takes a set of cores as input and determines whether these cores are overloaded or underloaded. It outputs a recommendation to either *scale up* the number of cores or *scale down* the number of cores being used to handle the workload being offered to this set of cores.

How does this core estimation module work? Estimating the cores required for running a set of threads requires making a tradeoff between core utilization and fast response time. Even fast core allocation [41] takes on the order of microseconds, so reallocating cores every time a thread

is created is too expensive in terms of latency. If the estimator attempts 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, the estimator could optimize for fast response time by requesting enough cores to handle the worst-case load, but this would result in low utilization of cores. The more bursty a workload, the more resources it must waste in order to get a fast response.

Arachne's core estimator navigates this tradeoff by providing applications with a set of intuitive knobs for tuning the system between the dual extremes of the lowest possible latency and highest possible efficiency. The mechanisms described below for scaling up and down are parameterized: an application can select how aggressively it wants to favor latency over efficiency or vice versa, as well as how frequently to re-evaluate its decision.

The core estimator runs periodically and averages the measurements collected by the runtime since the previous time it ran to make core count recommendations. Because it averages measurements over time, core counts will not rapidly oscillate over a short period of time. The downside of averaging measurements is the estimator cannot react rapidly to extreme (and unsustained) bursts in load, on the order of microseconds.

The core estimator uses different mechanisms for scaling up and scaling down. Scaling up is intended to reduce response time, so it makes sense for the estimator to scale up based on a metric that is intuitively correlated with response time. One such metric is average number of runnable user threads on a core: the more runnable threads on a core, the longer a new thread will have to wait before it executes. For example, if three threads were runnable on a core at a particular time, then two of them had to wait while the other one ran. If there were more cores, they could run on those cores and complete in less time.

Unfortunately, computing the average number of runnable user threads directly is expensive, because doing so requires continuous monitoring of all the `wakeupTime` variables on every context. Such monitoring amounts to keeping a running weighted average of runnable threads that is updated on every thread creation, exit, block and unblock; this is costly because the each `wakeupTime` variable is updated by both itself and all cores that create threads on it, so the running average must also be updated by multiple cores, resulting in an extra cache miss on every thread creation and exit.

Due to the high cost, the core estimator instead uses a quantity called *load factor* as an approximation of the average number of runnable user threads; load factor is reasonably accurate, up-to-date, and cheap to compute. Thus, the decision to scale up is based on load factor: when the average load factor across all cores in the input set of cores reaches a configurable threshold value, the core estimator recommends an increase in core count. The intuitive correlation between load

factor and response time makes it easier for users to specify a non-default threshold value for load factor if needed. In addition, early performance measurements (not included in this dissertation) 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*: the average fraction of time that each Arachne kernel thread spends executing user threads. 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 core estimation module records the total utilization (sum of the utilizations of all cores in the input core set) each time it recommends an increase in the number of cores. When the total utilization for this set of cores returns to a level slightly less than this, the core estimator recommends a decrease in the core count (the "slightly less" factor provides hysteresis to prevent oscillations).

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 core estimation module defaults to a load factor threshold of 1.5, an averaging interval of 5 ms, and a hysteresis factor of 9% utilization. These parameter values are not sacred; they were picked empirically using the benchmarks in Chapter 7.

The approach to core estimation described above has many benefits. It enables an application to make a reasonable trade-off between its latency and efficiency needs. Moreover, it is relatively cheap to compute, imposing very little computational overhead on the applications.

Computing load factor and utilization

Load factor is computed as the ratio of *weightedLoadedCycles* to *totalCycles*. After each pass of the dispatcher through all the contexts (as described in Section 5.3) during the measurement interval, *weightedLoadedCycles* is incremented by the number of threads ran multiplied by the number of cycles that it took for the dispatcher to scan all the contexts. This quantity is cheap to compute because the dispatcher merely has to increment a thread-local counter each time a runnable thread is found, and perform a single multiplication each time it reaches context 0. *totalCycles* is the number of cycles elapsed during the measurement interval. Load factor is a reasonable approximation of average runnable thread count on a core when the average thread creation rate is relatively stable over the time it takes the dispatcher to go through all the contexts. If the rate of thread creations keeps roughly N threads on a core during the scanning interval, the dispatcher will observe and run

roughly N threads in one pass, and multiply N by the number of cycles it took to go through them. When this quantity is normalized by the total cycles elapsed during the same period, the result is the average number of runnable threads.

The *utilization* used in the computations above is actually a lower bound on true utilization. It is computed by subtracting *idleCycles* from *totalCycles* and dividing by *totalCycles*. *idleCycles* is estimated as the number of cycles the core has been looking for work to do rather than executing application code. Intuitively, a core only spends a long time looking for work if there are no runnable threads: that is the definition of being idle. However, since a core must spend a nonzero number of cycles looking for work each time a thread exits or blocks in order to find the next thread to run, *idleCycles* can be slightly overestimated, resulting in an underestimate of utilization.

6.3.3 Using the core estimation module with multiple sets of cores

The core estimation module described above can be used by a core policy to determine how to reallocate cores between threads of different classes as well as determining whether more cores are needed from the core arbiter. The following algorithm describes how a core policy might use the core estimator to determine how to change its core use.

1. Iterate over each set of cores dedicated to handling a distinct thread class.
2. Run the core estimator on it to determine whether that set of cores needs to scale up or scale down. Record the new desired number of cores for each of these classes.
3. Reallocate existing cores owned by the application among different thread classes to satisfy the new desired core counts.
4. Request more or fewer cores from the core arbiter if the net number of desired cores across all sets of cores has changed.

6.4 Interactions with the Arachne runtime

Chapter 5 hinted at interactions between the Arachne runtime and core policies but did not flesh out the details because core policies were not yet introduced. This section makes those interactions explicit and ties up the loose ends.

6.4.1 Gaining and Losing Cores

Section 5.5 described how the Arachne runtime initializes and deinitializes cores. Core policies are notified of these events through the core policy API described in Section 6.2.2. Core initialization informs the core policy that a core is ready for use by invoking `coreAvailable`. After this call returns, a core policy may allow thread creations to the new core. It also has the option to idle the core to improve global performance for the application. For example, idling a core can enable its hypertwin to run faster. Core deinitialization notifies the policy that the core is no longer available by invoking `coreUnavailable`. After this call returns, a well-behaved core policy implementation will no longer offer the core as part of any core lists it returns. However, a thread creator holding a cached list of cores containing this core can still successfully create a thread on this core after `coreUnavailable` returns. The cleaner thread described in Section 5.5 deals with this problem; it ensures that subsequently created threads get migrated and no further creations are possible.

6.4.2 Thread creation

To create threads, application code invokes the Arachne runtime function `createThreadWithClass(int class,...)`. The implementation of this function invokes `getCores` on the core policy to determine where it should create the new thread. The runtime then uses the power of two choices [36] to select a core from the the options provided, as described in Section 5.3.1. That is, it will pick two cores from the list provided by `getCores` rather than from all the cores owned by the application.

6.4.3 Thread Migration

The Arachne runtime needs to migrate threads in two scenarios:

- It is cleaning up a core for return to the arbiter.
- It is migrating threads on behalf of a core policy.

In both of these cases, the runtime invokes `getCores` with the class of each thread being migrated to request acceptable cores for the thread to migrate to. Just as in creation, migration uses the power of two choices to select a migration target.

6.5 Default core policy

Arachne's current default core policy is a first attempt at building a useful core policy; it does not represent any beliefs about what thread classes an ideal default core policy should offer. Since core policies are relatively new to Arachne, there has not been much iteration on the default core policy, and it is likely that a better default would minimally support foreground and background threads in addition to the thread classes described below.

The core 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. Exclusive threads are useful for long-running, high-priority threads, such as 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. This pair of thread classes represent the bare minimum for getting reasonable performance from integrating Arachne with RAMCloud and memcached; both used exclusive threads as dispatch threads and normal threads for request handling.

6.5.1 Implementing exclusive threads

Offering an exclusive thread class introduces the challenge of managing dedicated cores (within an application) efficiently, with minimal waste of CPU cycles. A naive implementation might involve requesting a new core from the arbiter whenever the application requests the creation of an exclusive thread; this approach causes the thread creation to take tens of microseconds (while the arbiter grants a core). Another potential implementation might pre-request additional cores from the core arbiter in anticipation of exclusive threads being created, but this is inefficient if the core policy incorrectly guesses the number of exclusive cores needed by the application.

In the default core policy's implementation, cores are allocated and de-allocated from exclusive use in an on-demand fashion. When cores are granted by the core arbiter, they join the pool of shared cores used for running normal threads. The default core policy keeps a cache of cores recently used for exclusive threads, which is initially empty. If the Arachne runtime requests a core for creating an exclusive thread, the core policy first examines the cache to see if any of the cores in the cache are empty of threads using the runtime support function `findAndExtractUnusedCore`, described in Section 6.5.2. If a core is available, it is placed in a core list by itself and returned to the runtime. Otherwise, the runtime picks an arbitrary core from the pool of shared cores and invokes `prepareForExclusiveUse` (described in Section 6.5.2) on it to remove other threads

and prevent normal thread creations. Afterwards, it adds the core to the cache of exclusive cores and returns the core to the runtime. A core in the cache whose exclusive thread has exited may be converted back into a shared core if core estimation determines that more cores are needed.

Tracking exclusive thread termination

The default core policy's exclusive thread class introduces the problem of core reclamation: how does the core policy know when an exclusive thread has exited? It is beneficial for a core policy to discover the exit of an exclusive thread so that it can recycle the core, either for use by normal threads or for return to the core arbiter. If there are long periods of time where the core policy does not discover the exited threads, entire cores can be wasted during these periods.

There are multiple options for handling the core reclamation problem, but most of them boil down to one of two models. The first is to have the core policy receive notifications when an exclusive thread exits. This can be done by requiring application developers to make an explicit API call to signal the imminent termination of an exclusive thread, or by modifying the runtime to notify the core policy on every thread exit. The former is unreliable because it relies on all application developers to behave correctly and is unenforceable. The latter directly increases exit latency for every thread. In the second model, the core policy drives the core reclamation by asking the Arachne runtime whether the core that the exclusive thread was scheduled on is clear of threads. It might do this periodically or on demand when more cores are needed. This adds no overhead in the common case, but can result in periods of time when a core is idle and unavailable to run threads.

Since Arachne is designed for low latency, its design errs on the side of wasting resources in the name of lower latency. Arachne's default core policy checks for empty cores when the core estimator determines that more shared cores are needed for running normal threads. The same thread that runs the load estimator also periodically looks for exclusive threads that have exited and reclaims the cores they were running on.

6.5.2 Runtime support tailored to the default core policy

As mentioned in Section 6.2.4, the Arachne runtime's current implementation offers support that is tailored for the default core policy's implementation; there is substantial room for improvement in both the default core policy's implementation and the support functions (which are too special-purpose) offered by the runtime. The functions that are currently implemented by the runtime are listed below.

- `findAndExtractUnusedCore`: Invoked by core policies to analyze a list of cores and extract a core which is currently not running user threads. Core policies cannot perform this action because the data structures that track the occupancy of cores are internal to the Arachne runtime. This method handles the exclusive thread termination problem described in Section 6.5.1.

- `prepareForExclusiveUse`: Invoked by core policies to clear all threads from a core and prevent future creations. This method allows core policies to convert a core that runs threads of class A to a core that serves threads of class B, in the case where class B threads do not want to share cores with class A threads or vice versa.

6.5.3 Core estimation in the default core policy

The default core policy uses the built-in core estimator described in Section 6.3.2 to estimate and adjust the number of shared cores. If the core estimator determines that more cores are needed, the default core policy will first check the cache of exclusive cores to check if cores are available before requesting another core from the core arbiter.

6.6 Summary

This chapter introduced the notion of core policies and explained why they are useful for facilitating thread placement and core estimation on behalf of applications. It described the APIs for implementing new core policies, and pointed out how core policies interact with the Arachne runtime. Finally, it described Arachne's default core policy.

Chapter 7

Evaluation

This chapter evaluates Arachne's implementation using microbenchmarks and real applications. It seeks to answer 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 thread management 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 7.1 describes the configuration of the machines used for benchmarking.

CloudLab m510[48]	
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 Ethernet
Switches	HPE Moonshot-45XGc

Table 7.1: 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). Machines were not configured to perform packet steering such as RSS or XPS.

