iSCSI-SCST Storage Server Usermode Adaptation

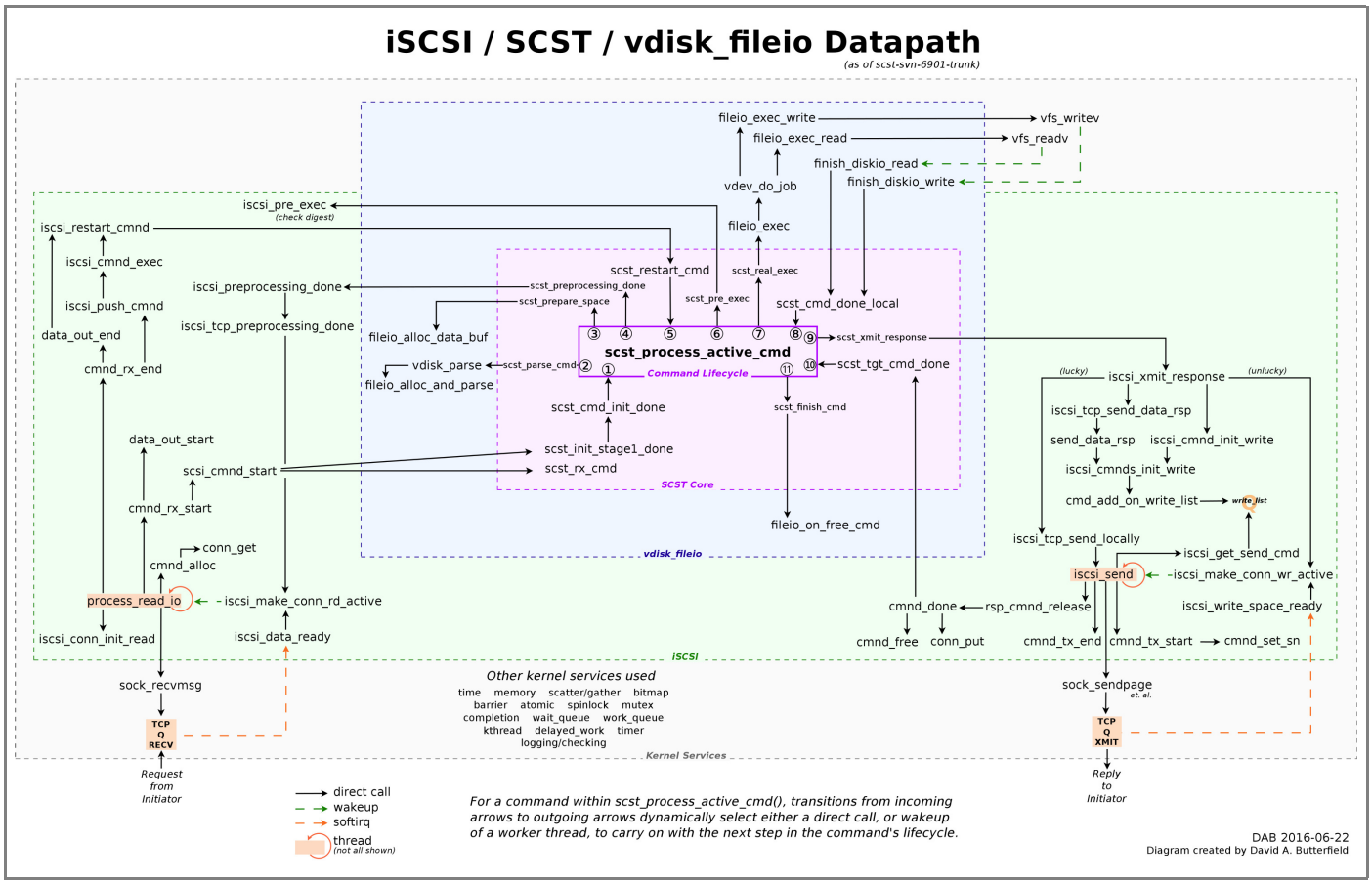
David A. Butterfield — February 2017

Introduction

This paper describes an adaptation of the iSCSI-SCST storage server software to run entirely in usermode on an unmodified Linux kernel; performance measurements and model; and an experimental algorithm to improve performance for small Read operations. The Appendix lists a few issues discovered along the way in the SCST source code.

The SCST Core is a performance-oriented SCSI command processing engine residing within the Linux kernel, supporting a variety of transport types and SCSI device types. One of the supported transport types is iSCSI, which transports SCSI Requests and Responses over TCP/IP networks. One of the supported device types is a virtual disk (implementing SCSI Block Commands), which can be backed by either a real block device or a file in the Server's filesystem. Together these provide a block storage service accessible by *iSCSI Initiator* clients in a TCP/IP network.

The following diagram shows part of an SCST configuration with those components. The areas with colored background depict parts of SCST, while the (slightly) grey area represents other (non-SCST) services within the kernel that the SCST implementation makes use of.



At the heart of the SCST Core *[pink background]* is **scst_process_active_cmd()**, the central state-machine function that drives each operation through the stages of its lifecycle. Other functions within the pink area comprise interfaces into and out of the central function.

The area with blue background represents SCST *vdisk_fileio* functions that interface with the backing storage. Some of those functions make calls out to non-SCST kernel services *[vfs_readv(), vfs_writev()]*, shown in the grey area at the upper-right of the diagram.

The area with green background represents the iSCSI transport layer, interfacing at the bottom left and right of the diagram with the kernel's internal socket I/O API. Requests from the Initiator traverse the network, arriving at the Server's TCP receive buffer *[bottom-left]*, from which they can be read by the iSCSI layer using sock_recvmsg() and sent to the core for further processing. As an OP is completing its lifecycle, sock_sendpage() may be used to enqueue the Response *[bottom-right]* for transmission to the Initiator.

A list of other services consumed by SCST appears near the center bottom of the diagram in the grey Kernel Services area.

The **iSCSI-SCST Usermode Adaptation** begins with components of SCST, including all the ones depicted in the colored part of the diagram, adds a few lines of #ifdef code in a few places, and wraps that into a process with other support components so that the SCST logic can run in usermode with virtually no source code change.

This could be described as analogous to lifting the colored part of the diagram out of the "*Kernel Services*" grey part, and dropping it into a new "*Simulated Kernel Services*" grey part implemented as a usermode process, with the colored part unaware of the difference.

