IXPUG Annual Meeting 2020

User-space thin file system coupled with an ultra-fast and low-latency IO stack as an alternative for use by database storage engines.

Jan Lisowiec Intel® Senior Member of Technical Staff
jan.lisowiec@intel.com

Legal Disclaimers

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies may require enabled hardware, software, or service activation.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Your costs and results may vary.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Why user-space file system ?

- Kernel overhead is no longer a negligible portion of I/O latency
  - The newest generation SSDs, Intel® Optane™ SSD as an example, have I/O latencies < 10 uSec
- Eliminate system call overhead
- Buffer zero-copy end-to-end
- Polling mode interfaces, avoid context switches
  - Remove bottom-half interrupt handling overhead
  - Provide better QoS, higher I/O determinism
- APIs tailored to specific application needs
  - C++ API with based on future-promise design pattern

Why database systems can benefit from user-space file system?

- DB servers “own” database files
  - Single process makes modifications to the files
  - Page cache complexity not needed
  - Simpler cache flush when only one process writes to a file
- DB servers don’t rely on file system locking
  - They implement their own locking
- DB servers do their own recovery
  - File system journaling not being used by DB engines
- DB engines typically rely on a subset of file API functions
  - open(), close(), stat(), lseek(), fsync(), fallocate(), rename(), unlink(), pread(), pwrite(), io_submit(), io_getevents()

Easily extendable

- Easier integration with various storage back ends
- Faster development
- Shorter turn-around cycles for bug fixes
- Apps layered on top of XFS/Ext4 can be easily migrated
  - Provide Posix compliant APIs
  - Apps such as DB Engines rely on a small subset of file system functionality
  - Enable key-value DBs to achieve higher throughput

Scales well with modern multi-core CPUs

- Frameworks such as SPDK can be used to build upon
  - SPDK provides block I/O layer and APIs
  - SPDK uses polling mode
  - SPDK facilitates cmd submission, completion, memory buffer allocation, etc
- Polling mode APIs not an issue on modern multi-core CPUs
  - Polling threads tie up CPU cores
  - Can be mitigated by polling threads sleep/wakeup
- Provide both buffered and unbuffered (O_DIRECT)

Geared for data consistency

- Write I/O atomicity
  - Kernel file systems can split an I/O into several smaller ones
  - SSDs guarantee write atomicity typically for 4kB I/O
  - RDBMSs can’t rely on write atomicity even if SSD provides one
  - User-space file system can guarantee write atomicity
- RDBMS DB Engines implement their own logging
  - DB Engines open table files with O_DIRECT (bypass Page Cache)
  - File system journal is not needed for DB recovery
  - Alignment restrictions can be handled by user apps

Cloud deployments

- In cloud deployments file access permissions are less critical
  - User-space file system owns a storage device
  - Storage device is not visible to the OS
  - Apps can access user-space file system via an Agent