Benchmark	Arachne		Arachne RQ		std::thread	Go	uThreads
	No HT	HT	No HT	HT			
Thread Creation	275 ns	320 ns	524 ns	520 ns	13329 ns	444 ns	6132 ns
One-Way Yield	83 ns	149 ns	137 ns	199 ns	N/A	N/A	79 ns
Null Yield	14 ns	23 ns	13 ns	24 ns	N/A	N/A	6 ns
Condition Notify	251 ns	272 ns	459 ns	471 ns	4962 ns	483 ns	4976 ns
Signal	233 ns	254 ns	N/A	N/A	N/A	N/A	N/A
Thread Exit Turnaround	328 ns	449 ns	408 ns	484 ns	N/A	N/A	N/A

Table 7.2: Median cost of scheduling primitives. Creation, notification, and signaling are measured end-to-end, from initiation in one thread until the target thread wakes up and begins execution on a different core. 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 immediately to the yielding thread. "Thread Exit Turnaround" measures the time from the last instruction of one thread to the first instruction of the next thread to run on a core. N/A indicates that the threading system does not expose the measured primitive. "Arachne RQ" means that Arachne was modified to use a ready queue instead of the queueless dispatch mechanism described in Section 5.3. "No HT" means that each thread ran on a separate core using one hyperthread; the other hyperthread of each core was inactive. "HT" means the other hyperthread of each core was active, running the Arachne dispatcher.

7.1 Implementation and source code structure

Arachne is implemented in C++ on Linux; source code is available on GitHub [44]. The core arbiter contains 4500 lines of code, the runtime contains 3400 lines, and the default core policy contains 270 lines. The initial design and implementation of the core arbiter was done in collaboration with Jacqueline Speiser. The source code is distributed as a git repository with the following submodules.

- **Arachne**: This repository contains the runtime, the core policy and their unit tests.
- **CoreArbiter**: This repository contains the core arbiter server and client library and their unit tests.
- **ArachnePerfTests**: This repository contains code that runs the microbenchmarks described later in this chapter.
- **PerfUtils**: This repository contains an assortment of performance measurement libraries and tools for processing performance statistics, developed in my research group over the years. Both the runtime and the core arbiter depend on libraries in this respository.

7.2 Threading Primitives

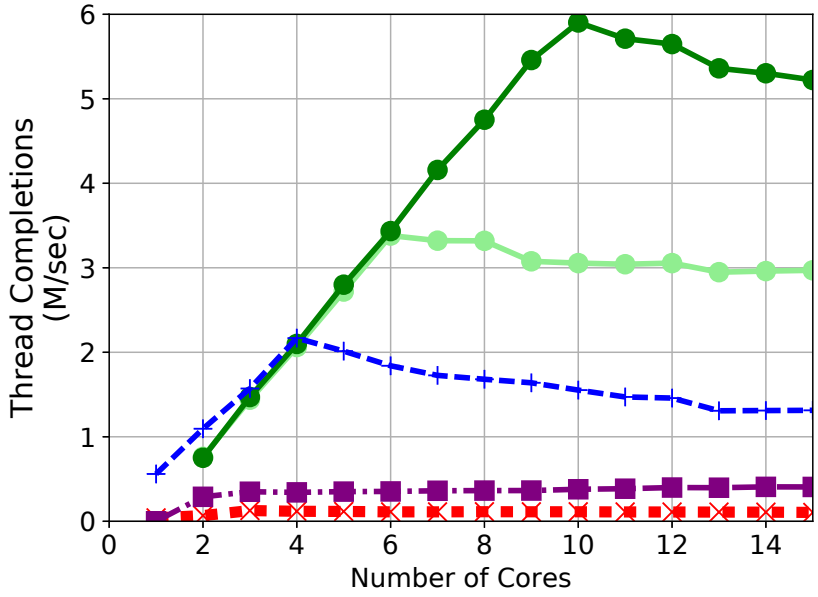
Table 7.2 compares the cost of basic thread operations in Arachne with those in C++ `std::thread`, Go, and uThreads [8]. `std::thread` is a wrapper around threads implemented by the Linux kernel; Go implements threads at user level in the language runtime; and uThreads uses kernel threads to multiplex user threads, like Arachne. uThreads is a highly rated C++ user threading library on GitHub and claims high performance. For each of the Arachne benchmarks, we report numbers under two configurations of hyperthreading. In the ”No HT” configuration, the hypertwins of the core(s) involved in the experiment are deliberately idled to remove hypertwin interference. In the ”HT” experiment, the hypertwins of the cores are running the Arachne dispatcher. The latter configuration is likely more representative of a real deployment of Arachne, since it is better for overall efficiency, but a system could choose to idle one hypertwin to get improved performance for a particularly important core. The measurements use microbenchmarks that repeatedly run an operation to find the median time, so they represent best-case performance.

The core semantics for multi-thread operations vary widely across thread libraries. Consider thread creation as an example. 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. In the case of `std::thread`, the kernel scheduler decides where the new thread will run.

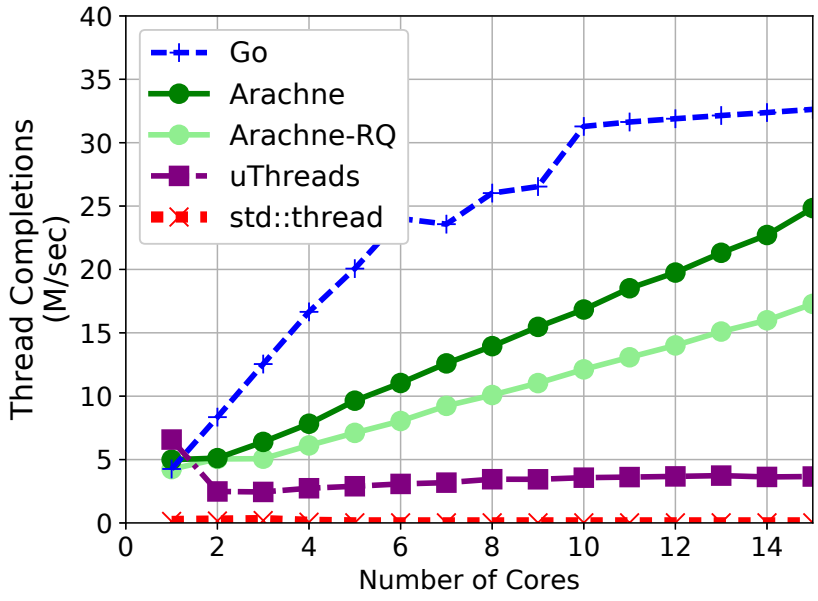
Arachne’s thread operations are considerably faster than any of the other systems, except that yields are faster in uThreads. Arachne’s cache-optimized design performs thread creation significantly faster than Go, even though Arachne places new threads on a different core from the parent while Go creates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

To evaluate Arachne’s queueless approach for finding threads to run, I modified Arachne to use a wait-free multiple-producer-single-consumer queue [11] on each core to identify runnable threads instead of scanning over all contexts. I selected this implementation for its speed and simplicity from several candidates on GitHub. Table 7.2 shows that the queueless approach is 28–40% faster than one using a ready queue (I counted four additional cache misses for thread creation in the ready queue variant of Arachne).

Arachne’s thread creation mechanism is designed not just to minimize latency, but also to provide high throughput. I ran two experiments to measure thread creation throughput. In the first experiment (Figure 7.1(a)), a single “dispatch” thread creates new threads as quickly as possible



(a)



(b)

Figure 7.1: 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.

(this situation might occur, for example, if a single thread is polling a network interface for incoming requests). The new threads spin for one µs, bump a counter to record completion and exit. 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, because threads are unable to run until they are first stolen by another core. At low core counts, Go exhibits higher throughput than Arachne because it runs threads on the creator's core in addition to other available cores, while Arachne uses the creator's core only for dispatching.

The second experiment measures thread creation throughput using a distributed approach. The experiment is initialized with $3N$ threads, where N is the number of cores. Each thread creates one child thread, records its own completion, and then exits. The results are shown in Figure 7.1(b). In this experiment both Arachne and Go's throughput scaled well with the number of available cores. Neither uThreads nor `std::thread` had scalable throughput; uThreads had 10x less throughput than Arachne or Go and `std::thread` had 100x less throughput. Go's approach to thread creation worked well in this experiment; each core created and executed threads locally 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. Arachne's performance reflects the costs of thread creation and exit turnaround from Table 7.2, as well as occasional conflicts between concurrent thread creations.

Figure 7.1 also includes measurements of the ready queue variant of Arachne. Arachne's queueless approach provided higher throughput than the ready queue variant for both experiments.

7.3 Integrating Arachne into memcached

Memcached [31] is a remote in-memory key-value cache that is widely deployed in industry and frequently optimized in research.

I worked with Qian Li to modify memcached [31] version 1.5.6 to use Arachne (as her rotation project); the source is available on GitHub [29]. The sections below describe memcached's original threading model and the changes introduced by integrating with Arachne.

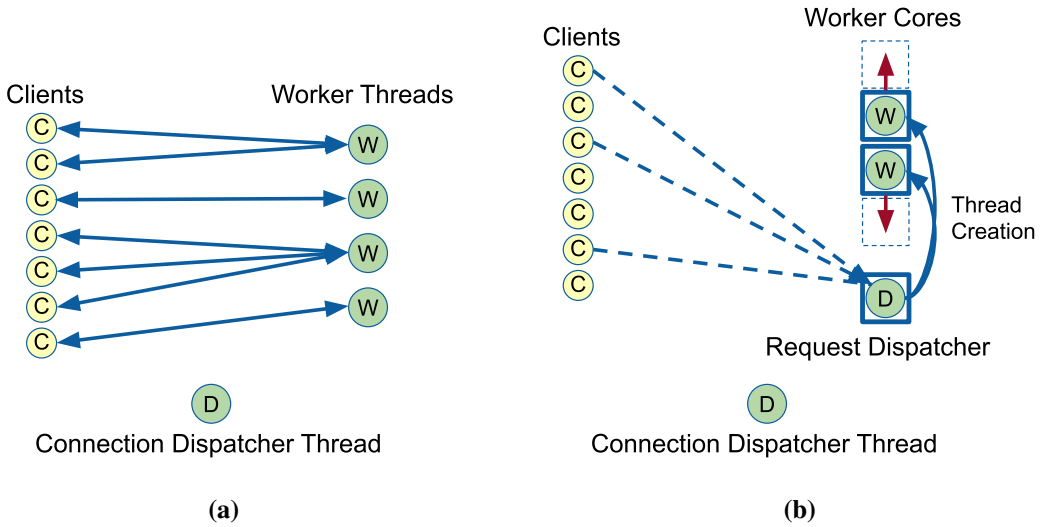


Figure 7.2: Memcached threading model before and after Arachne integration. In (a) connections are statically assigned to worker threads in a fixed-size pool. In (b) requests are dispatched to pool of cores which vary with load.

7.3.1 Memcached's original threading model

Memcached's original threading model (depicted in Figure 7.2(a)) consists of a single *connection dispatcher* thread and several *worker threads*. The number of worker threads is set by the operator when the memcached server instance is started. Each of these threads normally blocks in a `libevent` loop until a request arrives on one of its connections. When a memcached client sends its first request to a memcached server, a connection is established for use in future communications. The state for this conection is created by the connection dispatcher, which wakes up and assigns the connection to a worker thread. All future requests on the connection will be handled by the assigned worker thread. Connections are asigned in round-robin order across the worker threads.

Memcached's threading model delivers high throughput and efficient use of cores if the following assumptions hold.

- Request traffic is constant over time.
- Request traffic is roughly evenly distributed across connections.
- No other applications run on the system.

Since the assumptions above rarely hold in practice, memcached suffers from a variety of thread-model-induced problems. Load on memcached servers running in data centers varies with time of

day. Because memcached's threading model requires static setting of the number of worker threads at initialization, an operator must provision memcached servers with enough resources for peak load to ensure low latency. This approach wastes cores at lower loads. If one runs background applications (breaking the last assumption) to utilize the cores under low load, such applications will contend with memcached for cores at high load, resulting in latency increases when memcached threads are descheduled. An additional problem with the threading model is that if request traffic is not uniform across all connections, some worker threads may become overloaded while others are idle. This enables scenarios where the system as a whole is overloaded while some cores are idle, reducing overall throughput. Finally, because memcached uses kernel threads, the kernel can place two worker threads on the same core, causing requests handled by both to experience high latencies.

