

OS and Runtime Support for Efficiently Managing Cores in Parallel Applications

Kevin Klues

Dissertation Talk

University of California, Berkeley

Computer Science Division

May 11, 2015

Overview

- We assert that:
 - Parallel applications can benefit from the ability to explicitly control their thread scheduling policies in user-space
 - More importantly, they should have direct access to cores and the ability to manage those cores as they see fit

Overview

- We assert that:
 - Parallel applications can benefit from the ability to explicitly control their thread scheduling policies in user-space
 - More importantly, they should have direct access to cores and the ability to manage those cores as they see fit
- Current systems lack support for these capabilities → We attempt to remedy that



Overview

- OS Support (Akaros)
 - For direct access to cores
- Runtime Support (Parlib)
 - For developing user-level schedulers on top of those cores



Overview

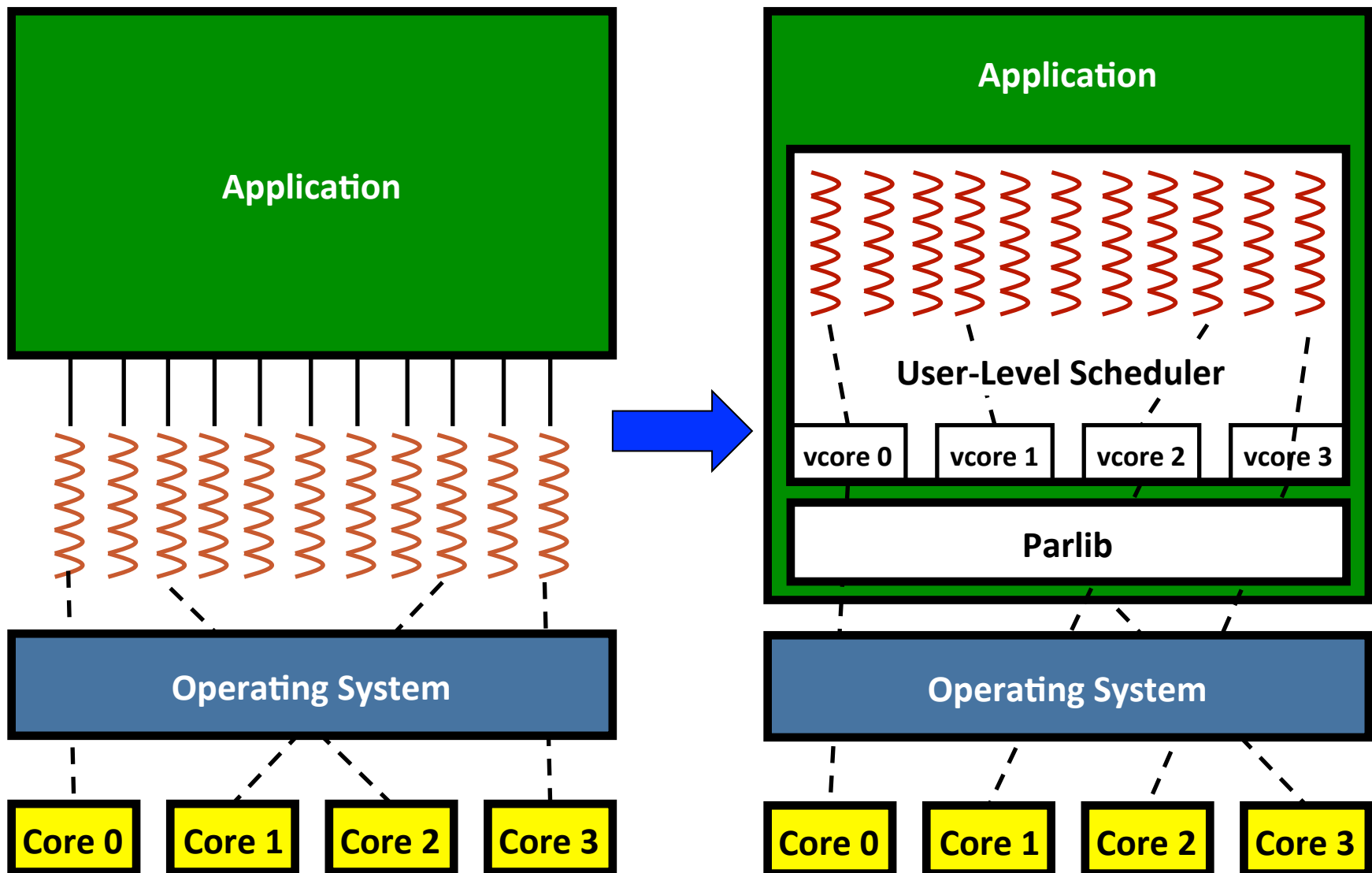
- OS Support (Akaros)
 - New, experimental OS we developed
 - The “Many-Core Process” (MCP)
 - APIs for **dedicated** access to cores (vcores)
 - Decoupled user/kernel threading model



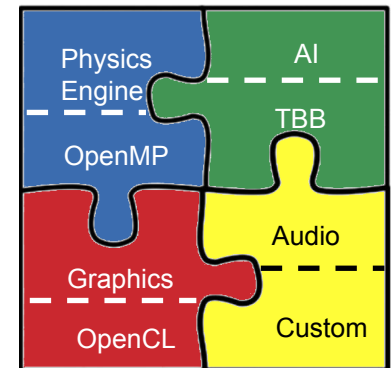
Overview

- OS Support (Akaros)
 - New, experimental OS we developed
 - The “Many-Core Process” (MCP)
 - APIs for **dedicated** access to cores (vcores)
 - Decoupled user/kernel threading model
- Runtime Support (Parlib)
 - User-level framework for parallel runtime development
 - Focus on **application-directed** core management and user-level threading abstractions (vcores / uthreads)
 - Canonical parlib-based **pthread**s implementation
 - Ports for *both* Akaros and Linux (with limited support)

Overview

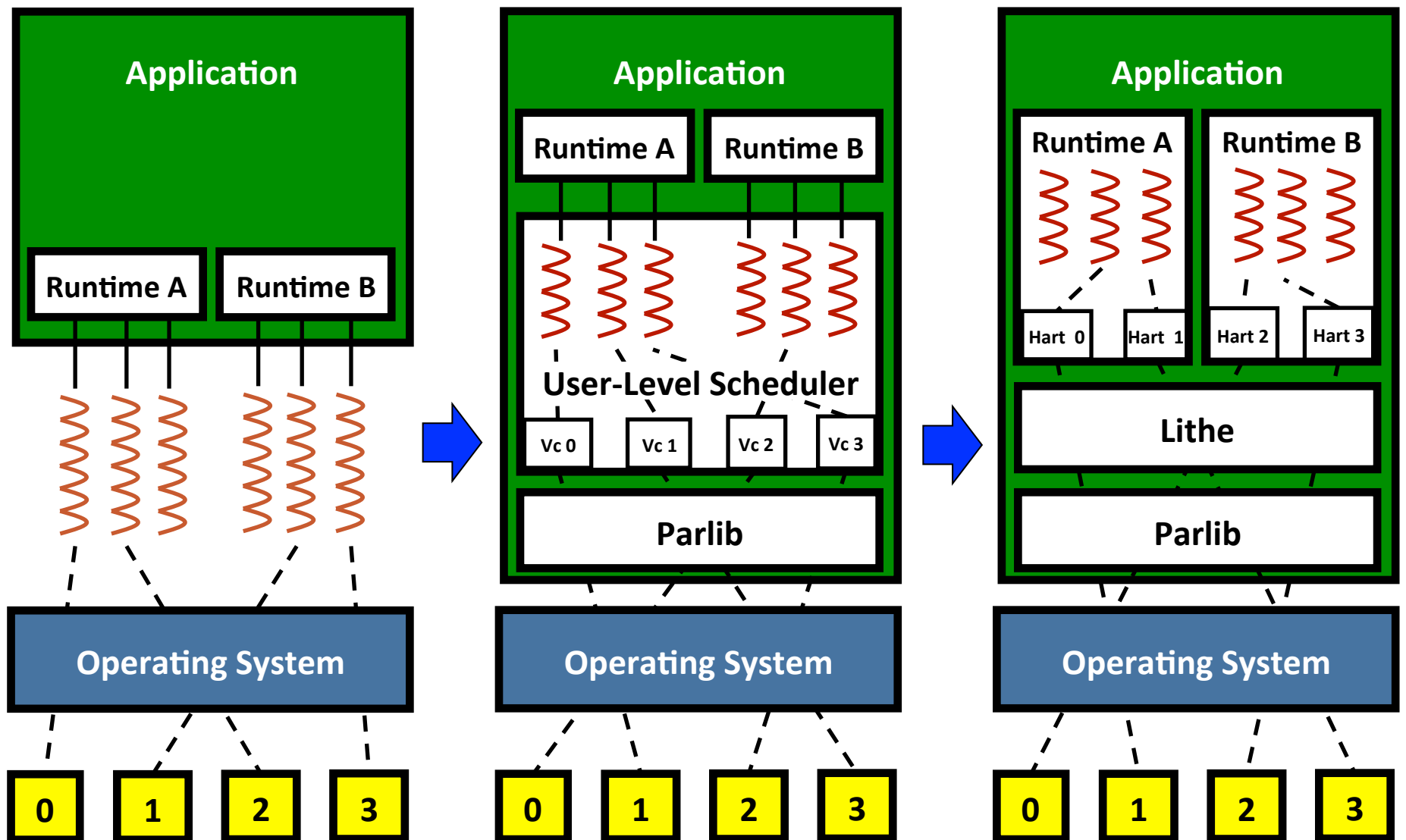


Overview



- Lithe (Pan et al. 2010)
 - Framework for composing *multiple* parallel runtimes in a single application
 - Structured sharing of cores between schedulers
 - Notion of a “parallel” function call
- My contributions
 - Complete rewrite of Lithe to sit on top of Parlib
 - Cleaner APIs, more complete implementation
 - Generalization of a common “fork-join” scheduler
 - Stable ports of TBB, OpenMP and now pthreads
 - Runs on both Linux and Akaros

Overview



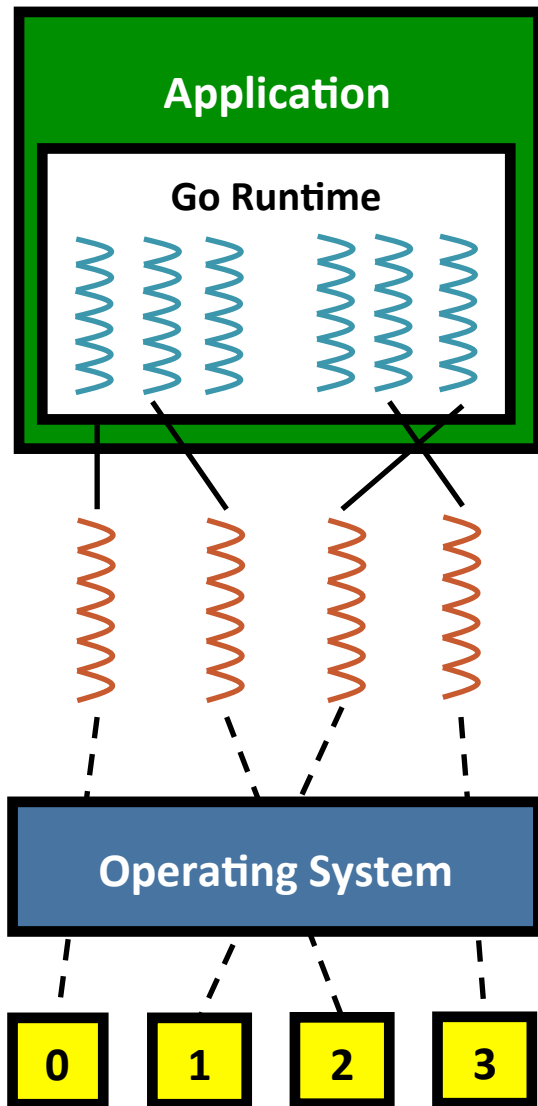
Overview

- Port of Go to Akaros
 - Go is an open-source language from Google designed for building highly concurrent systems
 - Has its own form of user-level scheduling (goroutines)
 - Initial port built on parlib-based pthread implementation
 - Took 1.5 years to complete
 - Drove massive amounts of innovation in Akaros

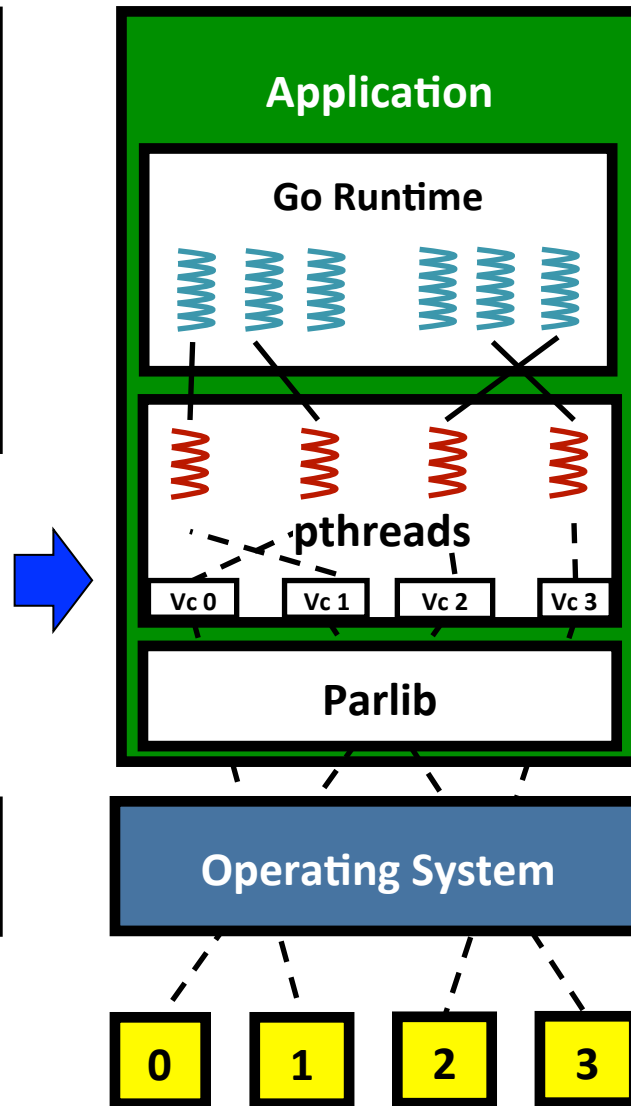


Overview

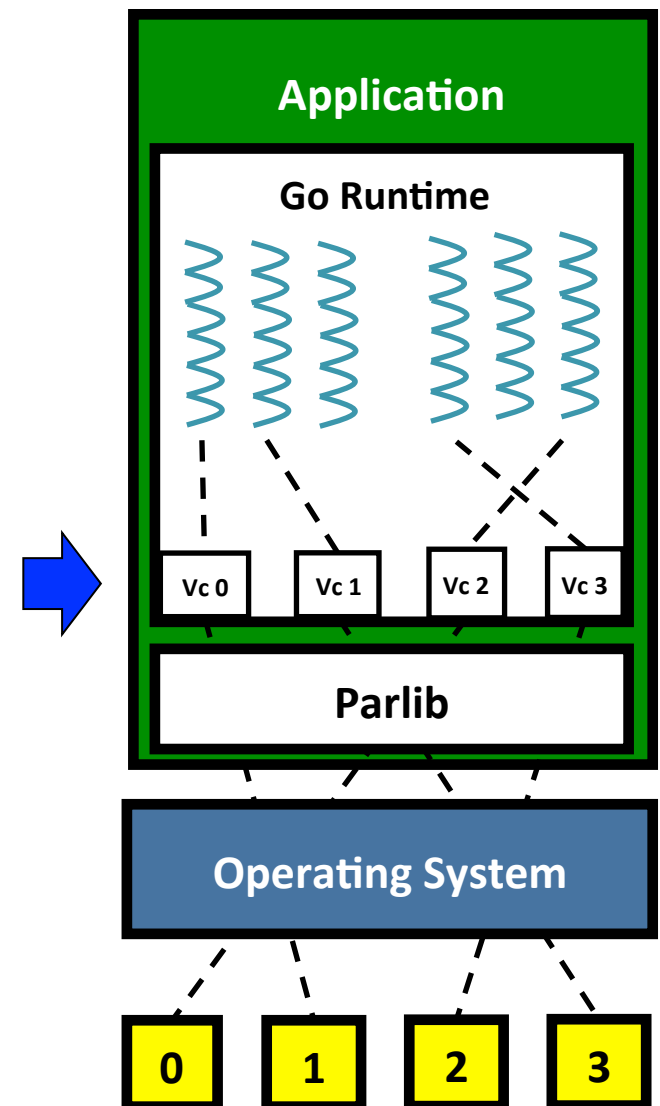
Linux



Current Akaros



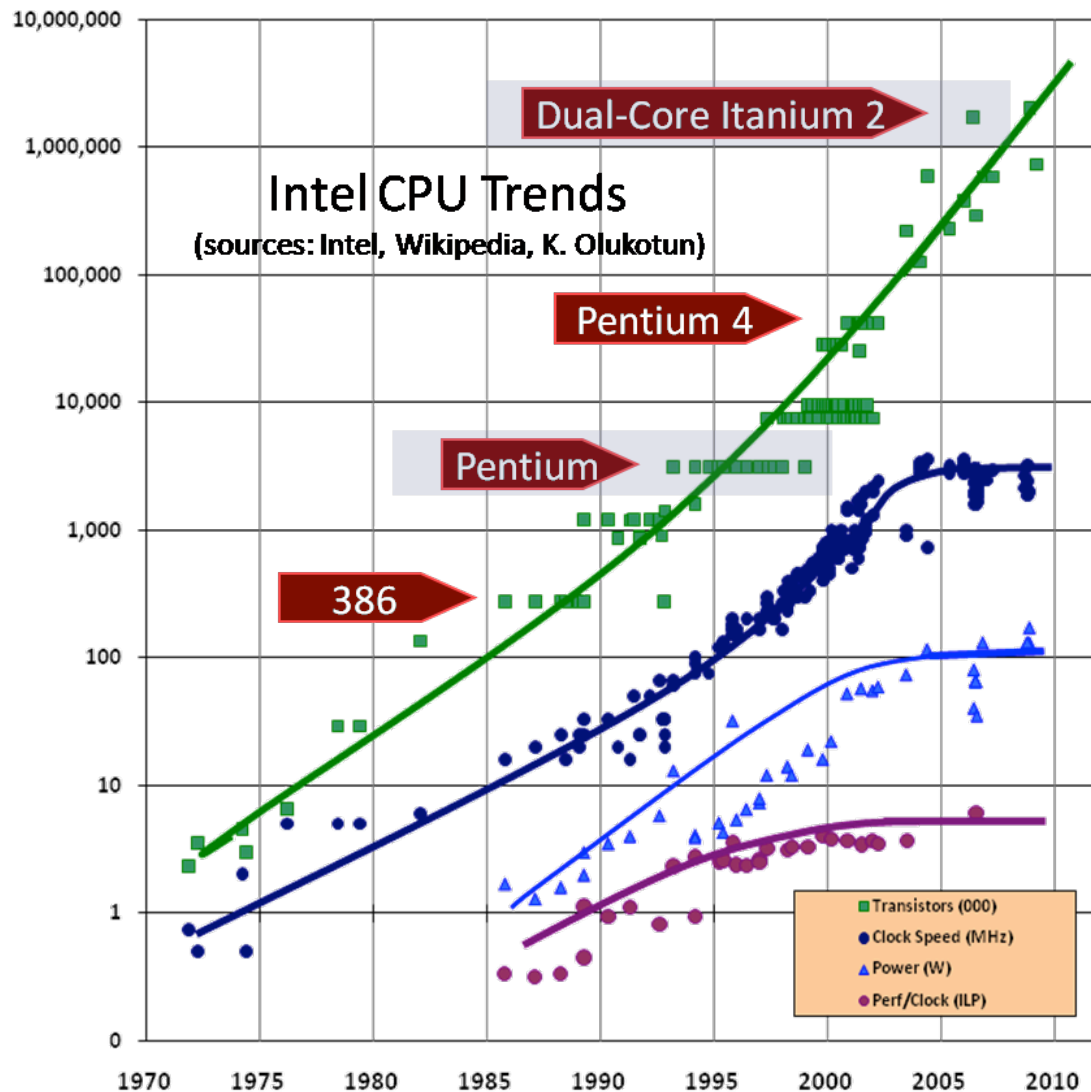
Future Akaros



Agenda

- Motivation (a.k.a. why are we doing all this?)
- Managing Cores in Akaros
- Parlib Overview and Evaluation
- Lithe Overview and Evaluation
- Experience porting Go to Akaros
- Summary