SCSI	Small Computer Systems Interface
iSCSI	Internet SCSI — network transport protocol for SCSI commands over TCP/IP
SBC	SCSI Block Commands — SCSI disk storage command protocol
IOPS	Input/Output operations per second
MBPS	Megabytes (10^6 Bytes) per second
QD	Queue Depth — maximum allowed commands initiated but not completed
MTU	Maximum Transmission Unit — largest network packet excluding MAC/PHY fields
AIO	Asynchronous Input/Output
autocorking	Linux kernel heuristic for aggregating TCP transmit data into fewer packets
procfs	Provides a filesystem view of Linux kernel parameters and state

Motivation

The idea to port SCST to run in usermode was originally conceived as a way to test the capabilities of the Multi-Threaded Engine (MTE) I wrote last year. The "*Simulated Kernel Services*" logic was the main component that had to be written to do this — implementing that code brought the further benefit of giving me some exposure to Linux internal kernel interfaces.

Other possible reasons to run SCST in usermode include:

- Access to a wider user base that includes those with unmodified kernels
- Run under future Linux kernels without additional "*backport*" work
  (could also run on other Unix/POSIX systems with GNU C, fixing a very few Linux dependencies)
- Usermode process runs as a regular user (permission to underlying storage required)
- If your kernel starts having problems, something other than SCST is broken (trouble localization)
- Stage updated versions onto production machines on alternative TCP port for pre-testing
- Much less concern about memory usage, especially stack depth
- CPU-time statistics separation of SCST logic (%USR) from net/storage I/O activity (%SYS)
- Ability to run valgrind(1) to find memory leaks
- Ease of debugging under gdb(1)
- Turnaround of a few seconds from compile to test (or SEGV to retry) when debugging new Server code
- Collecting very comparable data points for the performance debate between usermode and kernel

If iSCSI/VDISK are the only transport and device types you need, then the only reason I've thought of *not* to run SCST in usermode would be if its performance turns out to be significantly lower than running it in the kernel. (At 1 Gb network speed it seems to do very well.)

Implemented Functionality

The iSCSI-SCST Usermode Adaptation supports a subset of the full SCST function — it supports the iSCSI transport type, and the vdisk device type (NULLIO, FILEIO, and prototype BLOCKIO) implementing SCSI Block Commands (SBC). (vcdrom seems to be in there too, but I've never tried it.) It also supports the procfs logic using fuse(8) to export proc_dir_entries to the file namespace where they can be observed and manipulated, most notably by **scstadmin**.

Both files and real /dev block devices can be configured as backing storage in either of SCST's FILEIO or BLOCKIO storage access modes. BLOCKIO has some potential performance advantages (pipeline parallelism) at network speeds faster than the 1 Gigabit I tested with, but it is not clear if or at what point it would begin to outperform FILEIO. Except where noted, all the results reported here are for runs using FILEIO mode — it operates faster than BLOCKIO in these test scenarios, which use /dev/zero or keep SCSI volume data in kernel memory cache so Read operations can complete without sleeping to wait for completion of actual I/O to a storage device.

In extension to the NULLIO concept, /dev/zero can be opened and pretends to be a BLOCKIO or a FILEIO object. That option includes (in performance measurements) the context switches between usermode and the kernel for disk I/O, while still avoiding media delays. NULLIO avoids the trip into the kernel and back that is taken by I/O to /dev/zero.

Configuring in /etc/scst.conf according to /proc conventions:

```
    [HANDLER vdisk]
    DEVICE disk_NULLIO,/dev/zero,NULLIO,512,N000
    DEVICE disk_ZERO,/dev/zero,,512,Z000
    DEVICE disk_BZERO,/dev/zero,BLOCKIO,512,BZ000
```

```
NOTE:  To successfully configure, the scstadmin script must know where the SCST /proc
       mountpoint is -- scst_compat.c currently sets this to mount on /fuse/scst/proc
    --- /usr/local/share/perl/5.18.2/SCST/SCST.pm.ORIG
    +++ /usr/local/share/perl/5.18.2/SCST/SCST.pm
    -my $_SCST_DIR_           = '/proc/scsi_tgt';
    +my $_SCST_DIR_           = '/fuse/scst/proc/scsi_tgt';
```

Implementation Overview

In a standard installation of SCST the **iscsi-scstd** daemon runs as a single-threaded Linux usermode process that cooperates with the kernel-resident SCST implementation using ioctl(2) and netlink(7) for communication.

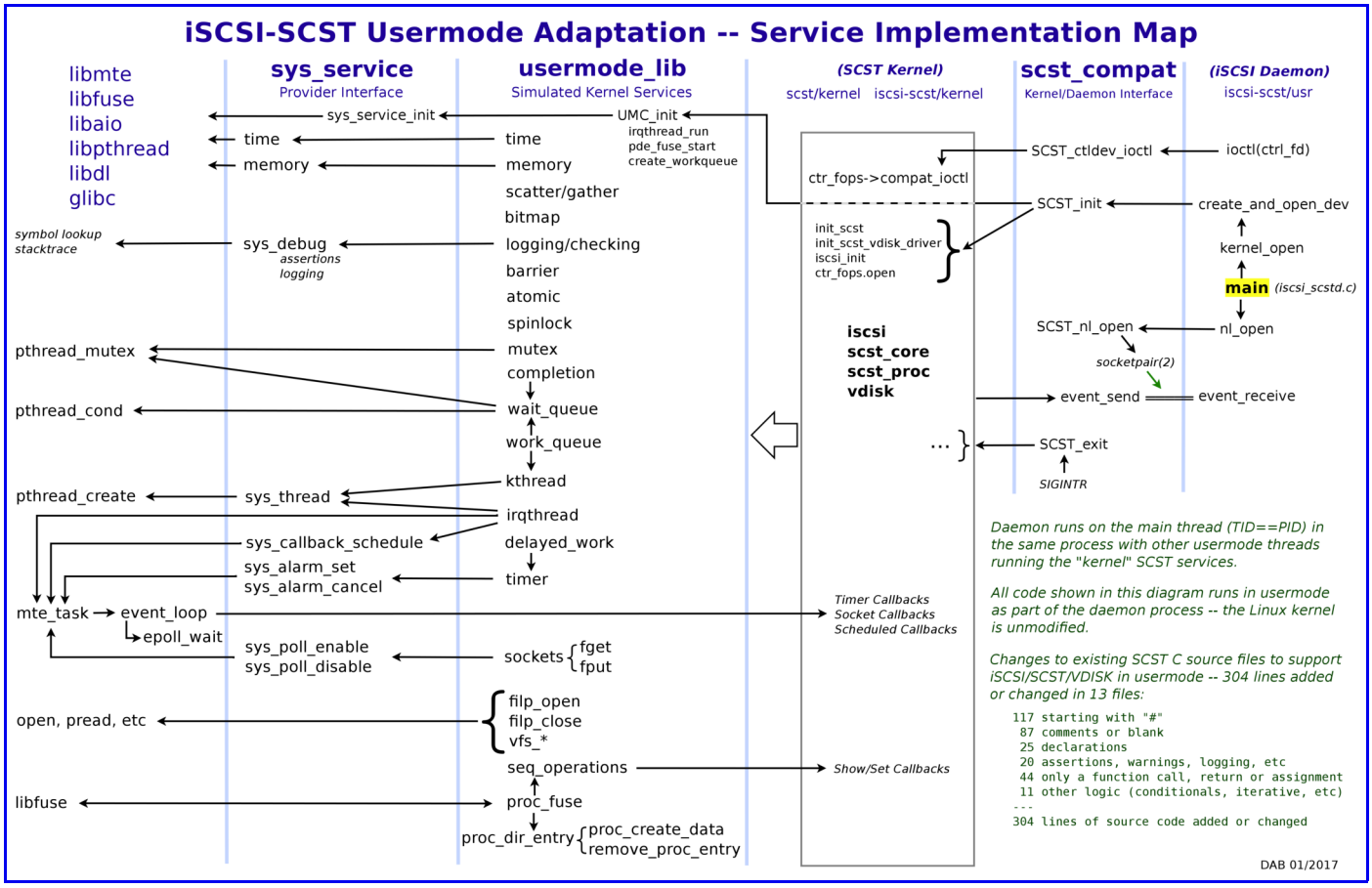
In the iSCSI-SCST Usermode Adaptation the iscsi-scstd daemon runs on the main thread (TID = PID) in a multi-threaded process in which other usermode threads are concurrently providing the services and executing the SCST code that would be running inside the kernel in a standard installation of SCST.