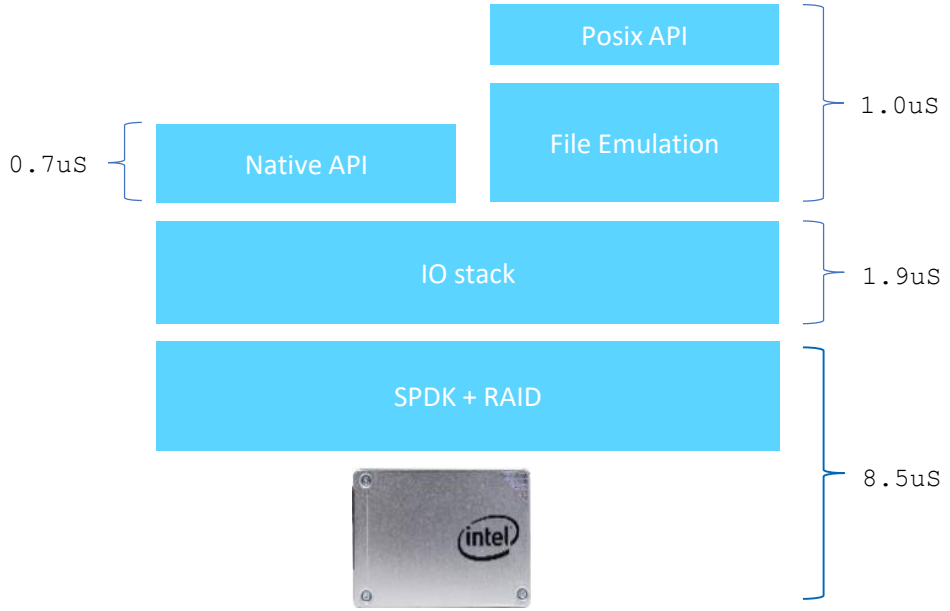
Reasons for not using FUSE

- Polling mode APIs
- Async APIs based on C++ future-promise programming pattern
- FUSE limited scalability
- Performance
    - FUSE registers itself with VFS
    - FUSE incurs system call overhead
    - FUSE module has its own submission queue – additional latency

Latency breakdown in layered architecture

End-to-end latency 10-12uS

- 4kB unbuffered write against Intel® Optane™ SSD DC P4800X
- Native API in polling mode
  - C++ Future object polled by calling thread
- Posix pwrite()
  - pwrite() blocking calling thread
  - pwrite() polls internally completion status
- Statically preallocated blocks in File Emulation



Source: "Breakthrough Performance Expands Datasets, Eliminates Bottlenecks" https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-dc-p4800x-p4801x-brief.html. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

C++ native APIs

Abstract C++ ApiBase

```
class ApiBase {
  public:
    virtual int open(const char *name, int flags, mode_t mode = S_IRUSR | S_IWUSR) = 0;
    virtual int close(int desc) = 0;
    virtual int read(int desc, char *buffer, size_t bufferSize, bool polling = true) = 0;
    virtual int write(int desc, const char *data, size_t dataSize, bool polling = true) = 0;
    virtual off_t lseek(int fd, off_t offset, int whence) = 0;
    virtual int fsync(int desc) = 0;
    virtual int stat(const char *path, struct stat *buf) = 0;
    virtual int unlink(const char *path) = 0;
    virtual int rename(const char *oldpath, const char *newpath) = 0;

    virtual int pread(int desc, char *buffer, size_t bufferSize, off_t offset, bool polling = true) = 0;
    virtual int pwrite(int desc, const char *data, size_t dataSize, off_t offset, bool polling = true) = 0;

    virtual FutureBase *read(int desc, uint64_t pos, char *buffer, size_t bufferSize, bool polling = true) = 0;
    virtual FutureBase *write(int desc, uint64_t pos, const char *data, size_t dataSize, bool polling = true) = 0;
};
```

Posix APIs (emulated APIs)

Minimum subset of APIs for InnoDB

```
int open(const char *name, int flags, mode_t mode = S_IRUSR | S_IWUSR);
int close(int fd);
uint64_t lseek(int fd, off_t pos, int whence);
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
int stat(const char *path, struct stat *buf);
int unlink(const char *pathname);
int rename(const char *oldpath, const char *newpath);
void pread(struct iocb *iocb, int fd, void *buf, size_t count, long offset);
void pwrite(struct iocb *iocb, int fd, void *buf, size_t count, long offset);
int fallocate(int fd, int mode, off_t offset, off_t len);
int io_setup(int maxevents, io_context_t *ctxp);
int io_destroy(io_context_t ctx);
int io_submit(io_context_t ctx, long nr, struct iocb *ios[]);
int io_cancel(io_context_t ctx, struct iocb *iocb, struct io_event *evt);
int io_getevents(io_context_t ctx_id, long min_nr, long nr, struct io_event *events, struct timespec *timeout);
```

pread() - polling implemetation

Pools atomic variable (no thread preemption)

```
int AsyncApi::pread(int desc, char *buffer, size_t bufferSize, off_t offset, bool polling) {
    uint64_t lba;
    uint8_t lun;
    if (getIoPosStriped(desc, static_cast<uint64_t>(offset), lba, lun) < 0)
        return -1;
    IoRqst *getRqst = IoRqst::readPool.get();
    if (polling == false) {
        . . . .
    } else {
        atomic<int> ready;
        ready = 0;
        getRqst->finalizeRead(nullptr, bufferSize,
             [&ready, buffer, bufferSize](
                 StatusCode status, const char *data, size_t dataSize) {
                 if (status == StatusCode::OK)
                     memcpy(buffer, data, dataSize);
                 ready = status == StatusCode::OK ? true : false;
             },
             lba, lun);
        if (spio->enqueue(getRqst) == false) {
            IoRqst::readPool.put(getRqst);
            return -1;
        }
        while (!ready) ; // wait for completion
    }
    ApiBase::lseek(desc, offset + bufferSize, SEEK_CUR);
    return bufferSize;
}
```