Resurgence of Parallelism



- Uniprocessor performance plateau



- Increasing transistors



- Multiple cores** at the max clock speed

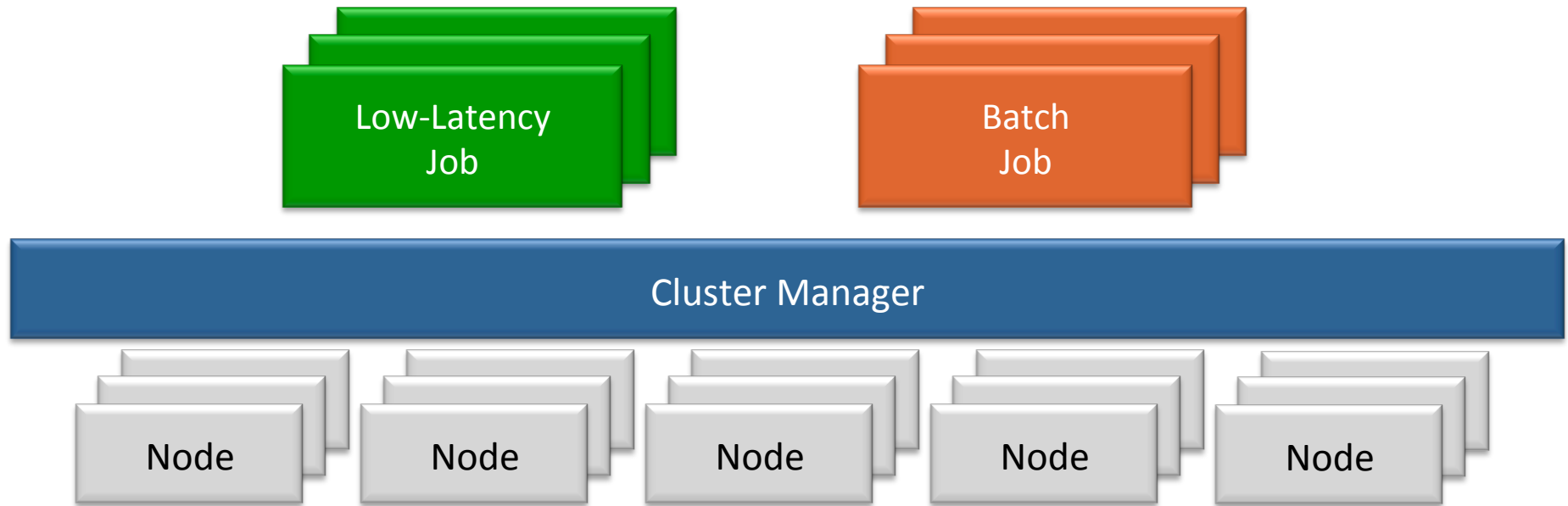
Graph source: "The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software"

<http://www.gotw.ca/publications/concurrency-ddj.htm>

Resurgence of Parallelism

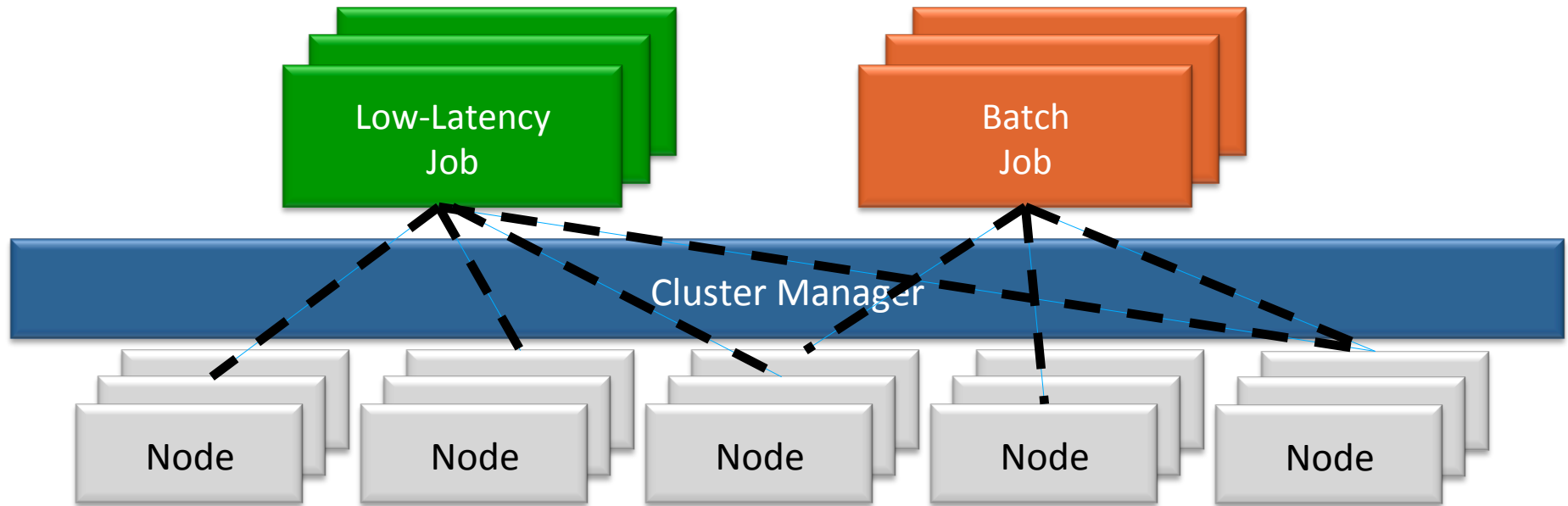
- Large-scale SMP machines are on the rise
 - What do we do with all these stinkin' cores?
- Parallelism is not new; can learn from the past
 - Large body of work spanning several decades
- The largest machines will be in data centers
 - Important enough to warrant engineering effort at all levels of the software stack

Low Latency vs. Batch Workloads



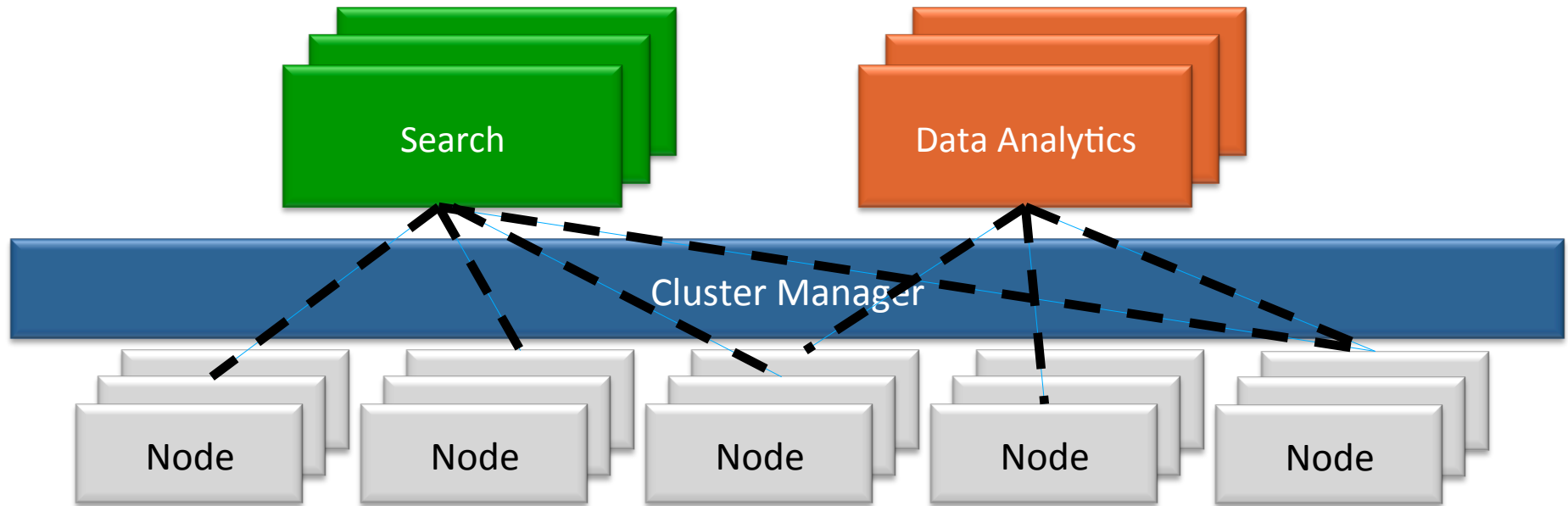
- Low-Latency Jobs
 - Long periods of inactivity → Bursts of high demand
- Batch Jobs
 - “Run-to-completion” model, no strict latency requirements

Low Latency vs. Batch Workloads



- Low-Latency Jobs
 - Long periods of inactivity → Bursts of high demand
- Batch Jobs
 - “Run-to-completion” model, no strict latency requirements

Low Latency vs. Batch Workloads



- Low-Latency Jobs
 - Long periods of inactivity → Bursts of high demand
- Batch Jobs
 - “Run-to-completion” model, no strict latency requirements

Data Center Provisioning

"Every year, we take the busiest minute of the busiest hour of the busiest day and build capacity on that"

– Scott Gulbransen, a spokesman for Intuit

Data Center Provisioning

- Current solutions based on static allocation of resources to individual jobs (done by cluster manager)
- Great for batch jobs! Crappy for low-latency ones ...

Data Center Provisioning

- Current solutions based on static allocation of resources to individual jobs (done by cluster manager)
- Great for batch jobs! Crappy for low-latency ones ...
- Low-latency jobs must be allocated all resources up-front for peak-demand → wastes resources when not actively used
- More resources? → global reshuffling of resources by cluster manager, killing batch jobs, etc.

Data Center Provisioning

- Current solutions based on static allocation of resources to individual jobs (done by cluster manager)
- Great for batch jobs! Crappy for low-latency ones ...
- Low-latency jobs must be allocated all resources up-front for peak-demand → wastes resources when not actively used
- More resources? → global reshuffling of resources by cluster manager, killing batch jobs, etc.

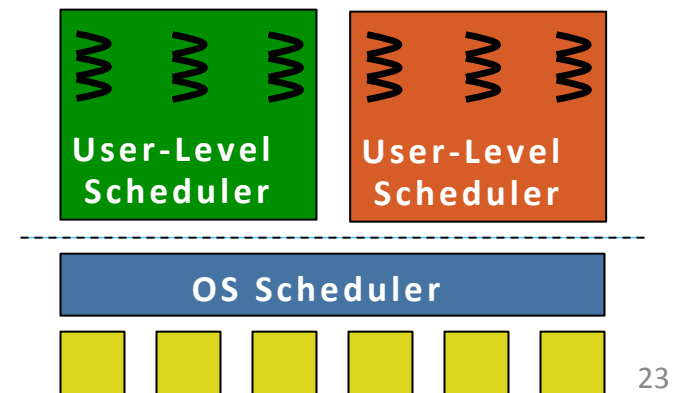
Good News: Akaros core management abstractions help to alleviate some of these problems

Bad News: Individual pieces work, but not yet integrated into to a fully usable system. We are working on it!

Managing Cores in Akaros

Managing Cores in Akaros

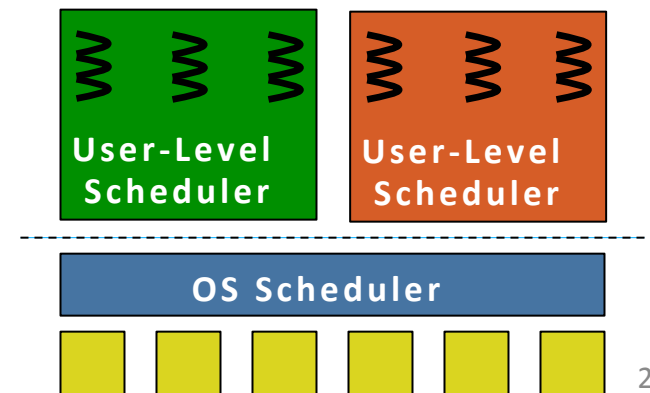
- The Many-Core Process
 - A single, unified abstraction for a parallel process
 - Request cores from the OS, not threads
 - All I/O completely asynchronous → process retains its cores at all times
 - Any application-level “threading” is implemented in user-space, completely decoupled from the kernel



Managing Cores in Akaros

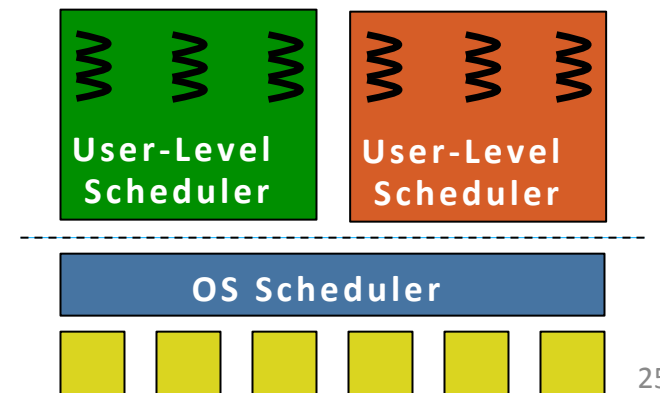
- The Many-Core Process

- A single, unified abstraction for a parallel process
- Request cores from the OS, not threads
- All I/O completely asynchronous → process retains its cores at all times
- Any application-level “threading” is implemented in user-space, completely decoupled from the kernel
- All cores gang scheduled by the kernel to increase performance of user-level locks/barriers
- No cores interrupted or preempted without prior agreement



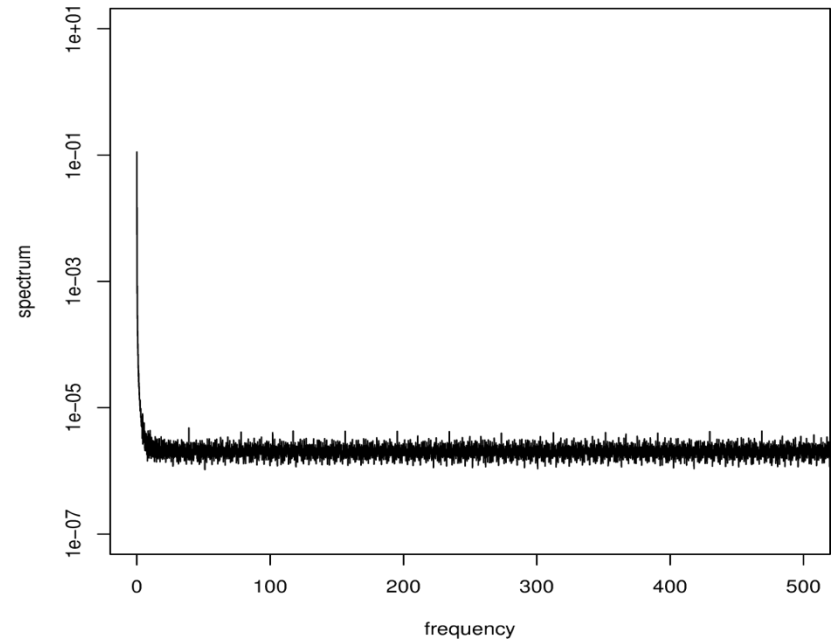
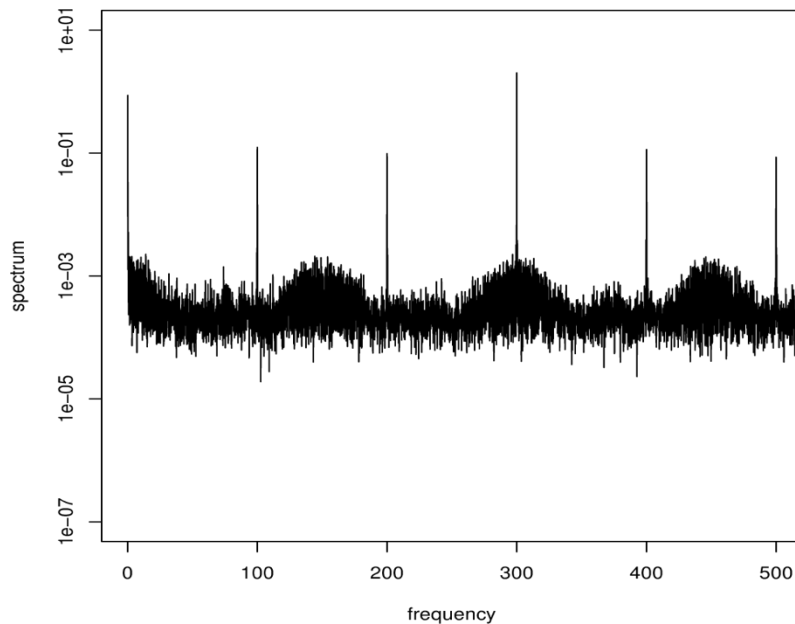
Managing Cores in Akaros

- The Many-Core Process
 - 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.
 - Express **parallelism** via core requests and **concurrency** via user-level threads.



Managing Cores in Akaros

- The Many-Core Process
 - Experiments show that MCP cores have an order of magnitude less noise and fewer periodic signals than Linux
 - Less interference → better CPU isolation
 - **Details in the FTQ section of Barret Rhoden's dissertation**



Managing Cores in Akaros

- Extended API for Provisioning Cores

Managing Cores in Akaros

- Extended API for Provisioning Cores
 - Separate notions of allocation and provisioning

	P0	P1					
Provisioned:	0	0					
Allocated:	0	0					
Wants:	0	0					

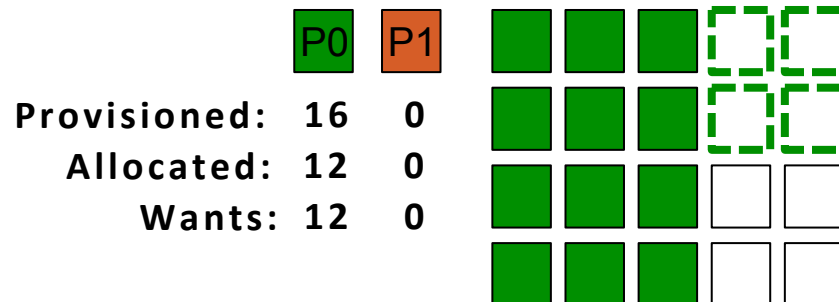
Managing Cores in Akaros

- Extended API for Provisioning Cores
 - Separate notions of allocation and provisioning
 - Low-latency jobs provision cores for future use (at peak-demand)

	<div>P0</div>	<div>P1</div>	
Provisioned:	16	0	<div><div></div><div></div><div></div><div></div><div></div></div>
Allocated:	0	0	<div><div></div><div></div><div></div><div></div><div></div></div>
Wants:	0	0	<div><div></div><div></div><div></div><div></div><div></div></div>





















Managing Cores in Akaros

- Extended API for Provisioning Cores
 - Separate notions of allocation and provisioning
 - Low-latency jobs provision cores for future use (at peak-demand)
 - Only request the number of cores necessary for real-time demand



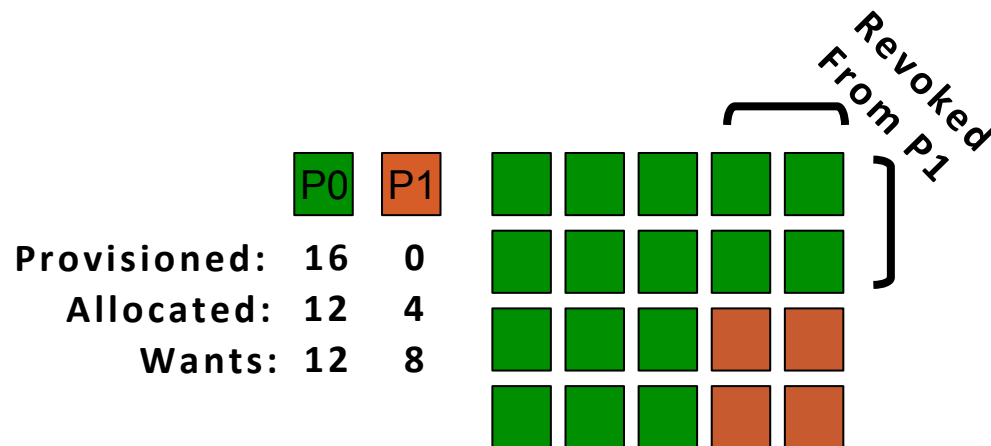
Managing Cores in Akaros

- Extended API for Provisioning Cores
 - Separate notions of allocation and provisioning
 - Low-latency jobs provision cores for future use (at peak-demand)
 - Only request the number of cores necessary for real-time demand
 - Batch jobs use remaining cores in the meantime

	P0	P1					
Provisioned:	16	0					
Allocated:	12	8					
Wants:	12	8					
							

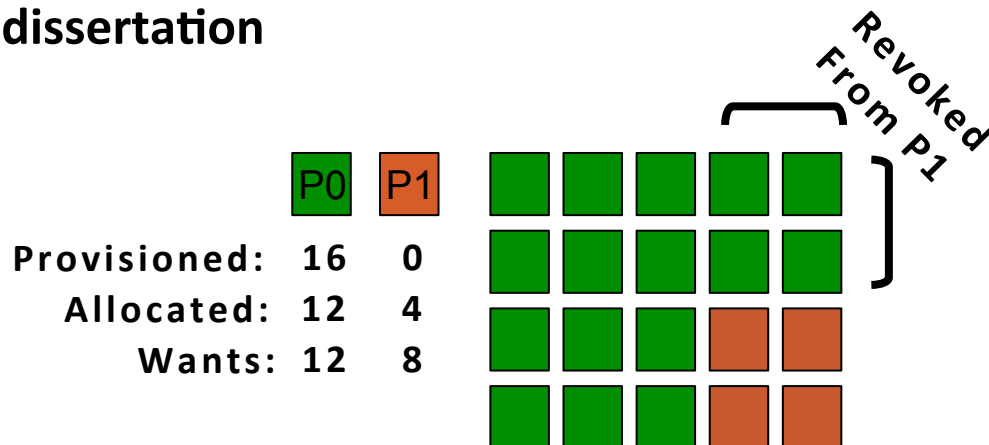
Managing Cores in Akaros

- Extended API for Provisioning Cores
 - Separate notions of allocation and provisioning
 - Low-latency jobs provision cores for future use (at peak-demand)
 - Only request the number of cores necessary for real-time demand
 - Batch jobs use remaining cores in the meantime
 - Revoke cores from batch jobs to meet increased real-time demand of low-latency jobs (but don't kill them unless absolutely necessary!)



