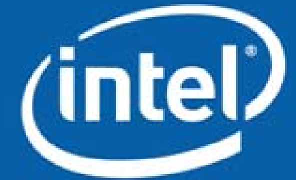


Kernel optimization for enterprise workload

Software and Solutions Group

Open Source Technology Center

Ken Chen, Staff Software Engineer

April 24, 2006

Agenda

What is enterprise workload

OLTP Workload characterization

Kernel optimization

- Generic components
- ia64 architecture components

Future work

Summary

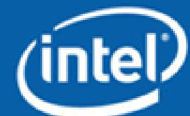
Enterprise workloads and benchmarks

Application server
- SPEC jAppServer2004

Database server
- TPC-C
- TPC-H

Web server
- SPEC web2005
- TPC-W

Many others ....



OLTP Workload characterization

Humongous memory requirement

- 64GB – 1 TB

Small / random disk I/O

- In excess of 100,000 / sec on mid class Itanium server
- Over 200,000 / sec on moderate sized ccNUMA box

High rate of interrupt

- From disk and network
- ~ 40K – 150K per second

High rate of context switch

- Synchronous I/O
- Network I/O
- SYSV IPC semaphore
- ~ 40K – 180K per second

Known optimizations

Hugetlb page for shared memory segment

Asynchronous I/O

Raw device, aka O_DIRECT

SYSV timed semaphore operation

Interrupt binding

Interrupt coalescing

Round robin process scheduling

CPU affinity for certain group of processes

Definition and Terminology

What is “performance”

- Throughput (amount of work done for a given time)
  - jobs per minute
  - operations per seconds
  - transactions per minutes

What is “performance optimization”

- Achieve higher throughput

Path length

- Number of instruction executed per unit of work

CPI

- Clock per instruction

Generic kernel optimization

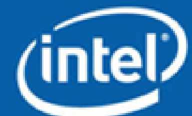
Shortening O_DIRECT path length

Optimize domain scheduler load balancing

Block layer I/O scheduler de-optimization

Block layer fast path request allocation

SCSI command structure pre-allocation



ia64 arch specific optimization

Hugetlb text

System call

- EPC based
- skip saving/restoring scratch register partition

removing dedicated TLB register for per-CPU data

Opening up high FP partition usage

Pre-fetching switch stack

Pre-fetching stack register backing store

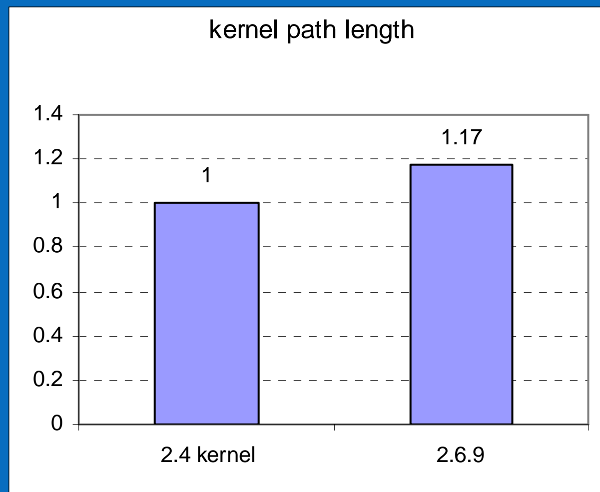
O_DIRECT bully

2.6 kernel has exceedingly long code path in O_DIRECT

- One size fit all – does everything from file system to block device
- 33 function calls before I/O submission reaches block layer
- 15 functions just to construct a bio descriptor

