OS and Runtime Support for Efficiently Managing Cores in Parallel Applications

Kevin Klues

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-188
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-188.html

August 13, 2015

OS and Runtime Support for Efficiently Managing Cores in Parallel Applications

by

Kevin Alan Klues

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Eric Brewer, Chair
Professor Krste Asanović
Professor Brian Carver

Summer 2015

OS and Runtime Support for Efficiently Managing Cores in Parallel Applications

Abstract

OS and Runtime Support for Efficiently Managing Cores in Parallel Applications

by

Kevin Alan Klues

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric Brewer, Chair

Parallel applications can benefit from the ability to explicitly control their thread scheduling policies in user-space. However, modern operating systems lack the interfaces necessary to make this type of "user-level" scheduling efficient. The key component missing is the ability for applications to gain direct access to cores and keep control of those cores even when making I/O operations that traditionally block in the kernel. A number of former systems provided limited support for these capabilities, but they have been abandoned in modern systems, where all efforts are now focused exclusively on kernel-based scheduling approaches similar to Linux's. In many circles, the term "kernel" has actually become synonymous with Linux, and its Unix-based model of process and thread management is often assumed as the only possible choice.

In this work, we explore the OS and runtime support required to resurrect user-level threading as a viable mechanism for exploiting parallelism on modern systems. The idea is that applications request cores, not threads, from the underlying system and employ user-level scheduling techniques to multiplex their own set of threads on top of those cores. So long as an application has control of a core, it will have uninterrupted, dedicated access to it. This gives developers more control over what runs where (and when), so they can design their algorithms to take full advantage of the parallelism available to them. They express their parallelism via core requests and their concurrency via user-level threads.

We frame our work with a discussion of Akaros, a new, experimental operating system we developed, whose "Many-Core Process" (MCP) abstraction is built specifically with user-level scheduling in mind. The Akaros kernel provides low-level primitives for gaining direct access to cores, and a user-level library, called parlib, provides a framework for building custom threading packages that make use of those cores. From a parlib-based version of pthreads to a port of Lithe and the popular Go programming language, the combination of Akaros and parlib proves itself as a powerful medium to help bring user-level scheduling back from the dark ages.

Contents

List of Figures

List of Tables

Acknowledgments

This disseration has been a long time coming. It would not have been possible without the help of a multitude of people, including professors, colleagues, co-workers, and friends over the last 10+ years.

First, I'd like to thank professors Adam Wolisz and Vlado Handziski from the Technical University of Berlin for first introducing me to the world of academic research. If it weren't for my experience working with them so many years ago, I would have never considered doing a Ph.D at all. Next, I'd like to thank Dr. Chenyang Lu from Washington University in St. Louis for advising me through the process of completing my Master's thesis. It was under his guidance that I published my first few academic papers, without which I never would have been accepted to the Ph.D. program at Berkeley. Next I'd like to thank professor Phil Levis of Stanford University for giving me the opportunity to work with him for a year between completing my Masters degree and starting my Ph.D. I'm fairly certain his letter of recommendation played a central role in my acceptance to Berkeley, and for that I am forever grateful. Next I'd like to thank professors Eric Brewer, Krste Asanović, Ion Stoica, David Wessel, and Brian Carver for serving on my qualifying exam and dissertation committees. Unfortunately, David passed away before getting the chance to see the final result of this work. He will be sorely missed.

Last, but not least, I'd like to thank Barret Rhoden, David Zhu, Andrew Waterman, Ben Hindman, Steve Hofmyer, Eric Roman, Costin Iancu, Ron Minnich, Andrew Gallatin, Kevin Kissel, Keith Randall, Russ Cox and others at Berkeley, LBNL and Google, with whom I've had endless discussions over the years about this work. It simply wouldn't have come together as it has without you.

I cannot end without thanking my family (this includes you, Amy!), whose constant support both morally and financially I have relied on for most of my academic career. It is to them that I dedicate this work.

Chapter 1

Introduction

Cores are the lifeline of an application. Without them, applications would never be able to make forward progress. However, modern operating systems lack the interfaces necessary to allow applications to directly control how they make use of cores. Access to cores is hidden behind the abstraction of a thread, and operating system kernels dictate where and when an application's threads will be scheduled on the various cores available in the system. Hints can be given to help the kernel make these decisions, but limited support exists to ensure *true* isolation from other threads running in the system. We argue that parallel applications can benefit from the ability to gain direct access to cores and schedule their own "user level" threads on top of them. From the kernel's perspective, treating cores this way essentially transforms a difficult thread scheduling problem into an easier core allocation problem. But it leaves applications asking: how do I make efficient use of these cores once I have access to them? In this thesis, we take a closer look at this problem and present a full-stack solution for efficiently managing cores in parallel applications. We focus our efforts specifically on applications in the datacenter.

Our basic approach is simple – make cores an allocatable resource akin to memory, and build the necessary software infrastructure around this abstraction in order to make it useful to applications. The idea is that applications request cores, not threads, from the underlying system and employ user-level scheduling techniques to multiplex their own set of threads on top of those cores. So long as an application has control of a core, it will have uninterrupted, dedicated access to it (not even the kernel will interrupt it without prior agreement). This gives developers more control over what runs where (and when), so they can design their algorithms to take full advantage of the parallelism available to them. They express their parallelism via core requests and their concurrency via user-level threads.

In joint work with Barret Rhoden et al. [1], we introduced a new research OS, called Akaros, whose primary goal is to provide better support for parallel applications in the datacenter. As part of Akaros's design, we explicitly provide the interfaces necessary to make the core management capabilities described above possible. In this thesis, we leverage these core management capabilities and expand on the design and implementation of the user-level libraries that sit on top of them. The details of the Akaros kernel itself can be found in Barret's dissertation [2].

At the heart of these user-level libraries is **parlib** – a framework for building efficient parallel runtimes based on user-level scheduling. Parlib leverages Akaros's ability to explicitly allocate cores to an application and provides a set of higher-level abstractions that make working with these cores easier. These abstractions are the subject of Chapter 2 and provide the foundation for the rest of this thesis. In the chapters that follow, we expand on our use of parlib to provide a complete rewrite of Lithe [3, 4] (a framework for composing multiple user-level schedulers in a single application) and a port of the Go programming language [5] to Akaros. Moving forward we envision these libraries (and those like them) to play a central role in future datacenter applications. We elaborate on the significance of these contributions in Section 1.3 of this chapter.

In the following section, we provide some background and motivation for this work. Why are datacenters organized as they are today? Why is it important for datacenter applications to take direct control of their cores?. Why aren't existing core "allocation" strategies based on kernel thread scheduling good enough? The following discussion helps to answer some of these questions and more.

1.1 Background and Motivation

In the late 1990s systems research in parallel computing hit a major setback when people began using clusters of commodity machines to perform many of the tasks once reserved exclusively for supercomputers [6]. This "solution" to problems plaguing researchers for decades came not in the form of a truly technological breakthrough, but rather for economic reasons. Once commodity machines were both fast enough and cheap enough to perform most computing jobs with better economies of scale than supercomputers, it made sense for all but those in the HPC community to abandon supercomputers all together. Research in parallel computing continued, but a large fraction of researchers began to switch their focus to the field of distributed computing, where the problem was no longer "how do I exploit parallelism in a single machine with multiple CPUs?", but rather "how do I exploit parallelism distributed across multiple machines, each with a single CPU?". Ironically, economics, not technology, is now driving us back in the opposite direction, as these clusters of commodity machines (or datacenters as they are called today) are being built with an ever increasing number of CPUs and forcing us to reexamine many of the techniques used on parallel machines in the past. The world is no longer a cluster of commodity *single*-core machines, but rather a datacenter of commodity *multi*-core machines.

This shift towards multi-core CPUs, which began in 2004 and continues through today, was not a welcomed change. On the contrary, if it were up to chip manufacturers like Intel and AMD, everyone would still be running single-core machines and the fight for the best processor would be about pushing clock speeds to their maximum potential. The truth of the matter, however, is that continuing to increase clock speeds has a number of unwanted side effects that make it no longer economically feasible. These side effects include increased heat (making it expensive to cool), increased power consumption (making it expensive to

operate), and increased leakage current (making it expensive to design and build these chips in the first place) [7]. At a certain point it stops making sense to keep pushing anymore.

Industry's solution to this problem has been simple – replicate a large number of reasonably clocked CPUs on the same die and link them together over a high-speed interconnect. The promise of CPU scalability makes this approach seem reasonable on the surface [8]. However, it has one very important drawback: all code written for these machines must be explicitly parallel. In fact, legacy code written for single-threaded execution actually slows down in many cases because the individual CPUs they run on are now slower.

This major shift in processor technology has forced us to reexamine a fundamental question put on hold for over a decade: how do we write software to efficiently utilize these increasingly parallel machines? In other words, what do we do with all these cores? Although researchers in the HPC community have been looking at a variant of this problem for quite some time [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19], examining it in the context of the datacenter requires a fresh perspective because of the different constraints placed on the applications that exist in this environment. Unlike HPC workloads, which typically follow a run-to-completion model with no specific latency requirements (e.g. genome sequencing, weather simulations, etc.), datacenters are dominated by long-running interactive services that have a strict low-latency component (e.g. webservers, search engines, etc.). No one expects the results of a weather simulation to be completed without some minimal delay, but they do expect search results to be updated instantaneously as you type.

The interactive nature of these datacenter services places a variable demand on the number of resources required to satisfy their users over time (of which the most important resources are cores and memory). Their usage patterns are characterized by long periods of low-to-average demand interspersed with bursts of high demand (e.g. everyone searching at once when a high-profile natural disaster occurs). This is in contrast to HPC-style workloads that typically require a constant (or at least predictable) number of resources from start to finish. To compensate for these bursts, datacenters must be over-provisioned to support the peak resource demands of all services they run simultaneously. That means more machines and more cost.

Unchecked, a large portion of these resources would simply sit idle a majority of the time. However, datacenters make up for this excess capacity by doing some amount of batch processing in the background. These batch jobs follow the standard HPC-style run-to-completion model and typically consist of either MapReduce jobs [20] or some type of "big data" analytics. They simply use the resources not required by the primary interactive services when running below peak demand. The end result is that datacenters require a much more fluid transfer of resources between their various jobs compared to traditional HPC workloads.

However, individual nodes in the datacenter are not in charge of allocating resources to applications themselves. Instead, all allocation decisions are made by a higher-level control system or "cluster OS" (e.g. Google Borg or Mesos) [21, 22], whose job it is to examine the state of the entire datacenter, figure out the best allocation of resources to applications, and spread those applications across tens-of-thousands of nodes. Individual nodes simply enforce

the allocations that have been imposed on them by the cluster OS. In practice, interactive jobs get more resources and batch jobs get fewer.

Under this model, existing cluster OSs follow strict allocation policies when handing out resources to applications. They continuously monitor the resource usage of all applications and allocate resources to them based on relative priorities and estimated future demand. Every time a new allocation decision needs to be made, a global reshuffling of resources takes place and nodes adjust themselves accordingly (usually by killing whatever batch jobs are running and handing their resources over to the interactive services that need them). Because this reshuffling procedure tends to be fairly heavy-weight, cluster OSs try to avoid making these allocation decisions very often. At one extreme, the cluster OS triggers a global reshuffling of resources every time there is a minimal change in resource demand. At the other extreme, each application is allocated all of the resources required to meet its peak demand up front, and most resources sit idle a majority of the time.

In practice, cluster OSs tend to follow a policy that falls somewhere between these two extremes. The overhead of a global reshuffling algorithm is just too high. However, this is in conflict with the "fluid" transfer of resources required to keep all nodes in the datacenter busy at all times (only a "choppy" transfer of resources is possible at best). The only way to achieve true fluidity of resources would be to modify the global reshuffling algorithm to make some number of *node-local* decisions as well. We've come up with such a scheme based on the provisiong interfaces we've built into Akaros (which are discussed further in Section 1.2.2).

The idea is to have the cluster OS no longer perform resource allocation directly. Instead, it would make global decisions about how to *provision* resources for guaranteed future use by an application, but leave the real-time allocation of those resources up to individual nodes themselves. Global decisions would be based on provisioning resources to an application for peak-demand; node-local decisions would be based on using these provisions to allocate resources to an application given the real-time demand. In practice, all resources in the datacenter would be provisioned exclusively to interactive jobs, and all remaining batch jobs would simply pick up the slack when real-time demand was low.

As a global policy, provisioning resources this way is much lighter-weight than allocating them directly. In the common case, changing an application's resource provision amounts to little more than changing a number in a resource vector and does not immediately affect its real-time allocation. This would be a major win over current algorithms, where even minor allocation adjustments force a node to immediately start killing jobs and moving resources around as instructed.

However, individual nodes in the datacenter currently lack the ability to enforce this more sophisticated policy. They tend to run a relatively traditional OS, and any code running on them is limited to the abstractions provided by that OS. For example, the desire for a cluster OS to "allocate" cores to an application is misaligned with the capabilities that actually exist on a node to enforce this allocation (what does it even mean to allocate a core in a traditional OS?). Moreover, there is no way for a cluster OS to actually set up a provision of resources so that node-local decisions could take advantage of it (traditional OSs simply lack the interfaces to support this).

Luckily, these problems are not inherent to datacenter nodes in general, but rather an artifact of history. As fate would have it, the original push towards single-core clusters in the late 1990's coincided with the growth in popularity of Linux, and over time Linux has become the de facto standard OS for nodes in the datacenter. However, Linux was originally designed specifically with uniprocessor systems in mind, and any multi-core support that has been added over the years still contains remnants of this original design. The kernel itself has been modified to scale past previous bottlenecks, but its design is still rooted in a model that conflates the notions of cores and threads into a single unified abstraction.

In order to "allocate" cores to an application, Linux relies on kernel scheduling techniques to try and isolate an application's threads to run on a specific subset of cores in the system. "Allocation" is therefore achieved when one application's threads are all scheduled on one core and another application's threads are all scheduled on another. However, there is no way to truly guarantee isolation between threads or prevent the kernel from interrupting an application at an inopportune time if it has some periodic task it would like to run. Even with the use of CPU sets, which purport to provide shielding from other processes and interrupts, Linux gives no guarantees that it will not periodically run one of its own tasks on a shielded core. In the context of the datacenter, this means that an interactive job may end up being temporarily descheduled without prior knowledge, degrading its performance and damaging the overall end-user experience.

By explicitly allocating cores to an application and providing guarantees for true dedicated access, this problem can mostly be avoided. Indeed, we ran a variant of the FTQ benchmark [23] for measuring OS 'noise' on Linux vs. Akaros, and found Akaros to have *an order of magnitude* less noise and fewer periodic signals on its cores than Linux [2]. Moreover, Rasumssen et al. [24] have demonstrated that, in general, parallel applications running on Linux tend to run far below their peak performance capabilities (partially due to OS noise and partially due to an inability to directly manage memory at the application level). By removing Linux from the picture, they were able to fine-tune a set of machines to execute a custom sorting algorithm called TritonSort, 60% faster than the previous winner of the Indy GraySort sorting benchmark – all with only 1/6 the number of nodes required by Linux.

Given these results, one has to ask themselves: why are we putting our large-scale, performance sensitive datacenter applications in the hands of an OS that was never designed to run these types of applications in the first place? Solutions have been bolted on over time to deal with some of the problems faced by these applications (CPU sets being a prime example), but any solution that does not directly align the desires of the cluster OS with the capabilities of the OS running on each node will ultimately fall short. Akaros has been designed specifically with these concerns in mind.

1.2 Akaros

Over the past 7 years we have been building an experimental operating system called Akaros [1], whose goals are directly in line with providing better support for parallel ap-

plications in the datacenter. Akaros itself was born out of the desire to rethink operating systems for large scale shared-memory multi-core machines, with a specific focus on (1) providing better support for parallel applications for both high-performance and general-purpose computing and (2) scaling the kernel to thousands of cores.

Although not initially intended specifically for datacenters, Akaros's design matches well with both datacenter applications as well as datacenter hardware. These applications require both high, predictable performance as well as low latency, and they are important enough to warrant engineering effort at all levels of the software stack. With Akaros we ask: what improvements can we make to traditional operating system abstractions to better support these types of applications?

The general philosophy behind Akaros is to:

Expose as much information about the underlying system to an application as possible and provide them with the tools necessary to take advantage of this information to run more efficiently.

Following this philosophy, significant work has gone into designing and building all of the abstractions that distinguish Akaros from a traditional OS. During its development we have been very careful not to simply build things that we eventually plan to throw away. Ultimately, we would like the actual code base of Akaros adopted as a useful datacenter operating system rather than simply viewing its implementations as an academic exercise. To this end, we have designed Akaros in such a way that it can be incrementally deployed in a datacenter, rather than forcing users to reinstall it on every one of their nodes. This means that jobs that have not yet been ported to take advantage of Akaros's features can continue to run on legacy OSs throughout the datacenter without any ill effects.

Akaros currently runs on x86-64 and RISC-V[25], an experimental multi-core architecture under development at UC Berkeley. We have ported GNU libc to Akaros, support dynamically and statically linked ELF executables, and provide thread-local storage (TLS). We have basic in-memory filesystem support (initrd and ext2) and a simple networking stack. The OS feels like Linux and runs busybox and pthread programs.

However, Akaros differs from traditional systems in several ways, most importantly in its treatment of parallel processes and the way they manage cores. In Akaros, we have augmented the traditional process with a new abstraction called the many-core process (MCP). The kernel gives cores (rather than threads) to an MCP, allowing applications to schedule their own user-level threads on top of those cores. Coupled with Akaros's asynchronous system call interface, MCPs never lose their cores, even when issuing a system call that traditionally blocks in the kernel (e.g. read, write, accept). These cores are also gang-scheduled [26] and appear to the process to be a virtual multiprocessor, similar to scheduler activations [27].

In addition to handing out cores to an MCP, Akaros also takes advantage of having a large number of cores by specializing a few of them for certain tasks. The main distinction we

make is between cores given to MCPs and cores used for everything else. These cores differ in both their workloads and in how long they spend on a specific task. Thus we denote cores as either coarse-grained (CG) for cores given to an MCP, or low-latency (LL), in reference to their short scheduling quanta. LL cores tend to run interrupt handlers, kernel background work, daemons, and other single-core processes. CG cores, on the other hand, are relatively simple, and are designed to provide an MCP with computation free from interference. These cores never run kernel tasks, nor do they receive any undesired interrupts or other sources of jitter. Ultimately, applications have control over their CG cores until they are either voluntarily yielded back to the system or explicitly revoked by the kernel because a higher priority application needs them. In practice, physical core 0 is the only core we currently designate as an LL core, and the kernel schedules tasks on it with a time quantum of 5ms (vs. no scheduling quantum for CG cores).

In this thesis, we focus specifically on CG cores and explore techniques for managing them efficiently in parallel applications. In the following sections, we expand on the ideas behind the MCP as well as provide an overview of our interfaces for resource provisioning and allocation on Akaros. Provisioning provides the foundation for true dedicated access to cores.

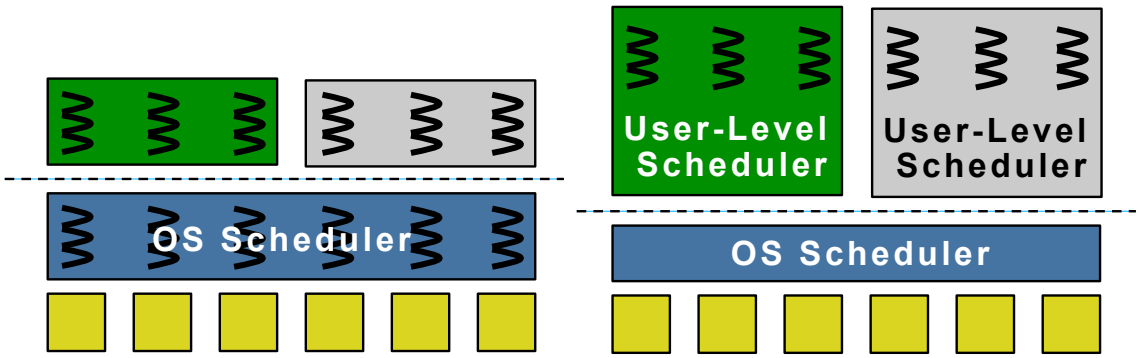
1.2.1 The Many-Core Process

In Akaros, we have augmented the traditional process model with a new abstraction called the many-core process (MCP). Similar to traditional processes, MCPs serve as the executing instance of a program, but differ in a few important ways:

- The kernel gives cores (rather than threads) to an MCP, allowing applications to multiplex their own user threads on top of these cores.
- All cores are gang-scheduled [26], and appear to the MCP as a virtual multiprocessor, similar to scheduler activations [27].
- There are no kernel tasks or threads underlying each thread of a process, unlike in a 1:1 or M:N threading model.
- All system calls in Akaros are asynchronous, so MCPs never lose their cores due to a system call, even if that system call blocks in the underlying kernel.
- An MCP is always aware of which cores are running, and will not be preempted or have any of its cores revoked without first being notified. Privisioning interfaces exists to also prevent revokation when desired.
- All faults are redirected to and handled by an MCP rather than by the kernel. Typically this is done via user-level library code (similar to Nemesis [28, 29]). We discuss this further in the following chapter on parlib.

Figure 1.1 shows a side by side comparison of the high-level differences between a traditional process and an MCP. In a traditional process, each user thread is launched with a backing kernel thread, and the kernel is in charge of all thread scheduling decisions. In an MCP, user

threads are both launched and scheduled by user space according to a two-level scheduling hierarchy; they are multiplexed on top of cores granted to them by the underlying kernel. The kernel then schedules MCPs as a single unit, with all cores granted to an MCP gang scheduled together.



(a) Traditional 1:1 Process (b) The MCP

Figure 1.1: A high-level comparison between a traditional 1:1 process and an MCP

By design, MCPs have good performance isolation. They take the set of resources allocated to them and run in isolation from other MCPs in the system. This isolation allows for better cache utilization and faster synchronization between cores compared to traditional processes. Moreover, since MCPs rely completely on asynchronous system calls and user-level threading, they tend to run faster than traditional processes (due to smaller context switch times) as well as scale better (due to the low overhead per thread). There is no need to oversubscribe threads to cores, and programmers are free to run custom scheduling policies based on the particular needs of their application. They can implement policies such as "one thread for every I/O", without worrying that these threads might thrash in the kernel. Support also exists to ensure that lock holders always run first (even if preempted), which is critical for the tail latency of operations involving non wait-free synchronization. These capabilities, and the implications they have on scheduling, are discussed further in Chapter 2.

1.2.2 Resource Provisioning and Allocation

Traditionally, resource allocation is done for one of two reasons:

- As a "reservation" mechanism, whereby resources are set aside for future use by an application
- As a "real-time assignment" mechanism, whereby resources are requested by an application just before they are actually needed

When used as a reservation mechanism, applications are always prepared with enough resources to satisfy their peak demand, but they may end up leaving those resources unused

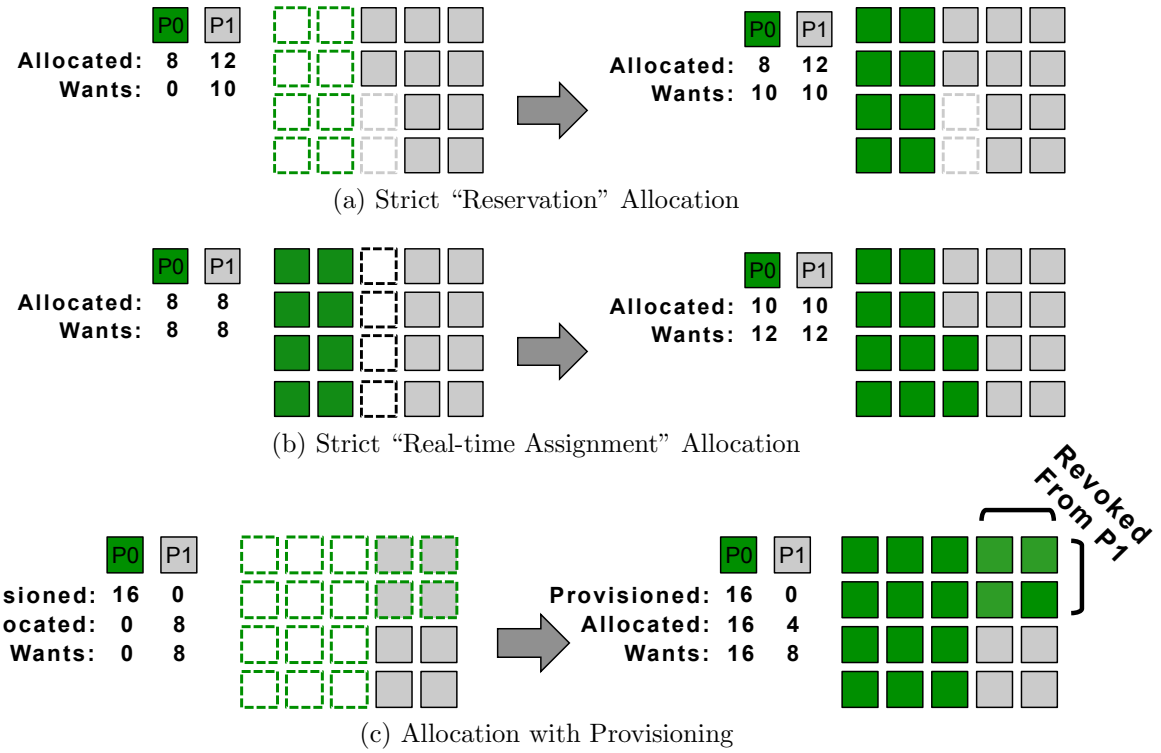
a large percentage of time if real-time demand for them is low. On the other hand, when used as a real-time assignment mechanism, applications never have to worry about leaving resources unused, but they run the risk of not getting a resource when needed because it is already allocated to another application. Examples of these two situations in the context of allocating cores are depicted in Figures 1.2a and 1.2b on the following page.

Akaros's provisioning interface decouples these two ideas. It provides separate mechanisms to first reserve a resource for guaranteed future use, and then (in a separate step) allocate that resource just before it is actually needed. In the meantime, provisioned resources can be used by other applications with the implicit contract that they may be revoked at anytime. Once revoked, they are given immediately to the application that provisioned them. In the context of cores, provisioning provides the means for true dedicated access to cores, with no opportunity for interruption from the underlying system (barring any minimal delay to revoke the core from another process at the outset). Figure 1.2c shows how provisioning cores in this way compares to the two pure allocation policies discussed above.

The primary advantage of this interface is that it allows applications to reserve resources to meet some minimal latency requirement, while still allowing those resources to be allocated to other processes when not currently in use. Moreover, this interface allows applications to allocate *more* than their provisioned resources (when available) in order to compensate for unforseen spikes in activity. When going over a provision in this way, the application is able to continue running at peak performance while the system adjusts the necessary provisions in the background. These capabilities are particularly important in the context of the datacenter to enable fluid sharing of resources between interactive jobs and batch jobs, as dicussed in Section 1.1 of this chapter. For the purposes of this thesis, however, we are interested exclusively in the ability of this interface to provide true dedicated access to cores. We leave it as future work to exploit these capabilities in a modified resource allocation policy for a cluster OS.

1.3 Contributions and Roadmap

With Akaros in place, we can start to see how applications could take advantage of the MCP and its provisioning interfaces to improve the overall performance of applications in the datacenter. Indeed, Rhoden provides a number of experiments in his dissertation [2] showing compelling evidence of this claim. From demonstrating that Akaros can provide *an order of magnitude* less noise on its cores than Linux to showing Akaros's provisioning interfaces in action, these experiments suggest that (at the very least) we are on the right track. However, many of these experiments are based on micro-benchmarks or contrived examples that are meant to demonstrate a particular kernel feature in isolation. There is still much work to be done before Akaros will be able to compete with Linux in terms of running any real-world datacenter applications. These applications often go through years of fine-tuning, and Akaros simply isn't mature enough to exploit the multitude of Linux-specific features that have been added to them over the years (nor is that necessarily desirable). Many times these



(a) Strict "Reservation" Allocation

(b) Strict "Real-time Assignment" Allocation

(c) Allocation with Provisioning

Figure 1.2: Three different core allocation schemes, based on (a) allocating cores strictly for future use, (b) allocating cores based solely on real-time demand, and (c) provisioning cores for future use, but allowing others to make use of them during periods of low demand. P0 and P1 indicate different process trying to allocate cores over time. Time flows from left to right, with the allocation state changing from the state on the left to the state on the right as time moves forward.

In (a), P0 starts off with 8 allocated cores with a real-time demand of 0, and P1 starts off with 12 allocated cores and a real-time demand of 10. As time moves forward, P0's real-time demand increases to 10, but only 8 of these cores can be satisifed by its current allocation. The remaining 2 cores are allocated to P1, and there is no chance for P0 to make use of them even though they are just sitting idle. They must remain idle because of the strict "reservation" scheme in use.

In (b), both P0 and P1 start off with 10 cores each, but their real-time demand increases to 12 as time moves forward. Since a real-time allocation scheme is in use, neither P0 nor P1 can meet their new real-time demand; both processes end up falling short by 2 cores.

In (c), P0 provisions 16 cores for future use, but starts off with a real-time demand of 0. P1, on the other hand, has 0 cores provisioned, but is able to meet its real-time demand of 8 by "borrowing" 4 of its cores from P0's provision. As time moves forward, P0's real-time demand increases, causing it to allocate all 16 of its cores at once. When this occurs, 4 cores are preempted from P1, and its allocated core count is reduced to 4 instead of 8. By provisioning cores in this way, P0 takes priority in allocating the cores it has provisioned, but allows P1 to use them during periods of low demand.

features are put in place to work around some fundamental mismatch between the desires of the application and the fact that Linux conflates the notion of threads and cores into a single abstraction (e.g. the requirement for epoll in many high-performance networking applications). On Akaros, with its ability to support lightweight user-level threading, many of these issues dissappear completely. The overall vision for Akaros is in place and bit by bit we are starting to fill in the individual pieces to make it fast.

In this disseration, we take a closer look at one of these pieces and explore what it takes to efficiently manage cores inside an application. The Akaros kernel provides the API necessary to gain direct access to cores. The work presented here provides the software infrastructure necessary to manage these cores efficiently in user-space.

The rest of this work is organized as follows:

Chapter 2: Parlib
Parlib is a user-level library we developed that helps applications exploit the core management capabilities provided by Akaros. It is the user-space counterpart to the Akaros kernel. Akaros provides the low-level primitives for gaining direct access to cores, and parlib provides a framework for building custom user-level schedulers that make use of those cores. Parlib itself does not define an explicit user-level scheduler, but rather provides a set of abstractions that ease the development of user-level schedulers built on top of it. In this chapter, we discuss each of these abstractions in detail and present our canonical user-level pthreads implementation built on top of these abstractions.

As part of our development, we have also built a port of parlib for Linux to act as a compatibility layer to compare user-level schedulers running on both systems. The Linux port is limited in its support for *true* dedicated access to cores, but is useful for providing a performance comparison between applications built with parlib-based schedulers and those running directly on the Linux NPTL. We provide this comparison and show that parlib-based schedulers are able to outperform the Linux NPTL across a number of representative application workloads. These results include a 3.4x faster thread context switch time, competitive performance for the NAS parallel benchmark suite, and a 6% increase in throughput over nginx for a simple thread-based webserver we wrote. Collectively, these benchmarks provide substantial evidence in favor of our approach to user-level scheduling – on both Akaros and Linux, alike.

Chapter 3: Lithe
With parlib in place, higher-level libraries can be developed that expand on its basic capabilities. One such capability is the ability to use multiple user-level schedulers inside a single application. Such functionality is useful, for example, when leveraging code developed by third-party developers from different frameworks or languages (e.g. OpenMP, TBB, or Go). Where parlib provides abstractions to build individual user-level schedulers, Lithe provides abstractions that allow multiple user-level schedulers to interoperate in harmony. Lithe was originally proposed by Pan et al. in 2010 [3, 4] and all work presented here is an extension of this previous work.

Although not originally proposed as such, Lithe can be thought of as a natural extension to parlib, with the added capability of coordinating access to cores among multiple user-level schedulers. In this chapter, we present our port of Lithe to parlib and discuss some results based on this new design. Most notably, we add support for asynchronous syscalls (which we get for free from parlib), as well provide a new generic "fork-join" scheduler on top of which all of our Lithe-aware schedulers are now based. We also provide a general cleanup of all of Lithe's outward facing interfaces.

In terms of results, we include a re-run of the SPQR benchmark [30] found in the original Lithe paper [3], as well as an extention to the Flickr-Like Image Processing Server presented there as well. Our results corroborate previous results, and show new evidence that supports Lithe's ability to efficiently manage cores among multiple user-level schedulers.

Chapter 4: Go on Akaros
Go [5] is a modern programming language from Google that lends itself well to the development of parallel applications in the datacenter. It has been steadily rising in popularity in recent years in part due to its elegant code composition model and in part due its concurrency model [31]. Go's concurrency model is based on launching light-weight user-level threads, called go routines, and synchronizing access to them through a language primitive known as a "channel". This concurrency model matches well with the user-level threading model promoted by Akaros, Parlib, and Lithe.

In this chapter, we present our port of Go to Akaros. The goal of the port is two-fold. First, it demonstrates the completeness of the Akaros kernel in terms of supporting a fully functioning runtime environment that matches well to its underlying threading model. Second, it opens up a new set of applications to Akaros that otherwise wouldn't be available.

Additionally, Go includes an extensive test suite that tests a wide-array of functionality from spawning subprocesses to launching http servers and handling file descriptors properly in the OS. By ensuring that Akaros can pass this test suite, we provide a fairly complete regression of many of Akaros's core features. At the time of this writing, Akaros passes 100% of the 2,551 required tests from this test suite (though it still skips 151 unrequired tests). Moving forward we plan to add support for these remaining tests and ultimately integrate our port with the official upstream Go repository.

Chapter 5: Related Work
Our work on Akaros and its related user-level libraries is grounded in a large body of previous work. In this chapter, we review this work and compare it to the approaches we've adopted in Akaros. The chapter is broken into multiple sections including: Threading Models, Concurrency vs. Parallelism, and Contemporary Systems. In each section, we begin with a discussion of previous work, followed by how our work relates to it.

Chapter 6: Conclusion
In this final chapter, we provide a summary of everything discussed in this disseration and provide some insight into the future directions of this work.

Chapter 2

Parlib

Parlib is a parallel runtime development framework. It was originally developed in conjunction with Akaros to provide useful abstractions for building user-level schedulers to run inside an MCP. It has since become the dumping ground for any abstraction that eases parallel runtime development. Ports exist for both Akaros and Linux. It is written in a combination of assembly and C.

The primary goal of parlib is to resurrect **user-level threading** as a viable mechanism for exploiting parallelism in modern-day systems. Parlib itself consists of a set of abstractions which developers can use to ease the development of new parallel runtimes built around the notion of user-level scheduling. Moreover, existing parallel runtimes such as OpenMP, TBB, and even the Linux NPTL can benefit from the abstractions provided by parlib.

The idea of user-level threading itself is not new. Indeed, systems such as Scheduler Activations [27] and Capriccio [32] make heavy use of user-level threads and have shown them to perform quite well in practice. However, both of these systems have inherent limitations that have kept them from wide-spread adoption in the past. For example, scheduler activations introduces complexities into both user-space and the kernel with only minimal performance benefits on non-compute-bound workloads. These complexities make the overhead of coordinating threads across the user/kernel boundary prohibitively expensive whenever a blocking system call is made. Many believe this overhead (and the complexities it brings with it) outweigh any benefits gained by being able to control the thread scheduling policy in user-space [33]. Moreover, scheduler activations has been found to scale poorly to a large number of cores. NetBSD actually introduced scheduler activations as their supported threading model in 2003 [34], only to remove it in 2009 in favor of a 1:1 model that scaled better [35]. Likewise, Linux had a set of patches for scheduler activations that were ultimately abandoned as well [36].

Capriccio improves upon scheduler activations by reducing the costs associated with blocking system calls [32]. Instead of allowing user threads to block in the kernel, Capriccio relies on asynchronous I/O to return control to user-space without the overhead of starting a heavy-weight activation. Capriccio simply intercepts each blocking call and transforms it into its asynchronous counterpart. Threads are then "blocked" in user-space, giving the

application the opportunity to schedule another thread in its place. Although this threading model leaves lots of room for improved performance, Capriccio also suffered from scalability problems, as it was only designed to work on a single core. Additionally, it only supported cooperative thread scheduling policies, and had no good mechanisms for directing POSIX signals to individual user threads. It also provided no explicit API for developers to customize their thread scheduling policies on an application specific basis. Any custom schedulers would have to be built into Capriccio itself, limiting its outward facing API to the pthreads interface.

In many ways, parlib can be thought of as the multi-core extension to Capriccio, with a number of added capabilities. Just like Capriccio, parlib makes heavy use of user-level threads and relies on asynchronous system calls to decouple the threading models in use by user-space and the kernel. However, parlib itself does not define an explicit user-level scheduler. Instead, it provides a set of abstractions which ease the development of user-level schedulers built on top of it. As an improvement over Capriccio, these abstractions provide support for both cooperative and preemptive scheduling policies, as well as support for directing POSIX signals at individual user threads.

Using these abstractions, developers can build custom user-level schedulers that expose any outward facing API they wish to present to an application. For example, we have used parlib to build a user-level variant of pthreads that is API compatible with the Linux NPTL (you still need to recompile). Additionally, we have built a webserver that includes an application-specific thread scheduler that operates independent of any underlying threading package. Both of these schedulers are discussed in more detail in Section 2.6 alongside a set of experiments that evaluate how well they perform compared to the Linux NPTL.

One key thing that distinguishes parlib from its predecessors is its reliance on the ability to request "cores" from the underlying system instead of threads. This capability, combined with a unified asynchronous event delivery mechanism, provides better core-isolation as well as improved performance across a number different metrics on representative application workloads. Moreover, recent advancements at the hardware level make parlib's threading model more feasible for adoption than it has been in the past. Specifically, the addition of the `rdfsbase` and `wrfsbase` instructions on recent Intel platforms allows for fast user-level context switching with full support for thread-local storage (TLS). We explore the implications of having these capabilities further in Section 2.6.

2.1 Abstraction Overview

Parlib itself is *not* a parallel runtime. Instead, it is a collection of abstractions that developers can use to build their own parallel runtimes. Specifically, these abstractions ease the development of custom user-level thread schedulers and provide the necessary support to make user-level scheduling efficient.