pread() - polling implemetation

Polls atomic variable (no thread preemption)

```
int AsyncApi::pread(int desc, char *buffer, size_t bufferSize, off_t offset, bool polling) {
    uint64_t lba;
    uint8_t lun;
    if (getIoPosStriped(desc, static_cast<uint64_t>(offset), lba, lun) < 0)
        return -1;
    IoRqst *getRqst = IoRqst::readPool.get();
    if (polling == false) {
        . . . .
    } else {
        atomic<int> ready;
        ready = 0;
        getRqst->finalizeRead(nullptr, bufferSize,
             [&ready, buffer, bufferSize](
                 StatusCode status, const char *data, size_t dataSize) {
                 if (status == StatusCode::OK)
                     memcpy(buffer, data, dataSize);
                 ready = status == StatusCode::OK ? true : false;
             },
             lba, lun);
        if (spio->enqueue(getRqst) == false) {
            IoRqst::readPool.put(getRqst);
            return -1;
        }
        while (!ready) ; // wait for completion
    }
    ApiBase::lseek(desc, offset + bufferSize, SEEK_CUR);
    return bufferSize;
}
```

Async write() implementation

write() returns C++ Future

```
FutureBase *AsyncApi::write(int desc, uint64_t pos, const char *data, size_t dataSize, bool polling) {
    uint64_t lba;
    uint8_t lun;
    if (getIoPosStriped(desc, pos, lba, lun) < 0)
        return 0;

    FutureBase *wfut;
    if (polling == true)
        wfut = WriteFuturePolling::writeFuturePollingPool.get();
    else
        wfut = WriteFuture::writeFuturePool.get();
    wfut->setDataSize(dataSize);

    IoRqst *writeRqst = &wfut->ioRqst;
    writeRqst->finalizeWrite(data, dataSize,
        [wfut](StatusCode status,
                const char *data, size_t dataSize) {
            wfut->signal(status, data, dataSize);
        },
        lba, lun, true);

    if (spio->enqueue(writeRqst) == false) {
        wfut->sink();
        return 0;
    }
    ApiBase::lseek(desc, pos + dataSize, SEEK_SET);
    return wfut;
}
```

write() Future implementation

get() polls atomic variable (no thread preemption)

```
int WriteFuturePolling::get(char *&data, size_t &_dataSize, unsigned long _timeout) {
    while (!state) ;
    if (opStatus)
        return -1;

    _dataSize = dataSize;
    return opStatus;
}

void WriteFuturePolling::signal(StatusCode status, const char *data, size_t _dataSize) {
    dataSize = _dataSize;
    opStatus = (status == StatusCode::OK) ? 0 : -1;
    state = 1;
}

void WriteFuturePolling::sink() {
    reset();
    WriteFuturePolling::writeFuturePollingPool.put(this);
}
```

Example of async write() usage

Single threaded execution of copy file

```
static int AsyncIoCompleteWrites(AsyncApi *api, write_futures, size_t &num_queued) {
    . . .
    for (size_t i = 0 ; i < num_queued ; i++) {
        int rc = write_futures[i]->get(data, dataSize);
        . . .
    }
    num_queued = 0;
    return 0;
}

int AsyncWriteIoTest(AsyncApi *api, size_t queue_depth, size_t num_ios) {
    . . .
    size_t num_queued = 0;
    for (size_t i = 0 ; i < num_ios ; i++) {
        write_futures[num_queued++] = api->write(fd, pos, io_buffer, io_size);
        if (i >= queue_depth)
                AsyncIoCompleteWrites(api, write_futures, num_queued);
    }
    AsyncIoCompleteWrites(api, write_futures, num_queued);
    . . .
}
```

