Analyzing the Effectiveness of Multicore Scheduling Using Performance Counters

Stephen Ziemba
Gautam Upadhyaya
Vijay Pai

Purdue University
Scheduler Goals

• Maximize CPU utilization
• Ensure access to CPU(s) is fair
• Minimize response time for real-time tasks
• Balance tasks evenly across cores
Scheduler Challenges

- Optimal scheduling is NP-complete
- Maximizing per task performance
  - Time slice, environment, etc. impact task IPC
- Number of cores increasing
  - Task balancing based on number of tasks or weight no longer sufficient
  - Parallelization of programs will likely increase inter-thread dependence
Scheduler Limitations

• Limited feedback
  • Task behavior & interaction

• Full environment not utilized
  • Thrashing difficult to pin down (which tasks)
  • Task performance based on previous task
  • Execution behavior of other cores

• Interrupt impact on performance

• Lock contention (especially the BKL & RQ)
Our Contributions

• Modified O(1) Linux scheduler to collect performance counter data
• Data shows that
  • Task performance is determined by environment
  • It can be beneficial to leave a CPU idle
  • Lock contention still plays a notable negative role in SMP scheduling
  • All CPU performance is not equal
  • Focus on work done per unit time to yield better throughput
Outline

• Scheduler Background
  • What, why & how

• Performance Counters
  • What & why

• Analysis
  • By criteria
  • Simple modification

• Conclusion
Introduction

- What limits thread behavior?
- What are bottlenecks in parallel scheduling?
- Is it possible to track thread behavior?
- What is best scheduling metric?
  - Fairness, Performance, etc.
- Is overhead of analysis too large?
- Will feedback be useful?
Scheduler Requirements

- Low overhead
- Scale with number of runnable threads
- Handle SMP
- Minimize task queue time, especially for real-time processes
- Prevent task starvation
Features of the O(1) Linux Scheduler

- Main Linux scheduler since 2.6 (until 2.6.23)
- Per CPU run queues for task selection
- Better SMP support
  - Promotes reuse of tasks on same CPU
  - No “big” scheduler lock (replaced with per RQ)
- Task priorities & preemption
- Dynamic priority based on task behavior
- Active load balancing
O(1) Linux Scheduler Runqueue

- Highest priority level is selected via a bitmap
- Non-empty levels have circular list of tasks
  - Tasks of same priority run in RR fashion

![Image of CPU-X Expired runqueue and CPU-X Active runqueue with priority levels and task priority FIFO lists.](http://www.ibm.com/developerworks/linux/library/l-scheduler/)
Tasks under the O(1) Linux Scheduler

- Tasks are called threads
  - Storage – Per thread structure
    - Contains parent, process, etc. information
    - Share a common signal struct with process
  - Priority – Based on thread behavior and nice value
    - Threads using less CPU time are given priority bonuses on recalculation. CPU bound threads are penalized
  - Time slice – Determined by priority
Balancing under the O(1) Linux Scheduler

- Load balancing is responsible for moving threads between cores
  - Types of load balancing
    - Active balance – Via 200 ms timer
    - Idle balance – When core goes idle
  - Migration occurs from busiest to calling core
  - Only movable threads are migrated
    - A thread determined to be “cache hot” is not moved
      - This is ignored if balancing fails too many times
  - Requires busiest CPU's run queue lock
Performance Counters

- Architecture level counters which OS can program to count specific events
- Limited in number
- Vary between architectures
  - Number
  - Size
  - Capability
Performance Counter Limitations

- Unable to interrupt (requires read & test)
- Accessible through local core only
- Event set varies between architectures
- Coherency events not fully countable
- Not enough counters
- Counters must be read serially
Modifications to the O(1) Linux Scheduler

- Data collected via series of hooks
  - Implemented as function pointers
- Task hook
  - Initialization and setup
- Sample hook
  - Record sample & calculation
- Cleanup hook
  - Cleanup for next call
Methodology

- Apache Web Server
  - SPECweb2005 Support workload
  - PHP-FCGI for dynamic content
  - Mix of PHP and http processing threads
- Test Platform
  - 2 dual-core 2.2 GHz Opteron
- Performance Counters
  - Retired Instructions
  - L2 Cache Data Misses
  - L2 cache Instruction Misses
Results – Per CPU Overall