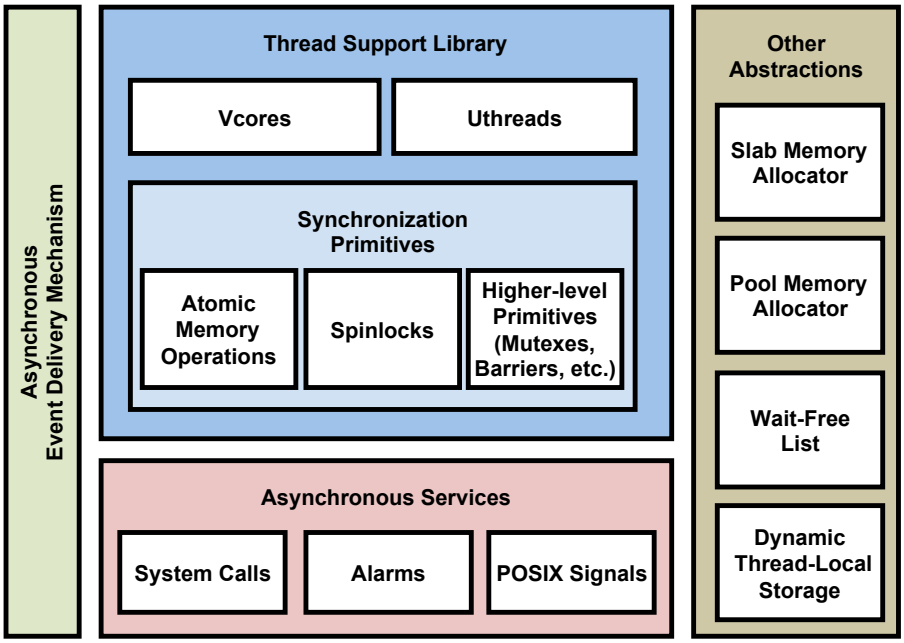


Figure 2.1: The collection of abstractions provided by parlib

Figure 2.1 shows an overview of all of the abstractions provided by parlib. These abstractions include:

1. A thread support library that provides the building blocks for developing arbitrary user-level threading libraries. This includes the **vcore** abstraction (for requesting/yielding cores), the **uthread** abstraction (for working with user-level threads, i.e. creating, running, yielding a thread, etc.), and a set of synchronization primitives for integrating blocking mutexes, futexes, condition variables, semaphores, and barriers into a user-level scheduler. A common set of atomic memory operations and various spinlock implementations are included as well.
2. An asynchronous event delivery mechanism that allows an application to interact with a set of underlying asynchronous services through a common API. Applications register callbacks for events associated with each service and then wait for those callbacks to be triggered at the appropriate time. The existence of this mechanism and the services it supports enable thread schedulers to retain control of their cores whenever operations are being performed on their behalf.
3. Asynchronous services for system calls, alarms, and POSIX signal delivery. The system call service provides interposition to ensure that all calls that normally block in the kernel will be issued in a non-blocking manner. Threads can then "block" themselves in user-space and wait for a callback to wake them up when their syscall completes. The POSIX signal delivery service provides a way of directing signals at an individual user thread rather than a kernel one. Alarms are useful for building preemptive scheduling policies as well as many other timer-based abstractions.

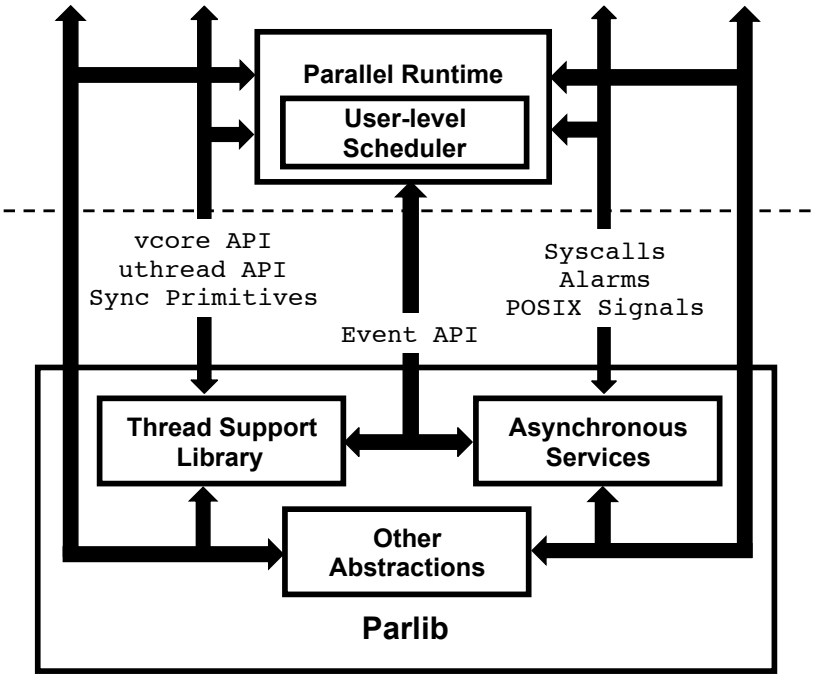


Figure 2.2: Parlib and the APIs it exposes in support of building parallel runtimes based on user-level scheduling.

4. A collection of other useful abstractions that developers can use in support of building efficient parallel runtimes. These abstractions include pool and slab allocators for dynamic memory management, a simplified wait-free list abstraction (WFL), and dynamic user-level thread local storage (DTLS). The slab allocator, WFLs, and DTLS are also used within parlib itself.

Figure 2.2 shows how these abstractions interact with one another in support of building custom parallel runtimes. Each abstraction provides a set of *bi-directional* APIs that other components are able to take advantage of. On one side of these APIs are *library calls*, implemented by the abstraction itself. On the other side are *callbacks*, which are declared by the abstraction, but implemented by the consumer of that abstraction. Library calls provide a way to trigger some operation to be performed. Callbacks provide a way to notify when those operations have completed, or otherwise branch back into the calling component to allow it to perform some custom operation. As an example, consider a user-level scheduler that wants to request direct access to a core. It first puts in a core request through the vcore API provided by the thread support library. The thread support library then submits this request to the system and returns. When a core eventually becomes available, it first pops up into code owned by the thread support library, which then triggers a callback into the user-level scheduler. At this point, the core is now controlled by the user-level scheduler and is able to execute whatever code it wants. We expand on this idea of *bi-directional* APIs as we explain each abstraction in detail in the following sections.

There are currently two ports of parlib, one for Akaros, and one for Linux. Although the ultimate goal is to provide a common, OS-independent API, the API is still in flux and currently differs slightly between the Akaros and Linux ports. Most of these differences arise from the fact that parlib was originally designed as the user-space counterpart to the Akaros kernel (much like glibc is to Linux). The parlib API is therefore designed to directly expose some of the abstractions we've built into Akaros. In order to provide a similar API, the Linux port has to fake some of these abstractions. Most notably, Linux has no way of providing true dedicated access to cores, and Linux's asynchronous system call mechanism is much heavier weight than Akaros's. Where emulation becomes prohibitive in terms of performance, we break the abstraction to keep both systems running fast.

For developers, this means that you can't just take a scheduler written against the Linux parlib API and run it directly on Akaros (or vice versa). However, the effort to port them has proved to be minimal (on the order of hours), and the difference between the APIs is continually getting smaller.

In the following sections, we walk through each of the abstractions listed above, and explain them in detail. Unless otherwise noted, each abstraction is described in an OS-independent manner, and the differences between the Akaros and Linux ports are pointed out, where appropriate.

2.2 Thread Support Library

The thread support library provides developers with a set of mechanisms for building custom user-level threading packages. Its API exposes operations common to existing threading packages such as creating, running and yielding threads, as well as not-so-common operations, such as requesting direct access to an underlying core.

The library itself is built on top of a unique underlying threading model. Unlike traditional threading models, which provide either a 1:1, N:1, or a hybrid M:N mapping of user-thread to kernel-thread, parlib strives to provide a threading model that completely decouples user-threads from kernel-threads. At a high level, this is achieveable, so long as the following two mechanisms are available:

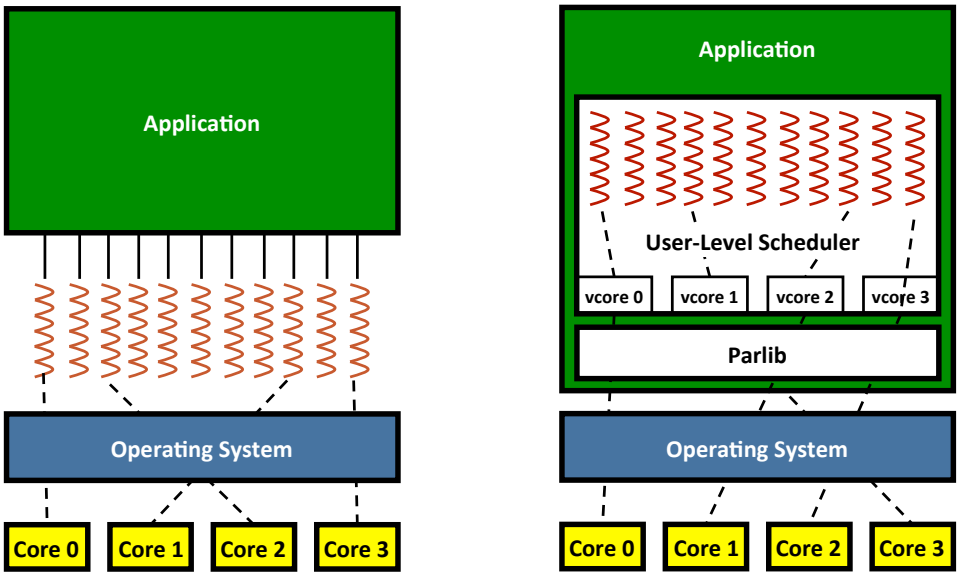
- The kernel hands out cores, *not* threads, as the allocatable processing unit available to an application.
- All system calls are completely asynchronous.

In Akaros, both of these capabilities are inherently available, as the Akaros kernel API was designed to explicity provide this functionality. On Linux, these mechanisms are only available in a limited sense, but we are able to "fake" them well enough that the threading model on both systems looks the same from the perspective of any code running on top of it. It simply runs less efficiently on Linux than it does on Akaros.

Using this threading model, processes request cores, not threads, from the underlying system. As these cores come online, there is a single entry point they pop in to *every* time

the kernel passes control of them to user-space (with one notable exception[1]). This is true, whether it is the first time the core is coming online, the core is being notified of a syscall completion event, or a core has been sent an interrupt (or fault) that is being reflected back to user-space. The entry point itself runs within a special transition context, which serves to boostrap the core and figure out what it should do next (usually handle some event and find a user-level thread to run and resume it). It can be thought of as the user-space counterpart to interrupt context in the kernel.

Combining this mechanism with a completely asynchronous system call interface means that a process will always get its core back after a system call has been issued. It typically uses this opportunity to save off the context that issued the system call and find another thread to run while the system call completes. Any threads that the kernel wishes to launch in order to complete the system call or do other kernel-specific tasks, are decoupled from the user-level threads that may have triggered them. Figure 2.3 shows how this threading model compares to that of a traditional 1:1 threading model from the perspective of an application built on top of it.



(a) Threads running in a traditional 1:1 threading model

(b) Threads running under parlib's decoupled threading model

Figure 2.3: This figure shows a direct comparison of threads running under a traditional 1:1 threading model, compared to parlib's decoupled threading model. Under parlib, all threading is done in user-space, and a processes request cores (not threads) from the kernel, which pop up on a transition context represented by the vcore abstraction.

[1] The only time the kernel *doesn't* pass control to user-space via this entry point is when the code running on a core voluntarily enters the kernel via a syscall. In this case, the kernel will pass control directly back to whatever context issued the syscall.

The two most important abstractions provided by parlib for expressing this threading model are the **vcore** and **uthread** abstractions. Vcores provide uninterrupted, dedicated access to physical cores, and uthreads provide user-level threads to multiplex on top of those cores. Together, these abstractions provide a *bi-directional* API, on top of which developers can implement custom runtimes with varying user-level scheduling policies. On one side of the API, library calls provide common operations that all user-level schedulers will likely wish to perform (e.g. requesting access to a core, launching a thread, etc.). On the other side, scheduler writers implement callbacks that are triggered by these library calls whenever a customized scheduling decision needs to be made. As discussed below, most of these callbacks are set at compile time in a global *sched_ops* struct, but a few are passed as arguments directly to the API calls that trigger them.

To understand how code on both sides of this bi-directional API interacts, it is important to define distinct notions of *vcore context* and *uthread context*. A context is defined to include a stack, some register state, and an optional thread-local storage region (tls). Every vcore or uthread in the system has its own unique context. At any given time, an underlying physical core is executing on behalf of exactly one of these contexts. When executing code on behalf of a vcore, it executes in vcore context. When executing code on behalf of a uthread, it executes in uthread context. Vcore context is the transition context that a core first pops into, and serves to run code related to handling events from the kernel, scheduling uthreads to run, etc. It is the user-space counterpart to interrupt context in the kernel. Uthread context, on the other hand, runs the actual application logic inside a thread, and yields to vcore context whenever a new scheduling decision needs to be made. It is the job of the scheduler to coordinate the passing from one context to another through the callbacks triggered by the bi-directional API. Parlib provides the mechanisms to easily pass from one context to another, leaving the actual policy of how and when to do so up to the scheduler.

As an example, consider Figure 2.4, which demonstrates the process of switching back and forth between uthread and vcore context via the `uthread_yield()` and `run_uthread()` library calls. On the left, a user-level thread voluntarily yields to vcore context, passing a callback to be executed once the yield completes. On the right, this callback is passed a reference to the thread that yielded, and calls `run_uthread()` to resume that uthread wherever it left off. Parlib takes care of all the logic required to package up the uthread state, swap contexts, and trigger the callback.

Some library calls can only be called from vcore context, while others can only be called from uthread context (and some can be called from either). However, all callbacks are *only* ever triggered from vcore context. This emposes certain restrictions on the code that can be executed in these callbacks, but it also simplifies the overall design of the bi-directional API. In the following sections we go into greater detail on the actual APIs provided by the vcore and uthread abstractions, and discuss the implications of running code in one context vs. the other.

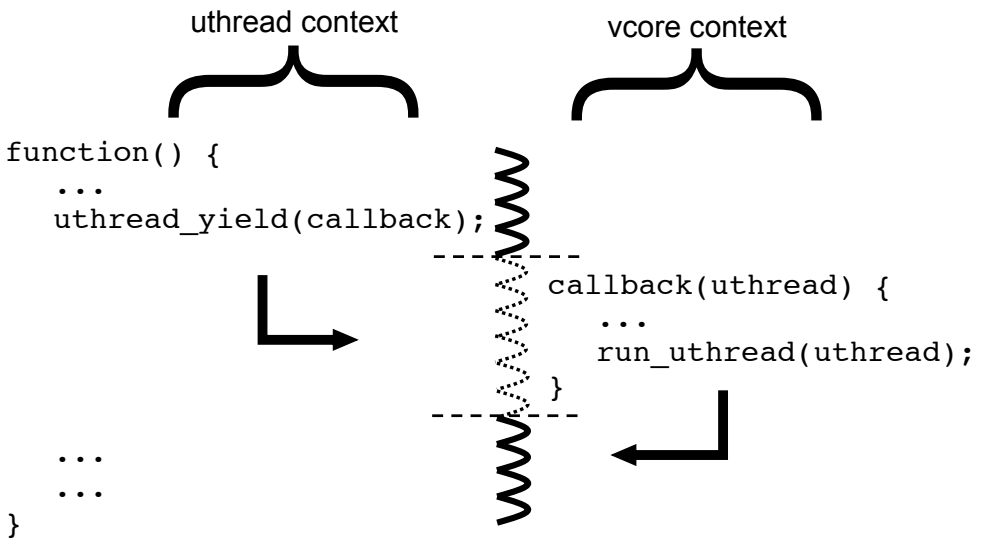


Figure 2.4: This figure shows the process of passing control from uthread context to vcore context and back. Parlib provides the mechanisms to easily pass from one context to another. Scheduler writers uses these mechanisms to implement their desired scheduling policies.

2.2.1 Vcores

The term vcore stands for "virtual core". Despite its name, a vcore is not meant to virtualize access to an underlying physical core. On the contrary, vcores are meant to provide uninterrupted, dedicated access to physical cores. The term "virtual" only refers to the virtual naming scheme used when handing out cores to a process. In this way, process 0 might have vcores 0-3 mapped to physical cores 0-3, while process 1 has vcores 0-3 mapped to physical cores 4-7.

As discussed in Section 1.2.1, this notion of a vcore is built directly into the Akaros kernel API. User-space programs request vcores from the system, and once granted, have exclusive access to them until they decide to return them (barring revocation as discussed later on). Contrast this with traditional systems, which attempt to multiplex tasks from different processes onto the same physical cores. In Akaros, the kernel scheduler knows nothing of such tasks, keeping the threading models between the kernel and user-space completely decoupled. All that Akaros knows is that the process has asked for access to a core and it grants this request if possible.

The vcore abstraction is simply the user-space counterpart to the Akaros kernel API. When a process successfully requests a vcore, Akaros finds a free a physical core, assigns it a vcore id, and then pops up on it in the address space of the requesting process (using a per-process, vcore-specific stack). It also switches into a tls region specific to that vcore. The kernel knows where to look for this stack and tls region because pointers to them exist in some per-vcore data structures shared with the kernel (i.e. `procdata`). All memory for these data structures is preallocated (by userspace) in one giant chunk the first time a vcore

request is made to the system. Once a vcore becomes live, it starts running in vcore context, and is ready to begin executing code on behalf of the process.

On Linux, this notion of a vcore is not supported directly. APIs exist to limit a process to run on a specific set of cores (using Linux containers), as well as to pin threads to individual cores within that set (using cpu affinities), but there is no coordinated way to give a process uninterrupted access to a core or fluidly pass cores back and forth between processes (the notions of cores and threads are too intimately intertwined in Linux). One could imagine modifying the kernel to provide such an abstraction, or running an external process to mediate the flow of cores between processes, but that still wouldn't provide complete isolation, as nothing prevents Linux itself from running a kernel task on one of your cores.

In light of this inherent limitation, we take a simple approach to "fake" the vcore abstraction on Linux. We don't even attempt to provide a system wide abstraction to coordinate access to cores across processes. Instead, within each process, we launch one pinned Linux task per physical core to act as a proxy for the vcore. We then park each task on a futex and wait for them to be requested by the process. When a vcore request comes in, we simply find a task that is parked and wake it up. Just as in Akaros, the vcore pops up on the core it is pinned to, running on a dedicated stack, with a private tls region.

This works well if there is only one process running on the machine. However, it breaks down as soon as you have multiple processes since the kernel is not involved in handing out vcores on a system wide basis (though each process will likely perform no worse than it otherwise would as a normal Linux process). The kernel will simply multiplex corresponding vcores from each process on top of the same physical cores. We leave it as future work to figure out how to improve this situation for Linux.

Vcore Context

Although the vcore abstractions provided by Akaros and Linux differ in terms of true dedicated access to cores, they look identical from the perspective of the code that runs on top of them. This means that any callbacks that get triggered will have the same semantics, independent of the OS they are running on (since callbacks always run in vcore context).

In general, vcore context is meant as the user-space counterpart to interrupt context in the kernel. As such, a certain set of rules apply to code running in vcore context.

- Vcore context is non-interruptable. As discussed in Section 2.3, parlib provides an asynchronous event delivery mechanism to pass messages back and forth between vcores. When running in uthread context, it's possible to receive a notification that will interrupt what you are doing and drop you into vcore context in order to process the event. Just as hardware interrupts are disabled when running in interrupt context in the kernel, notifications are disabled when running in vcore context in parlib. However, checks are in place to make sure we never miss the arrival of a new event, possibly returning back to vcore context immediately upon restoring a uthread, if necessary.

- You always enter vcore context at the top of the vcore stack. This makes it so that a full context switch is not necessary to drop into vcore context (since none of its register state needs to be restored). Instead, we simply swap in the vcore's tls, jump to the top of its stack, and execute a specified start routine (`vcore_entry()`, by default). Referring back to Figure 2.4, only one full context switch actually takes place in this code. Swapping into and out of vcore context is extremely light weight.

- Vcore context must never return. From the time you enter vcore context at the top of the stack, to the time you exit it, the only options available are to find a uthread context to switch into or yield the vcore back to the system.

In general, the rules listed above must be kept in mind by scheduler writers when implementing any callbacks triggered through parlib's bi-directional scheduler API. Furthermore, each individual callback may have its own set of semantics that need to be considered. In the following section, we introduce each of the actual API calls and explain their semantics in detail.

The Vcore API

The vcore API gives developers the ability to request and yield vcores, as well as query the system about the state of those vcores. The API is still in flux, but the most important calls, which are unlikely to change, are listed below. Each call is listed in relation to the callback that it ultimately triggers.

Library Call	Callback
`vcore_lib_init()`	-
`vcore_request(count)`	`vcore_entry()`
`vcore_reenter(entry_func)`	`entry_func()`
`vcore_yield()`	-
`vcore_id()`	-
`max_vcores()`	-
`num_vcores()`	-

`vcore_lib_init`: This call sets up the internal data structures required by the vcore subsystem, and initializes the system to be able to accept incoming vcore requests. On both Akaros and Linux, `vcore_lib_init()` is called automatically upon the first vcore request if it has not been called already.

`vcore_request`: This call puts in a request for a specified number of vcores. As vcores pop up, they each begin executing the body of a globally defined `vcore_entry()` callback. By default, the `vcore_entry()` callback is implemented by the uthread abstraction discussed in the following section. In theory, it can be overridden, but it most likely won't be so long as the uthread abstraction is in use.

vcore_reenter: This call jumps back to the top of the current vcore stack and starts running fresh inside the specified callback function. This call should only ever be made from vcore context (or after you have safely saved off your current uthread context), because the state of the calling context is lost. This call is used internally as part of yielding a uthread, as well as other places within vcore context where it is known that the state of previous stack frames are no longer needed.

vcore_yield: Yield the current vcore back to the system. It's possible that the yield might fail (if it detects that there are pending notifications, for example). When this happens, the vcore is restarted at the top of its stack in `vcore_entry()`. Once the vcore has truly been yielded, it will only come back online as the result of a subsequent `vcore_request()`.

vcore_id: This simply returns the id of the calling vcore. Its value is stored in the vcore's TLS and remains unchanged throughout the lifetime of the vcore.

num_vcores: This call returns the number of vcores currently allocated to the process. This call is inherently racy, and there is no guarantee that the vcore count does not change between calling this function and making any other vcore API calls (i.e. don't rely on this value to determine an exact count of how many vcores you already have before putting in a request for additional ones).

max_vcores: This call returns the maximum number of vcores that can ever be requested. On SMT systems, this count includes both hyperthreaded cores.

When a process first starts up it is not technically running on a vcore. Only after the main thread makes its first call to `vcore_request()` does it transition to running on top of a vcore. On Akaros, this transition happens via a system call which transforms the process from an SCP to an MCP, and then allocates a vcore for the main thread to resume on. On Linux, the main context is simply packaged up, a vcore is unparked from its futex, and the main context is passed off to it. In both cases, the allocated vcore starts executing the body of the `vcore_entry()` callback, whose job it is to resume the main context and continue executing the process.

In general, `vcore_entry()` is the *single* entry point into vcore context from the perspective of the underlying system. By default, `vcore_entry()` is implemented by the uthread abstraction. Its full implementation can be seen below:

```
void vcore_entry() {
    handle_events();
    if (current_uthread)
        run_current_uthread();
    sched_ops->sched_entry();   // should not return
    vcore_yield();
}
```

The general logic flows as follows:

1. Handle any outstanding events that have been posted since the last time we dropped into vcore context.
2. If the the `current_uthread` TLS variable is set, then we must have dropped into vcore context via a notification (as opposed to a voluntary yield). Since we just handled all outstanding events, go ahead and restart the interrupted uthread.
3. If `current_uthread` is not set, then defer the logic of what to do next to the user-level scheduler via its `sched_entry()` callback.
4. Finally, yield the vcore back to the system if `sched_entry()` ever returns (which it should not).

From a scheduler's perspective, `sched_ops->sched_entry()` is the entry point where an actual scheduling decision needs to be made. As discussed in the following section, all of the other callbacks into the scheduler are made in support of this decision (e.g. initializing a thread, placing a thread on a queue, etc.). The scheduler writer must take care to implement this callback, such that they do not violate any of the rules that apply to vcore context.

Caveats

Currently, the vcore API treats all cores as equal and there is no way to specify a request to a specific underlying physical core. Moreover, the current API only allows you to request vcores in an "additive" manner – there is no way to specify an absolute number of vcores that you want for the entire process. The biggest drawback of the additive interface is that there is no way of "canceling" a request once it has been submitted. The system will simply mark down each request and eventually grant you the sum of all of the vcores you have ever asked for (even if you don't need them anymore). With an absolute interface you would be able to increase or decrease your desired number of vcores at any time, giving the system better information about your real time needs. We are currently looking at ways to make these types of requests more expressive.

As alluded to in Section 2.2.1, it is possible for a vcore to be revoked from a process on Akaros without prior notice. Whenever this happens, the process needs some way of safely recovering the context of whatever was previously running on that vcore. In general, you can ask to be notified (via interrupt), or periodically poll to see if any of your vcores have been taken offline. However, there are many subtle issues related to actually recovering the context that was interrupted and restarting it without any unwanted side effects. On Linux, vcore preemption is a non-issue, because it is impossible for a vcore to be revoked in our "faked" vcore abstraction. All vcores persist throughput the lifetime of the process because there is no underlying system that is capable of revoking them. We refer the reader to Section 6 of Barret Rhoden's dissertation [2] for details on how we deal with this on Akaros.

Finally, the Linux implementation of vcores currently pulls in an undesired dependence on `lpthread`, polluting the `pthread` namespace and preventing us from providing an ABI

compatible (as opposed to API compatible) user-level pthread library on top of parlib. Conceptually, this dependence should be unnecessary: you can emulate vcores on top of raw linux tasks created via the `clone` system call. Doing so, however, has undesired consequences when calling into arbitrary glibc code on those vcores. For example, glibc's `low-level-lock` implementation relies on accessing a per-thread TLS variable in order to make its library calls thread-safe. The pthread library initializes this TLS variable to a unique value by passing the kernel a reference to it when the `clone()` system call is made (through the `ptid` parameter). In theory, we could do something similar, except that the required TLS variable is stored at a specific offset inside of the opaque `struct pthread` type internal to glibc. Emulating this same behaviour would therefore require exposing the `struct pthread` type to parlib (which could have unforeseen consequences on different versions of glibc). Furthermore, pulling in `lpthread` sets up a bunch of other global state required for glibc to be thread-safe (including setting up a function pointer to `pthread_mutex_lock()`, so that all necessary locking operations can be carried out). Without pulling in `lpthread`, we would have to provide this functionality ourselves.

Akaros doesn't sufffer from these problems since it builds a customized glibc with its own `low-level-lock` implementation (which we currently just implement with spinlocks). Additionally, Akaros doesn't emulate vcores using kernel supplied threads: they are actual dedicated physical cores (which is why using a spinlock for the `low-level-lock` is ok). Even if we were to build our own parlib-aware glibc for linux (or integrate parlib directly into glibc), we would still require some *pthread-like* abstraction to emulate our vcores and provide proper thread-based synchronization between them. We would just need to give all the functions in this library a different name to keep from polluting the `pthread` namespace. We leave this as future work.

2.2.2 Uthreads

Uthreads provide the base "user level" threading abstraction, which scheduler writers use to multiplex threads on top of vcores. All application logic is ran inside a uthread.

Figure 2.5, shows the layering of both the vcore and uthread abstractions with respect to a user-level scheduler they both interact with. In general, schedulers make library calls through both abstractions, but callbacks from the vcore abstraction get reflected through the uthread abstraction (unless they are passed as parameters to the actual library call). As we saw with `vcore_entry()` this reflection is necessary in order to run uthread specific code before ultimately triggering the callback to the scheduler (e.g. via `sched_ops->sched_entry()`).

The uthread abstraction itself consists of three primary components: a set of library calls, a set of callbacks, and a base `uthread_t` data structure. The library calls serve to provide common functionality that all user-level threading implementions will likely require (e.g. initializing a thread, running a thread, yielding a thread, etc.). The callbacks are triggered whenever a scheduler-specific action needs to be taken on behalf of a uthread. For example, consider Figure 2.4 again. The `uthread_yield()` call does the heavy work of packaging up the current context and yielding to the vcore. However, it doesn't actually *do* anything with

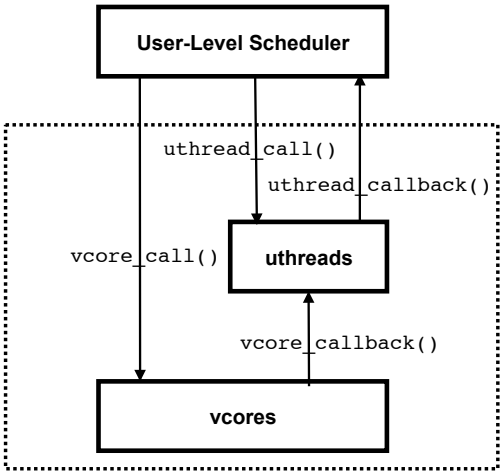


Figure 2.5: This figure shows the layering of the vcore and uthread abstractions with respect to the bi-directional API they provide to scheduler writers.

the context itself. Instead, that context is passed to a scheduler-defined callback which is able to decide what to do with it next. It may decide to resume the context immediately (as is the case in the example), or save it off and run a previously saved context in its place.

The `uthread_t` data structure is used to hold the uthread context as is gets passed around between the different library calls and callbacks. However, the uthread library calls never allocate memory for `uthread_t` structs themselves. Instead, it is up to the scheduler writer to allocate space for them before passing them as parameters to the various uthread library calls. Leaving this allocation up to the scheduler gives it the opportunity to "extend" the `uthread_t` struct to contain custom fields that would otherwise not be availalable. It does so by declaring a new struct type and placing the `uthread_t` struct as the first field. Using standard C casting operations, pointers to this extended struct can now be passed around and treated as either type. This is most useful in the callbacks, where accessing the custom fields associated with a uthread would otherwise be a cumbersome operation.

Each uthread maintains a pointer to its `uthread_t` struct in a TLS variable called `current_uthread`. This gives them easy access to the contents of this struct inside the body of the various uthread library calls. Likewise, a vcore's value of `current_uthread` is updated whenever a new uthread is started on it. In addition to helping us catch bugs, this allows us to determine which (if any) uthread was interrupted in the case that we drop into vcore context via an external interrupt. We've already seen this usage of `current_thread` in the implemention of `vcore_entry()` discussed in the previous section.

Uthread Callbacks

The uthread abstraction distinguishes between two types of callbacks: call-site specific and externally triggerable. Call-site specific callbacks are explicitly passed to uthread library calls and are triggered in direct response to making those call. For example, in our user-

level pthreads implementation `pthread_yield()` and `pthread_exit()` are both implemented with a call to `uthread_yield()`, but each passes a different callback to be triggered once the uthread has been yielded. Externally triggerable callbacks, on the other hand, are global callbacks that are triggered in response to external events occurring (e.g. vcores coming online, external components such as mutexes blocking one of your uthreads, etc.)

The full set of externally triggerable callbacks are defined in a global `schedule_ops` data structure, which has the following signature:

```
struct schedule_ops {
    void (*sched_entry)();
    void (*thread_blockon_sysc)(uthread, syscall);
    void (*thread_refl_fault)(uthread, num, err, aux);
    void (*thread_paused)(uthread);
    void (*thread_has_blocked)(uthread, flags);
    void (*thread_runnable)(uthread);
};
```

Scheduler writers instantiate an instance of this struct with their custom callbacks in place and assign it to a global variable called `sched_ops`. The uthread library then uses the `sched_ops` variable to trigger these scheduler-defined callbacks at the appropriate time.

The first four callbacks (`sched_entry()`, `thread_blockon_sysc()`, `thread_refl_fault()`, and `thread_paused()`) are all triggered independent of making an explicit uthread library call. Together, they represent the set of externally-triggerable callbacks which need to be implemented on behalf of the underlying system. They are discussed below. The remaining two callbacks (`thread_has_blocked()` and `thread_runnable()`) are *only* ever triggered as the direct result of making a library call. External components such as mutexes and condition variables, which are not necessarily part of a scheduler (but work on its behalf), trigger these callbacks whenever they *block* or *unblock* (i.e. make runnable) a uthread. The `thread_paused()` callback can also be triggered by a library call in addition to being triggered externally. These calls are discussed in the following section, alongside the library calls that trigger them.

`sched_entry`: The `sched_entry()` callback is simply a reflected version of the `vcore_entry()` callback discussed in the previous section. It is triggered whenever a new scheduling decision needs to be made on a vcore. Its primary purpose is to decide which thread to run next, and run it. If there are no threads to run, it can either spin in a loop waiting for a thread to become available, or yield the vcore back to the system. This function should never return.

If a scheduler writer decides to spin in `sched_entry()`, care must be taken to call `handle_events()` as part of the loop. Since vcores are non-interruptable, failure to call `handle_events()` could result in deadlock since some threads may only become available as the result of processing one of these events (e.g. the completion of an asynchronous system call). Details on the `handle_events()` call are discussed in Section 2.3.

thread_blockon_sysc: As mentioned previously, the threading model provided by parlib relies on the existence of a fully asynchronous system call interface. This callback is triggered whenever the kernel returns control of a core to user-space after an asynchronous system call has been issued. Although each system call is non-blocking from the perspective of the kernel, any user-level threads that issue them will likely need to block in user-space while they wait for them to complete. This callback provides a scheduler specific way of deciding what to do with these threads while they wait.

thread_refl_fault: This callback is triggered whenever the currently running uthread generates a processor fault (e.g. page fault, divide by 0, etc.). It gives the process the opportunity to patch things up and continue running the faulting thread, kill the faulting thread, or abort the process completely. On Akaros, these faults trap directly into the OS before being reflected back to user space on the vcore they occurred on. As discussed in the following section, this reflection happens inside the body of `run_current_uthread()` as called by `vcore_entry()`. On Linux, we don't yet support reflected faults, but the eventual plan is to first intercept them via the posix signals that are generated for them, and then reflect them back to the vcore in a manner similar to Akaros.

thread_paused: This callback is triggered whenever something external to the scheduler has caused a uthread to pause execution for some reason (but the uthread does not need to block). It gives the scheduler the opportunity to reschedule the paused thread for execution. On Akaros, this callback is triggered whenever a uthread is interrupted as the result of a preemption event taking a core away. As discussed below, this callback is also triggered by some of our synchronization primitives if they drop into vcore context but no longer desire to actually block the thread for whatever reason. Such functionality is required by the semantics of a `futex`, for example.

Uthread Library Calls

In the previous section, we presented the set of scheduler callbacks triggered by the underlying system. Scheduler writers must implement these callbacks to do something meaningful for the scheduling policy they are trying to enforce. In this section, we present the set of uthread library calls which most schedulers will likely need to call at some point. These library calls are typically made inside a scheduler defined API call (e.g. `pthread_create()`, `pthread_yield()`, etc.), or within one of the callbacks triggered on the scheduler. Some of the library calls trigger callbacks themselves, which, in turn make additional uthread library calls. The table below lists each of these library calls in relation to the scheduler callbacks they ultimately trigger.

uthread_lib_init: This library call initializes the uthread abstraction itself. It is called by the main thread of the process and does the work of packaging up the main context, putting in the initial request for a vcore, and preparing the main context to be resumed on that

Library Call	Callback
uthread_lib_init(main_thread)	-
uthread_init(uthread)	-
uthread_cleanup(uthread)	-
run_uthread(uthread)	-
run_current_uthread()	-
uthread_yield(yield_func, yield_arg)	yield_func(yield_arg) sched_entry()
uthread_has_blocked(uthread, flags)	thread_has_blocked(uthread, flags)
uthread_paused(uthread, flags)	thread_paused(uthread, flags)
uthread_runnable(uthread)	thread_runnable(uthread)

vcore once it pops up. It takes a reference to a pre-allocated `uthread_t` as a parameter, so that it has a place to store the main context during this transition.

`uthread_init`: This library call initializes a new uthread for use in the system. It is typically called as part of a scheduler-specific thread creation call (e.g. `pthread_create()`). It takes a reference to a pre-allocated `uthread_t` as a parameter and takes care of allocating and initializing TLS for the uthread and initializing any internal fields used by the uthread library. The scheduler allocates space for (an "extended" version of) the `uthread_t` struct and passes it to `uthread_init()` before initializing any custom fields itself. Once this call returns, the uthread is ready to be executed via a call to `run_uthread()`, as explained below.

`uthread_cleanup`: This library call cleans up a previously initialized uthread. It should only be called after the uthread has completed and has no more work to do. Once this call returns, it is safe to free any memory associated with the uthread.

`run_uthread`: This library call resumes execution of a previously initialized uthread. It takes a reference to this uthread as a parameter, and simply restores its context on the current vcore.

`run_current_uthread`: This library call performs the same task as `run_uthread()`, except that it restores the most recently running uthread in cases where we drop into vcore context via an interrupt of some sort. We special case this call because interrupted contexts are saved in a special location that this call knows to look in. Additionally, since we only ever call this function when we detect that we've been interrupted, we overload it to trigger the `thread_refl_fault()` callback and restart the core in cases where this is necessary. Faults themselves are detected by inspecting the trapframe of the saved context to if see one has occurred. Fault detection is currently only supported on Akaros.

`uthread_yield`: This library call pauses execution of the current uthread and transitions the core into vcore context so a new scheduling decision can be made. It first triggers the `yield_func()` callback passed to it as an argument before directly calling `vcore_entry()` to

(essentially) restart the core from scratch. From here, `sched_entry()` will be triggered, and a new scheduling decision will be made. This call is typically made directly by a scheduler specific yield function (e.g. `pthread_yield()`), or by a scheduler-aware synchronization primitive (e.g. the blocking mutexes, condition variable, etc. discussed in Section 2.2.3).

The `yield_func()` callback receives a reference to the uthread that yielded, as well as a custom argument passed via the `yield_arg` parameter to `uthread_yield()`. We pass `yield_func()` as an argument to `uthread_yield()` (rather than installing a global callback in the `sched_ops` struct) because yielding can occur for many different reasons, and the callback that should be run is often call-site specific. For example, a blocking mutex will want to do something very different with the yielding thread than a simple `pthread_yield()` will.

`uthread_has_blocked`: This library call informs the uthread subsystem that an external component has blocked a uthread from further execution. When called, it triggers the `thread_has_blocked()` callback in the global `sched_ops` struct in order to notify the scheduler that this blocking has occurred. A subsequent call to `uthread_runnable()` is necessary to make this thread schedulable again. As we will see in Section 2.2.3 this call is used heavily by the various user-level synchronization primitives provided by parlib.

`uthread_paused`: This library call is similar to `uthread_has_blocked()`, except that it notifies the uthread subsystem that an external component has briefly paused the uthread, but would like for it to remain schedulable. When called, it triggers the `thread_paused()` callback in the global `sched_ops` struct. However, no subsequent call to `uthread_runnable()` is needed, and the scheduler is free to schedule the uthread again as it sees fit. As discussed in Section 2.2.3, this call is currently used in the implementation of some of our synchronization primitives, such as `futexes`.

`uthread_runnable`: As mentioned previously, this library call is meant to inform the scheduler that a uthread is now schedulable. It is typically only called either immediately after a uthread has first been initialized, or after a previous call to `uthread_has_blocked()`. It triggers the `thread_runnable()` callback in the global `sched_ops` struct. This callback is typically implemented to put the uthread on some sort of queue, so that `sched_entry()` can find it and resume it when it makes its next scheduling decision.

Caveats

The uthread API presented above is identical on both Akaros on Linux. However, its underlying implementation differs in one very important aspect: every Linux uthread has a heavy-weight pthread backing it up. At the time `uthread_init()` is called, the uthread subsystem makes an internal call to `pthread_create()` and associates the newly created pthread one-to-one with the uthread being initialized. However, this pthread is not actually used to run the core logic of the uthread. Instead, it serves two primary purposes:

- To set up a thread-local storage (TLS) region for use by the uthread
- To help emulate asynchronous system calls by servicing blocking system calls on behalf of the uthread it backs.

Originally, we were using glibc's undocumented `_dl_allocate_tls()` to allocate TLS for each uthread (which is what we still do on Akaros). However, just as with our emulated vcores, uthreads suffer the same drawbacks discussed in Section 2.2.1 related to calling glibc functions in a thread-safe manner. By creating a full-blown pthread and then parking it on a futex, uthreads are able to "steal" their backing pthread's properly intialized TLS and use it as their own. Furthermore, some external mechanism (i.e. not a uthread running on a vcore) is required to emulate servicing asynchronous system calls on behalf of a uthread. We overload the backing pthread to provide this functionality as well. More information on the exact mechanism used to achieve this is discussed in Section 2.4.1.

Another caveat worth pointing out is the lack of a callback for signaling syscall completion through the uthread API itself. The API provides the `thread_blockon_sysc()` callback to notify a scheduler when a system call has been issued and a thread should be blocked. However, no callback exists to actually tell us when that system call has completed and the thread can be resumed. This functionality obviously exists, we just don't currently expose it through the uthread API as a callback. Instead, we rely on the event delivery mechanism discussed in Section 2.3 to notify us of syscall completion directly. However, it could (and arguably should) be wrapped in a scheduler callback to keep it symmetric with the `thread_blockon_sysc()` callback. We leave this as future work.

