When Poll is Better than Interrupt

Jisoo Yang Dave B. Minturn Frank Hady

Intel Corporation
Main Ideas

With ultra-low latency devices:

• The synchronous model completes an individual I/O faster despite having to poll
• IOPS scaling for increasing number of CPUs: better performance
• Model correctness for existing applications (meaning – no other SW changes needed)
Paper’s novelty

• The synchronous model completes an individual I/O faster despite having to poll
  Moneta (2010) already shows that. But:
    – Good analysis of time-waste inside a single I/O request
    – Better quantitative results
• IOPS scaling for increasing number of CPUs: better performance
• Model correctness for existing applications
  (meaning – no other SW changes needed)
/* In interrupt context */
request_fn(request_queue_t *q)
{
    struct request *req;
    while ((req = elv_next_request(q)) != NULL) {
        ...
    }
}
end_request(request_t *q){…}
/* In user-process thread */

Make_request()
{
    ...
    bio_end_io()
}
Related work

• Suggested file systems:
  – BPFS (“Better I/O Through Byte-Addressable, Persistent Memory”)
  – SCMFS (“A File System for Storage Class Memory”)

• Another block device implementation
  – Moneta (Included in our seminar):
    also spins instead of interrupt, but paper describes mostly the hardware architecture.
Prototype

• Software:
  – Linux OS
  – Dedicated block device driver (and nothing else)

• Hardware
  – PCIe SSD (emulated using DRAM)
  – NVM Express Interface (“SATA for SSD”): Linux driver (Kernel 3.3) developed by Intel…

*Moneta uses own PCIe interface implementation
Measurement techniques

• Time measure:
  – Kernel path: CPU timestamp counter – ‘rdtsc’ instruction (x86)
  – Application IOPS and latency: ‘fio’ benchmark

• Direct access to Block layer
  – No file system
  – No buffer cache

• Async: I/O scheduler off – ‘noop’
  – A note from Moneta: even that adds a few µs to each I/O request.
Latency comparison

• The important questions:
  – How fast each method completes an I/O request? (total time)
  – How much CPU time spent? (time without HW latency)
  – Async method: How much CPU made available for another process?
Why not to skip OS?

- Sync completes faster than Async OS time
- Real OS cost should add hardware-overlapping OS time
Latency Breakdown (4K xfer)

Async (interrupt-driven)

- CPU
  - System call
  - Context Switch
  - Ta' = 9.0 μs
  - Tb = 1.4 μs
  - Tu = 2.7 μs
  - Ta''
  - Return to user
  - User level prog. On same CPU taking times

- Storage Device
  - Device command
  - 4.1 μs

OS cost = Ta + Tb
= 4.9 + 1.4
= 6.3 μs

Sync (polling)

- CPU
  - System call
  - 4.4 μs
  - Return to user

- Storage Device
  - Polling
  - 2.9 μs

OS cost = 4.4 μs

In Async, only 2.7 μs is available for user app to make progress

Why the HW difference?
Less HW latency = Less SW overhead

• The same does not hold for async model.
• Works In other direction also…
  – “So what about long erase cycles? CPU blocked for a long time?”
  - Indeed. The solution could be only a hybrid one (discussion).
IOPS comparison

- Both models use 100% of each CPU.
  - Sync model needed a single thread for that.
  - Async model needed up to 8 threads per CPU.
Issues with interrupt driven I/O

*Only on very low latency devices!...

• Interrupt overhead
• Cache and TLB pollution
• CPU power state complications
Issues with interrupt driven I/O

• Interrupt overhead:
  – Not only true in general, but this fast device creates much more interrupts/sec.
  – Kernel treats HW interrupts with high priority.
  – Interrupt coalescing doesn’t improve the situation

• Cache and TLB pollution

• CPU power state complications
Issues with interrupt driven I/O

- Interrupt overhead
- Cache and TLB pollution:
  - When another thread takes place, HW resources repopulated.
  - Available time is 2.7µs (8000 cycles) – very little can be done.
  - In most cases the HW completion interrupt will take over again, so what’s the use?
- CPU power state complications
Issues with interrupt driven I/O

- Interrupt overhead
- Cache and TLB pollution
- CPU power state complications
  - Irony: when no other thread found, latency grows, and not so much power saved.
  - Sync mode prevents this.
Model correctness

• Correctness achieved when ordering of I/O requests, made by the client, is preserved.

• A general test case: A client performs an I/O call ‘A’, and then, an I/O call ‘B’. The requirement: The device should start working on ‘A’ before ‘B’.
Assumptions to be discussed:
1. Application uses blocking I/O systems call
2. Application is single threaded

A conclusion for multithreaded I/O:
in the sync model, no I/O barriers needed.
Discussion

• Although the synchronous model is correct for applications and OS to use, the async model will still perform better on large transfers or long HW stalls.
• The wanted solution: a hybrid.
Hybrid implementation (suggestions)

- An ioctl() extension for application to query whether the device is low-latency.
- Operating system CPU usage statistics: account separately for ‘spinning I/O wait’. Currently HW time is taken as OS time.
Implications on application level

• Unneeded methods in applications:
  – Non-blocking I/O calls (AIO)
  – I/O buffering
  – I/O Prefetching

• Exist in:
  – Databases
  – OS page cache
  – Disk-swap algorithms