Perfmon shows the evidence

- 17% increase in kernel path length
- Increase not solely from O_DIRECT



sys_io_submit
io_submit_one
__aio_run_iocbs
aio_run_iocb
aio_pread
generic_file_aio_read
__generic_file_aio_read
generic_file_direct_IO
blkdev_direct_IO
__blockdev_direct_IO
filemap_write_and_wait
get_user_pages
follow_page
follow_huge_addr
find_extend_vma
find_vma
vm_normal_page
mark_page_accessed

do_direct_IO
direct_io_worker
kmem_cache_alloc
dio_get_page
get_page
submit_page_section
get_more_blocks
dio_zero_block
dio_bio_submit
dio_cleanup
finished_one_bio
dio_send_cur_page
dio_new_bio
dio_bio_add_page
dio_refill_pages

Optimize block device O_DIRECT

It's really not that complicated

- Check alignment
- Get struct page* for the user address + page reference count
- Add each page to bio descriptor
- Submit bio

Replace 15 dio* functions with a single function

- Implement either in raw device or block device
- Simple double loop

Result

- 8% reduction in kernel path length
- 2% performance gain

Domain scheduler load balancing

CPU load balancing is a complex problem to solve

- HT, SMP, ccNUMA consideration
- Priority, fairness, and maximum throughput consideration

3 load balancing points

- At recurring interval
- At scheduling to idle
- At wake-up

Imperfect wake-up balancing algorithm

- Biased towards waking CPU
- Expensive in making load balancing decision
- Worse, excessive process migration – destroy cache affinity

Scheduler load balancing – cont'

Optimization

- Reduce load balancing action in wake-up path
- Queue waking process on CPU it was previously ran

Result

- Improve process cache affinity

User CPI	-3.38%
Combined L3 Miss Ratio	-8.40%
L3 Combined Data Miss Ratio	-8.74%
L3 Combined Read Miss Ratio	-14.81%
L3 Data Read Miss Ratio	-8.91%
L3 Combined Write Miss Ratio	-7.02%

- Shorter kernel path length in wake-up
- 2.4% overall performance gain

Hugetlb Text

Same motivation for Hugetlb data segments

- Reduce ITLB pressure

A garden variety of implementations

- Completely in user space
  - Hugetlb-text loader
  - Via LD_PRELOAD
- Kernel and glibc / linker support
  - Linker puts text in a new PT_GNU_HUGEPAGE segment in elf binary
  - Kernel skips PT_GNU_HUGE_PAGE segment
  - Kernel passes hugetlb info via auxiliary vectors to dynamic loader
  - Dynamic loader uses the auxiliary vectors to
    - Create a file on hugetlbfs file system
    - Copy PT_GNU_HUGE_PAGE segment
    - mmap hugetlb pages into user processes
- Kernel loader

1.8% performance gain

Speeding up ia64 system call

Via ia64 EPC instruction

- Light-weight system call
  - gettimeofday, getpid, getppid, …
- Fall back to full blown system call for others
- Kernel entry is slightly faster than "break" instruction

Streamline system call processing (entry/exit)

- System call is like a function call
  - Follows software runtime convection on register usage
- Reduce cache line footprint at the kernel entry point
  - Avoiding storing majority of scratch register partition only touch 2 cache lines (out of total of 4)
- Reduce data stall at the exit point
  - No need to reload scratch register partition from memory

Freeing up kernel TLB

Application and OS keeps on growing

- Hardware TLB size is never big enough!

Kernel locks down 3 kernel mappings in TLB registers

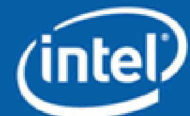
- Kernel data (16MB mapping)
- Per-CPU data (64KB mapping)
- Per-task kernel stack (16MB mapping)

Move per-CPU data from TLB register to TLB cache

- Remove references to per-CPU data out of critical stack switching
- Add special case in kernel DTLB miss handler
  - No extra execution latency due to large parallel execution resource EPIC saves the day!

