Case for Transforming Parallel Run-times Into Operating System Kernels

Paper Reading Group

Kyle Hale
Peter Dinda
Presents: Maksym Planeta

18.02.2016

Table of Contents

Table of Contents

What is this project?

1. Northwestern University, Sandia Labs, Oak Ridge
2. Part of Hobbes Project
3. They also develop Palacios

Why is it interesting for us?

- Proposes a microkernel

Why is it interesting for us?

- Proposes a microkernel
- Uses hyperthreads in HPC context

Why is it interesting for us?

- Proposes a microkernel
- Uses hyperthreads in HPC context
- Targets Xeon Phi

Why is it interesting for us?

- Proposes a microkernel
- Uses hyperthreads in HPC context
- Targets Xeon Phi
- It cites L4 paper:

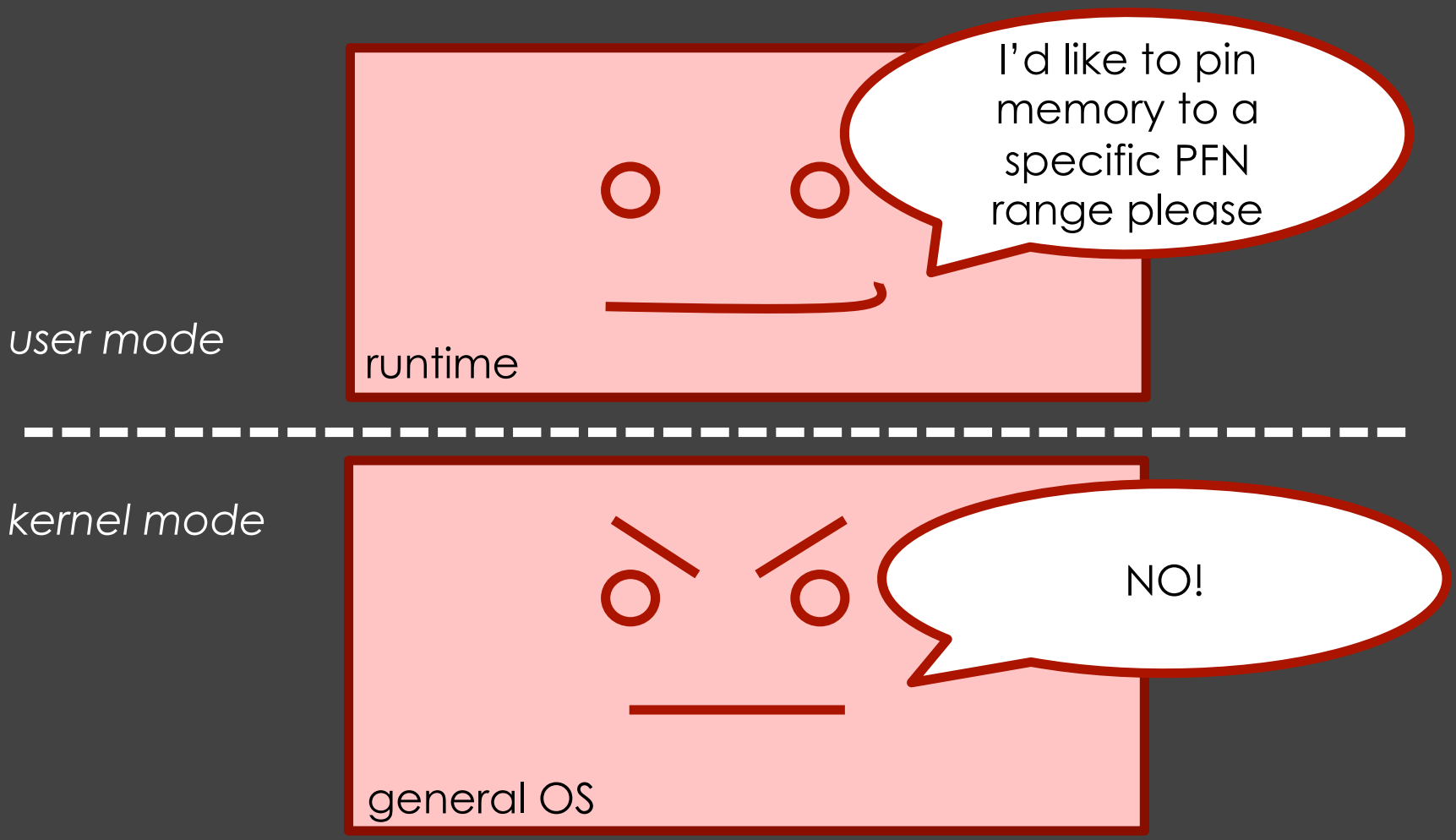
[40] J. Liedtke. On micro-kernel construction. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 237–250, Dec. 1995.

Idea

1. HPC app runs in user mode
2. Hardware available in kernel mode
3. When an HPC program runs in kernel mode:
   3.1 All nice features are directly available

