Abstractions for Scalable Operating Systems on Manycore Architectures

Why a New OS?
- Problems with current OSs:
  - Interference with parallel applications
  - Scalability as the number of cores increases
  - How to preserve these gains:
    - Asymmetrically structured OS
      - Increases scalability
      - Gets the kernel off userspace’s cores
      - Guaranteed partitioned resources (QoS)
    - Parallel applications are more sensitive to dynamically changing resources
      - Enables predictable application performance
      - Increases isolation between processes
    - Changes to the traditional process abstraction
      - No longer a single thread in a virtual processor
      - Multiple cores ‘owned’ by a single process
      - All cores gang scheduled
      - Information exposed up, requests sent down
    - Private Memory Ranges
      - Per core / per context virtual memory mappings
      - Enables fast page remapping
    - Eases data parallel application development

Asymmetrically Structured OS
- Why do we want to structure the OS asymmetrically?
  - Increase per core cache locality
  - Decrease cross core lock contention
  - Limits kernel interference with applications
  - Asymmetric Control
    - Manages what processes run where
    - Eliminates need for per core run queues
    - Asynchronous System Calls
    - Syscalls services asynchronously / remotely
    - Communication done via message passing

Guaranteed, Partitioned Resources
- Problem: Parallel applications are very sensitive to dynamic changes in underlying resource allocations
- Solution:
  - Resources partitioned amongst processes based on explicit requests
  - Processes scheduled based on meeting resource guarantees (QoS)
  - Resources include ‘discrete’ resources and ‘rate-based’ resources
    - Discrete: cores, physical memory pages, cache, etc.
    - Rate-based: memory / network / disk BW, etc.
  - Resource guarantees enforced either in hardware or in software in the Partition Mechanism Layer

Process Model
- Traditional 1:1 Process
  - User threads map 1:1 onto underlying kernel threads scheduled by the kernel
  - Provides scalability advantages over traditional process models
  - No mapping of user-level threads to kernel threads
    - (the kernel is completely event-based)
  - No per-core run queues
  - Provides richer set of resource guarantees to processes
  - Kernel exposes more information about resources provided by the system
- Multi-Cored Process
  - User threads scheduled in user-space
  - Cores allocated by kernel for users use
  - Allows multiple contexts per process, but kernel manages them as a unit
  - All cores granted to a process are gang scheduled
  - Processes can have hints for co-scheduling with other processes
  - Blocking system calls and interrupts don’t limit user level processing
  - Can direct interrupts to designated interrupt handling cores
  - Asynchronous I/O interfaces notifying user-space when threads block
  - Provides means for out of band processing of time-critical events
  - Always runnable, not gang-scheduled
  - Examples: UI events, TCP acks, etc.
  - Supports traditional single core-processes without guarantees as well as multi-core processes with lots of resource guarantees.

Private Memory Ranges
- Reserve range of addresses in an Address Space for processing per-context (or per-core) private data
- Logically the same address space, but specific ranges of the virtual address space are mapped to different physical pages
  - Most data is shared (e.g. file descriptors, security properties)
  - But not everything (e.g. data to be processed in SIMD fashion)
- Similar to Corey’s Address Ranges
- Process with
  - Traditional Process
    - Page Table
      - PPN 1
        - PPN 2
          - PPN 3
            - PPN 4
  - Private Memory Ranges
    - Page Table
      - PPN M
        - PPN N
          - PPN P

implementation features:
- Kernel includes a slab based memory allocator
- Use remote syscall server to handle unimplemented syscalls
- Implementation Features:
  - Kernel includes a slab based memory allocator
  - Page coloring support for cache partitioning
  - NE2000 and Realtek 8111D network driver support
  - Preliminary TCP/UDP stack using LWIP
  - Arbitrary routing of interrupts using the x86 IO APIC
  - Asynchronous remote system calls

Implementation Status
- Prototype implementation running on 2 platforms
  - x86: on QEMU / KVM and our 8-core Nehalem test machines
  - Sparc: on RAMP software simulator / FPGA hardware simulator
- Compiled with support for applications written using newlib
- Not all syscalls implemented natively
- Use remote syscall server to handle unimplemented syscalls

Acknowledgements
- From the UC Berkeley Parallel Computing Lab:
  - Ben Hindman, Juan Colmenares, Sarah Bird, Heidi Pan, Zach Anderson, Andrew Waterman, Krste Asanovic
- From Lawrence Berkeley National Labs:
  - Eric Roman, Steven Hofmeyr, John Shalf, Costin Iancu, Kathy Yelick

Research supported by Microsoft (Award #024263 ) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).