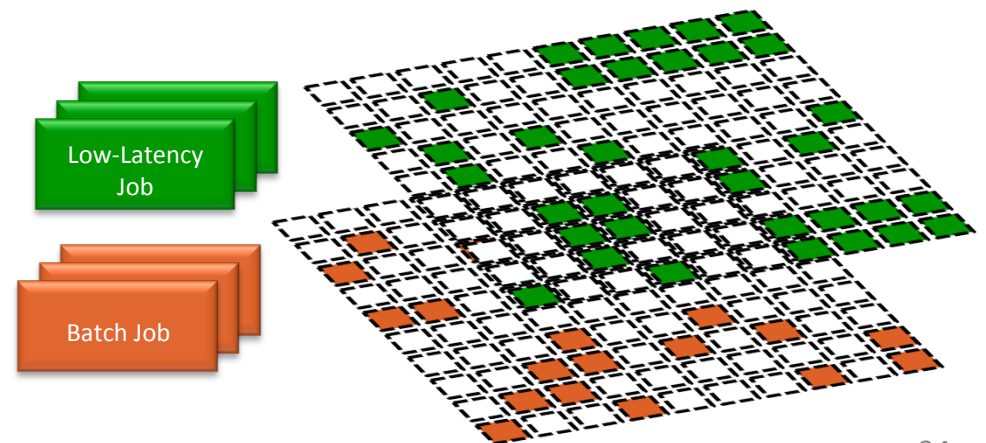
Managing Cores in Akaros

- Extended API for Provisioning Cores
 - Separate notions of allocation and provisioning
 - Low-latency jobs provision cores for future use (at peak-demand)
 - Only request the number of cores necessary for real-time demand
 - Batch jobs use remaining cores in the meantime
 - Revoke cores from batch jobs to meet increased real-time demand of low-latency jobs (but don't kill them unless absolutely necessary!)
 - **Experiments showing this in action can be found chapter 7 of Barret Rhoden's dissertation**



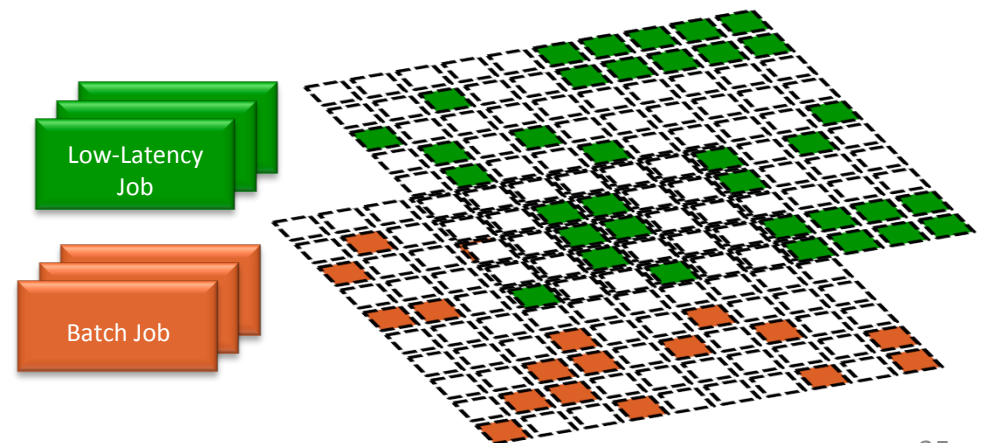
Integration with Cluster Manager

- Grand Vision (not yet implemented anywhere!):
 - Partition jobs by type (Low-Latency, Batch)
 - Provision for Low-Latency → Allocate for Batch
 - Schedule and run batch jobs as before
 - Schedule low-latency jobs on same resources, and let provisioning take care of the rest



Integration with Cluster Manager

- Assuming we have a system like this
 - Individual jobs need to take control of their cores!
 - Request cores as they need them, release them as they don't
 - Mux user-level threads on top of them
 - **This is where Parlib, Lithe, and Go come in**



Traditional Threading Models Used for User-Level Scheduling

- 1:1 – User tells kernel what to schedule where/when
 - **Mach** and recent **Linux *switchto*** extensions
- M:N – Multiplex user-threads onto kernel-threads
 - **Psyche**, **Scheduler Activations** and **Capriccio**
- How to deal with Blocking I/O?
 - *Delegate* or *Activation* spawned as notification
 - Don't! → Make all I/O non-blocking
- Akaros/Parlib follows a combination of these approaches

Akaros Threading Model

- Decouple threading models in Kernel and User-Space
 - Kernel doesn't know about user threads
 - User doesn't know about kernel threads
 - Transition context between the two: **vcore context**
- Enabled by:
 - Ability to request cores, not threads from the underlying OS
 - Fully asynchronous system call interface

Parlib

- Framework for building user-level schedulers under the Akaros threading model (written in C)
- Developers use parlib to build application-specific schedulers, customized to their own particular needs.

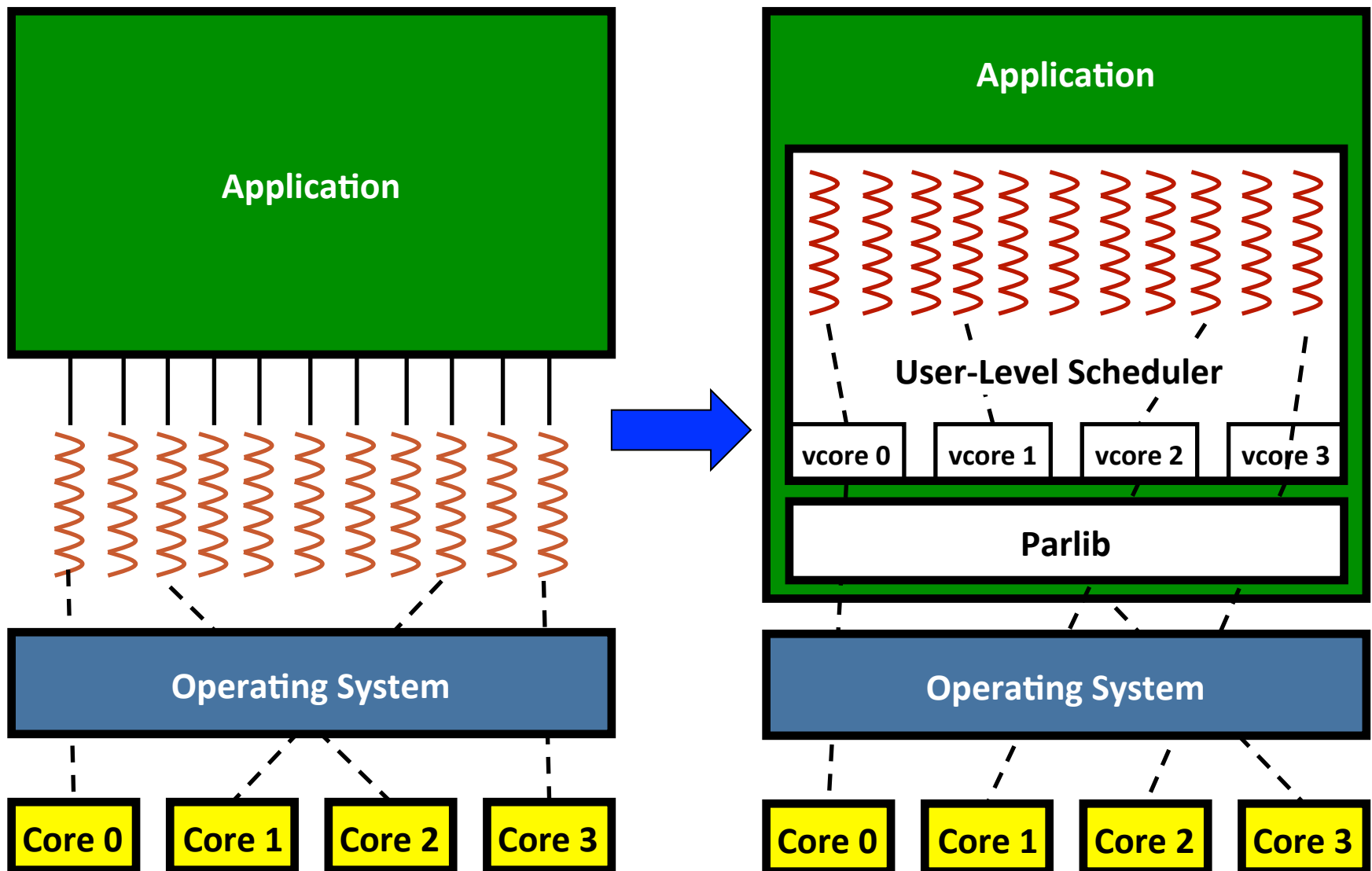
Parlib

- Framework for building user-level schedulers under the Akaros threading model (written in C)
- Developers use parlib to build application-specific schedulers, customized to their own particular needs.
- For legacy applications (or those that don't care as much about their scheduling), we provide a parlib-based implementation of **pthread**s that is API compatible with the Linux NPTL

Parlib

- Framework for building user-level schedulers under the Akaros threading model (written in C)
- Developers use parlib to build application-specific schedulers, customized to their own particular needs.
- For legacy applications (or those that don't care as much about their scheduling), we provide a parlib-based implementation of **pthread**s that is API compatible with the Linux NPTL
- Supplement to glibc, which is also supported on Akaros (c, c++, libgomp)
- Limited port of parlib to Linux

Parlib Overview



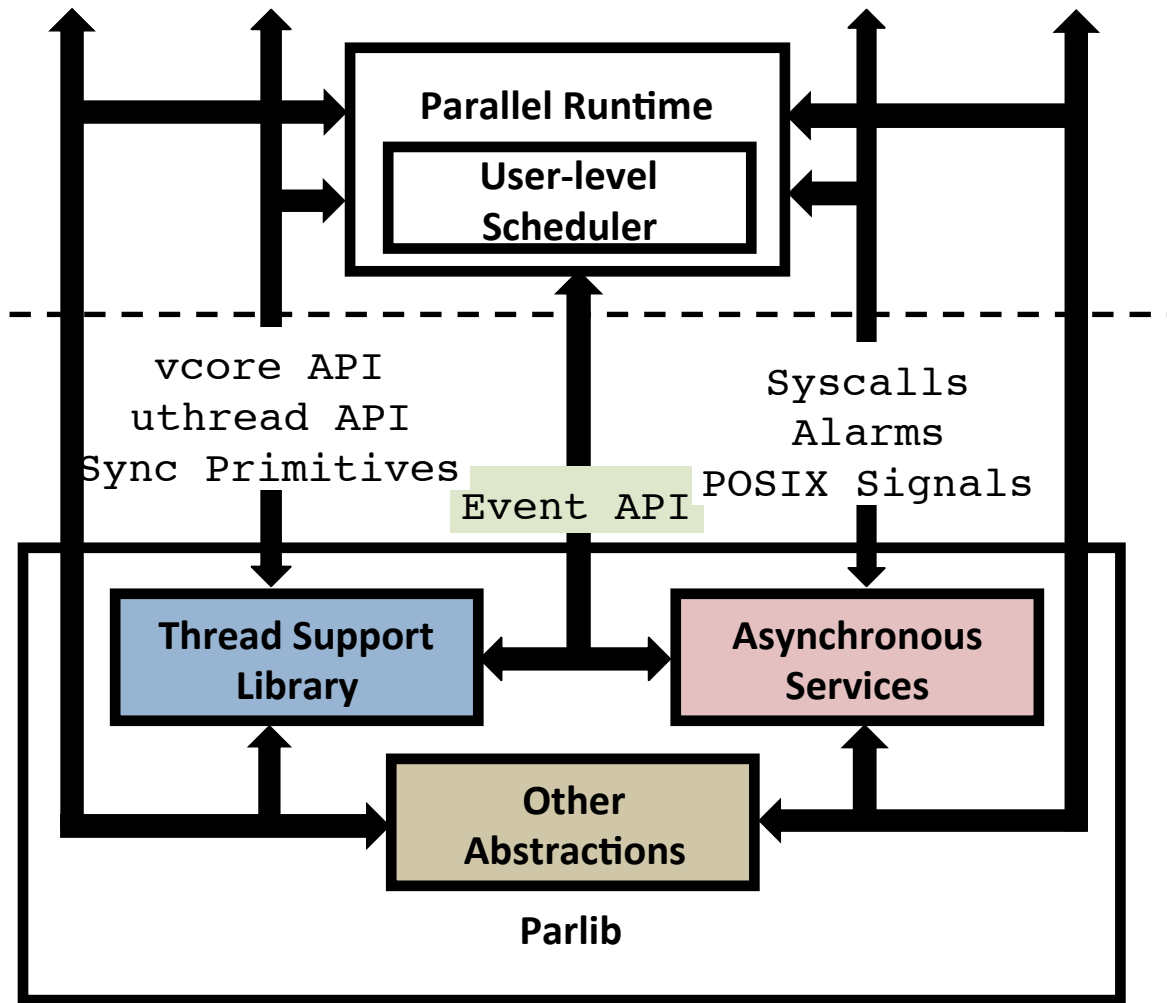
Parlib Abstractions

- Thread Support Library
- Asynchronous Services
- Asynchronous Event Delivery Mechanism
- Other Abstractions

Parlib Abstractions

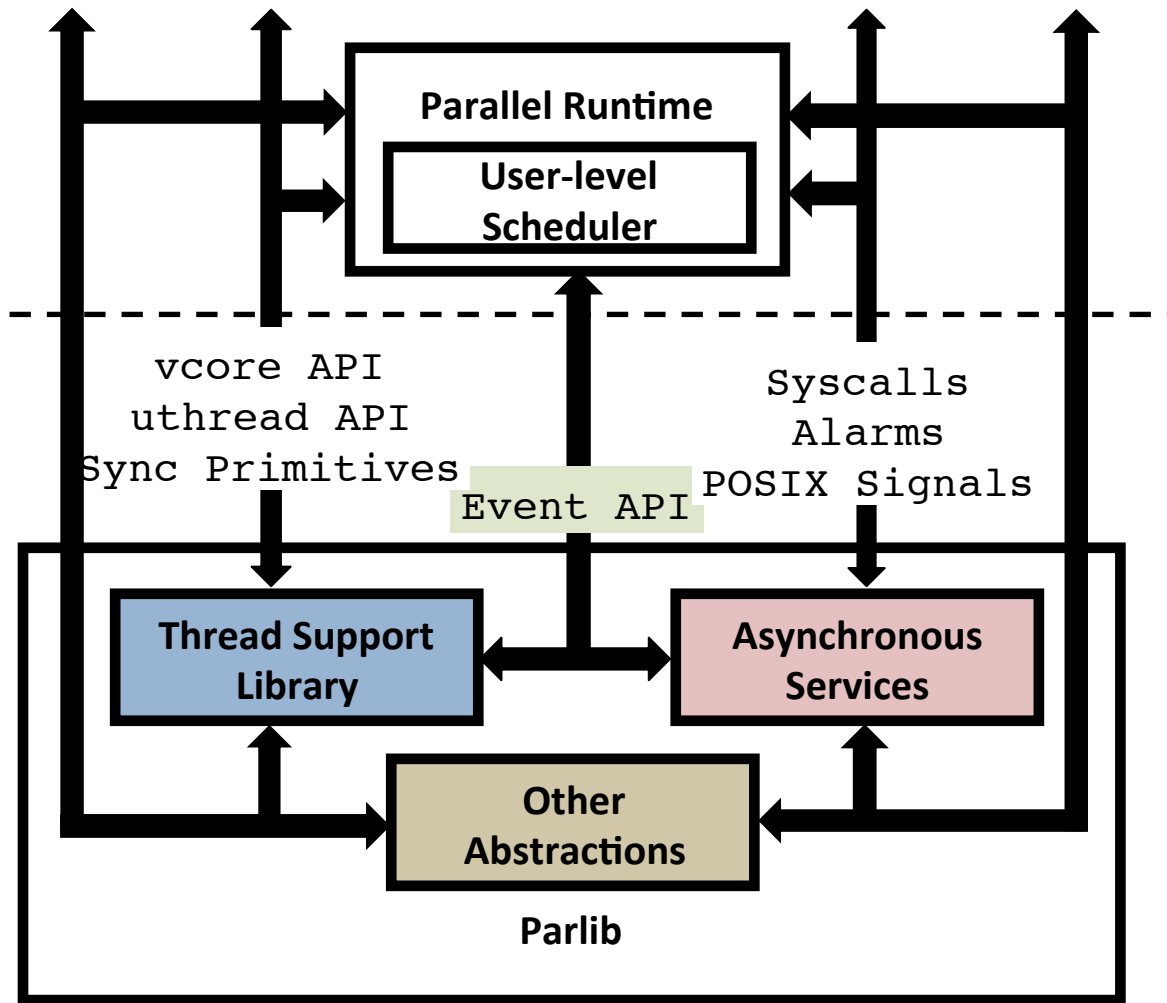
- Thread Support Library
 - Vcores, uthreads, and synchronization primitives
- Asynchronous Services
 - Syscalls, Alarms, POSIX Signal Delivery
- Asynchronous Event Delivery Mechanism
 - Active Messages triggered by events from async services
- Other Abstractions
 - Wait Free List, Slab Allocator, Dynamic Thread Local Storage

Parlib API



- Bi-directional API
 - Library Calls implemented by abstraction itself
 - Callbacks implemented by consumer of that abstraction

Parlib API



- Bi-directional API
 - Library Calls implemented by abstraction itself
 - Callbacks implemented by consumer of that abstraction
- Library Calls initiate operation
- Callbacks run scheduler code, or respond to external events

Uthread Context vs. Vcore Context

- Vcore Context
 - a.k.a. scheduler context
 - Transition context popped into whenever the kernel hands a core to user space (first time coming up, notification, etc.)
 - Has own stack, register state, and thread local storage (TLS)
 - Serves to handle events, run scheduler code, etc.