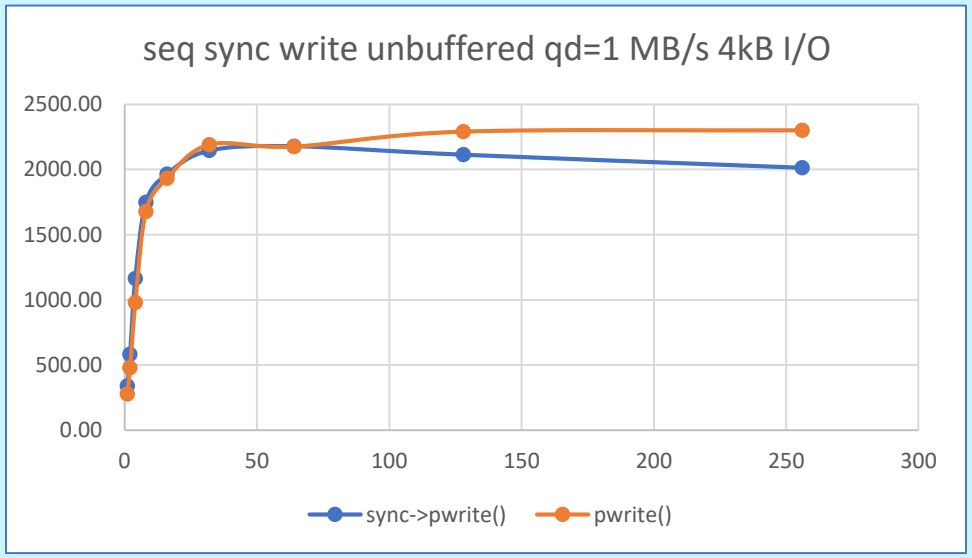
Lightweight memory/object pools

- C++ Template library
- Used across the entire stack
- Pools of preallocated and preconstructed class instances
- Provide low-contention and lock-free pools
- Can grow or shrink on demand
- In shared-nothing dedicated pools for per-device DMA buffers
- Most of the computational overhead removed from the I/O path

Benchmarks - seq write unbuffered vs. # files 4kB[1]

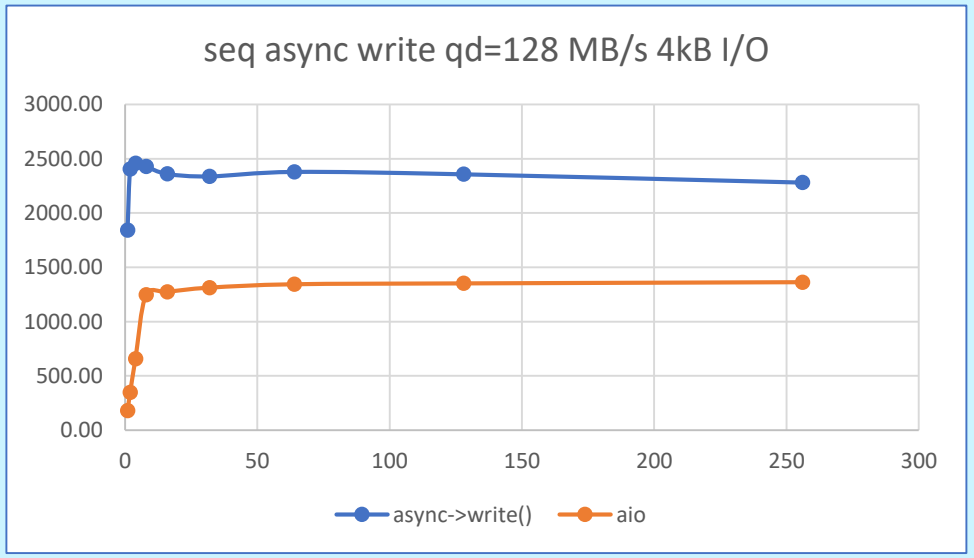
1 x Intel® Optane™ SSD DC P4800X		
Seq sync write unbuffered qd=1 MB/s 4kB I/O		
#files/threads	sync->pwrite()	pwrite()
1	340.00	278.00
2	582.00	478.00
4	1165.00	980.00
8	1747.00	1678.00
16	1965.00	1932.00
32	2145.00	2190.00
64	2178.00	2175.00
128	2113.00	2290.00
256	2013.00	2300.00

sync->pwrite() - User-space
pwrite() – Ext4
Queue depth = 1
Drive saturation @ 2.5 GB/s



