IBM Software Lab, India

Linux Asynchronous I/O Design: Evolution & Challenges

Suparna Bhattacharya

suparna@in.ibm.com

Senior Technical Staff Member,

Linux Technology Center

IBM Systems and Technology Lab, India

Linux Kernel Developers Summit - 2002

Can you guess: How many changes (lines of code) make their way into the mainline linux kernel every day ?

Introduction to AIO

- **AIO overlaps processing with I/O Operations**
  - App can submit (batch) IO w/o waiting for completion
    - Separate calls for submission & completion indication
    - Pipeline operations for improved throughput
- **Improved utilization of CPU and devices**
  - Web servers, databases, I/O intensive applications
    - Avoid need for lots of threads, event driven model
  - Application and system performance
    - Adapt to dynamically varying loads
    - Optimize disk activity (e.g. combining/re-ordering requests)

***Food for thought**:: What makes having lots of threads a problem ?*

AIO Architecture Decisions

- **External interface (API) choices**
  - Common interface for sync & async (Example ?)
  - Unique set of interfaces for AIO
    - Can address specific requirements, e.g. batch submission
- **Alternative system design principles**
  - Sync and async share a common code path
    - e.g. sync = async + wait
  - Sync and async paths diverge as needed
    - May be tuned for different performance characteristics

Linux AIO API

- **Native Linux AIO API (libaio)**
  - io_setup, io_destroy [queue setup/teardown]
  - io_submit (e.g. IO_CMD_PREAD, IO_CMD_PWRITE)
  - io_getevents [completion status notification]
  - io_cancel
- **POSIX AIO API (glibc)**
  - aio_read/aio_write/aio_fsync
  - lio_listio
  - aio_cancel, aio_suspend, aio_return/aio_error

Linux File System IO - Recap

- **Generic file read**
  - For each page in range
    - page_cache_readahead
    - lock_page
    - aops->readpage if not uptodate
      - map blocks & issue read
    - wait till page is unlocked (indicates IO completion)
    - copy data to user buffer

Question: Can you detect other blocking points besides the ones marked above ?

- **Generic file write**
  - For each page in range
    - map (and read) blocks
    - copy data from user buffer
    - mark pages dirty
  - If (O_SYNC)
    - writeout dirty mapping pages (use radix tree)
    - sync meta-data updates
    - wait for writeback to complete on these pages

(inode sem locking, journal)

Linux File System Direct IO - Recap

- **O_DIRECT option**
  - Streams entire IO direct to BIO
    - inode sem locking, consistency wrt concurrent/buffered IO
- **Block device FS direct IO**
  - Walk user pages and the file range
    - get_user_pages (pin some user buffer pages)
    - Map blocks to disk
    - Submit io (collated)
  - Wait for completion of all submitted IO
    - DIO structure (tracks count of BIOs)
  - Post-processing for completed IO (dirty pages)

Question: Can you detect other blocking points besides the ones marked here ?

Alternate Design Models for AIO

- **Offload entire IO to thread pools**
  - User level threads (e.g. glibc implementation)
  - Kernel threads
- **Fully async state machine for every operation**
  - Series of event driven non-blocking steps
  - Map user buffers to process context indep. form
- **Hybrid approach with split phase I/O**
  - Async submission, pool of threads to wait for completion
    - Per-address space threads for user context dependencies
  - e.g. SGI KAIO

Linux AIO Evolution

- **POSIX AIO implementation in glibc**
- **SGI KAIO patches**
- **Linux 2.4 distro add on patches (RHEL, SLES)**
  - General FSAIO
- **Linux 2.6 mainline**
  - AIO Direct IO
- **Linux 2.6 external patches**
  - General FSAIO, AIO-epoll, POSIX AIO enablement
  - Syslets & threadlets (general async system calls)

Linux Kernel 2.6 AIO – Basic Infrastructure

- **Data structures**
  - IO context (ioctx)
  - IO control block (iocb)
  - Ring buffer - completion events
  - AIO workqueue
- **A few implementation issues**
  - Tricky race conditions (submit/complete/cancel paths)
  - Latency, fairness, batching, ordering
  - Resource limits and scaling
  - Process exit conditions

Quick Check: Can you identify the AIO design model used here ?

