Operating System Support for Parallel Processes

Barret Rhoden

2014-12-08

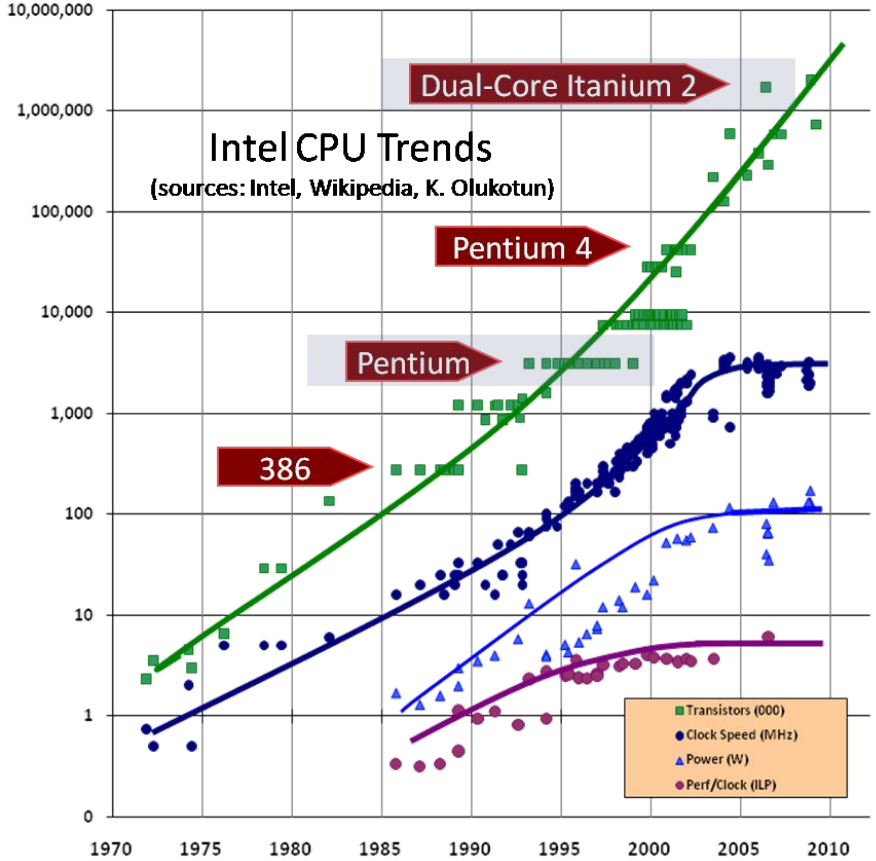
High Level View

- Akaros: research operating system
  - Large-scale SMP / many-core architectures
  - Single nodes, typically in a data center
  - Mix of high-priority and low-priority applications
- Support for high-performance, parallel apps
  - Transparent access to physical resources
  - Spatial allocation of cores to processes
  - Applications schedule and manage their threads
  - Performance isolation / minimize interference

Agenda

- Motivation
- Akaros background
- A brief history of threading and parallelism
- Parallel processes in Akaros (the MCP)
- Event delivery and preemption
- Evaluation

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
- Parallelism is not new; can learn from the past
- The largest machines will be in data centers

Data Centers: Low Latency and Batch Workloads

- Live service jobs (low latency, high priority):
  - Minimize latency, especially tail latency
  - Predictable, efficient performance
  - Guaranteed resources for peak workload
- Batch jobs:
  - Low priority
  - Fill in the peak-to-average gap
  - No guarantee for resources

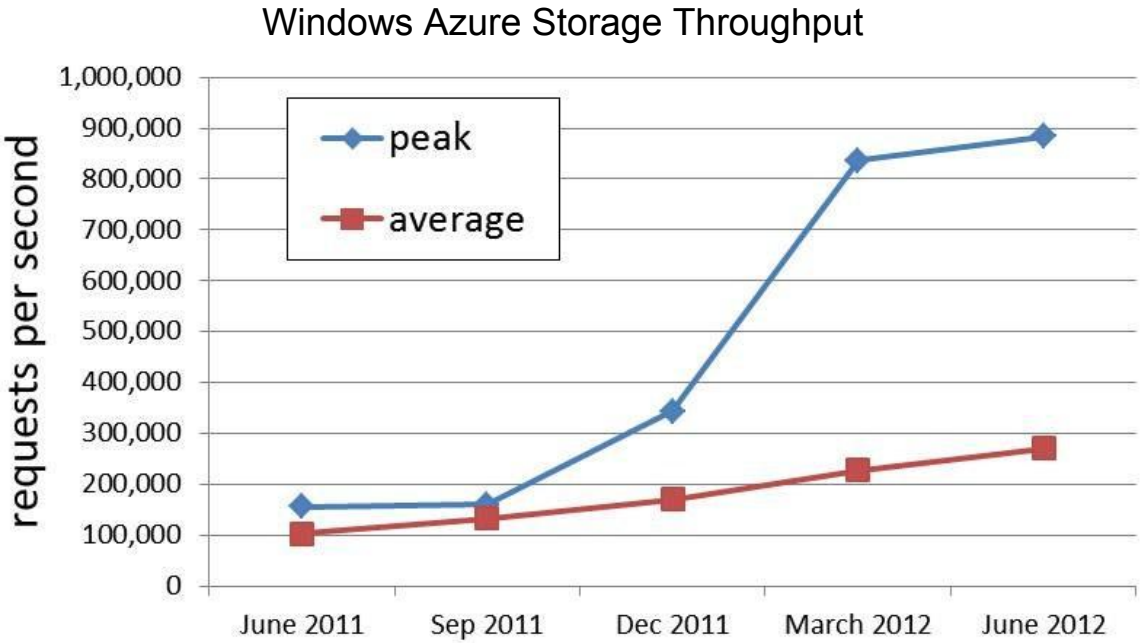
Data Center Provisioning

"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

http://news.cnet.com/TurboTax-e-filing-woes-draw-customer-ire/2100-1038_3-6177341.html

Peak-to-Average Gap



http://www.ditii.com/2012/07/19/windows-azure-storage-hits-4-trillion-objects-mark-process-270k-rps-increases-windows-azure-web-sites-reserve-instances/

Parallel OSs Should Provide:

- High, predictable performance
  - Minimal interference
  - Raw access to physical resources
- Adaptability to dynamic workloads
  - Exploit the peak-to-average gap
- Abstractions and interfaces for utilizing parallel processors

Akaros: Philosophy of Transparency

- Expose info about the underlying system
- Provide interfaces to control guaranteed, allocated resources
- Virtual resources for naming, not for deception
  - Processes use virtual memory and paging
  - Can view their page tables
  - Physical memory is pinned - no swapping

Akaros's Features

- Spatially allocated, dedicated cores to processes
- User-level thread schedulers running at high frequency (or any frequency)
- Low frequency resource reallocation, typically driven by cluster managers
- Control over IRQ routing: no unexpected interrupts on dedicated cores

Provisioning vs. Allocation

Provisioning:

- Guaranteed future access to resources
- Used for low-latency services
  - Amount based on peak load
  - Amount used at any time may be less

Allocation:

- The actual granting of the resource (dynamic)
- When provisioned, uninterruptible, irrevocable
- Without, can be revoked at any time
  - Used for batch jobs

Akaros Programming Environment

- GCC toolchain, x86 and RISCV, 32/64 bit
- Glibc and the Go runtime ported
- Some POSIX support (basic pthread apps)
- Plan 9 namespaces and network stack
- Custom extensions for Akaros (parlib)
- Barebones system

Classic Threading Models - 1:1

1:1 - One kernel thread/task per user thread

- Heavy-weight threads, visible to the kernel
- Threads shared an address space
- Scheduling decisions made by kernel
- Mesa/Cedar, Topaz, pthreads on Linux.

Classic Threading Models - M:1

M:1 - Many user threads per kernel thread

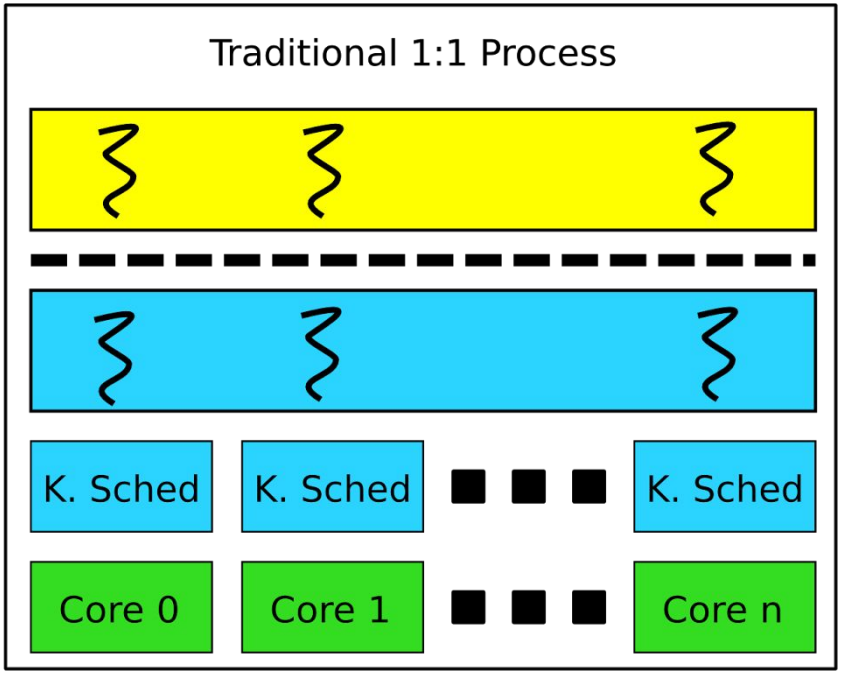
- Lightweight threads, fast context switches
- Scheduling decisions made by the application
- If one thread blocks, the entire process stalls
- Cannot exploit multiple processors
- Coroutines, Java’s Green Threads, Capriccio (which avoided blocking I/O)