2.2.3 Synchronization Primitives

Any reasonable threading library will require some set of synchronization primitives in order to synchronize access to critical sections of code. For basic synchronization, simple atomic memory operations will often suffice. Beyond that, spinlocks and higher-level synchronization constructs such as mutexes, barriers, and condition variables are often necessary.

Atomic Memory Operations

At the lowest level, parlib provides a set of standard atomic memory operations for synchronizing access to individual words in memory. Most of these operations are just wrappers around gcc's `__sync*` operations (e.g. `atomic_read()`, `atomic_add()`, `atomic_cas()`), but others provide a bit more functionality (e.g. `atomic_add_not_zero()`). These atomic memory operations provide the foundation on top of which all other synchronization primitives are built. They are also useful in their own right for implementing constructs such as reference counters, getting and setting flags atomically, etc. Both Linux and Akaros provide the same set of primitives.

Spin_locks, Mcs_locks, and Pdr_locks

Beyond simple atomic memory operations, parlib also provides spinlocks to protect access to short critical sections of code. In total, parlib provides four different variants of spinlocks: basic `spinlocks`, `mcs_locks`, and a new type of lock called a `pdr_lock` for which there are both `spinlock` and `mcs_lock` variants. At a fundamental level, these operations all rely on the existence of an atomic `test_and_set()` operation in the underlying hardware.

Basic spinlocks are the simplest. A single variable is used to represent a lock, and contenders of this lock spin on this variable while waiting to acquire it. Mcs locks are similar, except that each contender spins on a local copy of the lock to reduce cache coherency traffic under high contention. Contenders themselves are maintained in a linked list, and control of the lock is passed from one contender to the next in the order in which they arrived at the lock.

In general, spinlocks and mcs locks are useful for protecting access to the low-level data structures used when building higher-level synchronization primitives for uthread context. Moreover, they are *essential* to protect *all* critical sections of code when executing in vcore context. Recall that vcore context is the user-space counterpart to interrupt context in the kernel and therefore can't rely on yield-based synchronization constructs for protection. There is no other context for the vcore to yield to.

However, spinlocks alone are not sufficient protection in an environment where vcore's can be preempted for arbitrary amounts of time (as is the case with Akaros). If the vcore holding a lock is preempted, all other vcores will spin indefinitely, waiting for the preempted vcore to come back online and release the lock (which may never happen). To solve this problem, Akaros provides a new type of lock called a "pdr" lock, which stands for "Preemption Detection and Recovery". These locks are, in a sense, vcore "aware" and can detect when a vcore holding a lock goes offline and recover from it.

Pdr locks operate by atomically acquiring a lock and grabbing a reference to the lock holder at the same time. Contenders of the lock use this reference to continuously check who the lock holder is as part of their busy-waiting loop. Using a kernel provided flag (`VC_PREEMPTED`), contenders are able to see if the lock holder is still online or not. If it is, they just continue spinning. If it is not, they attempt to "become" that vcore by issuing a special `sys_change_vcore()` syscall. This call essentially takes the calling vcore offline and restores the preempted vcore in its place. This allows the restored vcore to continue running and eventually release the lock. At the same time, the vcore issuing the `sys_change_vcore()` is marked as preempted, and its `VC_PREEMPTED` flag is set accordingly. This step is essential, because that vcore may hold other locks that need to be recovered in the same way. Eventually, all preempted vcores that hold locks will be allowed to run, and the system will reach steady state again. The details of these locks and the issues around solving all their potential races can be found in Chapter 6.2 of Barret Rhoden's dissertation [2].

As mentioned previously (and discussed in detail in Section 2.3), parlib provides the ability to asynchronously "notify" a vcore and cause it to drop immediately into vcore context (if it isn't already). By default, notifications are enabled while running in uthread context,

meaning that a uthread could be forced offline at any time. As such, leaving notifications enabled could prove problematic when holding locks to protect critical sections of code when running in uthread context.

To solve this problem, pdr locks are overloaded to provide `irq_save`-like features when called from uthread context. This functionality is accomplished by disabling notifications before attempting to acquire the lock, and reenabling notifications after releasing the lock. As discussed in Section 2.3, a `uthread_disable_notifs()` function exists to both disable notifications and ensure that a uthread is pinned to its vcore the entire time that notifications are disabled (which is essential for the pdr locks to work). Additionally, both `uthread_disable_notifs()` and its notification-enabling counterpart maintain a "disable depth" so that nested locks can be acquired without re-enabling notifications or unpinning a uthread from its vcore prematurely.

On Linux, none of this "pdr" functionality is necessary because vcores can never be preempted by the underlying system. However, the `irq_save()`-like features are essential on both systems. Because of this, we adopt the same API as Akaros even though the locks have no actual "pdr" capabilities under the hood.

In terms of correctness, pdr locks must *always* be used when acquiring a lock in vcore context, but are optional in uthread context so long as you don't need notifications disabled and you are careful not to nest them.

Higher-Level Synchronization Primitives

In addition to the low-level atomic memory operations and spinlocks discussed above, parlib also provides higher-level synchronization primitives for use within a uthread. Specifically, it provides **mutex**, **futex**, **barrier**, **semaphore**, and **condition variable** primitives. These primitives are useless on their own, and must be integrated with a user-level scheduler to provide any meaningful functionality.

In contrast to spinlocks, these higher-level constructs attempt to "block" or "yield" a uthread instead of simply spinning in a busy loop waiting to acquire a lock. However, under the hood they all use spinlocks to protect access to their underlying data structures.
Although the exact logic of each primitive is different, they all follow a similar pattern when blocking a uthread:

1. Attempt to acquire a lock based on some condition
2. If successful, continue
3. Otherwise, yield to vcore context, place the yielded context on a block queue, and inform the user-level scheduler that the uthread has been blocked.

As an example, consider the code for locking a mutex below (argument checking and the logic for recursive mutexes is removed for brevity):

```
int uthread_mutex_lock(uthread_mutex_t* mutex)
{
    spin_pdr_lock(&mutex->lock);
    while(mutex->locked) {
        uthread_yield(block_callback, mutex);
        spin_pdr_lock(&mutex->lock);
    }
    mutex->locked = 1;
    spin_pdr_unlock(&mutex->lock);
    return 0;
}

void block_callback(struct uthread *uthread, void *arg)
{
    uthread_mutex_t *mutex = (uthread_mutex_t *) arg;

    uthread_has_blocked(uthread, UTH_BLK_MUTEX);
    STAILQ_INSERT_TAIL(&mutex->queue, uthread, next);
    spin_pdr_unlock(&mutex->lock);
}
```

This code first acquires a spinlock to protect access to the higher level mutex lock (`mutex->lock`) and its block queue (`mutex->queue`). It then checks this lock to see if there is a current lock holder or not. If there is, it yields the current uthread via a call to `uthread_yield()`, passing `block_callback()` and the mutex as arguments. If there is not, it becomes the lock holder by setting the mutex lock, unlocking the spinlock, and returning.

In the case where it yields, the `block_callback()` first informs the scheduler that the uthread has blocked by calling `uthread_has_blocked()` and then places the yielded context on a block queue. Finally, it releases the spinlock. Under the hood, the call to `uthread_has_blocked()` triggers the global `thread_has_blocked()` callback. The scheduler only uses the `thread_has_blocked()` callback for bookkeeping and otherwise forgets about the uthread until it is unblocked at some point in the future. As such, the block queue itself is local to the mutex instance as opposed to some queue maintained by the scheduler.

It's also worth pointing out the usage of the spinlock here. The spinlock is acquired in the body of `uthread_mutex_lock()`, but released in the body of `block_callback()`. This technique is used to ensure that the mutex lock is accessed atomically with placing the uthread on the block queue. This pattern is used pervasively.

Additionally, returning from `uthread_yield()`, does not necessarily guarantee that you become the lock holder. Instead, it follows Mesa-like monitor semantics, forcing you try and obtain the lock again [37]. As such, the call immediately following the return from `uthread_yield()` grabs the spinlock and rechecks the value of the mutex lock in a loop. Only once it actually is able to acquire the lock does it return from the call, otherwise it just yields again.

Although not present in this example, there may be times when a synchronization primitive decides to yield, but never actually blocks the context because the condition it used

to decide to block has now become false (as is done in the the futex primitive). In these cases, `uthread_paused()` is called instead of `uthread_has_blocked()`, allowing the scheduler to re-enqueue the uthread for execution instead of forgetting about it.

When unblocking, a reverse pattern is followed by all synchronization primitives:

1. Unlock the lock
2. Find a uthread in the block queue
3. Inform the scheduler that it should wake it up

Again, consider the example below; this time for unlocking a mutex:

```
int uthread_mutex_unlock(uthread_mutex_t* mutex)
{
    spin_pdr_lock(&mutex->lock);
    mutex->locked = 0;
    upthread_t upthread = STAILQ_FIRST(&mutex->queue);
    if(upthread)
        STAILQ_REMOVE_HEAD(&mutex->queue, next);
    spin_pdr_unlock(&mutex->lock);

    if(upthread != NULL) {
        uthread_runnable((struct uthread*)upthread);
    }
    return 0;
}
```

As before, this code first acquires a spinlock to protect access to the higher level mutex lock and its block queue. It then releases the mutex lock and attempts to find a uthread in its block queue. If there is one, it removes a *single* uthread from the queue, releases the spinlock, and then unblocks that uthread via a call to `uthread_runnable()`. Internally, this triggers the `thread_runnable()` callback on the scheduler, allowing it to reschedule the uthread for execution (with emphasis on being *rescheduled* rather than switched-to, as consistent with Mesa-like monitor semantics). If no uthread was found, it simply releases the spinlock and returns.

As mentioned previously, all of the high-level synchronization primitives provided by parlib follow a similar pattern for blocking and unblocking a uthread. The details may be different, and the conditions for blocking may be different, but they all use a similar set of mechanisms to achieve the same goal. They all yield via the `uthread_yield()` call, inform the scheduler about the yield via either `uthread_has_blocked()` or `uthread_paused()`, and they all resume a uthread via `uthread_runnable()`. The details of these remaining primitives are omitted for brevity.

As an aside, a user-level scheduler may wish to provide the ability to kill a uthread at an arbitrary point in time (including while it is blocked inside one of these synchronization primitives). As presented, this would cause problems for mutexes because only a single uthread is woken up by a `uthread_mutex_unlock()` operation, and the uthread chosen might be one

that is already dead (meaning it will not able to actually grab the lock and unlock it again so another thread can be woken up). A simple fix for this would be to change the implementation of `uthread_mutex_unlock()` to support a Mesa-like broadcast() operation since the `uthread_mutex_block()` is already setup to handle Mesa-like blocking semantics. However, this operation could prove prohibitively expensive if there are many uthreads blocked on the mutex. A better solution would be to disallow threads from being killed except at specific synchronization points (e.g. following semantics similar to that of `pthread_cancel()`). Such semantics would need to be enforced inside the user-level scheduler itself rather than inside any of the individual synchronization primitives, e.g. by only killing a uthread at its next scheduling boundary. We leave it as future work to explore the various possibilities available here.

2.3 Asynchronous Events

In addition to the thread support library presented in the previous section, another key abstraction provided by parlib is its **asynchronous event delivery mechanism**. This abstraction provides a standardized way for the underlying system to communicate asynchronously with vcores. Moreover, vcores themselves can use this mechanism to communicate asynchronously with each other. In practice, we use this mechanism to signal system call completions, signal alarm events, and direct POSIX signals to individual uthreads, among other uses.

At a high level, this mechanism follows a basic producer/consumer model, whereby event messages are posted into queues that vcores asynchronously consume and process. Each queue can be configured such that IPI-like notifications will be sent to a vcore whenever a producer posts a message to it. The event messages themselves are modeled after *active messages* [38], allowing you to send payloads with each message and execute an arbitrary handler when that message arrives. Using active messages also keeps the logic of the consumers relatively straightforward (the entirety of the consumer logic is contained within a single `handle_events()` call).

Producers can either be a well defined *service* provided by the underlying system (e.g. asynchronous syscalls, alarms, etc.), or any arbitrary vcore that wishes to send another vcore a message. Each asynchronous service is statically assigned a unique `event_type`, which it uses to tag event messages that it sends to an event queue (e.g. `EV_SYSCALL`, `EV_ALARM`, etc). Processes use this `event_type` to register unique handlers that execute when messages of that type are consumed by a vcore. Vcores consume messages by running a special `handle_events()` function, which knows how to cycle through all event queues currently associated with a vcore, pull messages off of them, and execute their registered event handlers. Vcore-to-vcore messages have no assigned event types, and can use any free message type they wish (or no message type at all) when passing messages back and forth. In the future, we will likely add some notion of *pluggable* services that can be assigned a unique `event_type` at runtime.

One of the key features provided by this mechansim is the ability to send IPI-like notifications to a vcore to inform it of an event that has just been posted. As discussed in Section 2.2.1, these notifications cause a vcore to drop into vcore context and start executing at the top of its `vcore_entry()` function (assuming it wasn't already in vcore context). This function, in turn, immediately calls `handle_events()`, triggering all outstanding events for the vcore to be processed. Once all events have been processed, the vcore is free to resume the uthread that was interrupted (if there is one), or call out to the user-level scheduler to decide what to do next. For convenience, the code for `vcore_entry()` is reproduced here for reference.

```
void vcore_entry() {
    handle_events();
    if (current_uthread)
        run_current_uthread();
    sched_ops->sched_entry();
    vcore_yield();
}
```

2.3.1 Akaros vs. Linux

The event delivery mechanism described above was born directly out of the event delivery subsystem we built for Akaros. Because of this, most asynchronous services currently built for Akaros originate in the kernel and post messages to queues set up for them in userspace. For example, syscall completion events and alarm events are both sent by the kernel.

The event delivery subsystem itself provides a rich set of features that allow a user to control basically every aspect of event delivery you could imagine. For example, event queues (i.e. mailboxes in Akaros terms) can be set up as message endpoints on either a global basis, a per-vcore basis, a per-event basis, or any combination thereof. Moreover, each queue can be configured with a set of flags that indicate if-and-how to notify a set of vcores when a producer posts messages to that queue. For example, a user level-scheduler may choose to set a global queue as the endpoint for all syscall completion events, and then indicate that all messages that post to that queue should send notifications to vcore 0 for processing. Alternatively, queues could be set up on a per-vcore basis, and notifications could be directed back to whatever vcore the syscall was originally issued from. Because vcores can appear and disappear arbitrarily in an Akaros MCP, a majority of the features provided by Akaros's event delivery subsystem exist soley to ensure that messages never get lost in the system. Moreover, because this subsystem allows for direct communication between the kernel and userspace, a special type of queue called a UCQ is used to ensure the safety of all communication across the user/kernel boundary. Full details on this subsystem can be found in Chapter 5 of Barret Rhoden's dissertation [2].

On Linux, we don't support all of these features provided by Akaros. In fact, only a single configuration exists for all available services. One queue per-vcore is setup when a

process is first started, and those queues are the only endpoint for all event messages from all services throughout the lifetime of a process. Moreover, all services that send event messages originate in userspace, which allows us to implement these event queues as simple linked lists protected by a lock. A service simply chooses which vcore it would like to send a message to and sends it. All messages are followed by a notification, ensuring that the vcores they are destined for will eventually wake up and process those events. The reason for this limited setup is threefold:

1. Linux vcores can never disappear, making it impossible for messages to get lost in the system due to vcores going offline. Even if a vcore is currently alseep, notifications will ensure that it wakes up to process an event that has been sent to it. Any of the features present in Akaros to deal with these issues are unnecessary on Linux.
2. Akaros's event delivery subsystem is extremely complex, and it's unclear what (if any) benefit would be gained from porting all of its features to Linux. Moreover, whatever features we did port would only be emulated in userspace, limiting their effectiveness.
3. It keeps the interface simple. In practice, this simple configuration seems to work well for all of the use cases we've encountered on Linux so far. We can add more features to it later if needed.

2.3.2 The Event Delivery API

The event delivery API is designed to allow user-level schedulers to gain access to all of the event delivery features described above. In general, these features include setting up queues for event delivery, registering handlers for different event types, enabling / disabling notifications, etc. However, this API differs significantly between Akaros and Linux. On Akaros, only a small shim layer exists, which exposes userspace to the full set of features provided by the underlying system. In total, this API consists of over 20 different library calls, which control everything from enabling events on a particular `event_type`, to registering event queues for that `event_type`, to extracting a single message from an event queue and processing it. Applications (or more appropriately, user-level schedulers) use this API to customize the way in which they receive events on a service-by-service basis. In contrast, the full Linux event delivery API can be seen below:

Global Variables
`ev_handlers[MAX_NR_EVENT];`

Library Calls
`event_lib_init()`
`send_event(ev_msg, ev_type, vcoreid)`
`handle_events()`
`enable_notifs(vcoreid)`
`disable_notifs(vcoreid)`

On Linux, the only configuration necesssary to start receiving events for a particular `event_type` is to register a handler for it in the global `ev_handlers` array. For example, user-level schedulers typically register a handler with the `EV_SYSCALL` event type in order to process syscall completion events sent by the underlying system, i.e:

```
ev_handlers[EV_SYSCALL] = handle_syscall;
```

Once a handler is registered, events can be sent to a specific vcore by calling `send_event()` with the proper set of parameters. In practice, this call is never made by a user-level scheduler itself, but rather by the various services registered with each `event_type`. When calling this function, a service uses the `event_msg` parameter to pass a payload specific to the `event_type` being posted. The handler registered for each `event_type` knows how to parse this payload and process it appropriately. Continuing with the example above, system call completion events are sent as follows:

```
send_event(event_msg, EV_SYSCALL, vcoreid);
```

In this case, the `event_msg` payload will contain a pointer to the syscall that has completed (which in turn contains a pointer to a uthread that has blocked on it). The handler registered for this `event_type` (i.e. `handle_syscall()`) can then use this syscall pointer to unblock the uthread waiting for the sycall to complete (or do whatever it wants).

Contrast this with the code necessary to achieve something similar on Akaros.

```
enable_kevent(EV_USER_IPI, 0, EVENT_IPI | EVENT_VCORE_PRIVATE);
register_ev_handler(EV_SYSCALL, handle_syscall, 0);
sysc_mgmt = malloc(sizeof(struct sysc_mgmt) * max_vcores());
mmap_block = (uintptr_t)mmap(0, PGSIZE * 2 * max_vcores(),
                             PROT_WRITE | PROT_READ,
                             MAP_POPULATE | MAP_ANONYMOUS, -1, 0);
for (int i = 0; i < max_vcores(); i++) {
    sysc_mgmt[i].ev_q = get_big_event_q_raw();
    sysc_mgmt[i].ev_q->ev_flags = EVENT_IPI | EVENT_INDIR | EVENT_FALLBACK;
    sysc_mgmt[i].ev_q->ev_vcore = i;
    ucq_init_raw(&sysc_mgmt[i].ev_q->ev_mbox->ev_msgs,
                 mmap_block + (2 * i    ) * PGSIZE,
                 mmap_block + (2 * i + 1) * PGSIZE);
}
```

As with Linux, a method is called to register a handler for syscall completion events, i.e.

```
register_ev_handler(EV_SYSCALL, handle_syscall, 0);
```

However, much more work goes into setting up the queues to receive these events. Recall that Akaros directs all event messages at queues (not vcores), because more than one queue might be associated with a vcore, and each service sets up its queues independently with different configuration settings, etc. The code above allocates space for these event queues, associates each with a specific vcore, and configures them such that message delivery will occur in a manner similar to Linux (i.e. one queue per core, notifications enabled, etc.). However,

it does nothing to associate the actual syscall completion events with those queues. As explained in Section 2.4.1, this happens on a syscall-by-syscall basis at the time the syscall itself is issued (to avoid sending an event if not actually necessary).

In general, so long as the kernel knows which queue to send an event to, it sends it via its own internal `send_event()` call, i.e.:

```
send_event(proc, event_q, event_msg, vcoreid);
```

As with Linux, the `event_msg` parameter is used to pass a payload specific to the `event_type` being sent. However, this event type is now embedded in the `event_msg` itself, as opposed to being sent as its own unique parameter as is done with Linux. Additionally, since messages are directed at queues instead of vcores, the `vcoreid` parameter is only used as a fallback for deciding which vcore to notify in cases where notifications are enabled, but no vcore is explicitly associated with the event queue. Once this event arrives at its destination, it is processed by its registered handler (i.e. `handle_syscall()`) in a manner similar to how it is done on Linux.

In addition to setting up queues on a service-by-service basis as shown above, Akaros also provides two per-vcore event queues that services can use by default. One queue is for private messages which will only ever be processed by the vcore they were sent to (useful for uthreads to manual drop themselves into vcore context to process events immediately), and the other is for public messages that can be stolen and processed by other vcores if necessary (i.e. because a vcore has been preempted and taken offline). These queues are stored in the `procdata` region of an MCP and are initialized at process startup. The most notable use of these queues is by Akaros's mechanism for sending vcore-to-vcore messages via calls to `sys_self_notify()`, i.e.:

```
sys_self_notify(vcoreid, ev_type, event_msg, priv)
```

This call says send an `event_msg` with a specific `ev_type` to `vcoreid`, using its private event queue if `priv` is `true` and its public event queue if `priv` is false. This call is also useful for "re-notifying" a vcore when it misses a notification event, as discussed below.

2.3.3 Event Notifications

Event notifications are parlib's way of delivering software interrupts to userspace. They operate on the principle that vcores in userspace are analogous to physical cores in the kernel. Hardware interrupts force physical cores to bundle up whatever context they are running and drop into IRQ context to run a predefined IRQ handler from an IRQ table. Likewise, event notifications force vcores to bundle up whatever context they are running and drop into vcore context to handle any outstanding events. They are the userspace counterpart to hardware interrupts in the kernel.

Just as interrupts are disabled when running interrupt handlers in the kernel, so are notifications disabled when running code in vcore context in userspace. This ensures that vcore context is non-reentrant and can be reasoned about as such. Uthreads, on the other

hand, *can* be interrupted by notifications, but functions exist to temporarily enable and disable notifications when necessary. While notifications are disabled, no interrupt will be delivered to the vcore that uthread is running on. However, we keep track of the fact that a notification was attempted, and force a vcore to "re-notify" itself as soon as notifications are re-enabled (if necessary).

On Akaros, notifications are implemented using an interprocessor interrupt (IPI) multiplexed on top of the kernel's routine messaging mechanism. Notifications are directed at vcores, but they cause IPIs to be generated on the physical core that a vcore is currently mapped to. When one of these IPIs comes in, the physical core packages up whatever context was running, drops into interrupt context in the kernel, and schedules a routine message for sending the actual notification. When this routine message runs (usually as a result of simply exiting interrupt context), it first checks to see if notifications are enabled for the vcore or not. If they are, it puts the saved context into a special *notification slot* in `procdata` and jumps to the `vcore_entry()` function of the vcore. From there, `handle_events()` is called, followed by the restoration of the saved context via a call to `run_current_uthread()`. However, if notifications are disabled, it sets a per-vcore `notif_pending` flag to indicate that a notification was attempted, restores the saved context, and continues where it left off.

On Linux, the mechanism is similar except that it uses the `SIGUSR1` signal as a proxy for the IPI generated on Akaros. Likewise, the `SIGUSR1` handler acts as a proxy for the routine kernel message that triggers the actual notification. The `SIGUSR1` signal itself is triggered by calling `pthread_kill()` on the pthread backing the vcore being notified. Once this signal fires, it first checks to see if notifications are enabled or not. If they are disabled, it sets a per-vcore `notif_pending` flag (similar to Akaros) and simply returns from the signal handler (leaving the kernel to take care of restoring the interrupted context for us). However, if notifications are enabled, it has to do something quite different from Akaros to trigger the actual notification (since the entire mechanism is implemented in userspace instead of the kernel). It explicitly calls `uthread_yield()` from within the signal handler to package up the interrupted uthread and manually drop us into vcore context. When this context is later restored, it will simply exit the signal handler normally and continue where it left off. Because we are yielding from within a signal handler, however, the callback we pass to `uthread_yield()` has to do a number of non-standard things to ensure that we are able to properly restore the uthread later:

- It makes a call to `sigaltstack()` to swap out the current pthread's signal handler stack with a new one. Since we yield from within the signal handler, we end up bundling the current signal stack with the uthread context being yielded. This step is necessary so that subsequent signal handlers have a unique stack to run on.
- It manually unmasks the `SIGUSR1` signal for this pthread. Normally, this is done automatically by returning from the signal handler. However, we do not technically return from the signal handler (from linux's perspective) until the uthread that has yielded is restored at some point in the future. Unmasking is safe at this point because notifications are disabled while running this callback in vcore context (meaning we will

simply return to where we left off instead of handling the notification right away if a signal actually does manage to come in).

- It manually calls `vcore_entry()` (which never returns) instead of leaving it up to the uthread library to do this for us upon returning from the callback. This step is necessary to ensure that the uthread that has yielded is still considered *running*, and that the value of `current_uthread` is still set. In essence, this emulates Akaros's process of interrupting the uthread via an IPI and putting its context in the `procdata` notification slot. From here, `handle_events()` is called, followed by the restoration of the saved context via a call to `run_current_uthread()` as per usual.

As mentioned above, both Akaros and Linux set a per-vcore `notif_pending` flag to indicate that a notification was attempted, but not delivered (because notifications were disabled). In general, this flag is used to allow a vcore to "re-notify" itself upon re-enabling notifications, and thus drop back into vcore context to check for more outstanding events. Akaros "re-notifies" by issuing a `sys_self_notify()` on the current vcore; Linux resends the `SIGUSR1` signal.

However, this flag does not necessarily mean that a vcore must always "re-notify" itself in order to process a missed notification. Notifications themselves are only meant to ensure that `handle_events()` is called as soon as possible in order to process any outstanding events. If a vcore is already in vcore context and notices that the `notif_pending` flag is set, it can simply call `handle_events()` and unset the `notif_pending` flag itself. In fact, the body of `handle_events()` actually unsets this flag just before it begins to process events. When returning from vcore context, the mechanism to "re-notify" is used solely to solve the race between calling `handle_events()` while still in vcore context, and deciding to return to uthread context (which automatically re-enables notifications). However, "re-notification" is necessary when explicitly re-enabling notifications from uthread context because the uthread will not have had the opportunity to call `handle_events()` itself.

The function to explicitly enable and disable notifications from uthread context are `enable_notifs()` and `disable_notifs()` respectively. However, these functions are never called directly by uthread code itself. Instead, two other functions exist that wrap these functions for use by uthreads (`uthread_disable_notifs()` and `uthread_enable_notifs()`, respectively). These differ from the regular `enable_notifs()` and `disable_notifs()` calls in two important respects:

1. They ensure that a uthread doesn't migrate off of its vcore between the time that any local copies of `vcore_id()` are made and notifications are actually disabled.
2. They keep track of a "disable depth" so that nesting calls that disable notifications work properly (i.e. you up the depth when you call disable, down the depth when you call enable, and only re-enable notifications once the depth reaches 0).

Within parlib itself, these calls are used by the uthread abstraction to protect critical library calls from being interrupted by notifications (e.g. `uthread_yield()`). They are also

used as a building block for parlib's low-level-lock primitives (in a manner similar to an `irq_save_lock` in the kernel). These locks are discussed in more detail in Section 2.2.3.

2.4 Asynchronous Services

The event delivery mechanism discussed in the previous section provides the foundation for a rich set of asynchronous services to be built on top of parlib. The most important of which is the **asynchronous system call service** discussed below. Additionally, a number of other services exist including general purpose alarms and POSIX signal emulation. Each of these is discussed below as well.

2.4.1 System Calls

Traditionally, I/O-based system calls such as `read()`, `write()`, and `accept()` block in the kernel until their corresponding operations complete. Such a model is acceptable when the kernel is the one responsible for managing all concurrency in the system: it will simply find another thread to run in its place and execute it. However, this model breaks down under parlib, where a thread blocking in the kernel is tantamount to a thread blocking access to an entire core.

To prevent this situation, parlib ensures that all system calls are only ever issued to the kernel in an asynchronous manner. In cases where a system call would normally have blocked, control is immediately returned to userspace, and some external entity is handed the responsibility of completing the syscall at a later time. Once the system call completes, an `EV_SYSCALL` event message is generated and sent to a process via the asynchronous event delivery mechanism described in the previous section. User-level schedulers set up event queues and register an event handler for `EV_SYSCALL` messages, enabling them to respond to these syscall completion events appropriately. Uthreads that wish to "block" on one of these syscalls do so in userspace by packaging up their state and passing it to a user-level scheduler via the `thread_blockon_sysc()` callback. The handler for `EV_SYSCALL` messages then takes care of waking those uthreads once their system calls complete.

Unsurprisingly, the exact method used to support this type of "user-blockable" system call differs significantly between Akaros and Linux. On Akaros, all system calls are inherently asynchronous, making this process relatively straightforward. When issuing a system call, uthreads allocate a special `syscall` struct on their stack, which persists for the lifetime of the system call. The kernel and user-space share access to this `syscall` struct which contains fields for all of the syscall arguments, as well as a field for the return value and a flag indicating if the syscall has completed yet or not. Additionally, it contains a pointer to the event queue to use when posting its syscall completion event.

When a syscall is first issued, the kernel parses its arguments and attempts to complete the syscall immediately. If it completes, it simply sets the return value in the `syscall` struct with the proper value, sets the flag to indicate completion, and returns control to userspace.

If it cannot complete the syscall immediately (e.g. because it needs to wait on some sort of I/O operation), it still returns control to userspace (this time without setting the syscall completion flag), but it also spawns off an internal *kthread* to complete the syscall later.

Once back in userspace, the uthread checks the completion flag to see what it should do next. If the flag is set, the uthread simply extracts the return value from the `syscall` struct and continues. If it is not set, the uthread makes an explicit call to `uthread_yield()`, returning control to the user-level scheduler. The uthread passes `thread_blockon_sysc()` as the callback to `uthread_yield()`, which gives the scheduler the opportunity to associate the uthread with its `sycall` struct so it knows which uthread to wake up once the syscall completes. It also sets the event queue pointer inside the `syscall` struct to an event queue set up previously by the user-level sheduler. At some point later, the spawned kthread completes the syscall and posts an `EV_SYSCALL` event message to this event queue. The process then handles this event inside one of its vcore's `handle_event()` functions, triggering it to resume the uthread blocked on the syscall. Details on the inner workings of this mechanism, including how to avoid races on the value of the syscall completion flag can be found in Sections 3.2 and 4.4 of Barret Rhoden's dissertation [2].

On Linux, the process of ensuring that system calls do not block in the kernel is not as straight forward. We rely on an unmodified version of glibc to issue our system calls, and as such, are limited in our techniques to make them appear asynchronous. In fact, we currently *only* guarantee non-blocking support for a handful of syscalls we explicitly wrap in a special non-blocking syscall wrapper (namely, `read()`, `write()`, and `accept()`). We also wrap the `open()` and `socket()` syscalls with a different wrapper, but only in support of making the `read()`, `write()`, and `accept()` syscalls non-blocking. The general logic behind this scheme is as follows:

1. Intercept calls to `open()` and `socket()` and force them to open their respective file descriptors with the `O_NONBLOCK` flag set.
2. Intercept calls to `read()`, `write()`, and `accept()` (whose file descriptors are now all set as `O_NONBLOCK`), and attempt to issue the original, non-wrapped version of the syscall.
3. If the syscall completes successfully (or errors with an `errno` other than `EWOULDBLOCK`), continue on.
4. Otherwise, voluntarily yield to the user-level scheduler, passing it a callback which both triggers `thread_blockon_sysc` (similar to Akaros) as well as knows how to complete the syscall "out-of-band" (i.e. on a linux pthread other than those being used to emulate vcores). This latter process simulates the operation of completing the syscall in an Akaros kthread.

As mentioned at the end of Section 2.2.2, the "out-of-band" pthread used to service a uthread's syscall is the same pthread created to initialize its TLS. We simply wake this pthread from the futex it is sleeping on and pass it a reference to a structure similar to the Akaros `syscall` struct (the main difference being that it also contains a function pointer to a *differently wrapped* version of the syscall). Once the pthread wakes up, it calls the *differently*

wrapped syscall, which first issues a `select()` call to wait for data on the `O_NONBLOCK` file descriptor, followed by the syscall itself (which is now guaranteed to complete since `select()` was called first). As with Akaros, it then generates an `EV_SYSCALL` event and posts it to the event queue of whichever vcore the syscall originated from. That vcore is then forced into vcore context (if it wasn't already) and handles this event inside its `handle_event()` function, triggering it to reschedule the uthread blocked on the syscall.

Although the uthread and its backing pthread share the same TLS, only one or the other of them will be accessing that TLS at any given time: we only spin up the backing pthread to service a syscall once its corresponding uthread has been blocked in the user-level scheduler. In fact, we *want* the pthread issuing the syscall to have the same TLS as the original uthread because this may have implications when calling into the glibc wrappers around each syscall. By servicing syscalls using the backing pthread, we avoid such problems altogether.

Unfortunately, there is no good way of wrapping *all* potentially blocking system calls in Linux short of modifying glibc or rolling our own stdlib of some sort. Even wrapping the ones we have is not done in a uniform manner. For example, glibc weak aliases `read()`, `write()`, and `open()` to versions of themselves that have the form: `__<func>()`. To wrap these, we simply had to override the `read()`, `write()`, and `open()` symbols and call the `__<func>()` versions of those functions where appropriate. However, `accept()` and `socket()` provide no such weak symbols, forcing us to use the `--wrap` option provided by `ld` instead. The major disadvantage of this approach is that wrappers provided via `--wrap` don't automatically override syscalls inside the parlib library itself. Instead, *applications* have to manually pass the `--wrap` option at link time, indicating which syscalls they would like to wrap and the name of the wrapper function to use. We leave it as future work to figure out how to improve this situation somewhat.

2.4.2 The Alarm Service

Another important service based on parlib's event delivery mechanism is the **alarm service**. This service provides microsecond precision alarms for use within an application. Its API can be seen below:

Library Calls
`init_awaiter(alarm_waiter, callback(alarm_waiter))`
`set_awaiter_abs(alarm_waiter, abs_time)`
`set_awaiter_abs_unix(alarm_waiter, abs_time)`
`set_awaiter_rel(alarm_waiter, usleep)`
`set_awaiter_inc(alarm_waiter, usleep)`
`start_alarm(alarm_waiter)`
`stop_alarm(alarm_waiter)`
`reset_alarm_abs(alarm_waiter, abs_time)`

Individual alarms are created by allocating space for a unique `alarm_waiter` struct and initializing it with a callback to be fired once the alarm expires (using the `init_awaiter()` function listed above). Once initialized, different methods exist to set the expiration time of the alarm, including:

set_awaiter_abs: Set an alarm to expire at an absolute time relative to the time since the machine first booted.

set_awaiter_abs_unix: Set an alarm to expire at an absolute time relative to unix time (i.e. POSIX time or Epoch time).

set_awaiter_rel: Set an alarm to expire "x" microseconds in the future (i.e. relative to the current time).

set_awaiter_inc: Add "x" microseconds to the current time set for the alarm to expire. The value of "x" can be negative if desired.

Once the expiration time is set, the alarm can be started via a call `start_alarm()`, or stopped prematurely via a call to `stop_alarm()`. To both stop the alarm and start it again with a different absolute expiration time, `reset_alarm_abs()` can be used. Once the alarm expires, the callback that the alarm was initialized with will be fired.

Both `stop_alarm()` and `reset_alarm_abs()` return booleans indicating whether the alarm was successfully stopped or reset respectively. This information is useful for keeping track of the exact number of outstanding alarms in the system. If a call to `stop_alarm()` returns `true`, then it is guaranteed that the alarm will not fire and that its callback will never be triggered. If it returns `false`, it is guaranteed that the alarm has either already triggered, is in the process of triggering, or will trigger very soon. In both cases, you have exactly one code path that you can use to keep an exact reference count for the alarm. Resetting is similar except that `true` indicates that the alarm will fire at the new time, and `false` indicates that it will fire at the old time. Either way, you only have a single code path for keeping track of the alarm.

On Akaros, this alarm service is implemented by multiplexing each alarm onto a single underlying hardware timer provided by the kernel. The kernel exposes this timer as a dedicated "alarm device", which userspace can set to fire at some absolute time in the future. Alarm completion is then signaled via the event delivery mechanism described in the previous section with `event_type EV_ALARM`. Inside parlib, the alarm service registers a single event queue on vcore 0 to handle all alarm completion events. It also registers a handler which knows how to demultiplex these completion events and trigger the proper alarm callbacks. One side effect of multiplexing alarms in this way is that alarms started via the API do not map 1-1 with alarms started on the kernel alarm device. More importantly, it means that all alarm callbacks are triggered on vcore 0 rather than the vcore they were originally started from. This has implications on trying to use these alarms for things like preemptive schedulers, which require every vcore to be interrupted at some periodic interval.

For this we provide a special "per-vcore-alarm service" (i.e. pvc-alarms), which allows you to specify which vcore you would like an alarm callback to fire on. Under the hood, this service attaches each vcore to a separate instance of the kernel alarm device, allowing it to multiplex alarms destined for each vcore separately.

On Linux, we took a simpler approach to implementing the alarm service. When an alarm is started, we simply launch an "out-of-band" pthread to act as a proxy for the kernel alarm device and call `usleep()` on it. When the `usleep()` returns, we post an alarm completion event to whatever vcore was used to start the alarm. Although less efficient, this approach has the nice property that every alarm started via the API maps 1-1 with each completion event generated by the service. There is no multiplexing and demultiplexing going on, and alarm completion events are always directed back to the vcore they originated from. This avoids the problems mentioned above that motivated the creation of the pvc-alarm service on Akaros. To make thing slightly more efficient, we actually use a pool of pthreads sleeping on futexes rather than launching a brand new pthread each time an alarm is started. In the future we could also imagine using actual linux alarms instead of our pthread proxy, but so far this hasn't proven necessary.

2.4.3 POSIX Signal Support

POSIX signals provide a standardized means of performing inter-process communication on POSIX compliant operating systems [39]. Moreover, they provide a mechanism for the operating system to directly notify a process of some outstanding condition. Examples of POSIX signals include `SIGHUP`, `SIGINT`, `SIGKILL`, and `SIGSEGV`. On Linux, POSIX signal support is built directly into the OS. On Akaros, signals are superceded by the existence of its unified event delivery subsystem. Akaros does not support the POSIX signal API directly.

However, many applications that we would like to support on Akaros are built to rely on the existence of POSIX signals (most notably bash and Go). In theory, we could rewrite these applications to run on top of events instead, but doing so would discourage portability and would add (arguably unnecessary) maintenace costs down the road. Because of this, parlib on Akaros includes a generic signaling service that a user-level scheduler can use to provide POSIX signal support to an application. This service provides:

- Default handlers for all 64 POSIX signals
- Standard signal(), sigaction(), and sigmask() API calls for use within an application
- A method of automatically triggering a signal handler for signals generated as part of inter-process communication
- A `trigger_posix_signal()` function to explicitly trigger a POSIX signal handler from within a user-level scheduler for all other types of signals

In general, a user-level scheduler uses the signal service to trigger POSIX signals in each the following scenarios:

1. After receiving an EV_POSIX_SIGNAL event from the kernel. This event is designed for sending inter-process signals over Akaros's event delivery subsystem (e.g. via the kill command). The signal library itself takes care of registering event queues for this event_type and will automatically trigger any POSIX signal handlers as these events come in. The actual POSIX signal number associated with each event is passed in the event_msg payload.
2. After receiving a hardware fault from the kernel. In Akaros, all hardware faults are reflected to user-space. This gives them the chance to handle the fault before the kernel takes some default action (usually just kill the process). As discussed in Section 2.2.2, a user-level scheduler is notified about these faults via the thread_refl_fault() callback. Using this callback, a scheduler can trigger a fault's equivalent POSIX signal handler using the trigger_posix_signal() function mentioned above.
3. After receiving a POSIX signal directly, using some scheduler specific intra-process signaling mechanism (e.g. pthread_kill()).

