Low-Level Optimizations in the PowerPC/Linux Kernels

Dr. Paul Mackerras
Senior Technical Staff Member
IBM Linux Technology Center OzLabs
Canberra, Australia

paulus@samba.org
paulus@au1.ibm.com

Outline

- Introduction
  - PowerPC® architecture and implementations
  - Optimization techniques: profiling and benchmarking
- Instruction cache coherence
- Memory copying
- PTE management
- Conclusions

PowerPC Architecture

- PowerPC = “POWER Performance Computing”
  - POWER = “Performance Optimization with Enhanced RISC”
- PowerPC is a specification for an Instruction Set Architecture
  - Specifies registers, instructions, encodings, etc.
  - RISC load/store architecture
    - 32 general-purpose registers, 2 data addressing modes, fixed-length 32-bit instructions, branches do not have delay slots
  - Designed for efficient superscalar implementation
  - 64-bit architecture with a 32-bit subset
    - 32-bit mode for 32-bit processes on 64-bit implementations

PowerPC Architecture

- Caches
  - Instruction and data caches may be separate
  - Instructions provided for cache management
    - `dcbst`: Data cache block store
    - `dcbf/dcbi`: Data cache block flush/invalidate
    - `icbi`: Instruction cache block invalidate
  - Instruction cache is not required to snoop
    - Hardware does not maintain coherence with memory or Dcache
  - Data cache coherence with memory (DMA/other CPUs)
    - Maintained by hardware on desktop/server systems
    - Managed by software on embedded systems

PowerPC Architecture

- Memory management
  - Architecture specifies hashed page table structure.
    - Implemented in desktop/server CPUs
    - 4kB pages; POWER4™ also has 16MB pages
    - One hash table for all processes + kernel
    - Process “effective” addresses are translated to “virtual” addresses using segment table
      - 256MB granularity
      - In Linux: used to implement MMU “contexts”
    - Execute permission is by segment, not by page
  - Embedded processes use software-loaded TLB
    - PTE format tends to vary between implementations

PowerPC Implementations

- 32-bit implementations: IBM and Motorola
  - Desktop/server: 601, 604, 750 (Apple G3), 74xx (Apple G4)
    - Used in Apple machines, previous IBM RS/6000® machines, and embedded applications
  - Embedded: IBM 4xx series, Motorola 8xx series
- 64-bit implementations: IBM
  - POWER3™: designed for scientific/technical applications
  - RS64 family: designed for business applications
  - POWER4, POWER4+™
    - Used in current pSeries™ and iSeries machines
  - PPC970 (Apple G5)

Optimizing the kernel

- Want the kernel to go faster
- How do we know what to change?
  - Profiling: where is the kernel spending most of its time?
  - Micro-benchmarks: what operations are particularly slow?
  - Code inspection + intuition: what code looks slow?
    - often unreliable
- How do we know if our changes have done any good?
  - Profiling
  - Benchmarks
- User-space code usually dominates execution time

Profiling

- Measures the time spent in individual kernel routines
- Periodically sample next instruction pointer
  - Can use timer interrupt or other periodic interrupt
- Construct histogram
  - Map NIP to bucket index, increment bucket
- Postprocessing: map histogram buckets to routines
- Limitations
  - Sampling leads to noise in the results
  - Usually can't profile code that disables interrupts

Benchmarks

- Micro-benchmarks
  - Measure speed of individual kernel operations
  - LMBench™
    - Well-known suite written originally by Larry McVoy
- Application-level benchmarks
  - Run some specific user-level application and measure how long it takes
  - Many exist, both proprietary and open-source
  - Kernel compile
    - Ensure same source tree, same config, same target architecture and same compiler in order to be able to compare results

---

- Apple PowerBook® G3 laptop
  - 400MHz PowerPC 750™ processor (32-bit)
  - 32kB I + 32kB D L1 cache, 1MB L2 cache, 192MB RAM
- IBM® pSeries™ model 650 server
  - Eight 1.45GHz POWER4+ processors (64-bit)
  - 64kB I + 32kB D L1 cache per cpu, 1.5MB L2 cache per 2 cpus, 8GB RAM
- IBM “Walnut” embedded evaluation board
  - 200MHz PowerPC 405 processor (32-bit)
  - 16kB I + 8kB D L1 cache, 128MB RAM

