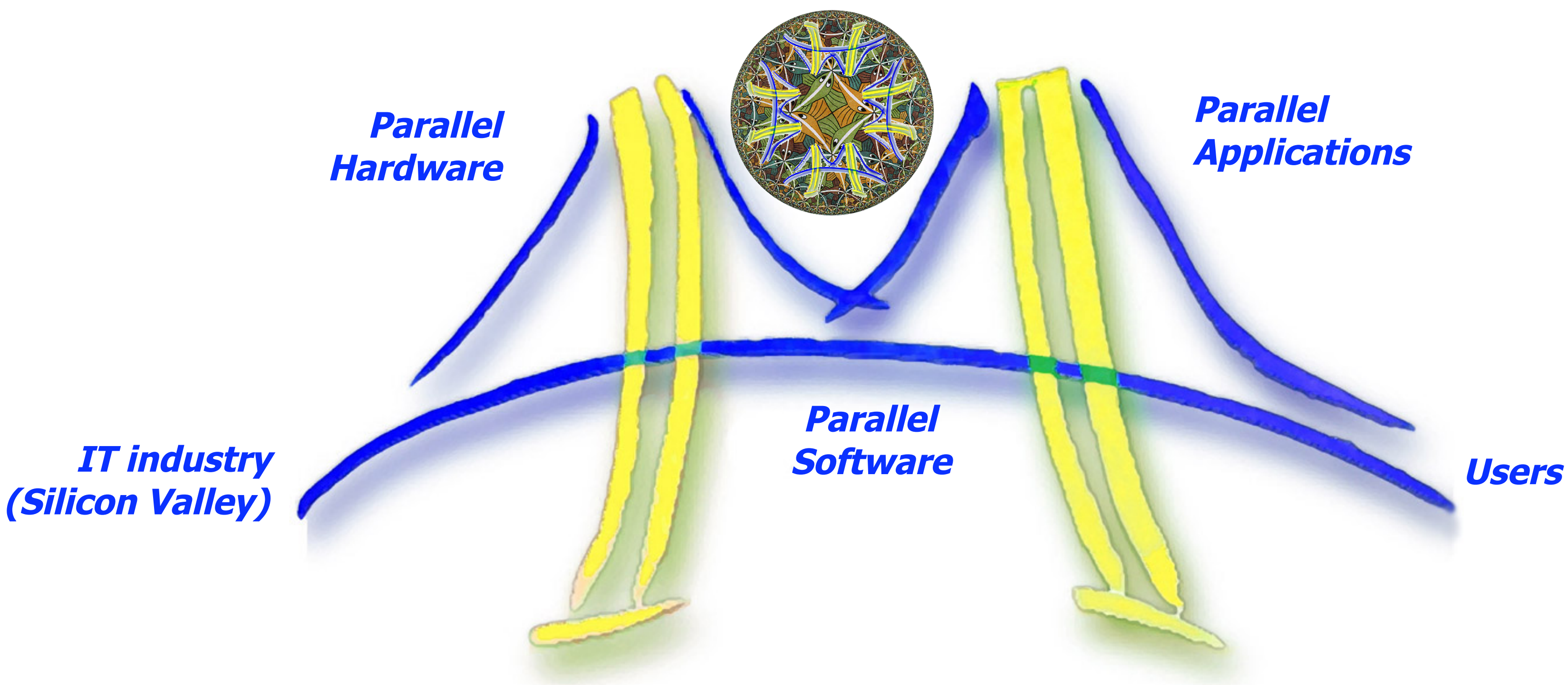


ROS: Redesigning the OS System Call Interface for Manycore

Kevin Klues, Barret Rhoden, David Zhu

CS 262b Presentation

Overview

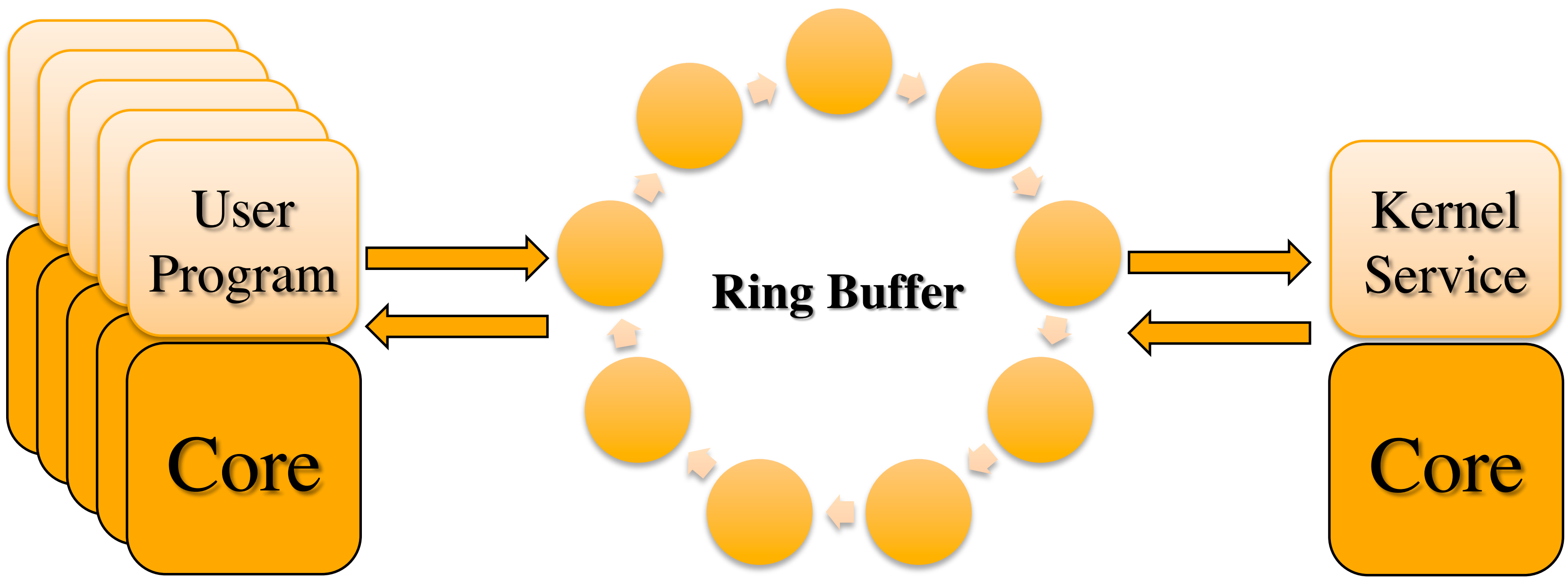
- Problem: Current operating systems are not designed for manycore architectures
  - Do not scale well in a multi-core environment
  - Do not support high performance parallel applications
- Our idea: Structure the operating system asymmetrically
  - Provide an asynchronous syscall interface to users
  - Service syscalls on dedicated kernel cores
- Solution:
  - Built an OS from scratch
  - Implemented asynchronous remote syscalls
  - Compared them to traditional approach

Outline

- Architecture
- Implementation
- Evaluation Methodology
- Results
- Conclusion
- Future Work