For inter-process signaling via the EV_POSIX_SIGNAL event, no specific uthread is the target of the signal being generated. For faults and intra-process signals, however, a specific uthread is associated with the signal (i.e. faults occur because a uthread has performed some illegal operation, and intra-process signals have a specific uthread to which they are targeted). A user-level scheduler must be careful to distinguish between these two cases.

In general, the details of how to use the POSIX signal service are left up to user-level scheduler writers themselves. However, it is instructive to walk through an example of how to use this service, since there are many pitfalls you must be aware of. For example, our user-level pthread scheduler on Akaros chooses to run handlers for EV_POSIX_SIGNAL events and faults as soon as possible, but wait to run signal handlers triggered by pthread_kill() until the next time their target uthreads are scheduled.

Originally, we thought we'd be able to provide these semantics by simply triggerring the posix signal handlers to run from vcore context at the appropriate time. When an EV_POSIX_SIGNAL event or fault handler came in, we were already in vcore context and would just trigger the signal handler directly (restarting the offending uthread afterwards in the case of a fault). If a signal was sent via pthread_kill(), it was registered in a bitmask on its target uthread, and the handlers for all pending signals were ran in a loop just before the next time that uthread was scheduled to run again.

Triggering signal handlers in this way makes intuitive sense, because vcore context already has an alternate stack for the handlers to run on. However, there are a couple of problems with this approach. First, signals sent to a particular uthread typically assume their handlers have access to the thread-local variables of that thread. Running from vcore context causes these handlers to access the wrong thread-local variables since they are running with the vcore's TLS (not the uthread's). Second, signal handlers may make calls that attempt to yield. Yielding while running on the vcore stack can cause memory corruption, since the stack will be reused by the vcore as soon as the yield completes.

For the case of an EV_POSIX_SIGNAL event, we were able to address these problems by

simply setting up a transient uthread to run the handler code. We don't have to worry about TLS because there is no particular uthread associated with the signal, and, as a uthread, it can be yielded and restarted as many times as it likes.

For signals triggered by faults or `pthread_kill()`, we tried a couple of different approaches before settling on a working solution. First, we tried to simply switch into the target uthread's TLS and jump to an alternate stack before calling each signal handler. This approach seems reasonable on the surface, but problems arise because we don't properly exit vcore context before running the handlers. Specifically, notifications are still disabled, and proper synchronization with vcore preemption is not preserved.

The correct solution was to create a new user context for the signal handler to run in, and restart the uthread with this context instead of its original one. In this way, the uthread's TLS is preserved, but we are able to set the context up with a new stack and point the program counter directly at the signal handler. The signal handler, in essence, hijacks the uthread, so it can access its TLS variables and yield freely as if it were the original uthread itself. Once the signal handler completes, the original uthread context is restored, and rescheduled to run.

2.5 Other Abstractions

Up to this point, all of the abstractions discussed have been essential to the development of parlib itself. Without them, parlib would not be able to provide a fully featured parallel runtime development framework based on user-level scheduling. However, parlib also provides a number of other tools, which are not as fundamental, but provide useful abstractions that developers may choose to integrate into their parallel runtimes if desired.

2.5.1 Memory Allocators

In addition to the default memory allocators provided by glibc (i.e. `malloc`, `posix_memalign`, etc.), and the `mmap` capabilities provided by the kernel, parlib provides a few special-purpose memory allocators that developers may wish to take advantage of. Specifically, it provides a general-purpose **slab allocator** [40] as well as a specialized fixed-size **pool allocator**.

Both of these memory allocators are designed to reduce memory fragmentation in cases where fixed-size objects are continually being allocated and freed (but not reinitialized). Allocating and freeing from these allocators is fast because all memory blocks are the same size. They simply maintain a list of free blocks and allocate them instantly upon request. Moreover, object initialization only needs to occur the first time a block is allocated. From then on the object can be used immediately. Examples of such objects include thread structures, event messages, TLS data, etc.

The primary difference between the two approaches is that the size of the slab allocator can grow dynamically, while the size of the fixed-size pool allocator cannot. For the slab allocator, you simply initialize an instance of it with the size of the objects you would like

to allocate, and it takes care of dynamically growing the memory required to back those allocations over time. With the fixed-size pool allocator, enough memory to satisfy `N * sizeof(object)` allocations is requested from the system at initialization time and no more memory will be added to satisfy allocations beyond those limits.

A fixed-size pool allocator has the potential to waste more memory than a slab allocator if outstanding allocations are significantly smaller than the size of the preallocated pool. Moreover, a fixed-size pool will not be able to handle a large number of real-time allocations that grow beyond the size of the preallocated pool. Slab allocators are able to dynamically compensate for each of these situations at the cost of more complicated code (i.e. longer, more time consuming code paths), and the need to periodically make expensive calls to the system to add more backing memory if the number of real-time allocations gets too large.

Currently, we use slab allocators within parlib for maintaining data associated with the Dynamic TLS (DTLS) abstraction discussed in Section 2.5.3. On Linux, we also use them to allocate and free job structures associated with the "out-of-band" pthreads used by the alarm service discussed in Section 2.4.2. Pools were used at one time in various ways, but are no longer actively used anywhere.

2.5.2 Wait-Free Lists

Parlib provides a simple wait-free-list abstraction (WFL), with semantics similar to that of a *concurrent bag* [41]. The interface to the WFL can be seen below:

Library Calls
`wfl_init(list)`
`wfl_cleanup(list)`
`wfl_insert(list, data)`
`wfl_insert_into(list, slot, data)`
`wfl_remove(list)`
`wfl_remove_from(list, slot)`
`wfl_remove_all(list, data)`
`wfl_capacity(list)`
`wfl_size(list)`

Traditional wait-free data structures are complicated by the desire to preserve some notion of ordering, prevent concurrent list mutatations, and manage the memory of their internal data structures on the fly. The WFL relaxes these constraints, allowing for a simple, high-performance data structure, at the cost of no guaranteed ordering, a more restrictive consistency model, and a little wasted memory. Items themselves are inserted and removed from a WFL atomically, but no guarantee is made that a removal will find an item if it is inserted concurrently.

The WFL abstraction is useful anytime you need to maintain a cache of similarly typed objects, but you don't care which particular object you get back when you pull from that

cache. We currently use WFLs as the basis for thread pools and other data structures inside our pthread scheduler, as well as for internal data structures maintained by our POSIX signal support library and futex abstractions.

Internally, a WFL is implemented as a simple linked list of slots capable of holding arbitrary data items. Empty slots contain a NULL pointer, and occupied slots contain a pointer to the item they are holding. Whenever a new item is added to a WFL, it is either atomically inserted into a pre-existing NULL slot, or space for a new slot containing the data item is allocated and atomically appended to the end of the list. Whenever an item is removed from a WFL, the list is searched, and the element from the first non-empty slot is atomically removed and returned. If no item is found, we return NULL.

Once created, a slot is never freed until `wfl_cleanup()` is called and the entire WFL is destroyed. In practice, the number of slots tends to remain low, and the fact that any new slots are only ever appended to the end of the list means that a simple CAS loop is sufficient to add them. Additionally, since item insertion and removals are not synchronized, they can each be performed using a simple `atomic_swap()`.

Although not the normal usage of a WFL, the API also provides hooks to let you insert and remove items from a particular slot in a WFL(`wfl_insert_into()` and `wfl_remove_from()`, respectively). References to these slots are obtained via the return value to the normal `wfl_insert()` and `wfl_remove()` calls. At one time we planned to use these hooks to help implement per-vcore runqueues without the use of locks, but this idea has since been abandones, and we don't currently use this functionality anywhere at the moment.

2.5.3 Dynamic TLS

Similar to the posix `pthread_getspecific()` and `pthread_setspecific()` API calls, parlib provides a Dynamic-TLS (DTLS) abstraction for managing thread-local storage at runtime. Unlike `pthread_getspecific()` and `pthread_setspecific()`, which are designed specifically for the POSIX pthread scheduler, the DTLS abstraction is designed as a building block for providing runtime-managed TLS by custom user-level schedulers. The interface to the Dynamic-TLS abstraction can be seen below:

Library Calls
`dtls_key_create(destructor)`
`dtls_key_delete(key)`
`set_dtls(key, storage)`
`get_dtls(key)`
`destroy_dtls()`

Using DTLS, multiple software components can dynamically set up unique thread-local storage for use within a thread. Each software component is assigned a unique key via a call to `dtls_key_create()`, and individual threads use this key to access the thread-local storage associated with that component. The size and layout of each component's thread-local

storage is typically defined inside a struct, which individual threads must allocate memory for before making any DTLS API calls. A thread calls `set_dtls(key, storage)` to associate newly allocated thread-local storage with a key and `get_dtls(key)` to retrieve it.

DTLS is mosly provided for legacy code that relies on dynamically allocatable thread-local storage. It is generally accepted that statically allocated TLS with integrated compiler support (e.g. __thread in gcc) is more performant than runtime allocated TLS. Currently, the DTLS abstraction is used by our user-level pthread implementation to implement its `pthread_getspecific()` and `pthread_setspecific()` API calls. Additionally, our Lithe port of Intel's Thread Building Blocks library (TBB) uses it inside its custom memory allocator as well across several other components (as discussed in the following chapter).

When adding DTLS capabilities to a custom user-level scheduler, the API calls they provide should be able to just wrap most of the core DTLS calls directly. Care must be taken, however, to ensure that each thread calls `destroy_dtls()` just before exiting. Once called, all internal memory associated with DTLS for the exiting thread will be freed, and each *destructor* callback passed as a parameter to `dtls_create()` will be executed. A scheduler can either dictate that threads must make a call to `destroy_dtls()` themselves, or embed a call to it inside each thread's exit function in the scheduler. In our ports of pthreads and TBB, we take the latter approach.

2.6 Evaluation

Collectively, the abstractions presented above constitute a complete framework for developing efficient parallel runtimes based on user-level scheduling. The primary goal of these abstractions is to allow an application to take control of its own thread scheduling policy, rather than leave all scheduling decisions up to the kernel. These thread scheduling policies should be both easy to develop and high-performant. In this section, we evaluate the ability of these abstractions (and parlib as a whole) to meet this primary goal.

As mentioned at the beginning of this chapter, parlib was born out of the need to provide a user-space counterpart for the abstractions we built into Akaros; the Linux port came later. As such, many of these abstractions were designed to provide functionality lacking in traditional systems (including Linux). For example, the vcore abstraction and event delivery mechanism map 1-1 with abstractions provided directly by the Akaros kernel. Linux is not able to provide direct support for these abstractions and relies on workarounds to provide a similar API as Akaros. However, these "fake" abstractions only work well under a limited set of assumptions (e.g. running only one application on the system at a time, using a limited set of system calls, etc.).

Despite these limitations, most of our experiments to evaluate parlib have been conducted exclusively on Linux. Ideally, we would run all of our experiments on Akaros and only use the Linux port of parlib as a point of comparison. However, Akaros is still a relatively young system, and as such, has not yet been optimized for high performance. Because of this, it's often hard to distinguish whether a particular negative result can be atrributed to a problem

with parlib or to an artifact of a poorly performing system underneath. The infrastructure is there, but many of Akaros's subsystems still need quite a bit of work before they come close to providing the same performance as Linux. For example, Akaros's networking stack has scalability problems for a large number of connections and is generally much less performant than the Linux networking stack. As we show in Section 2.6, a webserver built on top of parlib performs quite well on Linux, but very poorly on Akaros due to this substandard networking stack. By running most of our experiments on Linux, we are able to isolate parlib more effectively and evaluate it independent of the problems that exist in disjoint parts of Akaros. However, this comes at the cost of only evaluating parlib under the limited setup dictated by Linux's "faked" abstractions. For now, this is sufficient.

The rest of this section goes into the details of the exact experiments we use to evaluate parlib. All of these experiments depend on a reference implementation of the POSIX thread API we built on top of parlib. We call this library `upthread` (short for user-level pthreads). Additionally, the final set of experiments rely on a thread-based webserver we wrote, which integrates a custom user-level scheduler directly into the application. We call this webserver kweb. The details of the `upthread` library and `kweb` are dicusssed below, followed by the details of the experiments themselves.

2.6.1 The upthread Library

The upthread library is our canonical example of how to build a user-level threading package on top of parlib. It provides a version of the standard POSIX thread API, but replaces all instances of `pthread` in the API with instances of `upthread` (i.e. `upthread_create()`, `upthread_join()`, etc). Ideally, we would provide the pthread API directly, but were forced to make this name change for the reasons outlined in Sections 2.2.1 and 2.2.2. However, we do provide a header file that `#defines` the `upthread` names back to their `pthread` counterpart. This allows applications and libraries written against the standard pthreads API to, at least, re-compile against upthreads without modification. In this way upthreads is API compatible with pthreads, but not ABI compatible. However, not all functions from the API are available (e.g. those related to canceling and some of the lesser used attribute functions), but a non-trivial subset are supported. The salient features of the upthread library include:

- The inclusion of per-vcore run queues with the ability to steal from other queues when a local queue falls empty. All queues cycle through their available threads in a round-robin fashion.
- The ability to set the scheduling period of the library via an extended API call. All vcores are interrupted at this scheduling period and forced to make a new scheduling decision.
- Integration with the high-level synchronization primitives discussed in Section 2.2.3. These form the basis for the `mutex`, `barrier`, and `condition variable` interfaces that are part of the standard pthread API. We also provide support for `futexes`.

- Integration with the syscall service discussed in Section 2.4.1. This allows us to properly handle threads which have "blocked" in userspace and are waiting for a syscall to complete. While those threads wait, we are free to schedule other threads in their place.
- A rudimentary affinity algorithm, whereby threads will always be added back to the vcore queues they were previously assigned to whenever they are yielded.
- Careful cache alignment of all data structures associated with vcores and pthreads. All per-vcore data is allocated in a single array and cache aligned at the bottom of a page. All memory for an individual pthread (ie. its stack, uthread struct, etc.) is allocated in one big chunk and cache aligned at the top of a page with a random offset of `rand()` % `max_vcores()` * `ARCH_CACHE_LINE_SIZE`. Aligning to the top of the page minimizes cache interference with the per-vcore data at the bottom, and adding a random offset minimizes cache interference with other pthreads (especially if they all run the same function and hit their stacks at the same place at the same time).

In addition to the standard pthreads API, we also provide a few custom API calls for the upthread library:

upthread_set_sched_period: This function takes a `period` parameter in microseconds and uses it to set a scheduling period for the upthread library. Every `period` microseconds, every online vcore will be interrupted and forced to make a new scheduling decision. This function can be called at any time, but each vcore will only adjust its preempt period the next time it drops into vcore context. This can result in some amount of skew across vcores. The preempt period is set to 0 by default, indicating that no alarms should be set and no preemption should occur (i.e. it is configured for cooperative scheduling only).

upthread_can_vcore_request: This function takes a boolean indicating whether automatic requesting and yielding of vcores should be enabled or disabled. With this feature enabled, the upthread library runs a basic algorithm to decide when to request more cores when new work becomes available (i.e. request a vcore whenever a new thread is created or unblocked in some manner). Moreover, it yields vcores back to the system when it detects that a vcore has no more work to do (i.e. its run queue falls empty). With this feature disabled, vcores can only be explicitly requested and yielded by an application, and will sit in its `sched_ops->sched_entry()` loop waiting for work to become available. This feature is *enabled* by default.

upthread_can_vcore_steal: This function takes a boolean indicating whether or not one vcore is allowed to steal threads from another vcore's run queue when their run queue falls empty. As part of stealing, the preferred affinity of a thread is updated to its new queue. The work stealing algorithm follows a simple procedure where it picks two queues at random and then chooses the queue that has the most number of threads in it. It then steals half of the threads in that queue. If both of the queues it chooses are empty, it cycles through all queues until it finds one that has threads in it. It then steals half of that one. No guarantee

is made for stealing threads that become available concurrent to the run of the algorithm: it simply gives up after one complete pass. This feature is *enabled* by default.

upthread_set_num_vcores: This function takes an integer number of vcores (`nr_vcores`) to use as a hint for the total number of vcores an application plans to have once it is up and running. This feature is useful for precisely allocating threads to vcores before you've actually put in a vcore request. By default, the upthreads library will cycle through all *online* vcore queues as new threads are created and need to be added to a queue. With this feature, it will cycle through `nr_vcores` queues instead (including vcores that are still offline). This feature is usually used in conjunction with `upthread_can_vcore_request(FALSE)` and `upthread_can_vcore_steal(FALSE)`, in order to have better control of all thread placement. The default value of `nr_vcores` is `max_num_vcores()`.

The upthread library provides the basis for all of the experiments discussed throughout this section. For most of these experiments we compare the performance of pthread-based applications running on upthreads vs. running on the Linux NPTL. Additionally, we have hooked our upthread library into versions of `libgomp` and TBB, allowing OpenMP and TBB-based applications to run in top of upthreads as well. All of these experiments are discussed in detail in Section 2.6.3.

2.6.2 The kweb Webserver

Kweb is thread-based webserver we wrote to test parlib's ability to integrate custom user-level schedulers directly into an application. Different variants of kweb can be compiled that allow us to hook in either the Linux NPTL, upthreads, or an application specific scheduler that optimizes the thread scheduling policy specifically for webserver workloads. Additionally, kweb is written such that it can be compiled for both Linux and Akaros. Akaros obviously doesn't provide an implementation of the Linux NPTL, but its pthread library is similar to upthread, and its application-specific scheduler runs the same logic as on Linux.

Like many thread-based webservers, kweb uses a single, dedicated thread to accept incoming TCP connections and a pool of worker threads to service those connections. However, kweb does not maintain a traditional "one-thread-per-connection" model. Instead, it uses a pool of threads to process multiple requests from the same connection concurrently. Paul Turner from Google comments on the superiority of such a model in his talk at the Linux Plumbers Conference in 2013 [42].

Under this model, the accept thread sits in a tight loop: accepting new connections, enqueueing them at the tail of a connection queue, and waking up worker threads to service them. Once awake, a worker thread pulls a connection off of the connection queue, extracts a single request from it, and then re-enqueues the connection for another thread to process. Sometimes they re-enqueue connections at the head of the queue, and other times at the tail, depending on the value of a configurable "burst-length" setting. We set the burst-length to 10 by default.

In addition to the basic logic required to run a webserver, kweb also provides a `url_cmd` interface, which lets you run custom commands inside the webserver by accessing a specific url. These commands are useful for externally triggering the webserver to take some action without needing to restart the webserver itself. These commands exist purely for research purposes and would never be included in a real webserver. Examples include the ability to request or yield a vcore, start and stop debug output, and terminate the webserver completely.

When running on the Linux NPTL or upthreads, most of kweb's application logic is identical, with a few configuration differences to account for setting timeouts on socket reads and writes. Additionally, upthreads turns off preemptive scheduling and explicitly sets `upthread_can_vcore_request(FALSE)` and `upthread_can_vcore_steal(TRUE)`. It does this to ensure that all of the vcores it requests stay online at all times, but can steal from each other's work queues if theirs become empty. In both of these setups, the main thread acts as the accept loop and all worker threads are pulled from a centralized thread pool to process incoming requests. The accept loop places connections onto a global connection queue and pulls a worker thread from the thread pool to start processing them. When a worker thread starts up, it grabs a connection from the connection queue, extracts a request from it, and then re-inserts it back into the connection queue (as described above). As part of the re-enqueue process, it pulls another thread from the thread pool and wakes it up. In this way, we ensure that there is always at least one thread awake to process requests so long as connections still exist in the connection queue. When a thread is done processing a request, it simply re-inserts itself into the thread pool and waits to be woken up again. All threads (including the accept thread) are scheduled according to the policy of the underlying threading library they are running on.

The logic for the application specific scheduler is somewhat different. First, only even numbered physical cores are ever used to run any application logic. This keeps hyperthreaded pairs from potentially interfering with one another. As vcores are requested, odd numbered cores may still trickle in, but they are immediately halted in user-space and a new request is put in (hopefully returning a vcore that maps to an evenly numbered physical core this time). The main thread is still responsible for the accept loop, but it is pinned to vcore 0 and never migrated from it. Furthermore, this thread is the *only* thread that ever runs on vcore 0: all worker threads are spread out among the other cores. When a new connection comes in, the accept thread places it into a *per-vcore* connection queue, and a thread is pulled from a *per-vcore* thread pool in order to service it. Connections are assigned to vcores in a round-robin fashion according the order in which they arrive. On a given vcore, multiple threads may be used to service requests from a single connection concurrently, but they are never migrated to other vcores for processing. Because of this, no locks are needed to protect operations performed on behalf of a request, and it's likely that threads will maintain better cache affinity since they are never migrated.

In Section 2.6.3 we discuss an experiment that compares the performance of all of these variants of kweb to a highly optimized instance of `nginx` [43] running on Linux. For both upthreads and the application specific scheduler, we adjust the number of vcores used to

process incoming requests with a special `vcore_request` url command. In the future, we plan to update kweb to adjust its vcore counts automatically for these schedulers based on real-time demand.

2.6.3 Experiments

In this section we discuss the exact experiments used to evaluate parlib. These experiments attempt to evaluate parlib as a whole, rather than evaluate each of its individual abstractions in isolation. To do this, we rely on the `upthread` library to provide a direct comparison between running applications on a parlib-based thread scheduler vs. running them on the Linux NPTL. Additionally, we use kweb to evaluate the impact of integrating a parlib-based scheduler directly into an application. Collectively, these experiments measure the ability to run both compute-bound and I/O-bound workloads on top of parlib. The experiments themselves are listed below:

Context Switch Overhead: In this experiment, we measure the overhead of performing a context switch with upthreads vs. the Linux NPTL on a varying number of cores. The goal of this experiment is to see how much faster context switches on upthreads can be performed, as well as see a breakdown of the various components that make up each context switch. We vary the number of cores to see if performing context switches on one core have any effect on simultaneously performing context switches on another core.

Flexibility in Scheduling Policies: In this experiment, we compare the performance of the Linux NPTL to different variations of our upthread library when running a simple compute-bound workload. Each upthread variant differs only in the frequency with which it makes a new scheduling decision. The goal of this experiment is to show the effect of varying the scheduling frequency, as well as demonstrate the flexibility in scheduling policies that parlib allows. Although we only vary the scheduling frequency here, these results demonstrate the overall flexibility of changing the underlying scheduling policy itself.

NAS Benchmarks: In this experiment, we measure the impact of running the full set of OpenMP benchmarks from the NAS Parallel Benchmark Suite [44] on top of upthreads vs. the Linux NPTL. The goal of this experiment is to show that parlib-based schedulers are robust enough to run well established benchmarks, rather than just the simple benchmarks we construct ourselves. Moreover, this experiment demonstrates that upthreads is capable of being integrated successfully with a general-purpose parallel runtime, i.e. OpenMP.

Kweb Throughput: In this experiment we measure the impact of running kweb on top of the Linux NPTL, upthreads, and a custom user-level scheduler built directly into kweb. Specifically, we measure the maximum achievable throughput in each of these setups (in requests/sec) when presented with a throughput-oriented workload. We also compare these results to running kweb on Akaros, as well as to running an optimized version of `nginx` on Linux. The goal of this experiment is to show how well parlib-based schedulers perform in

the face of I/O-bound workloads. Additionally, we demonstrate the impact of integrating a scheduler directly into an application, rather than relying on a predefined underlying threading library.

All of these experiments have been run on a machine with the following specifications (here on out referred to as c99):

- 2 Ivy Bridge E5-2650v2 8-core 2.6Ghz CPUs (16 cores, 32 hyperthreads total)
- 8 16GB DDR3 1600Mhz DIMMs (128GB total)
- 4 Intel i350 1GbE interfaces
- 1 1TB Hard Disk Drive

Additionally, all experiments were performed on an x86_64-based Linux kernel. Specifically, we used the 3.13.0-32-generic kernel with a set of patches provided by Andi Kleen [45] to enable calling the x86 `rdfsbase` and `wrfsbase` instructions from userspace. Although not yet enabled in Linux by default, these instructions are essential for performing high-speed, user-level context switches on x86_64 based hardware. Since all of our experiments are run on this hardware, we provide an overview of why these instructions are so important below. Thereafter, we discuss the results of each of the experiments themselves.

Swapping TLS on x86

As mentioned briefly in Section 2, the introduction of the `rdfsbase` and `wrfsbase` instructions mark a significant milestone toward realistic support for user-level threading on x86 platforms. These instructions provide a fast path in user-space for reading and writing the base address of the fs-segment register on an x86 cpu. Modern (64-bit) x86 compilers use the base address of the fs-segment register to provide thread-local storage (TLS) capabilities to an application. The base address of a TLS region maps 1-1 with the base address of the fs-segment register. Since part of the work in swapping threads involves swapping TLS, speeding up access to the base address of the fs-segment register translates to a direct speedup in thread context switch times. These instructions are available in all new and upcoming x86 cpus (ivy-bridge and beyond) [46].

Nowadays, TLS is so ingrained in libraries such as glibc, that not supporting it would prohibit applications from using these libraries effectively. For example, the low-level-locks in glibc rely on TLS to keep track of the current lock holder and protect access to every reentrant library call (e.g. `malloc()`, `read()`, `write()`, etc). If a user-level threading package did not provide TLS it would either have to roll its own locks to protect access to these calls at the application level, or modify glibc to change the low-level lock implementation to not rely on TLS. Moreover, things like errno are stored in TLS, and many applications make use of TLS variables themselves.

Previously, the only way to access the base address of the fs-segment register was with a privileged cpu instruction. This model is fine so long as the kernel is the only one in charge of reading or writing this address. However, this model breaks down as soon as you

have user-level threading packages that wish to read and write this address as well. As a workaround, Linux provides the `arch_prctl()` system call, which gives userspace this ability, but at the cost of making a system call for every read or write.

The table below shows the cost of calling the `rd/wrfsbase` instructions directly, compared with using the `arch_prctl()` system call. We measured these costs by running each operation in a loop for 10 seconds and calculating the average *cycles/operation* over that interval. The measurements were taken on a single core of c99 (core 0). They have been normalized to remove the overhead associated with the loop variable required to keep track of how many times the operation was performed.

	rd/wr fsbase instruction	arch_prctl()
Write TLS base addr	8 ns 23 cycles	395 ns 1025 cycles
Read TLS base addr	2 ns 6 cycles	387 ns 1005 cycles

Table 2.1: Time to read and write the TLS base address via different methods on x86_64

As you can see, the speed of writing the base address is $45x$ faster via the `wrfsbase` instruction, and the speed of doing a read is $167x$ faster via the `rdfsbase` instruction. These numbers are not surprising given that every call to `arch_prctl()` incurs the cost of a system call. The write is slower than the read, because it is a serializing instruction and thus stalls while flushing the cpu pipeline between subsequent writes. These results demonstrate just how important it is to have access to the `rd/wrfsbase` instructions for building user-level threading packages on x86 platforms.

It is worth pointing out that the need to introduce the `rd/wrfsbase` instructions is a *highly* x86 specific problem. Other architectures, such as ARM and RISC-V simply dedicate a register for holding the TLS base address (i.e. `CP15` and `tp` respectively). Swapping out the TLS is then a simple `load` operation as with any other general-purpose register. x86 chose not to adopt this model because of the limited number of general purpose registers they have available.

Additionally, not *every* user-level threading package actually requires TLS capabilities (Go being a very noteworthy example). However, parlib's reliance on glibc, and our desire to support generic applications written against glibc, force us to provide proper TLS support for any threading packages built on top of parlib.

Context Switch Overhead

In this experiment, we measured the overhead of performing a context switch with upthreads vs. the Linux NPTL on a varying number of cores. The goal of this experiment was to see how much faster context switches on upthreads could be performed, as well as see a breakdown of the various components that make up each context switch. We varied the number of cores to see if performing context switches on one core had any effect on simultaneously performing

context switches on other cores. Additonally, we varied the number of threads/core to see if this had any affect as well.

We set up the experiment to run either 1, 2, 4, 8, 16, 32, or 64 threads/core, with the number of cores ranging from 1 to 32. Each thread sat in a tight loop, constantly yielding control back to the scheduler and counting the number of times it performed a context switch. The experiment ran for a duration of $5s$, and at the end, we summed the total number of context switches performed across all threads on all cores. We then took this sum and divided it by the duration of the experiment to calculate the total number of context switches/second that were performed. Additionally, we took the inverse of this value and divided it by the number of cores to calculate the average context switch latency/core.

For the Linux NPTL, we pinned threads to cores via `sched_setaffinity()`, and then ran the experiment as described above. For upthreads, we set up per-vcore run queues and placed the appropriate number of threads into each queue before starting the experiment.

For upthreads, we also ran this experiment multiple times to get a breakdown of the different components that make up a full context switch. The overhead of a full context switch is defined as the time from which `pthread_yield()` is called on one thread, to the time that `pthread_yield()` returns on a newly scheduled thread. We identify the components that make up a full context switch below.

- **save-fpstate:** The cost of saving the floating point state of the current thread and restoring the floating point state of the thread we switch into.
- **swap-tls:** The cost of swapping the TLS, which involves little more than calling `wrfsbase` with the TLS base address of the thread we are switching into.
- **use-queue-locks:** The cost of acquiring and releasing a `spin_pdr_lock` to access one of upthreads's per-vcore run queues. This cost represents the uncontended case of acquiring the lock every time.
- **handle-events:** The cost of running the `handle_events()` function upon dropping into vcore context. This cost represents the case when there are no events to handle.
- **raw-ctxswitch:** The cost of doing an entire context switch, minus each of the components defined above. This includes the overhead in calling `pthread_yield()`, choosing a new thread to run, saving and restoring all general-purpose registers, etc.

Figure 2.6 shows the average context switch latency/core for 1, 8, 16, and 32 cores (i.e. a single core, one full socket without hyperthreading, both sockets without hyperthreading, and the full machine with hyperthreading). For upthreads, the components of the stacked bar graph represent the different components that make up a full context switch, as listed above. For the Linux NPTL, the different components of the stacked bar graph represent the cost of context switching only a single thread in and out of the kernel and context switching between two distinct threads. We distinguish between these two cases because the Linux NPTL provides a fast path when it detects that it doesn't need to actually perform a full context switch. Parlib provides a similar fast path, but it has been disabled for the purpose of this experiment, since the multiple thread case is more interesting.

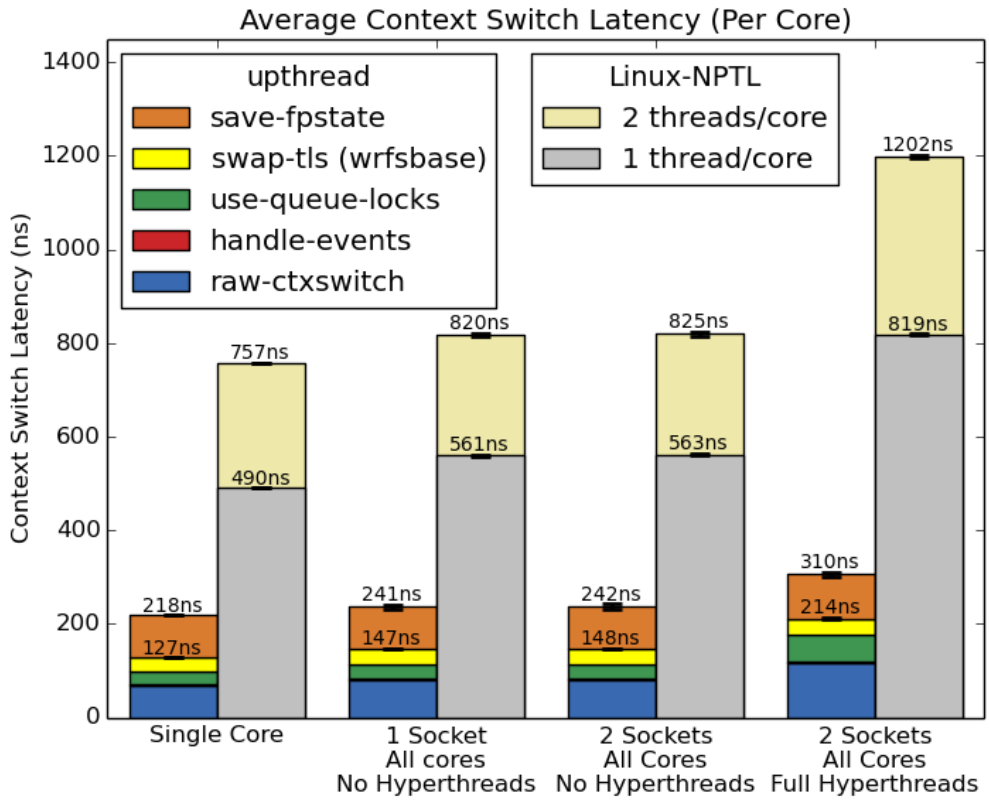


Figure 2.6: Average context switch latency/core of upthreads vs. the Linux NPTL. For upthreads, we show the various components that make up the total context switch latency. For the Linux NPTL, we show the latency when context switching a single thread vs. two. Error bars indicate the standard deviation of the measurements across cores. In all cases, this standard deviation is negligible.

The major takeaways from this graph are:

- Without hyperthreading, context switches in upthreads are $3.4x$ faster than context switches in the Linux NPTL.
- With hyperthreading, context switches in upthreads are $3.9x$ faster than context switches in the Linux NPTL
- Saving the floating point state adds a significant overhead to the overall context switch time in upthreads.
- The cost of calling the `handle_events()` function is negligible when there are no events to process.
- The cost of having queue locks is noticable in the context switch overhead, even when these locks are uncontended.
- The cost of performing a tls swap via `wrfsbase` is consistent with what we measured previously ($\sim 10ns$).

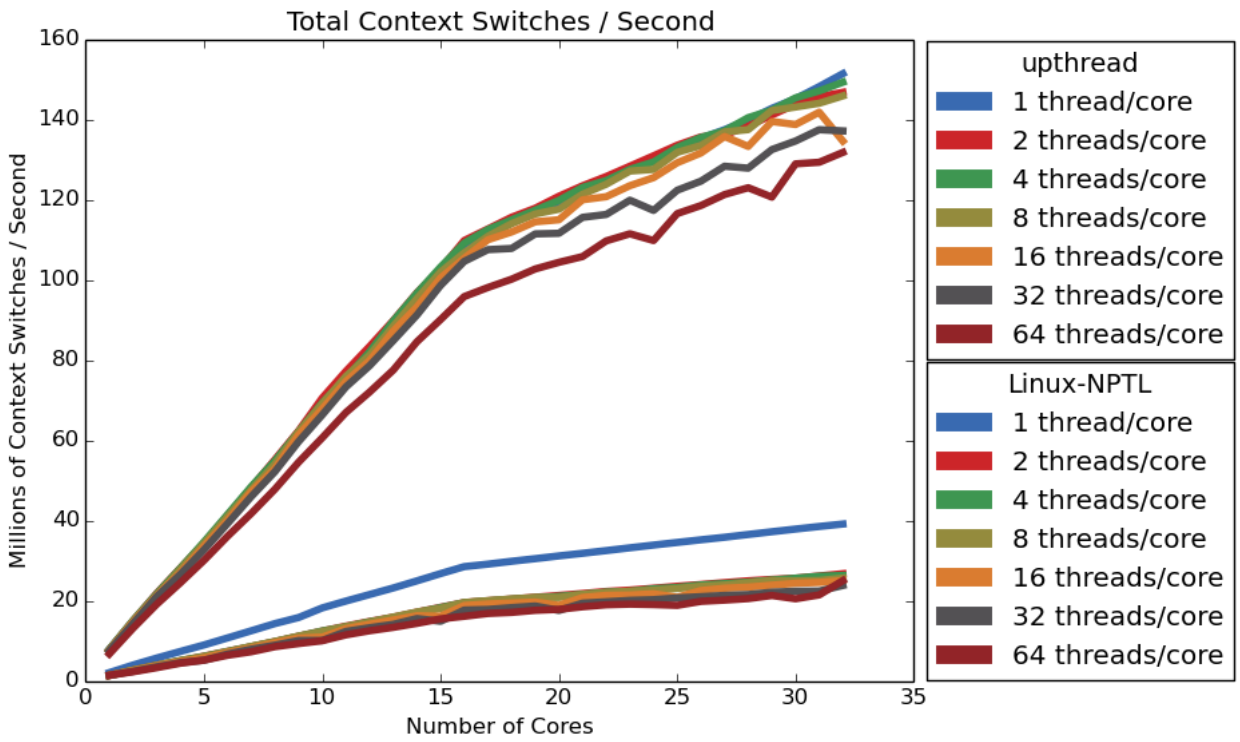


Figure 2.7: Total context switches per second of upthreads vs. the Linux NPTL for an increasing number of cores and a varying number of threads per core.

- In the Linux NPTL, yielding a single thread is about $1.5x$ faster than context switching between threads (though both are significantly slower than doing a context switch on upthreads).
- Both the Linux NPTL and upthreads experience a $1.5x$ increase in average context switch latency when all hyperthreaded cores are used (i.e. the 32 core case).

Overall, these results are not surprising. A context switch in parlib is very light weight and it makes sense that it would outperform Linux by such a high margin. Traditionally, user-level context switches have used the `getcontext()` and `setcontext()` calls from glibc to save and restore a context. However, these calls require you to save and restore *every* cpu register, as well as make a system call to save and restore a context's `sigmask`. On parlib, we do not need to pay these extra costs because all context switches are performed via a function call (meaning we only need to save and restore registers according to the AMD64 System V ABI), and saving and restoring the sigmask is unnecessary because POSIX signal handling is emulated in user-space (as described in Section 2.4). The end result is that context switches on parlib (and consequently upthreads) are extremely fast.

To demonstrate this further, Figure 2.7 shows the total number of context switches/second that can be performed on upthreads vs. the Linux NPTL. These results are gathered from the same experiment above, but presented in a different format.

The major takeaways from this graph are:

- In all cases, upthreads is able to perform more context switches/second than the Linux NPTL, regardless of the number of cores in use.
- For both upthreads and the Linux NPTL, the number of context switches/second increases linearly with the number of cores.
- For both upthreads and the Linux NPTL, the slope of this linear increase is steeper between 1 and 16 cores than between 17 and 32 cores (i.e. when cores begin to use their hyperthreaded pair).
- On both sides of this knee, the slopes of the upthreads results are much steeper than the slopes of the Linux NPTL results.
- For the Linux NPTL, having only 1 thread per core is much better than having multiple threads per core.
- For both upthreads and the Linux NPTL, having more threads per core decreases the number of context switches/second that can be performed. This effect is more pronounced on upthreads than it is for the Linux NPTL.

It's not surprising that adding hyperthreads affects the slope of the linear increase, since hypertheads naturally interfere with one another and compete for cpu time on the same core. It's also not surprising that the number of threads/core affects the number of context switches/core that can be performed. The more threads you run per core, the higher the chances are that their stacks will collide in the cache when swapping among them. All things considered, these results correspond to upthreads consistently performing between 5.5 and 5.7 times more context switches/second than the Linux NPTL under all scenarios measured. We couldn't have asked for better results.

Flexibility in Scheduling Policies

In this experiment, we compare the performance of the Linux NPTL to different variations of our upthread library when running a simple compute-bound workload across a large number of threads. We create 1024 threads to do a fixed amount of work in a loop and measure the time at which each of these threads completes. We run this experiment for the Linux NPTL as well as six variations of our upthread library. Each upthread variant differs only in the frequency with which it makes a new scheduling decision. The goal of this experiment is to show the effect of varying the scheduling frequency, as well as demonstrate the flexibility in scheduling policies that parlib allows. The work loop run by every thread can be seen below:

```
for (int i = 0; i < 300000; i++) {
    for (int j=0; j < 1000; j++)
        cmb();
    #ifdef WITH_YIELD
    pthread_yield();
    #endif
}
```

Each thread runs for 300 million loop iterations, broken up across two nested loops. The outer loop contains 300,000 iterations and the inner loop contains 1000 iterations. We make this separation in order to force some of our experiments to voluntarily yield back to the scheduler every 1000 iterations via a call to `pthread_yield()`. We do this by setting the `WITH_YIELD` flag at compile time. The number of iterations was chosen arbitrarily, such that the full benchmark would take around 30s to complete for all experiments.