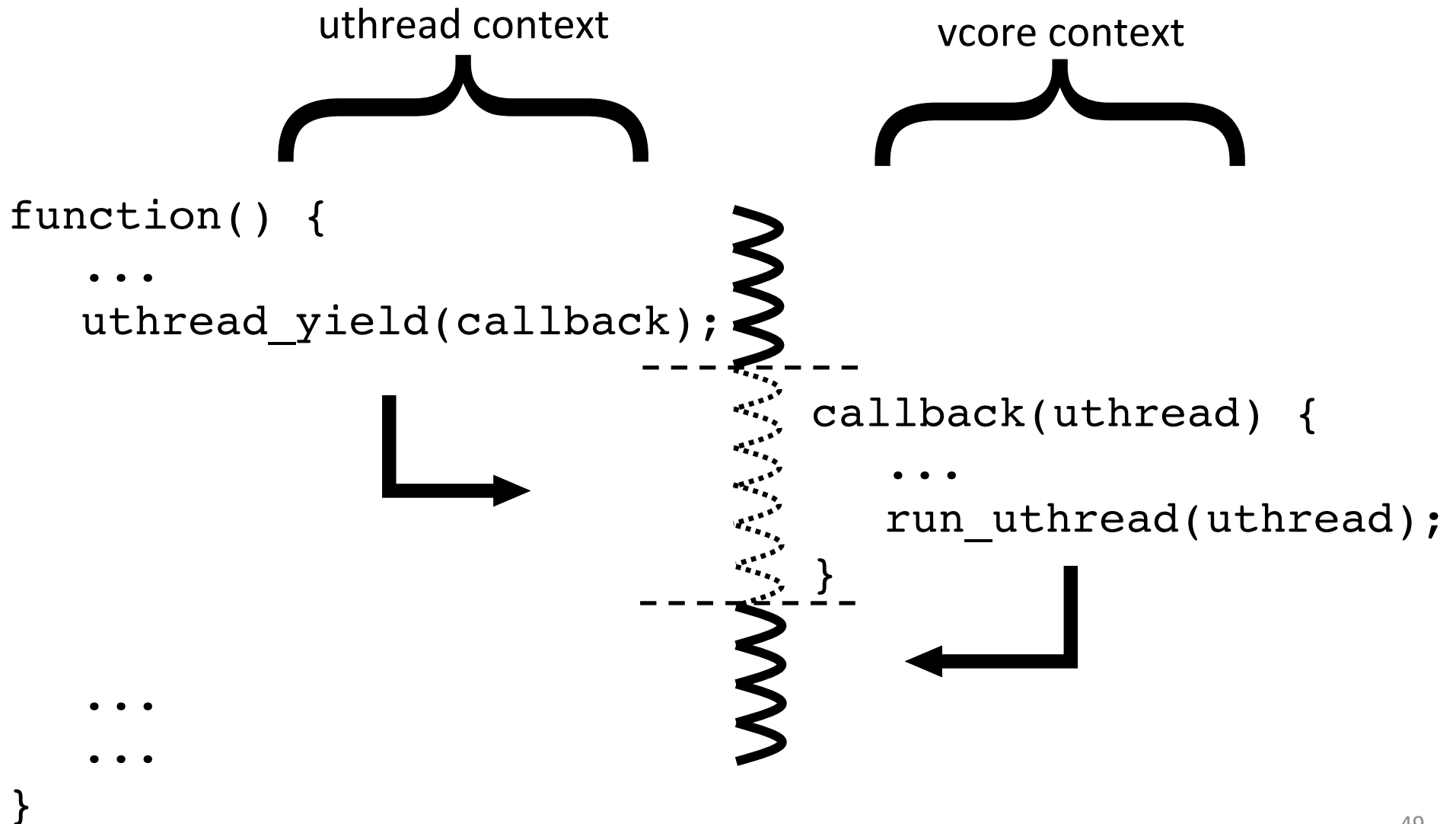
Uthread Context vs. Vcore Context

- Vcore Context
 - a.k.a. scheduler context
 - Transition context popped into whenever the kernel hands a core to user space (first time coming up, notification, etc.)
 - Has own stack, register state, and thread local storage (TLS)
 - Serves to handle events, run scheduler code, etc.
 - Always entered at the **top** of the stack and never returns (no need to save its context upon exiting)
 - Only options in vcore context are to find a uthread to run, or yield the vcore back to the system
 - User-space counterpart to interrupt context in the kernel

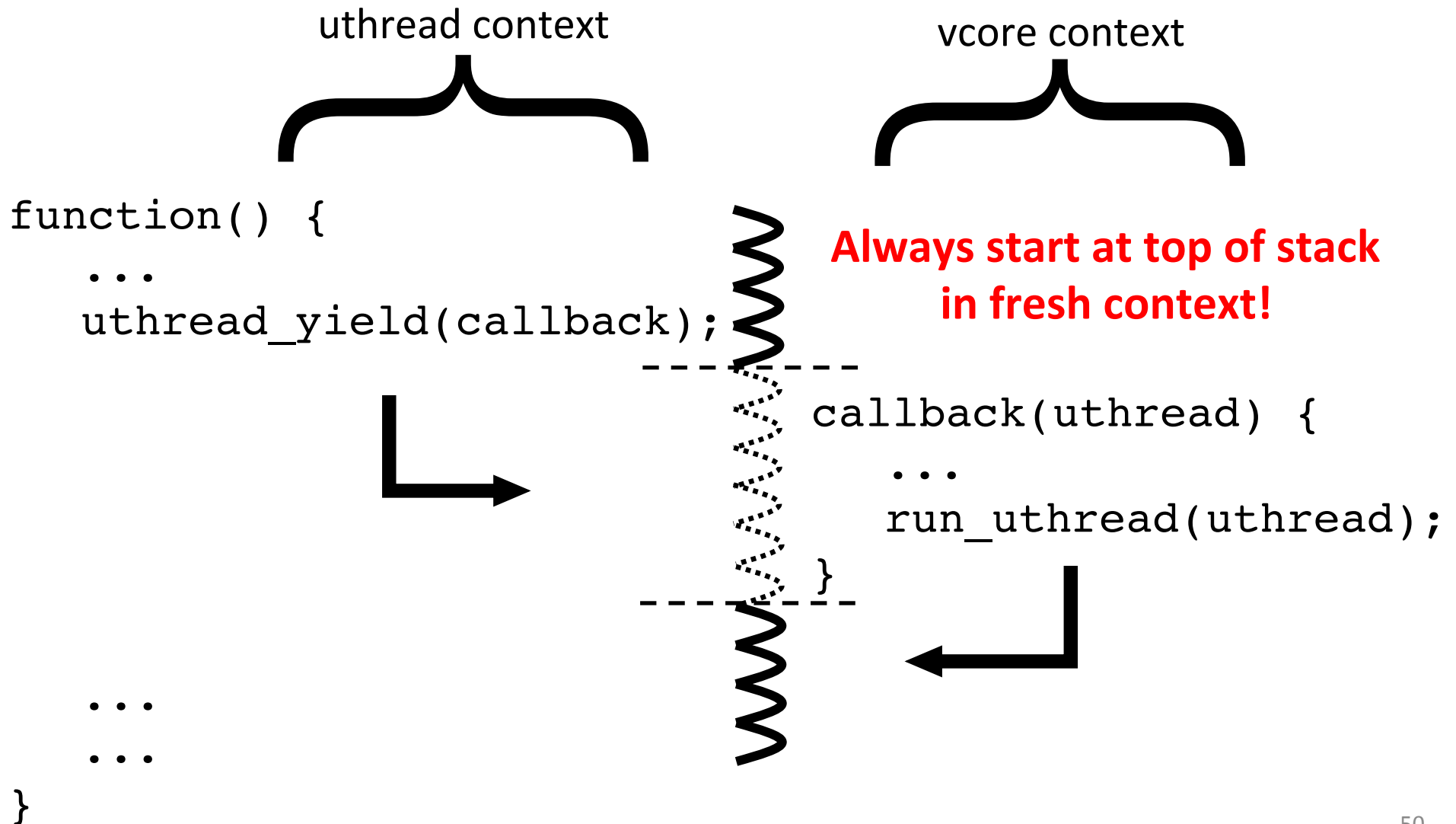
Uthread Context vs. Vcore Context

- Uthread Context
 - Normally what we think of as a thread context
 - Serves to run application code inside a thread
 - Transitions to vcore context to make scheduling decisions

Uthread Context vs. Vcore Context



Uthread Context vs. Vcore Context



Events and Async Services

Events and Async Services

- Event Delivery follows producer/consumer model
 - Vcores set up queues for event delivery (consumers)
 - Async services post events to those queues (producers)
 - Consumers register handlers based on **event_type** to be triggered when event extracted from queue
 - Events carry payloads specific to event type
 - Think active messages

Events and Async Services

- Event Delivery follows producer/consumer model
 - Vcores set up queues for event delivery (consumers)
 - Async services post events to those queues (producers)
 - Consumers register handlers based on **event_type** to be triggered when event extracted from queue
 - Events carry payloads specific to event type
 - Think active messages
- Optional notification when event posted to queue
 - Post event → Interrupt uthread → Run **vcore_entry()**

Events and Async Services

- Event Delivery follows producer/consumer model
 - Vcores set up queues for event delivery (consumers)
 - Async services post events to those queues (producers)
 - Consumers register handlers based on **event_type** to be triggered when event extracted from queue
 - Events carry payloads specific to event type
 - Think active messages
- Optional notification when event posted to queue
 - Post event → Interrupt uthread → Run **vcore_entry()**

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

Example: Syscall Completion

- Intercept syscalls, and block thread in user-space on I/O operation
- Notify process and unblock thread after syscall completion event

Example: Syscall Completion

Syscall Service

```
void akaros_syscall(args...) {
```

Scheduler Implementation

```
event_queue[max_vcores()];
ev_handlers[EV_SYSCALL] = handle_syscall;

void handle_syscall(event_t* e) {

}
```


Example: Syscall Completion

Syscall Service

- First, issue syscall ...

```
void akaros_syscall(args...) {  
    struct syscall syscall;  
    syscall.args = args;  
    do_syscall(&syscall); // always returns  
    ...  
}
```

Scheduler Implementation

```
event_queue[max_vcores()];  
ev_handlers[EV_SYSCALL] = handle_syscall;  
  
void handle_syscall(event_t* e) {  
  
}
```

Example: Syscall Completion

Syscall Service

- First, issue syscall ...
- If not done
 - Yield to vcore context
 - Save uthread reference
 - Set up event queue
 - Set to notify

```
void akaros_syscall(args...) {
    struct syscall syscall;
    syscall.args = args;
    do_syscall(&syscall); // always returns
    if ( !syscall.done ) {
        uthread_yield(callback, syscall);
    }
    void callback(uthread, syscall) {
        syscall->uthread = uthread;
        syscall->evq = event_queue[vcore_id()];
        syscall->notify = true;
    }
}
```

Scheduler Implementation

```
event_queue[max_vcores()];
ev_handlers[EV_SYSCALL] = handle_syscall;

void handle_syscall(event_t* e) {

}
```

Example: Syscall Completion

Syscall Service

- First, issue syscall ...
- If not done
 - Yield to vcore context
 - Save uthread reference
 - Set up event queue
 - Set to notify

```
void akaros_syscall(args...) {
    struct syscall syscall;
    syscall.args = args;
    do_syscall(&syscall); // always returns
    if ( !syscall.done ) {
        uthread_yield(callback, syscall);
    }
    void callback(uthread, syscall) {
        syscall->uthread = uthread;
        syscall->evq = event_queue[vcore_id()];
        syscall->notify = true;
    }
}
```

Scheduler Implementation

- Grap syscall from payload
- Grab uthread reference
- Make thread runnable again
- sched_entry() implicit

```
event_queue[max_vcores()];
ev_handlers[EV_SYSCALL] = handle_syscall;

void handle_syscall(event_t* e) {
    syscall = (struct syscall*)e.payload;
    uthread = syscall->uthread;
    uthread_runnable(uthread);
}
```

Parlib-based Pthreads

- Per-vcore run queues with rudimentary stealing algorithm
- Extended API for setting scheduling period based on alarms
- Integration with high-level sync primitives for **mutex**, **barrier**, **condition variables**, and **futexes**
- Integration with the Async Syscall service
- Rudimentary affinity algorithm for preferred vcore queues
- Careful cache alignment of all data structures (per vcore data and thread struct data interfere minimally)

Parlib on Linux

- Port of parlib to Linux (including pthread library)
- Fakes vcore abstraction and has limited support for asynchronous syscalls
- So long as only 1 application running, good enough for experiments → more fair comparison to Linux NPTL

Parlib on Linux

- Port of parlib to Linux (including pthread library)
- Fakes vcore abstraction and has limited support for asynchronous syscalls
- So long as only 1 application running, good enough for experiments → more fair comparison to Linux NPTL
- Experiments
 - Context Switch Overhead
 - Flexibility in scheduling policies
 - NAS Benchmarks (OpenMP)
 - Kweb throughput

Parlib on Linux

- Port of parlib to Linux (including pthread library)
- Fakes vcore abstraction and has limited support for asynchronous syscalls
- So long as only 1 application running, good enough for experiments → more fair comparison to Linux NPTL
- Experiments
 - **Context Switch Overhead**
 - Flexibility in scheduling policies
 - NAS Benchmarks (OpenMP)
 - **Kweb throughput**

Parlib Experiments

- Machine Specs (c99):
 - 2 Ivy Bridge, 8-core 2.6Ghz CPUs (16 cores, 32 hyperthreads)
 - 8 16GB DDR3 1600Mhz DIMMs (128GB total)
 - 4 Intel i350 1GbE interfaces
 - 1 1TB Hard Disk Drive

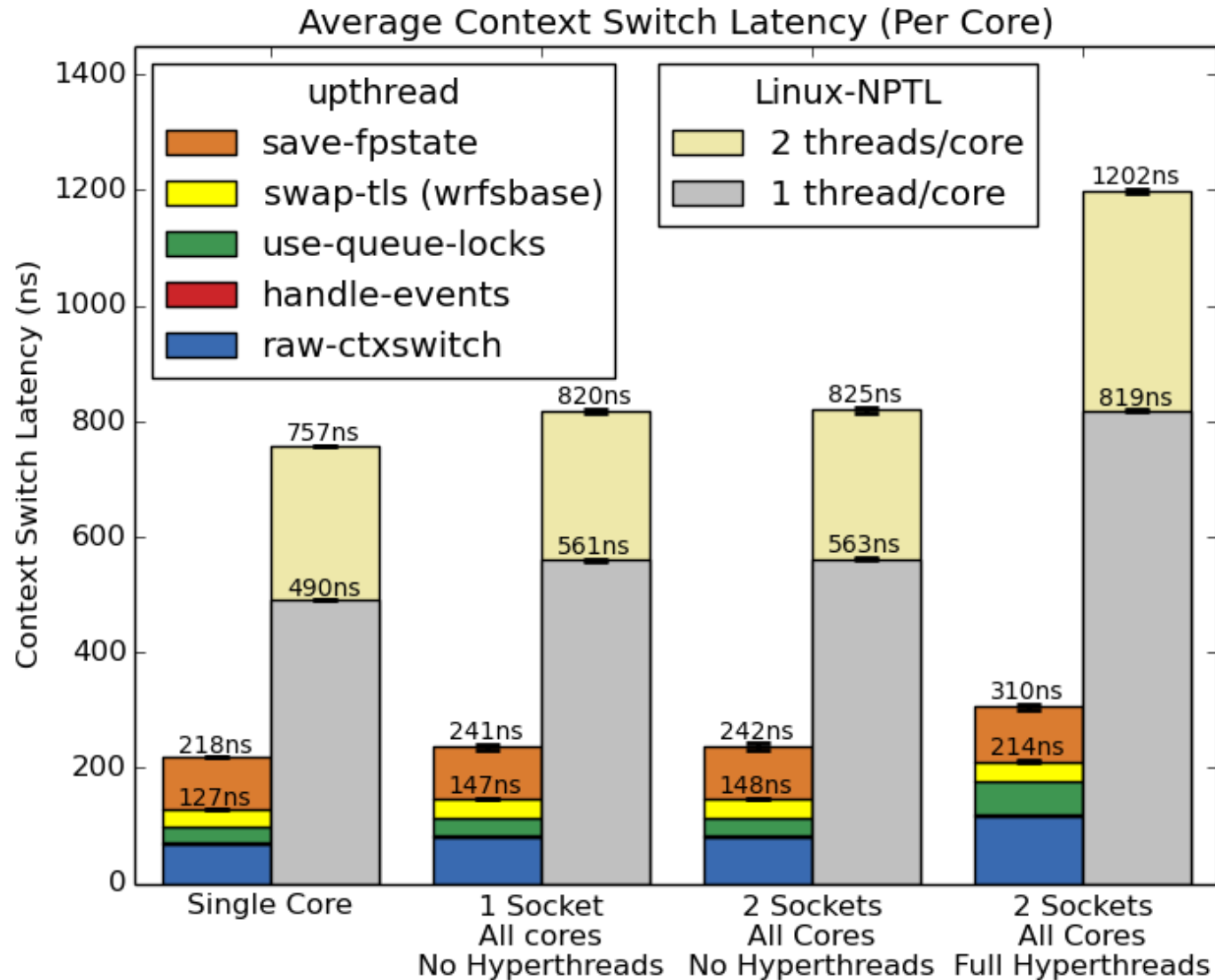
Parlib Experiments

- Machine Specs (c99):
 - 2 Ivy Bridge, 8-core 2.6Ghz CPUs (**16 cores, 32 hyperthreads**)
 - 8 16GB DDR3 1600Mhz DIMMs (128GB total)
 - 4 Intel i350 **1GbE** interfaces
 - 1 1TB Hard Disk Drive

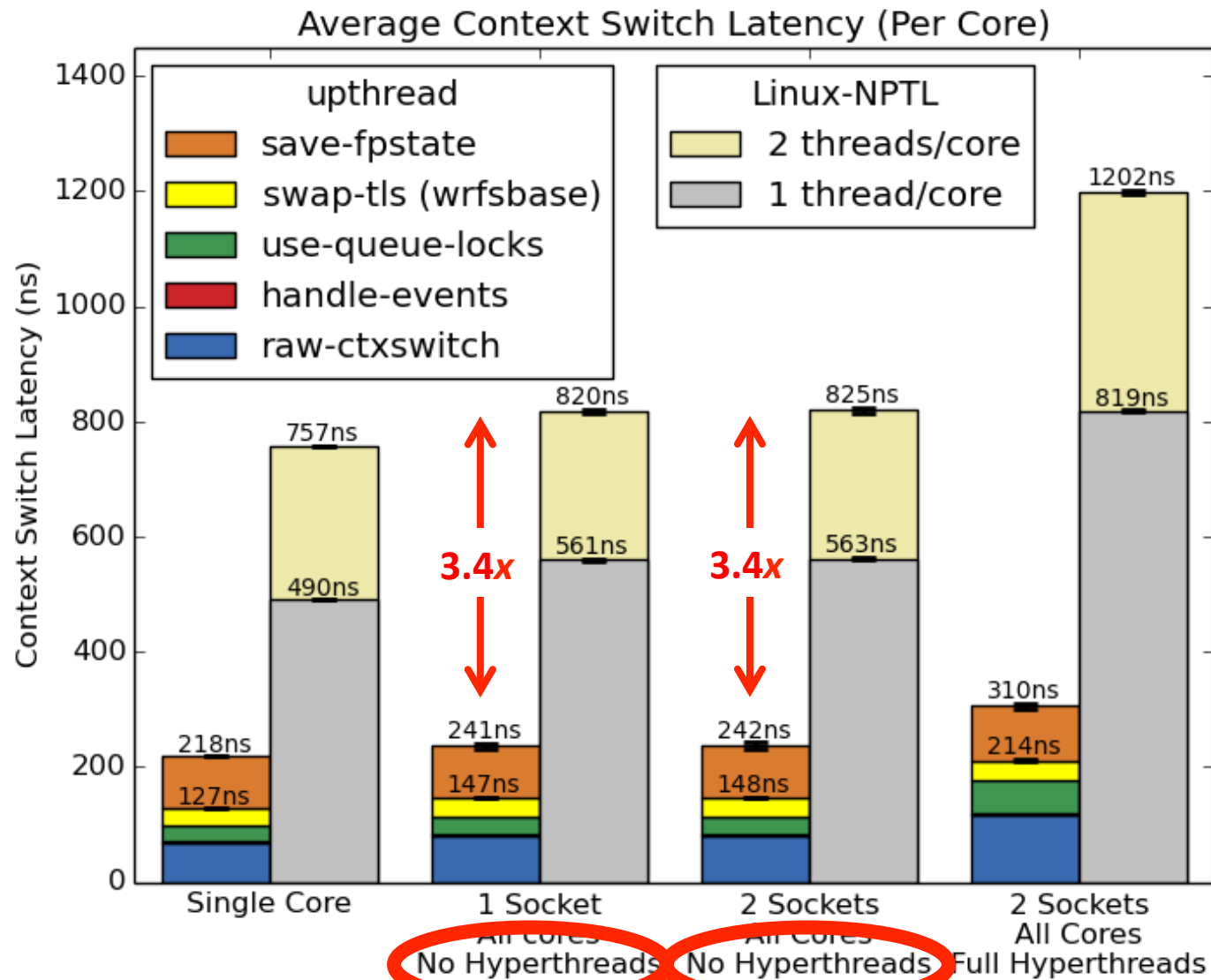
Parlib Experiments

- Machine Specs (c99):
 - 2 Ivy Bridge, 8-core 2.6Ghz CPUs (**16 cores, 32 hyperthreads**)
 - 8 16GB DDR3 1600Mhz DIMMs (128GB total)
 - 4 Intel i350 **1GbE** interfaces
 - 1 1TB Hard Disk Drive
- 3.13.0-32-generic kernel with patches from Andi Kleen to enable the **rdfsbase** and **wrfsbase** instructions (for user-level TLS support)