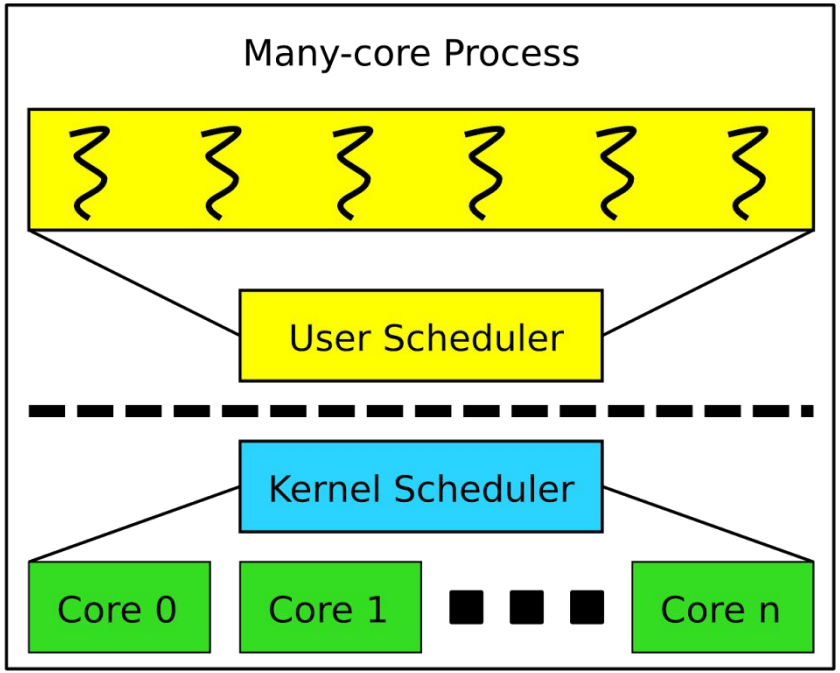
Classic Threading Models - M:N

M:N - Many user threads on many kernel threads

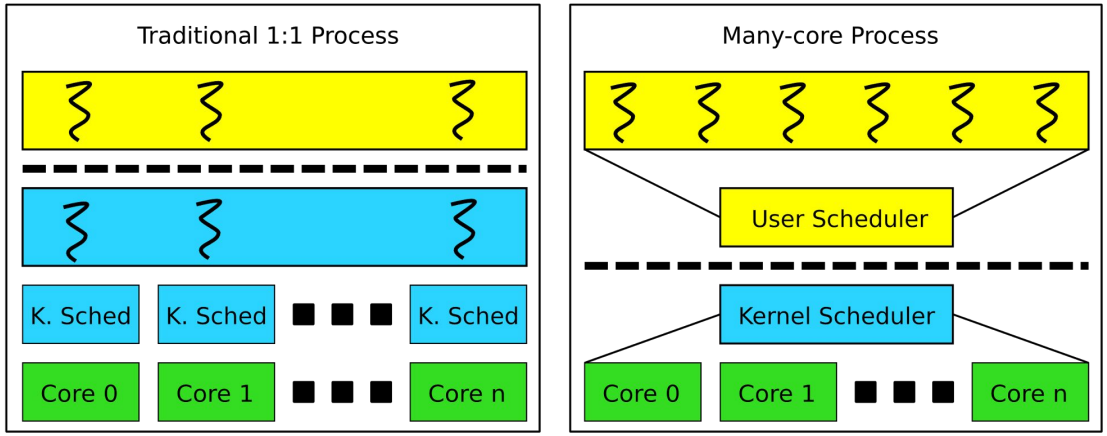
- User threads are scheduled and managed by the application on a set of kernel threads
- If a kernel thread blocks, the process can continue on other kernel threads
- The number of kernel threads may vary over time
- Solaris's Light Weight Processes, Scheduler Activations, Psyche, Go

Many-Core Process (MCP)





MCP



- Treat parallel processes as a single entity
  - Gang scheduled, no kernel thread per “pthread”/core
  - Single address space
- The process is aware of its state
  - Number of cores, which ones are running, etc
- Allows 2-Level scheduling (2LS), spinlocks, etc

Cores != Threads

- Cores are for parallelism
- Threads are for concurrency (blocking I/O)
- Kernel threads are **not** part of the interface
- Blocking (syscall, page fault) doesn’t mean the process loses the core
- Notified of and can handle changing numbers of cores
- Process has full control over upcalls/events

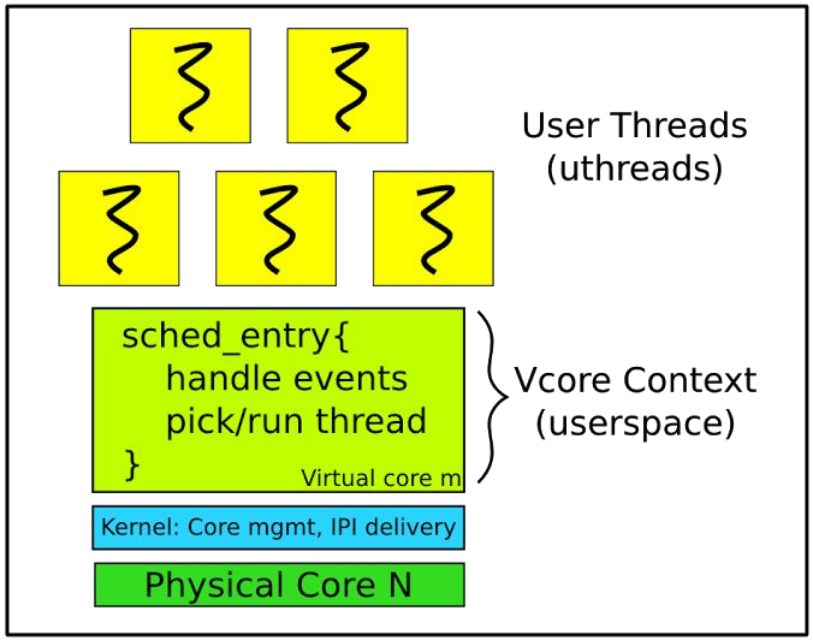
Life for an MCP

- No unexpected interrupts
- Long time quanta
- Shared memory pages with the kernel
  - Procinfo (read-only), procdata (read-write)
- Have a set of virtual cores (vcores)
  - Pinned to physical cores when running
  - Can see the vcoremap
  - Each vcore has an “interrupt handling” context
- Schedule your own threads

Kernel Scheduling

- Different types of cores (can be dynamic)
- MCPs run on Coarse-Grained (CG) cores
  - No timer IRQs or per-core scheduler
  - Will run in kernel mode for IPIs for start-up/tear-down
- SCPs (single core processes), daemons, etc, run on Low-Latency (LL) cores
  - Management tasks, high frequency timer tick
  - Scheduler runs on an LL core (Core 0)

Vcore Context



- Analogous to interrupt context in OSes
- Handles events and schedules threads
- Has its own stack and per-vcore storage
- Event driven
- IPIs / software IRQs disabled

Asynchronous Syscall Interface

- The struct syscall is the contract with the kernel
- The kernel may use threads and block internally, but userspace doesn’t know or care
- User threads (uthreads) that issued syscalls that blocked in the kernel hand off to the 2LS
- Userspace / 2LS can poll or request an event
- Can process syscalls on remote cores

What about Page Faults?

- Kernel will handle any soft faults (no blocking)
- Unhandled faults are reflected to userspace
- Faults in “vcore context” kill the process
- Pin critical code/data
- Uthreads that PF on file-backed mmaps are serviced by the 2LS via a syscall

Event Delivery

- Events are decoupled from vcore context
  - Unlike Scheduler Activations and Psyche
  - Much more flexible
- Important to never miss messages
  - User space manages its cores, so it is partly responsible
  - Analogous to the “wakeup waiter” problem
- User and kernel work together in shared memory
  - Vcore context code will not exit when notif_pending is set

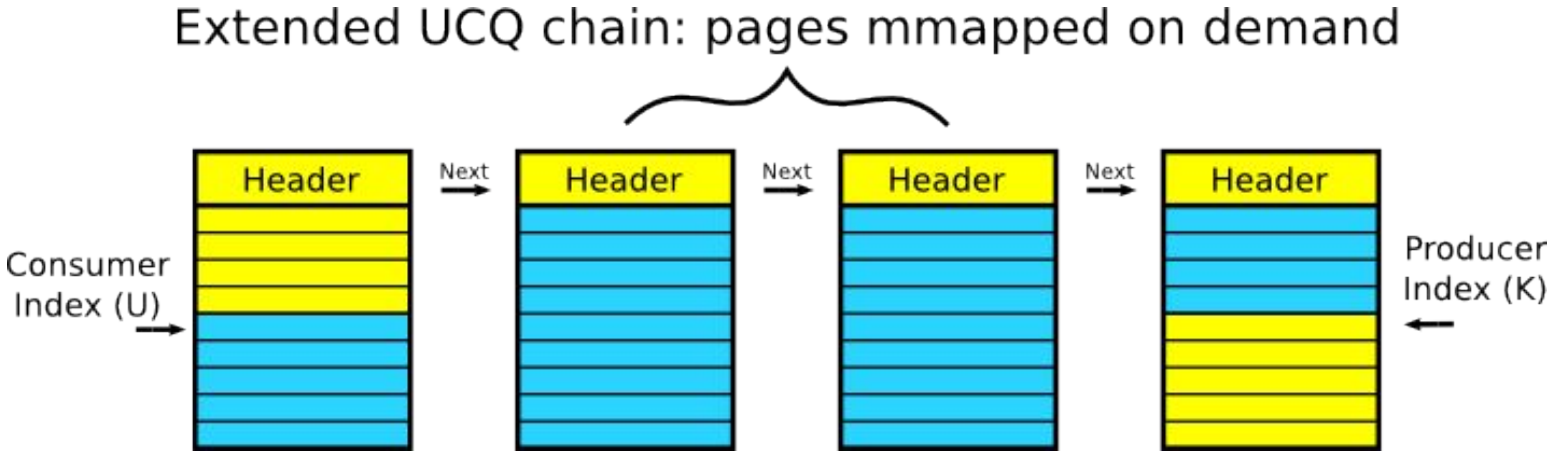
Event Queues

- Event delivery interface: struct event_queue
- Contains storage for events with payloads
- Can be used when **exactly** one message must be sent (e.g. syscall completion)
- Various delivery options:
  - IPI a particular vcore
  - Guaranteed delivery
- Shared memory; pass a pointer to the kernel

Unbounded Concurrent Queues (UCQ)

- Data structure for storing event payloads
- Multiple producer, multiple consumer
- Producer (kernel) does not trust consumer
- Linked-list of mmapped pages