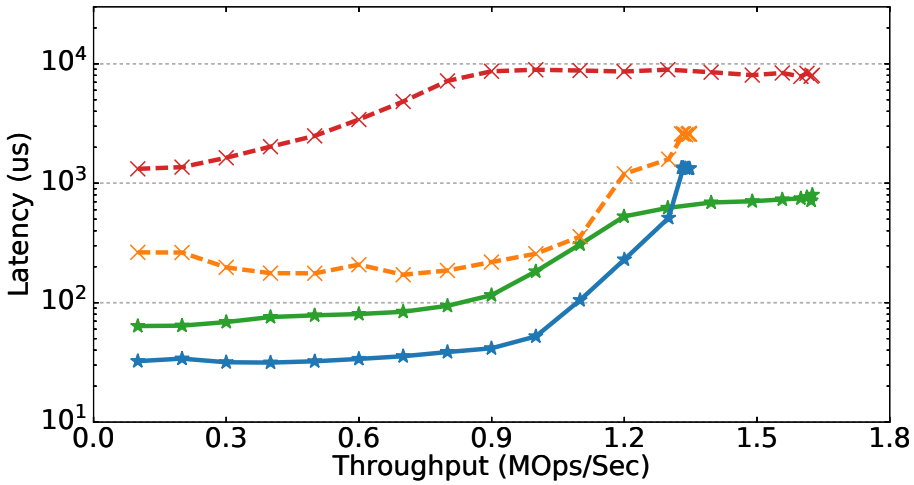
Memcached could theoretically avoid many of these problems if it used cores instead of kernel threads, performed load-balancing per-request instead of per-connection, and dynamically adjusted its core usage instead of statically setting offered parallelism. Under such a model, memcached would be able to enjoy low latency because its own kernel threads would not interfere with one another. Its performance would be unaffected by uneven loads across different connections. Finally, dynamic scaling can allow memcached to use cores commensurate with its load.

7.3.2 Memcached's threading model after integrating with Arachne

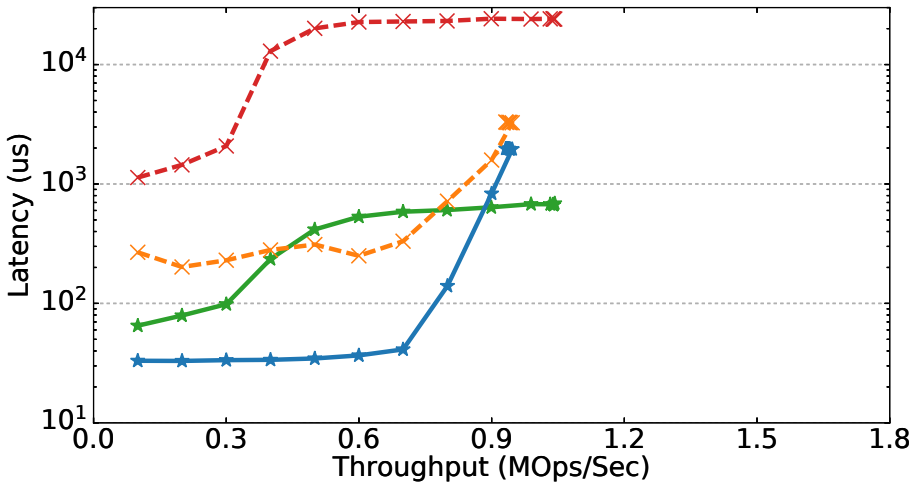
We hypothesized that a thread model based on Arachne could solve the problems above, so we modified memcached to use Arachne. In the modified version ("memcached-A") (shown in Figure 7.2(b)), the pool of worker threads is replaced by a single request dispatcher thread, which waits for incoming requests on all connections (we retain the original memcached connection dispatcher; it always assigns connections to this single worker thread). When a request arrives, the request dispatcher creates a new Arachne thread, which lives only long enough to handle all available requests on the connection. If the thread creation fails due to overload, the dispatcher handles the request in-place. Memcached-A uses Arachne's default core policy; the dispatch thread is "exclusive" and workers are "normal" threads. The number of cores used automatically scales with load.

7.4 Arachne's benefits for memcached

Memcached-A provides two benefits over the original memcached. First, it provides finer-grain load balancing (at the level of individual requests rather than connections). Request-based load balancing is agnostic to which connections requests come in, so connection skew no longer results in



(a) memcached: 16 worker threads, 16 cores



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

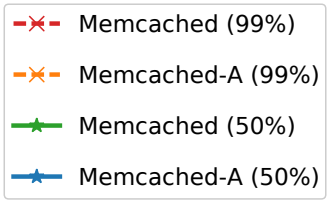


Figure 7.3: Median and 99th-percentile request latency as a function of achieved throughput for both memcached and memcached-A, under the Realistic benchmark. Each measurement ran for 30 seconds after a 5-second warmup. Y-axes use a log scale.

Experiment	Program	Keys	Values	Items	PUTs	Clients	Threads	Conns	Pipeline	IR Dist
Realistic	Mutilate [38]	ETC	ETC	1M	.03	20+1	16+8	1280+8	1+1	GPareto
Colocation	Memtier [32]	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 7.3: Configurations of memcached experiments. Program is the benchmark program used to generate the workload (the version of Memtier used here is modified [33] from the original). Keys and Values give sizes of keys and values in the dataset (ETC recreates the Facebook ETC workload [6], 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. 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.

degraded performance. Second, memcached-A reduces performance interference, both between kernel threads (there is no multiplexing) and between applications (cores are dedicated to applications). Consequently, it is possible to run background applications on the same machine as memcached-A to soak up CPU cycles without increasing its latency.

We performed three experiments with memcached; their configurations are summarized in Table 7.3. The first experiment, Realistic, measures latency as a function of load under realistic conditions; it uses the Mutilate benchmark [28, 38] to recreate the Facebook ETC workload [6]. Figure 7.3(a) shows the results. The maximum throughput of memcached is 20% higher than memcached-A (1.6 million QPS vs 1.3 million QPS). This is because memcached-A uses two fewer cores (one core is reserved for unmanaged threads and one for the dispatcher); in addition, memcached-A incurs overhead to spawn a thread for each request, because thread start and exit consume cycles. However, memcached-A has significantly lower latency than memcached. Thus, if an application has service-level requirements, memcached-A provides higher useable throughput. 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). For any SLO up to about 500 µs median latency, memcached-A provides higher throughput than memcached.

At the 99th percentile, memcached-A's latency ranges from 3–40x lower than memcached. Memcached never achieves 99th percentile latency lower than 1 ms, whereas memcached-A provides 99th percentile latency of a few hundred microseconds over most of its operating range. Low

latency is one of the primary reasons for using memcached, so Arachne provides a significant advantage. One of reasons for this difference is that Linux migrates memcached threads between cores frequently: at high load, we found that 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 Arachne's design goals 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 7.3(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 continued to operate efficiently: latency was about the same as in Figure 7.3(a). In contrast, memcached experienced significant increases in both median and tail latency, presumably due to additional multiplexing; with a median 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 Arachne's core estimator. It also measured memcached and memcached-A performance when colocated with a compute-intensive application (the x264 video encoder [34]). The results are in Figure 7.4. Figure 7.4(a) shows that memcached-A used 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 a background application, 99th-percentile latency was 10x higher for memcached than for memcached-A. To investigate the reasons for this difference, we instrumented memcached to find out where its worker threads were being placed by the kernel. We observed that the Linux scheduler often placed two of memcached's workers on the same hyperthread even when more hyperthreads were available, and hypothesize that this is one reason for the increased latency.

When a background video application was colocated with memcached, memcached's latency doubled, both at the median and at the 99th percentile, even though the background application ran at lower priority. 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 7.4(a) shows that memcached-A ramped up its core usage more quickly when

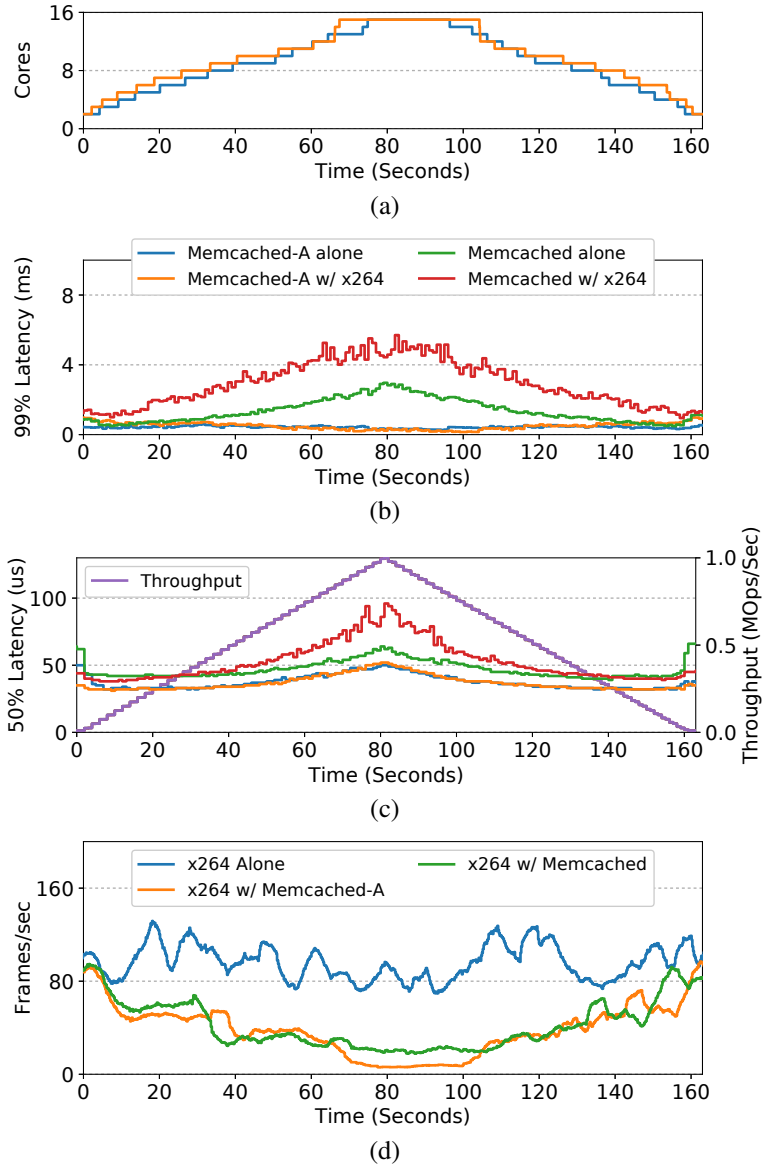


Figure 7.4: Memcached performance in the Colocation experiment. The request rate increased gradually from 10 Kops/sec to 1 Mops/sec and then decreased back to 10 Kops/sec. In some experiments the x264 video encoder [34] ran concurrently, using a raw video file (sintel-1280.y4m) from Xiph.org [30]. When memcached-A ran with x264, the core arbiter gave memcached-A as many cores as it requested; x264 was not managed by Arachne, so Linux scheduled it on the cores not used by memcached-A. When memcached ran with x264, the Linux scheduler determined how many cores each application received. x264 sets a "nice" value of 10 for itself by default; we did not change this behavior in these experiments. (a) shows the number of cores allocated to memcached-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 colocated with memcached or memcached-A.

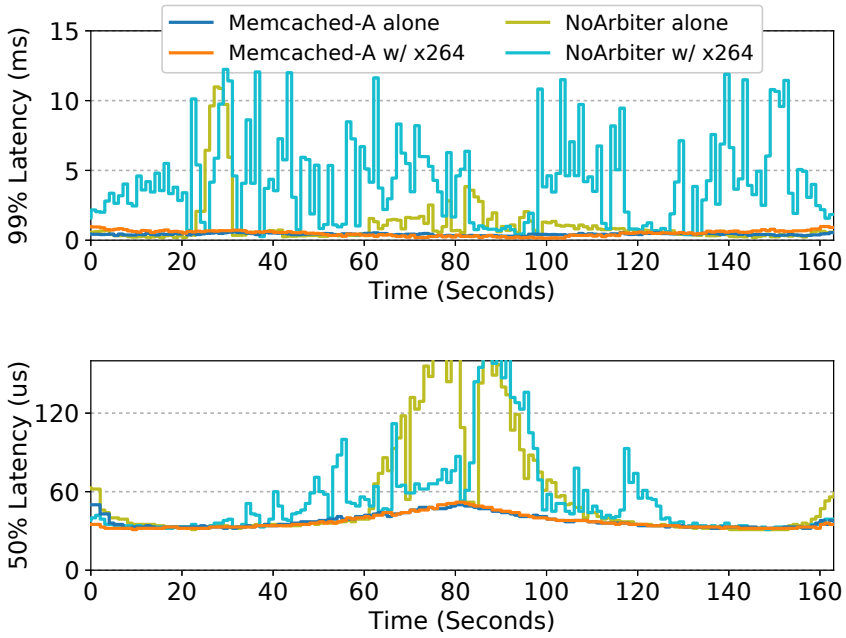


Figure 7.5: Median and tail latency in the Colocation experiment with the core arbiter ("Memcached-A", same as Figure 7.4) and without the core arbiter ("NoArbiter").

colocated with the video application. This suggests that there actually was some performance interference from the video application, but that the core estimator detected this and allocated cores more aggressively to compensate.