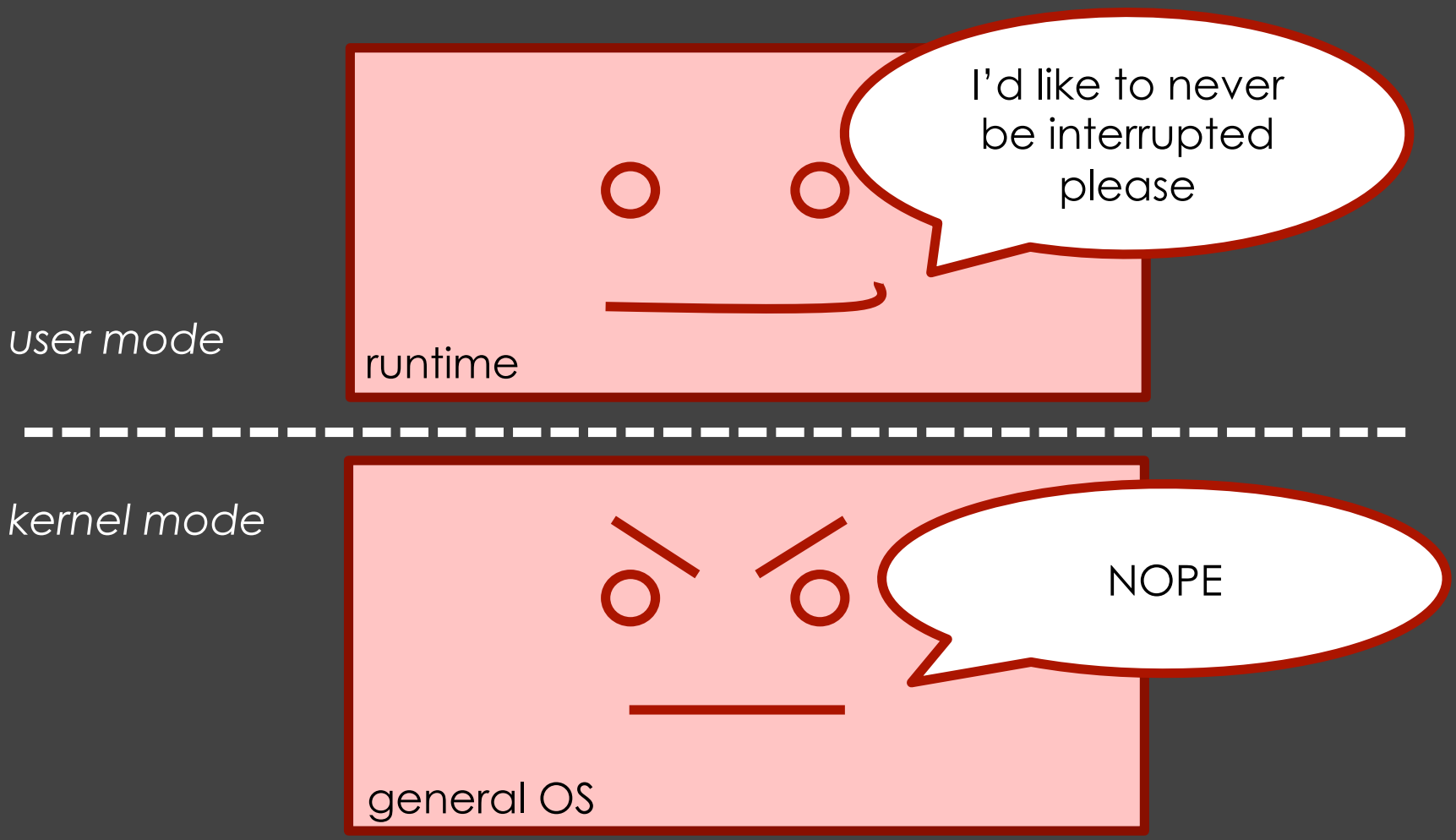
A Typical dialog with the kernel

ARE PROVIDED KERNEL ABSTRACTIONS THE RIGHT ONES?



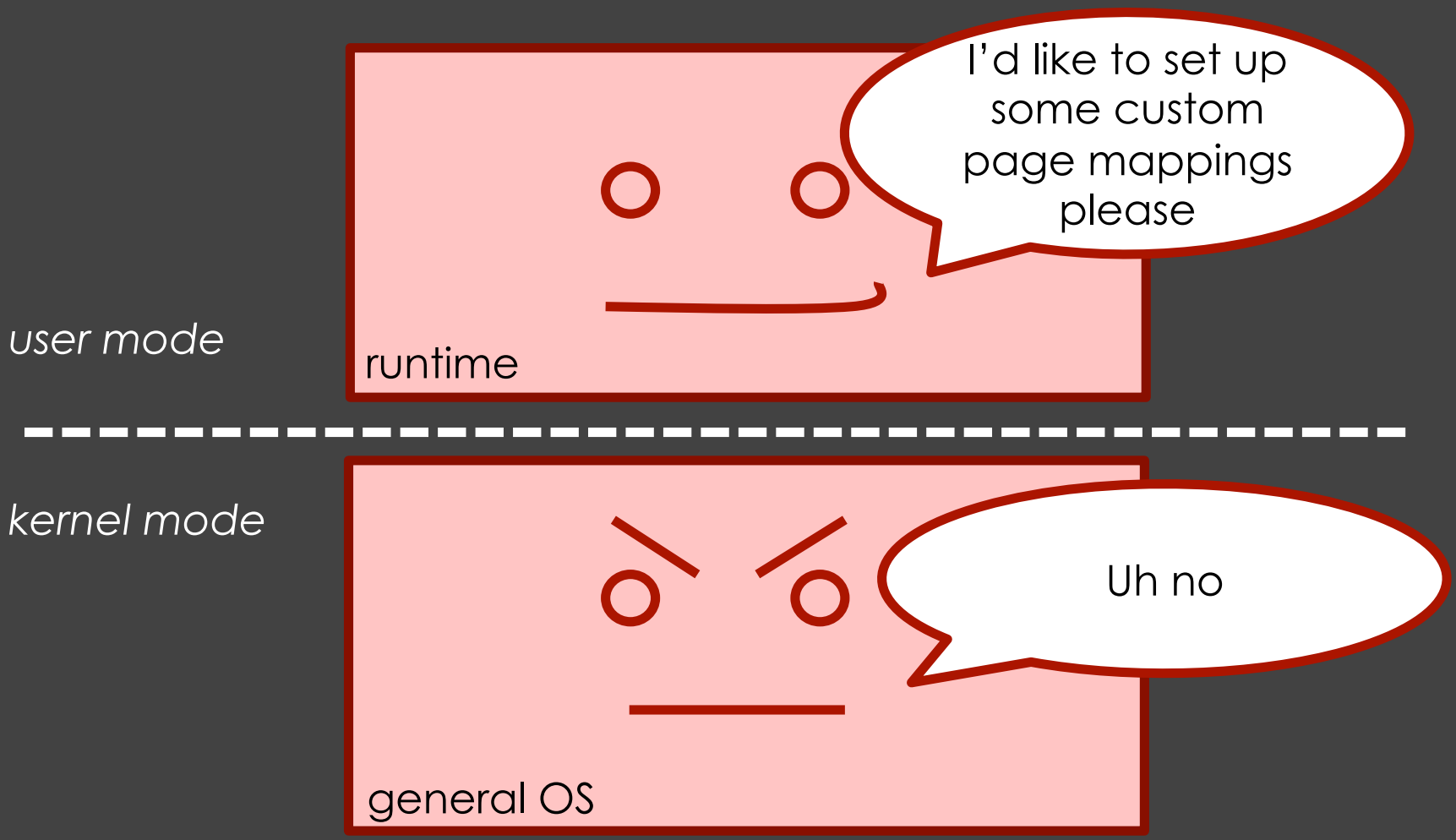
NOT ALWAYS

ARE PROVIDED KERNEL ABSTRACTIONS THE RIGHT ONES?