Cache flushing

- Userspace expects kernel to maintain I-cache coherence for pages mapped into user processes
  - Pages mapped from files
  - Pages copied on write (private mappings, fork)
  - Zeroed pages
- Flush sequence: `flush_dcache_icache()`
  - one `dcbst` per cache line
  - one `icbi` per cache line
- Only required after page has been modified
  - by this CPU, another CPU, or DMA

Original approach

- Use `flush_page_to_ram` hook in MM subsystem
  - Called whenever user page mapping is established.
- Profile results (G3 PowerBook, kernel compile)

```
flush_dcache_icache    6763
ppc6xx_idle            2238
do_page_fault           857
copy_page               537
clear_page              523
copy_tofrom_user        356
do_no_page              231
add_hash_page           220
flush_hash_page         195
do_anonymous_page       194
```

Optimized approach

- Record I-cache state for each page
  - `PG_arch_1` bit in `page_struct` structure
  - Cleared when page is allocated
  - Set after flush is done
- Flush in update_mmu_cache
  - Only if PG_arch_1 bit is clear
- Clear PG_arch_1 in flush_dcache_page
  - Called when kernel modifies a page
- Scheme suggested by David S. Miller

Optimized approach – results

- Profile results (G3 PowerBook, kernel compile)

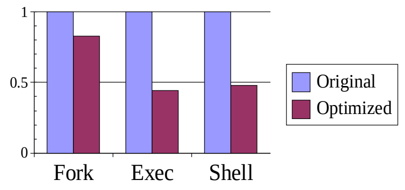
```
Routine             Original  Optimized

flush_dcache_icache    6763   2974
ppc6xx_idle            2238   2468
do_page_fault           857    667
copy_page               537    390
clear_page              523    509
copy_tofrom_user        356    299
do_no_page              231    129
add_hash_page           220     92
flush_hash_page         195    191
do_anonymous_page       194    224
```

- System time reduced to 29.9 seconds (from 46.0)
- Overall speedup (incl. user time): 5.1%

Optimized approach – results

- LMBench results (excerpt)

```
Host                 OS  Mhz fork exec sh
                             proc proc proc
--------- ------------- ---- ---- ---- ----
argo       Linux 2.5.66  400 795. 5065 23.K
argo       Linux 2.5.66  400 659. 2254 11.K
```



Further Optimization

- Why are we still flushing so much?
  - Page cache pages – flushing eliminated in steady state
  - Copy-on-write pages – may contain instructions
  - Zero pages – don't want to leak data through Icache
- Most COW or zero pages will never be executed
- Can defer flush if we have per-page execute permission
  - not in classic PowerPC architecture
  - implemented in POWER4 and in embedded CPUs
- Trap first attempt to execute from the page
  - Flush and then grant execute permission

Further optimization: results

- PPC405 embedded processor
- Kernel compile
  - System time reduced 5.9%
  - Overall time reduced 0.37%
  - Profile: `flush_dcache_icache` hits reduced from 1685 to 31
  - Floating-point emulation code is the most time-consuming

Memory copying

- To/from user process space: `copy_tofrom_user()`
  - Used for `read`, `write`, and many other system calls
- Copying pages, e.g. copy-on-write faults: `copy_page()`
- Copying kernel data structures: `memcpy()`
- `copy_page` and `copy_tofrom_user` are #4 and #6 in the profile for a kernel compile (G3 PowerBook)
  - Copying code is already well optimized for 32-bit processors.

Memory copy techniques

- Optimum memory copy routine depends on many factors.
  - Speed of unaligned accesses
  - Storage hierarchy – number of levels and latency
  - Automatic hardware prefetch mechanisms
  - Cache prefetch instructions
  - Load-to-use penalty
  - Out-of-order instruction execution capability
  - Penalty for conditional branches
  - Extended instruction sets (e.g. Altivec, MMX/SSE, VIS)

Memory copy techniques

- Optimum routine also depends on:
  - Size and alignment of regions to be copied
    - Aggressive loop unrolling may only help large copies
  - Whether the source or destination are present in cache
    - Extended instruction sets may only help if data is in cache
- Statistics for 64-bit kernel (POWER4):
  - copy_tofrom_user:
    - 84% of calls are for less than one cache lines (128 bytes)
    - 43% not 8-byte aligned
    - Most copies > 128 bytes were page-size, page-aligned
  - memcpy: 98% for < 128 bytes, 13% unaligned

POWER4-optimized memory copy

- Two separate routines.
- Small copy
  - handles arbitrary alignment, optimized for small copies.
  - Used for `memcpy()` and most `copy_tofrom_user()`
- Page copy
  - assumes cacheline aligned, 1 page copy
  - Used for `copy_page()` and page-sized and aligned `copy_tofrom_user()`
- `copy_tofrom_user()` versions have exception handling hooks.