UCQs



- Kernel mmaps extra pages on demand
  - Limited by the RAM available to the process
  - Maintains a swap page to avoid excessive mmaps
  - Consumer munmaps extra pages

Dealing with Preempted Cores

- The kernel will revoke cores from low-priority processes to satisfy a provisioned request
- Provisioned resources will not be revoked
- Any delay in preempting backfilled resources hurts the rightful owner of the resources
- When cores are revoked, the code halted could be in a critical section (holding locks)

How Older Systems Coped

- Psyche: “two-minute warning” message
  - User should clean up and yield
  - If not, preempt, and hope that “vcore” will run again
  - “vcores” are always pinned to specific physical cores
- Scheduler Activations: send an activation
  - Preempt, then halt **another** core to send a message
  - Userspace must choose which halted core to run
  - Tried to detect critical sections; not foolproof

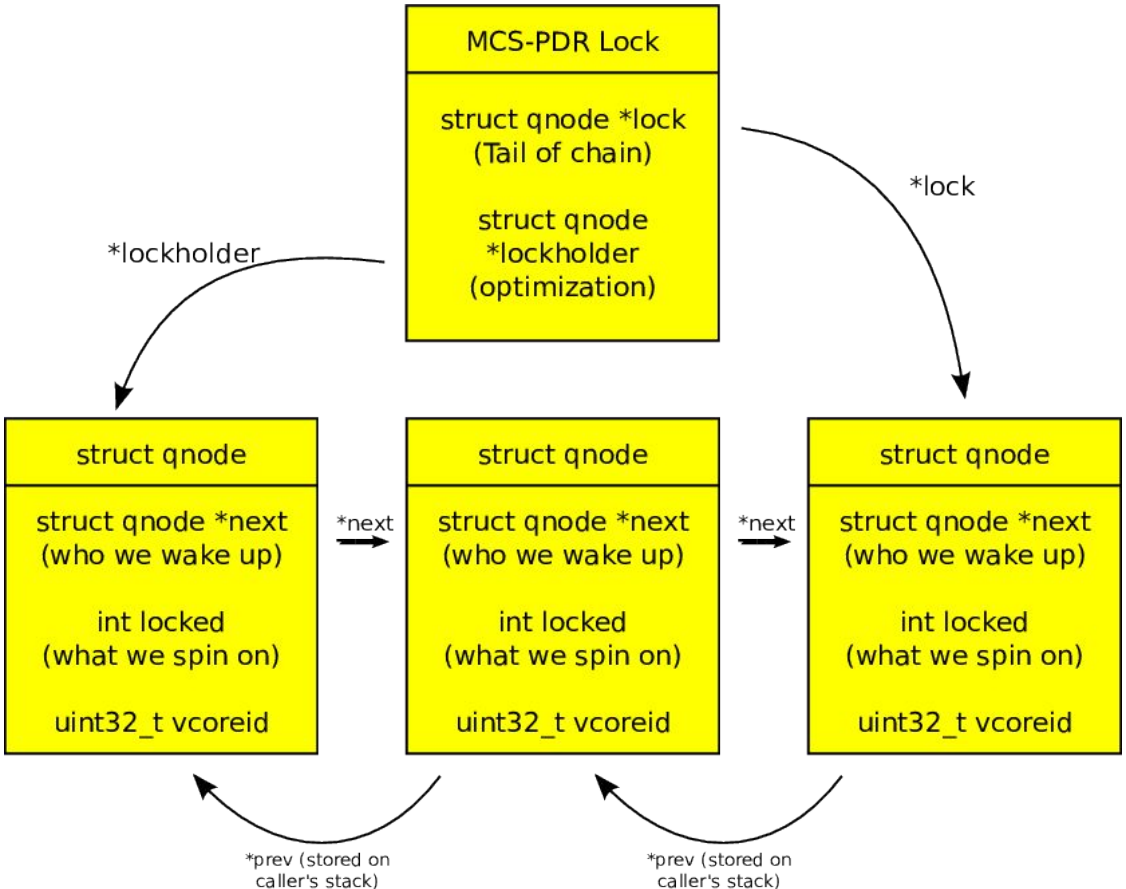
Akaros Preemption Handling

- Akaros's tools:
  - Processes can see which vcores are online
  - Vcores can context switch to preempted vcores
- Two-part solution to preemption:
  - Preemption Detection and Recovery locks (PDR)
  - Spam preemption events, idempotent handler

PDR Locks

- Threads that spin on locks ensure the lock-holder is not preempted
- Atomically “sign up” and publish vcoreid
- Simple for basic spinlocks:
  - Compare-and-swap, -1 means unlocked, vcoreid o/w
  - Spinners change to the lock-holder if it is preempted
- More complicated for MCS (queue) locks

MCS-PDR Locks



Preemption Event Handler

- Key insight: clean up, like with a “two-minute warning”, but use time in the future
  - The **handling** vcore cleans itself up, then changes to the preempted vcore
  - Can always send more messages in corner cases
- More advanced: the handler can steal a uthread from the other vcore’s shared memory
- Must check for messages in a vcore’s queue

Evaluation / Microbenchmarks

- Intel Xeon E5-2670, 2.6GHz
- Sandy Bridge
- 16 Cores, 32 hyperthreads
- 256 GB RAM
- Linux 3.11, Ubuntu
- This machine is called “c89”

Thread Context Switch Latency

- Thread context-switch latency
- Pthread program:

```
pthread_thread() {
    for num_loops
        pthread_yield();
}
```

Thread Context Switch Latency

Values in nsec	Linux Pthreads with TLS	Linux Uthreads with TLS	Linux Uthreads without TLS	Akaros Uthreads with TLS	Akaros Uthreads without TLS
1 Thread	254	474	251	340	174
2 Threads	465	477	251	340	172
100 Threads	660	515	268	366	194
1000 Threads	812	583	291	408	221

- Thread local storage (TLS) hurts
- Uthread (2LS) scheduler is slow

Akaros User Context Switch Latency

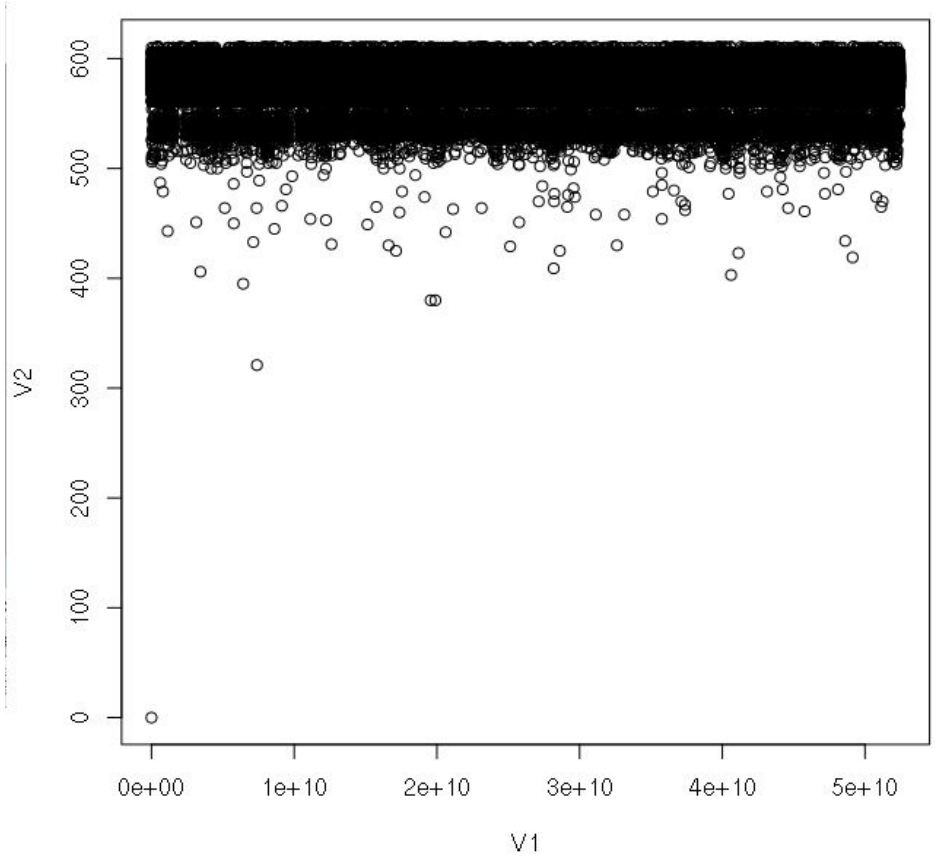
Times in nsec	With TLS	No TLS	No Locking in Scheduler	No Locking, No asserts	Switch_to (bypass 2LS decision)
2 threads	340	172	95	88	55
100 threads	366	194	113	105	

- TLS, dumb scheduler, untuned
- Akaros's user threading library (uthread.c) allows individual threads to have TLS or not
- All context switches drop into vcore context

