Operating Systems as Parallel Pipelines

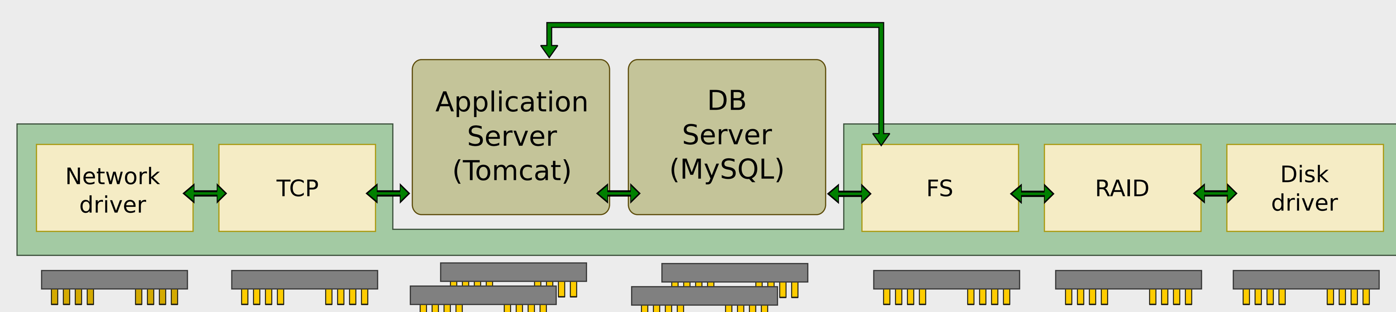
Anton Burtsev, John Regehr
Flux Research Group

Motivation

- Operating systems need to support multicores
    - Bunch of unutilized parallelism

- Pipelined parallelism is practical
    - Simple way to utilize multiple cores
    - Feasible changes to the existing software stack

- OS is structured as a set of pipelines
    - Multiple stages run different cores in parallel
    - Connected with asynchronous IPC



Pipeline example: a seven-stage pipeline of a TCP-W benchmark, which simulates a typical business oriented transactional web server solution.

Advantages

Controlled sharing:
- Data is communicated between neighboring stages only by IPC
    - Simple memory structures
    - Small TLBs = cheap memory operations [Corey, OSDI'09]

Simple synchronization:
- Only stage-to-stage IPC synchronization
    - Plus limited application-specific synchronization when several cores run the same stage

Efficient data movement:
- Explicit data path
    - Memory hierarchy is a communication medium

Spatial locality:
- Better I-cache utilization (Cohort scheduling [USENIX'02])
- Uninterruptable execution

Heterogeneous processing:
- Explicit mapping of stages to heterogeneous cores, and heterogeneous I/O paths

Design Issues

Code re-structuring:
- Linux -- modular kernel
    - Multiple subsystems -- candidates for pipeline stages
    - Connected with simple queues

Pipeline performance metrics:
- No current means to reason about pipeline performance
    - Existing CPU-based profiling methods are helpless
        - CPU is not always a bottleneck
        - Lots of wait cycles
            - I/O, memory starvation, synchronization overheads

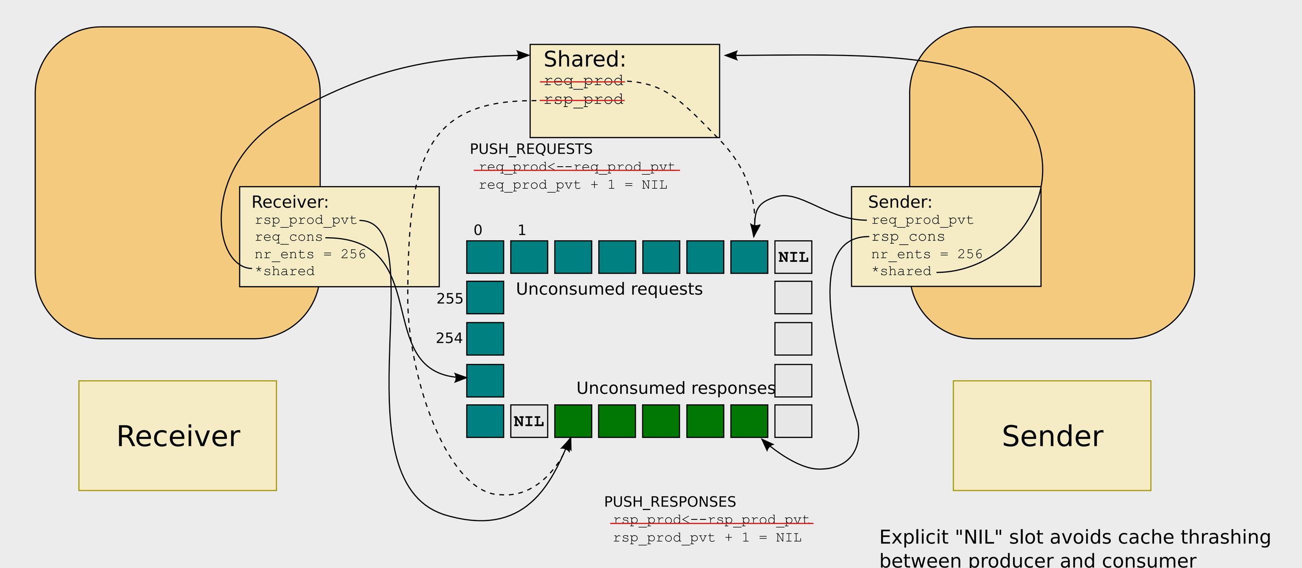
- Performance is determined by the
    - Slowest stage
    - Dependency distance of the critical loop