The daemon is modified to drive initialization and shutdown of the "system" components and the threads to run them, through a small *scst_compat* wrapper component that has knowledge of the various main components and how to initialize and configure them together: including the SCST kernel components, the *usermode_lib* kernel service simulation logic, and the *Multi-Threaded Engine* (MTE) that provides (for example) asynchronous notification services for network connections.

The *scst_compat* wrapper also provides interfaces for the ioctl(ctlfd) and netlink(7) communication between the daemon code and the SCST kernel logic.

The subset of SCST used includes the SCST Core, the iSCSI daemon and kernel logic, the vdisk device, and the /proc interface; comprising about 80,000 lines of code. To support running in usermode, around 55 lines of *executable* C code have been added or changed in SCST source files (about 300 lines in total, but 80% of those lines are comments, declarations, or begin with "#").

The SCST kernel code is not aware of the MTE library, nor is it aware that it is running in usermode and that the "kernel functions" it calls are the ones in the *usermode_lib* module and not the real ones. The *scst_compat* wrapper uses socketpair(2) to create a simulated netlink(7) interface; and intercepts ioctl(2) calls made from the daemon, directing them to SCST's ctr_fops->compat_ioctl() entry point.



The *usermode_lib* module is a functionally-approximate reimplementation of the kernel service calls that the SCST kernel logic makes into a Linux 2.6.24 kernel, with SCST's backport.h taking up the slack. It is not specifically aware of SCST, or that it is supporting a storage server — it merely provides service similar to the real kernel functions; but the subset of functions chosen for reimplementation are those required by SCST running on a 2.6.24 kernel. [This could be expanded, of course — I only implemented what I needed to get SCST to run in usermode under the Multi-Threaded Engine; backport.h made that much easier.]

The main kernel services (partially) reimplemented include:

time, memory, scatter/gather, bitmap, logging/checking, barrier, atomic, spinlock, mutex, completion, wait_queue, work_queue, kthread, delayed_work, timer, fget/fput (sockets), filp_open/filp_close (files and block devices), seq_operations, proc_dir_entry.

> [Many functions were reimplemented with semantics deduced from nothing more than the function name and arguments — one reason the reimplementations must be considered approximate. Also, data structures probably do not much resemble the originals. But keep in mind that the reimplementation need not be true and complete to the original; only to the requirements of SCST's usage. This is a tradeoff.]

The *usermode_lib* module also implements *irqthreads* which ultimately run the MTE event loop, using epoll_wait(2) to receive socket ready and other events for delivery to consumers.

Many of the services provided by the simulated kernel functions in *usermode_lib* are implemented directly using POSIX or glibc calls (e.g. wait_queue and mutex in the diagram above). I did not go out of my way to avoid using Linux-specific system call options, but there aren't very many in the code and they should be easy enough to abstract out, opening the possibility to run SCST in usermode on *non*-Linux POSIX systems having glibc and the other support libraries.

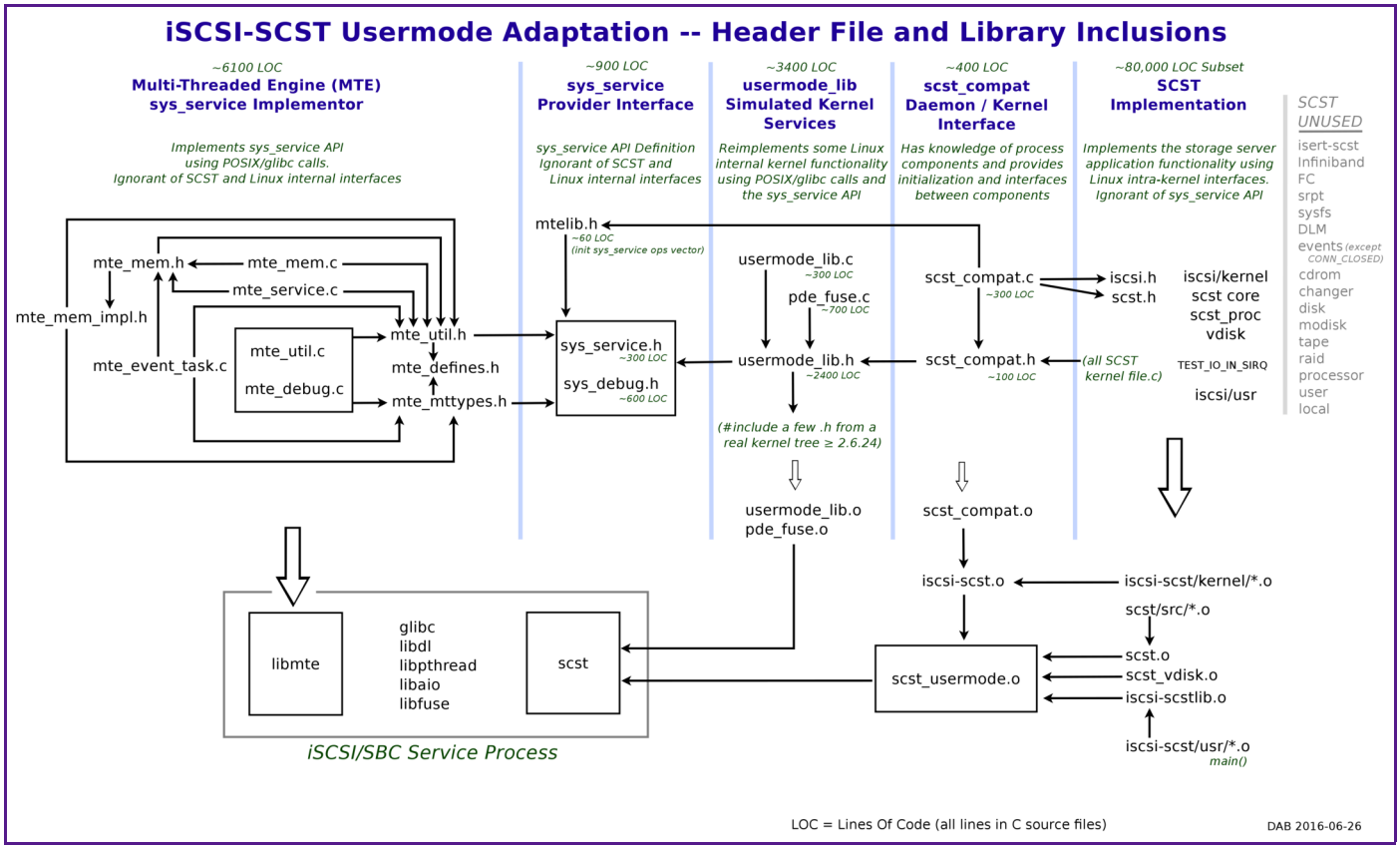
Other simulated kernel functions are implemented using the services of the MTE library (e.g. timers and simulated SIRQ delivery). The API implemented by MTE is the *sys_service provider interface*, defined in sys_service.h (and sys_debug.h). All direct use of MTE in the process is by the kernel simulation logic and through the sys_service provider interface. The SCST storage server logic makes only indirect use of MTE through the simulated kernel functions. This maintains decoupling of the (socket/event/scheduling) "system" logic in the usermode process from the (iSCSI Server) "application" logic, minimizing change to the latter.