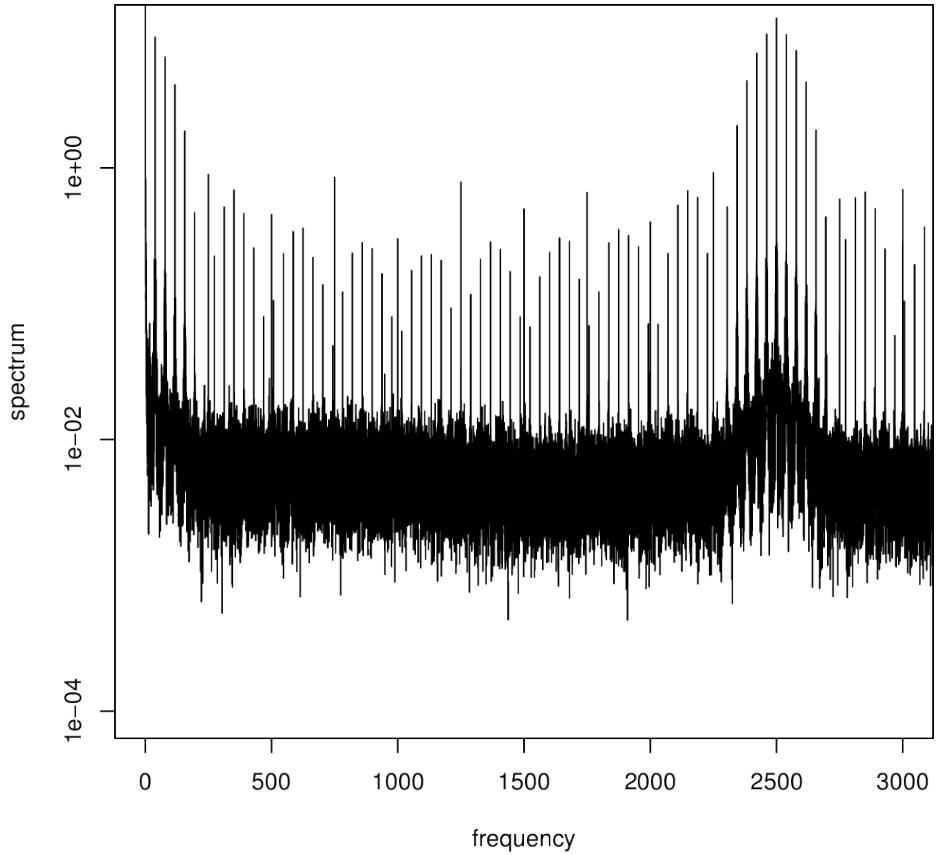
Isolation, Interference, and Noise

- Fixed Time Quantum benchmark
  - Sottile and Minnich, *Analysis of Microbenchmarks for Performance Tuning of Clusters*, Cluster 2004
  - github.com/rminnich/ftq
- Perform work in a constant time interval
  - FTQ parameter: *frequency* of samples (e.g. 10KHz)
- FFT the result to detect periodic interference