Architecture

- Asymmetric OS on symmetric hardware



Asynchronous Remote Syscall Interface

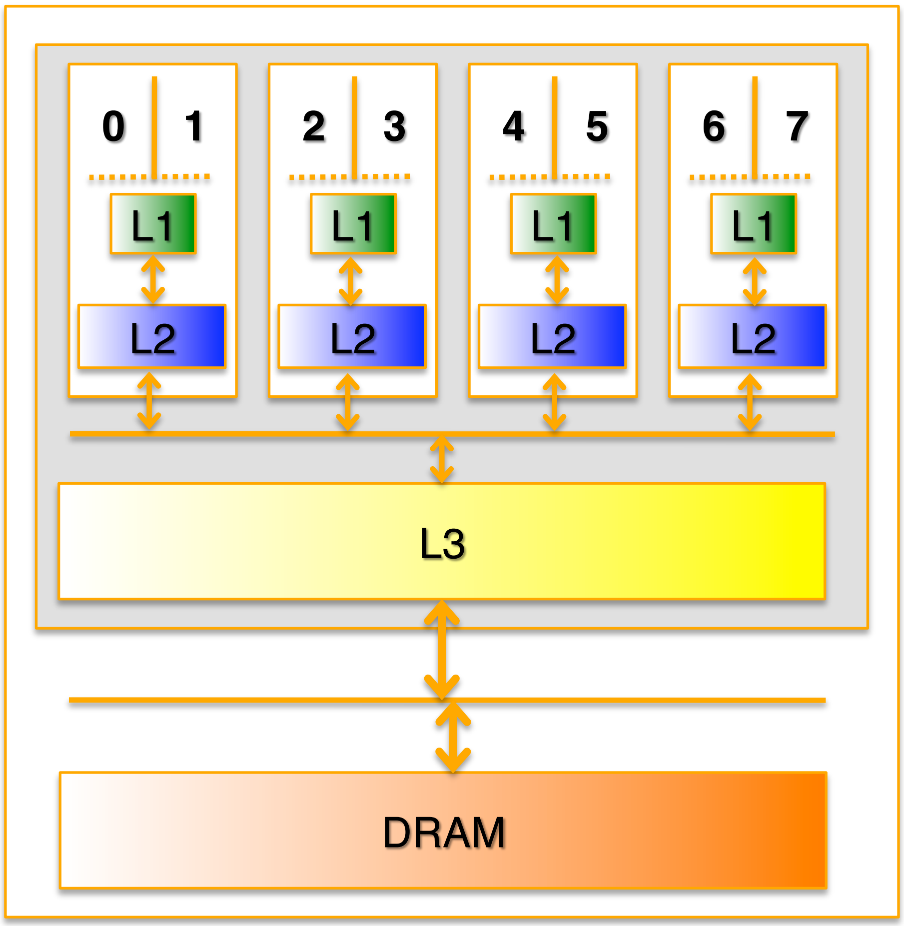
- Syscall interface
  - Regular syscall marshaled in a structure and copied into shared memory Xen-style ring buffer
  - Kernel polls for new requests and user process polls for responses
  - Notification via IPI in the future
- User level library
  - Library calls (eg. printf_async) provide descriptors that the user program can wait on
  - Single library call can contain many syscalls
  - Wait on a group of syscall descriptors

Asynchronous Remote Syscall Interface

- Advantages
  - Less contention on shared data structure
  - No cache interference between kernel and user level programs
  - Saves the cost of switching between user and kernel mode
  - Batching and reordering of system calls
- Disadvantages
  - Higher latency for a single call
  - Potentially more copying to maximize asynchrony