1 x SSD DC P4800X		
Seq async write unbuffered qd=128 MB/s 4kB I/O		
#files/threads	async->write	aio
1	1841.00	179.00
2	2405.00	349.00
4	2458.00	658.00
8	2427.00	1245.00
16	2360.00	1273.00
32	2337.00	1312.00
64	2378.00	1343.00
128	2356.00	1351.00
256	2280.00	1362.00

async->pwrite() - User-space
aio – Ext4
Queue depth = 128
Drive saturation @ 2.5 GB/s



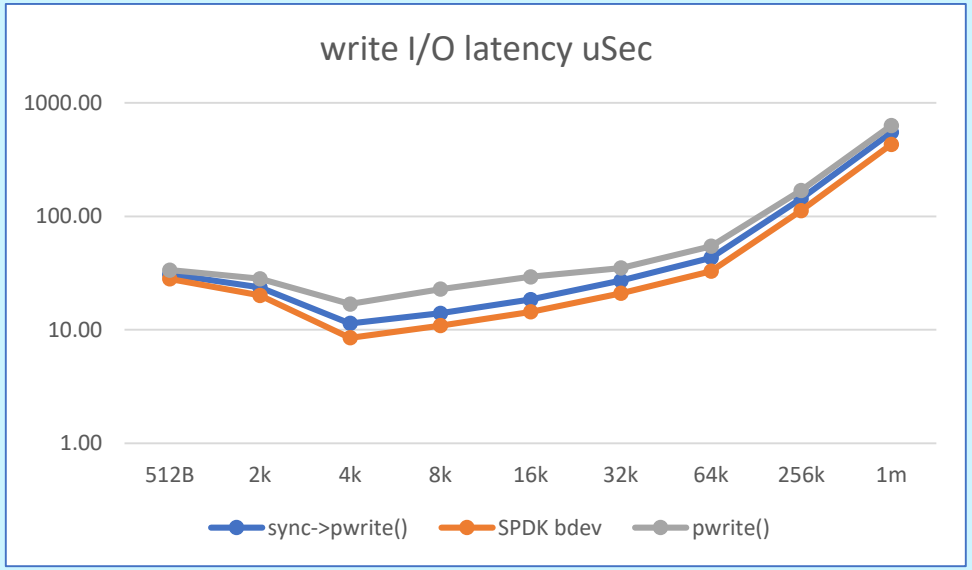
[1]See appendix for footnote 1. For more complete information on performance and benchmark results, visit www.intel.com/benchmarks

Benchmarks - latency vs. I/O size unbuffered[1]

- User-space I/O latency consistently lower
- Compared using unbuffered I/Os
  - All tests using O_DIRECT flag
  - Bypass Page Cache

1 x Intel® Optane™ SSD DC P4800X			
write I/O latency uSec			
I/O size	sync->pwrite()	SPDK bdev	pwrite()
512B	30.90	28.10	33.60
2k	23.70	20.10	28.20
4k	11.40	8.50	16.90
8k	14.00	10.90	22.80
16k	18.50	14.40	29.30
32k	27.10	21.00	35.10
64k	43.10	32.80	54.50
256k	143.50	112.50	169.40
1m	549.00	431.00	632.00

sync->pwrite() - User-space
SPDK bdev – SPDK portion
pwrite() – Ext4
Queue depth = 1
P4800X latency < 10uS