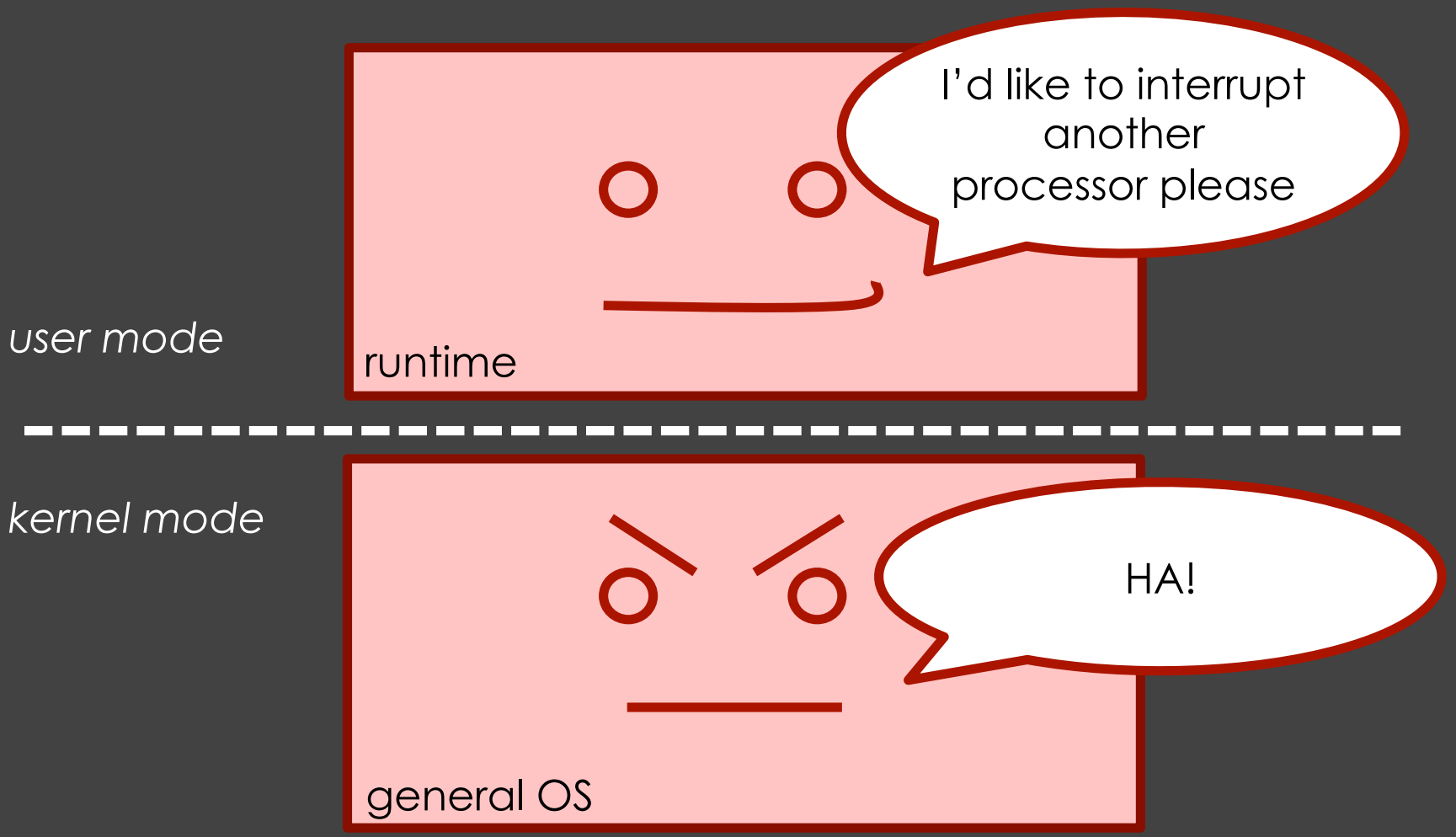


NOT ALWAYS

RESTRICTED ACCESS TO HARDWARE



RESTRICTED ACCESS TO HARDWARE



Motivation

1. HPC app runs in user mode
2. Hardware available in kernel mode
3. When an HPC program runs in kernel mode:
   - 3.1 All nice features are directly available
   - 3.2 Kernel does not restrict the program with bad abstractions:

     For example, the run-time might need subset barriers, and be forced to build them out of mutexes.

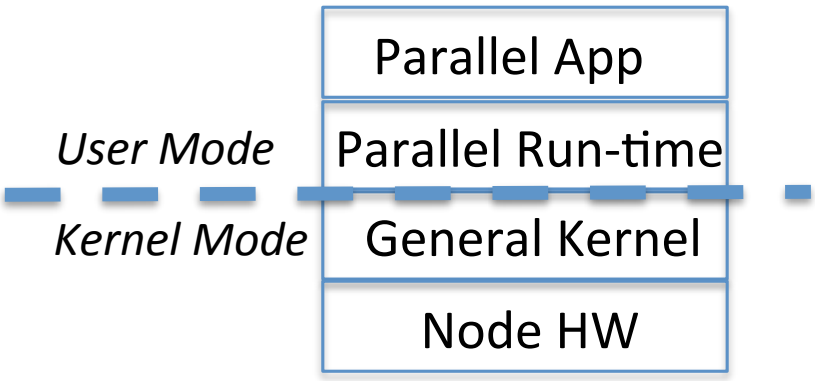
Motivation

1. HPC app runs in user mode
2. Hardware available in kernel mode
3. When an HPC program runs in kernel mode:
   - 3.1 All nice features are directly available
   - 3.2 Kernel does not restrict the program with bad abstractions:
     For example, the run-time might need subset barriers, and be forced to build them out of mutexes.
   - 3.3 Kernel may waste resources for the features the application doesn't need:
     For example, the run-time might not require coherence, but get it anyway.