The MTE library is ignorant of both the SCST application, and the fact that what it is supporting in this configuration is a simulation of kernel functions. MTE is a general-purpose multi-threaded event-driven engine designed for high performance usermode dispatching and I/O, intended to support event-driven applications generally, not this storage server application in particular.

Only the *scst_compat* wrapper is aware of the whole picture, so that it can initialize and shut down the various components it configures together to form the service process, and provide communication interfaces between them.

The ~300 lines of code change to support running in usermode are mostly within #ifdef SCST_USERMODE. Some additional changes to improve performance while running in usermode (but not necessary just to have it work) are mostly under other #ifdef symbols. The main exception is the change that splits conn_rd_[un]lock from conn_pool_rd_[un]lock, which would have been too messy left as #ifdefs, and fortunately is usermode-independent.

#ifdef	Purpose
SCST_USERMODE	Basic changes to support running SCST code in usermode
CONFIG_SCST_TEST_IO_IN_SIRQ	Updated to allow iscsi_restart_cmnd() to call scst_restart_cmd(SCST_CONTEXT_DIRECT)
CONN_SIRQ_READ	Allow iscsi_data_ready() to read socket directly instead of only iscsi_make_conn_rd_active()
CONN_LOCAL_READ	Allow iscsi_tcp_preprocessing_done() to read socket directly
SCST_USERMODE_AIO	Prototype support for BLOCKIO using aio(7)
ADAPTIVE_NAGLE	Experimental performance improvement for small I/O sizes
VALGRIND	Set DEF_READ_ZERO and SGV_POOL_ALLOC_NO_CACHED by default for valgrind



Threading

The threading arrangement can be varied by compile-time parameters (#ifdefs above), but this describes the way I have been testing it. Discussion omits consideration of exception paths, such as for low memory.

Each iSCSI session has one dedicated thread on the Server that (partially or fully) services each I/O operation for the session. It receives all iSCSI Requests from the socket and carries them at least as far as the storage I/O request. In synchronous disk I/O cases the same thread also carries the Response back to the socket write system call after the storage I/O operation completes; in asynchronous disk I/O cases another thread carries the OP back through the Response path.

For NULLIO, operations are synchronous, and an entire operation (from network Request receive to Response transmit) is executed synchronously straight out of the socket "*receive data available*" (simulated SIRQ) notification callback function, on its MTE event thread.

FILEIO is similarly synchronous, except for the sleep that may occur to the thread during the disk I/O calls to vfs_readv() *[preadv(2)]* and vfs_writev() *[pwritev(2)]*. My testing was on Read workloads, to /dev/zero or with the volume data entirely resident in kernel memory cache, so *[I suppose]* these sleeps did not occur and the Read operations completed synchronously.

This synchronous operation is most efficient, avoiding context switches in an OP's lifecycle; but it is not necessarily the fastest method, because it does not exploit potential pipelining of a session's operations, which could allow multiple CPUs to work a session concurrently.

For example, instead of one dedicated thread per session, each session could be serviced by a pair of threads — one thread on one CPU could carry operation Requests from socket Request receive to AIO (asynchronous disk I/O) submit; while a second thread on another CPU could carry operation Responses from AIO completion to socket Response transmit.

If the Request and Response paths happen to split the work evenly (and if there are more CPUs than CPU-bound sessions), using a pair of threads instead of a single thread could conceivably double throughput for a CPU-bound workload.

On the other hand, a single medium-speed processor can keep a 1 Gb Ethernet connection saturated to full bandwidth at all but the smallest couple of I/O sizes; so a 1 Gb network does not seem like a case likely to benefit much from parallelizing a session's OPs across multiple CPUs — the OP multi-threading overhead might be better spent on maximizing throughput across a larger number of sessions.

For small-size Read workloads that *are* CPU bound, in cases where there are more CPU-bound Sessions than CPUs: I would expect FILEIO mode always to support more *total* throughput across all sessions than BLOCKIO mode, because it's more efficient (fewer context switches, less lock contention). But presumably BLOCKIO wouldn't exist unless there were situations I didn't reach where it improves throughput for an individual session having sufficiently high available bandwidth, when there are spare CPUs that can be put to use.

The BLOCKIO prototype *[not shown on diagrams]* uses an MTE service that invokes an aio(7) io_submit() call to issue asynchronous disk I/O requests to the kernel, and gets completion events from the MTE event_loop dispatching a callback when the AIO completion event file descriptor indicates ready. For BLOCKIO, the functions vfs_readv() and vfs_writev() *[upper-right grey area of first diagram]* are replaced by functions calling io_submit(), and the FILEIO functions *[in the blue area]* are replaced with the analogous BLOCKIO functionality.