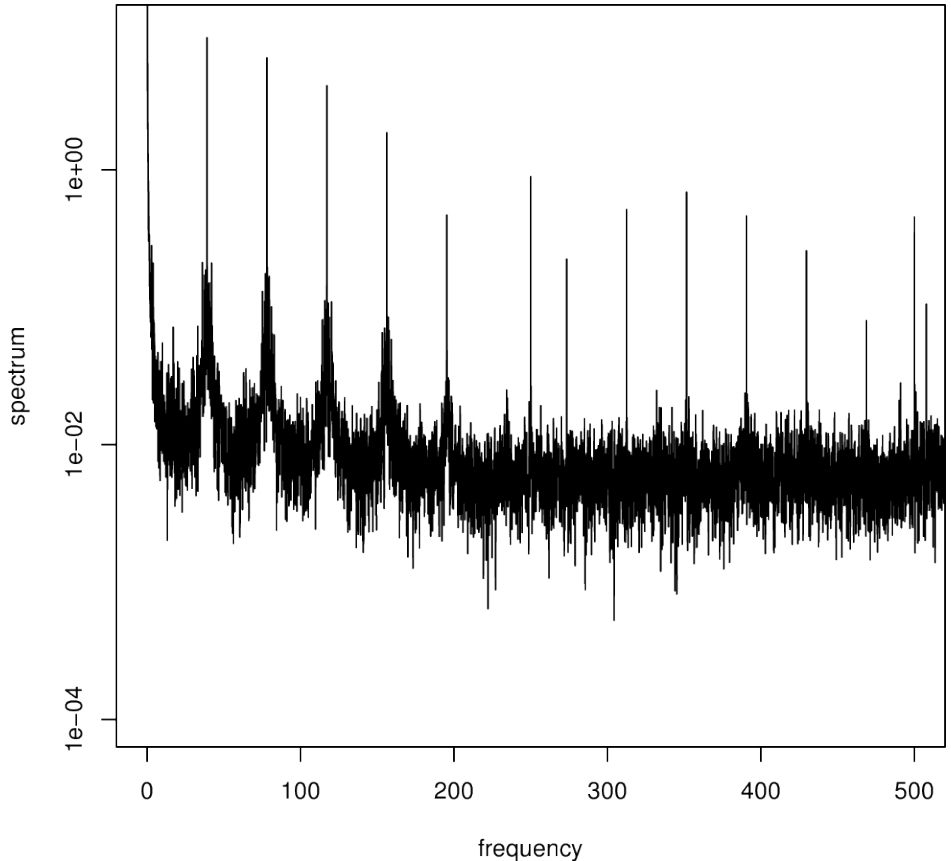
Raw Data: Linux, Single Core



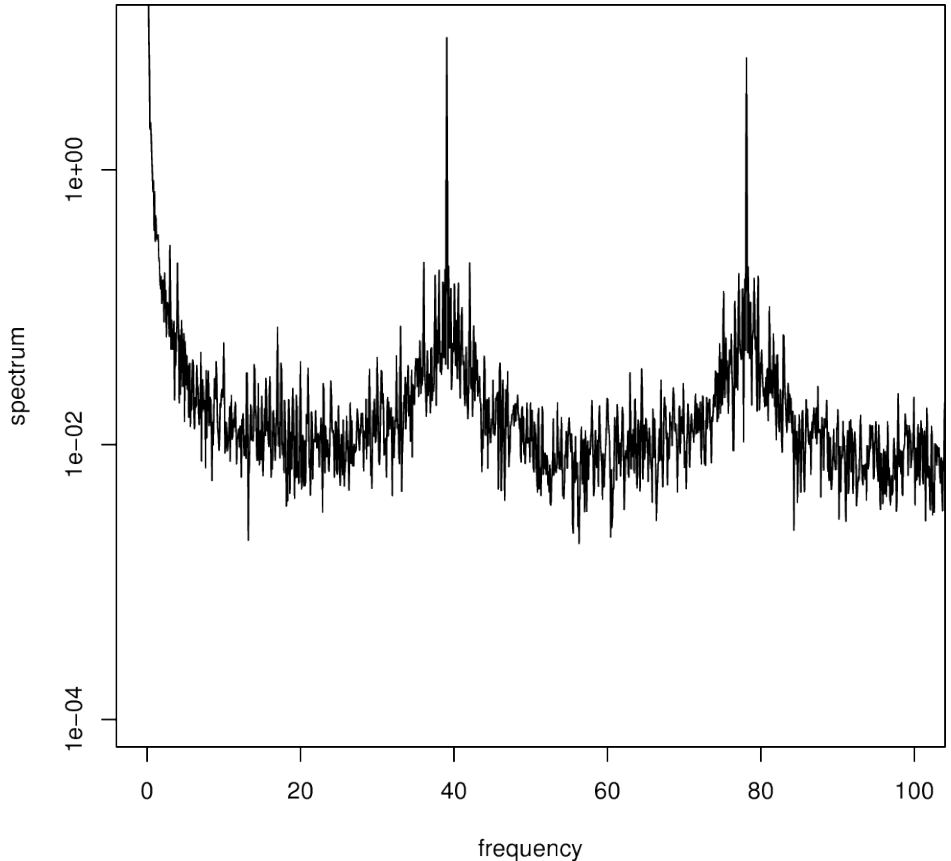
Linux, Single Core (31), 3000 Hz



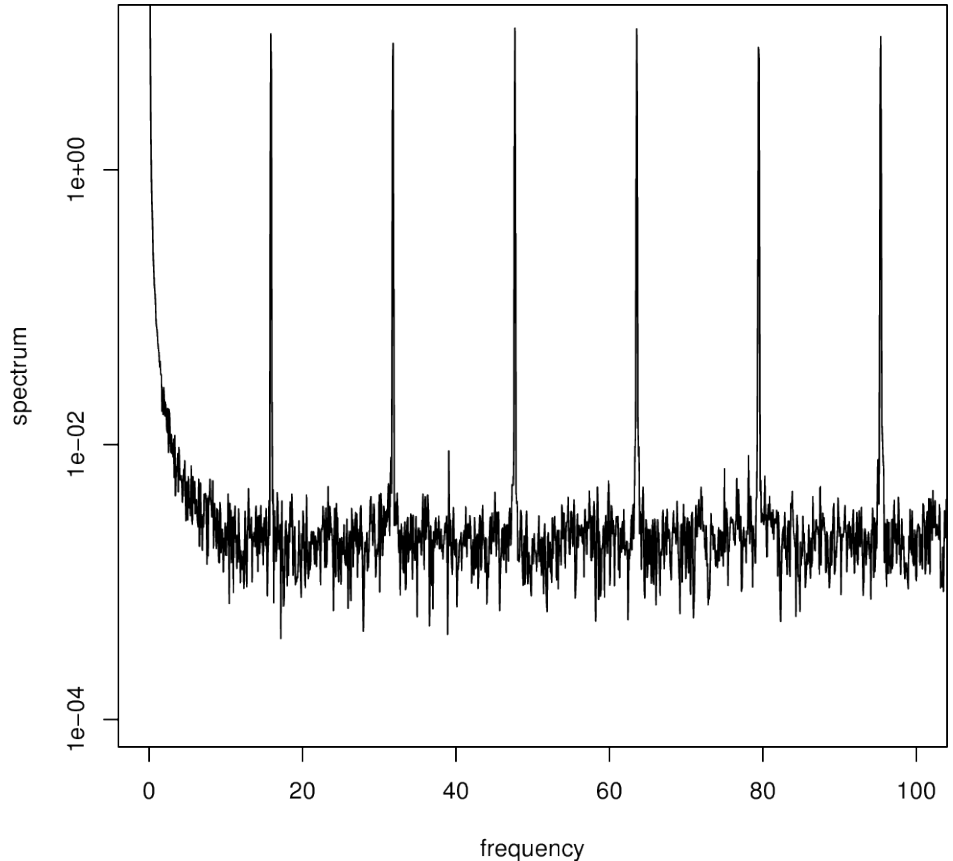
Linux, Single Core (31), 500 Hz



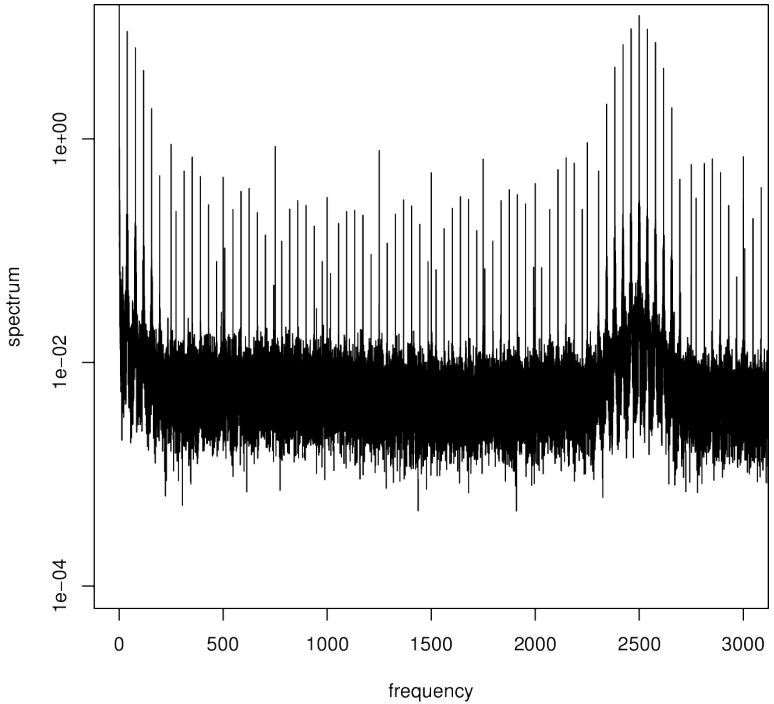
Linux, Single Core (31), 100 Hz



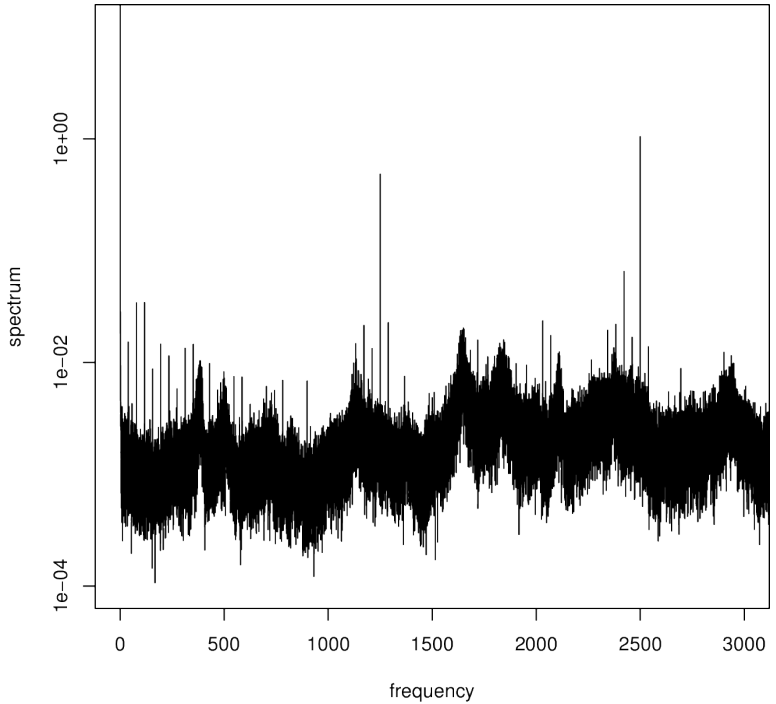
Old Akaros, Single Core, 100 Hz



FFTs: c89, Core 31, 3000 Hz

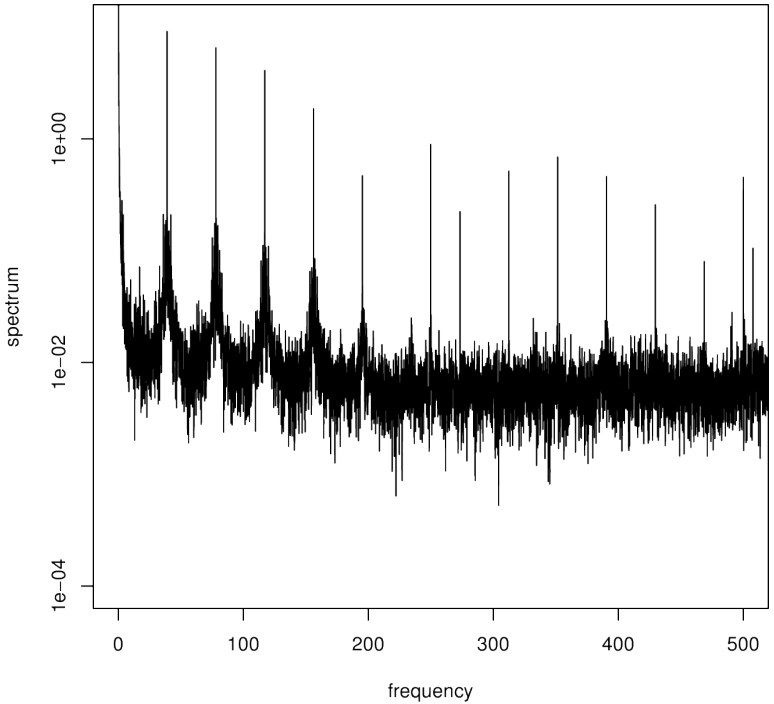


Linux