The application itself only calls `pthread_create()` 1023 times, with the main function acting as one of the overall threads. A barrier is used to make sure that every thread starts up and is sitting at a point just before we are ready to start taking measurements. Once all threads have reached the barrier, we start the clock, and whatever thread happens to be on each core starts running its work loop.

In each of the upthread variants, threads are striped evenly across all of the vcores in the system before the experiment begins. With 1024 threads and 32 cores, each vcore is responsible for 32 threads. For the Linux NPTL, we simply create all 1024 threads and let the underlying scheduler figure out what to do with them. The details of each setup are described below.

upthread-no-preempt: This is a variant of upthreads set up with per-vcore run queues, vcore yielding disabled, and stealing turned off. No preemption is used, and all threads run to completion before starting the next thread. Also, the `WITH_YIELD` flag is *not* set, so the application will never voluntarily yield to the scheduler during its work loop.

upthread-{1000, 500, 100, 10}ms: These are preemptive versions of the setup described above, with preempt periods of $1000ms$, $500ms$, $100ms$, and $10ms$ respectively. Each variant runs with the same configuration as `upthread-no-preempt`, except that when their preemption period is up, all of their vcores are interrupted and forced to schedule another thread from their queues. Each interrupted thread moves to the end of its respective queue.

upthread-yield-1000: This variant of upthreads is identical to `upthreads-no-preempt`, except that the `WITH_YIELD` flag is set at compile time, forcing the application to yield back to the scheduler every 1000 loop iterations.

Linux-NPTL: This is the native Linux NPTL implementation. We determined the scheduling period of this algorithm to be $10ms$ by calling `sysconf(_SC_CLK_TCK)`. This value is set at the time the kernel is compiled, and we did not change it from its default value.

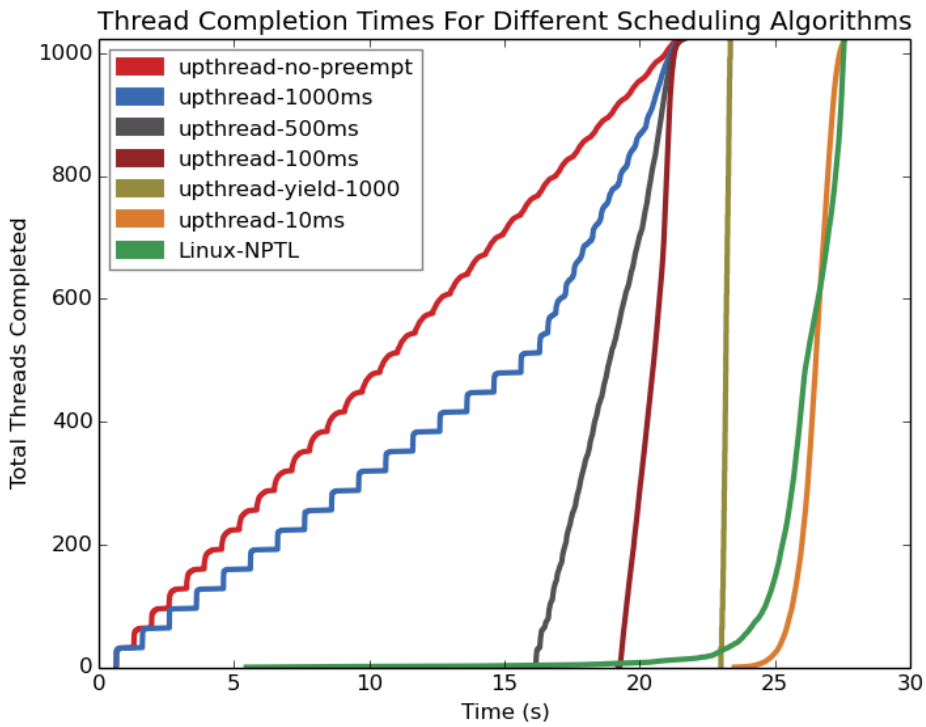


Figure 2.8: Thread completion times of a fixed-work benchmark for different scheduling algorithms. The benchmark runs 1024 threads with 300 million loop iterations each, averaged over 50 runs.

Figure 2.8 shows the results of running this experiment. The x-axis is time and the y-axis is the total number of threads that have completed by that time. The results themselves are an average over 20 runs. The major takeaways from this graph are:

- All upthread variants complete the full benchmark before the Linux NPTL does.
- The `upthread-no-preempt` implementation completes threads in a "stair-step" pattern, with 32 threads (1 thread per core) completing every $650ms$. The stair-step pattern flattens out over time because the benchmark was averaged over 20 runs, and the slight variations in start times on each core compound on each iteration. The total duration of the benchmark is $650ms * 1024/32 = 20.8s$.
- The `upthread-1000ms` implementation follows a "stair-step" pattern up to $16s$ and then ramps up, completing its remaining threads very quickly. The full benchmark finishes at about the same time as the `no-preempt` case.
- The `upthread-500ms` and `upthread-100ms` implementations show a similar pattern to the "ramp-up" phase of `upthread-1000ms`, but without the initial "stair-step" pattern. The ramp-up phase starts later with a preempt period of $100ms$ than with $500ms$. In both cases, the full benchmark finishes at about the same time as the `no-preempt` case.

- The `upthread-yield-1000` implementation is (mostly) a straight vertical line, indicating that all threads complete at roughly the same time. The overhead of explicitly yielding every 1000 iterations (300,000 yields in total), pushes the completion time of the benchmark to $23s$ instead of $20.8s$.
- The Linux NPTL performs the worst, but follows a similar curve as the `upthread-10ms` implementation which interrupts its cores at the same scheduling frequency. Although context switches are faster on upthreads, each of these context switches is paired with a high-overhead signal that make the overall context switch cost similar to that of the Linux NPTL. The completion time of the benchmark in both of these cases is approximately $27.5s$

In general, these results are not surprising. A longer preempt period means fewer context switches, which translates to less overhead and a faster running benchmark. The `upthread-no-preempt` implementation has no overhead, whatsoever, and is able to complete the benchmark in exactly the amount of time it takes to cycle through all threads on the number of cores available. In the end, all threads get the same amount of CPU time, but the fairness of the scheduling algorithm is not very good. Likewise, the `upthread-yield-1000`, `upthread-10ms`, and `Linux-NPTL` cases are not surprising either. The act of explicitly yielding in `upthread-yield-1000` injects overhead into each threads computation, which moves the benchmark completion time forward a bit. For `upthread-10ms` and `Linux-NPTL`, the yields are not explicit, but the effect is similar. In all three cases, most threads finish at roughly the same time demonstrating a much fairer scheduling algorithm.

The `upthread-1000ms`, `upthread-500ms`, and `upthread-100ms` cases are a bit more interesting. The `upthread-1000ms` implementation follows a "stair-stop" pattern up to $16s$ and then ramps up, completing its remaining threads very quickly. Each step is $1000ms$ wide, with 64 threads/sec being started, but only 32 threads/sec actually running to completion. As with the `upthread-no-preempt` case, the first 32 threads are able to complete their full $650ms$ computation with the $1000ms$ time window. However, the remaining 32 threads are only able to complete $350ms$ of their computation before being interrupted and put to the back of the queue. After $16s$, all threads will have started ($1024/64 = 16s$), but only half of them will have completed. The ones that remain only have $350ms$ of computation left and are all able to complete quickly within their next scheduling quantum. Over the lifetime of the computation, each core is only interrupted 20 times, so the overhead of the timer interrupt ends up being negligible and the full benchmark finishes at about the same time as the `no-preempt` case.

The `upthread-500ms` and `upthread-100ms` implementations show a similar pattern to the "ramp-up" phase of `upthread-1000ms`, but without the initial "stair-step" pattern. No stair-step pattern exists because it takes $650ms$ for any round of threads to finish its computation, but the preemption timer fires before this time is reached. With a preemption period of $500ms$, 64 threads/sec will be started (as before) but each will only have completed $500ms$ of computation before moved to the back of the queue. After $16s$, all threads will have started, but none of them will have completed yet. Each thread is then able to quickly complete its

remaining $150ms$ of computation during its next scheduling quantum. Similar logic follows for a preempt period of $100ms$, except that we start 320 threads/sec, and each thread goes through 6 rounds of scheduling before being able to complete its full $650ms$ of computation. As seen on the graph, this results in the "ramp-up" time starting at $1024/320 * 6 = 19.2s$ instead of $16s$. In both of these cases, the preemption timer overhead is still negligible, as the full benchmark completes at approximately the same time as the `no-preempt` case.

NAS Benchmarks

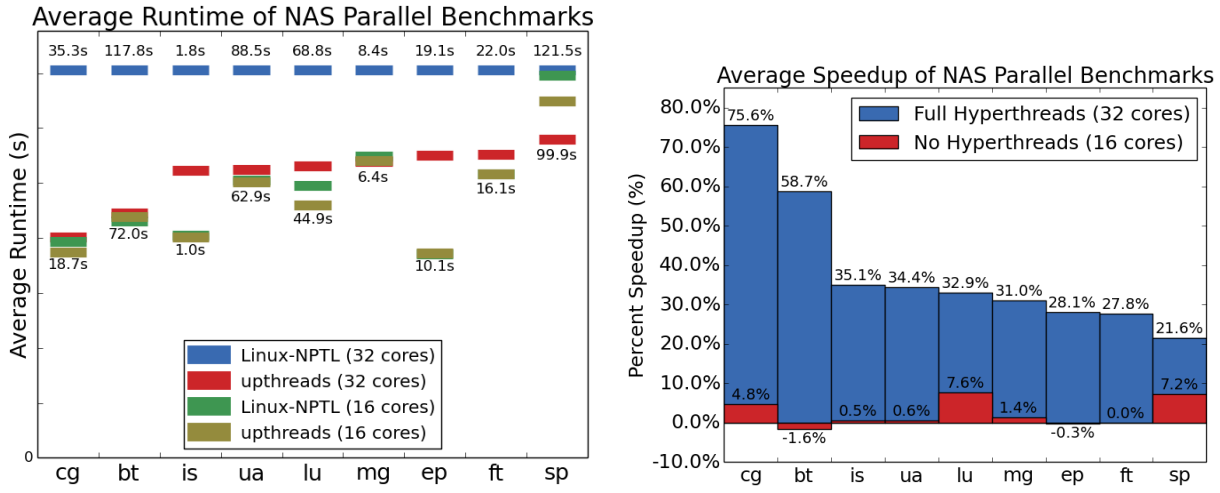
In this experiment, we measured the impact of running a set of benchmarks from the NAS parallel benchmark suite on top of upthreads vs. the Linux NPTL. Specifically, we ran the full set of OpenMP benchmarks with problem sizes in class "C". Class "C" problems are the largest of the "standard" test sizes, and provide a good tradeoff between being computationally intensive and still being able to complete in a reasonable amount of time. The full set of benchmarks we ran are listed below with the type of computation they perform:

- **cg** - Conjugate Gradient
- **bt** - Block Tri-Diagonal Solver
- **is** - Integer Sort
- **ua** - Unstructured Adaptive Mesh
- **lu** - Lower-Upper Gauss-Seidel Solver
- **mg** - Multi-Grid on a Sequence of Meshes
- **ep** - Embarrassingly Parallel
- **ft** - Discrete 3D Fast Fourier Transform
- **sp** - Scalar Penta-Diagonal Solver

The goal of this experiment is to show that parlib-based schedulers are robust enough to run well established benchmarks, rather than just the simple benchmarks we construct ourselves. Moreover, this experiment demonstrates that upthreads is capable of being integrated successfully with a general purpose parallel runtime, i.e. OpenMP.

Because upthreads is *not* binary compatible with the Linux NPTL, we had to extract libgomp from gcc and recompile it against our upthread library in order to run this experiment. Additionally, we configured upthreads with `upthread_can_vcore_request(FALSE)`, `upthread_can_vcore_steal(FALSE)`, and `upthread_set_num_vcores(OMP_NUM_THREADS)`. This configuration allowed us to ensure that threads created by OpenMP would be evenly distributed across exactly `OMP_NUM_THREADS` vcores, and that those threads would never migrate across cores.

`OMP_NUM_THREADS` is an environment variable which tells OpenMP how many threads it should create to manage its parallel tasks. Typically this value is set to the number of cores you wish to make available to OpenMP (even when using the Linux NPTL). Under the hood, OpenMP will pin these threads to cores and multiplex its tasks on top of them. For the purpose of these experiments, we ran each benchmark with `OMP_NUM_THREADS` set to both 16 and 32 (i.e. all the cores on the system with and without hyperthreading).



(a) Average Runtime

(b) Speedup

Figure 2.9: Speedup and average runtime of the OpenMP-based benchmarks from the NAS Parallel Benchmark Suite on upthreads vs. the Linux NPTL. Figure (a) shows the average runtime over 20 runs when configured for both 16 and 32 cores. Figure (b) shows the speedup. The red bars represents the speedup when all benchmarks are ran on 16 cores. The blue bars represents the speedup when all benchmarks are ran on 32 cores.

For each value of `OMP_NUM_THREADS` we ran each benchmark 20 times and calculated its average runtime. Additionally, we calculated the percent speedup of running each benchmark on upthreads vs. the Linux NPTL. Figure 2.9 show the results of these benchmarks. Figure (a) shows the runtimes on both 16 and 32 cores. Figure (b) shows the speedups. The red bars represents the speedup when all benchmarks are ran on 16 cores. The blue bars represents the speedup when all benchmarks are ran on 32 cores.

The major takeaways from these graphs are:

- For all benchmarks, the slowest configuration is the Linux-NPTL on 32 cores.
- For all but 2 of the benchmarks, upthreads provides the fastest configuration (usually with 16 cores, but for `sp` it was 32).
- The times when the Linux NPTL is faster, upthreads is within a couple percent difference (i.e. 1.6% for `bt` and 0.3% for `ep`).
- For all benchmarks, upthreads is significantly faster on 32 cores than the Linux NPTL on 32 cores (between 21.6% and 75.6%s faster). Most of the time this is true for 16 cores as well, but the effect is much less pronounced (between -1.6% and 7.6% faster).
- Most benchmarks on the Linux NPTL tend to run significantly faster on 16 cores than 32 cores. The only exception is the `sp` benchmark, where the Linux NPTL doesn't differ much between 16 and 32 cores (although 16 cores is still faster).
- For upthreads, sometimes the gap between 16 core performance is big and other times it is small. In all but the `sp` benchmark, however, the 16 core performance is better.

As with the previous experiments, these benchmarks only tested upthread's ability to run computationally expensive workloads. Unlike those experiments, however, these benchmarks do not exploit the advantages of having a user-level threading library; the number of threads they use is always less than or equal to the number of cores on the machine. As such, upthreads is not the clear winner in this experiment as it was for the previous two experiments. These results leave it inconclusive as to whether or not it is better to run OpenMP on upthreads vs. the Linux NPTL. We do, however exercise parlib's barrier and semaphore paths extensively, as well as demonstrate that upthreads is at least competitive with the Linux NPTL when running OpenMP code.

Probably one of the more interesting results to come from this experiment is how well upthreads performed on 32 cores compared to the Linux NPTL. For whatever reason, the Linux NPTL falls apart as soon as these benchmarks start to use hyperthreaded cores. The upthread library likely does better because of the alignment tricks it uses to keep threads from interfering with one another in the cache. In practice, this property could be used to better align application data structures with those used by the threading library. Moreover, the threading library itself could be replaced with one that better matched the needs of the application itself. We expand on this idea in our discussion of the Lithe port of OpenMP in the following chapter.

Kweb Throughput

In this experiment we measured the maximum achievable throughput of a thread-based webserver we wrote, called kweb. Specifically, we compared the throughput of different variants of kweb when running on the Linux NPTL, upthreads, and a custom user-level scheduler built directly into kweb itself. We also compared these results to running an optimized version of `nginx` with the same workload (i.e. an *event-based* webserver, which is currently popular for serving static content on the web). The goal of this experiment was to show how well parlib-based schedulers perform in the face of I/O-bound workloads. Additionally, we demonstrate the impact of integrating a parlib-based scheduler directly into an application, rather than relying on an underlying threading library. Although the primary experiment is ran on Linux, we provide results from running kweb on Akaros as well.

A number of different webserver architectures have emerged over the past 20 years that argue for either threads, events, or some hybrid of the two [47, 48, 49, 43]. In recent years, however, the popularity of thread-based webservers has diminished in favor of event-based servers due to low-performance and scalability concerns. Events themselves are not inherently better than threads, but system support for high-performance threading is lacking in systems of today, artificially favoring event-based architectures over threaded ones. One of the goals of parlib is to fix this system support so that thread-based servers can, once again, be competitive with their event-based counterparts. Just as Capriccio used the knot webserver to demonstrate high performance and scalability for thread-based servers in the early 2000s, parlib does the same with kweb today.

For our experiment, we started each variant of kweb with a maximum of 100 threads and let it run according to the policy described for each variant in Section 2.6.2. For nginx, we followed the advice of several blog posts [50, 51] to try and configure it in the most optimized way possible. Specifically, we configured it to run with `epoll` instead of `select`, the `sendfile` system call, logging disabled, file-caching enabled, multi-accept enabled, etc.

Additionally, we configured each setup to restrict itself to a specific number of cores for the duration of each experiment. In this way, we were able test the scalability of each webserver by varying the number of cores used to service any incoming HTTP requests. For upthreads and the custom-kweb scheduler we explicitly requested `n` vcores before the start of each experiment. For the Linux NPTL we used the `taskset` instruction to restrict the application to only run on the first `n` cores in the system. For nginx, we modified the `worker_processes` variable in `nginx.conf` to restrict it to `n` worker processes (with 1 event loop per process).

We ran this experiment by blasting kweb and nginx with requests from a simple http-request generator we wrote. We configured our http-request generator to operate as follows:

- Start 100 TCP connections in parallel and prepare to send 1000 http-requests/connection.
- Batch requests to allow 100 concurrent requests/connection on the wire at a time.
- After each request receives a response, place a new request on the wire.
- After each connection completes all 1000 requests, start a new connection.

For each webserver variant at each core count, we ran the http-request generator for 10 minutes and recorded the number of requests that completed every 5 seconds. We then took the average of these values over the complete 10-minute period to compute the total throughput of the webserver in requests/second.

With this setup there are always approximately 10,000 outstanding requests on the wire at any given time. Requests themselves consist of an HTTP 1.1 GET request for an empty file. By requesting a file of minimum size, we maximize the throughput we can measure before hitting the bandwidth limitations of the network card. For each request, the server sends a response consisting of a 242-byte HTTP 1.1 response header and a 0-byte body. The 242-byte header is standard for nginx, and we customized the kweb header to match this size for a fair comparison.

This workload is unrealistic in terms of requests/connection, but is great for stress-testing the webserver in terms of how fast it can process raw requests. Sending a more realistic workload would have required a more complicated setup involving multiple servers and would not have contributed much to our overall goal of demonstrating how well parlib-based schedulers are able to perform in the face of I/O-bound workloads.
The results of this experiment can be seen in Figure 2.10. The major takeaways are:

- The `kweb-linux-upthread` and `kweb-linux-custom-sched` webservers have the highest peak throughput of all webserver variants measured.
- The peak throughput of these webservers is 6% greater than nginx.

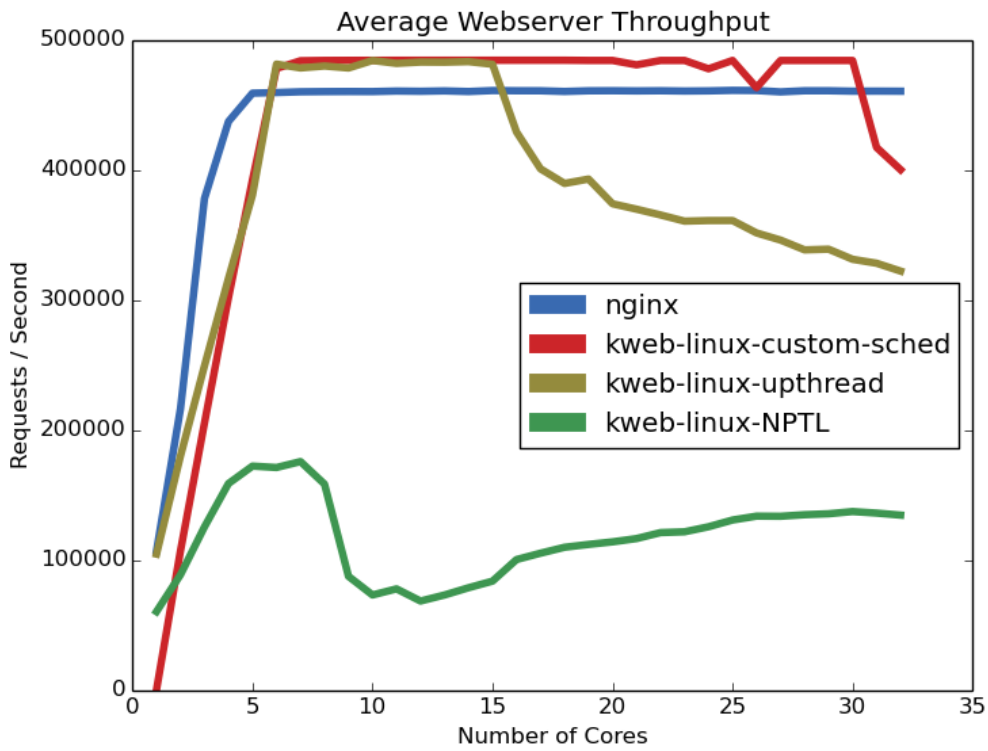


Figure 2.10: Webserver throughput with 100 concurrent connections of 1000 requests each. Requests are batched to allow 100 concurrent requests/connection on the wire at a time. After each request is completed, a new one is placed on the wire. After each connection completes all 1000 requests, a new connection is started.

- The Linux-NPTL based webserver has, by far, the worst throughput of all of the webserver variants ($2.7x$ less than upthreads/custom-sched and $2.5x$ less than nginx).
- In general, all of the webservers operate at their peak throughput with 7 cores, and either stay the same, or get worse with additional cores.
- The Linux-NPTL webserver is the only one to "recover" after this, steadily increasing its throughput for every new core added beyond 10.
- The `kweb-linux-upthread` webserver starts to see decreases in throughput after 16 cores (i.e. once hyperthreading comes into play).
- The `kweb-linux-custom-sched` webserver starts to see fluctuations in throughput beyond 25 cores (it is unclear exactly what causes this).

We could not have asked for better results. Our two parlib-based webservers were able to outperform an optimized event-based webserver by 6%. Even though their performance starts to fall off after a certain point, this problem is artificial, as `nginx` actually only spins up a maximum of 12 processes even when configured for up to 32. Kweb could easily be made to throttle itself similarly and the line on the graph would continue straight as with `nginx`.

Unfortunately, we are not able to surpass the peak values we see for `kweb-linux-upthread`, `kweb-linux-custom-sched`, and `nginx` because we start to approach the 1Gb/sec limit of our network card at these peak values. Table 2.2 shows the TX bandwidth of each webserver at its peak value when running on 7 cores (as obtained by `iftop`).

	Bandwidth
kweb-linux-custom-sched	923 Mbps
kweb-linux-upthread	920 Mbps
nginx	900 Mbps
kweb-linux	330 - 400 Mbps

Table 2.2: CPU usage and bandwidth consumed by different variants of kweb and nginx when configured for up to 7 CPUS (i.e. when all configurations are at their peak).

We use 7 cores as our point of reference because all webservers are at their peak value with this number of cores (though it only really matters for `kweb-linux-NPTL` that we pick this exact value). The `kweb-linux-NPTL` webserver likely peaks at this value because 7 cores is just shy of utilizing one full socket without hyperthreading. Exactly why this seems to matter or why 8 isn't the magic number is currently unknown (though it's likely due to some weird scheduling constraints imposed by the Linux kernel scheduler in relation to network interrupts, etc.). Additionally, it's unclear why `kweb-linux-NPTL` is able to recover slightly as additional cores are added beyond 10. It is likely an artifact of how we chose to add additional cores via the `taskset` command. If we had started at core 8 instead of 0, for example, the curve might have looked different (though the peak would likely not have been much different than it is currently).

It's possible we could achieve higher throughputs if we ran the experiment on a NIC with a higher bandwidth limitation. However, it is unclear why `nginx` does not actually peak at the same value as `kweb-linux-upthread` and `kweb-linux-custom-sched`. It's possible that we have actually hit the limit of `nginx` (although this is doubtful). It's more likely that we simply haven't optimised `nginx` as best as we possibly could.

What's more interesting, however, is how well `kweb-linux-upthreads` performs compared to `kweb-linux-NPTL`. The application code for these 2 variants of kweb is almost identical, varying only in the way it sets timeouts on its socket `read()` and `write()` calls. These results truly show the power of user-level scheduling combined with asynchronous system calls to improve the performance of thread-based applications with I/O-bound workloads.

For reference, Figure 2.11 shows the results of running this experiment on Akaros with its version of upthreads and the custom-kweb scheduler. We only show the results from running the experiment on the first 16 cores, as beyond this value the throughput falls to 0. We also show the `kweb-linux-NPTL` curve as a point of reference to see just where the throughput on Akaros falls in comparison to Linux. As mentioned previously, the poor results on Akaros aren't the fault of kweb or parlib, but rather a poorly performing networking stack under the hood. One thing we are able to take away from this experiment, however, is that the

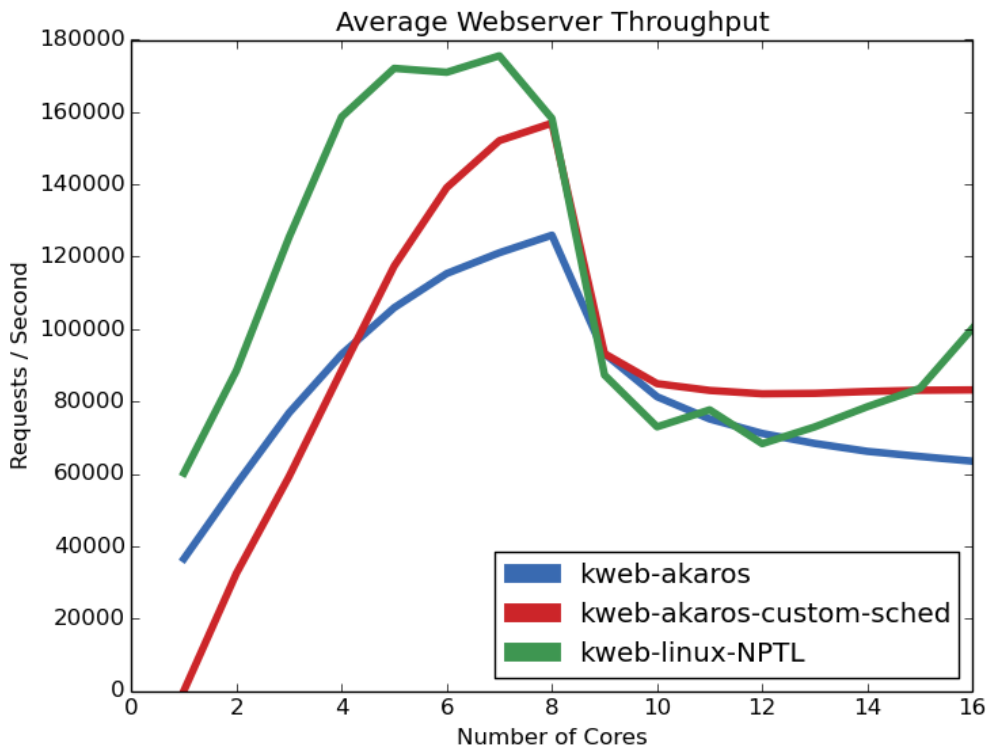


Figure 2.11: Results from running the kweb throughput experiment on Akaros. The results are much worse due to a substandard networking stack on Akaros. The Linux NPTL results are shown for reference.

`kweb-akaros-custom-sched` has a higher peak throughput than `kweb-akaros`. This shows that the optimizations we put into this custom scheduler actually do serve to help the performance of kweb overall. We weren't able to see this on Linux because we hit the limitations of the networking card before these results could become obvious. As the Akaros networking stack improves, we expect these results to improve as well.

2.7 Discussion

In this chapter we presented parlib, a framework for developing efficient parallel runtimes based on user-level scheduling. The two most important abstractions provided by parlib are the **vcore** and **uthread** abstractions. Vcores provide uninterrupted, dedicated access to physical cores, and uthreads provide user-level threads to multiplex on top of those cores. These abstractions, combined with a fully asynchronous system call interface and a unified event delivery mechanism, make it possible to construct highly efficient parallel runtimes on modern day multi-core architectures.

Ports of parlib exist for both Linux and Akaros, though both systems currently have some deficiencies. On Linux, many of parlib's abstractions are "faked" due to limitations of the

underlying OS. On Akaros, disjoint parts of the system adversely affect parlib's performance, making it hard to evaluate. Despite these deficiencies, the results presented in Section 2.6 affirm that parlib is a system that warrants further exploration. To underscore this further, it's useful to point out a few experiments from Barret Rhoden's dissertation that evaluate parts of Akaros that have direct impact on the abstractions provide by parlib. Specifically, we discuss the results from his *FTQ*, *"pdr"*, and *provisioning/allocation* benchmarks.

The Fixed Time Quantum (FTQ) benchmark was used to evaluate how well Akaros could isolate cores compared to Linux. This benchmark indirectly shows the effectiveness of the vcore abstraction in terms of handing out dedicated cores to an application. The less noise on a core, the more isolated it is when Akaros hands it out. All things equal, the FTQ benchmark shows that Akaros has an order of magnitude less noise and fewer periodic signals on its cores than Linux. The noise that does exist can mostly be attributed to improperly initialized hardware and unavoidable interrupts from System Management Mode (SMM) on the x86 hardware used in the experiment.

The "pdr" benchmark was used to evaluate the "Preemption Detection and Recovery" locks discussed in Section 2.2.3. Preemption itself is only superficially discussed in this chapter, but handling it properly is crucial for full support of the vcore abstraction and its ability to provide true dedicated access to underlying cores. Overall, the results of these experiment show that these locks are, in fact, able to properly recover from preemption. Additionally, they are shown to have a modest overhead compared to traditional spinlocks.

Finally, the provisioning/allocation benchmark evaluates Akaros's ability to fluidly pass cores back and forth between applications according to a specific core provisioning policy. Provisioning itself was briefly discussed in Section 1.2.2 and is crucial to the overall goal of efficiently managing cores in future systems. However, none of the user-level schedulers we have built so far are smart enough to make intelligent use of provisioned cores. Provisioning itself is most useful for managing cores between low-latency and batch jobs, and future user-level scheduler we build will need to take these considerations into account to automatically adjust their core requests to accommodate these types of workloads.

In general, the benchmarks described above all demonstrate features of parlib that Akaros is able to support, but Linux is not. Although it's conceivable that Linux could add support for true dedicated access to cores and the preemption capabilities that come with it, it's likely that this will not happen any time soon. In the end, we believe Akaros is the way forward, but Linux provides a good stepping stone for demonstrating the ideas in their current form.

As a final note, it's necessary to point out that parlib is designed specifically for applications that rely on a single runtime to manage all of its parallelism. But what happens when an application requires multiple runtimes? How do these runtimes coordinate access to cores requested through the vcore abstraction? How do they enforce unique scheduling policies for their own particular set of threads? Our answer to these questions is **Lithe**, and it is the subject of the following chapter.

Chapter 3

Lithe

This chapter presents Lithe: a framework for composing multiple parallel runtimes. Where parlib provides abstractions to build individual parallel runtimes, Lithe provides abstractions that allow multiple parallel runtimes to interoperate in harmony. Lithe was originally proposed by Pan, et al. in 2010 [3, 4], and all work presented here is an extension of this previous work.

Figure 3.1 shows a representative application of the type targeted by Lithe. This application consists of four distinct components, each of which is developed using a different parallel runtime. Such an application is typical in today's open source community, where highly-tuned libraries designed for specific tasks are readily available. Application writers are able to leverage these libraries instead of writing their own from scratch.

However, arbitrarily composing libraries from a disparate set of developers can sometimes lead to performance problems (even if each individual library is highly-tuned itself). Different developers have different preferences about which parallel runtime they prefer to use. Moreover, different parallel runtimes provide different programming models that tailor themselves to certain types of computation. By choosing the best runtime for the job, individual libraries have the potential to perform extremely well when run in isolation. However, combining runtimes from multiple libraries can cause performance problems if those runtimes interfere with one another by obliviously using the same physical cores.

Individual parallel runtimes tend to assume they are the only runtime in use by an application and configure themselves to run on all cores in the system. Indeed, Intel's Thread Building Blocks Library (TBB), OpenMP, and Google's Go language runtime all configure themselves to run this way by default. As such, using multiple parallel runtimes in a single application can lead to over-subscription of cores and sub-optimal scheduling of threads mapped to those cores. This problem is exacerbated by the fact that many parallel computations rely on barriers and other synchronization primitives, where waiting an arbitrary amount of time for any given thread can have major consequences on the performance of the overall computation. Hooks typically exist to statically partition cores to each runtime, but tuning them properly for each application can be time consuming and error prone. Moreover, the optimal number of cores required by each runtime may change

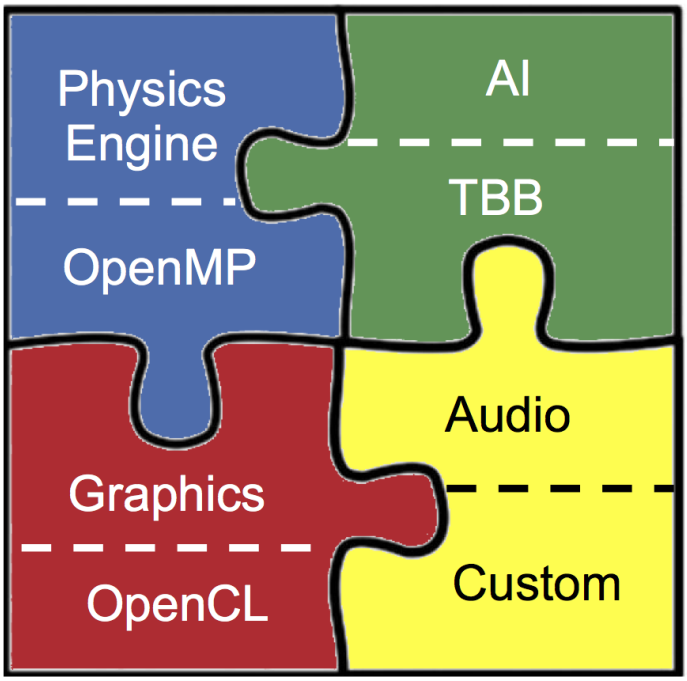
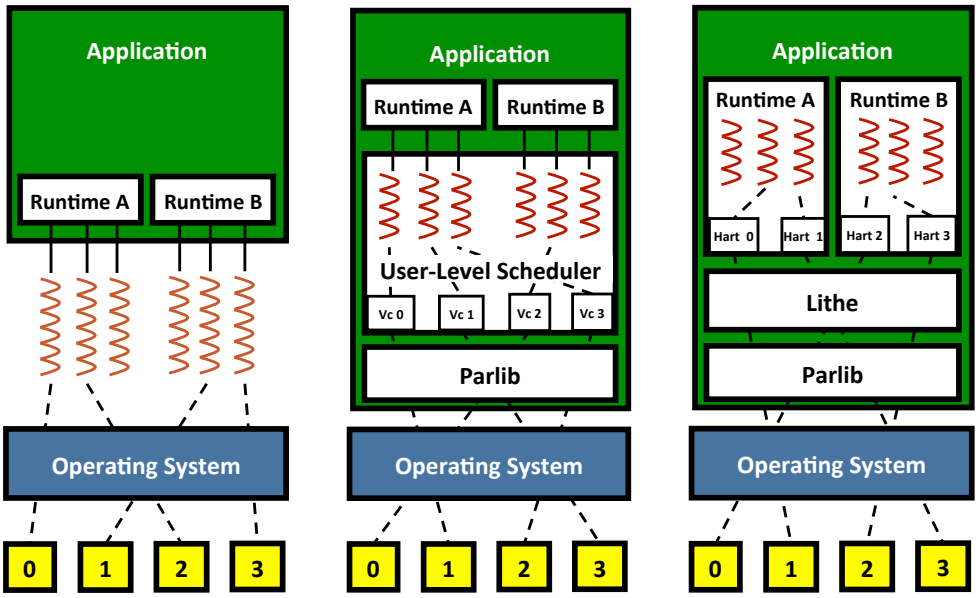


Figure 3.1: Representative application of the type targeted by Lithe. Different components are built using distinct parallel runtimes depending on the type of computation they wish to perform. These components are then composed together to make up a full application.

throughout the lifetime of an application, so creating fixed partitions may cause more harm than good.

For example, consider an application written primarily in Go that makes calls to the Math Kernel Library (MKL) [52] to execute an optimized linear algebra routine. Under the hood, MKL invokes TBB, which spreads its computation across all the cores it sees available in the system. Meanwhile, Go schedules go routines on all of the cores it sees available, unaware that TBB is trying to do the same. Both runtimes operate independently and provide no opportunity for coordinating access to cores, or otherwise communicating with each other. Neither traditional OS interfaces, nor the interfaces provided by parlib provide facilities to directly address the problems that can arise from this setup.

Parallel runtimes based on traditional OS threading abstractions (e.g. Linux NPTL) typically spawn one pinned OS thread per core, and multiplex whatever parallel tasks they have on top of them (similar to how parlib emulates the vcore abstraction on Linux). However, the existence of multiple parallel runtimes causes thrashing in the OS as it must multiplex threads from all runtimes on top of the same set of underlying cores. Static partitioning alleviates this problem somewhat, but causes another set of problems. Some of the cores partitioned to one runtime might end up sitting idle while other runtimes could have used them to do useful work. Furthermore, one runtime might want to voluntarily lend some of its cores to another runtime in order to speed up a computation being performed on its behalf. Such a feature would have been useful in the example of Go and TBB provided above.



(a) Traditional parallel runtimes

(b) Parallel runtimes with parlib

(c) Parallel runtimes with Lithe

Figure 3.2: The evolution of parallel runtime scheduling from traditional OS scheduling in the kernel (a), to user-level scheduling with parlib (b), to user-level scheduling with Lithe (c). Lithe bridges the gap between between providing an API similar to parlib's for building customized user-level schedulers, and allowing multiple parallel runtimes to seamlessly share cores among one another.

As discussed in the previous chapter, parlib provides developers with a set of tools to gain direct access to cores and schedule user-level threads on top of them. Using these tools, individual runtimes could be modified to implement their own user-level threading abstractions on top of parlib. Moving each runtime's thread scheduler into the runtime itself helps to solve the problem of multiplexing threads from different runtimes on the same underlying physical cores. However, different runtimes still have no way of communicating with each other in order to coordinate the sharing of cores. One runtime could monopolize all the cores on the system, oblivious to the fact that other runtimes would like to use them. Alternatively, all runtimes could be configured to run on top of a single underlying user-level scheduler (e.g. our upthread library described in Section 2.6.1). However, this would only serve to push the problem seen on existing systems up a level, as thrashing would now occur inside the user-level scheduler instead of in the OS.