Evaluation Methodology

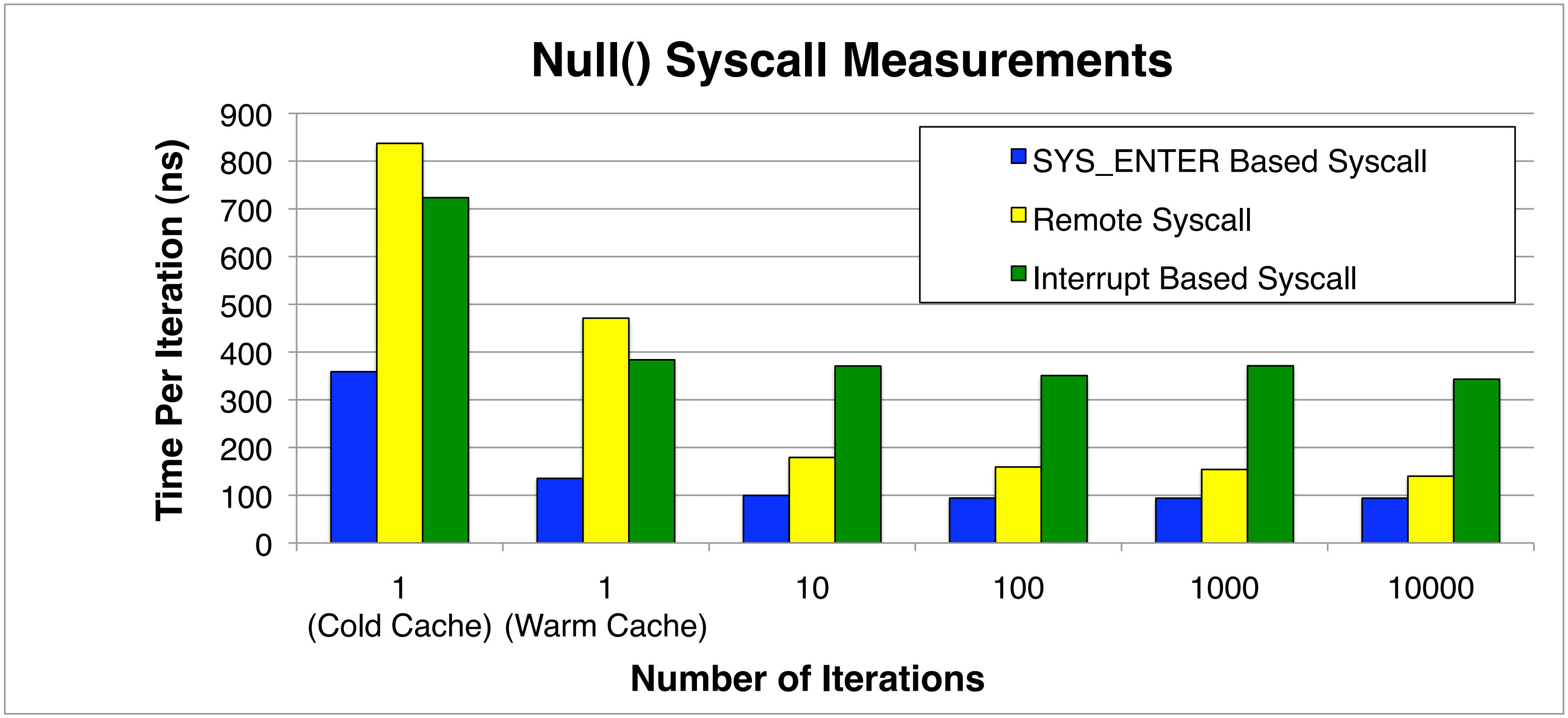
Nehalem Intel Core i7



- Compare to traditional synchronous syscalls
  - Null syscall
  - Cache contending syscall
  - User process interference
- Measure
  - Latency
  - Throughput