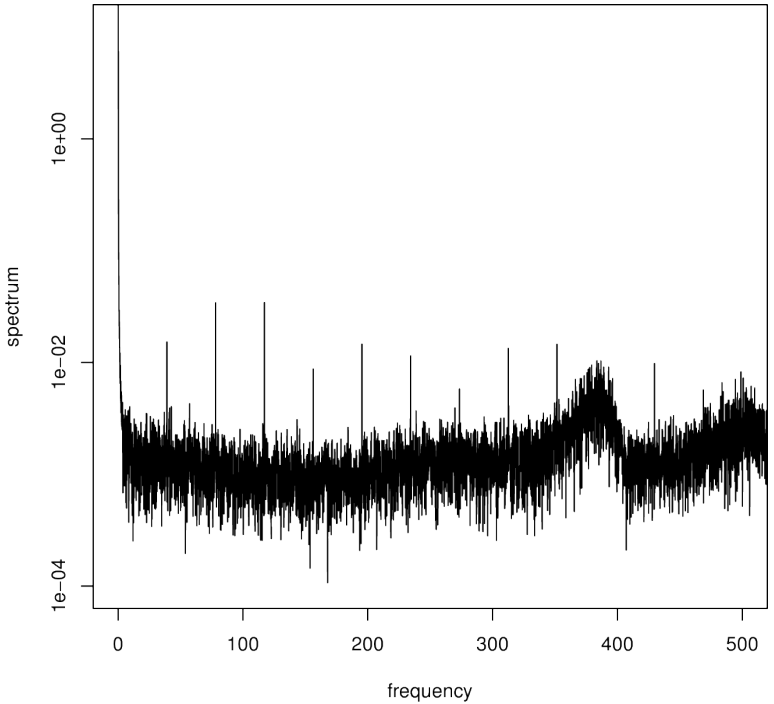


Akaros

FFTs: c89, Core 31, 500 Hz

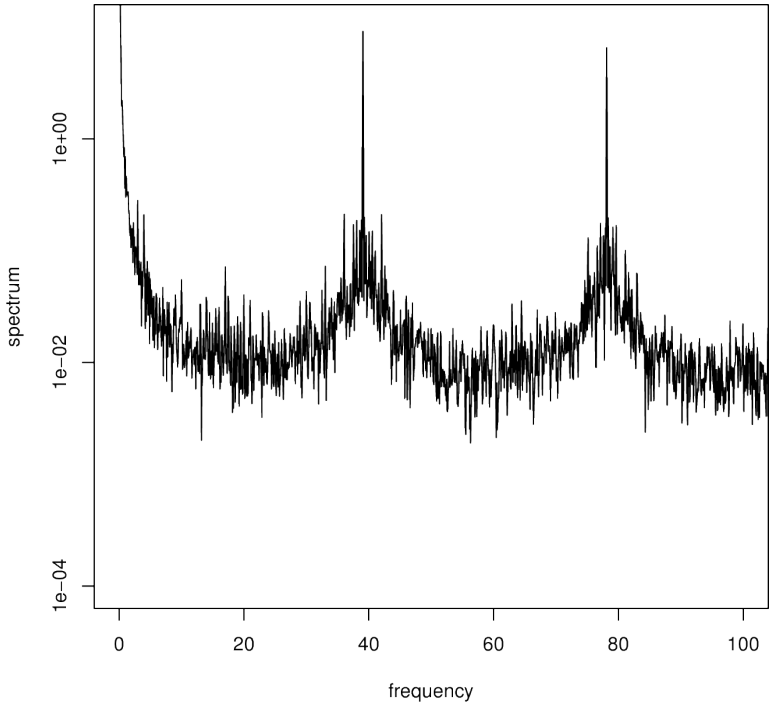


Linux

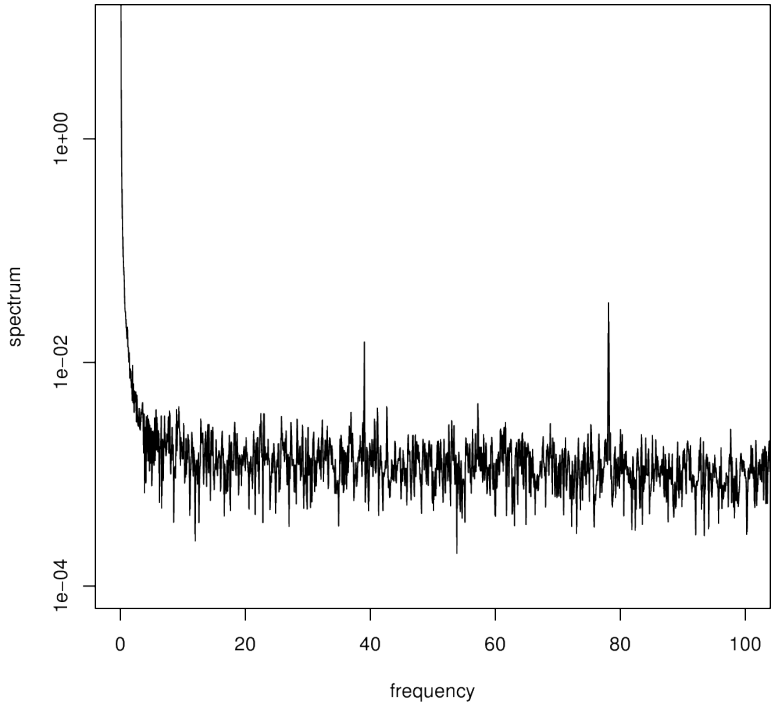


Akaros

FFTs: c89, Core 31, 100 Hz



Linux

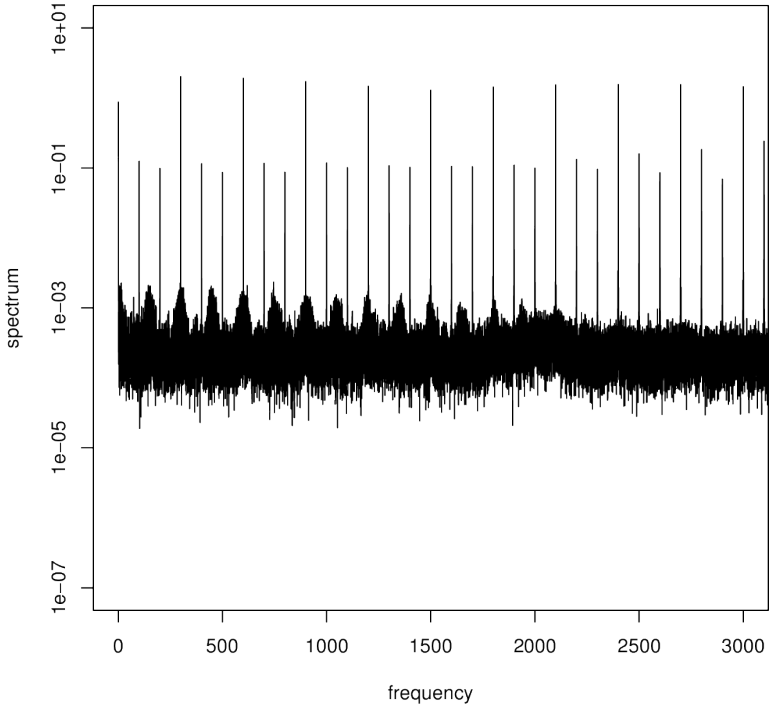


Akaros

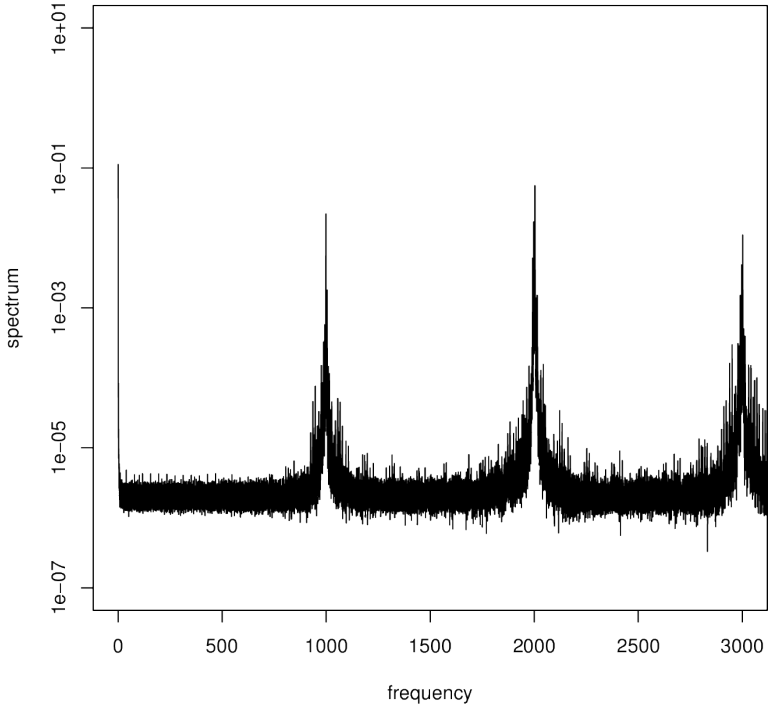
Let’s Try a Smaller Machine

- Intel Nehalem Core i7-920, 2.6GHz
- Single Socket
- 4 Cores, 8 hyperthreads
- 3 GB RAM
- Linux 3.11.1-gentoo
- This machine is called “hossin”