Figure 7.4(d) shows the throughput of the video application. At high load, the video application's 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; unfortunately, this degraded the performance of memcached.

Figure 7.5 shows that dedicated cores are fundamental to Arachne's performance. For this figure, we ran the Colocation experiment using a variant of memcached-A that did not have dedicated cores: instead of using the core arbiter, Arachne scaled by blocking and unblocking kernel threads on semaphores, and the Linux kernel scheduled the unblocked threads. The number of kernel threads was the same in both cases. As shown in Figure 7.5, this resulted in significantly higher latency both with and without the background application. Additional measurements revealed that latency spikes occurred when Linux descheduled a kernel thread but Arachne continued to assign new user

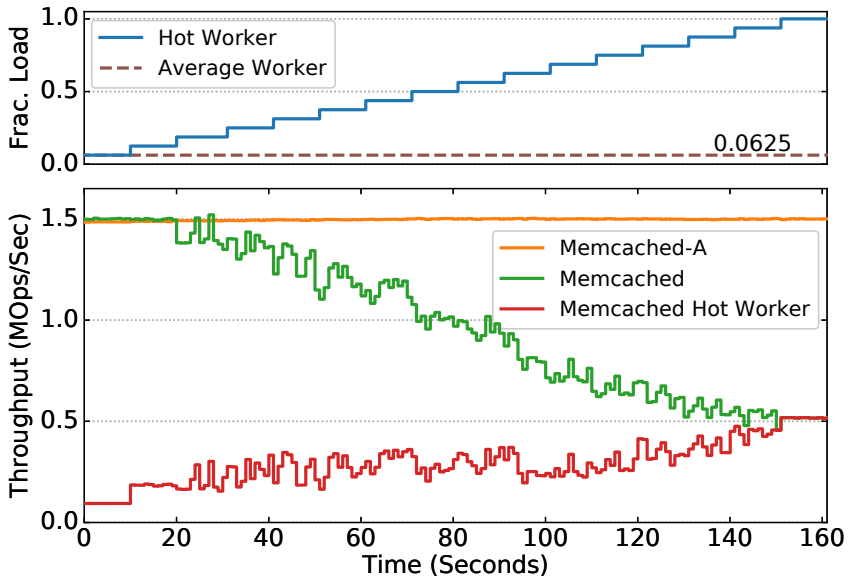


Figure 7.6: The impact of workload skew on memcached performance with a target load of 1.5 Mops/sec. Initially, the load was evenly distributed over 512 connections (each memcached worker handled 512/16 = 32 connections); over time, an increasing fraction of total load was directed to one specific "hot" worker thread by increasing the request rate on the hot worker's connections and decreasing the request rate on all other connections. The bottom graph shows the overall throughput, as well as the throughput of the overloaded worker thread in memcached.

threads to that kernel thread; during bursts of high load, numerous user threads could be stranded on a descheduled kernel thread for many milliseconds. Without the dedicated cores provided by the core arbiter, memcached-A performed significantly worse than unmodified memcached.

The final experiment for memcached is Skew, shown in Figure 7.6. This experiment evaluates memcached performance when the load is not balanced uniformly across client connections. Since memcached statically parititions client connections among worker threads, hotspots can develop, where some workers are overloaded while others are idle; this can result in poor overall throughput. In contrast, memcached-A performs load-balancing during each request, so performance is not impacted by the distribution of load across client connections.

7.4.1 Caveats and Limitations

In addition to the connection dispatcher and worker thread pool, memcached contains a few periodically running maintenance threads. A full integration with Arachne would convert these threads into

Arachne threads so that they do not affect offered parallelism, but our integration is incomplete, so these threads run on the unmanaged cores in our experiments. As far as we could tell, not converting these threads did not impact our measurements, and work by others [16] indicates that converting them is feasible.

Comparisons between our evaluation and both past [10, 53] and later [41] efforts to improve memcached performance might lead one to conclude that Arachne introduces less substantial improvements to memcached's latency and throughput than other systems. For example, Shenango [41] reports five million requests per second with a 99.9% response time of 93 µs. This perception is misleading because Arachne's evaluation measures the impact of core-aware thread management in isolation, while other works combine networking improvements with carefully hand-tuned thread placement and load balancing. IX [10] rewrite's the kernel network stack, and Shenango bypasses the kernel's network stack using dpdk [2]. Previous work [10] and our measurements confirm that memcached spends about 70% of its time doing networking-related tasks, so optimizations that do not change the networking stack will necessarily show more modest gains compared to optimizations that replace the networking stack. Note that Arachne's techniques are orthogonal and complementary to techniques for speeding up the networking stack, such as kernel bypass [2] and new transports [37]. Arachne could be combined with a higher performance network stack to produce much higher performance.

7.5 Arachne's Benefits for RAMCloud

I worked with Peter Kraft to modify RAMCloud [43] to address the efficiency issues in RAMCloud described in Chapter 2 using Arachne. In the modified version ("RAMCloud-A"), the long-running pool of worker threads is eliminated, and the dispatch thread creates a new Arachne thread for each request. Threads that are busy-waiting for nested RPCs to complete yield after each iteration of their polling loop (blocking rather than yielding is a more correct design, but it would have required more substantial changes to RAMCloud). This allows other requests to be processed during the waiting time, so that the core is not wasted. As in memcached-A, the number of cores automatically scales with load.

Figure 7.7 shows that RAMCloud-A has 2.5x higher write throughput than RAMCloud. On the YCSB benchmark [14] (Figure 7.8), 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 and thread

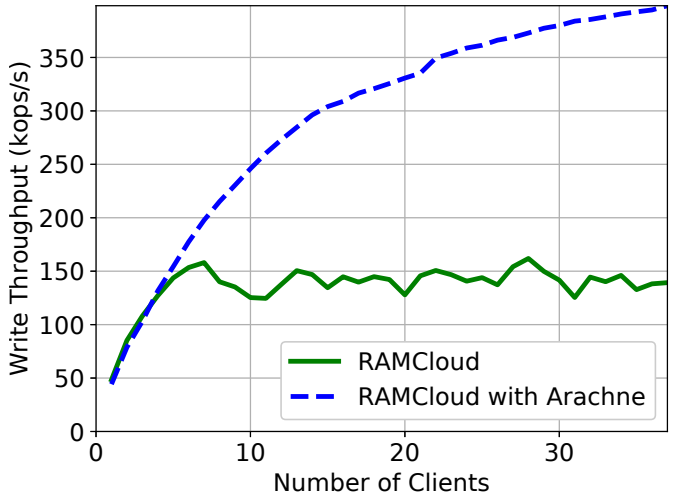


Figure 7.7: Throughput of a single RAMCloud server when multiple clients perform continuous back-to-back write RPCs of 100-byte objects. Throughput is measured as the number of completed writes per second.

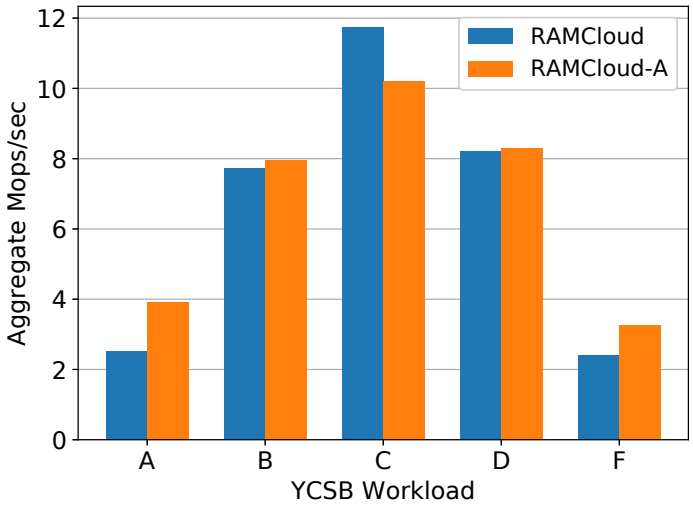


Figure 7.8: Comparison between RAMCloud and RAMCloud-A on a modified YCSB benchmark [14] using 100-byte objects. Both were run with 12 storage servers. Y-values represent aggregate throughput across 30 independent client machines, each running with 8 threads.

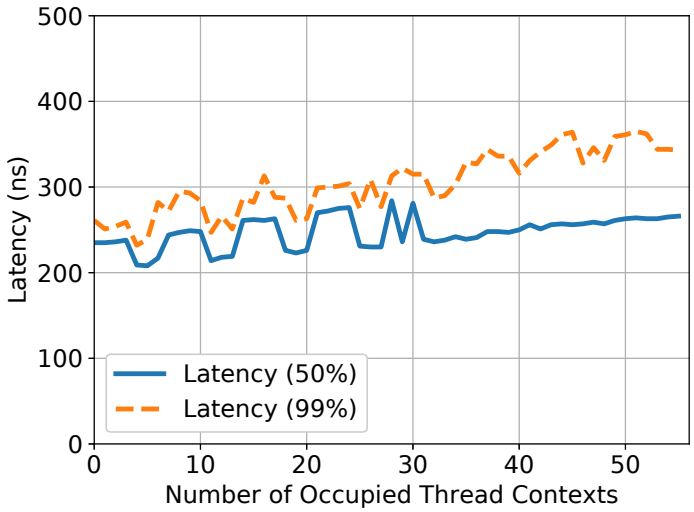


Figure 7.9: Cost of signaling a blocked thread as the number of threads on the target core increases. Latency is measured from immediately before signaling on one core until the target thread resumes execution on a different core.

exit. These experiments demonstrate that Arachne makes it practical to schedule other work during blockages as short as a few microseconds.

7.6 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; instead, the dispatcher scans the `wakeupTime` variables for occupied thread contexts until it finds a runnable thread. Consequently, as a core fills with threads, its dispatcher must iterate over more and more contexts. 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 7.9 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 miss time. 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. Arachne's worst-case performance in Figure 7.9 is still better than the ready queue variant of Arachne in Table 7.2.

Figures 7.10 and 7.11 show the performance of Arachne's core allocation mechanism. Figure

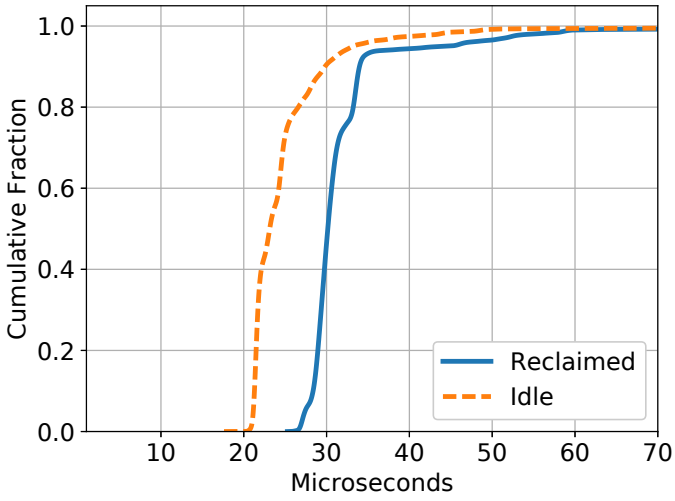


Figure 7.10: Cumulative distribution of latency from core request to core acquisition (a) when the core arbiter has a free core available and (b) when it must reclaim a core from a competing application.

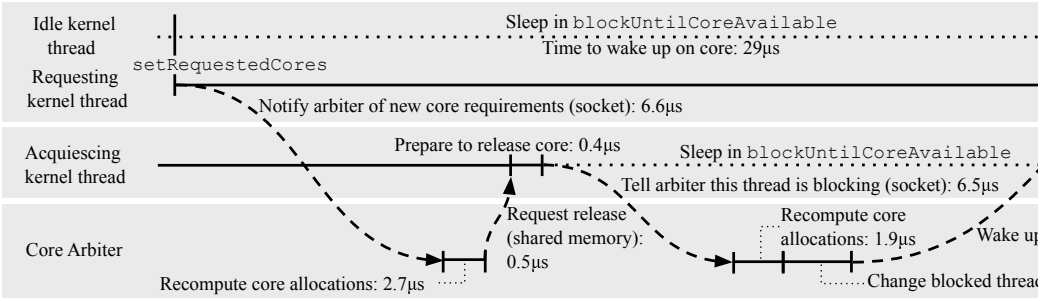


Figure 7.11: Timeline of a core request to the core arbiter. There are two applications. Both applications begin with a single dedicated core, and the top application also begins with a thread waiting to be placed on a core. The top application has higher priority than the bottom application, so when the top application requests an additional core the bottom application is asked to release its core.

7.10 shows the distribution of allocation times, measured 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 includes signaling another process and waiting for it to release a core. Figure 7.10 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 7.11 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. Recent work on core allocation [41] has shown that much of this cost could be eliminated (down to 8-10 µs for reallocating a core) using a shared `eventfd` in place of a Unix domain socket.