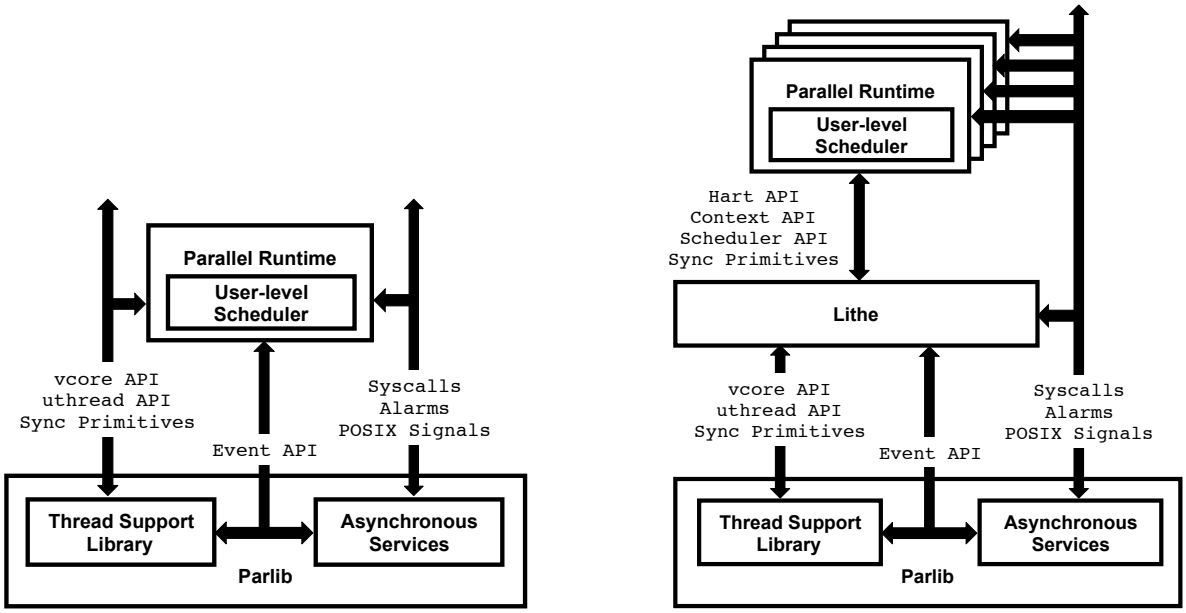
Lithe bridges the gap between between providing an API similar to parlib's for building customized user-level schedulers and allowing multiple parallel runtimes to seamlessly share cores among one another. Using this API, each runtime is able to maintain full control of its own thread scheduling policy, while keeping the underlying physical cores from being over (or under) subscribed. Figure 3.2 shows the difference between running multiple parallel runtimes in a traditional OS, running them directly on parlib, and running them on Lithe.

Using Lithe's API, legacy runtimes can be ported into the Lithe framework to provide bolt-on composability without the need to change existing application code. For example, when used in the context of a multi-frontal, rank-revealing sparse QR factorization algorithm (SPQR) [30], Lithe has been shown to outperform versions of the code that hand-tuned the static allocation of cores to each of its underlying parallel runtimes [3] (i.e. TBB and OpenMP). Lithe not only allowed the application to perform better, but also removed the need for fragile manual tuning. Similarly, Lithe has been shown to improve the performance of a Flickr-Like Image Processing Server [4] and a Real-Time Audio Application [3].

Although not originally proposed as such, Lithe can be thought of as a natural extension to parlib, with the added capability of coordinating access to cores among multiple runtimes. In Lithe terminology, the most fundamental primitive it provides is the *hart abstraction*. Harts (short for HARdware-Thread contexts) represents physical processing resources that can be explicitly allocated and shared within an application. Typically a hart is just a core, but it could be one half of a hyper-threaded core or a single lane in a multi-lane vector processor. In the context of parlib, harts map 1-1 with vcores. Lithe also defines primitives for *contexts* and *schedulers*, which map 1-1 with parlib's notion of uthreads and user-level schedulers respectively.

However, Lithe's original reference implementation was written before parlib existed, and little effort was made to ensure that it could easily be ported to run on any system other than Linux. Moreover, not all of Lithe's proposed features were included in this reference implementation (e.g. the ability to interpose on system calls to ensure that a scheduler doesn't lose its core when blocking system calls are made). The idea was that Lithe would be the lowest-level library that developers would port, leaving the implementation of its hart, context and scheduler abstractions intimately intertwined. With the introduction of parlib, we are now able to separate those features which pertain only to hart and context management from Lithe's primary innovation: it's extended scheduler API. We map harts 1-1 with vcores, contexts 1-1 with uthreads, and Lithe's scheduler API becomes a superset of the scheduler API provided by parlib.

In this chapter we present our port of Lithe to run on top of parlib. In the process, we have cleaned up many of Lithe's outward facing interfaces and made its usage semantics much more consistent. The result is a better designed, cleaner implementation of Lithe that runs on any platform that supports parlib (i.e. Linux and Akaros). We begin with an overview of Lithe, including the details of how we have re-architected it to run on top of parlib. We then delve into the details of Lithe itself and discuss the new API it presents to developers (and how this differs from both the original Lithe API and the API provided by parlib). Next, we discuss the development of a generic "Fork-Join" scheduler on top of which the OpenMP, TBB, and a new "lithe-aware" version of upthreads are now all based. Finally, we evaluate our port of Lithe by re-running the SPQR benchmark and an expanded version of the Flickr-like image processing application presented in the original Lithe paper [4]. Just as with parlib, we perform these experiments exclusively on Linux to provide a better comparison with existing systems. In both experiments our new version of Lithe performs extremely well.



(a) Parlib and its various abstractions and APIs sitting below a single parallel runtime.

(b) Lithe interposed between parlib and multiple parallel runtimes.

Figure 3.3: Lithe interposing on parlib to expose its own API for parallel runtime development. Figure (a) shows a single parallel runtime sitting directly on parlib. Figure (b) shows Lithe interposing on the parlib API to support the development of multiple, cooperating parallel runtimes.

3.1 Architectural Overview

Lithe itself is not a parallel runtime. However, from the perspective of parlib, Lithe *acts* as a parallel runtime, making library calls and implementing the necessary callbacks to request and yield vcores, manage uthreads, etc. Lithe does this in order to interpose on the API presented by parlib and expose its own API for parallel runtime development. Where parlib's API supports the development of a single parallel runtime, Lithe's API supports the development of multiple parallel runtimes that inter-operate in harmony. Figure 3.3 shows this interposition.

Lithe defines three basic primitives: **harts**, **contexts**, and **schedulers**. In terms of parlib, harts map 1-1 with vcores, contexts map 1-1 with uthreads, and schedulers map 1-1 with the notion of a user-level scheduler embedded in a parallel runtime. Just as with parlib, "lithe-aware" schedulers request access to harts and multiplex contexts on top of them. However, multiple schedulers can be active at a time (each with their own unique set of contexts), and Lithe facilitates the sharing of harts between them.

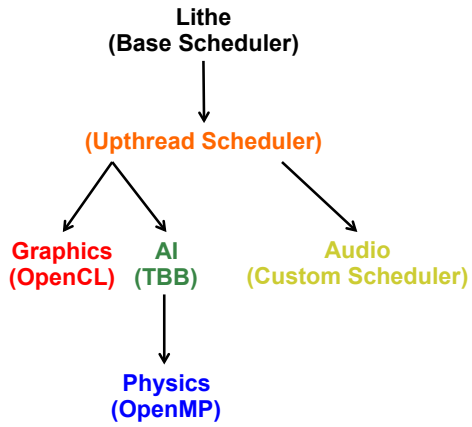
Schedulers are arranged in a hierarchy, with Lithe itself implementing a simple *base scheduler* at the root. The main thread of an application is associated with the base scheduler, and new schedulers are added to the hierarchy as function calls are made into libraries managed by different "lithe-aware" runtimes. In contrast to normal function calls, these calls

```
main() {
    upthread_sched_enter();
        p1 = upthread_create(renderScene);
        p2 = upthread_create(playMusic);
        upthread_join(p1, p2);
    upthread_sched_exit();
}

renderScene() {
    while(1) {
        Graphics.drawWorld();
        moveAvatar() {
            AI.decideWhereToGo() {
                Physics.calculate();
                …
            }
        }
    }
}

playMusic() {
    while(1) {
        Audio.playMusic();
    }
}
```

(a) Example program that calls functions inside various lithe-aware runtimes.



(b) The resulting scheduler hierarchy from the program shown on the left.

Figure 3.4: An example scheduler hierarchy in Lithe. Figure (a) shows an example program invoking nested schedulers. Figure (b) shows the resulting scheduler hierarchy from this program. The Graphics and AI schedulers originate from the same point because they are both spawned from the same context managed by the upthread scheduler.

can be thought of as "parallel" function calls because they trigger a new scheduler to be spawned and will not return until that scheduler has completed whatever task it was created to do. In the meantime, the newly spawned scheduler is free to request harts and multiplex contexts on top of those harts in order to complete its computation. Once the computation is complete, the scheduler removes itself from the hierarchy and returns from the original function call. The Lithe API exists to help developers implement their parallel runtimes such that they operate in this "lithe-aware" fashion. This is discussed in more detail in the following section.

The scheduler hierarchy itself can grow arbitrarily deep (and arbitrarily wide) as contexts from one parallel runtime continually make calls into libraries governed by other parallel runtimes. Figure 3.4 shows an example of such a hierarchy, based on the schedulers depicted in Figure 3.1.

At any given time exactly *one* scheduler is active on a given hart. As new schedulers are added to the hierarchy, they implicitly inherit whatever hart they are spawned from and begin executing immediately. Additional harts can be requested by each scheduler, but, unlike with parlib, these requests are not made to the system directly. Instead, child

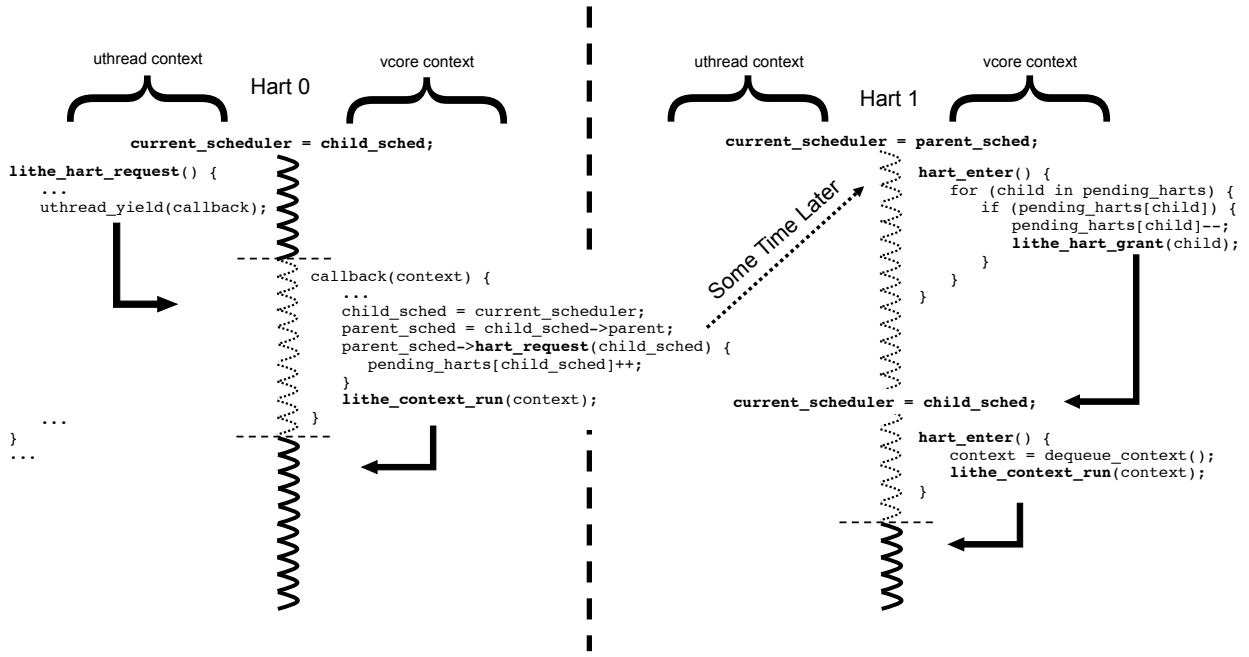


Figure 3.5: The process of requesting and granting a hart in Lithe. A child process first puts in a request through `lithe_hart_request()`, which triggers a `hart_request()` callback on its parent. The parent uses this callback to register the pending request and grant a new hart to its child at some point in the future. The initial request is initiated in uthread context, but all callbacks and the subsequent `lithe_hart_grant()` occur in vcore context.

schedulers request harts from their parent, and parents explicitly grant harts to their children. The only exception is the base scheduler, which actually does request harts directly from the system through parlib's vcore API. In this way, parent schedulers are able to precisely control how many harts they would like to *lend* to their children in order to perform computations on their behalf.

Just as with parlib, individual schedulers use the harts they have been granted to spawn contexts and schedule them for execution. As such, harts transition between running code in vcore context and running code in uthread context, with the additional notion of a `current_scheduler` attached to the code being executed. Lithe stores the value of the `current_scheduler` in a TLS variable associated with the hart and updates it every time a hart changes hands in the scheduler hierarchy. Tracking the value of `current_scheduler` is important because Lithe needs to know which scheduler to trigger a callback on whenever a context blocks or a notification comes in through parlib's asynchronous event delivery mechanism.

Once a scheduler is done with a hart, it yields it back to its parent. In this way, harts trickle up and down the scheduler hierarchy as children continually request and yield harts back to their parent. Eventually harts make their way back to the base scheduler and are yielded to the system through parlib's vcore API.

Individual schedulers do not make direct calls into other schedulers. Instead, all calls are made through the Lithe API, and Lithe dispatches those calls to the appropriate scheduler in the form of callbacks. For example, consider Figure 3.5, which shows the process of requesting a hart through the Lithe API. First, a context running on behalf of a child scheduler issues a call to `lithe_hart_request()` to request an additional hart. Internally, Lithe implements this call to yield to vcore context and trigger a `hart_request()` callback on the scheduler's parent. Since the parent does not own the current hart (the child does), the parent uses this callback to simply register the request in a hash table and returns. At some point later, a hart becomes available in the parent, and it checks to see if it has any pending requests from its children. If it does, it calls `lithe_hart_grant()` to transfer control of the hart to its child. Internally, Lithe updates the value of `current_scheduler` and triggers the `hart_enter()` callback on the child. The child is then free to do with the hart as it pleases (i.e. use it for its own computation or pass it down to its own child schedulers if it has any). In this way, the `hart_enter()` callback is analogous to the `sched_entry()` callback in parlib.

Just as with parlib, some of Lithe's API calls can be made from either uthread context or vcore context, but all callbacks run *exclusively* in vcore context. As we saw in the example above, a library call made in uthread context will transition to vcore context before the callback is issued. This differs from the original Lithe design which had no notion of vcore context. Transition stacks were sometimes used to execute certain callbacks, but their use was inconsistent and non-uniform. By running *all* callbacks in vcore context, developers have a consistent notion of what they can and can't do in these callbacks (see Section 2.2.1 for a reminder of the restrictions placed on code running in vcore context). In the following sections, we elaborate on the full set of API calls provided by Lithe.

3.2 The Lithe API

Lithe provides a *bi-directional* API similar to parlib's. Developers use this API to build "lithe-aware" runtimes that expose an outward facing API of "parallel" function calls (of the type described in the previous section). On one side of this API are *library calls*, implemented by Lithe, that parallel runtimes call in order to have Lithe perform an operation on their behalf. On the other side are *callbacks*, which are declared by Lithe, but implemented by each of the schedulers that get spawned whenever a "parallel" function call is invoked. Depending on the library call, these callbacks are triggered on either the `current_scheduler`, its parent, or one of its children in the hierarchy of schedulers maintained by Lithe.

Many of Lithe's library calls and callbacks mimic those defined by parlib's vcore and uthread APIs. Examples include the ability to request and yield harts (vcores), start and stop contexts (uthreads), etc. However, Lithe provides additional calls and callbacks that allow individual schedulers to track other schedulers in the scheduler hierarchy and explicitly transfer harts among them. Table 3.1 shows the full set of Lithe library calls and the corresponding callbacks they trigger. Each callback is annotated with the particular scheduler on which it is triggered (i.e. `parent`, `child`, or `current_scheduler`).

Hart Management	
Library Call	**Callback**
lithe_hart_request(amt)	parent->hart_request(amt)
lithe_hart_grant(child)	child->hart_enter()
lithe_hart_yield()	parent->hart_return() parent->hart_enter()

Scheduler Management	
Library Call	**Callback**
lithe_sched_init(sched, callbacks, main_ctx)	-
lithe_sched_enter(child)	parent->child_enter(child) child->sched_enter()
lithe_sched_exit()	parent->child_exit(child) child->sched_exit()
lithe_sched_current()	-

Context Management	
Library Call	**Callback**
lithe_context_init(ctx, entry_func, arg)	-
lithe_context_recycle(ctx, entry_func, arg)	-
lithe_context_reassociate(ctx, sched)	-
lithe_context_cleanup(ctx)	-
lithe_context_run(ctx)	-
lithe_context_self()	-
lithe_context_block(callback, arg)	current_scheduler->context_block(ctx) callback(ctx, arg) current_scheduler->hart_enter()
lithe_context_unblock(ctx)	current_scheduler->context_unblock(ctx) current_scheduler->hart_enter()
lithe_context_yield()	current_scheduler->context_yield(ctx) current_scheduler->hart_enter()
lithe_context_exit()	current_scheduler->context_exit(ctx) current_scheduler->hart_enter()

Table 3.1: The full set of Lithe library calls and the corresponding callbacks they trigger. The calls are organized by those that pertain to hart management, scheduler management, and context management respectively. The callbacks are annotated with the particular scheduler on which they are triggered (i.e. `child`, `parent`, or `current_scheduler`).

Before explaining each of these calls in detail, it is useful to walk through an example of how a parallel runtime can use them to implement a "parallel" function call. The pseudo code below demonstrates this:

```
void parallel_function() {
    lithe_context_t main_ctx;
    sched = sched_alloc();
    lithe_sched_init(sched, callbacks, &main_ctx);
    lithe_sched_enter(sched);
    for (int i = 0; i < num_contexts; i++) {
        ctx = context_alloc();
        lithe_context_init(ctx, do_work, i);
        enqueue_context(sched, ctx);
    }
    lithe_hart_request(num_contexts);
    join_on_all_contexts();
    lithe_sched_exit();
}
```

The basic logic flows as follows:

- **Enter a new scheduler:** Allocate a new scheduler, initialize it, and enter it. Upon entering the scheduler, Lithe will transfer control of the current hart over to it, add it to the scheduler hierarchy, and set the value of `current_scheduler` to it. The scheduler itself is initialized with a full set of callback functions and a place to store the current context in case it is blocked at some point in the future.
- **Create and enqueue `num_contexts` contexts:** Allocate `num_contexts` contexts, initialize them to perform the `do_work()` function, and add them to a queue associated with the newly allocated scheduler. This code assumes that the scheduler's `hart_enter()` callback is implemented such that it knows how to dequeue contexts from this queue and execute them.
- **Request `num_contexts` harts:** As these harts trickle in, their corresponding `hart_enter()` callbacks will be called and they will dequeue contexts and execute them.
- **Join on all outstanding contexts:** This code assumes that the join operation will block the currently executing context until all other contexts have completed. Upon blocking, Lithe will trigger the current hart's `hart_enter()` callback, giving it the opportunity to dequeue a different context and execute it. After all other contexts have completed, the blocked context will be unblocked and allowed to continue executing.
- **Exit the current scheduler:** Internally, Lithe will remove the current scheduler from the scheduler hierarchy and set the value of `current_scheduler` to that scheduler's parent (i.e. the original scheduler that first initiated the parallel function call).
- **Return from the parallel function call.**

3.2.1 Lithe Callbacks

A `lithe_sched_funcs` struct is declared to hold the full set of callbacks that schedulers spawned from a "lithe-aware" runtime must implement.

```
struct lithe_sched_funcs {
  void (*hart_enter)();
  void (*hart_request)(child, amt);
  void (*hart_return)(child);
  void (*child_enter)(child);
  void (*child_exit)(child);
  void (*sched_enter)(child);
  void (*sched_exit)(child);
  void (*context_block)(context);
  void (*context_unblock)(context);
  void (*context_yield)(context);
  void (*context_exit)(context);
};
```

Conceptually, these callbacks serve a similar purpose as the callbacks defined in parlib's global `sched_ops` struct. However, unlike `sched_ops`, which can only be implemented by a single scheduler, Lithe allows multiple schedulers to coexist, each with their own definitions of the `lithe_sched_funcs` callbacks. Lithe itself implements parlib's global `sched_ops` struct and serves to dispatch the `lithe_sched_funcs` callbacks to the proper scheduler on a given hart. As we saw in the example above, these callbacks are associated with a scheduler at the time it is initialized through a call to `lithe_sched_init()`.

Callback Types

In general, these callbacks are triggered by Lithe under one of three conditions:

1. To transfer control of a hart from one scheduler to another
2. To update state in a scheduler other than the `current_scheduler` without actually transferring control of the hart to it.
3. To yield control to the `current_scheduler` in vcore context when blocking, yielding, or exiting a lithe context.

When a scheduler makes a call that transfers control of a hart to another scheduler, Lithe internally updates the value of `current_scheduler` and triggers `hart_enter()` on the scheduler receiving the hart. The library calls that trigger this type of callback are `lithe_hart_grant()` (to transfer control from parent to child) and `lithe_hart_yield()` (to transfer control from child to parent). The original version of Lithe aptly referred to this type of callback as a *transfer callback*.

When a scheduler makes a call to update state in another scheduler, Lithe internally drops into vcore context, triggers the callback on the other scheduler, and then restores the original calling context on the hart after the callback completes. An example of this type of callback is `hart_request()`, which is triggered on a parent scheduler in response to a child issuing a `lithe_hart_request()` call. As we saw in the example from Figure 3.5, this callback is implemented by the parent to simply mark down the request for later and return. Only once an actual hart becomes available at some point in the future is a transfer of control initiated. Another example is the `child_enter()` callback, which informs a parent scheduler when one of its contexts spawns a new child scheduler in response to a making a "parallel" function call. This callback gives the parent scheduler the opportunity to track the child until a subsequent `child_exit()` callback is triggered. The original version of Lithe aptly referred to this type of callback as an *update callback*.

Callbacks that trigger a transfer of control to the current scheduler include `context_block()`, `context_yield()` and `context_exit()`. Each of these callbacks is triggered in direct response to `lithe_context_block()`, `lithe_context_yield()`, and `lithe_context_exit()` respectively. After each of these callbacks is triggered, an implicit call to `current_scheduler->hart_enter()` is triggered to force the current scheduler to make a new scheduling decision.

Handling Asynchronous Events

In general, all of Lithe's callbacks are paired with one or more library calls and are only ever triggered in direct response to issuing one of those calls. However, Lithe may also periodically trigger the `context_unblock()` callback in response to an asynchronous event from one of parlib's asynchronous services, independent of issuing a direct library call.

Normally the `context_unblock()` callback is issued just like any other callback: in response to a `lithe_context_unblock()` library call. However, Lithe also triggers it in response to syscall completion events and events notifying Lithe about core preemption as well. Recall from Section 2.3 that syscall completion events can be used to wake uthreads that have "blocked" on a specific syscall in user-space. Likewise, core preemption events hold a reference to whatever uthread was running on a core at the time it was preempted (assuming it was running in uthread context at all). Lithe implements handlers for these events to trigger the `context_unblock()` callback for the uthread associated with each of these events. For syscall completion events, Lithe unblocks the context waiting on the syscall; for preemption events, Lithe unblocks the preempted context.

In both of these cases, the event handlers may end up running on a hart whose `current_scheduler` is different from the scheduler associated with the context being unblocked. When this occurs, the value of `current_scheduler` is temporarily set to the scheduler associated with the blocked context before issuing a `context_unblock()` callback on that scheduler. Much like the `uthread_runnable()` callback from parlib, `context_unblock()` is an *update* callback and is used to unblock a context and make it schedulable again (e.g. put it on a queue for later execution, but not start running it right now). Once this callback

returns, the value of `current_scheduler` is restored, and the hart continues where it left off in the original scheduler.

This use of the `context_unblock()` callback is a special case in Lithe, and is the only callback that resembles something like the *externally triggerable* callbacks from parlib. In general, Lithe itself takes care to implement all of the externally-triggerable callbacks from parlib and only issue callbacks to "lithe-aware" schedulers in response to direct library calls. In the following section we discuss each of these callbacks along side the library calls that actually trigger them.

3.2.2 Lithe Library Calls

Table 3.1 lists the full set of Lithe library calls. As we saw in the example of implementing a "parallel" function call above, lithe-aware runtimes use these library calls to create new schedulers, request/yield harts, spawn new contexts, etc. As such, we separate our discussion of these calls into **scheduler management**, **hart management**, and **context management** respectively. All calls are made in the context of the `current_scheduler`, but depending on the call, one or more callbacks may be triggered on either the `current_scheduler`, its `parent`, or one of its `children` in the scheduler hierarchy. We discuss each of these library calls and the callbacks they trigger below.

For reference, we also provide a table that summarizes which library calls in the new Lithe API correspond to library calls in the original Lithe API from 2010 [3]. All of the original calls have corresponding calls in the new API that implement similar functionality, but many of their names have changed to more adequately describe the functionality they provide. Additonally, the new API has been expanded to include calls that didn't exist in the original API. This information is provided without further discussion in Table 3.2 at the end of this section.

Scheduler Management

The scheduler management functions serve to initialize new lithe-aware schedulers, as well as enter and exit them every time a new "parallel" function call is made. To facilitate this, Lithe defines a basic `lithe_sched_t` type that it operates on internally. However, Lithe never allocates this structure itself. Instead, Lithe follows a pattern similar to the one used for the `uthread_t` data structure defined by parlib. Parallel runtimes implemented on top of Lithe "extend" the `lithe_sched_t` struct, allocate space for it themselves, and then pass it to the various scheduler management functions discussed below. In this way, arbitrary schedulers can be defined with their own custom fields, but references to them can be passed to Lithe and treated as basic `lithe_sched_t` types. When these schedulers are later passed to their corresponding runtimes via callbacks, they can be cast to the proper type and operated on accordingly. Lithe defines something similar for its contexts with the `lithe_context_t` type, which, in turn, is already extended from parlib's basic `uthread_t` type.

We saw a limited version of these structures being allocated in the pseudo code for the example "parallel" function call discussed previously. A more complete example will be seen with the definition of the `lithe_fork_join_sched_t` and `lithe_fork_join_context_t` structures of the "Fork-Join" scheduler discussed in Section 3.3.

Library Call	Callback
`lithe_sched_init(sched, callbacks, main_ctx)`	-
`lithe_sched_enter(child)`	`parent->child_enter(child)` `child->sched_enter()`
`lithe_sched_exit()`	`parent->child_exit(child)` `child->sched_exit()`
`lithe_sched_current()`	-

`lithe_sched_init`: This function serves to initialize a new lithe-aware scheduler for insertion into the scheduler hierarchy. As parameters, it takes references to the various structures required to later "enter" the scheduler via a call to `lithe_sched_enter()`. These parameters consist of an "extended" `lithe_sched_t` struct, a set of callbacks, and an "extended" `lithe_context_t` struct. The "extended" `lithe_sched_t` struct is used to hold a reference to the runtime-specific scheduler being initialized, and the callbacks structure defines the runtime-specific callback functions associated with that scheduler. The "extended" `lithe_contex_t` structure is necessary for "promoting" the current context to the proper context type once the scheduler is "entered" later on. Without this promotion, the current context would continue to exist as a context type associated with the parent scheduler (which would still be extended from a basic `lithe_context_t` type, but may have a different size, set of fields, etc.). This would have obvious negative consequences if passed to the various context-related callbacks discussed later on. No callbacks are triggered as a result of this call.

`lithe_sched_enter`: This function causes a previously initialized scheduler to be added to the scheduler hierarchy. Internally, Lithe sets the value of `current_scheduler` to the new scheduler, adds it to the scheduler hierarchy, and promotes the current context to the context struct passed in when the scheduler was initialized. As a result of this call, two callbacks are issued on the current hart: one on the parent and one on the child. The parent receives the `child_enter()` callback (to indicate that a *child* has been added to the scheduler hierarchy) and the child receives the `sched_enter()` callback (to indicate that the `current_scheduler` has just been added to the scheduler hierarchy). In practice, the parent uses `child_enter()` to add the child to a list of children so it can keep track of future hart requests that come in from that child. The child uses `sched_enter()` to do any runtime initialization necessary before actually taking control of the hart and beginning to execute in the context of the new scheduler.

`lithe_sched_exit`: This function exits the current scheduler and removes it from the scheduler hierarchy. Internally, Lithe undoes the "promotion" operation it performed previously:

it reassociates the current context with the context struct of the parent that was taken over upon entering the scheduler. It then sets the `current_scheduler` back to the parent, waits for all harts from the child to be yielded, and removes the child from from the scheduler hierarchy. Just as `lithe_sched_enter()` triggered two callbacks (one on the parent and one on the child), `lithe_sched_exit()` follows a similar pattern. The parent receives the `child_exit()` callback (to indicate that a *child* has been removed from the scheduler hierarchy) and the child receives the `sched_exit()` callback (to indicate that the `current_scheduler` has just been removed from the scheduler hierarchy). In practice, the parent uses `child_exit()` to remove the child from its list of children, and the child uses `sched_exit()` to cleanup any initialization done in `sched_enter()` (e.g free any allocated memory, etc.).

`lithe_sched_current:` This function simply returns a pointer to the current scheduler on the current hart. It is nothing more than an API wrapper around the `current_scheduler` TLS variable on the hart. No callbacks are triggered as a result of this call.

Hart Management

The hart management functions serve to transfer control of harts between child schedulers and their parents. Calls exist to request harts from their parent, grant them to down to a child, and yield them back to their parent. Each of these calls and the callbacks they trigger are discussed in detail below.

Library Call	Callback
`lithe_hart_request(amt)`	`parent->hart_request(amt)`
`lithe_hart_grant(child)`	`child->hart_enter()`
`lithe_hart_yield()`	`parent->hart_return()` `parent->hart_enter()`

`lithe_hart_request:` This call is used to incrementally request new harts from a parent. However, its semantics differ slightly from both the original version of Lithe and its `vcore_request()` counterpart in parlib. Rather than being a true incremental request (i.e. give me x more harts if they become available), `lithe_hart_request()` can also accept *negative* values as its request amount. By allowing negative values, Lithe gives schedulers the ability to incrementally change their total desired hart count to match their true real-time need (even if that need is now less than it was before). With the old interface, there was no way to encode this information, causing unneeded harts to continuously trickle into a child even if it no longer needed them. In practice, schedulers call `lithe_hart_request()` with an amount that reflects an adjustment in the number of *runnable* contexts that it currently has. That is, they call it with a positive value when new contexts are first created or unblocked, and they call it with a negative value whenever a context blocks or exits. As such, these calls are typically made in the `context_block()` and `context_unblock()` callbacks discussed later on.

As discussed previously, a call to `lithe_hart_request()` will trigger a `hart_request()` callback on the `current_scheduler`'s parent. Internally, Lithe will pass this callback a pointer to the scheduler making the request as well as the incremental request amount. It is up to the parent to make sure it honors this request according to the semantics described above.

As discussed above, the `hart_request()` callback is an *update* callback, so the `current_scheduler` will retain control of the hart after the callback has completed. If the call was issued from uthread context, that context will also be restored and continue where it left off.

`lithe_hart_grant:` This call explicitly grants a hart from a parent to a child. As parameters, it takes the child scheduler to grant the hart to as well as a *call-site-specific* callback that will be triggered once Lithe has completed the transfer internally. Lithe will first update the value of `current_scheduler` to the new child scheduler, trigger the call-site-specific callback, and then trigger the `hart_enter()` callback on a child. In practice, this call is only ever made in the body of `hart_enter()` when a parent detects that it has children in need of a hart (as demonstrated in Figure 3.5). The call-site-specific callback is typically used to implement the bottom-half of any scheduler specific synchronization needed to protect the transfer of the hart (i.e. release locks, down reference counts, etc.).

`lithe_hart_yield:` This call yields control of the current hart back to the parent. It can be called from either uthread context or vcore context, but in practice it tends to only be called from the bottom of the `hart_enter()` callback once it decides it has nothing else to do.

Once issued, Lithe triggers the `hart_return()` callback on the parent scheduler to inform it that a hart is about to be returned to it. The parent can use this callback to update its total hart count and otherwise prepare for the return of the hart. Once this call completes, Lithe will internally update the value of `current_scheduler` to the parent and jump to its `hart_enter()` callback. We separate the `hart_return()` and `hart_enter()` callbacks in order to maintain the invariant that `hart_enter()` is the single entry point into a hart (much like `vcore_entry()` is in parlib) as well as to allow Lithe to internally update some state between triggering `hart_return()` and `hart_enter()`.

Context Management

The context management functions serve to initialize new contexts, start and stop them, block and unblock them, etc. Internally, Lithe references these contexts using a basic `lithe_context_t` struct that is "extended" from parlib's underlying `uthread_t` type. Just as with `lithe_sched_t` structs, Lithe never allocates space for these structs itself. Parallel runtimes implemented on top of Lithe "extend" the `lithe_context_t` type to contain runtime specific fields and then allocate space for them before passing them to the various library calls discussed below. When these contexts are passed back to their corresponding runtimes via callbacks, they can be cast to the proper type and operated on accordingly.

Library Call	Callback
lithe_context_init(ctx, entry_func, arg)	-
lithe_context_recycle(ctx, entry_func, arg)	-
lithe_context_reassociate(ctx, sched)	-
lithe_context_cleanup(ctx)	-
lithe_context_run(ctx)	-
lithe_context_self()	-
lithe_context_block(callback, arg)	current_scheduler->context_block(ctx) callback(ctx, arg) current_scheduler->hart_enter()
lithe_context_unblock(ctx)	current_scheduler->context_unblock(ctx) current_scheduler->hart_enter()
lithe_context_yield()	current_scheduler->context_yield(ctx) current_scheduler->hart_enter()
lithe_context_exit()	current_scheduler->context_exit(ctx) current_scheduler->hart_enter()

lithe_context_init: This call is used to initialize a context with a start function (and its argument) so that it can be started at some point in the future. As parameters, it takes a reference to an "extended" lithe_context_t struct and the start function and argument it should execute once it begins to execute. It is assumed that memory for the context has already been allocated and that the stack field of the context is already pointing to a valid stack before being passed to this function. As part of initializing the context, Lithe assign it a unique context id and call uthread_init() under the hood, which will (among other things) create a new TLS region for the context to run with. A subsequent call to lithe_context_run() will trigger this context to being executing on its stack at the start function specified with the newly allocated TLS installed. This call should be paired with the lithe_context_cleanup() function described below so Lithe has a chance to cleanup any internal state associated with the context. No callbacks are triggered as a result of this call.

lithe_context_recycle: This call is used to recycle a previously initialized context. Just like lithe_context_init(), it also takes a start function and argument as parameters and a subsequent call to lithe_context_run() will cause it to being executing at this start function. However, lithe_context_recycle() does not call uthread_init() under the hood (leaving its TLS unchanged), and the context id remains the same as it was previously. Schedulers that use this library call must be careful to reinitialize any relevant TLS variables as desired. In practice, we use this call in our Lithe port of OpenMP because applications tend to make assumptions about the TLS *not* being reinitialized in threads used across different OpenMP parallel regions (even though the spec doesn't guarantee this). Whether this a bug or a feature is up for debate, but the lithe_context_recycle() call exists nonetheless to allow these semantics to persist. A context can be recycled as many times as desired before

ultimately calling `lithe_context_cleanup()` once it is no longer needed. No callbacks are triggered as a result of this call.

`lithe_context_reassociate:` This call is used to reassociate a context with a new scheduler. Typically contexts are created specifically for use by a particular scheduler instance and then destroyed at some point before that scheduler is exited. This call allows a blocked context to be passed to a new scheduler instance so that when it resumes it will begin executing on behalf that new scheduler. This call is typically paired with a subsequent call to `lithe_context_recycle()` in order to reinitialize the context within the new scheduler (though this is not strictly necessary). In practice, we use this call in our port of OpenMP to reuse threads across different OpenMP parallel regions. No callbacks are triggered as a result of this call.

`lithe_context_cleanup:` This call cleans up all internal state associated with a previously initialized context. It should be called exactly *once* for each context initialized via a call to `lithe_context_init()`. After this call completes *no* subsequent calls should be made to `lithe_context_recycle()` or `lithe_context_reassociate()`, and it is safe to free all memory associated with the context (including its stack). No callbacks are triggered as a result of this call.

`lithe_context_run:` This call is used to start executing a previously initialized context. This call is used to both start a newly initialized context as well as resume a context that has been yielded in some way. No callbacks are triggered as a result of this library call.

`lithe_context_self:` This call is used to return a unique id associated with a context. The actual type returned is opaque (though in practice it is just a pointer to the context's underlying `lithe_context_t` struct). No callbacks are triggered as a result of this library call.

`lithe_context_block:` This call is used to block the currently executing context. As parameters, it takes a *call-site-specific* callback to execute once the context has been blocked. Under the hood Lithe calls `uthread_yield()` in order to yield the context to the current scheduler in vcore context. It then triggers the `context_block()` callback on the current scheduler before executing the call-site-specific callback that was passed in. Once both callbacks have completed, Lithe triggers the `hart_enter()` callback, giving the scheduler the opportunity to make a new scheduling decision.

`lithe_context_unblock:` This call is used to unblock a previously blocked context. Under the hood Lithe calls `uthread_runnable()` in order to trigger parlib's `thread_runnable()` callback internally. Lithe then reflects this callback through `context_block()` so the current scheduler can reenqueue the context for later execution.

`lithe_context_yield:` This call yields the currently executing context to the current scheduler, triggering a `context_yield()` callback as a result. Just like `lithe_context_block()`,

it calls `uthread_yield()` under the hood, but passes a different *call-site-specific* callback in order to reflect `context_yield()` back to the scheduler instead of `context_block()`.

`lithe_context_exit:` This call is identical to `lithe_context_yield()` except that it triggers the `context_exit()` callback instead of `context_yield()`.

3.2.3 Lithe Synchronization Primitives

We provide the same synchronization primitives as the original version of Lithe (i.e. mutexes, condition variables, and barriers) with additional functionality provided for futexes as well. Providing these synchronization primitives was trivial because we were able to leverage parlib's implementations of them instead of reimplementing them specifically for Lithe. All we had to do was provide wrappers around them, e.g.:

```
int lithe_mutex_lock(lithe_mutex_t *mutex)
{
  return uthread_mutex_lock(mutex);
}
int lithe_mutex_unlock(lithe_mutex_t *mutex)
{
  return uthread_mutex_unlock(mutex);
}
```

The reason this works so nicely is that all Lithe contexts are actually uthreads under the hood, and Lithe implements the `uthread_has_blocked()`, `uthread_paused()`, and `uthread_runnable()` callbacks from parlib's global `sched_ops` struct to simply reflect them back through corresponding `lithe_sched_funcs` callbacks on the `current_scheduler`. Specifically, Lithe implements `uthread_has_blocked()` to reflect `context_block()` and `uthread_paused()` and `uthread_runnable()` to reflect `context_unblock()`.

```
void lithe_thread_has_blocked(uthread_t *uthread)
{
  current_scheduler->funcs->context_block(current_scheduler, uthread);
}
void lithe_thread_runnable(uthread_t *uthread)
{
  current_scheduler->funcs->context_unblock(current_scheduler, uthread);
}
void lithe_thread_runnable(uthread_t *uthread)
{
  uthread_runnable(uthread);
}
```

Since all of parlib's synchronization primitives only ever trigger one or the other of these callbacks, our job is done from the perspective of Lithe.

