I/O Latency Optimization with Polling

Damien Le Moal

Vault – Linux Storage and Filesystems Conference - 2017
March 22\textsuperscript{nd}, 2017
Outline

• I/O models
  – IRQ vs polling
  – NVM is coming! Is this relevant?

• Linux implementation
  – Block Layer and NVMe driver

• Evaluation Results
  – Classic polling vs Hybrid polling
  – Impact of process scheduling
  – Comparison with user level drivers

• Conclusion and next steps
I/O Models: IRQ Based Completion

Asynchronous command completion detection

- Device generated interrupts (IRQ) are asynchronous events
  - Device driver IRQ handler signals completion to waiters

![Diagram showing the process of I/O models with IRQ based completion]

1. User Process context:
   - System call (read, write, ...)
   - VFS + BIO stack
   - Device driver command submission
   - Wait (take a nap)
   - VFS + BIO stack

2. Other context:
   - IRQ handler (device driver)

3. Hardware:
   - Command execution

©2017 Western Digital Corporation or its affiliates. All rights reserved.
I/O Models: Polling Based Completion

*Synchronous command completion detection*

- Polling is a CPU driven synchronous operation
  - Active command completion detection from user process context

![Diagram of I/O Models: Polling Based Completion]

- User Process context
  - System call (read, write, ...)
  - Device driver command submission
  - Are you done?
  - VFS + BIO stack

- Other context
  - Command execution

- Hardware
  - Device driver command submission
  - VFS + BIO stack

©2017 Western Digital Corporation or its affiliates. All rights reserved.
**IRQ vs Polling**

*Trade-off CPU load for lower I/O latency*

- Polling can remove context switch (CS) overhead from I/O path
  - And more: IRQ delivery delay, IRQ handler scheduling, ...
  - But CPU spin-waits for the command completion: higher CPU load

---

**Diagram**

- **IRQ**
  - Command execution
  - Application perceived I/O latency
  - syscall -> BIO stack -> Device driver -> CS (Sleep) -> ISR (Wake) -> IRQ

- **Polling**
  - Command execution
  - Application perceived I/O latency
  - syscall -> BIO stack -> Device driver -> Are you done? -> BIO stack

---

©2017 Western Digital Corporation or its affiliates. All rights reserved.
Is It Worth It?

It depends on your system and storage device

- Main saving comes from avoiding execution context switches
  - Cost can vary, but typically 0.5~2us, or more...

- Latency relative improvements compared to IRQ depend on the typical command execution time of the device
  - For disk drives with milliseconds or more command service time, polling makes no sense

- Clearly polling becomes very interesting for very fast Flash SSDs and emerging NVM devices
  - Device access latency near or lower than context switch cost
NVM is Coming! Is This Relevant?

Vast majority of applications are not ready for NVM

- The vast majority of applications deployed today are not ready for NVM
  - Relying on known block I/O interface / POSIX system call for data management
  - Switching to NVM puts the burden of data integrity management on the application
    - No file system in between “I/O” operations (memory copies) and the storage medium
  - Block based interface likely to be present anyway
    - Memory cell error management

- PCI Express devices are getting really fast
  - Micro second order access speeds
  - Works well with current early “slow” NVM medium
    - DRAM-like performance will need more time

- Optimizing for block devices is still very relevant
  - Other kernel components benefit too
Linux Block I/O Polling Implementation

Block layer

- Implemented by `blk_mq_poll`
  - `block-mq` enabled devices only
  - Device queue flagged with “poll enabled”
    - Can be controlled through sysfs
    - Enabled by default for devices supporting it, e.g. NVMe

- Polling is tried for any block I/O belonging to a high-priority I/O context (IOCB_HIPRI)
  - For applications, set only for `preadv2/pwritev2` with RWF_HIPRI flag
  - Not related to `ioprio_set`!

```c
static ssize_t do_iter_readv_writev(struct file *filp, struct iov_iter *iter,
    loff_t *ppos, int type, int flags)
{
    struct kiocb kiocb;
    ...
    init_sync_kiocb(&kiocb, filp);
    if (flags & RWF_HIPRI)
        kiocb.ki_flags |= IOCB_HIPRI;
    ...
    if (type == READ)
        ret = call_read_iter(filp, &kiocb, iter);
    else
        ret = call_write_iter(filp, &kiocb, iter);
    ...
    return ret;
}
```
Linux Block I/O Polling Implementation