7.7 Summary

This chapter evaluated Arachne along different dimensions against various systems using a variety of benchmarks. Arachne's thread primitives are faster than thread primitives implemented by Go, uThreads and `std::thread`. Its cache-optimized thread dispatching mechanism is significantly faster than using a wait-free queue. Adding Arachne to memcached 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.

Chapter 8

Related Work

Improving the efficiency of CPU scheduling has been of interest to academia and industry since the mid-1970's [20, 27]. Arachne was not the first and will not be the last system to improve the software stack for CPU scheduling; it inherits many ideas from the past. This chapter illustrates where Arachne fits into this rich research ecosystem.

8.1 Core-aware scheduling

The idea of core-awareness in thread management was first introduced by scheduler activations [4], and has inspired several recent systems [47, 41] targeting improved performance for data center applications, including Arachne itself. The systems differ in semantics, complexity of implementation, core management mechanism and in the degree of core awareness visible to applications.

Arachne's formulation of core-awareness in thread management consists of the following ideas; each subsection that follows will discuss which of these ideas the other systems implement.

- **Core allocation**: The operating system or a process operating on its behalf allocates dedicated cores to applications rather than threads.
- **Topology-aware core allocation**: The system performing core allocation among applications allocates cores to applications in a manner that promotes locality within an application and reduces interference between applications. In practice, this means that they try to offer hyper-twins to the same application, and avoid splitting applications across CPU sockets.
- **Cooperative core preemption**: Cores are never taken away from applications without warning; applications are notified and have time to clean up before losing cores that have been

granted to them.

- **Core estimation**: Applications determine for themselves how many cores they need and negotiate with the system for the number of cores they need.
- **Core-aware thread placement**: Applications have visibility into system topology and can implement a mechanism for controlling thread placement on cores. In Arachne, this is the thread placement component of core policies.

8.1.1 Scheduler activations

In proposing scheduler activations [4], Anderson et. al pointed out the limitations of using either only user threads or only kernel threads discussed in Section 3.1, and argued that naive multiplexing of user threads over kernel threads does not eliminate their shortcomings. Instead, kernel support for user threads is needed to "combine the functionality of kernel threads with the performance and flexibility of user-level threads". Towards this end, scheduler activations introduces the notion of a *virtual multiprocessor*, an abstraction of a dedicated physical machine. Although the framing differs, a virtual multiprocessor is semantically equivalent to a collection of Arachne's dedicated cores. As in Arachne, each application knows exactly how many and which cores are part of its virtual multiprocessor, and has full control over which threads run on which cores. In contrast to Arachne's cooperative preemption, the operating system kernel of scheduler activations can remove a core from an application at any time and notifies the application afterwards of the loss.

In the virtual multiprocessor model, the kernel offers cores to applications by means of a *scheduler activation*, which is an upcall from the OS kernel into userspace in modern terminology. This upcall grants control to a user level thread scheduler, which puts it in control of a core. It is also used to notify the user level thread scheduler of other cores being preempted, threads being blocked in the kernel, and threads being unblocked in the kernel. Scheduler activations are never resumed after blocking, so the user threading system must migrate the context for a blocked user thread to a new activation. Compared to Arachne, the scheduler activations upcall mechanism is more heavyweight, requires kernel modifications, and permits preemptions to occur at potentially awkward times for the application, such as inside a critical section. To work around the last issue, scheduler activations has a cumbersome, special-purpose workaround. When an upcall reports a preemption, the thread system checks if the preempted user thread was in a critical section. If so, the thread is immediately migrated to the activation that did the upcall, and continues execution until the critical section is complete.

In contrast, Arachne's formulation of core aware thread management, which is based on kernel threads rather than upcalls, involves fewer kernel crossings and no kernel modifications. Scheduler activations paid significant complexity to enable processor preemption during blocking kernel calls. Arachne's implementation does not address the blocking kernel call problem, but its design is compatible with modern solutions proposed in other recently developed thread packages [56, 8, 17]. Scheduler activations does not implement core estimation, core-aware thread placement, topology-aware core allocation, and cooperative core preemption; it defers user level thread management to the FastThreads [5] library.

8.1.2 Akaros

Akaros [47] is a full-fledged research operating system that follows the tradition of scheduler activations in allocating dedicated cores to applications. It claims the "Many-Core Process" abstraction as its primary contribution, and presents experiments showing an order of magnitude less interference (from kernel activity to application threads) on isolated cores compared to Linux. As with Arachne, processes in Akaros are spawned as single-core and perform user space scheduler initialization. Then, they make a request to the Akaros kernel to morph into a multi-core process, which results in the kernel executing the application's user space schedulers on dedicated cores. Akaros processes do not suffer from loss of cores when they make blocking system calls, because Akaros makes all system calls functionally asynchronous. If a system call needs to block, the kernel will automatically return control to userspace by invoking the userspace scheduler.

Akaros' functionality is similar to the Arachne core arbiter, and its applications enjoy a greater level of core isolation because it can remove in-kernel threads from dedicated cores. It also implements cooperative core preemption. Akaros does not implement topology-aware core allocation, core estimation, or core-aware thread placement. Moreover, its benefits cannot be practically enjoyed because it does not appear to have reached a level of maturity that can support meaningful performance measurements. On the contrary, Arachne's core arbiter runs on commodity Linux and exhibits measurable performance gains from core isolation.

The creators of Akaros made an additional contribution to core-aware scheduling: they ported the Go runtime to Akaros. While the performance of the port has not been measured due to Akaros' lack of maturity, its existence proves the feasibility of moving Go on top of an allocated cores abstraction. This suggests that Go could also be ported to other systems that offer allocated cores, such as Arachne.

8.1.3 Shenango

Shenango is a new runtime system that combines kernel bypass networking, network packet steering, and fast core allocation to achieve low latency and high throughput in data center workloads. Shenango consists of an independent process called the IOKernel and a runtime thread scheduler. The IOKernel steers network packets to the appropriate core for an application, estimates core needs every five microseconds, and performs core allocations using CPU affinities. Like Arachne, Shenango uses kernel threads as proxy for applications to control cores. Like the core arbiter, the IOKernel uses asymmetric communication when communicating with applications: it uses shared memory to pass information to applications, while applications use a shared `eventfd` to communicate with the IOKernel, which doubles as a means of parking client threads.

Shenango implements some of Arachne's functionality with lower overhead; Arachne might benefit from adapting some of its techniques. Shenango's core estimation is simpler and faster than Arachne's default core estimation: if there are new packets or threads queued in the last five microseconds, then it allocates a new core to the application. If a kernel thread in an application runs out of work and cannot find work to steal, it yields its core. This differs from Arachne, in which core estimation results in a core request to the core arbiter, which then requests a preemption. In order to ramp up quickly, Shenango does not do cooperative core preemption. Shenango's core allocation is topology-aware, and runs 3x faster than Arachne's core allocation (8-9 µs vs 30 µs) because it uses a shared `eventfd` instead of Unix domain sockets.

Like Arachne, Shenango's runtime offers fast, lightweight thread operations on top of allocated cores. The published results indicate that its threading primitives are faster than Arachne on a single core, but they do not show benchmarks for cross-core operations. One of the authors indicated that the Shenango runtime always creates threads on the same core and uses work stealing to do load balancing after the thread creation. The published benchmark does not include the cost of work-stealing or any other form of load-balancing, so Shenango's published thread creation latency is not directly comparable to Arachne's load-balanced, cross-core thread creation latency. Shenango also does load balancing based on software RSS when the IOKernel steers packets to individual cores owned by an application. Thus, a measurement of the time from when the IOKernel first receives a packet to when a user thread starts executing would be more directly comparable to Arachne's load balanced cross-core thread creation. Alternately, a measurement of the time it takes from a thread creation to starting execution on a different core after work stealing would also be directly comparable. Neither of these measurements are available.

By placing threads on cores based on network packet core affinities and using per-core network queues to estimate core needs, Shenango's architecture enables network-based applications to achieve low latency and high throughput. However, this tight coupling limits its applicability to applications that do not use the network, and limits its flexibility with respect to thread placement, since it relies on software RSS to place threads when a request arrives. Shenango applications cannot do core-aware thread placement because the network dictates where the threads are placed.

8.2 User-level threading

Numerous user-level threading packages have been developed over the last several decades. Chapter 7 compared the performance of Arachne with Go [17] and uThreads [8]; this secton will describe several others as well.

8.2.1 M-to-N two-level schedulers

Recall from Chapter 3 that M-to-N scheduling is characterized by multiplexing multiple user threads over multiple kernel threads. This section discusses systems that offer this form of scheduling.

Go

Go [17] implements M-to-N scheduling [57] using three abstractions: Processors, OS Threads, and Goroutines. OS threads are equivalent to Arachne's kernel threads, and Processors correspond roughly to Arachne's cores, but they represent generic cores rather than specific cores. Goroutines are roughly equivalent to Arachne's threads; the Go runtime multiplexes them over OS Threads. Go's runtime limits each application's offered parallelism using the Processor abstraction: an OS Thread is only permitted to execute user space code in a Goroutine after it successfully acquires a Processor, which is an abstract resource with limited quanity. An operator can set the the number of Processors by assigning the environment variable `GOMAXPROCS`. If a Goroutine blocks in the kernel, the OS Thread executing it relinquishes its Processor. The Golang runtime documentation [19] implies that Go has a mechanism for detecting when OS threads block in system calls, and Austin Clements from the Go team at Google confirmed that the runtime uses a simple 10 ms timer to decide if a Goroutine has blocked in the syscall. With this scheme, the parallelism of Go applications must be statically assigned by an operator and its kernel threads are neither isolated from each other nor isolated from threads of other applications, so true parallelism may be lower than desired. In

contrast, Arachne uses one OS thread per core and allows applications to dynamically adjust their true parallelism based on load without static configuration.

The key contributions of Go are lightweight threading, explicit control over offered parallelism, and handling blocking IO. Arachne's runtime subsumes the first contribution by offering similar thread abstractions with lower latency. As explained in Section 2.3.2, control over offered parallelism is insufficient for achieving low latency and high throughput. Arachne's core arbiter gives applications a strict upgrade: explicit control over true parallelism. Arachne's current implementation does not handle blocking IO: it will temporarily lose an entire core if an Arachne thread blocks in the kernel. However, the architecture does not preclude adding a mechanism such as Go's for handling blocking IO. Arachne could feasibily be modified to use a different kernel thread to execute user threads on a particular core if the current one blocks, using the same mechanism as Go. In particular, an API call could be added to the core arbiter for an application to select a different kernel thread to execute on a core already owned by an application.

uThreads

uThreads [8] implements a hierachical M-to-N scheduling scheme using three abstractions: Clusters, kThreads, and uThreads. The kThread abstraction corresponds one-to-one with an operating system thread. uThreads are schedulable units equivalent to Arachne's threads. For clarity of exposition, this section will use the term *user thread* to refer to a uThread object and disambiguate between the name of the library and the name of the thread. The Cluster abstraction represents a set of kThreads. Application writers must specify a target Cluster when user threads are created; user threads can also be migrated explicitly between clusters using an API call. uThreads explores different types of queues for scheduling threads; it offers four different schedulers all based on some form of queuing. A compile-time flag is used to experiment with different implementations of queues with different locking and organizations.

- Global ReadyQueue per Cluster: When a new user thread is created, the scheduler enqueues it onto the unbounded intrusive ReadyQueue associated with a specific Cluster, protected by a kernel mutex and CV. Each kThread dequeues a batch of user threads when it runs out of work, to amortize the cost of global synchronization over multiple user threads.

- Local RunQueue per kThread using mutex and cv: The scheduler assigns new user threads to kThreads in a round-robin manner.