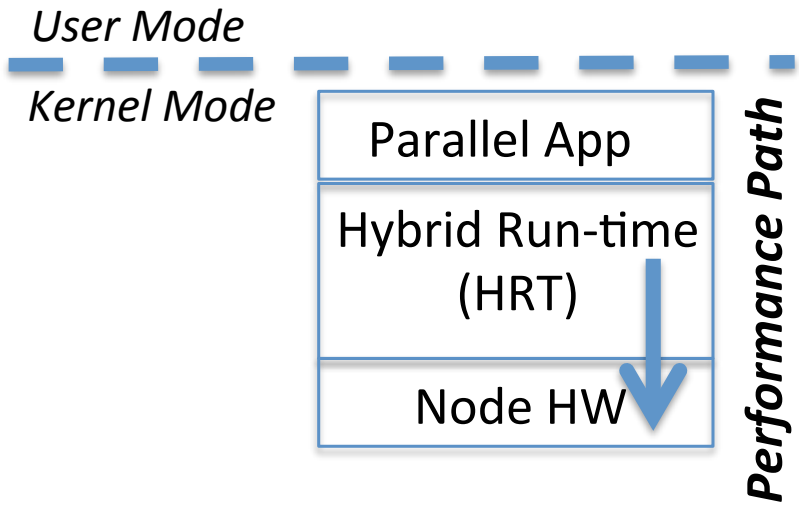
Contributions

1. Criticize traditional architecture
2. Propose a new OS structure
3. Port some of the existing runtimes

Hybrid Run-Time (HRT)



(a) Current Model



(b) Hybrid Run-time Model

What is HRT?

- The runtime is the kernel, built within a kernel framework

What is HRT?

- The runtime is the kernel, built within a kernel framework
- Everything is kernel space

What is HRT?

- The runtime is the kernel, built within a kernel framework
- Everything is kernel space
- HRT has full access to the hardware

What is HRT?

- The runtime is the kernel, built within a kernel framework
- Everything is kernel space
- HRT has full access to the hardware
- HRT can pick its own abstractions

Aerokernel

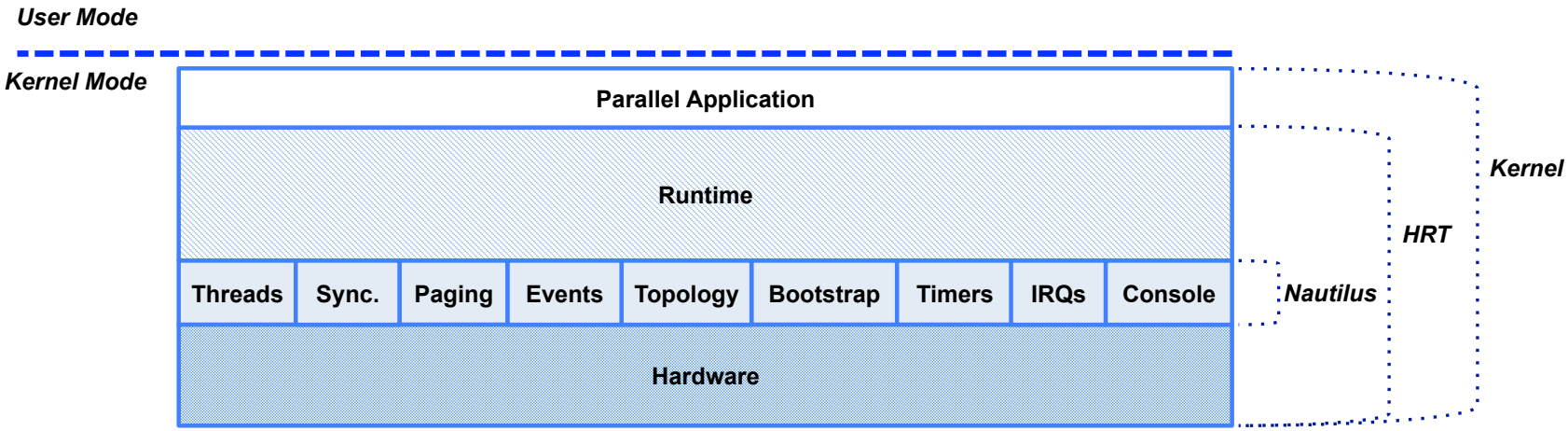
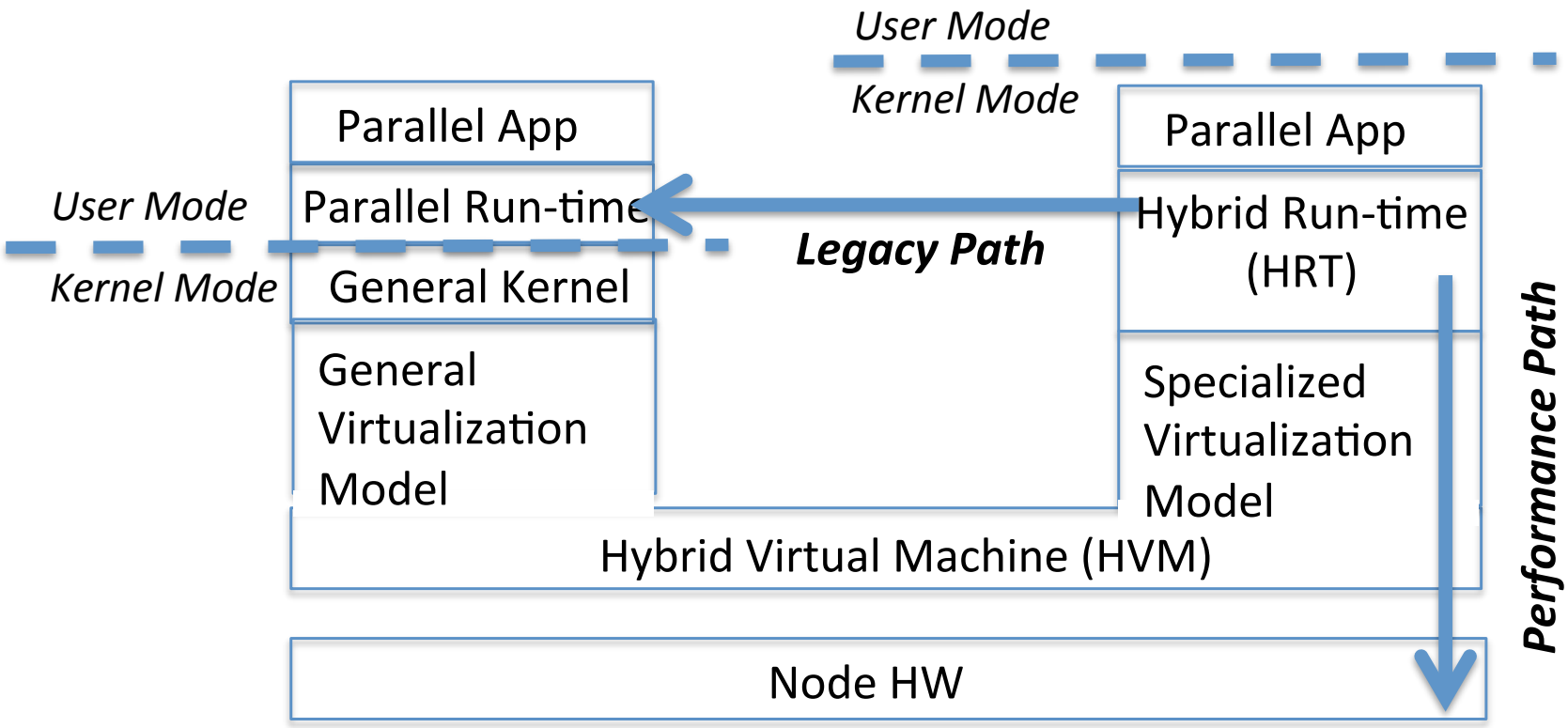