**NVMe driver**

- Currently, only NVMe supports I/O polling
  - *poll* block-mq device operation
- Polling is done on the completion queue of the hardware queue context assigned to the calling CPU
  - Does not impact other queues assigned to different CPUs
  - Catches all command completions until the command being polled completes, including low priority commands
- **IRFs** are **NOT** disabled
  - ISR is still executed as the device signals completions
    - But ISR sees a completion slot already processed

```c
static int nvme_poll(struct blk_mq_hw_ctx *hctx, unsigned int tag)
{
    struct nvme_queue *nvmeq = hctx->driver_data;
    if (nvme_cqe_valid(nvmeq, nvmeq->cq_head, nvmeq->cq_phase)) {
        spin_lock_irq(&nvmeq->q_lock);
        __nvme_process_cq(nvmeq, &tag);
        spin_unlock_irq(&nvmeq->q_lock);
        if (tag == -1)
            return 1;
    }
    return 0;
}
```
Evaluation Environment

Fast DRAM based NVMe test device

• Latest stable kernel 4.10
  – No modifications

• Random 512B direct reads
  – Raw block device access from application
  – Synchronous read from a single process
    • Queue depth 1

• Application process tied to a CPU
  – sched_setaffinity / pthread_setaffinity_np
  – Avoids latency variation due to process/thread migration to different CPU

• Standard PC
  – Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
  – 32 GB DDR4 RAM

• DRAM based NVMe test device
  – PCI-Express Gen2 x 4 interface
Evaluation Results: Classic Polling

25% lower latency with polling
- 6us average latency with IRQ, 4.5us with polling
- 25% lower latency with polling
- 166 KIOPS vs 222 KIOPS

Slightly sharper distribution
- Lower variance

But 100% load on the polling CPU
- Only 32% CPU load with IRQ model
Improving CPU load: Hybrid Polling

Polling all the time until completion is not efficient

- If the device access time exceeds the IRQ model overhead, sleeping before the I/O completion will not hurt latency
  - But the process must be woken up before the I/O completion, with heads-up time for a context switch
Linux Block I/O Hybrid Polling Implementation

Adaptive and fixed time polling

- Controlled using the `io_poll_delay` sysfs file
  - `-1`: classic polling (default)
  - `0`: adaptive hybrid polling
  - `<time in ns>`: fixed time hybrid polling

- Implemented at the block layer level, within `blk_mq_poll` function
  - The device driver does not need to have special support
  - For the adaptive mode, polling delay (sleep time) is set to half the mean device command service time obtained with classic polling
    - Enabled once command statistics is gathered

```c
> cat /sys/block/nvme0n1/queue/io_poll_delay
-1

static unsigned long blk_mq_poll_nsecs(struct request_queue *q,  
    struct blk_mq_hw_ctx *hctx,  
    struct request *rq)
{
    ...  
    /*
    * As an optimistic guess, use half of the mean service time
    * for this type of request. We can (and should) make this smarter.
    * For instance, if the completion latencies are tight, we can
    * get closer than just half the mean. This is especially
    * important on devices where the completion latencies are longer
    * than ~10 usec.
    */
    if (req_op(rq) == REQ_OP_READ && stat[BLK_STAT_READ].nr_samples)
        ret = (stat[BLK_STAT_READ].mean + 1) / 2;
    else if (req_op(rq) == REQ_OP_WRITE && stat[BLK_STAT_WRITE].nr_samples)
        ret = (stat[BLK_STAT_WRITE].mean + 1) / 2;
    return ret;
}
```
Evaluation Results: Hybrid Polling

Adaptive hybrid polling as efficient as classic polling

- Adaptive hybrid polling results in an almost identical service time distribution as classic polling
  - Lowest latencies achieved
- Fixed time polling efficiency directly depends on the sleep time set
  - Best results with polling delay set to half the command mean service time (~2us)
  - Degraded latencies with higher polling delay (4us)
    - Delay still lower than command service time, but not enough spare time for context switch and other overhead
  - Intermediate delays (3us) fall in between classic polling and IRQ latencies
Evaluation Results: Hybrid Polling

Significant CPU load reduction without average latency degradation

- Adaptive hybrid polling gives the same average latency of 4.5us as classic polling
  - But with only 58% CPU load
  - 32% with IRQ model

- Fixed time polling allows controlling the CPU vs latency trade-off
  - IRQ like average latency and CPU load for high polling delay
    - Better use the IRQ model
  - Lower CPU loads with a small average latency degradation achieved with intermediate polling delay
Evaluation Results: Hybrid Polling Sleep Time

Better sleep time estimate needed

- Sleep time set to half mean command service time works well only for constant I/O size
  - With larger I/O sizes, sleep time increases and small I/O completion do not get caught with polling
  - Back to IRQ model performance

![512B I/O Latency Average](image)
Evaluation Results: Latency Exceedance Distribution

Beware of process scheduling!