FFTs: Hossin, Core 7, 3000 Hz

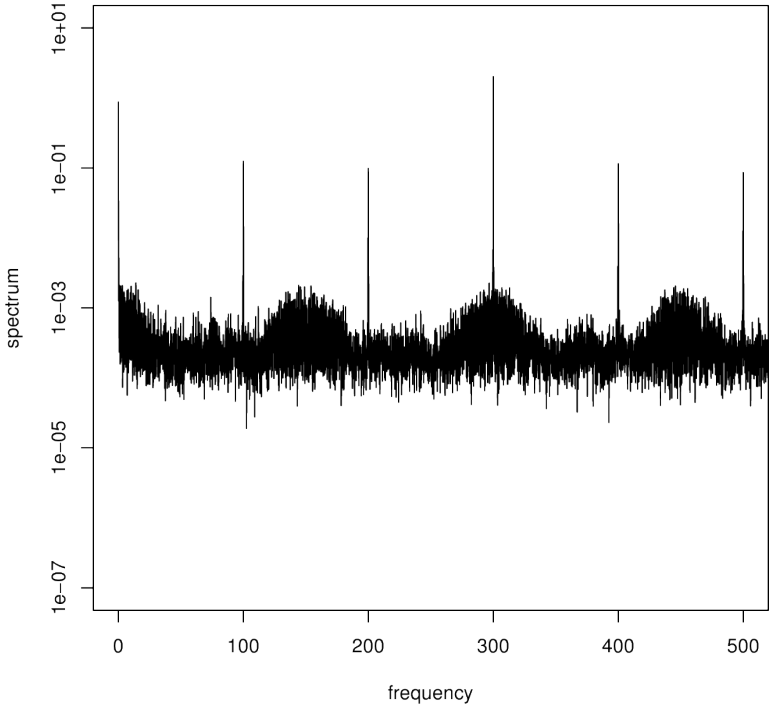


Linux

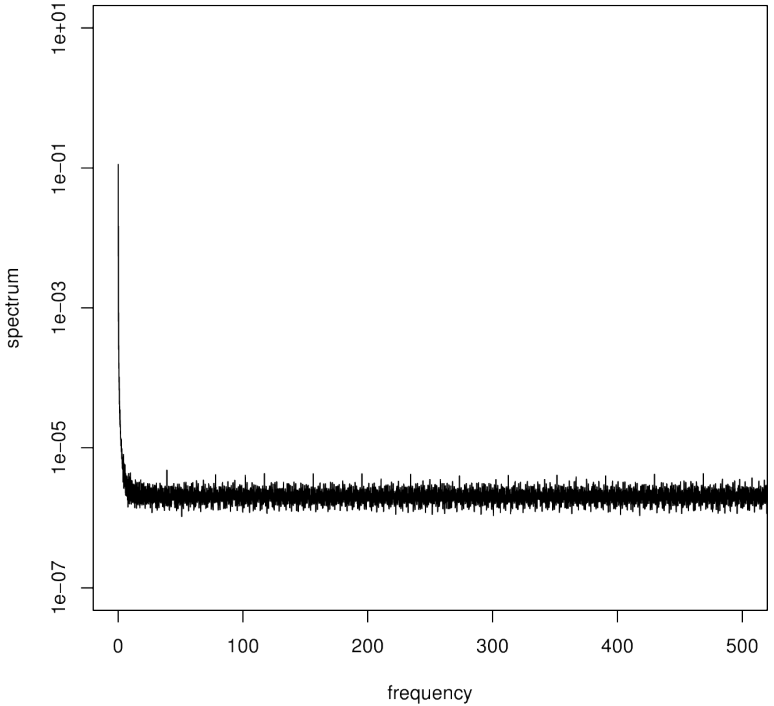


Akaros

FFTs: Hossin, Core 7, 500 Hz

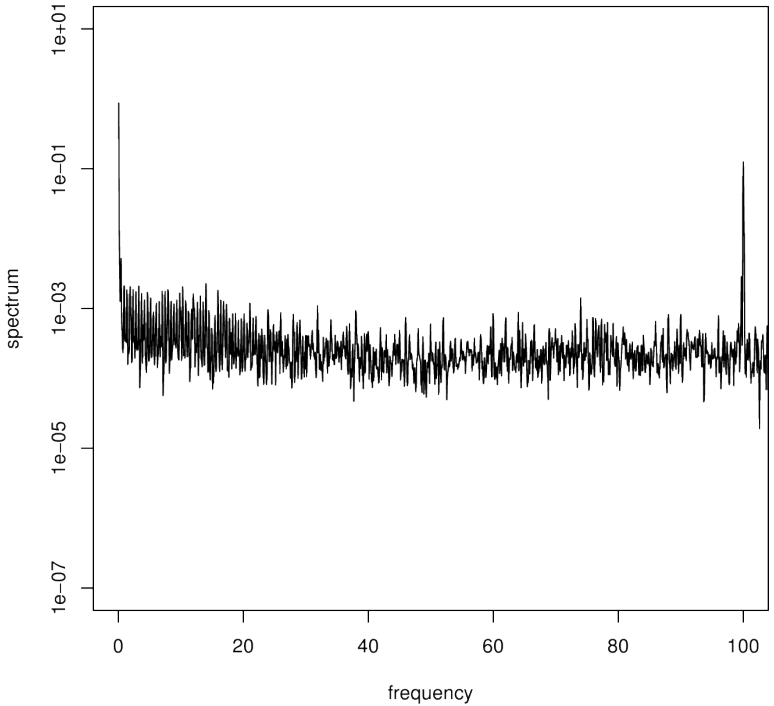


Linux

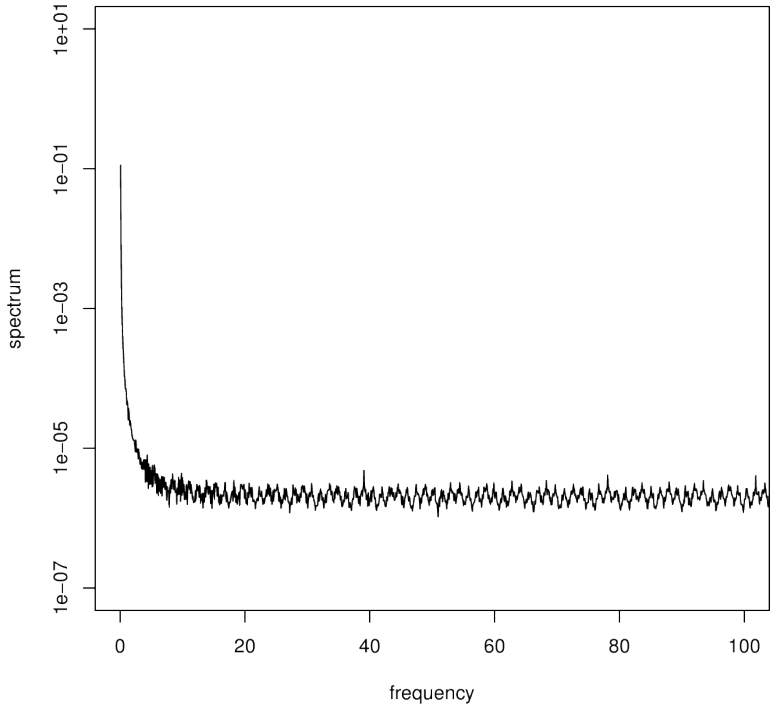


Akaros

FFTs: Hossin, Core 7, 100 Hz



Linux



Akaros

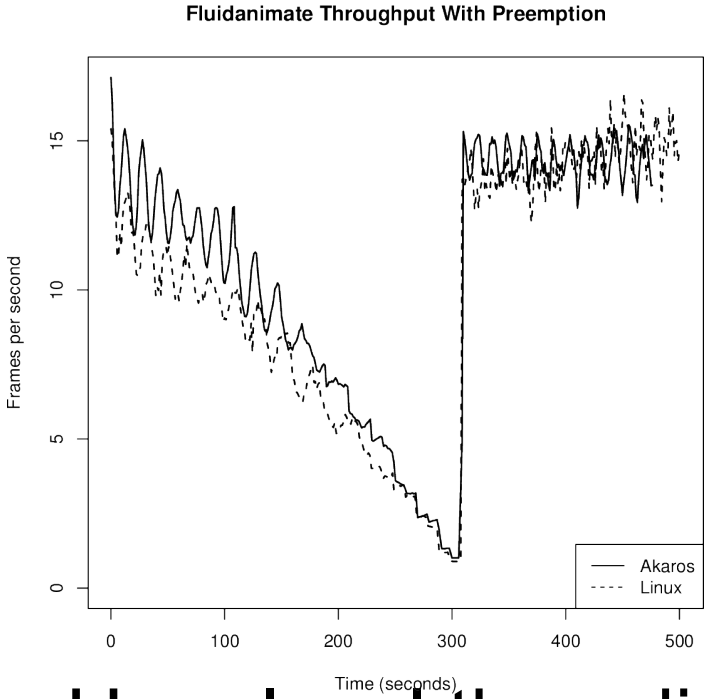
Isolation Summary

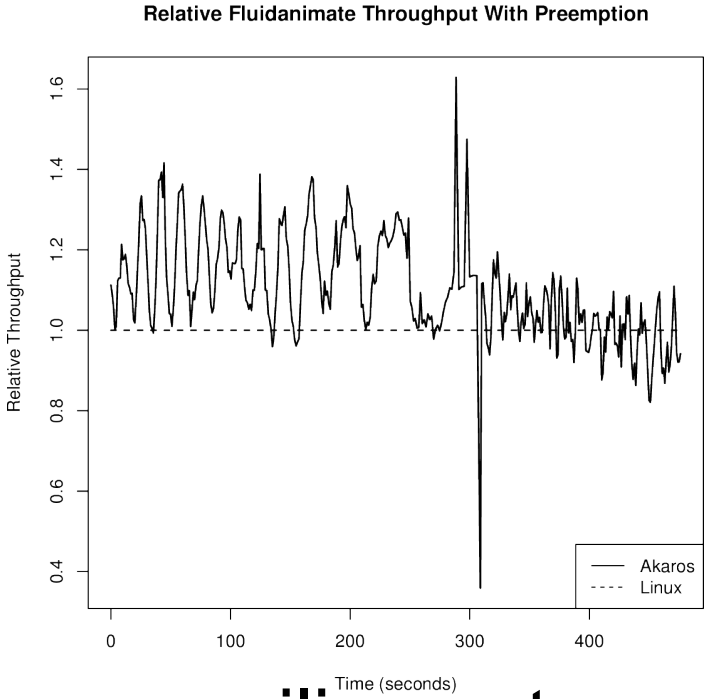
- Akaros's MCP cores have less noise and fewer signals compared to Linux
- The platform may be the source for various signals. e.g. System Management Mode
- Isolated cores come at a cost: housekeeping and IRQs are routed to Core 0

Applications

- Fluidanimate (PARSEC benchmark)
  - Requires a power-of-two number of threads
    - For a non-ideal number of cores, we may want many threads
  - Compute bound, operates in phases with barriers
  - Uses pthreads
- Kweb
  - Simple webserver, serves static files
  - Can use pthreads or a customized scheduler

Fluidanimate: Adapting to Preemption





User-level threading increases resilience to losing cores

Kweb: Customized Scheduling

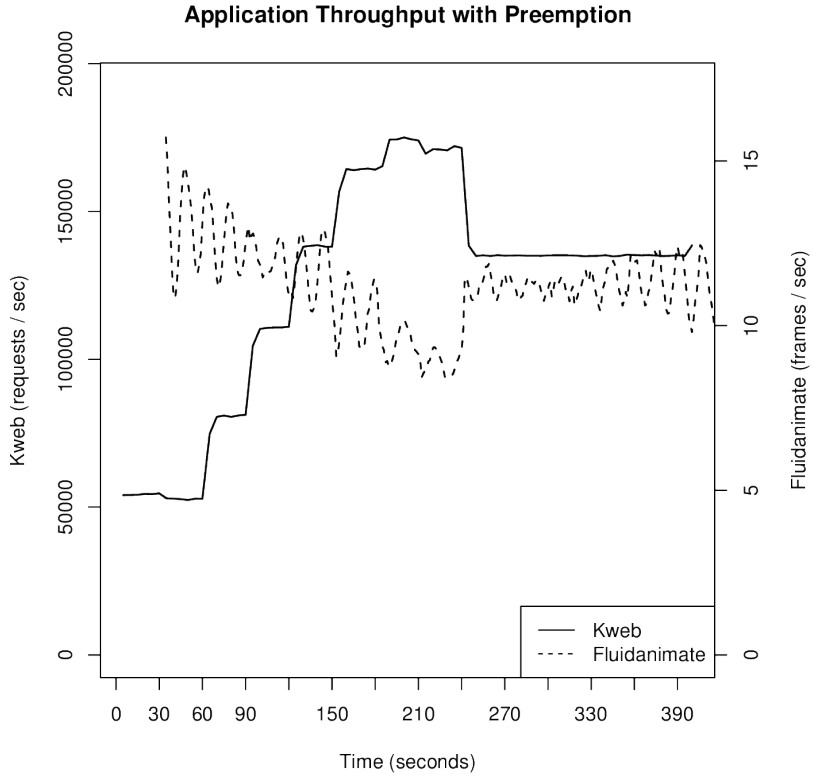
- Want all cores on a single socket (IRQ routing)
- Spin-poll for events (syscall completion)
- Turn off one sibling in a hyperthreaded pair
- We **could** process new requests as soon as a syscall blocks
  - Too much concurrency actually decreases overall performance (in this case)
  - Akaros's networking stack needs work

Kweb Throughput

Requests per second	**Avg.**	**Min.**	**Max.**	**Std.**
Linux	165,558	159,103	167,703	2395
Akaros, Pthreads, 6 VCs, Hyperthreads	134,676	133,121	135,567	635
Akaros, Pthreads, 6 VCs, No Hyperthreads	162,885	160,488	166,035	1661
Akaros, Custom 2LS, 6 Worker cores	**170,315**	**168,094**	**172,045**	766

httperf: 100 connections of 100,000 calls each, bursts of 100

Adapting to Changing Demands



- Kweb changes its resource requests
- Kernel preempts from fluidanimate to satisfy kweb's provision
- Kweb is not affected by fluidanimate

Summary

- Akaros: OS for high perf / parallel apps
  - Transparent access to physical resources
  - Provision and allocate ‘bare-metal’ resources
  - Single nodes, typically in a data center
- OS parallel process abstraction: MCP
  - Cores != Threads
  - Spatial allocation of cores to processes
  - Applications schedule and manage their threads
  - Performance isolation / minimize interference

Thanks!

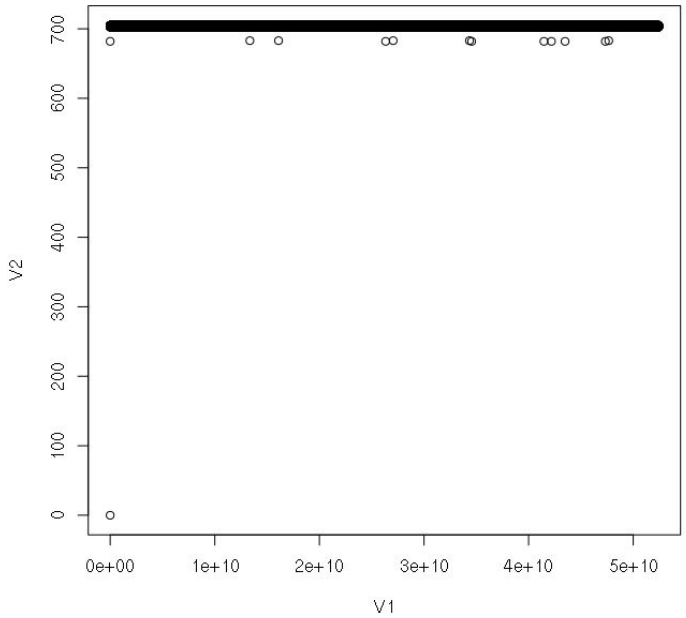
- Eric Brewer
- Krste Asanović, David Culler, David Wessel, John Chuang
- Kevin Klues, David Zhu, Andrew Waterman, Paul Pearce
- Ron Minnich, Andrew Gallatin, Tamara Broderick
- NSF grants #1320005 and #1016714