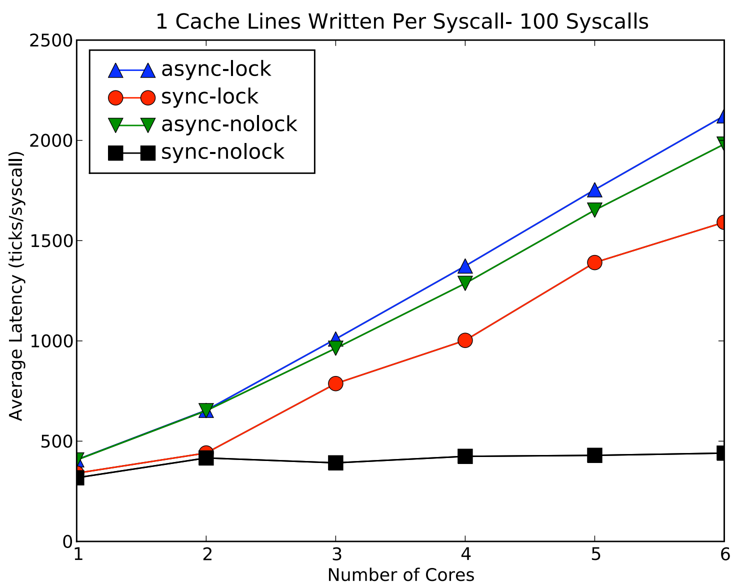
Evaluation - Null Syscall Latency

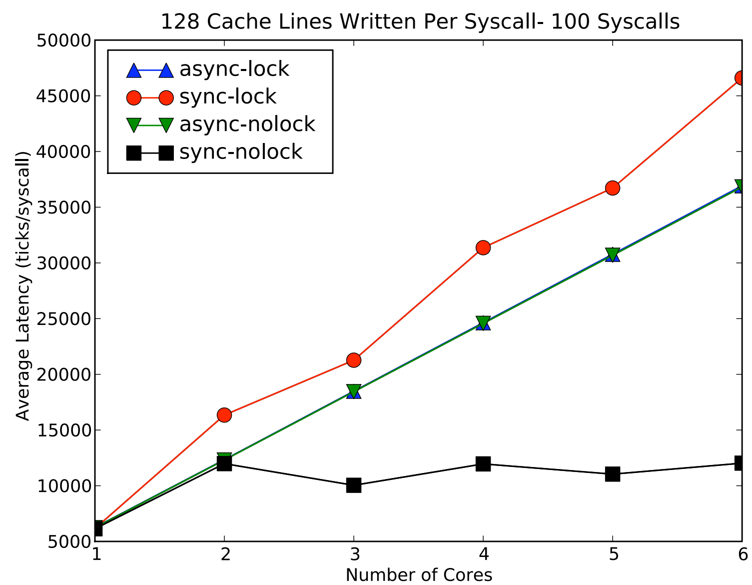
- SYSENTER 4x faster compared to interrupt-based implementation
- Our implementation is comparable



Evaluation – Cache Contending Syscall

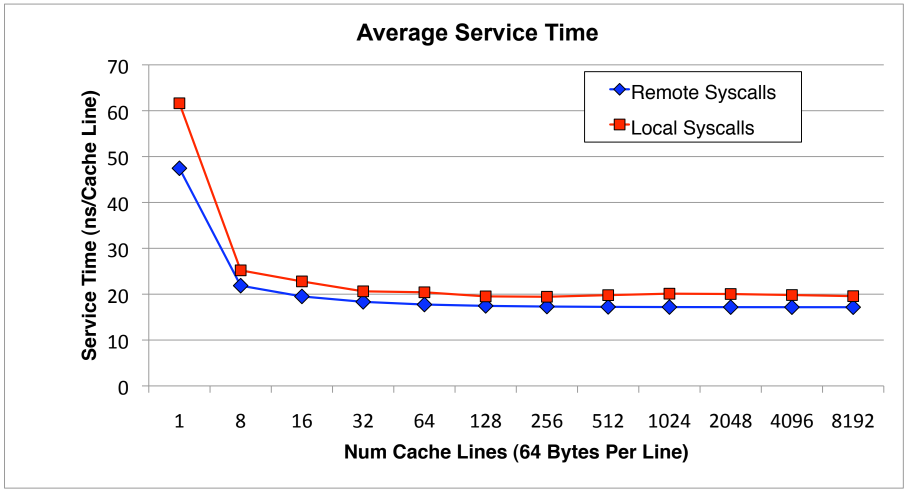
- Designed a syscall that writes to many cache lines to investigate cache contention
- Simulates a kernel intensive workload (e.g. file system)