- CPU behavior is not as uniform as expected
  - CPU 3 IPC is ~70% of CPU 0-2
  - CPU 3 has best D-miss rate and average I-miss rate
- Counter Intuitive behavior
  - Overall metrics tell little
  - Deeper analysis necessary
Results – Task Type Schedules

- Thread behavior varies by type
  - 95% of PHP threads schedule on all cores
  - Cores see 37 (min)-59 (max)% of http threads
- Monotonic behavior
  - PHP decreases
  - Http increases
- Nonuniform environment
Results – Task Type Time

- CPU time varies by task type
  - Time for PHP/http monotonic

- Performance
  - Worst gets most http time
  - Others get 25-35% less
    - 10% time no IPC impact?
Implications – Per Core & Generic Analysis

- Task characteristics vary wildly
  - Can be grouped
    - Behavior
    - Process
- Generic characterization not plausible
  - Requires both schedule count and CPU time
  - Thread behavior within same group must be similar
- Provides limited insight in core performance
Results – Core Distribution

- Task behavior has two distinct groups
  - ~25% threads have IPC more than 1 $\sigma$ from $\mu$
  - High/low performing groups should not be mixed
- IPC has higher $\sigma$
  - No clear tie to cache behavior
Results – Single PHP Thread Behavior

- Thread performance differs across cores
  - Performance impacted by environment
- Per core behavior differs on threads
  - Migration to a core does not guarantee performance
  - High as 10x difference
- IPC follows miss rate
Implications – Individual & Distribution Analysis

- Breaking down by process necessary
  - Generalized variance too high
    - Individual group/thread characteristic lost
  - No clear metric to schedule with
    - No way of determining why a thread behaved good/bad
    - Mean IPC & cache behavior is good for benchmarking, but does not explain what decisions should be made

- CPU environment affects performance
  - Thread behavior is volatile
  - Longer running threads have higher IPC
Results – PHP Performance

- Context time impacts performance
  - 51% IPC performance vs. average for <1 ms
  - 176% IPC for >5 ms
- Thread warm up needs further analysis
  - 2-3 ms worse 1-2 ms
  - >6 ms worse 5-6 ms
Results – HTTP Performance

- Previous thread impacts performance of thread
  - HTTP improves performance
    - 7-10% IPC improvement
    - 7.5-10% lower I-miss
    - 60-70% lower D-miss
  - PHP lowers performance
    - 45% IPC for long running
    - 62% higher I-miss
    - 570% higher D-miss
Results – Performance Break Down cont.

- Thread run time increases due to lower IPC
  - Threads take 200% time following long PHP
  - Threads take ~17% time with ≥5 previous http

- CPU time is “wasted” getting less work done per unit time
  - .16 ms – long PHP
  - .028 ms – ≥5 http versus average
Implications – Process Breakdown

- Thread time can not always be extended
  - Thread may yield
  - Page fault
  - Higher priority thread becomes schedulable

- Running specific tasks sequentially is hard
  - Balancer may move tasks to different cores
  - Threads may sleep or die

- Running groups sequentially is easier but ... 
  - Can promote unfairness
  - Increase overhead
Results – Isolating HTTP Threads

- Performance penalty
  - CPU 3 penalty order of magnitude larger
  - Penalty decreases slightly from CPU 0 to CPU 2

- Queuing issue
  - Spike in http requests results in long run queue

- More idle CPU time
  - 26-30% http
  - 4% PHP
Implications – Not all CPUs are equal

- Lock contention gives priority to lower CPUs
- Interrupts are transparent to the thread
  - Time not attributed to thread, but cache is modified
- Leaving a CPU idle can be beneficial
- Balance operations lock remote run queues and can prevent remote context switching
Online Analysis Challenges

- Data collection overhead
- Storage
- Thread sleep behavior
- Thread behavior not necessarily consistent
- Environment impact requires many samples
- Data relevancy by time used in analysis
- Thrashing of shared structures of same process
- Thread uniformity in a process not guaranteed
Conclusions

- Modified O(1) Linux scheduler to collect performance counter-based feedback
- Feedback shows that execution environment (e.g., task mix, ordering, and runtime) have first order performance impact
- Scheduler may exploit this to improve IPC by isolating task types, etc.
- Future work to target online analysis, policy decisions
Questions?