New Lithe Library Call	Original Lithe Library Call
lithe_hart_request(nharts)	lithe_sched_request(nharts)
lithe_hart_grant(child)	lithe_sched_enter(child)
lithe_hart_yield()	lithe_sched_yield()
lithe_sched_init(sched, callbacks, main_ctx) lithe_sched_enter(sched)	lithe_sched_register(callbacks)
lithe_sched_exit()	lithe_sched_unregister()
lithe_sched_current()	-
lithe_context_init(ctx, entry_func, arg)	lithe_ctx_init(ctx, stack)
lithe_context_recycle(ctx, entry_func, arg)	-
lithe_context_reassociate(ctx, sched)	-
lithe_context_cleanup(ctx)	lithe_ctx_fini(ctx)
lithe_context_run(ctx)	lithe_ctx_run(ctx, fn) lithe_ctx_resume(ctx)
lithe_context_block(callback, arg)	lithe_ctx_block(ctx)
lithe_context_unblock(ctx)	lithe_ctx_unblock(ctx)
lithe_context_yield()	lithe_ctx_pause(fn)
lithe_context_exit()	-
lithe_context_self()	-
Builtin __thread support or set_dtls(key, storage)	lithe_ctx_setcls(cls)
Builtin __thread support or get_dtls(key)	lithe_ctx_getcls()

New Lithe Callback	Original Lithe Callback
hart_enter()	enter()
hart_request(child, amt)	request(child, nharts)
hart_return(child)	-
child_enter(child)	register(child)
child_exit(child)	unregister(child)
sched_enter(child)	-
sched_exit(child)	-
context_block(context)	block(ctx)
context_unblock(context)	unblock(ctx)
context_yield(context)	yield(child)
context_exit(context)	-

Table 3.2: A side-by-side comparison of library calls and callbacks from the new Lithe API and the original Lithe API as presented in [3].

3.3 The Fork-Join Scheduler

The Lithe "fork-join" scheduler provides a baseline scheduler to help implement the parallel function call pattern presented in Section 3.2. It provides both reasonable defaults for common scheduling operations, as well as a number of API calls that make it easily extendable for schedulers that need some level of custom functionality. So long as the runtime being ported is fairly well behaved, basic parallel function calls can be implemented without customization, as depicted below.

```
void parallel_function() {
    lithe_fj_context_t main_ctx;
    sched = lithe_fj_sched_create(main_ctx);
    lithe_sched_enter(sched);
    for (int i = 0; i < num_contexts; i++) {
        lithe_fj_context_create(sched, stack_sz, entry, i);
    }
    lithe_hart_request(num_contexts);
    lithe_fj_sched_join_all();
    lithe_sched_exit();
    lithe_fj_sched_destroy(sched);
}
```

The "lithe-aware" ports of TBB and OpenMP in the original version of Lithe were built independent of each other and no attempt was made to find a common abstraction between the two. In our new version of Lithe, both of these libraries are able to leverage the "fork-join" scheduler. As such, we were able to provide much cleaner ports of TBB and OpenMP than was done previously. The TBB port was able to use the "fork-join" scheduler without modification. The OpenMP port was able to leverage most of the "fork-join" functionality, with a few extended API calls and custom implementations for its `context_block()` and `context_exit()` callbacks. These customizations were required to accommodate assumptions in OpenMP about reusing contexts across multiple parallel function calls (so that their TLS could be preserved). These customizations could have been avoided at the cost of heavily modifying existing OpenMP code. In the end, we opted for less changes to the core OpenMP library code and a slightly customized scheduler to accommodate it.

Additionally, we have used the "fork-join" scheduler to provide a new "lithe-aware" port of our parlib-based pthreads library discussed in Section 2.6.1 (upthreads). The "fork-join" scheduler itself was modeled after our original upthread implementation, so all of the optimizations discussed in Section 2.6.1 for per-vcore run queues, cpu affinity, and careful cache alignment apply here as well.

In general, the pthreads API does not directly lend itself to a true "fork-join" model in terms of launching parallel function calls and joining on them. However, if we force our "lithe-aware" implementation of upthreads to *always* exists at the root of the Lithe sched-

uler hierarchy, then the we are able to make it follow a common "fork-join" pattern on an application-wide basis. We simply create our "lithe-aware" upthread scheduler before entering main (using a constructor function) and destroy it just before the application exits (using a destructor function). Any pthreads created throughout the lifetime of the application are then linked to to this single upthread scheduler instance, and are able to properly lend their harts to any other "lithe-aware" schedulers they invoke. The previous version of Lithe had no such ability to operate alongside pthread code.

3.4 Evaluation

To evaluate our new version of Lithe we recreated two experiments from the original Lithe paper [4]: the SPQR benchmark and the Flickr-like image processing server.

SQPR [30] is a sparse QR factorization algorithm used in the linear least-squares method to solve a variety of problems, including geodetic surveying, photogrammetry, tomography, structural analysis, surface fitting, and numerical optimization. It is used to quickly solve for x in the equation $Ax = b$, where x and b are column vectors, and A is a sparse matrix. For our experiments we measure the time it takes to calculate x using the `qrdemo` application from the SuiteSparse package [53]. This application takes the input matrix A as its only argument and sets b to a column vector of all $1s$ before solving for x. The SQPR algorithm itself is highly parallelizable and the SuiteSparse implementation relies on a mix of OpenMP and TBB to process its input matrix. As such, it relies on these libraries to interoperate (i.e. share cores) with as little interference from one another as possible.

The Flickr-like image processing server is an application that parallelizes the generation of multiple thumbnail images from a single input image. In total, nine thumbnail images are generated, and their sizes are consistent with those produced by Flickr when making thumbnail requests through its standard web API [54]. All thumbnails are generated in parallel (using pthreads), and each individual thumbnail is created using a parallel image processing algorithm from GraphicsMagick [55] (which uses OpenMP under the hood). As an extension to the original experiment, we have integrated the image processing server into our kweb web server (discusssed in Chapter 2.6.2) and serve web traffic along side processing each of the thumbnail generation requests. Input images are accepted via POST requests and all nine thumbnails are returned in a gzipped tarball.

In the sections that follow, we evaluate the performance of these applications and compare the results of running them natively on Linux to running them on Lithe. As with parlib, all experiments were run exclusively on Linux (rather than some combination of Akaros and Linux) to better isolate the effects of Lithe, and more effectively evaluate it independent of any problems that may exist in disjoint parts of Akaros. Likewise, all of our experiments were ran on 'c99', with the Linux kernel patches from Andi Kleen to expose the **rdfsbase** and **wrfsbase** instructions to user space (as described in Chapter 2.6.3). All hyperthreading was disabled for the duration of these experiments (leaving us with a total of 16 cores).

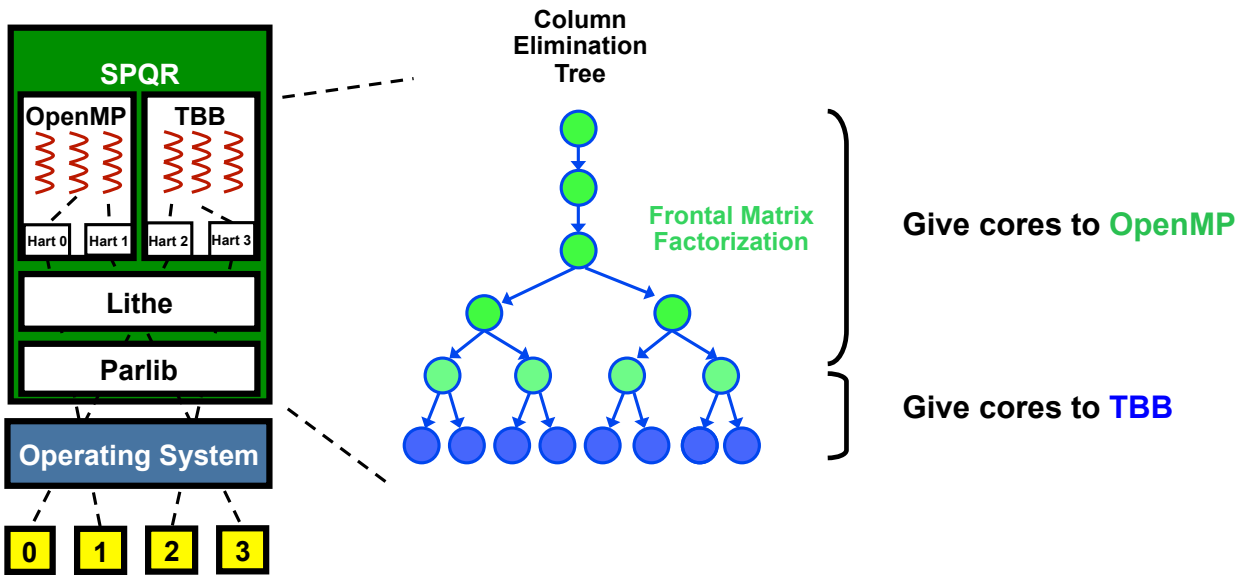


Figure 3.6: The software architecture of the SPQR benchmark on Lithe. Cores are shared between OpenMP and TBB at different levels of the SPQR task tree.

Running our experiments exclusively on Linux comes at the cost of only evaluating Lithe under the limited setup dictated by Linux's "faked" vcore abstractions (as discussed in detail in Chapter 2.2.1). However, so long as only one application is running in the system, this setup is sufficient to demonstrate the advantages of using Lithe over existing kernel-based scheduling approaches. For now, this is acceptable.

3.4.1 SPQR Benchmark

The SPQR algorithm operates by creating a tree of tasks, with each task responsible for performing a set of linear algebra matrix operations. At each level of the task tree, sibling tasks are parallelized using OpenMP, and the matrix operations within each task are parallelized using MKL (which uses TBB under the hood). Figure 3.6 shows how these tasks are broken up into their parallelizable components.

Without Lithe, cores must be statically partitioned between TBB and OpenMP to keep them from interfering with one another. Optimal partitioning is highly dependent on the input matrix, and some level of trial-and-error is required to find the best configuration to use for each input.

By design, Lithe removes the need for this static partitioning, simplifying the process of finding the optimal operating point for any given input matrix. Moreover, statically partitioning cores has the limitation of possibly leaving some cores unutilized in one library when they could be doing useful work in the other. As we show below, Lithe is able to overcome this limitation, outperforming versions of the code that have been hand-tuned with the best static partitioning possible.

Below is a list of the input matrices we used to evaluate SPQR on Lithe:

	Landmark	**DeltaX**	**ESOC**	**Rucci1**
Size	71,952 x 2,704	68,600 x 21,961	327,062 x 37,830	1,977,885 x 109,900
Nonzeros	1,146,868	247,424	6,019,939	7,791,168
Domain	surveying	computer graphics	orbit estimates	ill-conditioned least-square

These are the same matrices used by the SQPR evaluation in the original Lithe paper. They have been chosen to evaluate SPQR across a range of input sizes and application domains. For each matrix we ran it through the `qrdemo` application and measured the time it took to compute x. We compared the results from four different setups:

- **Out of the Box - Linux NPTL:** this setup is based on running SPQR directly on Linux without modification. Cores are split evenly between OpenMP and TBB (i.e. 8 cores each on our non-hyperthreaded 'c99' machine).
- **Manually Tuned - Linux NPTL:** this setup is based on partitioning cores between OpenMP and TBB in an optimal configuration. For each input matrix, we ran SQPR over every possible combination of splitting cores between OpenMP and TBB and chose the combination that was able to complete the fastest.
- **Out of the Box - upthread:** this setup is based on linking in our parlib-based upthread library as a drop in replacement for the Linux NPTL. As with the "Out of the Box - Linux NPTL" configuration, cores are split evenly between OpenMP and TBB (i.e. 8 cores each on our non-hyperthreaded 'c99' machine). We still use the default implementations of OpenMP and TBB for this configuration.
- **Lithe:** this setup is based on replacing the default OpenMP and TBB libraries with their "lithe-aware" counterparts and linking them together with Lithe. No partitioning of cores is necessary, and Lithe takes care of automatically sharing cores between the two libraries over time.

Figure 3.7 shows the results of this experiment. From a 2.3% speedup on the *landmark* matrix to a 10% speedup on *deltaX*, SPQR on Lithe outperforms the "Manually Tuned - Linux NPTL" setup by a significant margin on all input matrices. Lithe's ability to dynamically share cores betwen TBB and OpenMP provides these performance improvements without the need for static partitioning. These results reinforce the power of Lithe to efficiently manage cores between disparate parallel libraries in a single application.

However, Lithe is not the only setup worthy of note. The "Out of the Box –upthread" setup is only slightly different from the "Out of the Box - Linux NPTL" setup, but performs significantly better than it (in most cases better than the "Manually Tuned - Linux NPTL" setup as well). These results underscore the performance improvements possible just by moving over to a user-level scheduling approach independent of the advantages introduced by Lithe. This is especially important for applications that use libraries that have not yet been ported over to Lithe. So long as these libraries use pthreads, they can take advantage

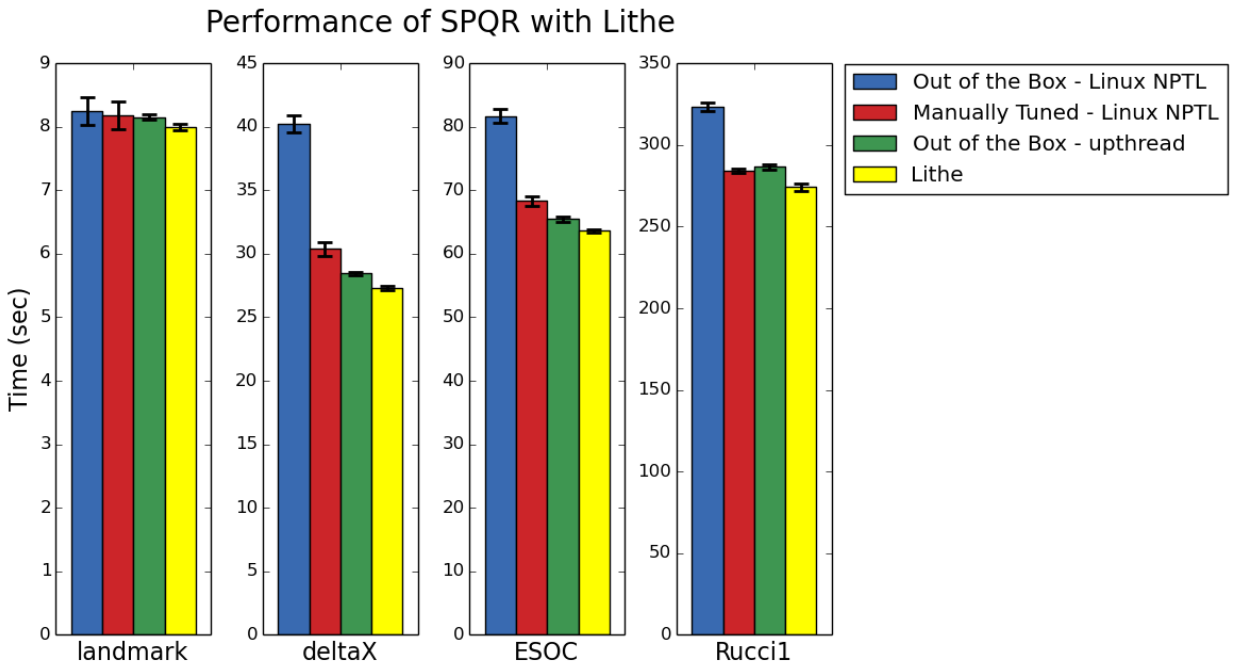


Figure 3.7: SPQR performance across a range of different input matrices. We compare its performance when running on the Linux NPTL (for both an 'out-of-the-box' configuration and a hand-tuned configuration based on optimal partitioning of cores) to running directly on parlib-based upthreads and Lithe. In all cases, Lithe outperforms them all.

of our new "lithe-aware" version of upthreads to coexist with other "lithe-aware" libraries in a single application. The image processing server discussed below demonstrates this.

3.4.2 Image Processing Server

As mentioned previously, we reimplemented the Flickr-like image processing server from the original Lithe paper as an add-on to our kweb webserver discussed in Chapter 2.6.2. We overloaded kweb's `url_cmd` interface to accept POST requests of an image file and return a gzipped tarball of nine thumbnails generated using GraphicsMagick. All nine thumbnails are generated in parallel using pthreads, and GraphicsMagick uses OpenMP to further parallelize the generation of each thumbnail at an algorithmic level. In this way, pthreads and OpenMP must share access to underlying cores in order to generate thumbnails efficiently. Moreover, since the server is now integrated with kweb, static file requests being serviced by pthreads must share access to these cores as well. The goal is to show that our "lithe-aware" implementations of upthreads and OpenMP are able to share cores more efficiently than running both libraries directly on the Linux NPTL. The experiment in the original Lithe paper was unable to demonstrate this capability because our "lithe-aware" version of upthreads had not yet been developed. Figure 3.8 shows the software architecture of this modified version of kweb.

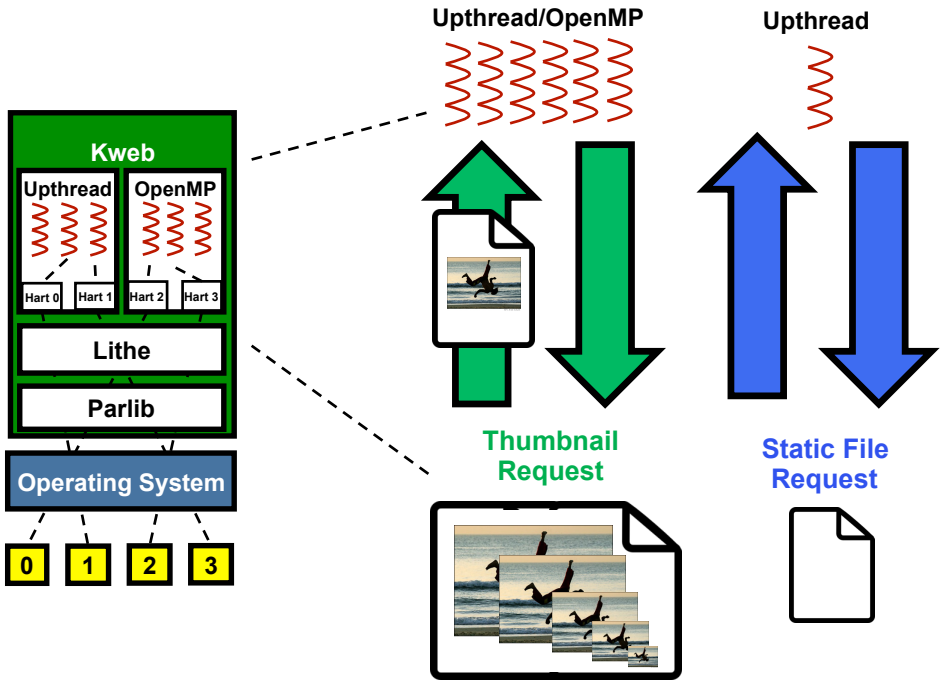


Figure 3.8: Integrating the Flickr-like image processing server into kweb. Static file requests serviced by pthreads must share cores with thumbnail requests serviced by a mix of pthreads and OpenMP.

To demonstrate our goal, we configured kweb to run under three different threading models: the Linux NPTL, parlib-based upthreads, and "lithe-aware" versions of upthreads and OpenMP. In all cases, we configured kweb to run with all of the cores in the system (i.e. 16 cores on 'c99' with hyperthreading disabled). We then measured the performance of kweb for each of these threading models when sending it a mix of static file requests and thumbnail requests.

As with our original kweb experiment, our static file requests were generated from a simple http-request generator we wrote. We configured our http-request generator to operate as follows:

- Start 100 TCP connections in parallel and prepare to send 1000 http-requests/connection.
- Batch requests to allow 100 concurrent requests/connection on the wire at a time.
- After each request receives a response, place a new request on the wire.
- After each connection completes all 1000 requests, start a new connection.

With this setup there are always approximately 10,000 outstanding static file requests on the wire at any given time. Each request consists of an HTTP 1.1 GET request for an empty file. Each response consists of a 242-byte HTTP 1.1 response header and a 0-byte body. By requesting a file of minimum size, we maximize the throughput we can measure before hitting the bandwidth limitations of the network card. The 242-byte header was chosen to match

the header size used in the original kweb throughput experiment (as dictated by nginx's standard response header size).

For our thumbnail requests, we wrote a separate thumbnail-request generator that we configured as follows:

- Generate TCP connections every 1/100th of a second, with 1 request per connection.
- Throttle our total number of outstanding connections to 32.

With this setup, thumbnail requests are spaced apart in order to give the server some time to process the previous request before starting on the next one. Moreover, we limit the maximum number of outstanding requests to 32 to keep from overloading the server with requests (since each thumbnail request takes much longer than a simple static file request). Requests themselves consist of a 3.96MB input image, and responses contain a 5.6MB gzipped tarball of all the generated thumbnail images.

We used the request generators described above to generate a mix of traffic to kweb in 5 minute intervals. The first interval consisted of only thumbnail requests, the second consisted of both thumbnail and static file requests, and the third consisted of only static file requests. We also included fourth and fifth intervals that repeated the first two intervals in reverse order to keep everything symmetric. By mixing requests in this way, we were able to see the effect that each of the request types had on the overall performance of kweb. Moreover, by including the reverse intervals, we were able to demonstrate kweb's ability to dynamically adapt to a mix of requests over time.

In total, we generated this traffic for three separate experiments: one for each of the different threading models we configured kweb to run under (i.e. the Linux NPTL, parlib-based upthreads, and "lithe-aware" versions of upthreads and OpenMP). We then computed the average throughput and average latency of each request type in 10 second increments for the duration of each experiment. Figure 3.9 shows the result of these experiments.

The graph in this figure is organized as follows:

- Horizontally, data is broken up into 5, 5-minute intervals, according to the traffic generation pattern described above. The first interval consists of only thumbnail requests, the second consists of both thumbnail and static file requests, and the third consists of only static file requests. The fourth and fifth intervals repeat the first two intervals in reverse order to keep everything symmetric.
- Each line in the graph represents either the average throughput or the average latency of a particular request type (i.e. static files, or thumbnails) for a given kweb threading model (i.e. the Linux NPTL, parlib-based upthreads, or "lithe-aware" versions of upthreads and OpenMP).
- The top 2 boxes contain the average throughput for each request type at 10 second intervals (first is thumbnails, second is static files).
- The bottom 2 boxes contain the average latency for each request type at 10 second intervals (first is thumbnails, second is static files).

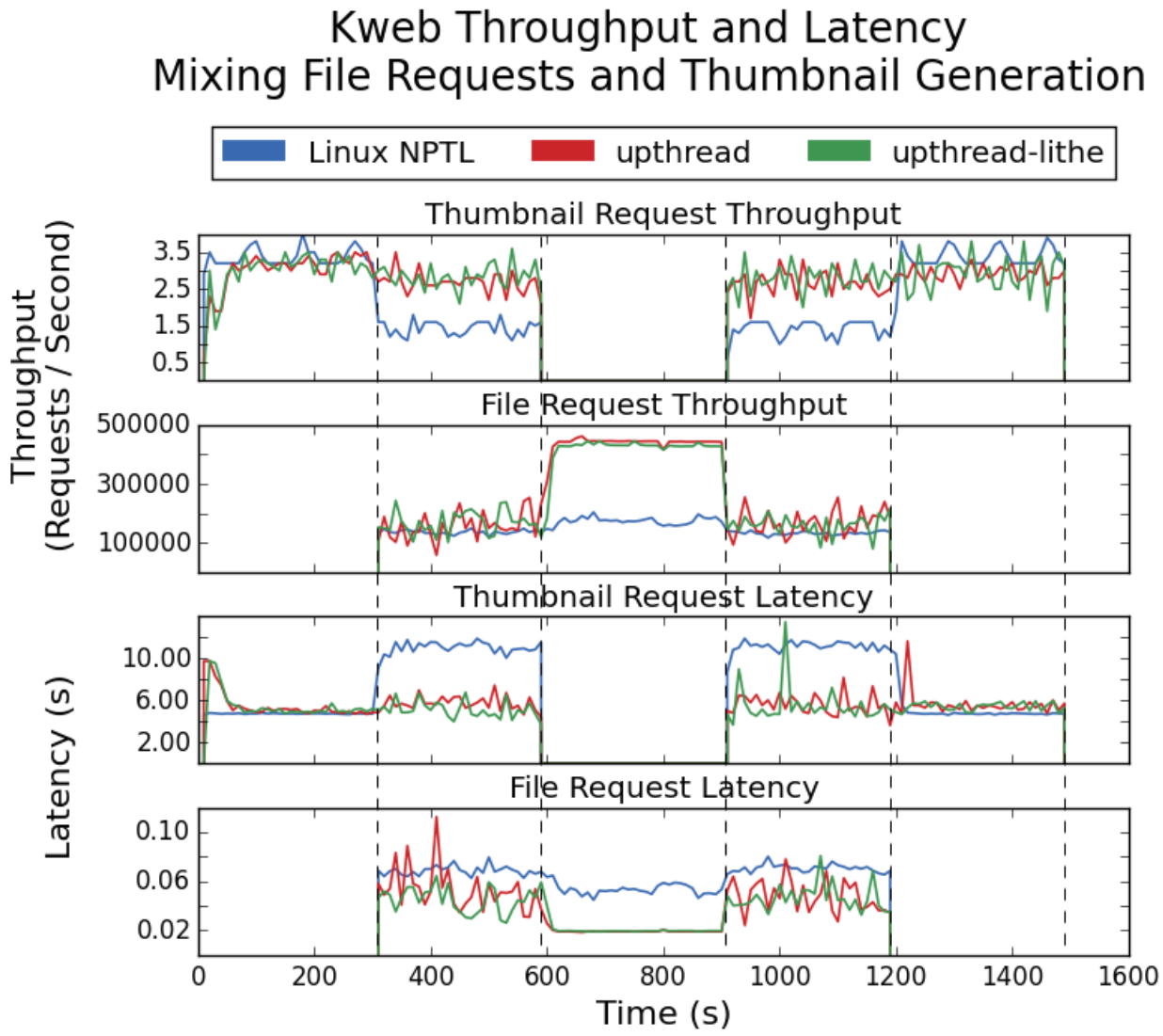


Figure 3.9: Mixing static file requests and thumbnail generation in kweb. Horizontally, data is broken up into 5, 5-minute intervals. The first interval consists of only thumbnail requests, the second consists of both thumbnail and static file requests, and the third consists of only static file requests. The fourth and fifth intervals repeat the first two intervals in reverse order to keep everything symmetric. Each line in the graph represents either the average throughput or the average latency of a particular request type (i.e. static files, or thumbnails) for a given kweb threading model (i.e. the Linux NPTL, parlib-based upthreads, or "lithe-aware" versions of upthreads and OpenMP). The top 2 boxes contain the average throughput for each request type at 10 second intervals (first is thumbnails, second is static files). The bottom 2 boxes contain the average latency for each request type at 10 second intervals (first is thumbnails, second is static files).

The major takeaways from this graph are:

- The throughput and latency for thumbnail generation during the first interval is comparable for all three threading models measured. The throughput is steady at around 3.0 requests/sec, and the latency is steady at around 5.1 sec/request. There is an unexplained spike at the beginning for both `upthread` and `upthread-lithe`, but it smooths out after that.
- With static file requests added during the second interval, the throughput and latency of static file requests is comparable for all three models at around 180,000 requests/sec and 60 msec/request, respectively. However, the throughput of thumbnail requests drops by 50% for the Linux NPTL, and its latency increases by 50%. The throughput of thumbnail requests for `upthread` and `upthread-lithe` only drops by about 6%, and the latency remains roughly the same, with a slight increase in variance.
- With thumbnail generation removed during the third interval, the throughput of static file requests for `upthread` and `upthread-lithe` jumps by 63% to around 492,000 requests/sec (which is the same result we saw in the previous chapter in our kweb throughput test). Likewise, the latency for `upthread` and `upthread-lithe` drops by 6% to about 20 msec/request with a variance of essentially 0. The throughput of the Linux NPTL increases by only about 6% and its latency drops by 6% as well.
- As expected, the fourth and fifth intervals mirror the first two intervals in reverse order. However, the startup effects are missing; which is good since it means they really were startup effects and not some bigger issue.

These results demonstrate that all three threading models are able to process thumbnail requests at a similar rate, so long as those are the only type of traffic they are processing. Once static file traffic is introduced, the Linux NPTL has trouble keeping up with `upthread` and `upthread-lithe`. These configurations see minimal degredation in thumbnail performance, but are still able to serve static file traffic at a comparable rate with the Linux NPTL. Interestingly, `upthread` and `utprhead-lithe` are able to keep pace with each other every step of the way, demonstrating that user-level threading alone is the dominant component in achieving these performance improvements. The dynamic sharing of cores across parallel function calls between upthreads and OpenMP appears to have minimal impact for this particular application.

3.5 Discussion

In this chapter, we presented Lithe: a framework for composing multiple user-level schedulers in a single application. Lithe was originally proposed in 2010, and the work in this chapter serves to improve and extend its original capabilities. As part of this effort, we ported Lithe to run on parlib, and cleaned up many of it's outward facing interfaces. Porting Lithe in this way, helps to separate those features which pertain only to hart and context management from Lithe's primary innovation: it's extended scheduler API. It also served to

provide certain features lacking in the original Lithe implementation (e.g. proper handling of asynchronous syscalls).

As part of this work, we also introduced a common "fork-join" scheduler on top of which our OpenMP, TBB and a new "lithe-aware" version of upthreads are now based. We used these libraries to rerun the SPQR and Flickr-like image processing benchmarks from the original Lithe paper. Our results corroborate those found in this paper, as well demonstrate Lithe's new ability to support pthread applications.

Moving forward, we plan to extend Lithe's capabilities even further. Specifically, we plan to add APIs for preemptive scheduler development. With the existing API, there is no way for one runtime to forcibly remove a hart from another runtime, or be woken up and passed a hart based on a timer interrupt. Such functionality is essential to support popular parallel runtimes such as MPI. We have some ideas on how to add this functionality, and will be working to add it soon.

Chapter 4

Go

Go [5] is a modern programming language from Google that lends itself well to the development of parallel applications in the datacenter. It has been rising in popularity in recent years in part due to its elegant code composition model and in part due its concurrency model [31]. Go's concurrency model is based on launching light-weight user-level threads, called *go routines*, and synchronizing access to them through a language primitive known as a *channel*. This concurrency model matches well with the user-level threading model promoted by Akaros, Parlib, and Lithe.

In this chapter, we present our port of Go to Akaros. The port is currently frozen at Go 1.3, but we have plans to upgrade it soon. The goal of the port is two-fold. First, it demonstrates the completeness of the Akaros kernel in terms of supporting a fully functioning runtime environment that matches well to its underlying threading model. Second, it opens up a new set of of applications to Akaros that otherwise wouldn't be available. Go is a fairly self-contained ecosystem with a small OS dependant layer, and by porting Go to Akaros, we are able to leverage applications written against its standard library and be assured of their correct functionality. Moreover, it provides external developers a fast-path for testing new applications on Akaros without the need to learn a whole new system or deal with incompatibilities with glibc.

Our current port is based on emulating the notion of a "kernel" thread using user-level threads from our canonical parlib-based pthread library (upthreads). Porting Go this way was easier as a first-pass, because other OSs, including Linux, have a version of Go that mutiplexes go routines directly on top of pthreads. We simply mimic the Linux port and make sure the Go runtime hooks into our upthreads package properly. Ideally, we would bypass the upthread layer and instead multiplex go routines directly on top of vcores. However, there are several challenges with porting Go in this way, and we discuss these challenges further in Section 4.2.

The port itself took 1.5 years to complete, with two-thirds of that time spent hunting down bugs in Akaros and adding new features to Akaros that Go required. For example, the Go port drove the addition of the `SYS_DUP_FDS_TO` syscall, which allows an application to map file descriptor 'x' in process `A` to file descriptor 'y' in process `B` (so long as `B` is a child of

A). Go requires this functionality in the implementation of its `StartProcess()` abstraction. On Linux (and any OS that relies on the fork/exec model), Go has to jump through a large number of hoops in order to achieve this functionality. However, since Akaros follows a create/run model rather than a fork/exec model, we are able to call `SYS_DUP_FDS_TO` to set a child's fd map between the time the child is created and the time it is actually run.

Go also includes an extensive test suite that tests a wide-array of functionality from spawning subprocesses to launching http servers and handling file descriptors properly in the OS. By ensuring that Akaros can pass this test suite, we provide a fairly extensive regression of many of Akaros's core features. Akaros currently passes 100% of the 2,551 required tests from the Go 1.3 test suite (though it still skips 151 unrequired tests). Moving forward we plan to add support for these remaining tests and ultimately integrate our port with the official upstream Go repository.

In the sections that follow, we relate our experience porting Go to Akaros. We begin with an overview of Go itself, with a focus on the design and implementation of its runtime and other internal details pertinent to our port to Akaros. We then provide specifics of the port itself, followed by a discussion of the results we've obtained in evaluating our port. We conclude with a discussion of future directions, including our plans for eventual inclusion in the upstream Go repository and integration with Lithe.

4.1 Go Overview

Development of the Go programming language began in 2007 as a research project at Google. It has since become a well-established language used by a number of high-profile companies including Google, Docker, Uber, Zynga, Heroku, Facebook, and Dropbox [56]. Its primary goal is to help developers overcome many of the problems faced when working with large-scale software projects on modern computing systems. Specifically, its simplified C-like syntax, package-based component model, and short compile times, make managing millions of lines of code across thousands of developers more tractable than with other popular languages such as C++, Java or Python [57]. Moreover, many of its language primitives have been designed specifically with multi-core cpus, networked systems, and datacenters in mind. As such, developer effort is significantly reduced when writing applications that both run efficiently and scale well on large systems.

Generally, Go programs are fairly self-contained, with no external dependencies on libraries written in other languages. Indeed, the Go standard library provides much of the functionality provided by other standard libraries, such as glibc. However, Go also provides a special 'CGo' library to allow Go programs to call out to arbitrary C code compiled with gcc. We make heavy use of this CGo library in our port of Go to Akaros.

One of the most important language primitives provided by Go (and the one that bears its name) is the use of the `go` keyword to launch light-weight user-level threads, called go routines. Applications use go routines to launch hundreds (or thousands) of concurrent operations with significantly reduced overhead compared to traditional threads.

No interfaces exist to launch traditional threads in Go, and all concurrent operations are carried out via go routines. Internally, however, Go's runtime library still uses traditional threads as a vessel for running go routines. It maintains (roughly) one thread per core and multiplexes any runnable go routines on top of them. The details of how this operates is important, as it directly influences our port of Go to Akaros (which has no native support for traditional threads itself).

During normal operation, go routines cooperatively yield themselves on every function call, freeing up the thread they were running on to schedule another go routine in its place. So long as no blocking I/O calls are made, these threads happily multiplex go routines forever, following a simple cooperative scheduling algorithm defined by the Go runtime. However, as soon as a go routine makes a blocking I/O call, it is locked to the thread it is executing on, and a new thread is allocated to schedule go routines in its place. Once the I/O call completes, its associated go routine is reinserted into the runtime scheduler and the thread it was running on is destroyed. In this way, the number of actively running threads is always equal to the number of cores on the system, plus the number of go routines with outstanding I/O calls. In practice, a pool of threads is used to reduce the overhead of creating and deleting threads, but conceptually, the process works as described above.

For better or worse, this algorithm assumes the existence of traditional threads, and its internal data structures and scheduling algorithm are intimately intertwined with this assumption. All Go routines are backed by a light-weight structure known as a G (short for go routine), all threads are backed by a heavy-weight structure known as an M (short for machine thread), and all cores on the system are backed by an even heavier-weight structure known as a P (short for processor). As such, we often refer to go routines as Gs, threads as Ms, and cores as Ps. It is important to understand the role of these data structures, as they influence the way in which we ultimately design our port of Go to Akaros.

At program start, exactly n Ps are created to hold all per-core data associated with executing the Go runtime (e.g. per-core run-queues, a pointer to the current M running on a core, etc.). Likewise, n Ms are created and mapped 1-1 with each of the P's that have been created. Whenever an application launches a go routine, the Go runtime finds the P with the shortest run-queue and adds the go routine's associated G to that queue. The M associated with that P then pulls the G off of the P's run queue and executes it. If a G ever makes a blocking I/O call, it is locked to its current M and the associated P is handed off to a newly allocated M so it can continue executing Gs. In this way, Ps are passed around like tokens among Ms, and only Ms that have an associated P are able to run Gs. Since the number of Ps is limited to the number of cores on the system, only while Gs are blocked on I/O is the system temporarily oversubscribed with more running Ms than Ps.

Though not ideal, this model actually works quite well so long as only one Go application is running on the system at a time. However, it is explicitly designed to accommodate operating systems such as Linux, where no abstract notion of a core is available. As such, it suffers from the classic problem of oversubscribing cores when multiple applications are run simultaneously. By porting Go to Akaros, we able able to avoid this problem altogether.

4.2 Porting Go to Akaros

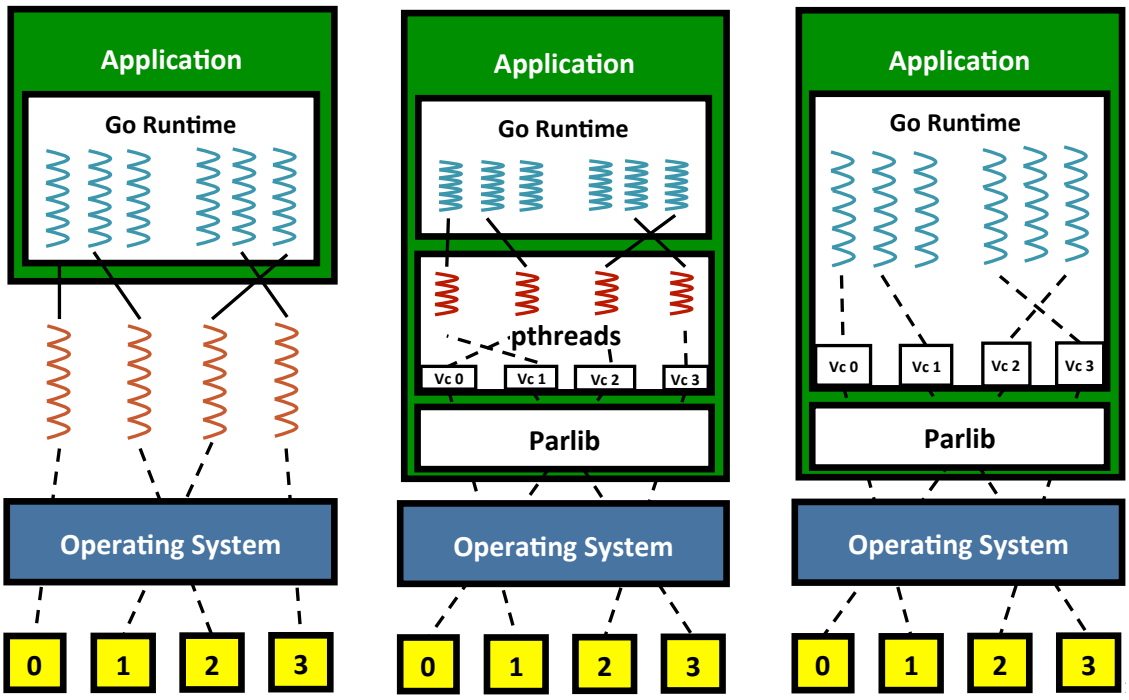
Normally when a Go program starts, it uses whatever facility is provided by the OS to launch M threads and begin multiplexing Go routines on top of them. For example, Go on Linux uses the `clone()` system call, Windows uses `CreateThread()`, and plan9 and BSD both use `fork()`. On Akaros, however, we have no convenient abstraction that maps well to Go's notion of an M thread. The closest thing we have is the vcore, but that actually maps better to a P than an M, and even then, it's not quite a 1-1 mapping.

Luckily, we are able to leverage Go's CGo library to emulate the notion of an M thread using our parlib-based upthread library. Regardless of the OS, whenever an application links in CGo, its default M thread implementation is replaced with pthreads to ensure that any C library calls are run inside a properly initialized pthread (as required by glibc). We simply leverage this capability and link in our upthreads library as the pthread library of choice. There are some caveats to this involving the proper handling of posix signals and issuing futex calls properly, but for the most part, Go is able to use our upthreads library without modification. With our upthread library in place, the rest of the port mostly involved brute engineering work, making Go issue our syscalls properly and hooking any Akaros specific functionality into existing Go code, where appropriate.