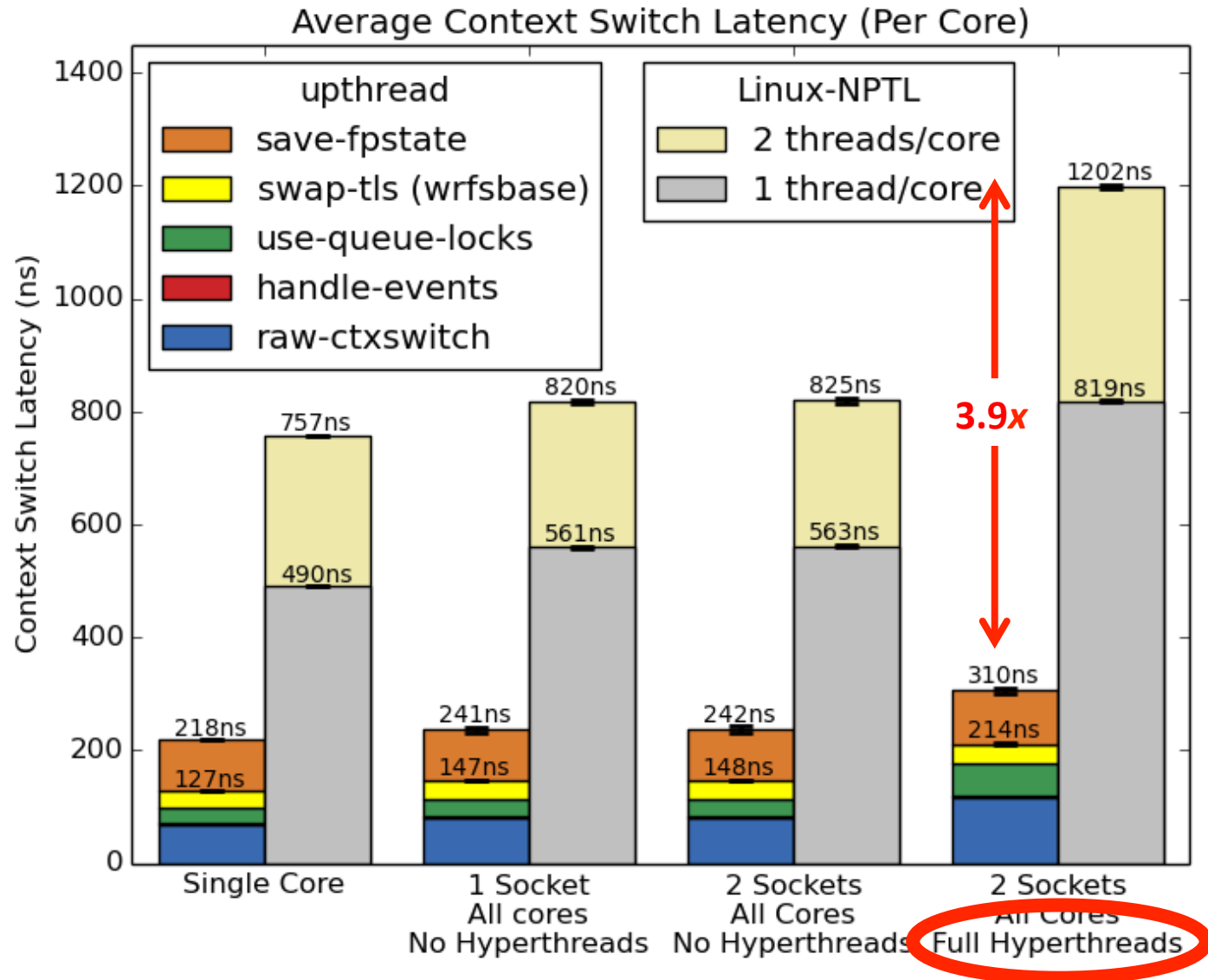
Parlib Experiments



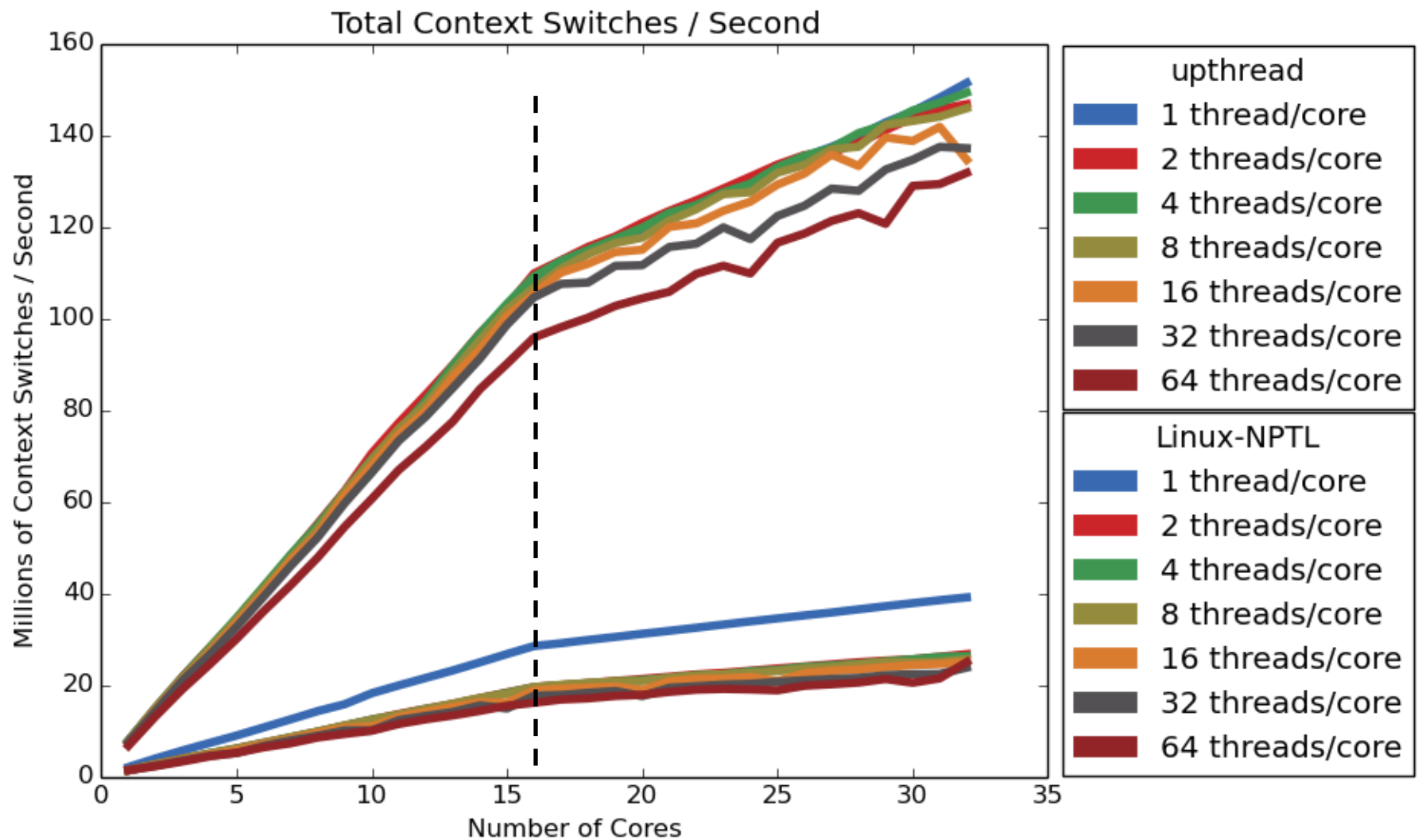
Parlib Experiments



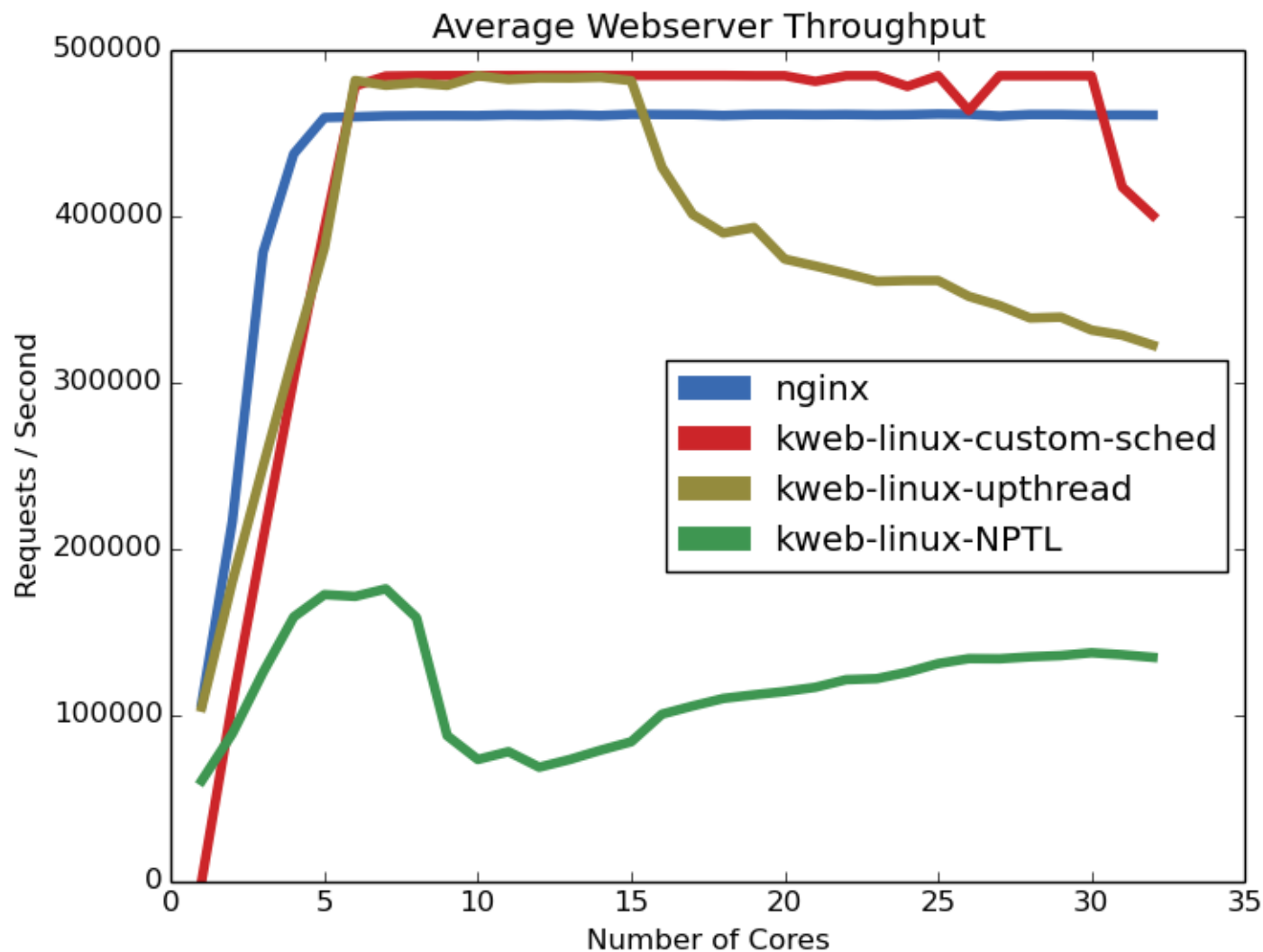
Parlib Experiments



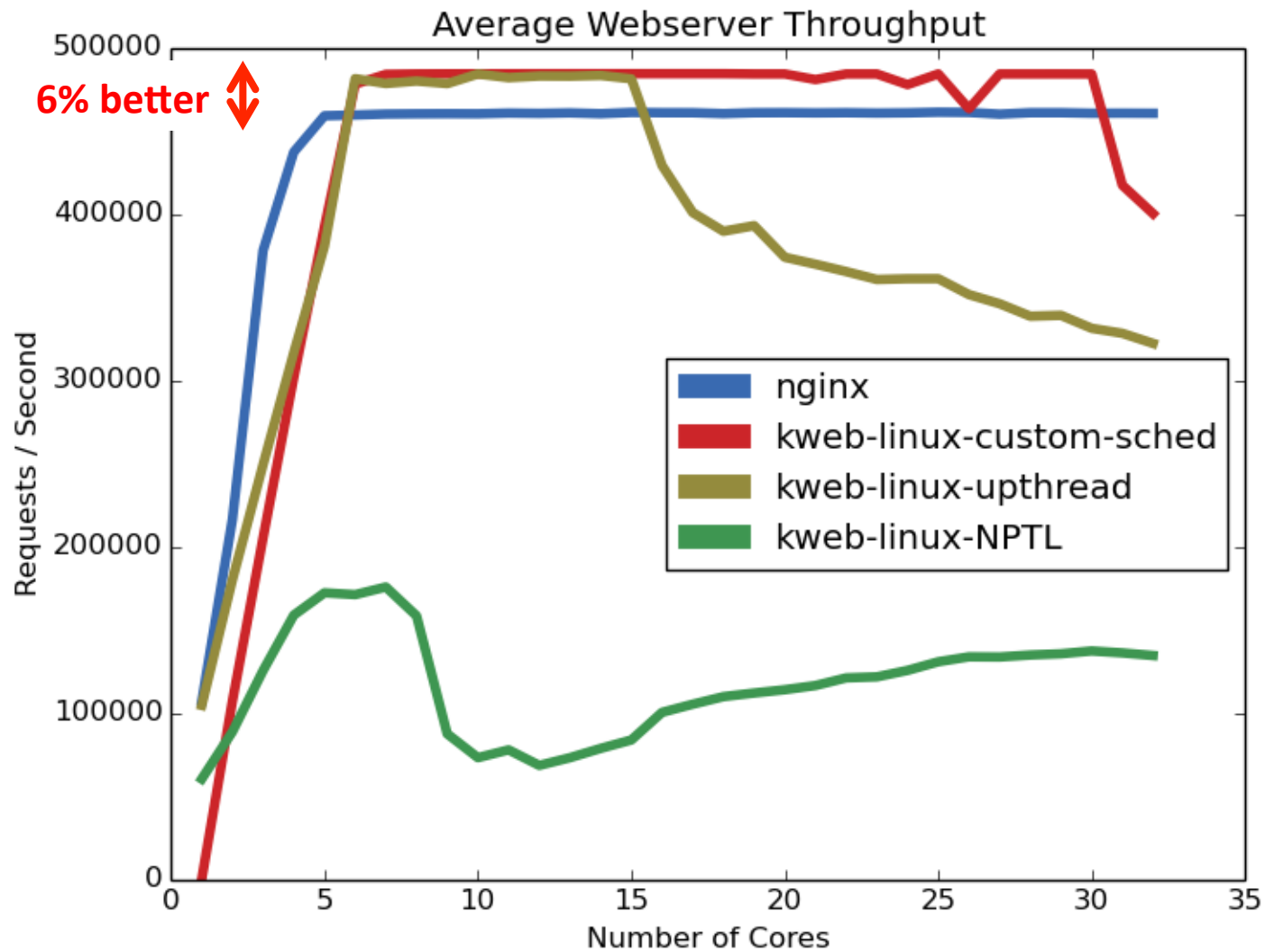
Parlib Experiments



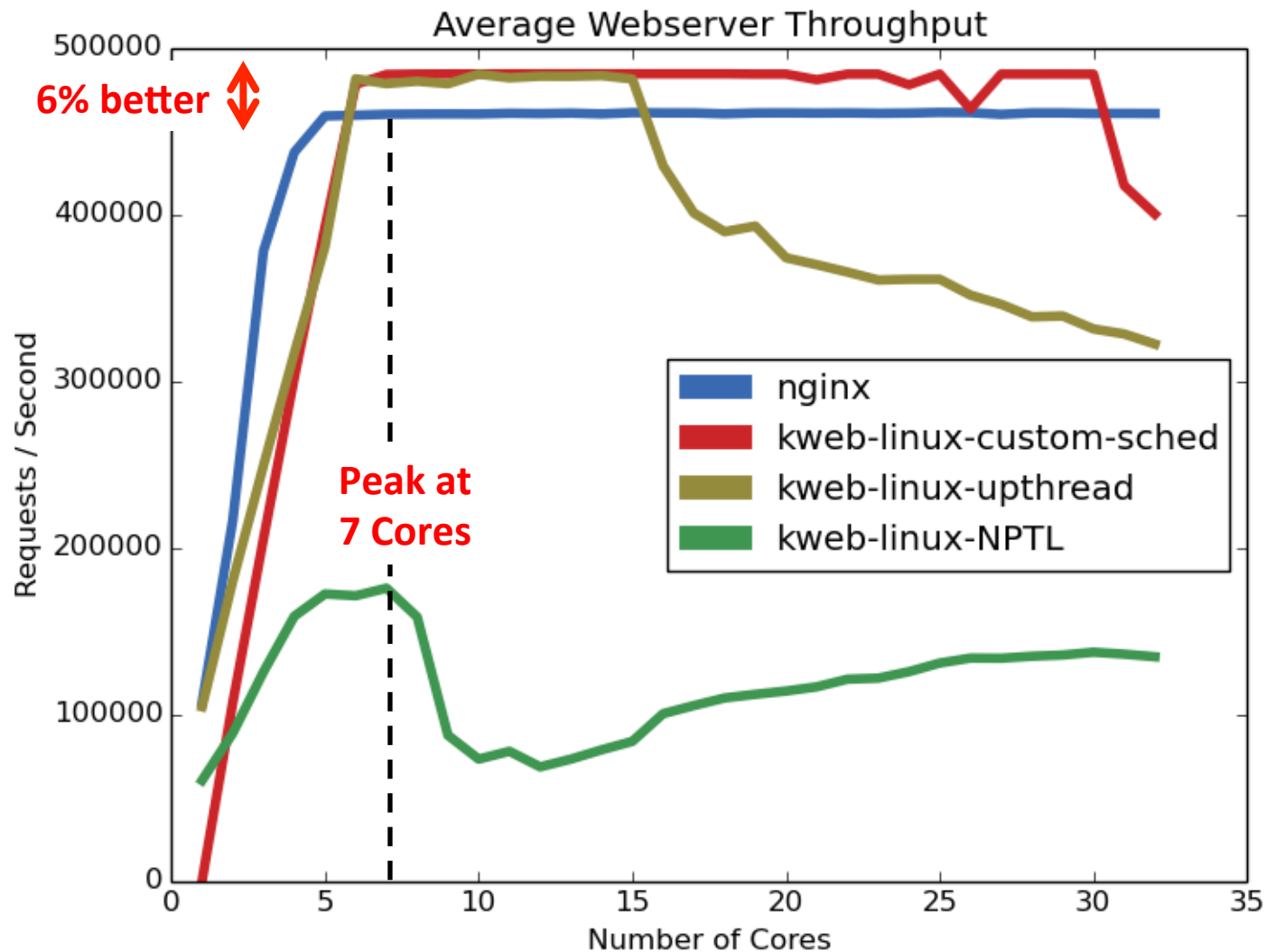
Parlib Experiments



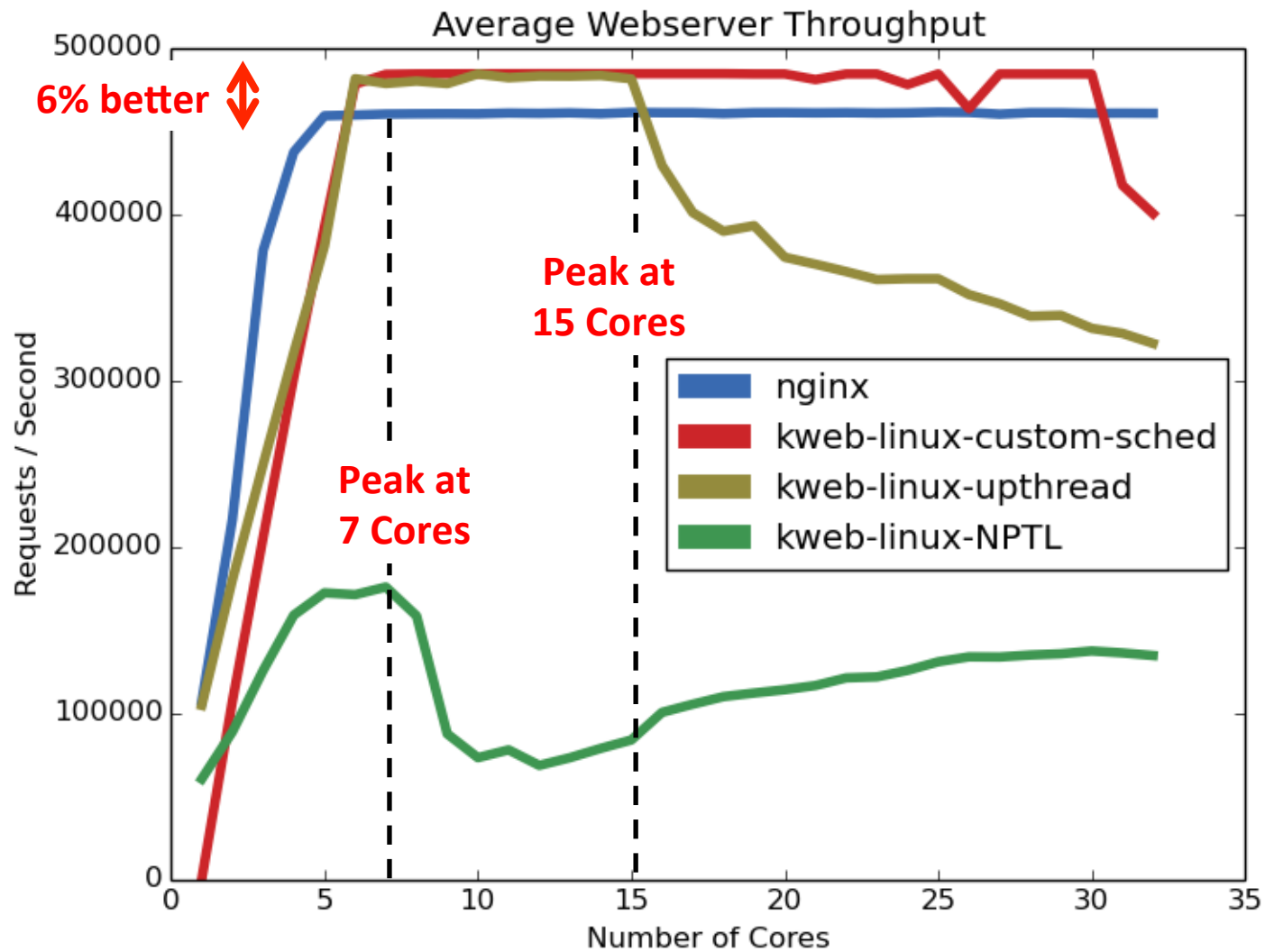
Parlib Experiments



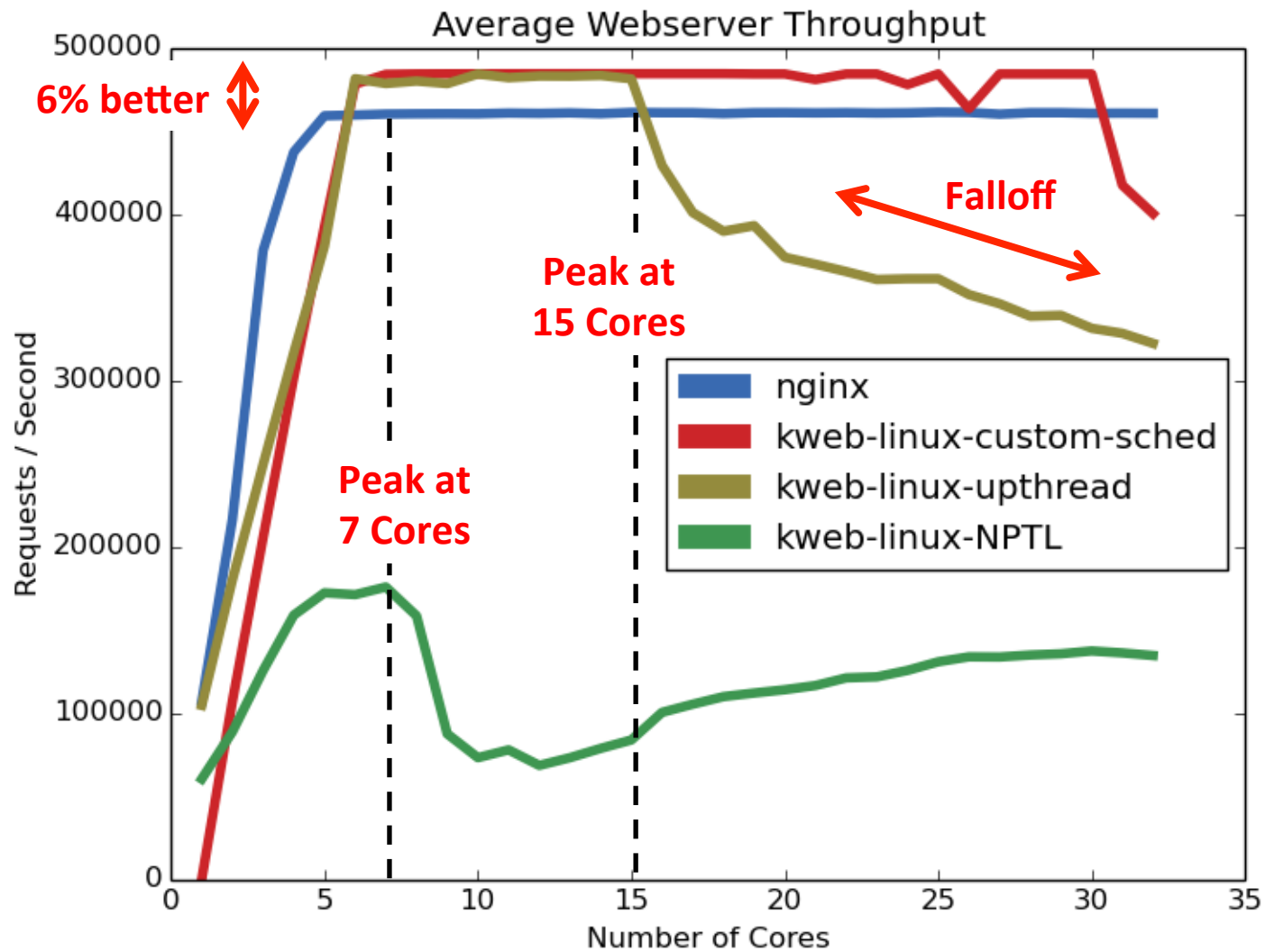
Parlib Experiments



Parlib Experiments



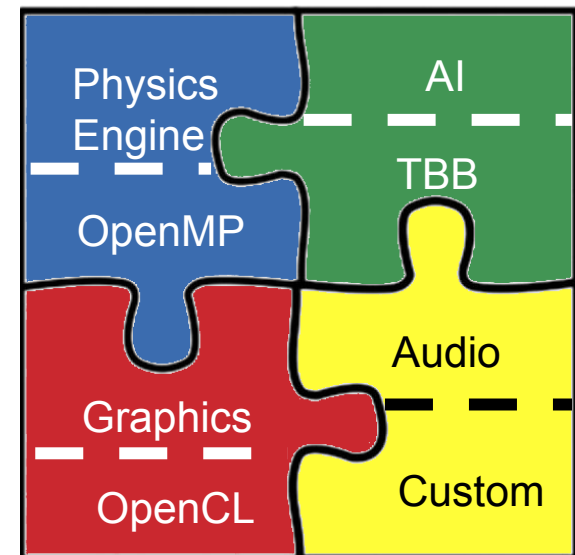
Parlib Experiments



Lithe

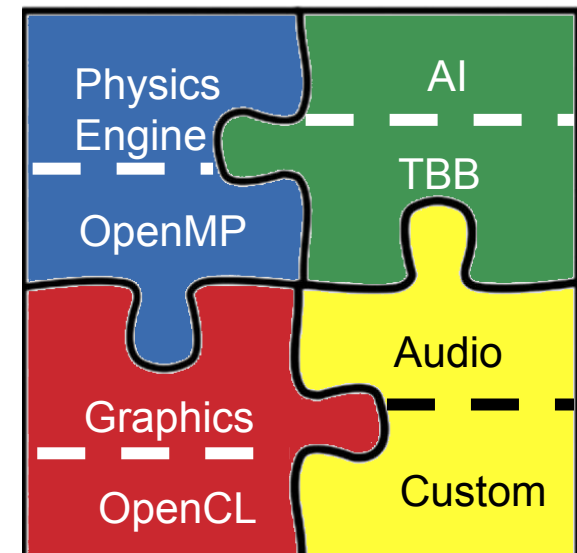
Lithe

- Proposed to solve the problem of **software composability** between parallel libraries



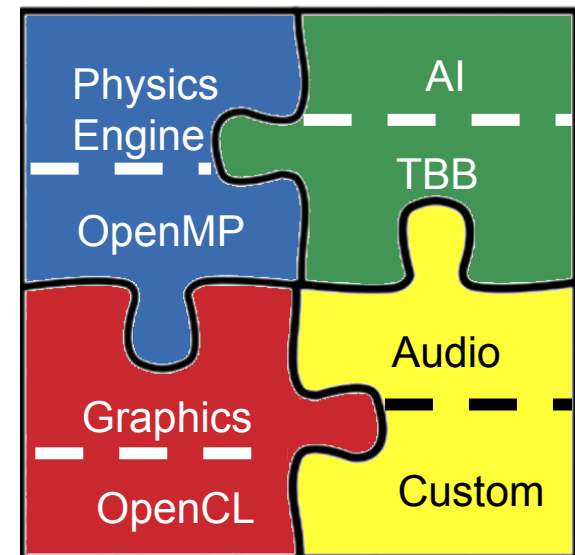
Lithe

- Proposed to solve the problem of **software composability** between parallel libraries
 - Can cause inefficiencies if libraries all launch threads and compete in the OS for scheduling
 - Exacerbated when libraries use barriers for synchronization
 - Lithe → structures sharing of cores



Lithe

- Proposed to solve the problem of **software composability** between parallel libraries
 - Can cause inefficiencies if libraries all launch threads and compete in the OS for scheduling
 - Exacerbated when libraries use barriers for synchronization
 - Lithe → structures sharing of cores