Under large-I/O, high-Queue-Depth Read workloads an iSCSI socket Writer Thread can get involved; but optimally operations in BLOCKIO mode occur on two threads: ***(1)*** the thread that gets the socket receive event: reads the Request from the socket and carries it through SCST all the way to io_submit() *[in the first diagram at scst_process_active_cmd() OP lifecycle step 7];* ***(2)*** the thread that receives the AIO completion event: resumes OP processing *[at step 8]* and carries the Response all the way back through the Response path to the iSCSI socket write and final free of OP resources.

In contrast to the hypothetical two-thread example suggested above, the BLOCKIO prototype maintains a single dedicated thread per *session*, to carry Requests as far as AIO submission; and each *volume* (Logical Unit) has one dedicated thread to carry Responses from that volume's AIO completions to the socket write call (with suitable synchronization for a session using multiple LUNs).

When using FILEIO or NULLIO the threading arrangement is such that most of the work for a session is done by a *Session Thread* dedicated to serving that session.

> The exception is under large-I/O, high-queue-depth Read workloads when an iSCSI socket Writer Thread can also get involved — **this cannot occur unless Queue_Depth > (server.tcp_wmem / (IOSIZE+48)),** the point at which it is possible to have more replies pending than will fit in the Server's kernel socket transmit buffer (and consequently possible for an attempted write to return short or EAGAIN). In such cases a Writer Thread continues the socket writes when space again becomes available in the kernel socket transmit buffer. [This situation is noted as the "*unlucky*" case in the iSCSI Response path in the green area at the right-side of the first diagram; though it isn't really luck because it depends deterministically on the tcp_wmem size, I/O size, and Queue Depth, according to the formula just given].

CPU Assignments

To get stable, consistent performance results it is very necessary to pin each Session Thread to its own CPU. Assuming the Server machine is otherwise mostly idle, it is not necessary to ban everything else from those CPUs — the pinning is mainly to prevent Session Threads from migrating CPUs frequently and incurring a substantial performance penalty. Notes:

> *Frequent and short Session Thread sleeps are inevitable under a network-bound workload (which most 1 Gb workloads are).*
>
> Based on its behavior, it appears that the Linux scheduler does *not* prefer to maintain the waking Session Thread on its prior CPU after a short sleep. (This observation appears to contradict inferences easily drawn from other writings concerning the spirit of the scheduler's behavior.)
>
> The Server does not pin its Session Threads automatically — in testing I used taskset(1) to manually pin Session Threads to processors after sessions connected. Also, when there are spare CPUs, I reduce noise by pinning a few select threads like xorg, compiz, top, and other instrumentation away from the Session Thread CPU(s).

In tests of small I/O sizes I also found it necessary to pin the IOmeter/dynamo thread on the Linux *Initiator* to keep it running efficiently longer before going CPU-bound and limiting testing of the Server. Pinning to a processor other than the one showing "si" (software interrupt processing) time when the network is under load gave better results than pinning to the processor incurring the "si" time.

For CPU-bound workloads: when there are spare CPUs on the Server (more CPUs than sessions), there is a significant performance advantage to running the Session Thread on a different CPU from the one servicing the network software interrupts. This would normally be done by pinning the network IRQ to some other CPU. For my testing I instead observed that the network software interrupts were always occurring on a specific CPU, and assigned the Session Thread to a different CPU. So part of the TCP protocol processing for a session would run on the extra CPU in parallel, freeing cycles for the Session Thread to process more IOPS.

Some charts in a later section show the different performance observed on different CPUs, but a simple experiment demonstrates the effect. On my test system it is easy to observe repeatable definite performance changes of at least few percent using taskset(1) to move the Session Thread around between CPUs while running a CPU-bound single-session IOmeter workload. I ran a 512 Byte Random Read workload on /dev/zero at Queue Depth 128, pinning the Session Thread to each CPU in turn, observing IOmeter throughput for each CPU.

The state of the test system during this experiment was such that all the network "si" time happened to be running on CPU1; and I had a few "noise-producing" processes pinned to (CPU2|CPU3) to keep them out of the way of my other testing. Due to the Session Thread being CPU bound, when it is pinned to CPU2 or CPU3 these noise-producing threads soon migrate to their other permitted processor.

The table below shows the results for each CPU. CPU1 is expected to show slower performance because the Session Thread received only about 3/4 of that CPU; and we can see in the table that CPU1 is indeed the slowest case. By running the Session Thread on some other CPU we get some parallelism with network software interrupt processing as suggested above.

But why is CPU0 faster than CPU2 and CPU3? (All CPUs are set not to vary from 2.4 GHz).

CPU	IOPS	Notes
0	62,798	
1	45,456	About 1/4 of CPU1 was consumed in top(1) "si" time, handling network interrupts
2	52,615	
3	54,514	

The test system CPU package is an Intel Core 2 Quad E6600. The Core 2 Quad has four processors, not hyperthreaded, so there are four CPU threads, and they each get a whole CPU. This is effectively the contents of two "Duo" CPU packages glued together into one "Quad" package — in particular, the two CPUs in each "Duo-pair" share their common L2 cache, and there are two of those L2 caches in the Quad package, one for each Duo-pair.

In general, depending on the interactions between threads in a program, if one thread is consuming CPU0, there may be cache advantages to running particular other threads on CPU1, sharing the L2 cache with CPU0; or *banning* particular threads from CPU1 to *avoid* sharing the L2 cache with CPU0. The same effect holds for threads sharing cache with software interrupt activity.

I attribute the ~17% higher throughput observed with CPU0, as compared with CPU2 and CPU3, to the fact that in the faster case the processor handling the interrupt-time network processing shares L2 cache with the CPU processing the network data.

Performance Measurements

Performance of the Usermode Adaptation seems to be very good; but I've never run SCST in the kernel, which would be the *best* baseline for comparison; and I don't really have the equipment here at home to max it out anyway. But I've done some measurement of Read operations using IOmeter on my fastest available system that isn't a laptop (so I can add enough network ports to give it some load). [I did not study Write performance.]

Beyond measuring Read performance of the existing SCST logic when running in the MTE usermode environment, I also measured some experimental code changes intended to improve throughput for small-size (CPU-bound, e.g. 512-Byte) Random Read operations (described in a later section). This optimization applies also to Sequential I/O using the Microsoft Initiator. Sequential I/O under the Linux Initiator is not improved much (with its OP coalescing enabled) because this optimization benefits CPU-bound workloads, and coalescing tends to drive the I/O size up into a range where the Session Thread is no longer CPU bound. This is optimal because when the Initiator supports it and the load is Sequential, OP coalescing brings a larger performance benefit than the experimental optimization.

My testing has been geared toward understanding the iSCSI Server performance behavior and capability when run in usermode on a plausible secondary home machine in a 1 Gb network — that probably being the speed of an affordable "fast" home or small-business network for the time being. These tests are not trying to measure performance of a storage device or drive.