AKAROS

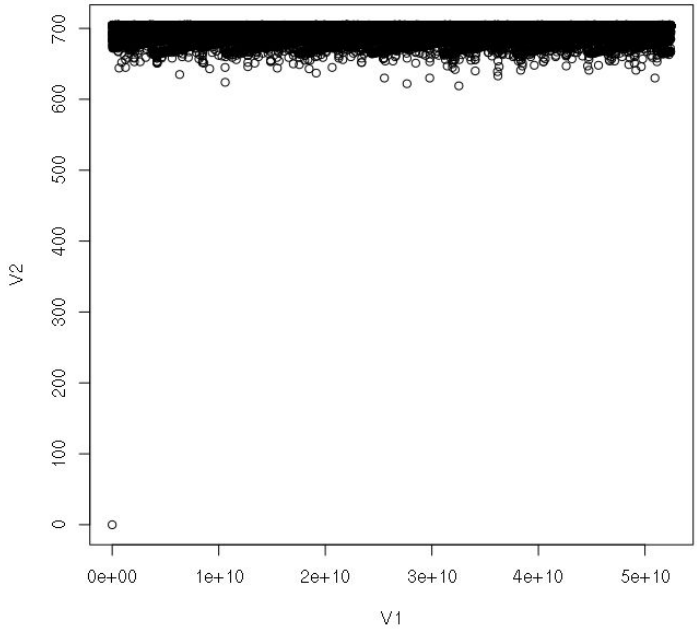
backup

Hossin Raw FTQ

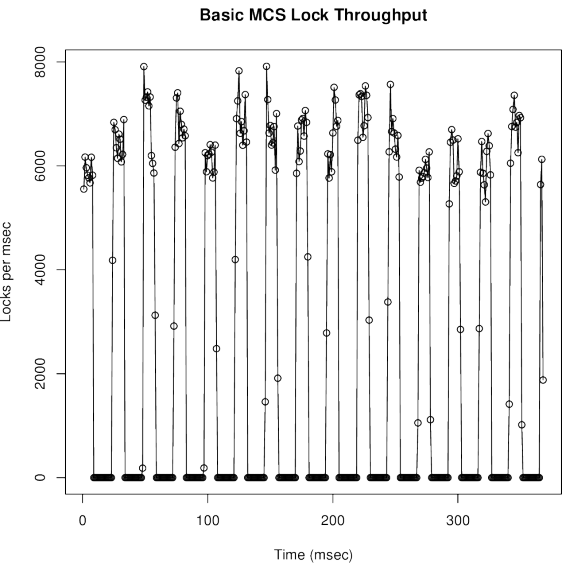
Akaros: only 12 variations out of 500,000+ samples.

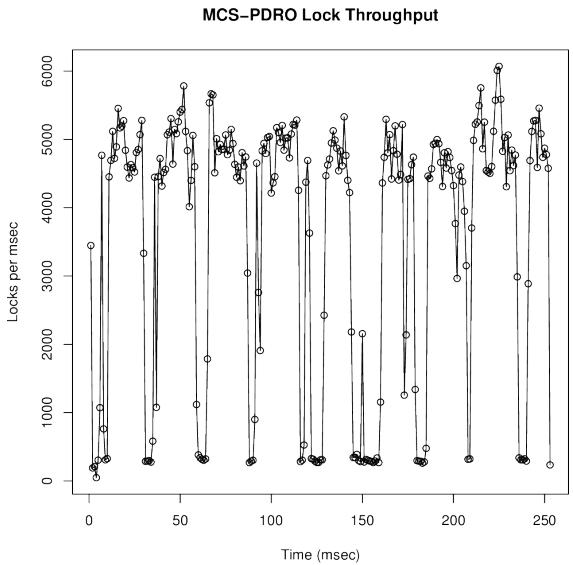


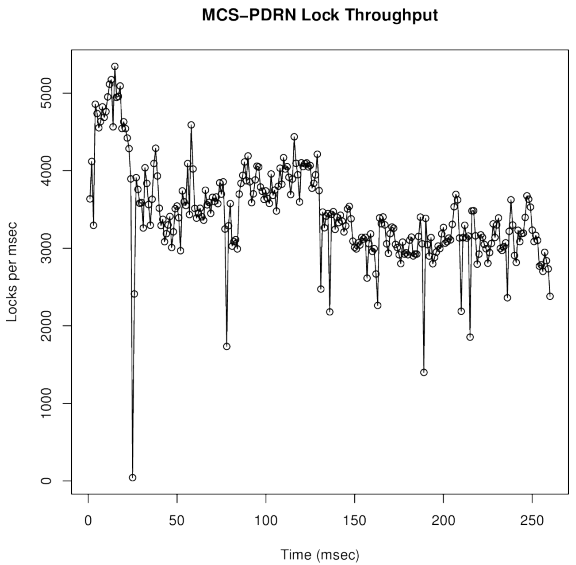
Linux over here



MCS locks with preemption



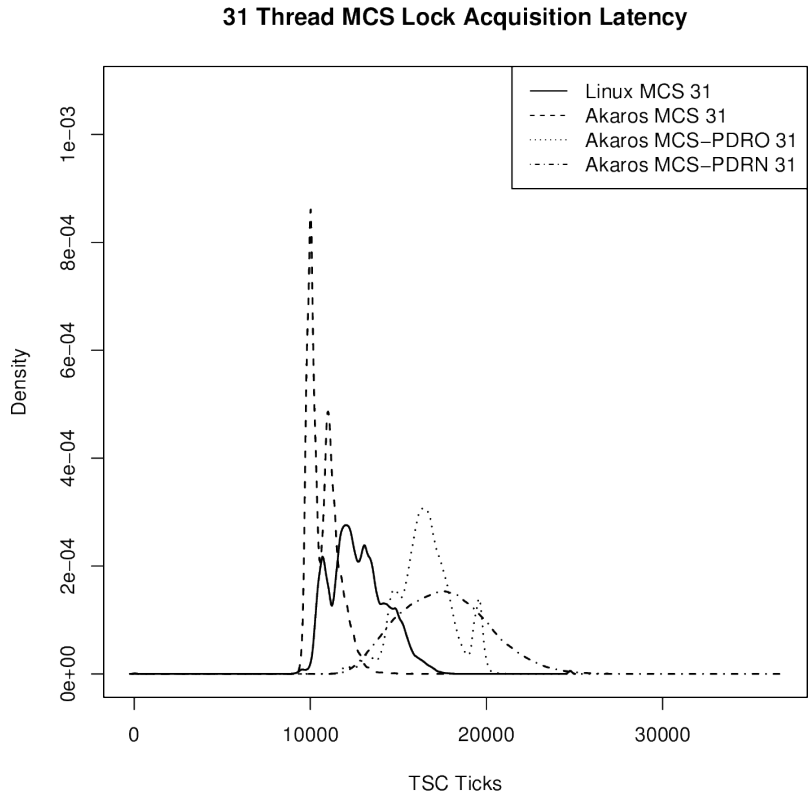




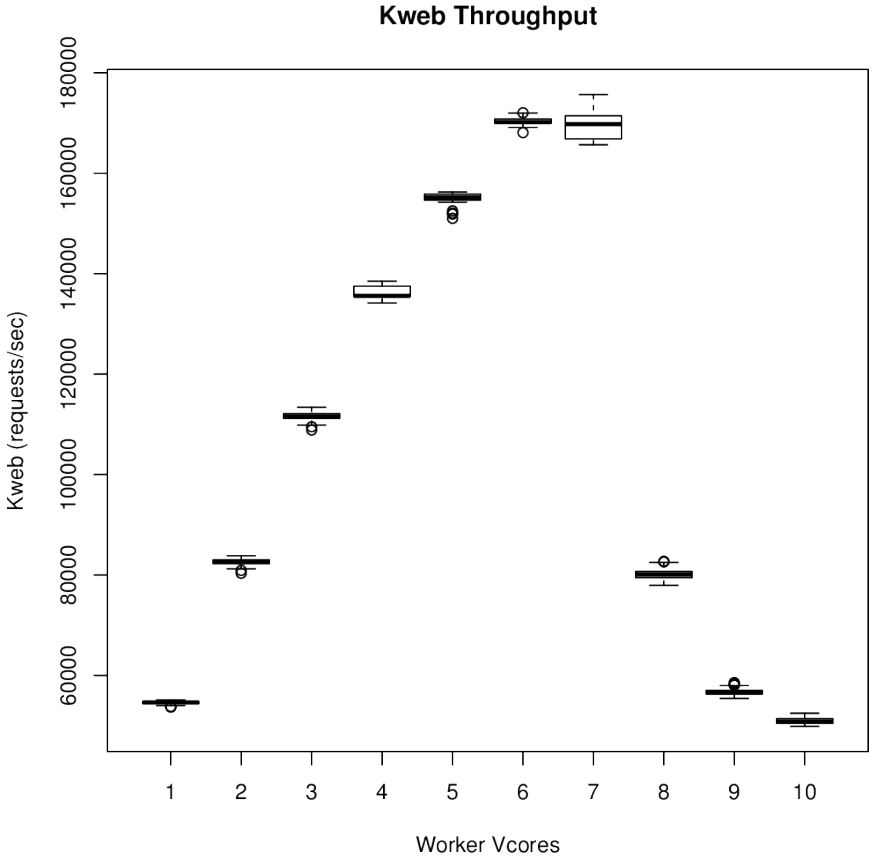
Plan 9 Stack

- Replacing our VFS with Plan 9 namespaces
  - Used Coccinelle to transform for Akaros
  - Ron and I can port a Plan 9 NIC driver in an hour
- Still have glibc, it just uses Plan 9 devices
- Work in progress to build mmap() for Plan 9
- Currently, we have an uneasy mix of VFS (with an in-memory FS) and Plan 9
- Plan 9's networking stack needs work

MCS Lock Acquisition Latency



Custom Kweb: Throughput vs Cores



Kernel Perspective

- Monolithic kernel
- Can run the kernel anywhere; choose to run most of the kernel on a subset of cores
- Userspace determines where syscalls run
  - Locally, via sysenter/syscall traps into the kernel
  - Remotely, via shared memory rings (requires server)
- Designed to handle tricky circumstances
  - e.g. syscall completion event sent during preemption recovery of a lock-holder, while yielding spare cores

Summary

- Akaros: research OS for high perf / parallel apps
- Provision and allocate ‘bare-metal’ resources
- Process model: cores != threads
- Go, Plan 9, and Glibc
- More info:
  - github.com/brho/akaros.git
  - http://akaros.cs.berkeley.edu/
- The giraffe’s name is Nanwan