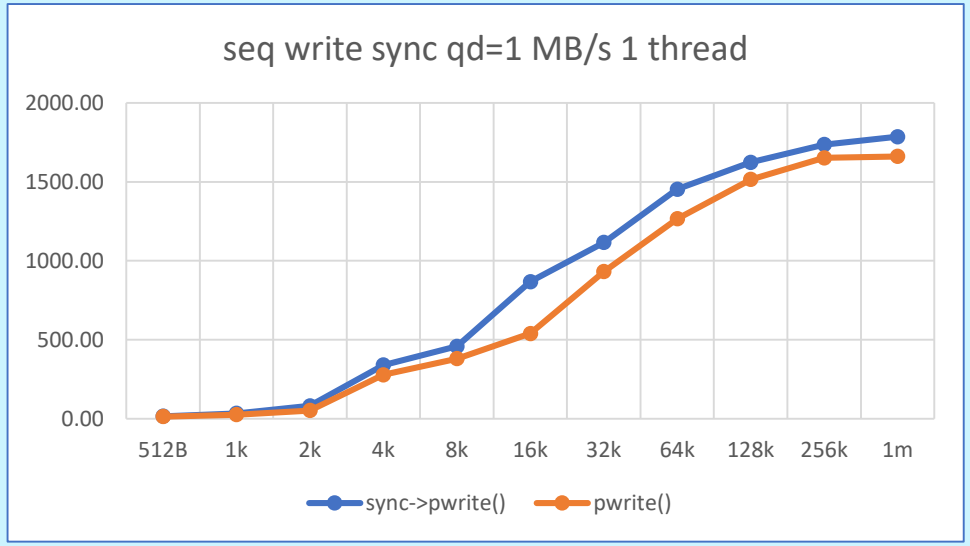


[1]See appendix for footnote 1. For more complete information on performance and benchmark results, visit www.intel.com/benchmarks

Benchmarks - seq write benchmarks vs. I/O size[1]

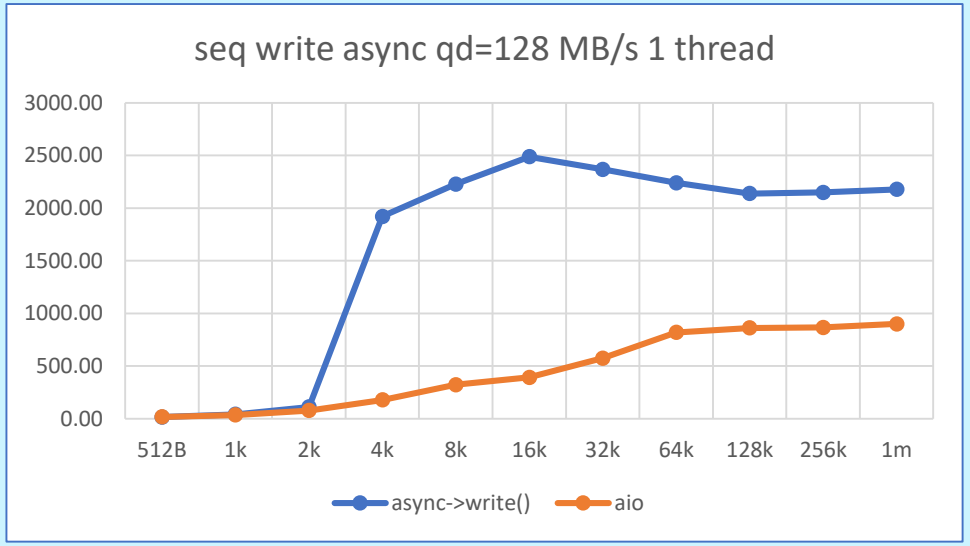
1 x Intel® Optane™ SSD DC P4800X		
write seq sync unbuffered qd=1 MB/s 1 thread		
I/O size	sync->pwrite()	pwrite()
512B	15.70	13.20
1k	33.70	25.80
2k	81.40	51.70
4k	340.00	278.00
8k	458.20	380.60
16k	868.10	539.20
32k	1116.00	931.80
64k	1453.50	1265.50
128k	1624.00	1515.00
256k	1736.00	1652.00
1m	1786.00	1661.00

sync->pwrite() - User-space
pwrite() – Ext4
Queue depth = 1
Drive saturation @ 2.5 GB/s



1 x SSD DC P4800X		
write seq async unbuffered qd=128 MB/s 1 thread		
I/O size	async->write()	aio
512B	16.30	17.20
1k	41.00	35.90
2k	111.50	77.90
4k	1921.00	179.00
8k	2229.00	323.50
16k	2489.00	392.00
32k	2369.00	575.00
64k	2240.00	820.00
128k	2139.00	863.00
256k	2150.00	867.00
1m	2178.00	901.00

async->pwrite() - User-space
aio – Ext4
Queue depth = 128
Drive saturation @ 2.5 GB/s