The system running usermode SCST in my tests is a vintage 2007 home desktop (cast off by adolescent gamers for being "too slow"): it's fine as a development workstation, but not blindingly fast by today's standards. Maybe it could be considered typical as a second, older machine that's lying around waiting to be turned into a storage server on someone's home 1 Gb Ethernet. A Quad PCI Express Ethernet card is inexpensive and was added for this project. **For best performance it is essential to use a PCI Express card for networking and not a PCI card.** In my testing using PCI I was able to get one session to run at about 3/4 of the 1 Gb network capacity; whereas using PCI Express I ran four concurrent 1 Gb sessions each at 99% network capacity.

Motherboard	ASUS P5K-V
CPU	Intel Core 2 Quad E6600 @ 2.4 GHz (set fixed at that speed except as noted)
RAM	4 * 2GiB DDR2-800 (8 GiB total RAM)
Network	Quad Gigabit Ethernet PCI Express Intel 82571EB (MTU=9000, tcp_wmem=1677721)
OS	Linux 3.13.0-101-generic #148-Ubuntu SMP Thu Oct 20 22:08:32 UTC 2016 x86_64
SCST	Usermode Adaptation based on sourceforge.net/projects/scst **scst-svn-7089-trunk** -O2 -DCONFIG_SCST_TEST_IO_IN_SIRQ -DCONN_LOCAL_READ -DCONN_SIRQ_READ -DCONFIG_SCST_PROC

The tests were done in such a way that disk I/O is excluded, so the speed of my disks should not matter. Most of the tests reported here were done to a FILEIO /dev/zero volume, using that rather than NULLIO so as to include in the performance numbers the overhead of switching context to the kernel for storage I/O. I also did some tests using 600 MiB volumes instantiated as vdisk_fileio files in /tmp and brought into the kernel memory cache using dd(1) before starting an IOmeter run. I expect that should include everything except media I/O and driver time into the timing results.

For single-Initiator tests I used a modern Intel laptop running Linux 4.4.0-45-generic #66-Ubuntu x86_64 with its 1 Gb Ethernet interface connected point-to-point to the SCST Server machine (no switch). CPU was set for "performance" but with turbo off to keep clock rate constant. This reduces noise in the test; but the more important reason is that ***if the Initiator CPU clock can vary then a bug in how IOmeter does timing will significantly (but often not obviously) bias its reported results!***

Except where otherwise indicated, Linux Initiator settings were:

Initiator	I/O Type	nomerges	session.cmds_max	session.queue_depth
Linux	Random	2	1024	384
Linux	Sequential	0	1024	24

Notes: Random I/O tests using the Linux Initiator ran with "nomerges=2" to avoid it randomly finding coalesce hits at an unusually high rate due to the small /tmp volume sizes used.

On my hardware the default session.cmds_max/queue_depth of 128/32 was too small to permit maximum IOPS for 512-Byte Random Read. (These are configured in /etc/iscsi/iscsid.conf on the Initiator.)

In the setup used for these Read experiments there was no media I/O; hence no performance advantage within the Server for Sequential I/O over Random I/O (no drive geometry to optimize access to; no opportunity to predict and do readahead). In this setup the Sequential advantage is entirely due to software on the *Initiator* machine coalescing smaller-size Read operations into fewer, larger-size Read operations to be issued to the Server. That tactic benefits network efficiency, and CPU consumption on both the Initiator and the Server.

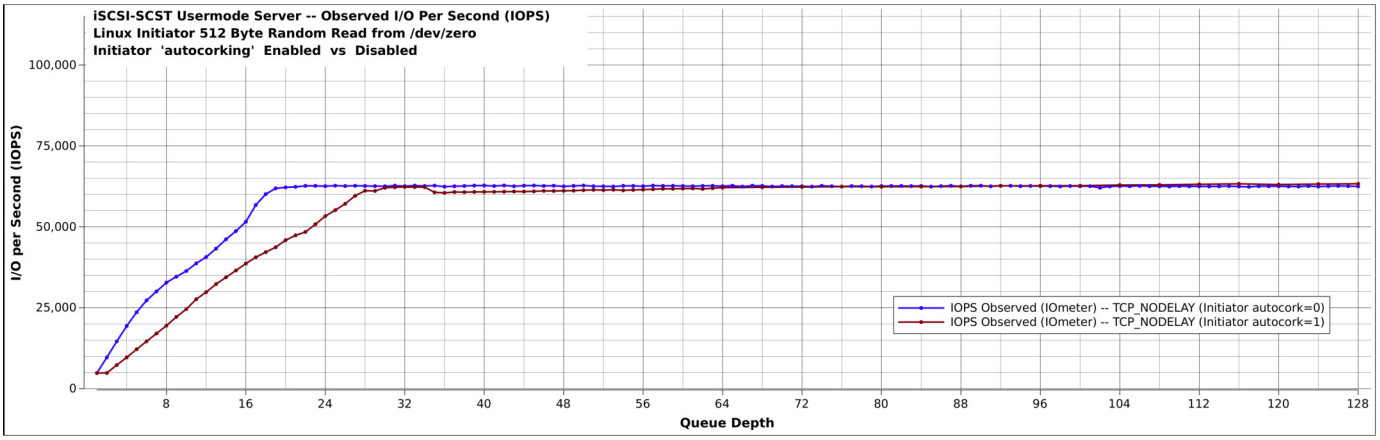
I measured and analyzed performance of Linux-Initiator Sequential I/O, but this paper focuses on the Server; so the experiments described here used Random I/O, which is simpler to analyze than Sequential I/O with coalescing enabled. As noted in the table above, for Sequential Read workloads I set session.queue_depth much lower than for Random workloads — that is because the aggressiveness of SCSI operation coalescing by the Linux Initiator is controlled or influenced by the session.queue_depth setting, and the chosen number produced a more optimal curve for the coalesce factor producing a more pleasing performance curve. *[This is not to be confused with the application-level queue-depth configured to IOmeter.]* On the other hand, when coalescing is *not* active the smaller session.queue_depth is way too small to support full throughput. **I did not find a configuration of these two parameters that can be optimal for both Random and Sequential loads.**

Measurements using the Microsoft Initiator gave about the same results for Random and Sequential workloads, both about the same as running Random workloads using the Linux Initiator. There were some very small but consistent differences in performance, with the Linux Initiator performing slightly better at larger I/O sizes and the Microsoft Initiator performing slightly better at smaller I/O sizes.