- Local RunQueue per kThread using lock-free non-intrusive Multiple-Producer-Single-Consumer Queue: Same as the above, but uses a lock-free queue (LFQ) instead of mutex and CV.
- Local RunQueue per kThread using lock-free intrusive Multiple-Producer-Single-Consumer Queue: Same as the above, but uses an intrusive LFQ.

uThread's Cluster abstraction offers superficially similar functionality to Arachne's core policies, but neither subsumes the other. The Cluster abstraction allows different groups of threads to be separated onto different groups of kernel threads; Arachne's core policies allow running different groups of threads on different sets of cores. The creator of uThreads recommends that operators manually pin kThreads to get Arachne-style core isolation. The Cluster abstraction is more general in that it allows distinct schedulers with entirely different data structures for representing scheduling information on each cluster, while Arachne's queueless representation of scheduling information is a key part of its design, and core policies cannot change it. Clusters are less general than core policies because the uThreads system does not allow migration of kThreads between Clusters, and nor does it allow a kThread to belong to more than one cluster simultaneously; this limits its ability to dynamically adapt to changes in threading needs.

uThreads suffers from the same limitation as Go with respect to Arachne: offered parallelism must be hand-specified by an operator, and lack of isolation means that true parallelism is not guaranteed. The creator of uThreads recommends pinning kThreads to specific cores to improve locality, but as discussed in Section 2.4.1, this approach cannot provide true parallelism in the presence of colocated applications.

uThreads addresses the problem of losing kernel threads when a user thread makes synchronous network calls into the kernel by offering a user-level blocking network API. This API allows user threads to block on network events at user level by making nonblocking network calls to the kernel and blocking the caller if the network is not ready. A single poller kThread polls the network devices and unblocks user threads when a related network event causes a user thread to be runnable. For example, suppose a user thread makes a `receive` network call on a socket and there are no packets available to receive. If the user thread made this network call by making a blocking system call, it would block inside the kernel and the uThreads runtime would lose the use of the kernel thread that was executing the user thread. If the user thread instead made the network call using uThreads' `receive` API, this call would get translated into an asychronous kernel call, whose return value would indicate the absence of packets. Upon seeing this return value, the runtime would block the user thread and the kernel thread could then run a different user thread. At some time in the future,

the poller kThread observes that the socket becomes readable, and marks the uThread as runnable.

Arachne does not addess the core loss problem when user threads block in the kernel, but its design is compatible with uThreads' technique of wrapping asynchronous kernel interfaces to provide synchronous user interfaces.

8.2.2 M-to-1 user-level threading systems

Recall from Chapter 3 that M-to-1 scheduling is characterized by multiplexing multiple user threads over a single kernel thread. Before multi-core hardware became widely available, several systems implemented M-to-1 user threading systems. Such systems do not enable parallel execution on multiple cores and make little sense on today's multi-core hardware, but they are discussed here because they provide historical context and present techniques for managing user threads that may generalize to M-to-N threading. Additionally, some implementations are compatible with running on multiple cores, but do not themselves provide mechanisms for actually managing multiple kernel threads, deferring them to the application.

Capriccio [56] implemented lightweight user threads with compiler support on a single kernel thread with a focus on making it feasible to use threading in Internet servers with the one-thread-per-connection programming style. At the time it was implemented, event-based systems were becoming prolific and the community believed that threads were less performant than events. The authors built Cappricio to offer performance parity with event-based systems and demonstrate that a good implementation of threads could be equally performant. Towards this end, they introduced techniques to make it feasible to support large numbers of threads and obtain the resource efficiency offered by event-based systems.

To address the problems of thread stacks consuming large amounts of address space, Cappricio introduces the notion of linked stack management, which allows non-contiguous stacks to grow and shrink at runtime. It uses compiler analysis to determine the maximum space each function's stack frame will consume. The same analysis is used to find cycles in the call graph and insert snippets of code that check whether the stack needs to grow. In today's 64-bit address machines, virtual address space is a less scarce resource, so Arachne forwent such complex stack management in favor of large, contiguous, pre-allocated stacks. Additionally, Arachne is not designed for huge numbers of threads.

To improve system efficiency, Cappricio introduces the idea of *resource-aware scheduling*, which involves generating a *blocking graph* (at runtime) based on the observed behavior of threads:

nodes represent points in execution where threads block, and edges represent periods of uninterrupted execution. The runtime then annotates the edges of the graph with information about thread behavior, such as average running time and changes in resource usage (memory, stack space, and sockets). When determining which thread to schedule, the runtime determines dynamically if each resource is near its limit and uses the annotated graph to prioritize threads which release such resources. Unlike Cappricio, the Arachne runtime only collects metrics about core usage and does not attempt to be clever about balancing other resources. It is unclear whether the overhead of Cappricio's sophisticated runtime analysis is compatible with Arachne's goals of low latency.

Like Go and uThreads, Cappricio offers a solution to the problem of blocking I/O causing the loss of a kernel thread. To avoid requiring changes in application code, Cappricio overrides the system call stub functions for blocking I/O in libc to block user threads by converting them to use asynchronous I/O mechanisms. Arachne could be modified to employee this technique but the current implementation does not.

Boost fibers [13] and Facebook's Folly [15] library implement lightweight user threads over a single kernel thread that are similar to Cappricio. Both are compatible with using multiple kernel threads, but the neither contains mechanisms for managing them or automatically distributing user threads across kernel threads; application writers must implement their own load balancing.

8.3 Automatic parallelization

Automatic parallelization frameworks, compilers, and libraries have the opposite goal of Arachne: they try to abstract away the physical machine and allow application writers to focus on algorithms instead of trying to make applications more aware of the physical hardware they are running.

8.3.1 Cilk

Cilk [12] is an extension of the C language designed to make it easy to write parallel programs without explicitly considering available cores. Cilk attempts to achieve theoretically optimal performance for large computations that can be split into tasks that can be run in parallel. For example, one might implement a recursive Fibonacci number calculator by spawning a new Cilk task for each recursive call. Cilk defines performance as the completion time for a large computation, rather than the completion time of individual tasks. In constrast, Arachne is designed around data center workloads in which the completion time of individual threads matters to end users. Another significant difference is that Cilk's thread model requires that threads are non-blocking and always run to

completion, while Arachne offers a fully general threading model in which threads can block and synchronize.

One implication of Cilk's non-blocking constraint is that parent threads cannot wait for the completion of their children. This complicates the Cilk threading model by forcing functions to be split up in all the places where they would logically join a child thread. Arachne's more general model allows parents to join their children as well as use any other synchronization patterns built on monitors or semaphores.

8.3.2 OpenMP

OpenMP [40] is a set of compiler directives (`#pragmas` in C/C++) directing the compiler to automatically parallelize particular sections of code. For example, one might prepend a loop with `#pragma omp parallel for` to instruct the compiler to automatically generate code to chop up different ranges of the loop and farm out the execution to different kernel threads. Like Go, OpenMP allows operators and developers to specify the amount of offered parallelism using an environment variable. Like Cilk, its target audience appears to be the speeding up of large computations using parallel computing, rather than handling large numbers of small computations with low latency. Unlike Arachne, it does not provide true parallelism to applications and also does not automatically scale offered parallelism.

8.3.3 Thread Building Blocks

Intel's Thread Building Blocks [54] is a C++ library which offers similar functionality to OpenMP. The library's API includes function calls like `parallel_for`. Thread Building Blocks additionally offers lower level synchronization primitives such as various types of mutexes and atomic operations, which makes it look closer to a standard threading library. However, Intel explicitly markets the ability to "specify logical parallelism instead of threads", which places it at the opposite end of the core-awareness spectrum from Arachne, which grants explicit control over cores.

8.4 Hardware scheduling optimizations

Carbon [26] aims to provide efficient hardware support for fine-grained parallelism. The authors point out that improved performance from a given granularity of parallelism depends on the cost of scheduling overheads compared to the length of tasks. For example, if one can schedule tasks in

a few hundred nanoseconds, it becomes feasible to run tasks that complete in a few microseconds. Carbon proposes the use of hardware queues that are closer to cores than the memory subsystem to cache tasks, and provides mechanisms for spilling tasks to memory and pulling tasks back from memory. The hardware queues are split into a centralized piece of hardware (GTU) which contains a queue per hardware thread, and a small hardware addition (LTU) to each core to prefetch tasks from the GTU and hide latency.

Since Carbon was evaluated on a simulator, it is difficult to directly compare performance with Arachne. However, it would be quite surprising if a specialized hardware implementation of queues failed to execute faster than Arachne, since it can bypass the memory system and eliminate cache miss times. However, Carbon requires changes to hardware and is limited to a fork-join model [7] of parallelism, while Arachne is a general-purpose thread management system without the need for specialized hardware.

8.5 Cache-optimized design

The notion of optimizing programs by improving cache behavior predated Arachne by over a decade [24]; CPU speeds have exceeded memory speeds for a substantial time. Cache optimization was originally introduced in the context of general-purpose software being written with access patterns and data structures that were cache-friendly, rather than as a communication bottleneck in shared-memory thread runtimes. However, it has recently become widespread in systems built on multiple threads. In addition to Arachne, several other recent systems [58, 41, 23, 49] are designed around limiting cache line transfers to achieve better performance.

8.6 Events as an alternative scheduling mechanism

Event-based applications such as Redis [46] and nginx [39] represent an alternative to user threads for achieving high throughput and low latency. Such models require application writers to separate their functions into non-blocking chunks, and manually save their state in between invocations [3]. In single core systems that do not offer true parallelism, event-based systems can avoid the use of expensive synchronization. However, applications that wish to take advantage of the parallelism on a multi-core system must still synchronize even if they use events. Behren et al. [56] 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.

Chapter 9

Conclusion

This dissertation described and evaluated Arachne, a system offering core aware thread management to provide a better combination of latency and throughput in data center applications. Fast networking and the increased use of memory have paved the way for applications to service requests in a few microseconds, but today's kernel-based threading mechanisms cannot support such services efficiently. Since kernel thread creation and wakeup take microseconds, threads must not block in applications requiring low latency. Consequently, they must busy-wait when waiting for the completion of operations taking microseconds, which reduces useful core utilization.

Arachne changes the interface between applications and the kernel scheduler from one based on threads to one based on cores. It implements several mechanisms to help applications achieve better performance based on this change.

- **Core Estimation**: Arachne introduces the idea that applications should determine how many cores they need, and implements a simple and effective heuristic to estimate the number of cores an application needs in order to keep latency low and utilization reasonable.

- **Core Policy**: Arachne proposes a simple API and a small set of support functions that allow applications to implement their own core estimation and determine their own thread placement.

- **Cache-Optimized Runtime**: While Arachne's APIs can support arbitrary thread runtimes on top of dedicated cores, its own cache-optimized runtime provides fast threading primitives that enable one-thread-per-request usage in many applications.

- **User Space Core Allocation**: By running entirely outside the kernel and requiring no kernel

modifications, Arachne enables applications using it to achieve low latency while coexisting with traditional applications.

Experiments show that integrating Arachne into applications can reduce tail latency and improve SLO-compliant throughput even in well-optimized applications. Furthermore, Arachne enables interference-free colocation of latency-sensitive applications and throughput-oriented applications, allowing data center servers to achieve low latency without wasting cycles.

Today's data centers frequently run applications in containers that are granted CPU cycles rather than specific cores. Such applications are subject to thread migration and kernel multiplexing, which makes their latency unpredictable and simple performance measurements difficult to reproduce. Moreover, important latency-sensitive workloads must be run on dedicated machines. I believe introducing Arachne into the data center in place of today's containers model can drive latency down and make performance experiments more reproducible. I am not alone in this view; Paul Turner recently stated that the Google kernel scheduler team's next major push is core-aware scheduling [55].

9.1 Future directions

The nature of research seems to be that interesting work tends to open up new research directions. While I hope that Arachne has made the need for efficient, high-performance applications to be core-aware clear, I believe Arachne represents a new beginning rather than an end to research on core-aware thread management. This field is still relatively unexplored, and there are several interesting and unexplored opportunities that fall directly out of Arachne's design and implementation.

9.1.1 Virtual machines

Arachne's core-aware approach to scheduling could be profitably applied to virtual machines. Today's hypervisors multiplex virtual cores over physical cores much like an operating system kernel multiplexes kernel threads over physical cores [52]. Moreover, physical cores are often oversubscribed by virtual cores. This means that application threads running inside a virtual machine can get descheduled in either of the following situations:

- The guest kernel deschedules the thread in favor of a different thread in the same virtual machine.
- The hypervisor deschedules the virtual core that the guest kernel had scheduled the applicatoin thread on.

Virtual machines could use a multi-level core-aware approach, where applications use Arachne to negotiate with their guest OS over cores, and the guest OSes use a similar approach to negotiate with the hypervisor. This would provide a more flexible and efficient way of managing cores than today's approaches, since the hypervisor would know how many cores each virtual machine needs.

9.1.2 Cluster scheduling

Today's datacenters are commonly orchestrated by container-oriented cluster schedulers such as Kubernetes [25], which schedule containers based on statically declared resource requirements. This approach to cluster scheduling is not responsive to workload changes and does not effectively isolate applications running on the same machine.

Core-aware scheduling can bring significant benefits to cluster schedulers for datacenter-scale applications. A cluster scheduler could collect information about core requirements from the core arbiters on each of the cluster machines and use this information to dynamically place applications and move services among machines. This would allow decisions to be made based on actual core needs rather than statically declared maximum requirements. Arachne's performance isolation would allow cluster schedulers to run background applications more aggressively without fear of impacting the response time of foreground applications.