POWER4-optimized small copy

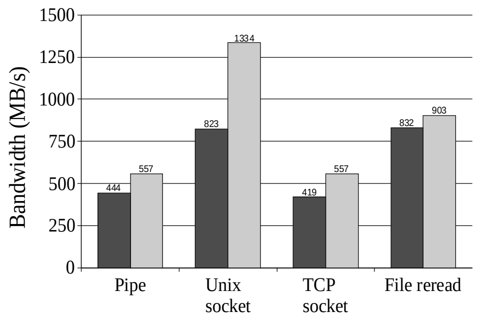
- Initially copy 1-7 bytes to get destination address 8-byte aligned
- Check whether source address is 8-byte aligned and branch to one of 2 copy loops:
  - Source aligned:
    - 2 loads, 2 stores per iteration (16 bytes)
  - Source unaligned:
    - 2 loads, 4 shifts, 2 ORs, 2 stores per iteration (16 bytes)
    - Loads and stores are 8-byte aligned
- Finally copy 0-7 bytes to finish

POWER4-optimized page copy

- POWER4 architecture details:
  - Stores go through to level 2 cache
  - L2 cache is organized as 3 interleaved banks
  - Optimum order of stores is to store into each bank in turn.
- Highest bandwidth page copy procedure:
  - Main loop copies 6 cache lines in interleaved fashion
    - 3 blocks of 6 loads, 6 stores (144 bytes)
    - Executed 5 times, followed by 6 more stores to complete the 6 cachelines
    - Iterated 5 times followed by smaller loop to finish the page
  - Uses lots of registers

Optimized memory copy: results

- Selected LMBench results:



Optimized memory copy: results

- Kernel compile:
  - System time reduced from 8.30 to 8.19 seconds (1.3%)
  - Overall time reduced by 0.13%.
- Improvement is very modest but still worthwhile.

PTE Management

- Hardware uses a hash table for storing page table entries (PTEs)
- Linux memory management (MM) subsystem uses a 3-level tree (on 64-bit architectures).
- Linux uses the hash table as a cache of PTEs (essentially a level-2 TLB).
- TLB flushing routines (`flush_tlb_page()`, `flush_tlb_range()`, `flush_tlb_mm()` etc.) have to invalidate the corresponding hash-table PTE.
- Use a bit in the Linux PTE to indicate whether a corresponding hash-table PTE exists: `_PAGE_HASHPTE`.

PTE Management Optimization

- Basic idea: invalidate hash-table PTE when the Linux PTE is changed, rather than in the TLB flushing routines.
- Reverse mapping (rmap) infrastructure gives us the necessary information to do this
  - Allows us to map from the address of a Linux PTE to virtual address and MM context that it maps.
- Further refinement: batch up the hash-table invalidations
  - Add an entry to a list when a Linux PTE is changed
  - Invalidate all the hash-table PTEs on the list on TLB flush.

Optimized PTE management: results

- 32-bit kernel (G3 PowerBook)
  - Kernel compile: no significant change in system time or overall time for either approach.
- 64-bit kernel (1.45GHz POWER4+):
  - Kernel compile: batched update vs. original implementation
    - System time reduced from 8.51 to 8.44 seconds
    - Total time reduced from 86.64 to 86.48 seconds
  - Batched update code turns out simpler and shorter than original implementation.

Conclusions

- I-cache flushing and memory copy optimizations produced worthwhile performance improvements.
- PTE management optimization gave no significant performance improvement.
- Measurement is key
  - Good ideas may not turn out to give any benefit in practice.
  - Need both micro-benchmarks and application-level benchmarks.
- Kernel profiling is a useful tool for finding profitable areas for optimization.

Legal Statement