Linux 2.6 – Asynchronous Direct IO

- **IO completion step async**
  - Return -EIOCBQUEUED after all IO is submitted
    - BIO completion callback completes iocb from interrupt context when entire DIO is done
  - Workqueue for post-processing which cannot be from interrupt context
    - Optimization: mark pages dirty before IO, redirty if needed
- **Caveats**
  - Multiple potential blocking points not converted to async
    - Works in practice for special requirement of databases
  - DIO code fragile, AIO-DIO error handling messy

AIO Results – OLTP example

Configuration	Relative throughput	Page cleaner writes (%)
1 page cleaner with AIO	133	100
55 page cleaners without AIO	122	70

- Update-intensive OLTP database workload, Derived from a TPC benchmark, but in no way comparable to any TPC results
- DB2 V8, Linux 2.6.1, 2-way AMD Opteron, QLogic 2342 FC, 2 storage servers x 8 disk enclosures x 14 disks each, RAID-0 configuration, stripe size 256KB

Generalized File System AIO – Linux 2.4 patches

- **Work-to-do callback driven async state machine**
  - (Almost) fully asynchronous but complex & hard to debug
- **Separate code paths for sync and async**
  - Allow special tuning for AIO, but duplication => maintainability issues
- **Pin user buffers**
  - Avoids extra threads for completing IO in caller's context but causes inefficient utilization of TLB for small buffers
- **Per filesystem impact**
  - *Why does that matter ?*

Linux wait queue mechanism - Recap

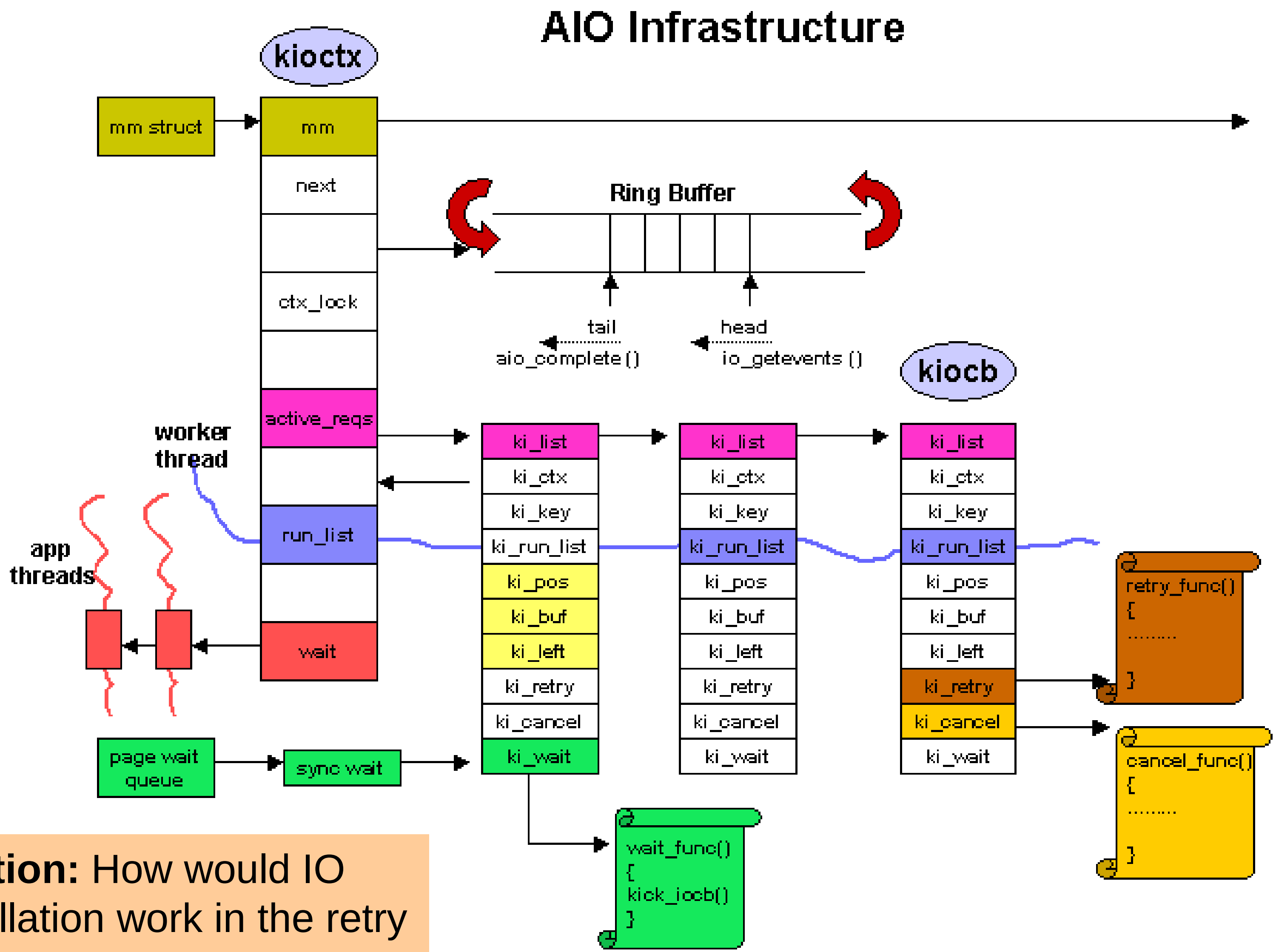
- **Basic mechanism**
  - wait_queue_head
  - wait_queue_t
    - wait_queue_function, task to wakeup
  - prepare_to_wait(), finish_wait(), wakeup()
    - Flags: TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE
  - io_schedule()
- **Hashed wait queues**
  - Filtered wakeups
  - Example: page wait queue