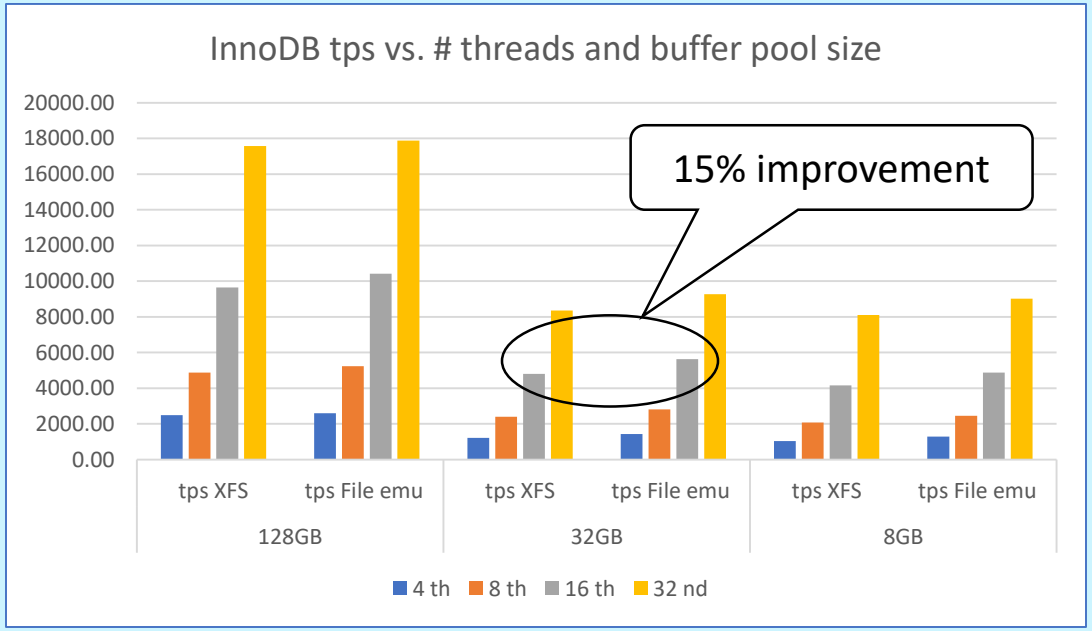
[1]See appendix for footnote 1. For more complete information on performance and benchmark results, visit www.intel.com/benchmarks

Benchmarks - InnoDB OLTP (tps vs. #threads and buffer pool size)[2]

OLTP sysbench tps

- I/O latency is only a small percentage of database transaction latency
- File emulation shows up to 15% improvement in tps
- Smaller buffer pool size generates higher I/O load
  - Database no longer entirely fits in buffer memory
  - InnoDB starts paging data in and out
  - Higher read/write I/O is observed
  - Performance improvements gets amplified
- InnoDB issues both buffered and unbuffered I/Os
  - Buffered I/Os benefit from XFS flush optimization
  - Potential for improvement by adding buffered I/O capability to user-space file system

tps vs. # threads and buffer_pool_size 1 x Intel® Optane™ SSD DC P4800X						
	128GB		32GB		8GB	
#threads	tps XFS	tps File emu	tps XFS	tps File emu	tps XFS	tps File emu
4 th	2492.00	2596.00	1212.00	1434.00	1042.00	1285.00
8 th	4883.00	5231.00	2404.00	2816.00	2084.00	2449.00
16 th	9641.00	10423.00	4811.00	5634.00	4160.00	4879.00
32 th	17575.00	17879.00	8346.00	9267.00	8098.00	9023.00



[2]See appendix for footnote 2. For more complete information on performance and benchmark results, visit www.intel.com/benchmarks

Appendix

1. Dual socket Intel® Xeon® Skylake 48 (hyperthread) core @ 2.8 GHz

192 GB DRAM, DDR4 2666 GHz; Intel® Optane™ SSD DC P4800X

Fedora Linux 29, kernel 4.16.3-301.fc28.x86_64

2. Dual socket Intel® Xeon® Skylake 48 (hyperthread) core @ 2.8 GHz; 192 GB DRAM, DDR4 2666 MT/s; Intel® Optane™ SSD DC P4800X; Fedora Linux 29, kernel 4.16.3-301.fc28.x86_64; MySQL 8.0.21

- XFS on 1 x Intel Optane SSD DC P4800X
- Binlog disabled
- Innodb_flush_method=O_DIRECT
- Innodb_doublewrite=1
- Innodb_buffer_pool_size=128,32,8GB
- Innodb_buffer_pool_instances=24


intel