9.1.3 Enhancements to Linux *cpusets*

Section 4.6 mentioned that the Linux cpusets mechanism can remove threads that belong to other user-level applications from managed cores, but they are unable to remove threads that belong to the kernel itself. Today, the only way to remove kernel-owned threads from a core is to take the core offline and then bring it back online again. If the Linux kernel is updated to enable cpusets to remove kernel-owned threads, it is likely that Arachne applications would be able to enjoy even lower tail latency, as well as the same latency at higher latency percentiles.

9.1.4 Arachne internals enhancements

There are a few aspects of Arachne that this dissertation has not fully explored. This dissertation describes only early efforts at implementing core policies, and our current core policy does not address issues related to NUMA machines, such as how to allocate cores in an application that spans multiple sockets. I hope that a variety of reusable core policies will be created, so that application developers can achieve high threading performance without having to write a custom policy for each

application. In addition, setting the parameters for core estimation is still more art than science. The current values were chosen based on a few experiments with benchmark applications. They provide a good trade-off between latency and utilization for our benchmarks, but it is unknown whether these will be the best values for all applications.

9.2 Lessons Learned

I have always believed and still believe there are two prototypical types of PhD students, and different students embody different linear combinations of them. The first type is output-oriented: their goal is to produce as much research as possible during their PhD using the resources provided by their advisor and university, and they will learn what they must to achieve this goal. The second type is input-oriented: their goal is to soak up as much knowledge as possible from their advisor and their PhD experience, and producing interesting research is a side-effect of that goal. I am the latter type, and so I believe it is important to discuss my experience in addition to my research.

The last seven years have been filled with wonder, joy, learning, illumination, euphoria, and the indulgement of intellectual curiosity. They have also been filled with missteps, wasted work, missed opportunities, and suboptimal choices. In this penultimate section of my dissertation, I share the insights and knowledge that would likely have made this journey even more glorious, joyful, and productive. These fall into two categories: wisdom directly gleaned from John, and insights I gathered from my own experience. Since I am not unique, perhaps some of my learnings will generalize enough to be useful to others who seek to follow this path.

9.2.1 Never trust a number produced by a computer

In early years of my PhD, I would bring performance measurements to John and he would scrutinize them for less than five minutes before finding myriad internal inconsistencies and anomolies that I could not explain. John would repeatedly tell me ”All young PhD students are too trusting“, and request and that I learn to be more suspicous of numbers and verify all numbers with deeper and differently designed experiments.

Over time, I learned to trust numbers less and less, but I do not know when I will achieve John’s level of distrust. If the system under test is not buggy, then the measurement code is buggy, or experiments are measuring the wrong numbers, or the compiler is reordering instructions, or statistics are being computed incorrectly. Because so many things can go wrong in an experiment, the probability of correct numbers appearing from an experiment in the first N attempts is vanishingly small. While

N may grow smaller as a researcher becomes more paranoid, it never seems to shrink below 3.

9.2.2 Always measure one level deeper

My very first research project with John was to answer the following question of the RAMCloud storage system [42]: "Where is the time going?" At the time, I lacked the knowledge to understand how to interpret the question, much less answer the question, but over the years it became a powerful way of thinking for me. Learning this lesson happened when I brought reasonable-looking performance measurements (after ironing out all the obvious bugs) to John and he made one of the following suggestions.

- Subdivide the time intervals until we could identify a single line of code or assembly that was taking a long time.
- Measure the same value using a different mechanism and see if the results lined up.
- Measure a different value that would be correlated with the given value in a correct implementation and see if it matches.

For example, at one point I counted cache misses with `rdpmc` and correlated the cost of the critical path with the number of cache misses on it.

This lesson superficially looks like the other side of the lesson on distrusting numbers, but is in reality much broader. Deeper measurements do not merely help us find errors and inconsistencies in our experiments; they can identify opportunities for further optimization and give us confidence in our designs.

9.2.3 Have a fast pipeline from data to graph-in-paper

I attribute this excellent advice to my labmate Collin Lee, who gleaned it from his early paper-writing experience. A naive and organic but inefficient way of generating graphs for papers might look something like the following:

1. Run experiment and produce data that is convenient for humans to consume.
2. Use a combination of miscellaneous scripts or manual editing to massage the data into a form that is easy to plot.
3. Write a script to turn that data into a plot.

4. Show the plot to advisor and discuss.

5. While either advisor or student finds problems, go back to Step 1. Otherwise, go to the next step.

6. Add the plot into the git repository for the paper.

This approach keeps the paper repository clean and allows papers written in LaTeX to build quickly. A problem arises when one discovers a need to rerun the experiment on newer hardware, or to verify reproducibility. Now there is a manual process of re-massaging the data and rerunning the plotting script, and re-copying the plot into the paper repository. Such a process is error-prone and can induce tremendous stress when submitting a paper under deadline pressure.

The right way to generate graphs and organize papers is to write experiments that generate data in an easy-to-parse format, and commit the data directly to the paper repository. The Makefile for the paper should perform any necessary data transformations and generate the plot from the data. Then, it will be possible to run an experiment, copy data, and immediately produce an updated paper.

9.2.4 Write one piece of code per graph

When I write software, I try to make it general purpose rather than hardcoding parameters; a good application can accept a variety of input formats and produce a variety of output formats based on options passed on the command line or in environmetal variables.

Naturally, I tried to apply this principle when writing the scripts used for generating graphs in papers. John has high standards for graphs, so I figured that having flexible plotting software would help me converge onto a reasonable looking graph faster, as well allow code reuse across graphs.

For example, I would have a script called `line_plot.py` which accepted command line arguments for all of the following parameters and more.

- Input format.
- Column names for use in legend.
- Colors for each line.
- Location of legend.
- Whether to use scale or linear scale on the X axis
- Whether to use scale or linear scale on the Y axis

- Whether to use gridlines or not
- If multiple data files are given, whether to plot them on the same graph or separate graphs.

After using this for a while, I realized that I had effectively moved the contents of my program, sans data I/O, from inside the plotting script to the Makefile containing the command line arguments that invoked the script. It turns out that programs that live almost entirely inside command line arguments are difficult to read, maintain, and version control.

Eventually, I switched to a one-script-per-graph model. Under this model, the parameters for a given graph are hardcoded directly into the script that produces it; only the path to the data is passed in on the command line. This decision resulted in simpler, more readable and maintainable plotting code, as well as readable command line invocations. Furthermore, it eliminated situations where changes to a plotting script to fix the appearance of one graph breaks other graphs generated from the same script.

9.2.5 Always output both vector and raster graphics

There are two broad ways of representing visual information in a computer. Raster graphics represent visual informmation as a two-dimensional array of pixels. Vector graphics represent visual information as a list of instructions for drawing lines and shapes. Most recently published academic papers use vector graphics for their architecture diagrams and graphs, because they can be rendered by software at arbitrarily high resolutions. Thus, it seemed natural for me to write plotting scripts that generating only vector graphics.

Unfortunately, there exists widely used collaborative presentation software [18] that does not support the importing of vector graphics. When creating presentations, I wasted nontrivial time finding the script that generated a particular graph in a paper and adding a line to output raster graphics. Consequently, I think all plotting scripts should just generate both types of images by default.

9.2.6 Making ideas extremely specific and concrete

When discussing system designs, algorithms, or theoretical performance, it often helps to make the discussions extremely specific and concrete. When iterating on the design of the Arachne runtime, John and I walked through a thread creation step by step and counted cache miss times at every step. We also examined very specific interleavings of threads that might cause problems for Arachne. Similarly, when Qian Li first described memcached's threading model to John, he asked her to

define exactly what each term meant, precisely what each thread was doing, and enumerate the steps for handling a request. I find that this process is useful for making my own thinking more precise, verifying my own understanding, and finding issues in designs before they are implemented.

9.2.7 Strong opinions, loosely held

One of the earliest and most important lessons I learned from working with John is that attachment to opinions in the face of conflicting evidence is a massive handicap to doing great work.

In my first year of working with John, we wanted to measure how much time various pieces of a RAMCloud read operation were taking and disagreed about the best way to do the measurements. I wanted to create an interface where intervals were first-class, named entities called Counters, which had a `startInterval / stopInterval` interface, because it was more closely aligned with my intuition of measuring intervals. John proposed a timestamp-based measurement scheme, in which timestamps were first-order entities and intervals were calculated during postprocessing by taking deltas. John pointed out the unnecessary verbosity in my design as well as its propensity for losing time, and I believed his arguments, but I held onto my prior belief that my design matched more closely with intuition and implemented it anyways.

My implementation became a RAMCloud module called `PerfCounters` which was almost never used again. John eventually implemented his design himself one summer. His implementation became a module called `TimeTrace`, which I eventually exported into `PerfUtils` and now use for almost all latency measurements.

I spent about 6 months implementing and polishing `PerfCounters`. Strong attachment to beliefs in the face of conflicting evidence is harmful.

9.2.8 Discipline in taking well-organized notes

Keeping well-organized notes that are easy to browse and search has been a significant challenge throughout my PhD career. I was surrounded by knowledge and there were many occasions to record information in written form:

- Research talks
- Advisor meetings
- Experiment setups
- Experiment data

- Design discussions
- Research ideas
- Feedback from talks I gave
- Course notes

I tried a variety of organizational strategies, such as having a file for each day, having a file per experiment, and one giant file for everything related to a project. Each approach had problems. The daily file made it difficult to view everything related to a given project at once, because it's difficult to extract unstructured notes from multiple files. A file per experiment made it hard to find the file for a given experiment without already knowing the file name and path; it also makes it difficult to logically link together the notes for related experiments. One giant (mostly unstructured) file for each project resolves the problem of finding related experiments, but makes it difficult to find the position in a file referring to a particular experiment.

Ultimately, the challenge of organizing notes in a useful way caused me to be undisciplined in how I managed them for most of my PhD career. Sometimes I would stick with one organization scheme for several months, but I never persisted because it was a lot of effort to remember the correct place in the correct file to add notes to. In such cases, I would relapse to doing the simplest short-term strategy: when I was having a meeting or receiving feedback, I would open a new file with an ad hoc name at an ad hoc path and write notes in it. A given note would thus only have a useful lifetime while I still remembered its full path; in practice this meant the notes would live only as long as I was actively working on the experiment or talk it was related to. After I forgot the path to a note, it would effectively be unreachable without a recursive grep through the home directories of all the machines I worked on. It did not help that I sometimes took notes on a server and sometimes on my laptop.

My lack of discipline in organizing notes caused all manner of frustration when I tried to find notes on a file from more than a year ago. Often it meant that experiments were difficult to reproduce because I had forgotten the exact configuration and commands used. Other times, it meant that interesting design ideas were lost and had to be re-discovered through discussion years later. John and I iterated on the design of each componeent of Arachne several times. At each iteration we enjoyed several interesting design discussions, producing useful insights that might generalize to other systems. An effective note taking discipline could have preserved much of the history of the design of Arachne, and this dissertation would have been more complete for it. Instead, those

insights live on only in the my intuition about systems, and this dissertation merely describes the present design of Arachne.

After six years of frustration with different organization schemes, I thought deeply about the problem and realized that the underlying organization problem was discoverability. That is, useful notes are structured in a way that made it easy to find notes related to a particular topic. An abstraction that satisfies this structure is a collection of *linked* text files. Then, one could create an index file for each project, which links to individual files for each experiment. Moreover, discussions concerning a given project or experiment can link to the file that describes the experiment. At the time, the closest approximation I could find to the desired linking functionality was Vim's `gf` command, which loads the filename under the cursor. However, its default behavior is somewhat limited and does not create new files on demand. It also did not solve the problem of individual files being unstructured.

One day, my labmate Behnam Montazeri observed me writing an outline for this dissertation and he recommended Vimwiki to me. Vimwiki is a Vim plugin that implements a wiki abstraction inside Vim. Unlike traditional web-based wikis which are linked HTML files, Vimwiki implements a collection of linked text files, which was precisely the abstraction I sought. In addition, it implements semantic headers, lists, tables, and syntax highlighting for both text and code, allowing text files to be well-structured and easy to navigate. With Vimwiki, it became easy to find the correct place to write notes, because the linked structure makes it easy to find files and the semantic structure makes it easy to find the right place in a file. The friction of placing notes in the right place decreased, and my notes became much more permanently useful.

I do not believe Vimwiki is the right tool for everyone. Some people prefer graphical tools such as Microsoft OneNote, Google Docs and Omnifocus; such tools can serve the same purpose as Vimwiki if they help reduce the friction of keeping organized notes. Vimwiki is the right tool for me because I am already a Vim user and fixed-width fonts and syntax highlighting for any programming language supported by Vim are important to me.