- Originally proposed in the Parlab by Pan et. al. in 2010
- All work presented here is an extension of this previous work



Lithe on Parlib

- Although not originally proposed as such, Lithe fits nicely as an extension to Parlib's user-level scheduling framework

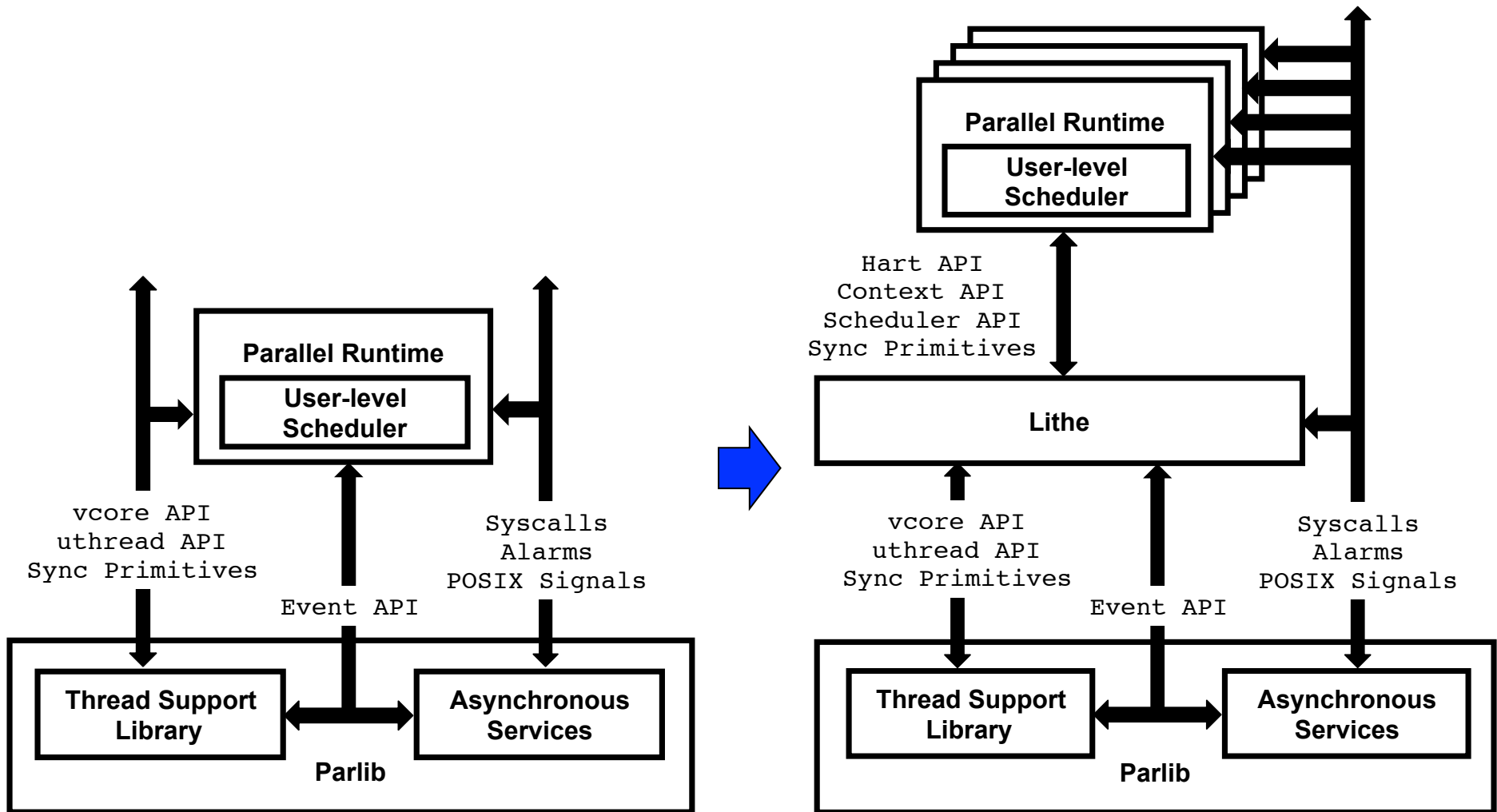
Lithe on Parlib

- Although not originally proposed as such, Lithe fits nicely as an extension to Parlib's user-level scheduling framework
 - Where Parlib only allows for a single user-level scheduler
 - Lithe allows for many, creating hierarchy of schedulers as new ones are added and removed dynamically

Lithe on Parlib

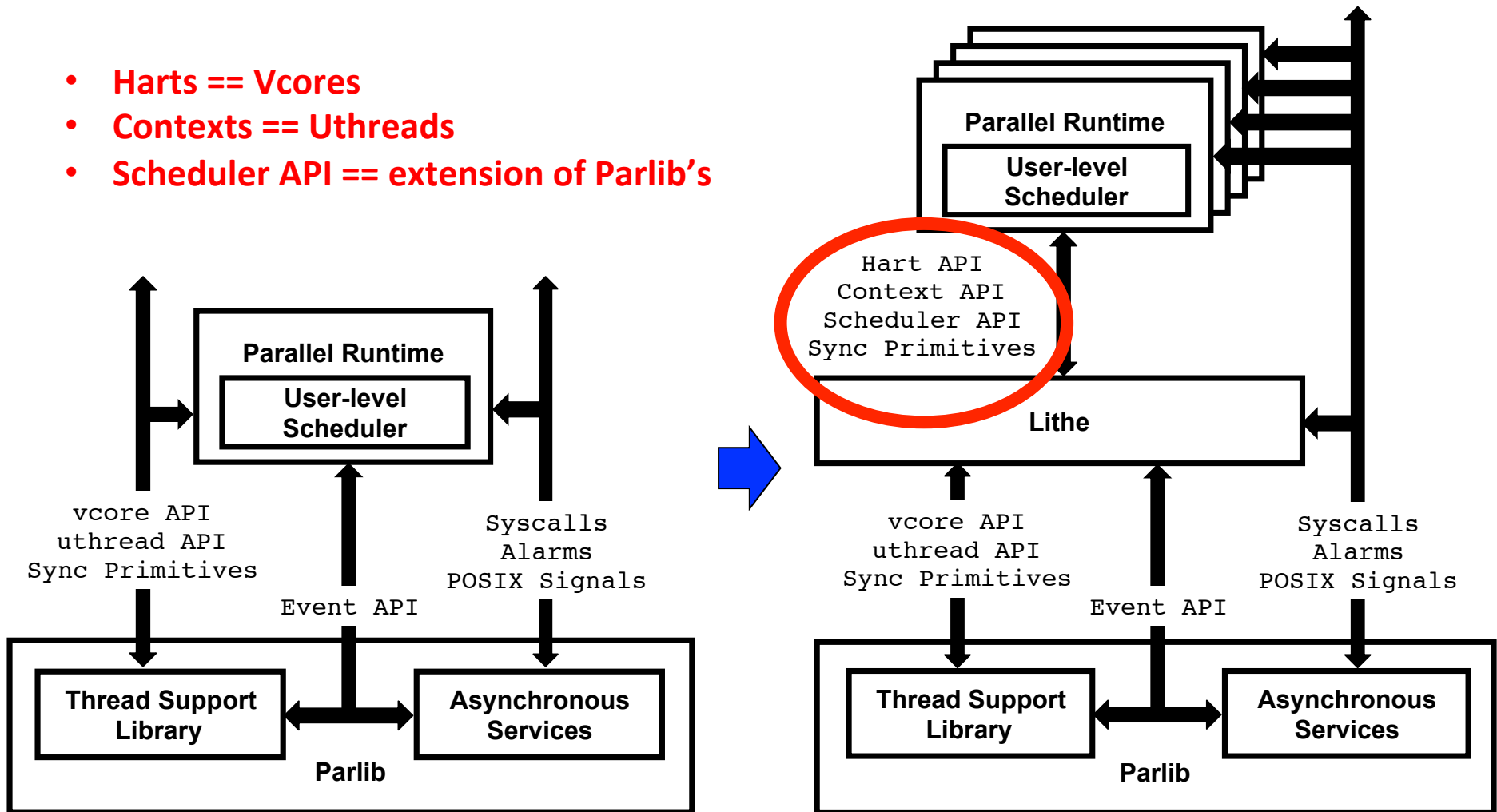
- Although not originally proposed as such, Lithe fits nicely as an extension to Parlib's user-level scheduling framework
 - Where Parlib only allows for a single user-level scheduler
 - Lithe allows for many, creating hierarchy of schedulers as new ones are added and removed dynamically
- Solves problems not addressed in original version
 - No async services in original (especially important for syscalls)
 - Cleaned up interfaces, and better usage semantics in the new version (e.g. vcore context for callbacks)
 - New generic ``fork-join'' scheduler implementation
 - New port of pthreads to Lithe