- Several metrics define delay and throughput
    - Length of receive queues
    - Stage length

- Queues should become visible to the scheduler
    - Contain valuable performance information
        - Long queue = bottleneck   - Empty queue = stall

Pipeline scheduling:
- Optimal stage placement
    - Topology of the memory hierarchy
    - Number of cores per stage
- Optimal stage length
    - If the data fits the cache, should stages share the core?
    - Stage fusion?

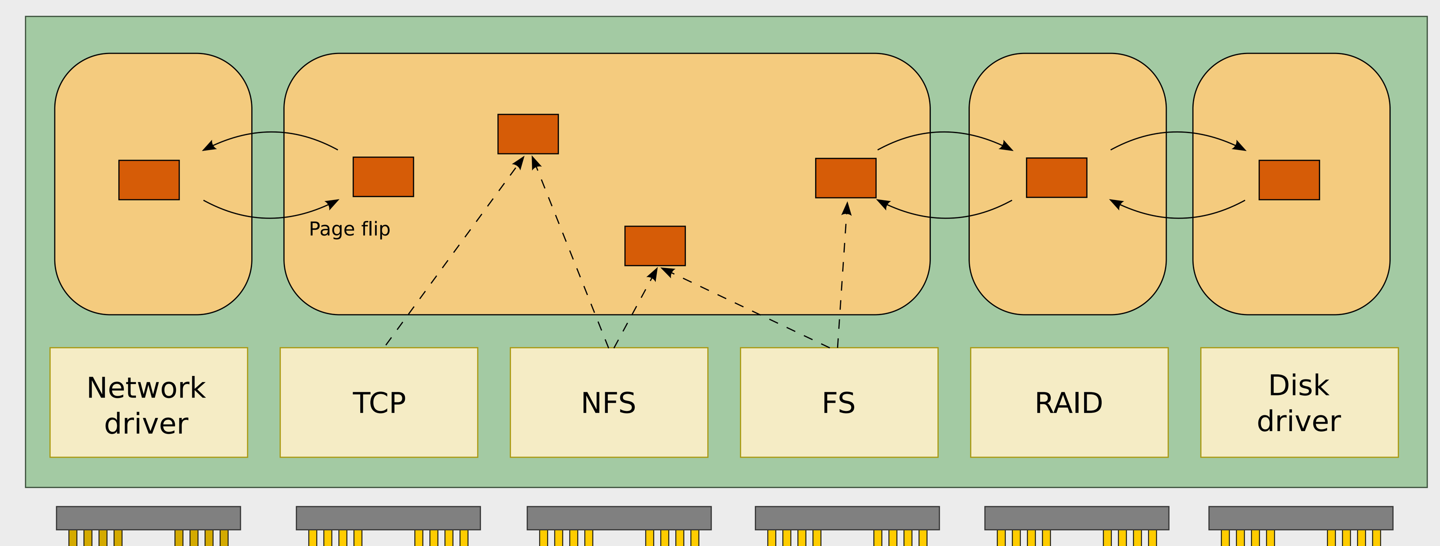
Cross-stage communication:
- Asynchronous
- Lock free
- Zero-copy
- Exchanges pointers to data
- Avoids heavyweight inter-core interrupts for notifications
    - Predominantly polling



IPC: Xen-like, single sender, unidirectional, lock-free producer-consumer ring, enhanced with FastForward [PPoPP'08] tricks: explicit empty slot ("NIL"), compulsary cache miss avoidance, flow control (temporal slipping), and pre-fetching or cache-to-cache push.

- Same primitive for user-level to kernel (no system calls)
    - Horisontal kernel invocation
        - Higher throughput
    - Needs address translation across user address spaces
        - Buffer-based memory management (Beltway [Infocom'08])

Memory management:



Memory banks: Network stack, NFS, and FS stages share memory due to complex cross-stage interfaces. Other components flip pages to preserve memory size invariant.

- Controlled sharing
    - Single-address space, but
        - Not shared
        - Not globally visible

- Page flipping
    - Simple memory management (due to memory size invariant)
    - May be prohibitively slow

- Buffer size estimation (to avoid dynamic buffer allocations)
- Global memory manager reclaims pages from private banks

Status

We are in the process of estimating performance benefits of pipelining and figuring out the key components of the system. We plan to instrument queues in the Linux kernel to identify possible pipeline bottlenecks. Even if we fail do provide pipelined architecture, we infer rules for reasoning about Linux perform-ance on a multicore platform. We see more open questions than we expected.