In retrospect, I could have saved myself significant pain and frustration by being disciplined enough to keep all my notes for a project in one directory on one server, even without the proper tool for organizing them perfectly. A little bit more thought earlier in my PhD career could have resulted in a rudimentary organization scheme that would be better than having no fixed scheme. For example, as a bare minimum, I could have had a directory containing a named and timestamped file for every meeting I ever had.

The discipline to keep everything in one place, even if the organization is not perfect, can save

time and reduce frustration in a PhD.

9.2.9 Separating planning from execution

My productivity varied substantially over the course of my PhD, and hindsight indicates that high productivity periods were characterized by having extremely clear and concrete goals each day. In the early years, those periods tended to be deadline-driven; an upcoming talk, poster presentation, or demo would offer a clear top-level goal, which I could then break down into smaller goals and execute on.

Eventually, I realized that the reason for increased productivity during deadline periods was not only the pressure of the deadline but also the behavior they induced: having a clear separation between a planning phase and an execution phase. That is, there are benefits to allocating a dedicated period of time to decide on the highest priority tasks to work on without actually working on them, and then allocating a different dedicated period of time to actually work on these tasks, without thinking about planning. Consequently, I was able to capture much of same productivity during non-deadline periods/latency by the simple act of deciding the tasks I would pursue each day on the night before.

9.2.10 "What" must always be tied to "why" in writing and speaking

One of the most important lessons I learned during my PhD is how challenging it is to clearly communicate technical ideas. A significant challenge in technical communication is learning to understand the difference between the *why* and the *what* and effectively communicating each.

In writing and speaking about systems or any other topic, the *what* is the presentation of facts: design decisions made, system architecture, implementation details, experiment designs, benchmark numbers, descriptions of related works and the like. The *why* is the presentation of concepts and intuition; it consists of the train of thought that led to the facts and explains how the what took on its current form. The why provides a mental framework for making sense of the facts, by sharing the mental context of the speaker or writer with the audience.

- Why was a particular design decision made?
- What problem is the system trying to solve?
- Why do each of the components of a system exist, and what role do they play in solving the overall problems?

- Why did we run experiment X? What did we hope to learn from it?
- Why is it reasonable to design experiment X thus?
- Why was the related work selected, and how does it relate to the current topic being discussed?

As a more concrete example, let us consider the following description of Arachne's core arbiter, taken from an early draft of the OSDI paper [45]:

> Requests from a
>
> *corearbiter*
>
> client to the
>
> *corearbiter*
>
> server are handled with Unix domain sockets. When the server needs to communciate with a client, it does so via a shared memory page specific to a given process. This shared memory page also provides basic statistics such as how many cores are currently unassigned and how many threads the process currently has waiting for cores. Client objects lazily establish a separate connection with the server per thread.

This is a wonderfully clear description of the communication mechanism between a core arbiter and a client application, but it does not address the questions of why we chose these particular communication mechanisms or why we did not use the same mechanism for the two directions of communication. Thus, the audience has no context for evaluating whether this design was good or bad or whether any thought went into the choice. In the revised version, we explicitly state the the reasons for the design:

> 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.

While a reader may disagree with our reasoning or discover a better way to solve the problems we aimed to solve, they are at least aware of how we arrived at the current design.

My earliest memories of John's feedback on both dry runs of presentations and drafts of papers are dominated by a single phrase: "It's all what and no why". Unfortunately, it took nearly six years for the reasoning behind this feedback to finally sink in. In the remainder of this section, I will share how I finally conceptualized the why behind this statement.

When I was working on my Bachelor's degree, I read a book called Made To Stick [21] to try to learn how to communicate better; this book introduced an idea called *the curse of knowledge*: A cognitive bias that makes it hard for people to explain ideas without unconsciously (and inadvertently) assuming some level of background knowledge on the part of the audience. This idea made sense to me intuitively, and it was only after connecting John's feedback to this idea that I finally understood it.

When we write and speak about technical ideas, the "why" associated with the ideas is embedded in the process of getting to the idea. This process is something that we experienced for ourselves, and so we know it quite deeply. Thus, it is particularly hard to realize that the "why" is not obvious to the audience of our talks and papers. And so, we omit the "why" without even considering that others might not know it, succumbing to the curse of knowledge.

The result is that our audience is left with some interesting ideas, but no context for how they came to be, and the intuition that gave rise to them. It is akin to reading code without documentation. One can tell what the code does, but not *why* the code does what it does. The "why" is really about the context; it captures the state of the creator's mind when they created an idea, and enables an audience to replicate that frame of mind when understanding an idea.

Thus, in order to communicate effectively, one must be vigilant about whether one has provided enough context for the ideas that one is communicating, and make an effort to always present the *why* along with *what*. Additionally, because it is impossible to fully escape the curse of knowledge, it is important to always collect feedback from others.

9.2.11 Low-level performance measurement is hard

I spent a great deal of time during my PhD measuring latency at the granularity of tens of nanoseconds, seeking explanations for the limitations of systems performance. On modern systems, measuring performance is extremely tricky. In addition to the issues mentioned in Section 9.2.1, modern hardware (especially the CPU cache subsystem) abstracts away functionality that has performance implications. For example, while it is possible to measure last level cache misses on each Intel core, there is little hardware support for precisely measuring the number of cache invalidations and the cross-core time cost of compare-and-swap in the context of other cache activity. Consequently, we can often only measure cycles spent on an instruction, and use published models of cache coherence protocols to reason about what might be happening in the hardware.

Since such reasoning is based on models and not measurement, it can be error-prone. Before my OSDI talk in October 2018, John and I believed that Arachne thread creation cost only 4 cache miss times, because we assumed that the cache coherence operation triggered by the compare-and-swap operation on the `MaskAndCount` variable could overlap with the cache miss to read the cache line containing the function and arguments. This was slightly inconsistent with our block-and-signal experimental results; we counted three cache miss times for signaling a blocked thread, but thread creation took more than four thirds as much time. In early 2019, I read a paper [51] describing the x86 memory model in great detail that was corroborated by the Intel manual. This memory model stated that the compare-and-swap instruction is fully serialized with other cache operations, so Arachne's thread creations actually took five cache misses rather than four. This is more consistent with the ratio of thread creation time to thread signal time, so we currently believe this is correct. However, the fact that we have no visiblity into the actual hardware behavior speaks to how difficult low-level performance measurement is, and how carefully it must be reasoned about.

9.3 Final Comments

9.3.1 Abstraction and performance opacity

To enable applications to make better scheduling decisions and achieve better performance, Arachne exposes information about physical cores to applications. This information was previously hidden behind the abstraction of kernel threads. I believe that the kernel threads abstraction is useful because it reduces the cognitive load on the application developer, but revealing the physical details hidden by the abstraction is necessary to achieve better performance.

More generally, abstractions are useful precisely because they hide information and complexity, but in doing so, they make it harder to reason about and improve performance. I speculate that there is a trade-off between the amount of information an abstraction hides and the ease of investigating performance. I believe there is value in considering this tradeoff when designing abstractions for high-performance systems.

Bibliography

[1] Green threads in the Java virtual machine. https://en.wikipedia.org/wiki/Green_threads#Green_threads_in_the_Java_virtual_machine. 18

[2] Intel Data Plane Development Kit. https://software.intel.com/en-us/networking/dpdk. 2, 70

[3] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative Task Management Without Manual Stack Management. In *USENIX Annual Technical Conference, General Track* (2002), pp. 289–302. 85

[4] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 53–79. 3, 16, 75, 76

[5] ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H. M. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers 38*, 12 (1989), 1631–1644. 77

[6] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), SIGMETRICS '12, pp. 53–64. 65

[7] BACCELLI, F., AND MAKOWSKI, A. *Simple computable bounds for the fork-join queue*. PhD thesis, INRIA, 1985. 85

[8] BARGHI, S. uThreads: Concurrent User Threads in C++. https://github.com/samanbarghi/uThreads. 3, 19, 59, 77, 79, 80

[9] BARROSO, L., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the Killer Microseconds. *Communications of the ACM 60*, 4 (2017), 48–54. 1

[10] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Oct. 2014), pp. 49–65. 70

[11] BITTMAN, D. MPSCQ - Multiple Producer, Single Consumer Wait-Free Queue. https://github.com/dbittman/waitfree-mpsc-queue. 59

[12] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1995), PPOPP '95, pp. 207–216. 30, 83

[13] Boost Fibers. http://www.boost.org/doc/libs/1_64_0/libs/fiber/doc/html/fiber/overview.html. 83

[14] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154. 70, 71

[15] Folly: Facebook Open-source Library. https://github.com/facebook/folly. 83

[16] FRIED, J. Private Communication, 2019. 70

[17] The Go Programming Language. https://golang.org/. 3, 19, 30, 77, 79

[18] Google Slides. https://slides.google.com/. 93

[19] Go runtime package. https://golang.org/pkg/runtime/. 79

[20] HANSEN, P. B. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975), 199–207. 75

[21] HEATH, C., AND HEATH, D. *Made to stick: Why some ideas survive and others die*. Random House, 2007. 99

[22] Java Fork/Join. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html. 30

[23] KAFFES, K., CHONG, T., HUMPHRIES, J. T., BELAY, A., MAZIÈRES, D., AND KOZYRAKIS, C. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 345–360. 85

[24] KOWARSCHIK, M., AND WEISS, C. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*. Springer, 2003, pp. 213–232. 85

[25] Production-Grade Container Orchestration. https://kubernetes.io/. 89

[26] KUMAR, S., HUGHES, C. J., AND NGUYEN, A. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), ISCA '07, pp. 162–173. 84

[27] LAMPSON, B. Experience with processes and monitors in Mesa. 75

[28] LEVERICH, J., AND KOZYRAKIS, C. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proc. Ninth European Conference on Computer Systems* (2014), EuroSys '14, pp. 4:1–4:14. 65

[29] LI, Q. memcached-A. https://github.com/PlatformLab/memcached-A. 61

[30] LORA, M. Xiph.org::Test media. https://media.xiph.org/. 67

[31] memcached: a Distributed Memory Object Caching System. http://www.memcached.org/. 8, 12, 43, 61

[32] Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark. 65

[33] Memtier SkewSyn. https://github.com/PlatformLab/memtier_skewsyn. 65

[34] MERRITT, L., AND VANAM, R. x264: A high performance H.264/AVC encoder. http://neuron2.net/library/avc/overview_x264_v8_5.pdf. 13, 66, 67

[35] MITTAL, R., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., ZATS, D., ET AL. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 537–550. 2

[36] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104. 35, 53

[37] MONTAZERI, B., LI, Y., ALIZADEH, M., AND OUSTERHOUT, J. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. *arXiv preprint arXiv:1803.09615* (2018). 1, 2, 70

[38] Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate. 65

[39] Nginx. https://nginx.org/en/. 85

[40] The OpenMP API specification for parallel programming. https://www.openmp.org/. 84

[41] OUSTERHOUT, A., FRIED, J., BEHRENS, J., BELAY, A., AND BALAKRISHNAN, H. Shenango: Achieving High {CPU} Efficiency for Latency-sensitive Datacenter Workloads. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 361–378. 49, 70, 74, 75, 85

[42] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 7. 5, 6, 8, 10, 12, 42, 43, 91

[43] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 7. 70

[44] QIN, H. Arachne. https://github.com/PlatformLab/Arachne. 58

[45] QIN, H., LI, Q., SPEISER, J., KRAFT, P., AND OUSTERHOUT, J. Arachne: Core-Aware Thread Management. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 145–160. 98

[46] Redis. http://redis.io. 85

[47] RHODEN, B. J. *Operating System Support for Parallel Processes*. PhD thesis, University of California, Berkeley, 2014. 75, 77

[48] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login: 39*, 6 (December 2014). 57

[49] ROGHANCHI, S., ERIKSSON, J., AND BASU, N. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 342–358. 85

[50] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)* (2014), pp. 1–16. 6

[51] SEWELL, P., SARKAR, S., OWENS, S., NARDELLI, F. Z., AND MYREEN, M. O. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM 53*, 7 (2010), 89–97. 100

[52] SIEBERT, E. Scheduling virtual CPUs. https://itknowledgeexchange.techtarget.com/virtualization-pro/scheduling-virtual-cpus/. 88

[53] STUEDI, P., TRIVEDI, A., AND METZLER, B. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *ATC* (2012). 70

[54] Intel Thread Building Blocks. https://software.intel.com/en-us/intel-tbb/. 84

[55] TURNER, P. Discussions around Google's fork of the Linux kernel. Private Communication. 88

[56] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: scalable threads for internet services. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, pp. 268–281. 3, 18, 77, 82, 85

[57] VYUKOV, D. Scalable Go Scheduler Design Doc. https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit?usp=sharing. 79

[58] YANG, S., PARK, S. J., AND OUSTERHOUT, J. NanoLog: a nanosecond scale logging system. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 335–350. 85