Ideally, we would bypass the upthread layer and instead multiplex go routines directly on top of vcores. However, this would require substantial changes to the Go runtime, forcing us to somehow consolidate the notion of the M and the P into a single abstraction that mapped nicely to our notion of a vcore. In essence, we would need to create an Akaros specific go routine scheduler that was independent of the default go routine scheduler used by other OSs. Although this is certainly possible (parlib itself is designed to make building user-level schedulers easy), discussions with Russ Cox (one of the lead developers of Go) led us to believe that this is highly undesirable. He is quoted as saying that Go is not in the business of supporting multiple schedulers, nor does it want to be. It is likely we will still try this out at some point in the future, but for now, we plan to keep improving our upthread-based implementation instead.

One downside of using upthreads instead of running directly on vcores is that it limits Go's ability to directly control which go routines run on which cores (and when). Scheduling of M threads is dictated by the upthread library rather than somewhere inside Go itself. However, this problem exists for Go regardless, normally leaving the scheduling of its M threads to some underlying kernel scheduler. We would only be improving on its existing implementation by adding more control. Figure 4.1 shows a comparison of Go on Linux, Go on Akaros with upthreads, and our ideal setup of Go on Akaros, with go routines mapped directly to vcores.

In general, the combination of Akaros and parlib, with its low-level notion of a vcore, still helps to alleviate the problem of oversubscribing cores to multiple applications running on a system at a time. Even with our initial port of Go to Akaros, which leverages upthreads instead of running go routines directly on vcores, the upthread library still requests and yield vcores on an application's behalf rather than oversubscribing cores. As Rhoden demonstrates



(a) Go on Linux with go routines on kernel threads

(b) Go on Akaros with go routines on upthreads

(c) Go on Akaros with go routines directly on vcores

Figure 4.1: The evolution of Go from running on Linux, to running on Akaros with upthreads, to running on Akaros with vcores directly.

in Chapter 7 of his dissertation [2], operating this way can provide substantial benefits over traditional kernel-based scheduling approaches.

4.3 Results

Unlike most of the other OSs supported by Go, Akaros does not yet support the ability to build native Go binaries itself. Instead, the entire Go setup is hosted on a non-Akaros machine, and all binaries are cross compiled. These binaries are then shipped over to an Akaros machine and executed.

As of Go 1.3, we are able to leverage the use of a `go_${GOOS}_${GOARCH}_exec` script to invoke Go commands on our host machine but execute any resulting Go binaries on a remote Akaros machine. So long as this script is in our path, calls to things like "`go run arguments...`" or "`go test arguments...`" do the job of compiling and building Go binaries for our desired target. However, instead of just executing the resulting binary, the path of the binary along with its arguments are passed to the `go_${GOOS}_${GOARCH}_exec` script, and that script is executed instead. We have simply implemented the Akaros version of this script to ship the resulting binary to a remote Akaros machine and invoke it via an

RPC call (using `listen1` [58] on the remote Akaros machine and `netcat` [59] on a local Linux machine).

While testing, our remote Akaros machine typically runs inside an instance of qemu on the same physical machine as the host. However, the remote machine is specified by IP address, so running Go on a remote physical machine is relatively straightforward as well. All results presented here were obtained both in quemu as well as on our standard 'c99' test machine described in Section 2.6.3.

With our `go_akaros_amd64_exec` script in place, we are able to run the following command to run the full test suite from Go's standard library on Akaros. This command can be found in the `run.bash` file of the Go source repository.

```
go test std -short -timeout=300s
```

The default timeout is 120s, but we had to increase it to 300s due to certain inefficiences in Akaros. Specifically, the `net/http` package will timeout early if we use the default timeout. We have not yet determined the exact cause of this particular inefficiency, but we are actively investigating it.

To run this command, we simply invoke `run.bash` directly, and it automatically calls our `go_akaros_amd64_exec` script for us. As part of this, it also runs a set of regression tests in addition to the standard test suite. The results of both the standard test suite and the regression test are presented below:

Standard Tests	
Total	2412
Passed	2261
Skipped	151
Failed	0

Regression tests	
Total	290
Passed	290
Skipped	0
Failed	0

These results demonstrate that we have provided a complete port of Go to Akaros. We pass all of the required tests from the standard test suite, skipping only a few of the non-required tests that some of the other OSs skip as well (mostly plan9 and nacl). Moreover, we pass all of the regression tests.

However, these results say nothing about the actual performance of Go on Akaros. Good performance results would obviously be desirable, but they were not the primary reason for porting Go to Akaros. Our primary goal was to demonstrate the completeness of the Akaros kernel in terms of supporting a fully functional runtime environment that maps well to its internal threading model. In the future, we plan to provide a more complete evaluation of Go on Akaros, including performance results.

As a more qualitative result of porting Go to Akaros, we provide the following antecdotal evidence of just how difficult it was to debug our port at times. Our port of Go to Akaros took 1.5 years to complete, and stories such as the following help to demonstrate why.

Example Debug Session

We wrote a multithreaded go networking benchmark similar to netperf in order to test the performance of our networking stack on Akaros. When running this application, we sometimes observed it non-deterministically crashing under heavy load. The crashes were always due to the application not being able to open a file. The file was normally one contained in /net/tcp/, but could also be /prof/mpstat, or any other file the application was opening. When printing the file name inside the kernel, we observed that it was often corrupt (filled with nulls, UTF-8, or otherwise unprintable characters). However, the same file name, when printed from the application (and inside the go runtime), was fine.

The initial diagnosis was kernel memory corruption. The akaros `open()` code snapshots the path via `user_strdup()` near the beginning of open. After lots of debugging, we observed that the snapshotted value (`t_path`) often did not match the contents of the userspace pointer (`path`). The first theory was heap corruption of the kmalloced buffer where `user_strdup()` stored `t_path`. However, we found cases where both the user pointer and the snapshotted path were corrupt.

The path to a solution was forged when we finally observed that the *contents* of the userspace pointer sometimes changed between entering and exiting the kernel. We initially suspected stack corruption, but Akaros's internal debugging tools helped us verify that the address was contained in a valid range of the application's memory map (i.e. not on the stack). Digging further, we realized that the syscall code in the go runtime was copying the path string into a temporary variable just before issuing the actual syscall. Looking at the code, we became suspicious that, since this temporary variable was not used anywhere else, and the syscall was using an "unsafe" access to it, that the garbage collection system may think that the temporary variable was unreferenced, and garbage collect it. We found backing for this theory in the Go docs [60], and eventually found a commit to the syscall code in a newer version of Go that fixed a similar issue in Solaris. Once we applied this commit, all of our problems went away.

So, in the end, a multithreaded go program was garbage collecting its syscall arguments out from under the kernel, and filling them with new contents. Most of the time, the kernel would win the race, and snapshot the path before its contents changed, but the kernel would occasionally lose the race, and the path would be changed before it was snapshotted, causing the observed corruption. In this particular case, the bug turned out to be in userspace, but working with an unstable kernel with incomplete debugging tools meant that the debug cycle was much longer than working with, say, Linux. The fix itself was a simple one line change, but the process of tracking it down took several days.

4.4 Discussion and Future Work

In this chapter we have presented our port of Go to Akaros. Our initial port is based on leveraging our parlib-based uthread library instead of multiplexing go routines directly on

top of vcores. Porting Go in this way was much easier as a first pass, as it let us focus on the other aspects of Go that we needed to bring in line with Akaros to get it working. It also let us keep most existing Go code *unchanged*, which will make it easier for us to eventually get our changes pushed upstream. Ultimately, we would like to provide a port of Go to Akaros that runs directly on vcores, but for now we are focusing exclusively on our upthreads-based port.

The goal of this port was not to attempt to outperform Go on any existing platforms, but rather to demonstrate the completeness of the Akaros kernel in terms of supporting a fully functioning runtime that maps well to its underlying threading model. Our results demonstrate that we have succeeded in providing such a port.

As mentioned in the previous chapter, Go could also benefit from a direct port to Lithe. Using Cgo, Go applications that make calls into OpenMP, TBB, or any other "lithe-aware" schedulers could benefit from such a port. A typical example would be calling into the MKL to perform an optimized linear algebra routine rather than implementing it from scratch in Go. Providing such a port is relatively straight-forward and work has already begun in supporting it. We hope to get these changes pushed upstream soon.

Chapter 5

Related Work

Our work on Akaros, Parlib, and Lithe, is grounded in a large body of prior work, stretching back as far as the early 1960s. From systems such as Psyche [61] and Scheduler Activations [27], which helped shape parlib's notion of vcore context and its overall method of handling asynchronous events, to seminal work from Needham and Lauer [62] on the duality of threads vs. events, our work has been motivated and influenced by a large number of previous systems. In this chapter, we provide a brief overview of these systems and show how they compare to our work. For a detailed overview of all the work that has helped to shape the ideas behind Akaros and its associated user-level libraries, we refer the reader to Chapter 2 of Barret Rhoden's dissertation [2].

5.1 Threading Models

Threading models are typically categorized as either 1:1, M:1, or M:N, showing the relation of user thread to kernel thread. In each of these models, some number of user threads are mapped directly onto, and managed by, some number of underlying kernel threads. As mentioned in Chapter 2, parlib and Akaros operate under a unique threading model, which looks similar to a hybrid of the M:1 and M:N models, but instead relies on direct access to cores instead of underlying kernel threads. We discuss each of these models below.

In an M:1 model, a single kernel thread is created to represent a process, and any user-level threads that get created are multiplexed on top of that single kernel thread. This is one of the earliest threading models, as all notions of threading are isolated completely to user-space. The single kernel thread represents the entire process and its address space, and it is unaware of any threading done on top of it. The M:1 model dates as far back as the early 1960s with work on coroutines [63], and more contemporary systems such as Green Threads and Java [64] continue to follow this model today.

The M:N model is similar to the M:1 model, except that N kernel threads are created to represent a process instead of just 1. In an ideal M:N model, the kernel would schedule its N kernel threads independant of the M user threads being multiplexed on top of them.

However, in order to accommodate blocking I/O calls, some method of coordinating threads across the user-kernel boundary must be in place. This model was popularized by Scheduler Activations [27] and Sun's Light Weight Processes (LWPs) [65] in the early 1990s. More recently, Apple's Grand Central Dispatch (GCD) [66] and Windows User-mode Scheduling (UMS) [67] have also emerged in the M:N space. Likewise, as described in Chapter 4, a form of M:N threading is used by the Go language runtime [5] to schedule go routines on top of its machine thread abstraction.

As it's name suggests, the 1:1 threading model maps each user thread to its own backing kernel thread. Unlike the M:1 or M:N models, the 1:1 model does not require a user-level scheduler to multiplex user threads on top of kernel threads. All scheduling is handled by the kernel. Likewise, no special coordination is needed to accommodate blocking system calls across the user-kernel boundary because each user thread simply blocks on its corresponding kernel thread. Because of its simplicity and relative efficiency, the 1:1 model is the most popular threading model in use today by systems such as Linux and BSD. Early systems to adopt the 1:1 model include Cedar [68, 69], Topaz [70], Mach [71], and System V [72].

Each of the threading models described above has certain advantages and disadvantages relative to one another. However, the one characteristic they all share is the notion that a user thread must somehow be tied to a backing kernel thread at the time it issues a blocking system call. Because of this restriction, complicated mechanisms are necessary to ensure that a process is able to continue scheduling threads while one of its threads is blocked on I/O. In an M:1 this is impossible: whatever thread happens to be running when the system call is made will block the entire process until that system call completes. This is highly undesirable on modern multi-core architectures and should mostly be avoided for efficient operation. In an M:N model, complex mechanisms are required to create a new backing kernel thread to represent the process while the thread issuing the system call is blocked in the kernel. Likewise, when a system call completes, another complicated mechanism is required to inform a process that it is done and allow it to continue running the thread that was blocked. Many believe these complexities (and the overheads they bring with them) outweigh any benefits gained by being able to control the thread scheduling policy in user-space [33]. In a 1:1 model, most of these problems are avoided, but less control is given to a process in terms of how and when any of its threads will ultimately be scheduled.

With Akaros and parlib, we do away with blocking system calls altogether. This allows us to maintain control of scheduling our threads in user-space, but avoid any complexities with blocking those threads in the kernel. Instead, our model relies on the ability for the kernel to hand out cores to an application rather than threads, so as to completely decouple the threading done at each level. In this way, we are similar to an M:1 model, in that the kernel schedules its threads unaware that any threading may be going on in user-space. But we also maintain the nice property of the M:N model, whereby multiple threads can be executing simultaneously and a process is always notified upon syscall completion. In a sense, we are an M:C model, where we multiplex M user threads on top of C cores handed directly to us by the kernel. This notion of decoupling user threads from kernel threads was popularized by Capriccio [32] in the early 2000s, and the model we adopt expands it for multi-core.

5.2 Concurrency vs. Parallelism

One of the biggest problems pervasive in both past and present systems, which we try to solve, is the lack of a good abstraction for expressing parallelism inside an application. Modern systems typically conflate the notions of concurrency and parallelism, providing a single abstraction to express the both: the abstraction of a thread. However, concurrency and parallelism are two distinct concepts and each requires its own abstraction. On Akaros, the kernel provides vcores for parallelism and parlib provides threads for concurrency.

Concurrency refers to the ability to break apart a computation into separate tasks, such that each task can run independently. Parallelism refers to the ability to run each of these tasks simultaneously, if possible. By giving an application the ability to express its parallelism via vcore requests, we give it explicit control (and awareness) over how many concurrent threads it is able to execute in parallel. Moreover, we give it the the freedom to choose whether to use threads or events as is primary method of expressing concurrency [62, 73, 49, 47, 74].

In the past, different techniques have been used to try and express this type of parallelism, with varying degress of success. For example, Mach was the first Unix-based system to enable intelligent thread scheduling across multiple cores in the system. In previous systems, such as Unix, all processes were single threaded, and any additional threading was completely a user-space construct. This restricted threads to the single core on which a process was currently scheduled. Mach expanded this process model to make the kernel aware of each of its threads and schedule them on cores independently. It is the father of the modern day 1:1 threading model.

In terms of actual abstractions, Psyche's *virtual processor* abstraction looks very similar to the "faked" vcore abstraction we provide for parlib on Linux, including the limitations that come with it. Virtual processors are represented by pinned kernel threads, and applications multiplex their user threads on top of them. However, every application has its own view of the virtual processor space, and the underlying kernel multiplexes corresponding virtual processors from each application onto the physical cores they are pinned to. In this way, true parallelism is not expressable by the application because the kernel intervenes to mutliplex virtual processors whenever it deems necessary.

Scheduler Activations introduced the notion of a *virtual multiprocessor*, which improves on the *virtual processor* concept, but has other problems which limit its usefulness. Using the virtual multiprocessor abstraction, applications are actually able to request dedicated access to a subset of cores on the machine. Each core is backed by an *activation*, on top of which an application can multiplex a set of user threads. However, multiple *activations* are used to represent each core over the lifetime of an application, making it difficult to maintain per-core data in any meaningful way (e.g. to implement per-core run queues or explicitly communicate between two cores via some kind of messaging mechanism). This occurs because the activation used to represent each core is essentially a kernel thread and can block in the kernel whenever a user thread running on it makes a blocking system call. When this happens, a new activation is created in its place, and passed back to the

application on the same core where the blocking system call was issued. Just as in parlib, control of the core is maintained by the application at all times, but the activation used to represent that core has changed, limiting its usefulness. Morover, the process of coordinating threads across the user/kernel boundary is more expensive than with parlib, or even with an optimized 1:1 threading model. This overhead (and the complexities it brings with it) tend to outweigh any benefits gained by being able to control the thread scheduling policy in user-space [33].

Both Psyche and Scheduler Activations follow an M:N threading model, where its somewhat natural to think about expressing parallelism by mapping M threads onto N abstractions that represent a core. However, even on Linux, with its 1:1 threading model, `tasksets` (or `cpusets` combined with `sched_setaffinity()`) can be used to express a limited form of parallelism. All thread scheduling decisions are still left up to the kernel, but tasksets allow an application to restrict which cores those threads will ultimately be scheduled on. Whenever those threads execute, they are isolated to those cores alone. In this way, an application is able to express the amount of parallelism it desires, but leave all thread scheduling decisions up to the kernel. As with Psyche and our vcore implementation on Linux, however, there is no way of providing dedicated access to the cores in a taskset, limiting the effectiveness of expressing parallelism in this way.

Relatedly, a technique known as "handoff scheduling" can be used in conjunction with tasksets to implement a form of cooperative scheduling under a 1:1 model. This technique was first introduced by Mach in the late 1980s, but Google has recently introduced a patch to the Linux kernel which brings a form of Handoff Scheduling to Linux as well [42]. At the root of this patch is a fast-path system call, called `switch_to`, which allows an application to synchronously transfer control of one thread to another, leaving the original thread unscheduled. Using this call, user-space is able to precisely control which of its threads are scheduled at any given time. Moreover, the `switch_to` call bypasses all of the scheduling logic of the kernel, making it $16x$ faster than a traditional `futex` call (which one could imagine using to implement similar functionality, albeit in an ad-hoc manner). The biggest advantage of such an approach is that it is a complete solution for cooperative user-level scheduling which can easily be retrofit into the existing Linux kernel. Likewise, it is fully compatible with existing code and tools that assume the 1:1 threading model of Linux. The biggest disadvantage is that it perpetuates the conflated concepts of threads vs. cores and does nothing to improve core isolation. Nor does it solve the classic problem of oversubscribing threads to cores when running multiple applications on the system at a time. Additionally, it does not allow an application to dictate the order in which each of its schedulable threads will run; that is still left completely up to the kernel. The combination of Akaros and parlib solves each of these problems and more.

5.3 Contemporary Systems

Our work on Akaros has been highly influenced by many of the principles first set forth by the Exokernel [75]. Both systems expose information about the underlying hardware resources to allow application writers to make better decisions. However, the Exokernel takes a more extreme view and advocates for full application control in both policy and mechanism. We do not want to "exterminate all abstractions" [76], rather we want new abstractions that do not hide the performance aspects of the system from the application. Applications should not need to write their own libOS to utilize the system. Instead, they should use the APIs we provide to control their resources, with sensible defaults for resources that are not critical for performance. Likewise, we advocate the separation of resource management from protection domains as advocated by Banga et al. [77]. Their resource containers provide proper accounting of resources, independent of the execution context to which they are currently tied. Akaros's threading model, which separates the notion of threads from cores, provides the perfect vessel in which to do such accounting.

Many of Akaros's other features are related to previous work as well. We agree with Drepper [78] that an asynchronous system call interface is necessary for Zero-Copy I/O, and with Soares et al. [79] that it will help with cache behavior and IPC. We further believe that system calls ought to be asynchronous because it allows applications to maximize outstanding I/Os and not lose their cores simply because a thread makes a system call that might block. Likewise, our application-level event handlers are similar to work done with Active Messages [38], and we may end up building a system similar to Optimistic Active Messages [80] if we want handlers that can block. In fact, within the Akaros kernel, our methods of spawning kernel threads on demand is similar in principle to OAM: plan on not blocking, then deal with it if it happens.

Our model for resource provisioning discussed in Section 1.2.2 is also grounded in a large body of prior work, mostly generated by research on multi-core operating systems and the real-time community [81, 82, 83, 84]. In particular, our model provides an abstraction similar to that of a Software Performance Unit (SPU) [85], with a few distinctions. SPUs rely on processes to cooperatively share unused resources with background tasks, instead of allowing the system to manage these resources itself. Moreover, we do not grant a process access to the resources contained in its resource partitions until the process explicitly requests them. Instead, resource partitions serve as a provision of resources similar to the notion of a "reserve" [86]. Reserves, however, were designed for predictable applications that have a predetermined schedule, whereas our provisioning interfaces were not. Our abstractions go beyond the initial steps put forth by these previous systems, and integrate these ideas into a single cohesive system.

Taken as a whole, custom OSs similar to Akaros have thrived in performance computing (HPC) for many years [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. For example, Kitten [87] is a lightweight kernel that typically runs on the compute nodes of supercomputers. Kitten focuses on reducing kernel interference and maximizing performance for batch style HPC workloads. Likewise, the IBM Blue Gene series of supercomputers [88, 89, 90] make use of

a very minimal operating system, called CNK, which gives applications full access to the memory and cpu resources available on its compute nodes. Many of Akaros's abstractions (especially the many-core process) would work well for HPC applications and could be built into these HPC OSs. Likewise, Akaros could run alongside these OSs in an HPC cluster, as long as the appropriate hardware and network protocols were supported.

A number of other operating systems designed for many-core architectures have also emerged over the last decade. Corey [91] is an exokernel that improves on kernel scalability by reducing sharing to the bare minimum desired by an application. Helios [92] focuses on seemlessly running on heterogeneous hardware. We do not explicitly focus on NUMA or heterogenity, but there is nothing that prevents our techniques from working in that environment.

Like Akaros, the fos [93] system is an operating system designed for the datacenter. Unlike Akaros, however, fos presents a single-image abstraction for an entire datacenter of machines, and would require the entire datacenter to run fos to be effective. Akaros strives to provide compatibility at a network protocol level, allowing incremental deployment in datacenters. Notably, we envision data-intensive and processing-intensive nodes to run Akaros, while the nodes on the control plane (e.g. GFS master, Hadoop scheduler) to run commodity OSs.

Likewise, Barrelfish and its multikernel [94] are an operating system designed for the datacenter. Barrelfish assumes the existence of heterogeneous hardware and structures the OS as a distributed system, with message passing and replicated operating system state. We do not treat the OS as a distributed system; instead we have a small set of cores (the low-latency cores) that make decisions for other cores. Moreover, a large amount of state is global in Akaros, much like in Linux. Although global state sounds like a potential problem, researchers have analyzed Linuxs VFS and determined a few bottlenecks that can be removed [95]. More importantly, Linux has its own set of VFS scalability patches that they have been working on for a while [96, 97]. Kernel scalability is a large concern of ours, but unlike these other systems, we focus more on abstractions for enabling high-performance parallel applications in the datacenter. Furthermore, Akaros should scale much better than systems with more traditional process models because there are no kernel tasks or threads underlying each thread of a process and because we partition resources.

Other systems that relate to our work include Mesos [21] and Google Borg [22]. As discussed in Section 1.1, these are cluster management systems (or cluster OSs) that decide when and where to allocate resources to applications across an entire datacenter. Our focus on resource isolation and two-level resource management should provide better support for these systems and others like them. Specifically, we can use Akaros's provisioning interfaces to make *node-local* decisions that enforce the resource allocations set up by these systems. This should both simplify the global resource allocation algorithm and reduce network communication.

Chapter 6

Conclusion

The overarching theme for all work presented in this thesis is the promotion of user-level scheduling as a viable means for exploiting parallelism in modern day systems. The move towards many-core CPUs, multi-lane high-speed networking cards, hundreds of gigabytes of memory, and terabytes of disk per-machine provides a unique opportunity to redesign the software stack of modern systems to run more efficiently. Our work on Akaros attempts to tackle this problem head-on, and this thesis focuses on one specific aspect of this task: building user-level libraries to take advantage of Akaros's ability to explicitly allocate cores to an application.

We began this thesis with a discussion of modern systems architectures, with a focus on the evolution of datacenters and why we believe a new operating system is warranted in this computing environment. We then provided a brief overview of Akaros, with a focus on its primary "many-core" process abstraction (MCP) and its resource provisioning interfaces. These abstractions provide the foundation for explicit core allocation on top of which the rest of the work in this thesis is based. We leverage these core management capabilities and present the design and implementation of the user-level libraries that sit on top of them.

At the heart of these user-level libraries is parlib – a framework for building efficient parallel runtimes based on user-level scheduling. Parlib leverages Akaros's ability to explicitly allocate cores to an application and provides a set of higher-level abstractions that make working with these cores easier. These abstractions make it possible to construct highly efficient parallel runtimes on modern day multi-core architectures.

However, parlib alone is insufficient to share cores efficiently between competing software components inside an application. Such functionality is essential in today's open source community, where highly-tuned libraries built with competing parallel runtimes are readily available. Applications writers should be able to leverage these libraries instead of writing their own or manually tuning them to operate together efficiently. To provide this functionality, we provide a reimplementation of Lithe on top of parlib. Although not originally proposed as such, Lithe can be thought of as a natural extension to parlib, with the added capability of coordinating access to cores among multiple user-level schedulers in a single application. Our work on Lithe extends and improves upon the original version of Lithe first

proposed by Pan et al. in 2010.

Finally, our port of Go to Akaros demonstrates the completeness of the Akaros kernel in terms of supporting a fully functioning runtime environment that matches well to its underlying threading model. By ensuring that Akaros can pass Go's extensive test suite, we provide a fairly complete regression of many of Akaros's core features.

Overall, our work is grounded in the assertion that traditional OS abstractions are ill-suited for high performance and parallel applications, especially on large-scale SMP and many-core architectures. By customizing Akaros to provide a few, well-defined mechanisms for fine-grained management of cores, this work demonstrates that it is possible to construct a user-level software stack that ensures scalability with increased performance and predictability. We are still a long ways from providing a fully-functioning system that is capable of handling production workloads on productions servers, but the work presented here helps bring us one step closer to this goal.

Bibliography

[1] Barret Rhoden et al. "Improving per-node efficiency in the datacenter with new OS abstractions". In: *SOCC '11: Proceedings of the 2nd ACM Symposium on Cloud Computing*. New York, NY, USA: ACM Request Permissions, Oct. 2011, 25:1–25:8.

[2] Barret Rhoden. "Operating System Support for Parallel Processes". Ph.D. dissertation. University of California, Berkeley, 2014.

[3] Heidi Pan. "Cooperative Hierarchical Resource Management for Efficient Composition of Parallel Software." In: *Massachusetts Institute of Technology* (2010).

[4] Heidi Pan, Benjamin Hindman, and Krste Asanovic. "Composing parallel software efficiently with lithe." In: *PLDI* (2010), pp. 376–387.

[5] *The Go Programming Language*. URL: https://golang.org/.

[6] David E Culler et al. "Parallel computing on the Berkeley NOW". In: *9th Joint Symposium on Parallel Processing*. 1997.

[7] Herb Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobbs journal* 30.3 (2005), pp. 202–210.

[8] Andrew A Chien and Vijay Karamcheti. "Moore's Law: The First Ending and A New Beginning". In: *Computer* 12 (2013), pp. 48–53.

[9] Ron Brightwell et al. "A Performance Comparison of Linux and a Lightweight Kernel". In: *IEEE International Conference on Cluster Computing*. 2003, pp. 251–258. DOI: http://dx.doi.org/10.1109/CLUSTR.2003.1253322.

[10] Hakan Akkan, Michael Lang, and Lorie Liebrock. "Understanding and isolating the noise in the Linux kernel". In: *Int. J. High Perform. Comput. Appl.* 27.2 (May 2013), pp. 136–146. ISSN: 1094-3420. DOI: 10.1177/1094342013477892. URL: http://dx.doi.org/10.1177/1094342013477892.

[11] Kazutomo Yoshii et al. "Extending and benchmarking the "Big Memory" implementation on Blue Gene/P Linux". In: *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '11. Tucson, Arizona: ACM, 2011, pp. 65–72. ISBN: 978-1-4503-0761-1. DOI: 10.1145/1988796.1988806. URL: http://doi.acm.org/10.1145/1988796.1988806.

[12] P. De, R. Kothari, and V. Mann. "Identifying sources of Operating System Jitter through fine-grained kernel instrumentation". In: *Cluster Computing, 2007 IEEE International Conference on*. 2007, pp. 331–340. DOI: 10.1109/CLUSTR.2007.4629247.

[13] Pradipta De, Vijay Mann, and Umang Mittaly. "Handling OS jitter on multicore multithreaded systems". In: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009. DOI: 10.1109/IPDPS.2009.5161046.

[14] R. Gioiosa, S.A. McKee, and M. Valero. "Designing OS for HPC Applications: Scheduling". In: *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*. 2010, pp. 78–87. DOI: 10.1109/CLUSTER.2010.16.

[15] M. Giampapa et al. "Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. 2010. DOI: 10.1109/SC.2010.22.

[16] Terry Jones. "Linux kernel co-scheduling and bulk synchronous parallelism". In: *Int. J. High Perform. Comput. Appl.* 26.2 (May 2012), pp. 136–145. ISSN: 1094-3420. DOI: 10.1177/1094342011433523. URL: http://dx.doi.org/10.1177/1094342011433523.

[17] William TC Kramer. "PERCU: A Holistic Method for Evaluating High Performance Computing Systems". PhD thesis. University of California, Berkeley, Nov. 2008.

[18] Stefan Wächtler and Michal Sojka. "Operating System Noise: Linux vs. Microkernel". In: *Proceedings of the 14th Real-Time Linux Workshop*. Open Source Automation Development Lab (OSADL), Oct. 2012.

[19] Adam Hammouda, Andrew Siegel, and Stephen Siegel. "Overcoming Asynchrony: An Analysis of the Effects of Asynchronous Noise on Nearest Neighbor Synchronizations". English. In: *Solving Software Challenges for Exascale*. Ed. by Stefano Markidis and Erwin Laure. Vol. 8759. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 100–109. ISBN: 978-3-319-15975-1. DOI: 10.1007/978-3-319-15976-8_7. URL: http://dx.doi.org/10.1007/978-3-319-15976-8_7.

[20] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. "MapReduce: simplified data processing on large clusters". In: *In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.

[21] Benjamin Hindman et al. *Mesos: A platform for fine-grained resource sharing in the data center," UCBerkeley*. Tech. rep. 2010.

[22] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. ACM. 2015, p. 18.

[23] Matthew Sottile and Ronald Minnich. "Analysis of microbenchmarks for performance tuning of clusters". In: *Cluster Computing, 2004 IEEE International Conference on*. IEEE. 2004, pp. 371–377.

[24] Alexander Rasmussen et al. "TritonSort: A Balanced Large-Scale Sorting System". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*. ACM, 2011.

[25] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. May 2011.

[26] Dror G Feitelson and Larry Rudolph. "Gang Scheduling Performance Benefits for Fine-Grain Synchronization". In: *Journal of Parallel and Distributed Computing* 16 (1992), pp. 306–318.

[27] T E Anderson and others. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". In: *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*. 1991.

[28] Steven M Hand. "Self-paging in the Nemesis operating system". In: *Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 73–86.

[29] I Leslie et al. "The design and implementation of an operating system to support distributed multimedia applications". In: *IEEE Journal on Selected Areas in Communications* 14 (1996), pp. 1280–1297.

[30] T A Davis. "Multifrontal multithreaded rank-revealing sparse QR factorization". In: *ACM Trans Math Softw under submission* (2008).

[31] Matt Asay. *Google's Go Programming Language: Taking Cloud Development By Storm*. 2015. URL: `http://readwrite.com/2014/03/21/google-go-golang-programming-language-cloud-development`.

[32] Rob Von Behren et al. "Capriccio: Scalable threads for internet services". In: *SOSP '03*. 2003.

[33] Ingo Molnar. *1:1 Threading vs. Scheduler Activations*. URL: `https://lkml.org/lkml/2002/9/24/33`.

[34] Nathan J Williams. "An Implementation of Scheduler Activations on the NetBSD Operating System." In: *USENIX Annual Technical Conference, FREENIX Track*. 2002, pp. 99–108.

[35] Mindaugas Rasiukevicius. "Thread scheduling and related interfaces in NetBSD 5.0". In: 2009.

[36] Vincent Danjean, Raymond Namyst, and Robert D Russell. "Integrating kernel activations in a multithreaded runtime system on top of Linux". In: *Parallel and Distributed Processing*. Springer, 2000, pp. 1160–1167.

[37] Butler W Lampson and David D Redell. "Experience with processes and monitors in Mesa". In: *Communications of the ACM* 23.2 (1980), pp. 105–117.

[38] Thorsten von Eicken et al. *Active Messages: a Mechanism for Integrated Communication and*. Tech. rep. Berkeley, CA, USA, 1992.

[39] Donald Lewine. *POSIX programmers guide*. " O'Reilly Media, Inc.", 1991.

[40] Jeff Bonwick et al. "The Slab Allocator: An Object-Caching Kernel Memory Allocator." In: *USENIX summer*. Vol. 16. Boston, MA, USA. 1994.

[41] Joseph Albert. "Algebraic Properties of Bag Data Types." In: *VLDB*. Vol. 91. Citeseer. 1991, pp. 211–219.

[42] Paul Turner. *User-level threads..... with threads*. Presented at the Linux Plumbers Conference, New Orleans, Louisiana. 2013.

[43] Will Reese. "Nginx: the high-performance web server and reverse proxy". In: *Linux Journal* 2008.173 (2008), p. 2.

[44] David H Bailey et al. "The NAS parallel benchmarks". In: *International Journal of High Performance Computing Applications* 5.3 (1991), pp. 63–73.

[45] Andi Kleen. *x86: Add support for rd/wr fs/gs base*. Published as a set of kernel patches to the Linux Kernel Mailing List. April 2014. URL: https://lkml.org/lkml/2014/4/28/582.

[46] Part Guide. "Intel® 64 and IA-32 Architectures Software Developers Manual". In: (2014).

[47] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. "Why Events Are a Bad Idea (for High-Concurrency Servers)." In: *HotOS*. 2003, pp. 19–24.

[48] Matt Welsh, David Culler, and Eric Brewer. "SEDA: an architecture for well-conditioned, scalable internet services". In: *ACM SIGOPS Operating Systems Review* 35.5 (2001), pp. 230–243.

[49] John Ousterhout. "Why threads are a bad idea (for most purposes)". In: *Presentation given at the 1996 Usenix Annual Technical Conference*. Vol. 5. San Diego, CA, USA. 1996.

[50] Stefanie Forrester (Dakini). *Tuning Nginx for Best Performance*. Personal Blog. April 2012. URL: http://dak1n1.com/blog/12-nginx-performance-tuning.

[51] Martin Fjordvald. *Optimizing NGINX for High Traffic Loads*. Personal Blog. April 2011. URL: http://blog.martinfjordvald.com/2011/04/optimizing-nginx-for-high-traffic-loads/.

[52] MKL Intel. *Intel math kernel library*. 2007.

[53] SuiteSparse. *SuiteSparse*. URL: http://faculty.cse.tamu.edu/davis/suitesparse.html.

[54] Flickr. *Photo Source URLs*. URL: https://www.flickr.com/services/api/misc.urls.html.

[55] GraphicsMagick. *GraphicsMagick*. URL: http://www.graphicsmagick.org/.

[56] The Go Authors. *GoUsers*. URL: https://github.com/golang/go/wiki/GoUsers.

[57] Rob Pike. "Go at Google". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: ACM, 2012, pp. 5–6. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384720. URL: http://doi.acm.org/10.1145/2384716.2384720.

[58] The Plan9 Authors. *The plan9 man pages: listen1*. URL: http://man.cat-v.org/p9p/8/listen1.

[59] The Linux Authors. *The Linux man pages: nc*. URL: http://linux.die.net/man/1/nc.

[60] The Go Authors. *Go 1.3 Release Notes*. URL: https://golang.org/doc/go1.3.

[61] M L Scott and T J LeBlanc. *Psyche: A General-Purpose Operating System for Shared-Memory Multiprocessors*. Tech. rep. Rochester, NY, USA, 1987.

[62] Hugh C Lauer and Roger M Needham. "On the duality of operating system structures". In: *ACM SIGOPS Operating Systems Review* 13.2 (1979), pp. 3–19.

[63] Melvin E Conway. "Design of a separable transition-diagram compiler". In: *Communications of the ACM* 6.7 (1963), pp. 396–408.

[64] Sun Microsystems Inc. *JDK 1.1 for Solaris Developerss Guide*. 2000. URL: http://docs.oracle.com/cd/E19455-01/806-3461/book-info/index.html.

[65] Ben Catanzaro. *Multiprocessor System Architectures*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN: 0-13-089137-1.

[66] Kazuki Sakamoto and Tomohiko Furumoto. "Grand central dispatch". In: *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 2012, pp. 139–145.

[67] *Windows User-Mode Scheduling*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd627187(v=vs.85).aspx.

[68] Butler Lampson. *A description of the Cedar language*. Tech. rep. Technical Report CSL-83-15, Xerox Palo Alto Research Center, 1983.

[69] Daniel C Swinehart et al. "A structural view of the Cedar programming environment". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.4 (1986), pp. 419–490.

[70] Paul R McJones and Garret F Swart. *Evolving the UNIX system interface to support multithreaded programs*. Vol. 28. Digital Equipment Corporation Systems Research Center, 1987.

[71] Mike Accetta et al. "Mach: A new kernel foundation for UNIX development". In: (1986).

[72] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4: an Open Systems Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN: 0-13-098138-9.

[73] Maxwell N Krohn, Eddie Kohler, and M Frans Kaashoek. "Events Can Make Sense." In: *USENIX Annual Technical Conference*. 2007, pp. 87–100.

[74] Frank Dabek et al. "Event-driven programming for robust software". In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM. 2002, pp. 186–189.

[75] D R Engler, M F Kaashoek, and J O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*. 1995.

[76] Dawson R Engler and M Frans Kaashoek. "Exterminate All Operating System Abstractions". In: *In the 5 th IEEE Workshop on Hot Topics in Operating Systems, Orcas Island*. IEEE Computer Society, 1995, pp. 78–83.

[77] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. "Resource containers: A new facility for resource management in server systems". In: *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*. 1999.

[78] Ulrich Drepper. "The need for asynchronous, zero-copy network i/o". In: *Ottawa Linux Symposium*. Citeseer. 2006, pp. 247–260.

[79] Livio Soares and Michael Stumm. "FlexSC: flexible system call scheduling with exception-less system calls". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.

[80] *Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation*. 1995.

[81] Amit Gupta and Domenico Ferrari. "Resource partitioning for real-time communication". In: *IEEE/ACM Trans. Netw.* 3.5 (1995), pp. 501–508.

[82] Aloysius K Mok, Xiang Feng, and Deji Chen. "Resource partition for real-time systems". In: *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*. IEEE. 2001, pp. 75–84.

[83] Kyle J Nesbit et al. "Multicore resource management". In: *IEEE micro* 3 (2008), pp. 6–16.

[84] Derek Wright. "Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor". In: *Conference on Linux clusters: the HPC revolution*. 2001.

[85] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. "Performance isolation: sharing and isolation in shared-memory multiprocessors". In: *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1998, pp. 181–192.

[86] Clifford Mercer, Stefan Savage, and Hideyuki Tokuda. "Processor Capacity Reserves for Multimedia Operating Systems". In: *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. 1994.

[87] J Lange et al. "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing". In: *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2010.

[88] Ruud A Haring et al. "The ibm blue gene/q compute chip". In: *Micro, IEEE* 32.2 (2012), pp. 48–60.

[89] Alan Gara et al. "Overview of the Blue Gene/L system architecture". In: *IBM Journal of Research and Development* 49.2.3 (2005), pp. 195–212.

[90] Gheorghe Almasi et al. "Overview of the IBM Blue Gene/P project". In: *IBM Journal of Research and Development* 52.1-2 (2008), pp. 199–220.

[91] Silas Boyd-Wickizer and others. "Corey: an operating system for many cores". In: *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*. 2008.

[92] Edmund B Nightingale et al. "Helios: heterogeneous multiprocessing with satellite kernels". In: *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009, pp. 221–234.

[93] David Wentzlaff and Anant Agarwal. "Factored operating systems (fos): the case for a scalable operating system for multicores". In: *SIGOPS Oper. Syst. Rev.* 43.2 (2009), pp. 76–85.

[94] Adrian Sch upbach and others. "Embracing diversity in the Barrelfish manycore operating system". In: *Proc. of the Workshop on Managed Many-Core Systems (MMCS)*. 2008.

[95] Silas Boyd-Wickizer et al. "An Analysis of Linux Scalability to Many Cores." In: *OSDI*. Vol. 10. 13. 2010, pp. 86–93.

[96] Jonathan Corbet. *JLS: Increasing VFS Scalibility*. URL: `http://lwn.net/Articles/360199/`.

[97] *VFS Scalability Patches in 2.6.36*. URL: `http://lwn.net/Articles/401738/`.