Lithe Interposing on Parlib

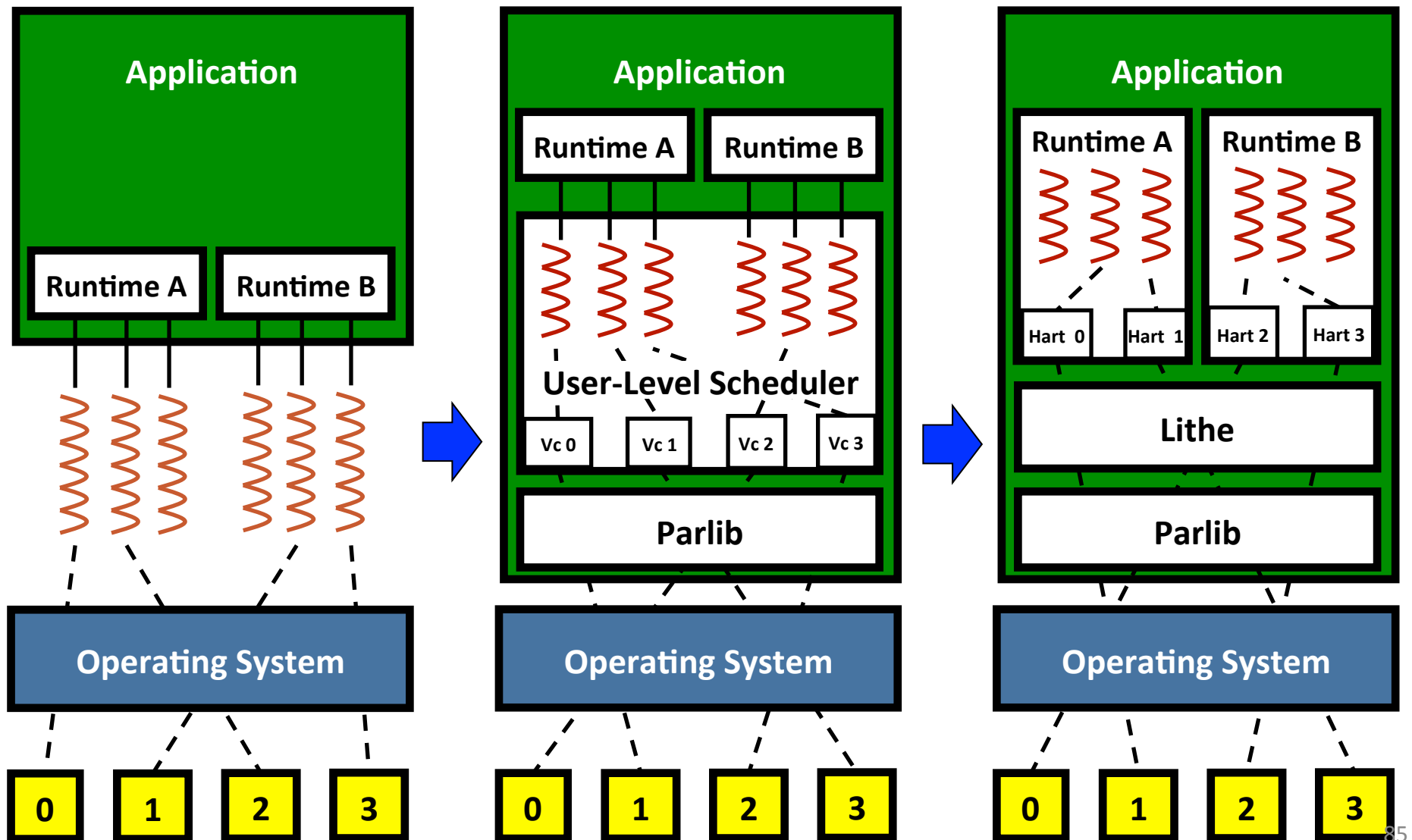


Lithe Interposing on Parlib

- Harts == Vcores
- Contexts == Uthreads
- Scheduler API == extension of Parlib's



Lithe Evolution



Parallel Function Calls

- Like parlib, also based on library calls / callbacks API
- Adds primitives for dynamically adding / removing schedulers as ``parallel function'' calls are made

Parallel Function Calls

- Like parlib, also based on library calls / callbacks API
- Adds primitives for dynamically adding / removing schedulers as ``parallel function`` calls are made

	<pre>void parallel_function() {</pre>
Enter Scheduler →	<pre> lithe_sched_enter(new_sched());</pre>
Spawn / Enqueue Contexts →	<pre> for (int i = 0; i < num_contexts; i++) {</pre>
	<pre> ctx = context_alloc();</pre>
	<pre> add_to_scheduler_queue(sched, ctx);</pre>
	<pre> }</pre>
Request More Cores →	<pre> lithe_hart_request(num_contexts);</pre>
Wait For all Contexts →	<pre> join_on_all_contexts();</pre>
Exit Scheduler →	<pre> lithe_sched_exit();</pre>
	<pre>}</pre>

Lithe API

Hart Management →

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

Scheduler Management →

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 →

Context Management	
Library Call	Callback
lithe_context_init(ctx, entry_func, arg)	-
lithe_context_reinit(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 Ports and Experiments

- Ports exist for
 - OpenMP
 - TBB (Intel's Thread Building Blocks Library)
 - Pthreads

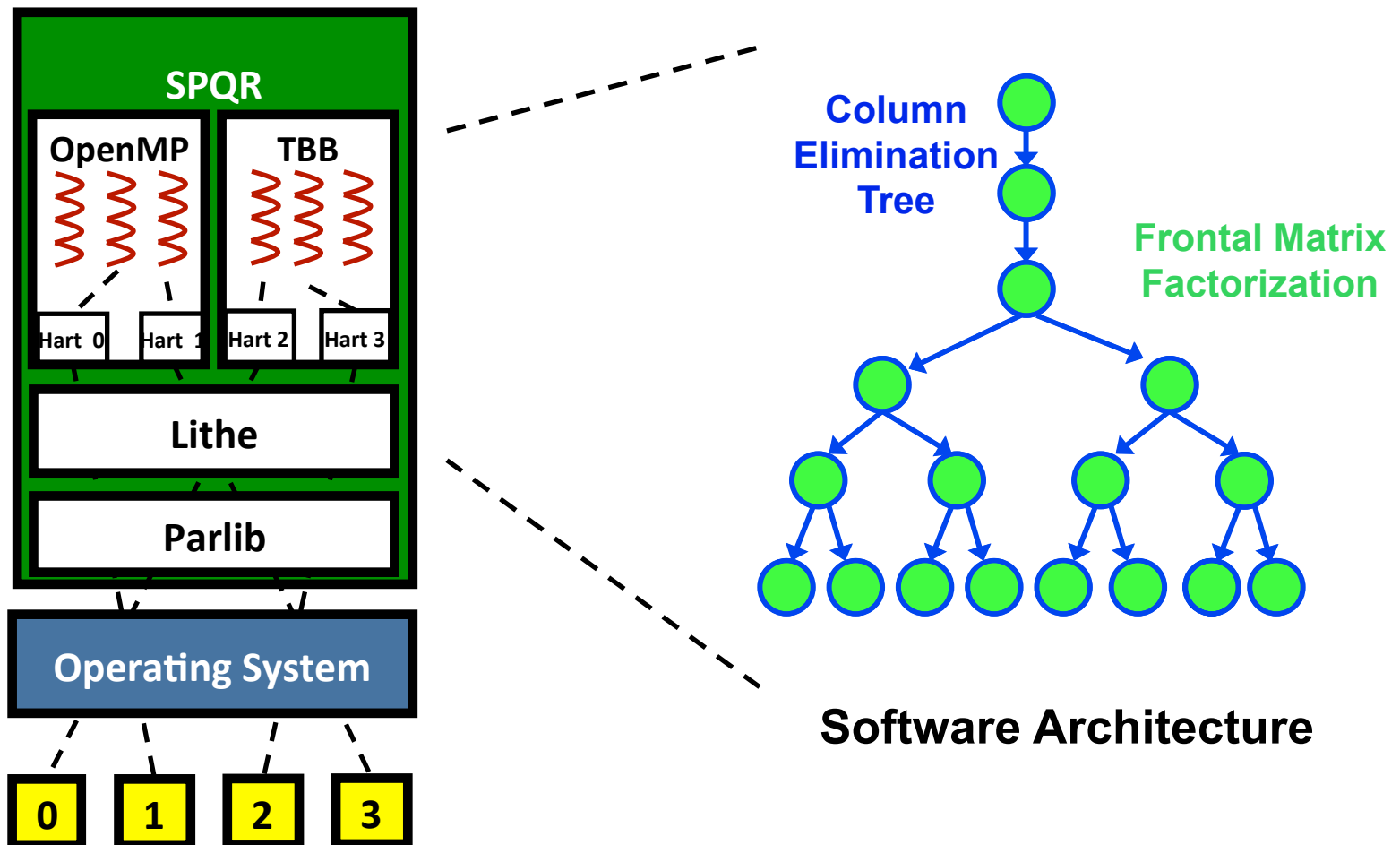
Lithe Ports and Experiments

- Ports exist for
 - OpenMP
 - TBB (Intel's Thread Building Blocks Library)
 - Pthreads
- All ports based on a common ``Fork-Join'' scheduler
- Experiments
 - SPQR Benchmark
 - Kweb File Throughput and Thumbnail Generation

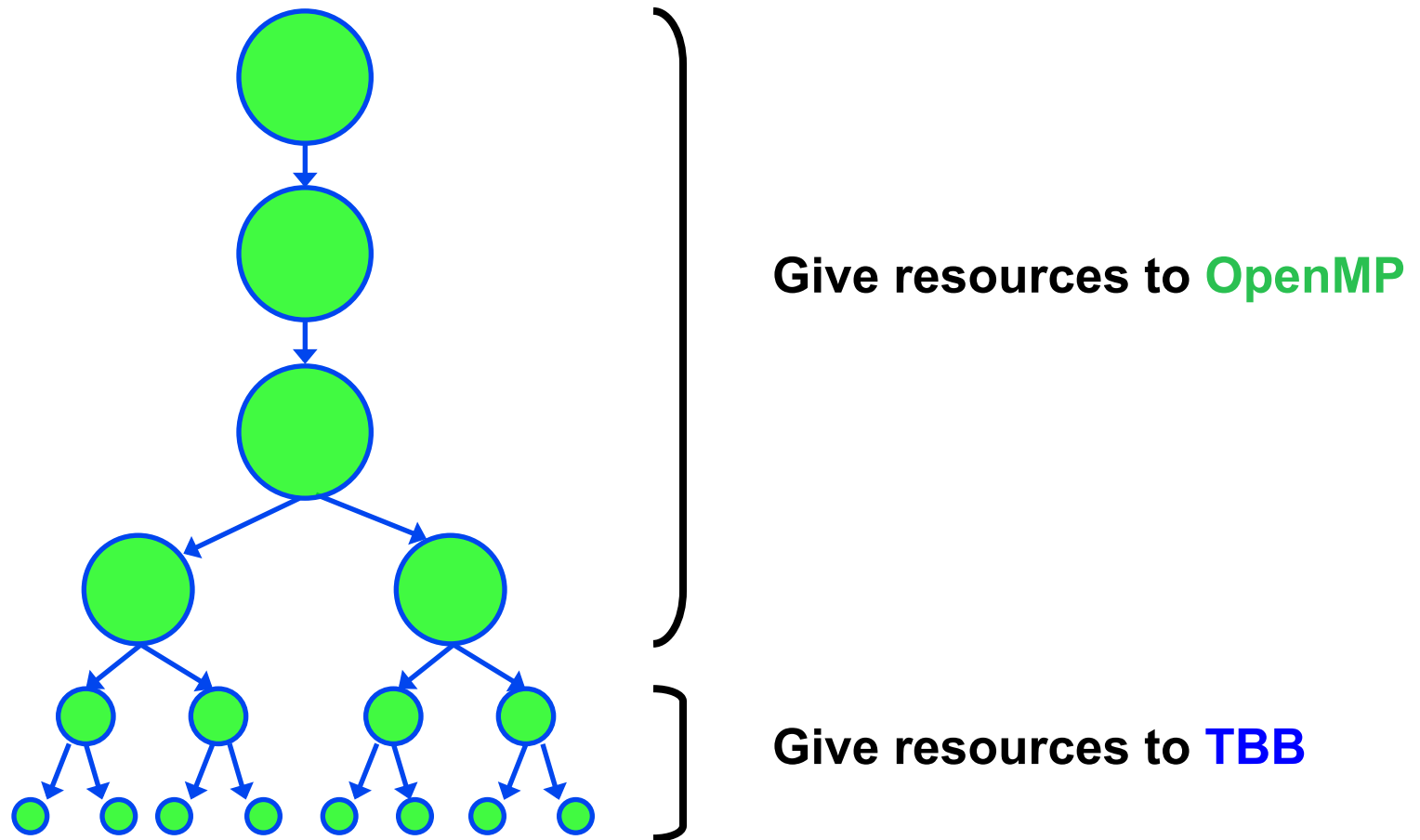
SPQR Benchmark

Sparse QR Factorization

(Tim Davis, Univ of Florida)