Figure 2: Structure of Nautilus.

Benefits

- Better abstractions
- Noiseless
- Lightweight

Legacy support



(c) Hybrid Run-time Model Within a Hybrid Virtual Machine

Table of Contents

Thread creation

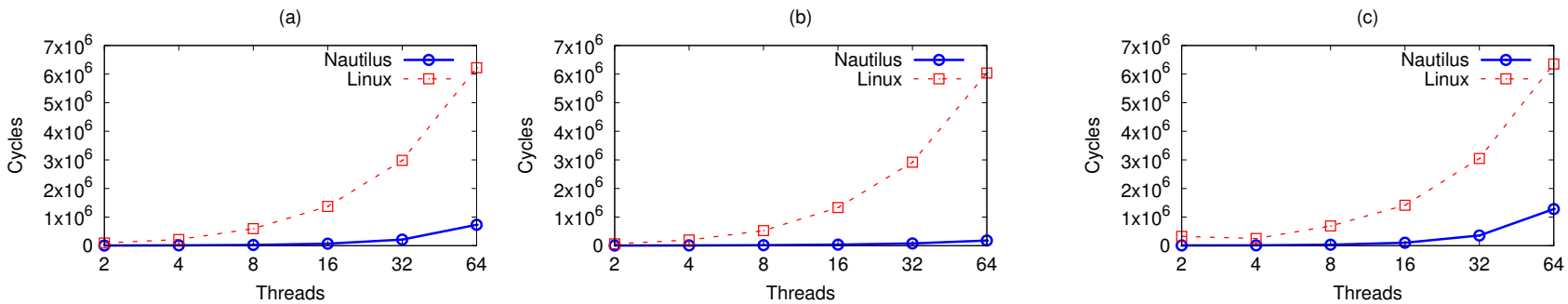


Figure: Average, minimum, and maximum time to create a number of threads in sequence.