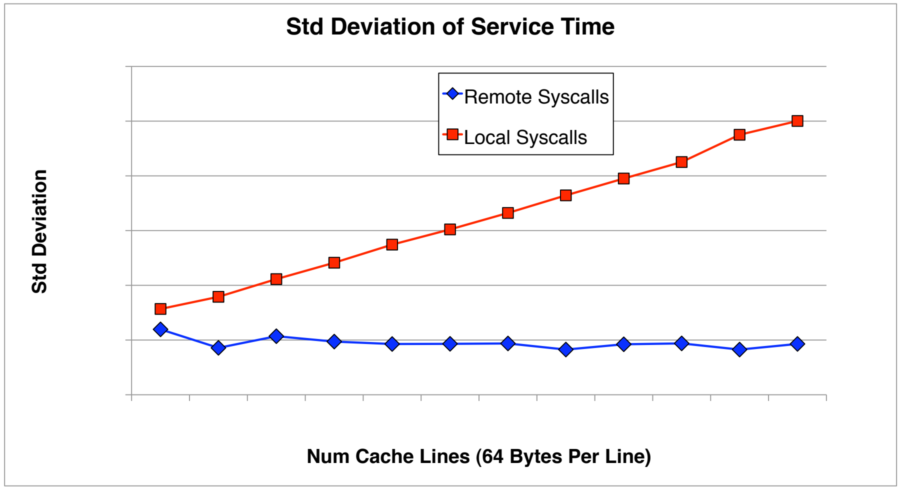




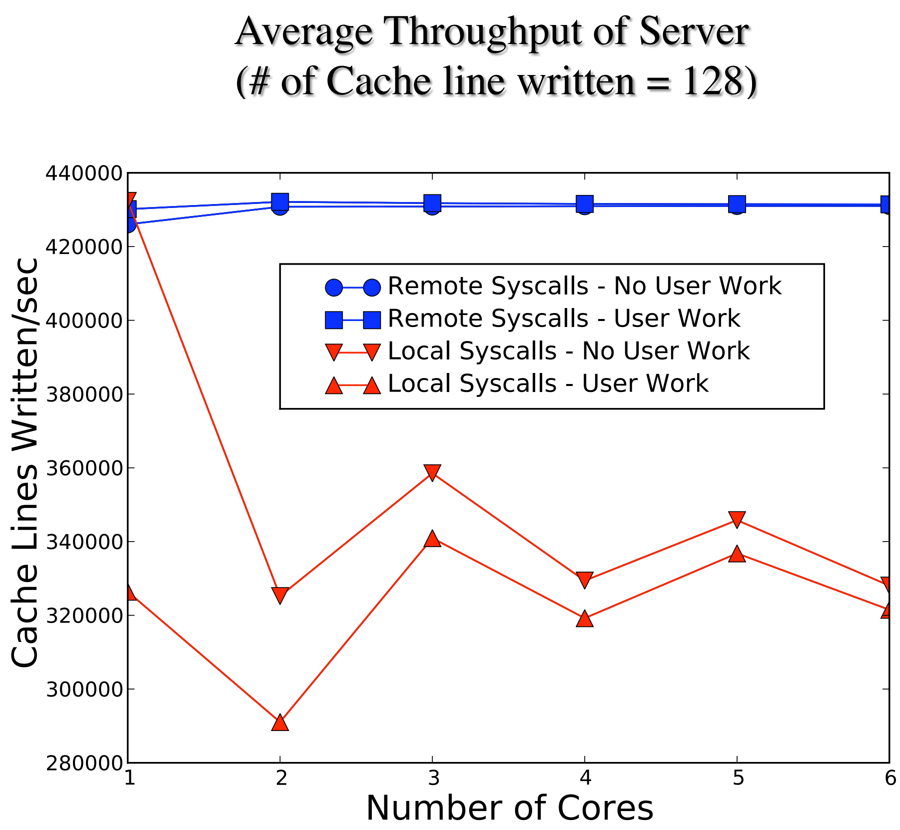
Cache Contending Syscall Internals

- What does it cost to run the cache contending syscall? Local vs. Remote
- Expected: Poor performance with multiple cores servicing kernel calls due to cache contention
- Surprise: average service time comparable





Evaluation – Throughput Comparison



- Remote locked syscalls generally have higher throughput
- Remote syscalls do not interfere with user progress

Conclusions

- Effect of cache contention was not as significant as initially thought
  - Cache contending syscall may not be representative of real workload
- Cost of code shipping may be higher than the cost of context switching and cache contentions
- Kernel processing on a remote core allows user processes to make more progress

Future Work

- Profiling different stages of both asynchronous remote syscall and synchronous syscall
- Performance counters for cache misses and other specific events
- Macrobenchmarks and a real kernel workload (file system, network stack, etc)
- Experiment with different architectures
  - More cores
  - No globally shared L3 cache
- Asynchronous notification through Interprocessor Interrupts
- Multiple kernel cores and load balancing issues

Questions?

This slide is intentionally left blank

- This slide is intentionally left blank