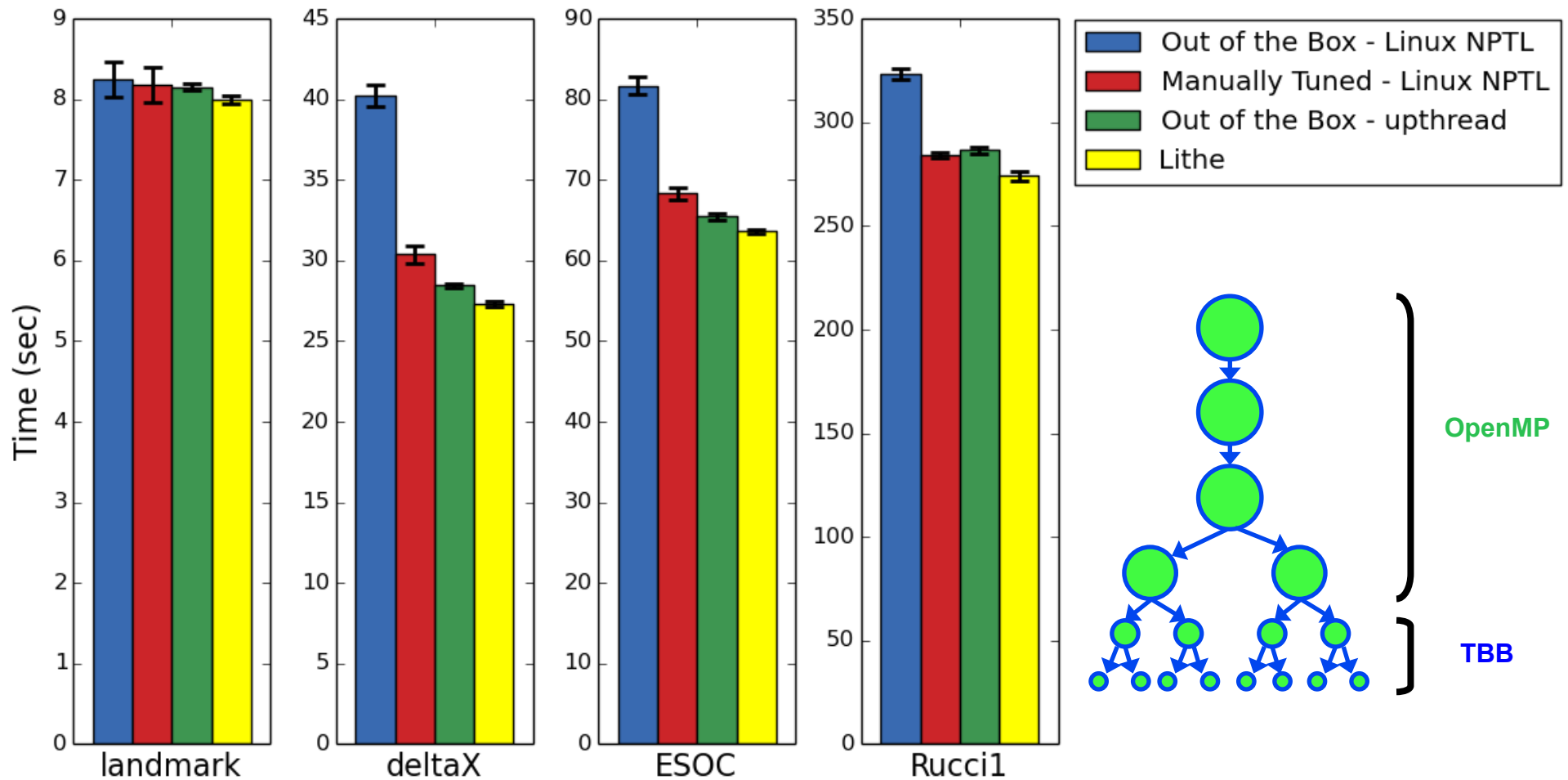


SPQR Benchmark

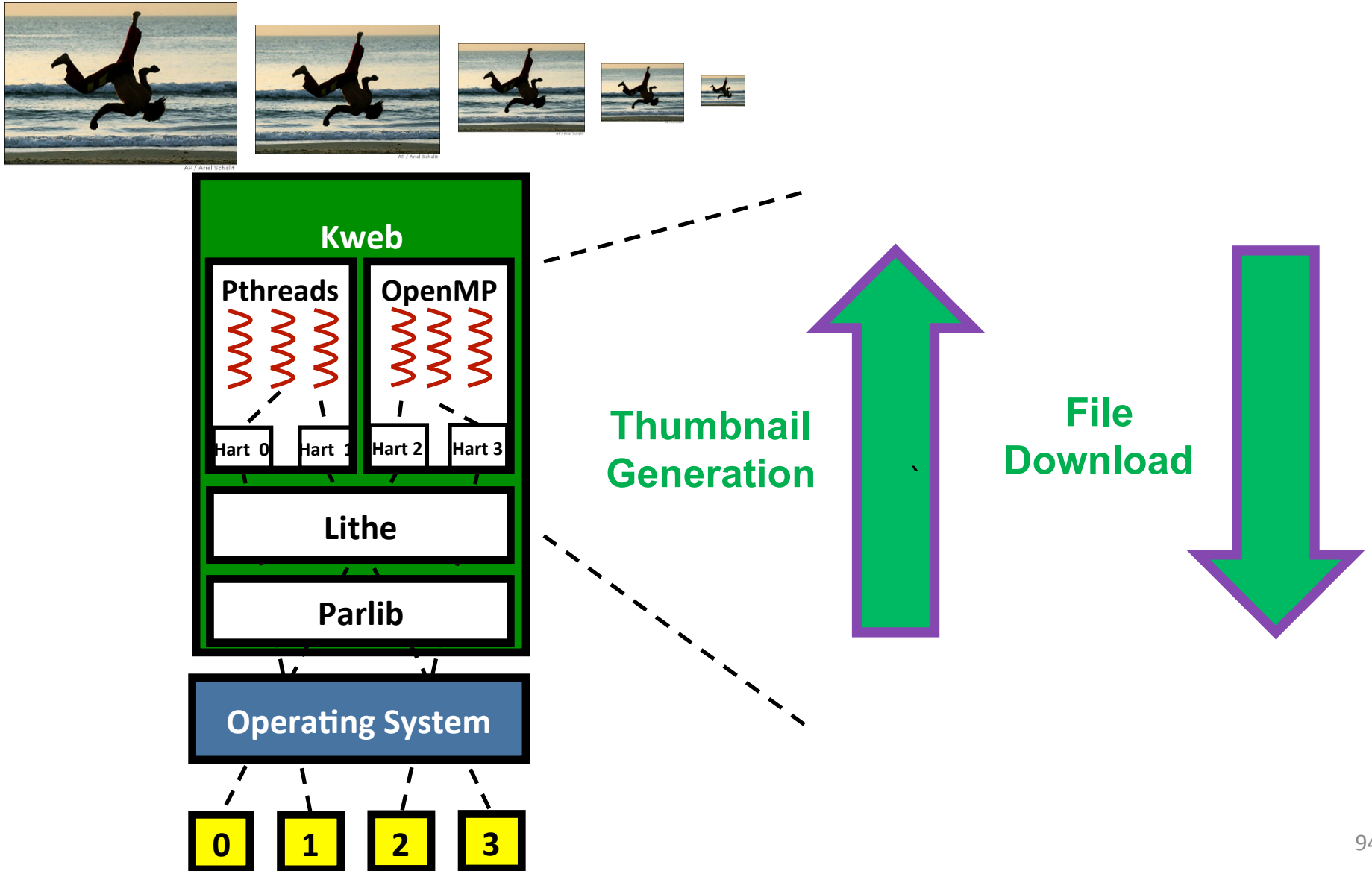


SPQR Benchmark

Performance of SPQR with Lithe

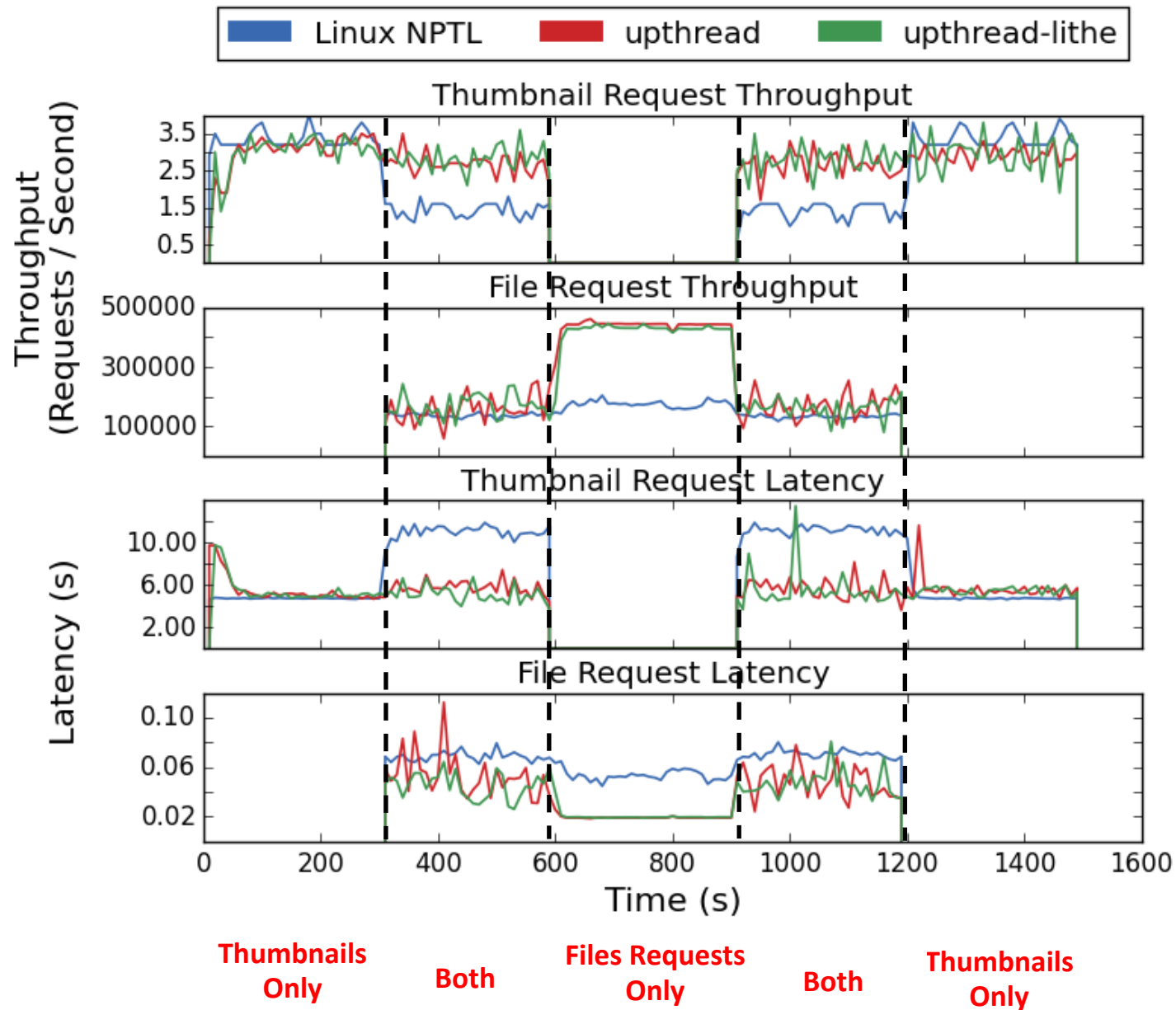


Thumbnail Generation



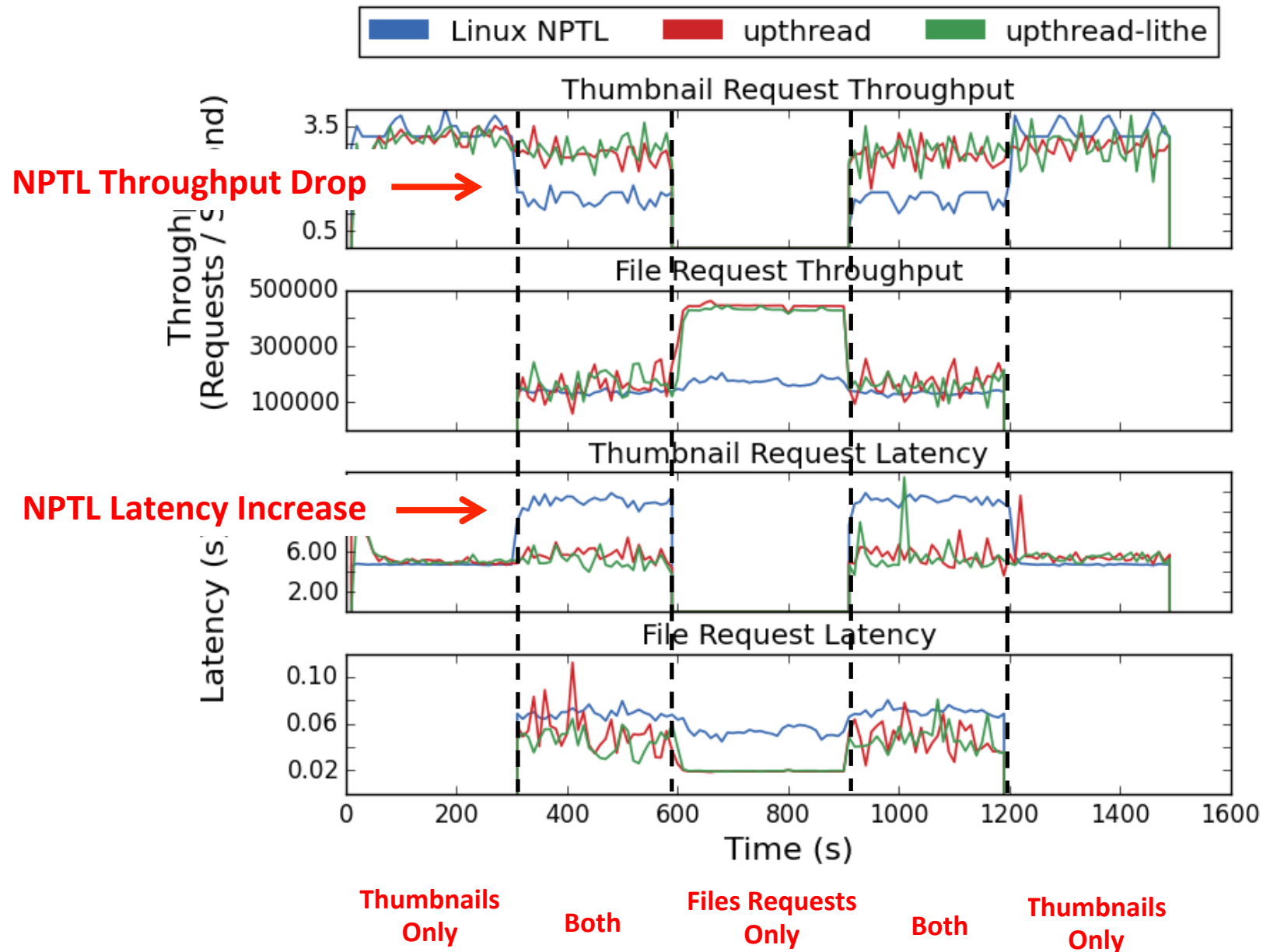
Kweb Throughput and Latency

Mixing File Requests and Thumbnail Generation



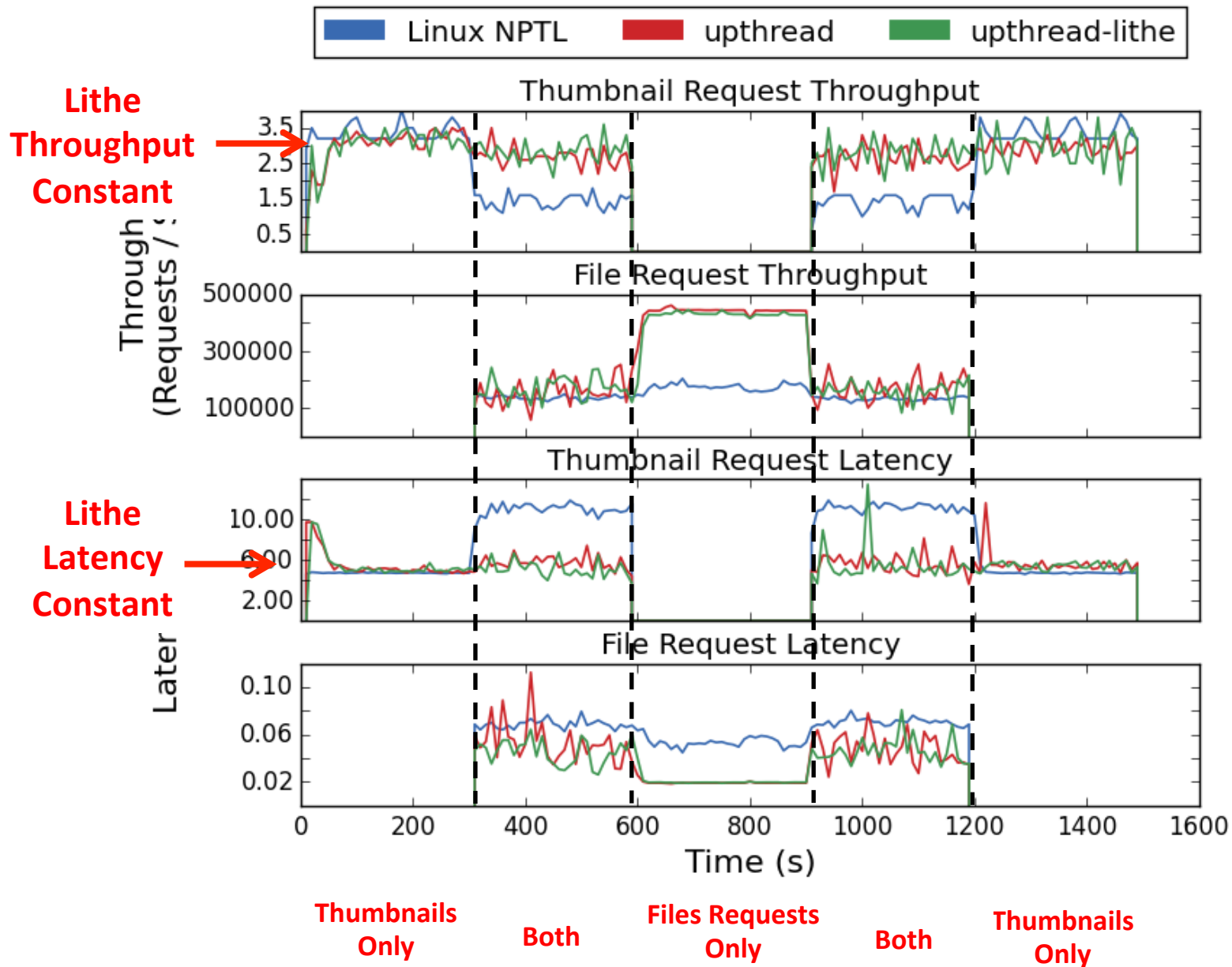
Kweb Throughput and Latency

Mixing File Requests and Thumbnail Generation



Kweb Throughput and Latency

Mixing File Requests and Thumbnail Generation



Go

Go

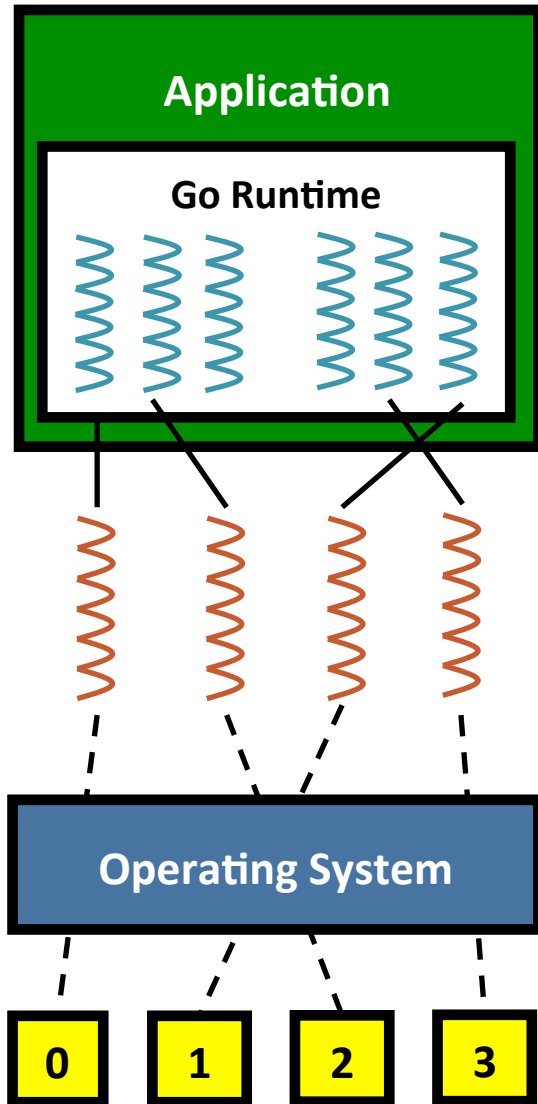
- We have a port of the Go programming language that runs on Akaros
- Porting Go to Akaros provides a fast path to getting real-world, production apps running on Akaros fast

Go

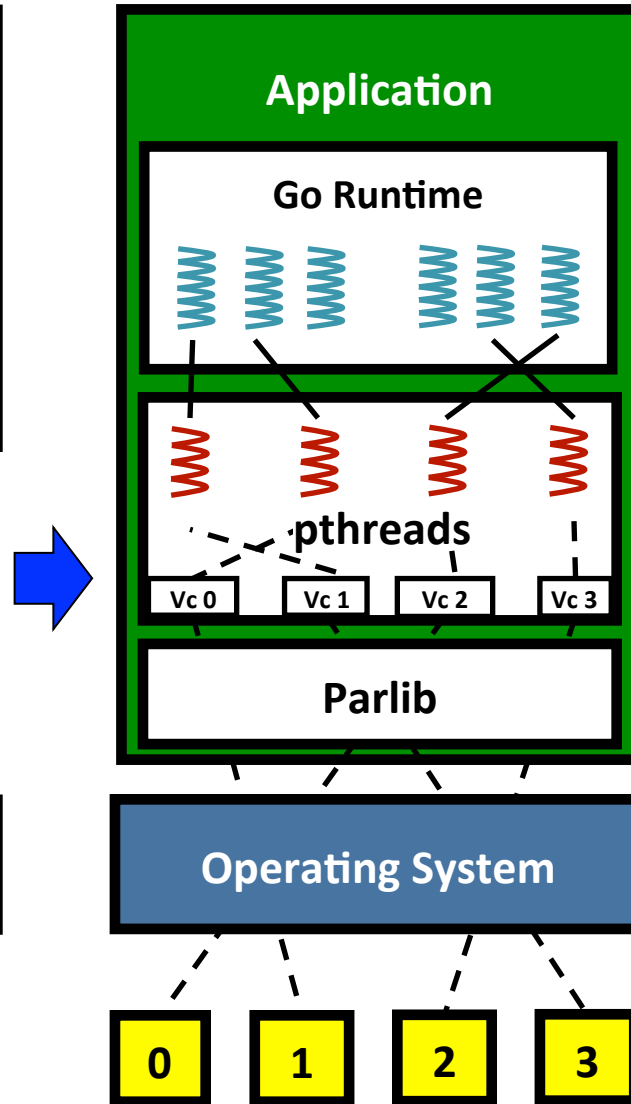
- We have a port of the Go programming language that runs on Akaros
- Porting Go to Akaros provides a fast path to getting real-world, production apps running on Akaros fast
- Leverages CGo
 - Allows us to call arbitrary C code from Go
 - Hook parlib underneath the Go runtime
 - Launch parlib-based pthreads to act as “m-threads”

Go

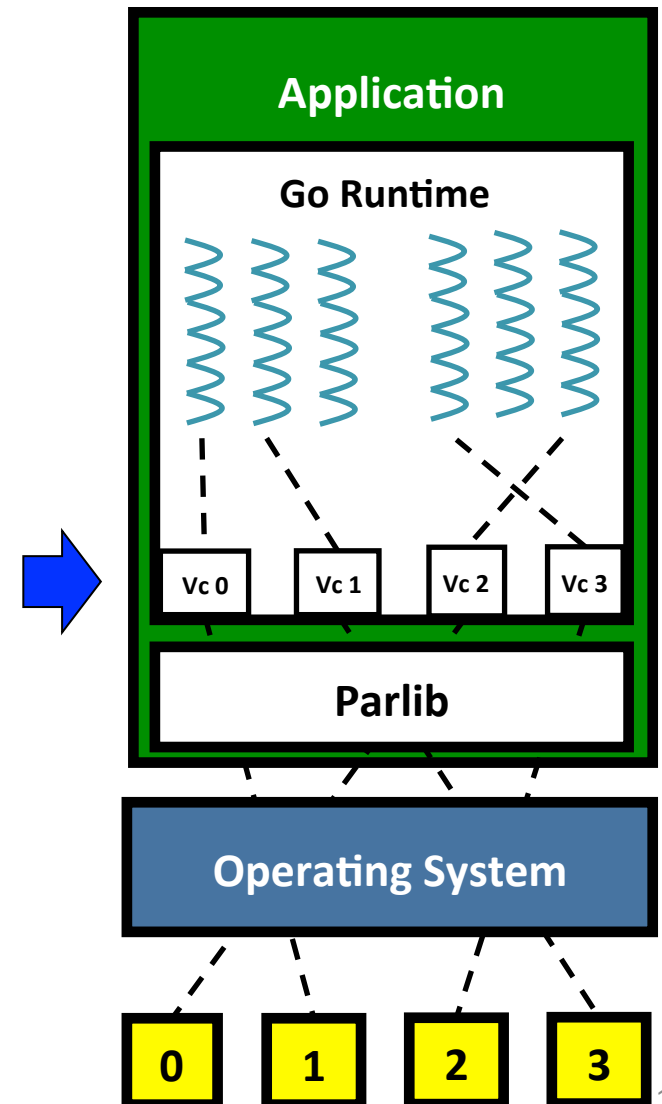
Linux



Current Akaros



Future Akaros



Porting Effort

- Lots of engineering effort to get it working
- Drove many bug fixes / developments in Akaros itself
- Took around 1.5 years to complete
- Passes full Go test suite

go test std	
Total	2412
Passed	2261
Skipped	151
Failed	0

regression tests	
Total	290
Passed	290
Failed	0

- Ported and running on Go 1.3
- Poised for inclusion as supported OS in Go 1.6
- Not yet at full integration (i.e. native parlib support), but getting closer, and good enough for inclusion

Work in Progress

- Go integration with Lithe
 - Works on Akaros out of the box
 - Still need to properly integrate asynchronous syscalls and synchronization primitives on Linux
 - Imagine efficiently calling MKL code from within Go!
- Native parlib support
 - Stop running on parlib-based pthreads library
 - Reimplement Go scheduler itself on top of parlib
 - Many complications with this
 - Read my dissertation to find out more!

Thank You!

- Dissertation Committee
 - Eric Brewer – Chair / Advisor
 - Krste Asanović
 - Brian Carver
 - David Wessel
- 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 many others

Thank You!

- Dissertation Committee
 - Eric Brewer – Chair / Advisor
 - Krste Asanović
 - Brian Carver
 - David Wessel
- 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 many others
- Last but not least, Amy Bryce
(for putting up with me through this whole process)

Conclusion

Conclusion

- Akaros targeted at data center applications
- Managing Cores in these applications is important

Conclusion

- Akaros targeted at data center applications
- Managing Cores in these applications is important
- Using Akaros and its MCP container, applications request cores, not threads from the OS

Conclusion

- Akaros targeted at data center applications
- Managing Cores in these applications is important
- Using Akaros and its MCP container, applications request cores, not threads from the OS
- Cores can be provisioned, allocated, and revoked

Conclusion

- Akaros targeted at data center applications
- Managing Cores in these applications is important
- Using Akaros and its MCP container, applications request cores, not threads from the OS
- Cores can be provisioned, allocated, and revoked
- Parlib provides user-level scheduling framework on top of Akaros's core management interfaces

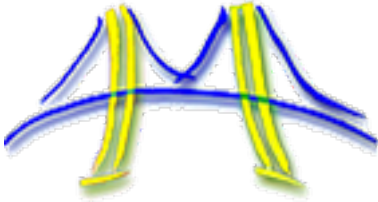
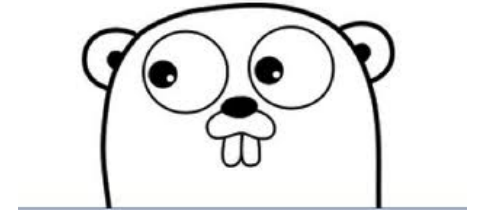
Conclusion

- Akaros targeted at data center applications
- Managing Cores in these applications is important
- Using Akaros and its MCP container, applications request cores, not threads from the OS
- Cores can be provisioned, allocated, and revoked
- Parlib provides user-level scheduling framework on top of Akaros's core management interfaces
- Lithe provides extension to Parlib for composability

Conclusion

- Akaros targeted at data center applications
- Managing Cores in these applications is important
- Using Akaros and its MCP container, applications request cores, not threads from the OS
- Cores can be provisioned, allocated, and revoked
- Parlib provides user-level scheduling framework on top of Akaros's core management interfaces
- Lithe provides extension to Parlib for composability
- Porting Go to Akaros provides a fast path to getting real-world, production apps running on Akaros fast

Questions?



—amplab 



Google