Thread creation

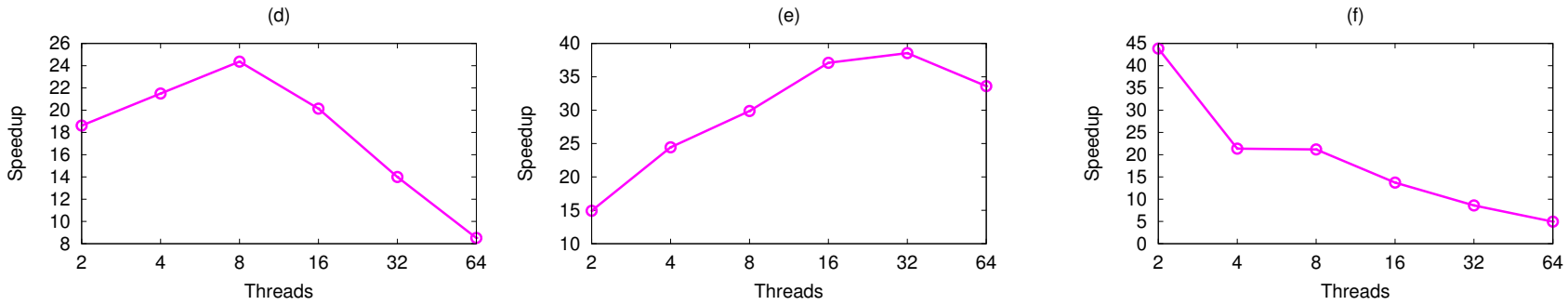


Figure: $\frac{Linux}{Nautilus}$ from previous figure

Thread creation

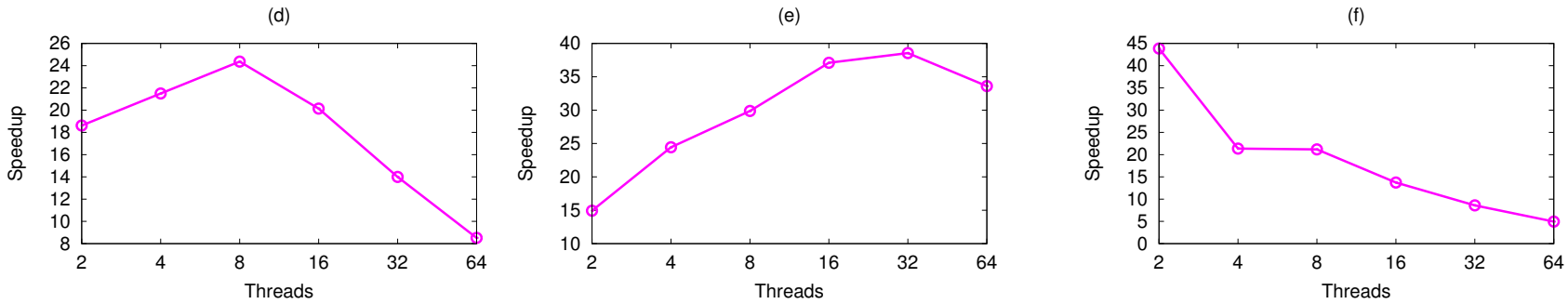


Figure: $\frac{Linux}{Nautilus}$ from previous figure

Why bends? At (d) at 8 threads, (e) at 32, and (f) at 8. Bugs?

Thread creation (summary)

OS	Avg	Min	Max
Nautilus	16795	2907	44264
Linux	38456	34447	238866

Figure 3: Time to create a single thread measured in cycles.

Spinlock microbenchmark

OS	**Execution time (s)**
Nautilus	13.72
Linux	12.53
OS	**Avg. acquire/release time (cycles)**
Nautilus	59
Linux	36

Figure 5: Total time to acquire and release a spinlock 500 million times on Nautilus and Linux, and average time in cycles for an acquire/release pair.

Wake-up microbenchmark

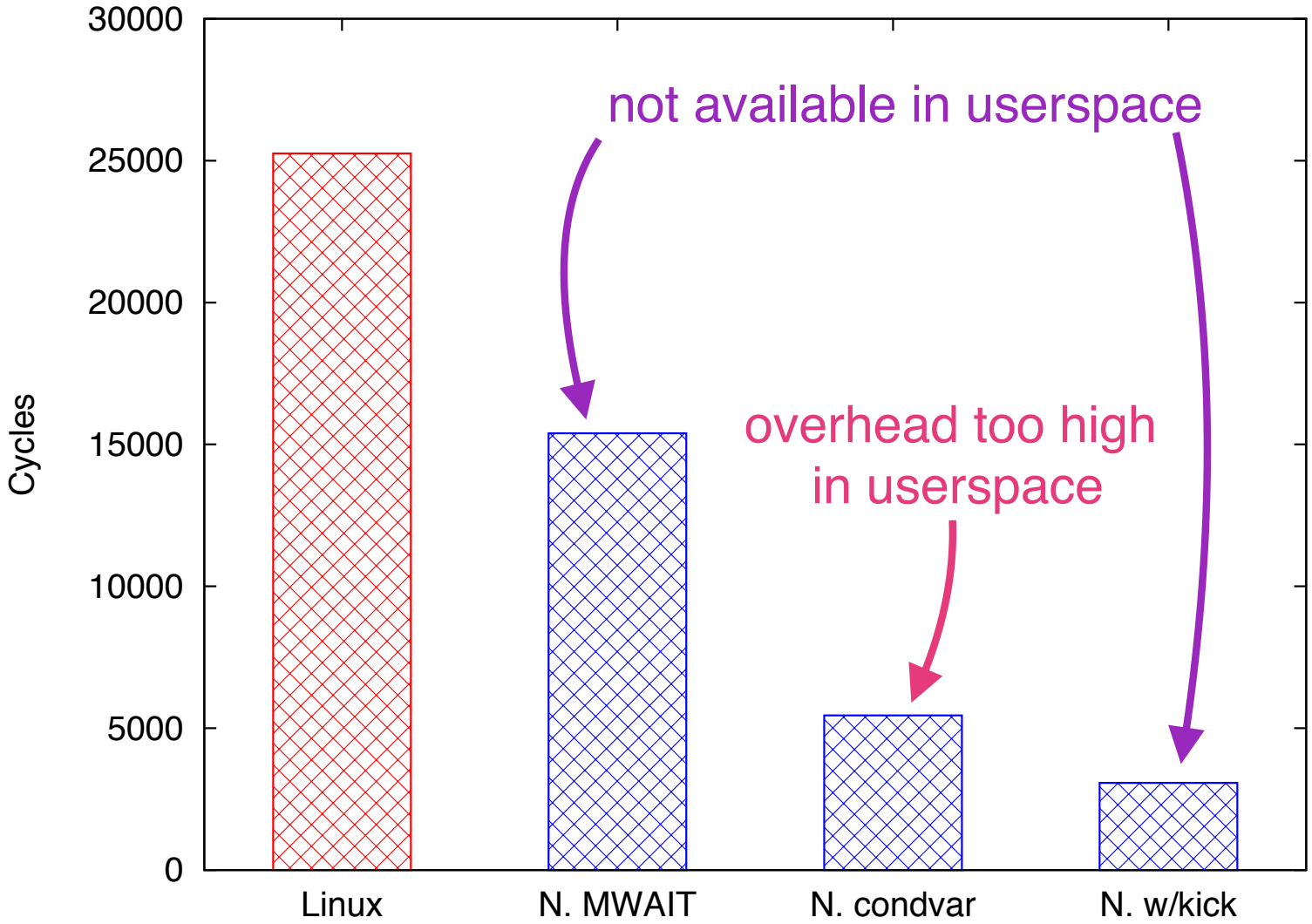


Figure 6: Average event wakeup latency.