For better performance at low Queue Depths: **"autocorking" must be disabled on both the Linux Initiator and the Server** to prevent them from delaying transmission of iSCSI Requests and Responses. In an experiment between two systems supporting autocorking, at Queue Depth 15 the performance fell by 25% if either side was autocorking, about 36% when both were autocorking.

```
echo 0 > /proc/sys/net/ipv4/tcp_autocorking
```

To illustrate the autocorking effect, this chart compares performance of 512-Byte Random Read from /dev/zero running the Linux Initiator *with* and *without* its autocorking enabled (no experimental optimizations enabled). The blue curve represents measurements done with autocorking disabled, which below Queue Depth 28 shows significantly better performance than with autocorking enabled *[red curve]*. At around Queue Depth 28 the autocorking performance catches up and the curves coincide fairly well above that.



The flattening of the curves in this workload represents reaching 100% CPU utilization on the Session Thread; we see it occurring at the same level of IOPS performance for the two curves, so they represent the same level of Server CPU efficiency.

The way I look at this chart is not that "*the red curve is shifted down from the blue curve*", but rather that "*the red curve is shifted to the right from the blue curve*" — the Queue Depth where the red curve flattens and joins the blue curve is several OPs higher than *[to the right of]* where the blue curve flattens; but otherwise they look pretty similar (considering the constraint of the initial condition at Queue Depth 1).

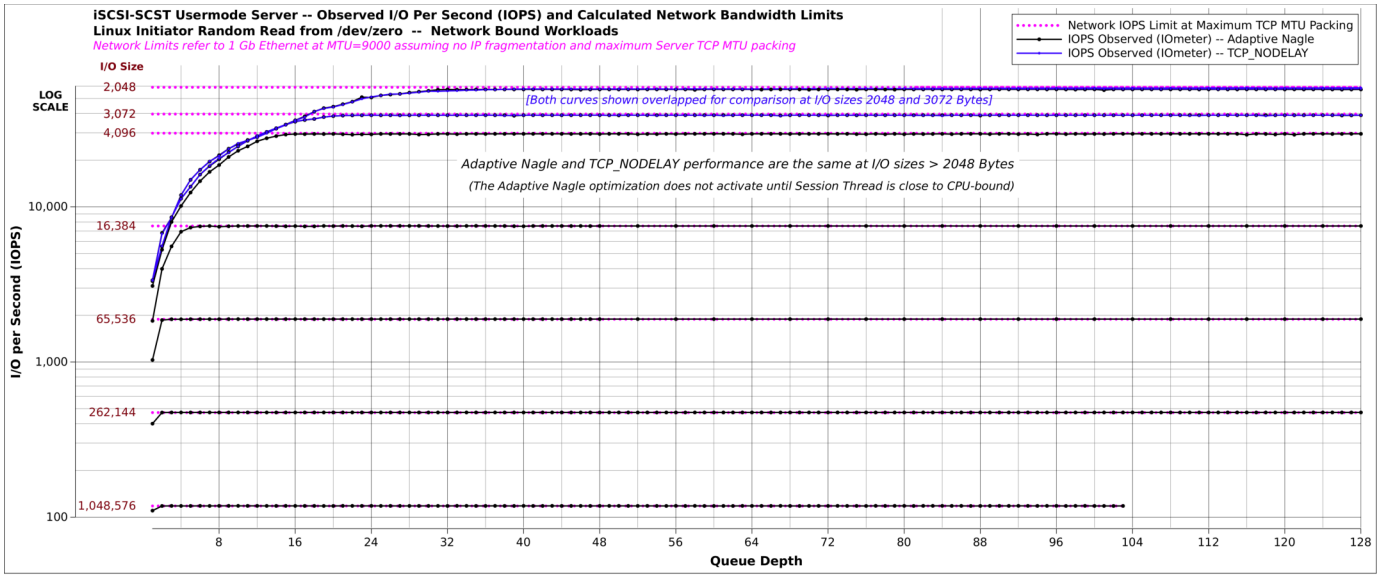
The increase in Queue Depth required to reach peak IOPS is a result of the Initiator delaying transmission of operation Requests due to autocorking: **an increased minimum Queue-Depth is required to overcome an increased latency in the operation pipeline** (see *Performance Model* section later). Except for this example, all measurements reported were done with autocorking disabled.

Notice the tiny but important effect visible at the left edge of the above plot: when autocorking is enabled *[red curve]* the performance at Queue Depth 2 is the same as it is at Queue Depth 1; whereas with autocorking disabled *[blue curve]* the Queue Depth 2 performance is roughly double.

I collected IOmeter data using the Linux Initiator for several runs of Random Read from /dev/zero through a range of Queue Depths at various settings for I/O size. I then used gnumeric spreadsheet to process and plot the data for comparison and analysis.

The chart below plots IOPS by Queue Depth for a selection of **network-bound** I/O sizes of Random Read from a /dev/zero volume. Observed data are shown in solid blue or black and their corresponding computed 1 Gb network bandwidth limits in dotted magenta. Ideally each solid performance curve would reach its corresponding dotted (network bound) limit at low Queue Depth. At 4096 Bytes the session saturates the 1 Gb network by Queue Depth 16. Larger I/O sizes *[lower curves on the chart]* reach maximum IOPS earlier, with 16,384 Bytes saturating at Queue Depth 5, and I/O sizes 65,536 Bytes and larger by Queue Depth 2.

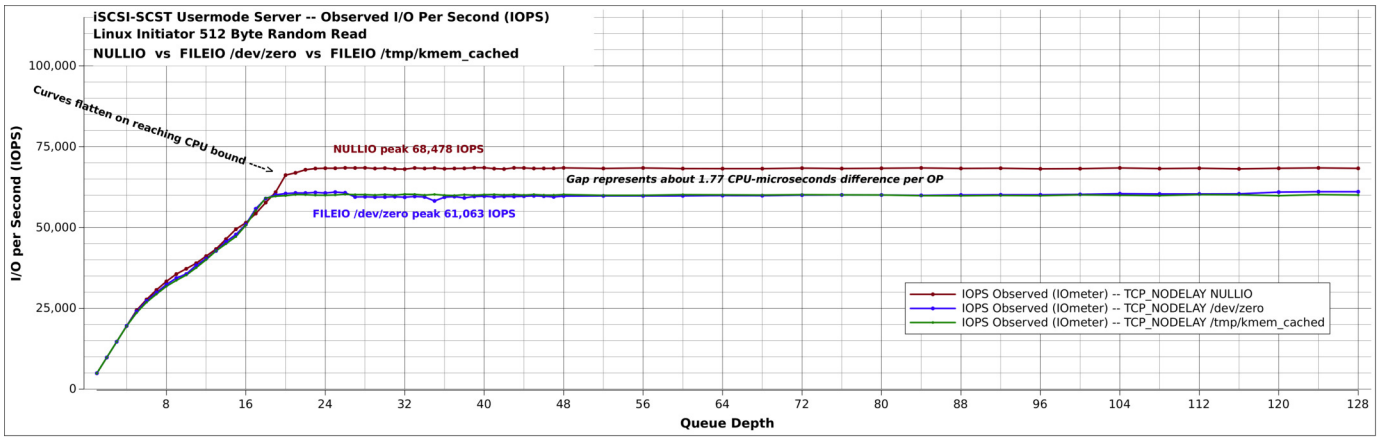
The black curves are from data collected with the experimental Adaptive Nagle optimization present; but that optimization has effect only when CPU bound — for these network-bound workloads it behaves the same as TCP_NODELAY *[blue curves]*. The 3072 Byte curve demonstrates this by being plotted for both TCP_NODELAY and Adaptive Nagle runs, and one is inked right on top of the other. Both 2048-Byte curves are also plotted, appearing within 2% of each other and of the network maximum. At 2048 Bytes the Server does not quite make it all the way to network bound in TCP_NODELAY mode, reaching the CPU limit just slightly before that.