• Classic polling is more likely to suffer from a longer latency distribution tail than hybrid polling and IRQ
  – In average, 1 in ~6,000 I/Os has a latency larger than IRQ
  – Significant difference in maximum latency with hybrid adaptive polling

• From the scheduler point of view, classic polling is a CPU intensive workload
  – Not an I/O intensive workload
  – Scheduling time slices expire and result in lower “nice” value for the process
    • Preemption, scheduling delays, etc.
  – IRQ and hybrid polling benefit from sleeping
    • “priority boost” on wake-up
Evaluation Results: More on Scheduling

**Process scheduling control matters**

- Competing CPU intensive processes can significantly degrade tail latencies with classic polling
  - Again, not an I/O intensive workload
  - In average, very long latency observed for 1 in 5000 I/Os

- Standard solutions work well
  - Application running with RT class priority (SCHED_RR) maintain (or improve) latencies observed with idle system

- Hybrid polling maintains good results without scheduling class change
  - Scheduling boost on wake up
Can We Do Better?

Software optimization

- With hardware driven (mostly) context switches overhead out of the way, further latency reduction can only be achieved with software optimizations
  - System call (VFS)
    - Optimizations introduced with kernel 4.10
  - BIO stack
    - blk-mq already very efficient
  - Device driver
    - NVMe driver submission and completion paths are very short
Kernel 4.10 Optimizations

Optimized small direct I/O accesses for block devices

- Kernel 4.10 introduced new direct I/O code for raw block device accesses
  - Optimization for small direct I/Os
    - I/O size <= 16KB
    - No DIO descriptor, on stack BIO and BIO vectors
      - No memory allocation
  - 0.3 us lower latency in average
    - Compared to kernel 4.9
  - Less variation
    - Sharper latency distributions, even with IRQ
What else?

*Not much left to work with in the kernel*

- Kernel 4.10 optimizations drastically reduced the amount of code that can be optimized
  - Not much left to work with... It is getting harder!

- More extreme approach has potential: User level drivers
  - Remove system call switch overhead
  - Simplify block I/O management
    - Almost direct access to NVMe
  - But loses POSIX interface
    - Application rewrite necessary

- Several choices available
  - Storage Performance Development Kit (SPDK)
    - [https://github.com/spdk/spdk](https://github.com/spdk/spdk)
  - libnvme (SPDK rewrite to remove DPDK dependencies)
    - [https://github.com/hgst/libnvme](https://github.com/hgst/libnvme)
  - User space NVMe Driver (unvme)
    - [https://github.com/MicronSSD/unvme](https://github.com/MicronSSD/unvme)
Evaluation Results: User Level Driver

Classic polling from application context

• Significant reduction in average latency with user level drivers
  – 3.16us in average
  • 47% lower than with IRQ
  • 30% lower than kernel classic polling
  – Very narrow distribution
  – But process scheduling, again, will significantly matter

• Classic polling tested here gives 100% CPU load
  – But adaptive and fixed time hybrid polling can be tuned per I/O at application level
Conclusion and Next Steps

More improvements possible

- Kernel based I/O polling has clear advantages for latency aware applications
  - Very fast blocks devices
    - Nonsensical on slow devices
  - Keeps legacy POSIX I/O interface
    - As opposed to pure NVM approach

- User level drivers are another solution for very low latency accesses
  - But application modifications necessary

- Going forward, more can be done
  - Polling relation to I/O priority
    - application side: ioprio_set
    - Device side: NVMe submission queue arbitration
  - blk-mq block I/O scheduling
    - Work on-going
    - Must integrate smoothly polled block I/Os to avoid gain losses
      - And what of NVMe submission queue arbitration?
  - Process scheduler awareness
    - Don’t preempt polling? Treat Polling as I/O time/sleeping?
  - Further code optimization
    - Disable useless interrupts at driver level
      - ISR prevents active polling from catching completion of some commands, or runs for nothing
    - Adaptive polling sleep time estimation improvements
      - Mean time of command service time as the sleep time works well only with same sized commands
Acknowledgement

• Jens Axboe and Christoph Hellwig for polling implementation and related code optimization
• Quingbo Wang for building a fast DRAM based NVMe test PCIe card
• Many other in the Kernel development community for their efforts to constantly improve Linux