Circuit simulator benchmark

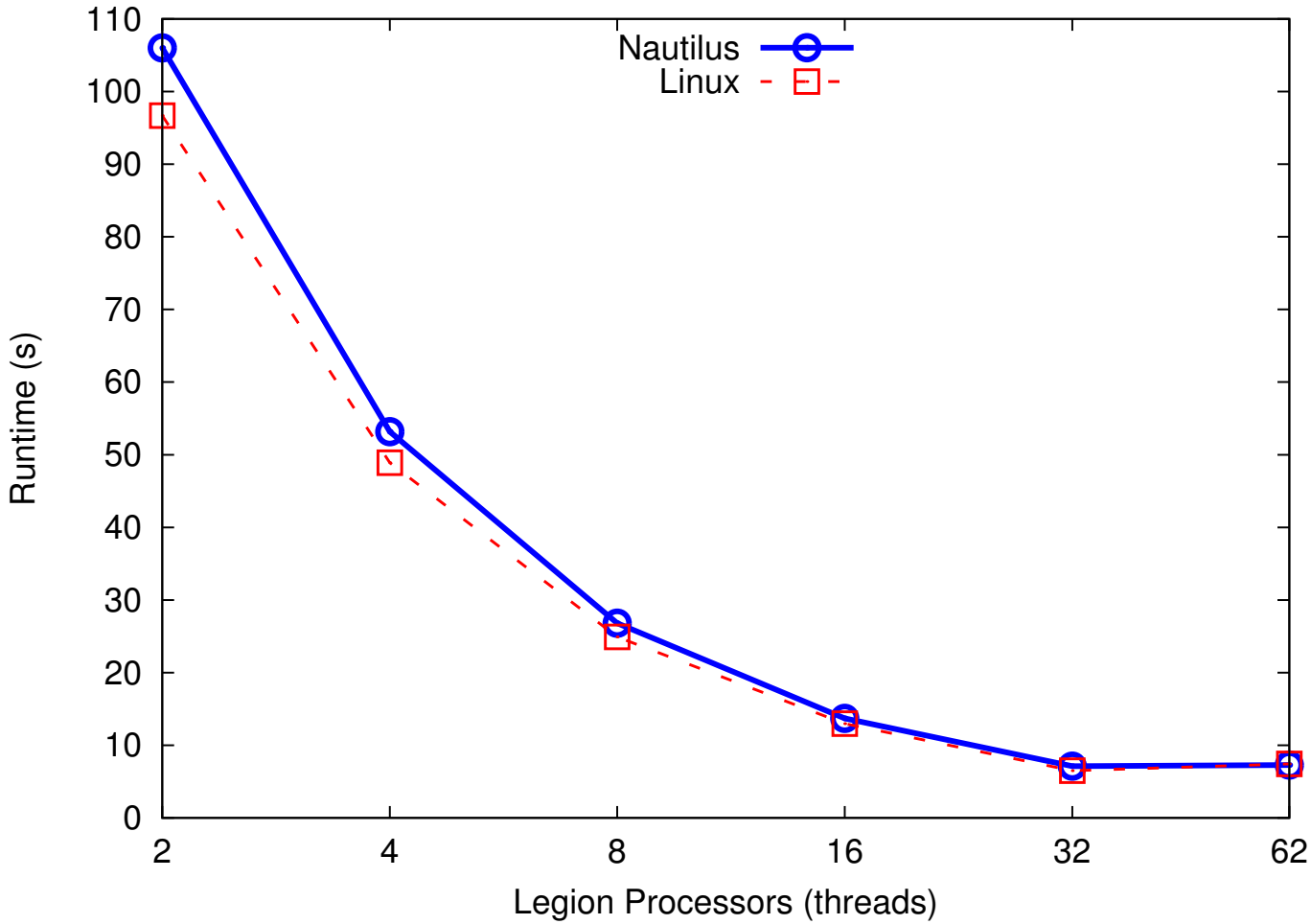


Figure 11: Run time of Legion circuit simulator versus core count. The baseline Nautilus version has higher performance at 62 cores than the Linux version.