Question: What purpose does the wait_queue_function serve ?

Generalized File System AIO – Linux 2.6 patches

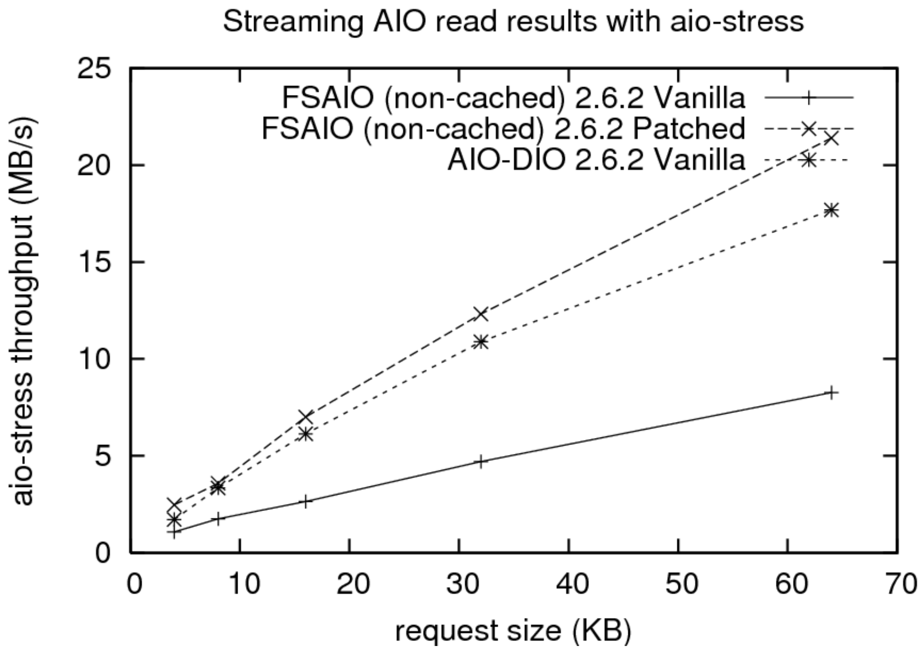
- **Retry based AIO model**
  - Convert main blocking points to retry exits in AIO context
    - Return no. of bytes completed or -EIOCBRETRY
  - Series of non-blocking iterations through an IO request
    - async wait callback schedules reissue of ***fop->aio_read/write*** with modified arguments representing the remaining IO
  - Retry threads take on caller's address space (use_mm)
- **AIO and Sync IO share a common code path**
  - AIO = Sync IO – wait + retry (vs Sync IO = AIO + wait)
    - e.g. iocb = container_of(current_wait()) in AIO context