Dedicate more TLB resource to application

Result – 1.6% performance gain



Opening up high FP partition usage

Not enough low cost low floating point registers (f2-f31)

- Run time software convention calls for
  - 20 preserved
  - 8 argument passing plus return register
  - 2 scratch

Application restrict FP register only to low FP partition

- Extremely expensive 2nd order effect in using high FP partition
- OS needs to save large high FP context at each context switch
  - 1536 bytes of data saved on stack
- Lazy FP loading → expensive faults on touching high FP
  - Fault + saving scratch integer register + handler + loading 1536 bytes
- Using high FP degrades performance by ~**1.7%**

Opening up high FP partition usage - cont'

Kernel optimization

- Mark high FP invalid upon system call
  both live register and in-memory flag
  →Not to trigger high FP saving at context switch

- Low level FP fault handler detect invalid in-memory FP context
  zeroing high FP in low level handler
  → Not to fault into "C" version of the handler

- All bets are off under interrupt

Recovered all the performance lose from using high FP in application

Enables compiler/application to use high FP more aggressively

Laundry list of smaller kernel optimizations

Pre-fetching kernel stacks

- At context switch, pre-fetch new/old tasks' kernel stack
  → hide memory latency, reduce kernel CPI

Pre-fetching stack register backing store

- At kernel exit path, pre-fetch user dirty register stack partition
  → hide "loadrs" latency, reduce kernel CPI

Block layer I/O scheduler de-optimization

- Remove O(n) sorting algorithm of "noop" I/O scheduler
  → reduce kernel path length

Block layer fast path request allocation

- Separating out request allocation in fast path

SCSI command structure pre-allocation

- Embed SCSI scatter-gather list inside command structure
  → reduce kernel path length

Things we tried

Asynchronous I/O group wake up

Block layer command pre-allocation

Block layer fiber channel storage driver

- Bypass entire SCSI layer

Aggressive idle CPU load balancing

Lockless wake-up queuing

Work in progress

Dynamic 3-level / 4-level page table

Elastic page table

Per-process hugetlb page size

Transparent hugetlb page size

Large page for application stack

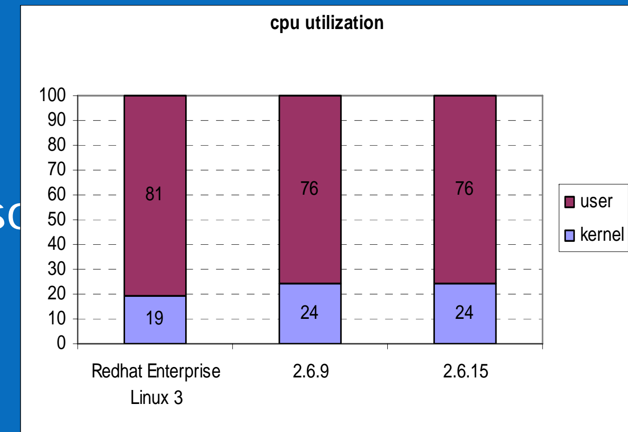
SOEMT multi-thread optimization

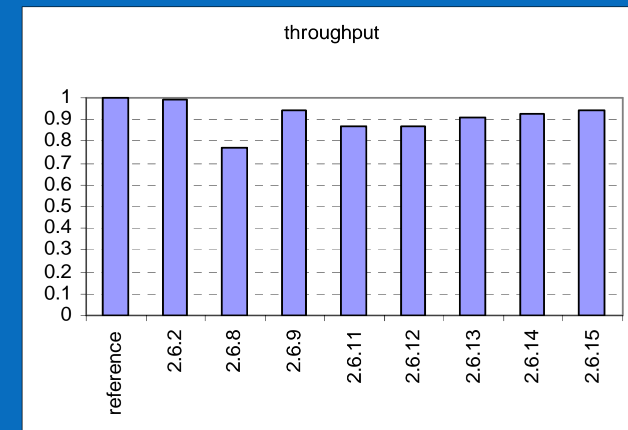
Future and challenges ahead

2.6 Kernel is regressing

Better automation tools

Better compiler support to map data EAR to sc





Legal Disclaimer

Intel® and Itanium® are registered trademarks of Intel Corporation.

Other trademarks may be claimed by their respective companies and owners

intel®
Leap ahead™

Q & A

Thank you