Circuit simulator benchmark

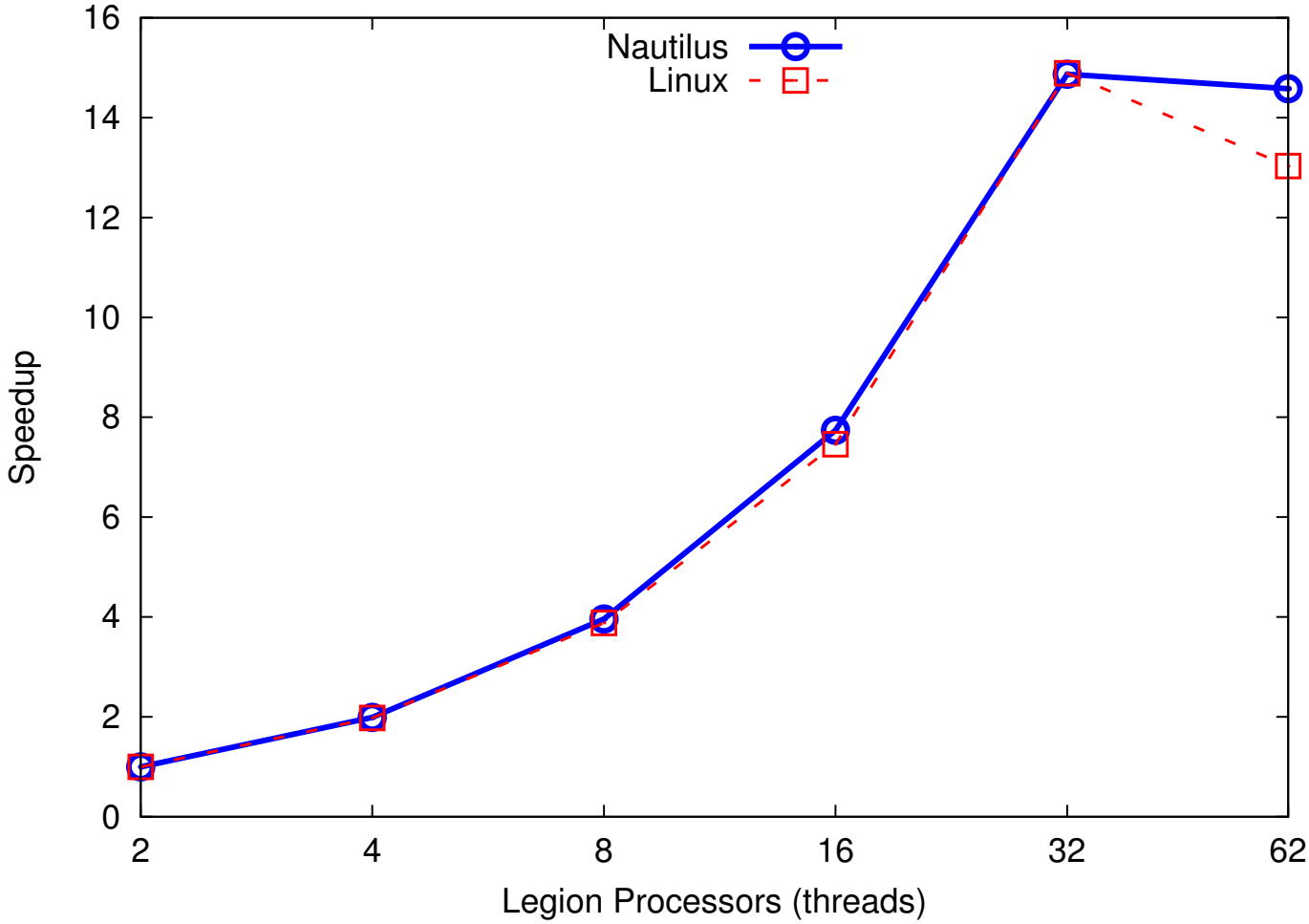


Figure 12: Speedup of Legion (normalized to 2 Legion processors) circuit simulator running on Linux and Nautilus as a function of Legion processor (thread) count.

Circuit simulator benchmark

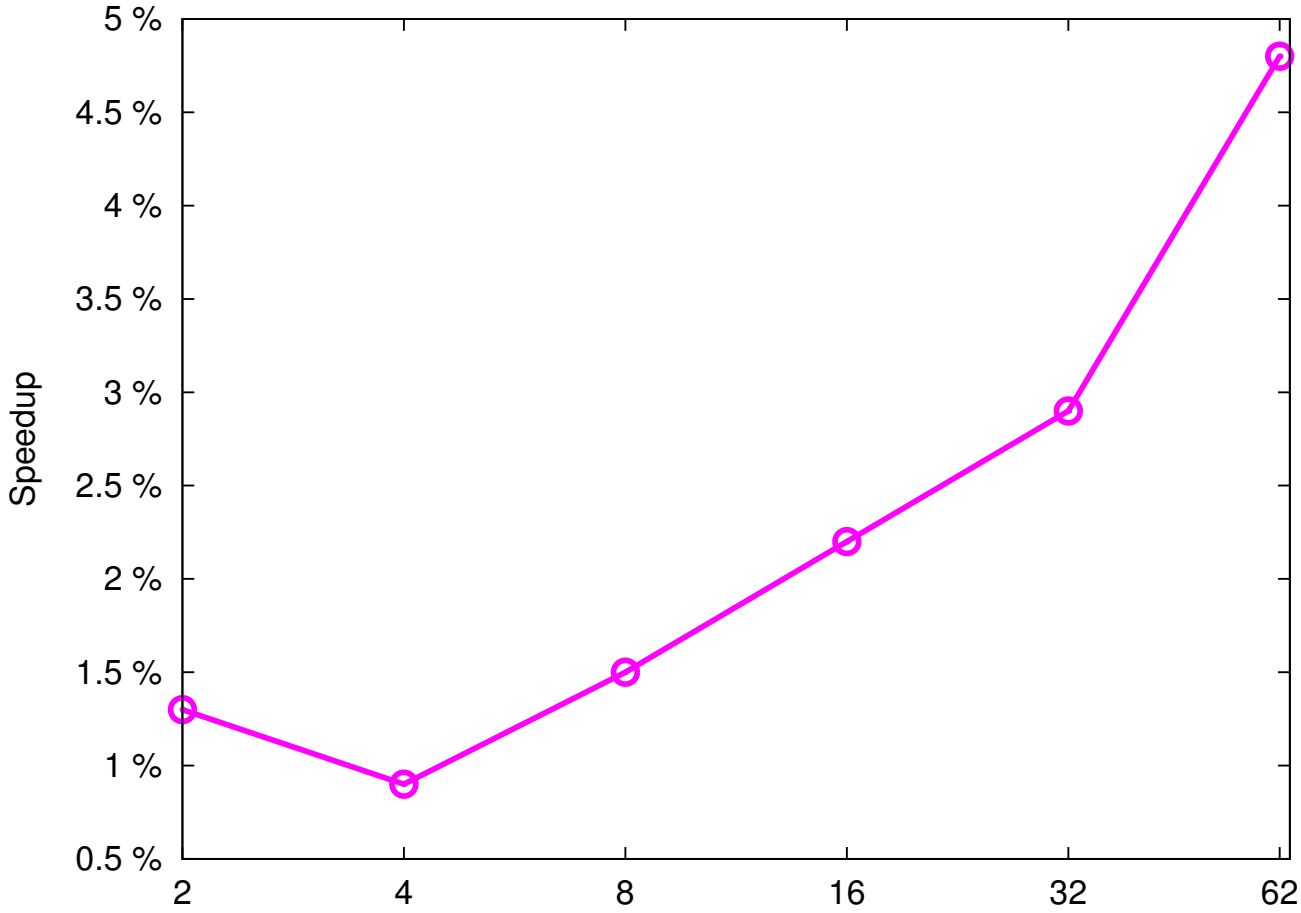


Figure 13: Speedup of Legion circuit simulator comparing the baseline Nautilus version and a Nautilus version that executes Legion tasks with interrupts off.

Table of Contents

Kernel development

The process of building Nautilus as minimal kernel layer with support for a complex, modern, many-core x86 machine took six person-months of effort on the part of seasoned OS/VMM kernel developers.

Language	**SLOC**
C	22697
C++	133
x86 Assembly	428
Scripting	706

Figure 9: Source lines of code for the Nautilus kernel.

Run-time support

Porting Legion:

- 43000 SLOC in C++
- Most of the work went into understating Legion
- Some code added to Nautilus

Language	SLOC
C++	133
C	636

Figure 10: Lines of code added to Nautilus to support Legion, NDPC, and NESL.

- Four person-months to port

Also porting NESL and NDPC (related to each other).

Table of Contents

Conclusion

- A mikrokernel
- And a lightweight kernel
- Requires effort for porting
- Early stage of development