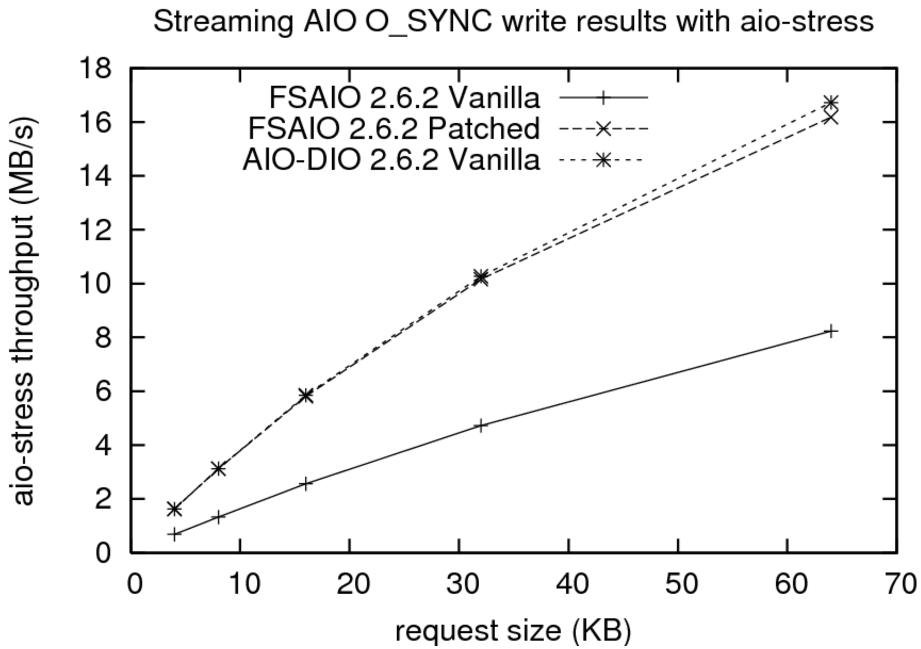
Question: Is there a pre-requisite for the retry model to be applicable ?



Question: How would IO cancellation work in the retry model ?

Filesystem AIO Results (Random read/write)





- Filesystem: Ext3, blocksize: 4KB, file : 1GB
- 4-way Pentium(tm) III, 700MHz, 512MB, AIC7896 Ultra2 SCSI
- Interesting issues: IO ordering with readahead, writeback & concurrency

Combining Network & File AIO – Linux 2.6 patches

- **Typical event loop**
  - Epoll (scalable file event polling) EPOLL_CTL_ADD/DEL
  - Socket read/write
    - O_NONBLOCK (readiness to send, available data to read)
- **Experimental**
  - AIO epoll: IO_CMD_EPOLL_WAIT
  - Simulating AIO using async poll & O_NONBLOCK retries
  - Kevent
- **Eventfd (now in mainline, 2.6.22 onwards)**

Food for thought: What makes network IO and file IO so different ? Why have so many alternatives emerged ?

Building POSIX AIO over Kernel AIO – Linux 2.6 patches

- **Signal notification**
- **lio_listio**
  - IO_CMD_GROUP
- **aio_cancel_fd**

- **AIO support for all types of file-descriptors**
  - Fallback implementation

Syslets & Threadlets: Generalized asynchronous systems calls – Linux 2.6 patches

- **“Cache miss” concept applied to threading**
  - On-demand parallelism (Only if the original context blocks)
  - Switch caller's user space context to a cache miss thread which continues user space execution without stopping
    - Spares users from setting up, sizing and feeding a thread pool
- **Threadlets (“Optional threads”)**
  - Small functions of execution
- **Syslets**
  - Small, kernel-side, scripted "syscall plugins"

“So all in one, I used to think that AIO state-machines have a long-term place within the kernel, but with syslets I think I've proven myself embarrasingly wrong =B-)”

- Ingo Molnar, Feb 2007

Food for thought: Are there real situations where the overheads matter ?

Observations

- **Many challenges beyond conversion to async**
  - API decisions, compatibility implications
  - AIO exposes scenarios and IO patterns less likely with synchronous workloads
    - Inherent concurrency, contextual assumptions
- **Shaped by real use cases that matter**
  - AIO direct IO driven by database requirements

Food for thought: Why has getting real use cases been a challenge ?

Credits & Thanks

- Benjamin LaHaise
- Zach Brown
- Badari Pulavarathy
- Chris Mason
- Andrew Morton
- Jeff Moyer
- Janet Morgan
- Kenneth Chen
- Ulrich Drepper
- Andrew Tridgell
- Laurent Vivier
- Sebestian Dugue
- Davide Libenzi
- Evgeniy Polyakov
- William Lee Irwin III
- Christoph Hellwig
- Ingo Molnar
- Stephen Tweedie
- Linus Torvalds

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document is provied ``AS IS,'' with no express or implied warranties. Use the information in this document at your own risk.