These measurements were done taking a dense selection of data samples across the interesting range of Queue Depths; and using a single Initiator and Session to produce "sharper signal definition" for more precisely deducing behavior from the data — as compared with running several sessions concurrently. (The combined statistical behavior of multiple concurrent Initiators can smooth out and/or obscure transitions that are in reality more definite than they end up appearing; I would expect this effect to be worse when using Initiator hardware of varying performance capabilities.)

The next chart compares performance of 512-Byte Random Read (CPU-bound workload) using NULLIO *[red]*, /dev/zero *[blue]*, and a FILEIO /tmp volume *[green]* brought resident into kernel memory cache using dd(1) before starting a test run.

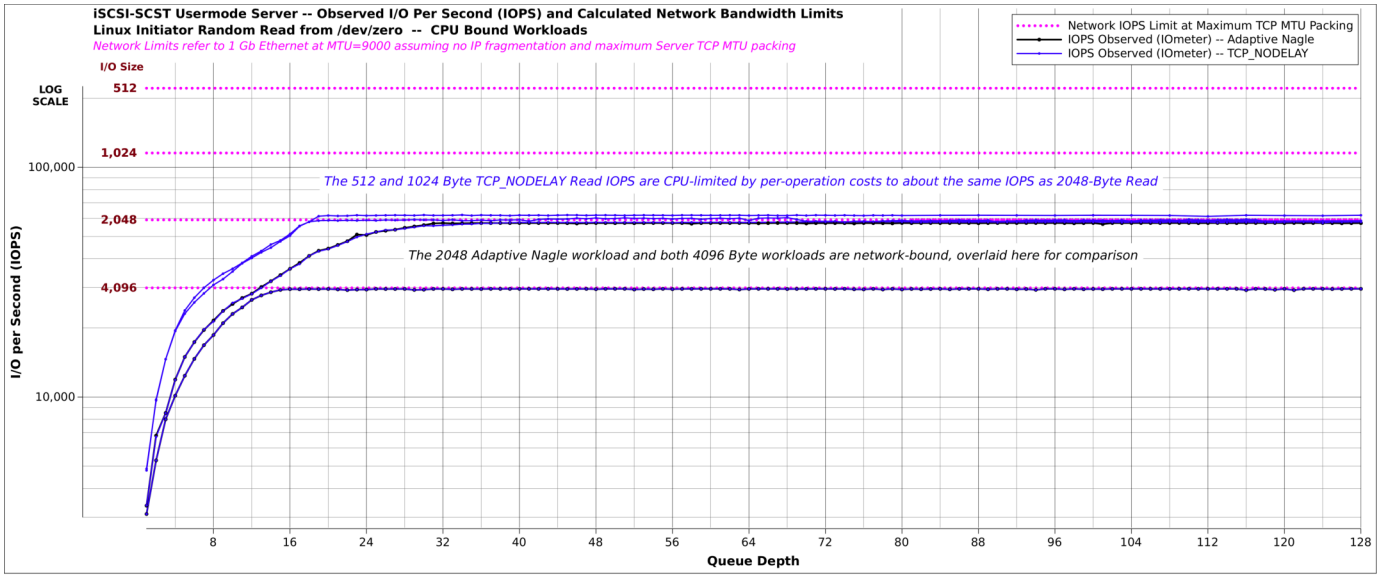
NULLIO omits making system calls related to disk I/O. FILEIO to /dev/zero includes those system calls in the timing, but avoids any media I/O. FILEIO to a /tmp file in kernel memory cache likewise avoids media I/O, but travels a different path within the kernel than /dev/zero.



The IOPS for Reads using NULLIO are higher than for FILEIO from /dev/zero, as expected due to avoiding system calls for storage I/O. When CPU-bound, the reciprocal of the measured IOPS indicate the per-OP CPU time (See *Performance Model* section later). The difference seen here between the NULLIO performance and the /dev/zero performance translates to a difference of about 1.77 microseconds per operation. That is an upper bound on the CPU time consumed to enter the kernel and read a block from /dev/zero. (Some portion of that time is I/O setup in the SCST logic.)

Experimental Performance Improvement for Small Read Sizes **[#ifdef ADAPTIVE_NAGLE]**

Below is another /dev/zero IOPS chart similar to one seen earlier, but this one depicts results of tests of **CPU-bound** workloads. (The 4096 Byte workload and the 2048-Byte Nagle-mode workload are network bound, shown here for comparison.)



It doesn't take long to notice that 512-Byte Random Read operations are not saturating the network. This is due to the Session Thread reaching CPU bound serving at around 1/4 of the 1 Gb network bandwidth capacity. The network utilization shortfall is evident in this IOPS plot as the 512-Byte and 1024-Byte *[blue]* measurement curves not reaching all the way up to their respective magenta network bandwidth limit lines; with performance petering out along with the CPU at around 63,000 IOPS, just slightly above the 2048-Byte IOPS level. *[I suppose but have not confirmed that similar behavior occurs with kernel-based SCST.]* Of all Read sizes, 512 Bytes incurs the highest per-sector Server CPU cost, so that is the worst case, used in most of my CPU-bound testing.

Let "*partial packet*" denote a packet smaller than some threshold at or near the network Maximum Transmission Unit (MTU) size. Sending a large stream of data using packets smaller than the MTU size is less efficient than using the full maximum packet size because there is CPU and network protocol overhead associated with each packet. Such inefficiency can significantly limit data throughput.

John Nagle's algorithm for TCP (RFC 896) seeks to improve network efficiency by causing a TCP protocol implementation to defer sending any partial packet out a TCP connection until all of the previously-transmitted packets on that connection have been acknowledged. The delayed transmission allows multiple small outgoing messages to be combined into a single network transmission, usually improving efficiency of the network and of CPUs at both ends of the connection. But it also introduces a transmission latency which can adversely affect throughput for some types of workloads, including low queue-depth iSCSI workloads.

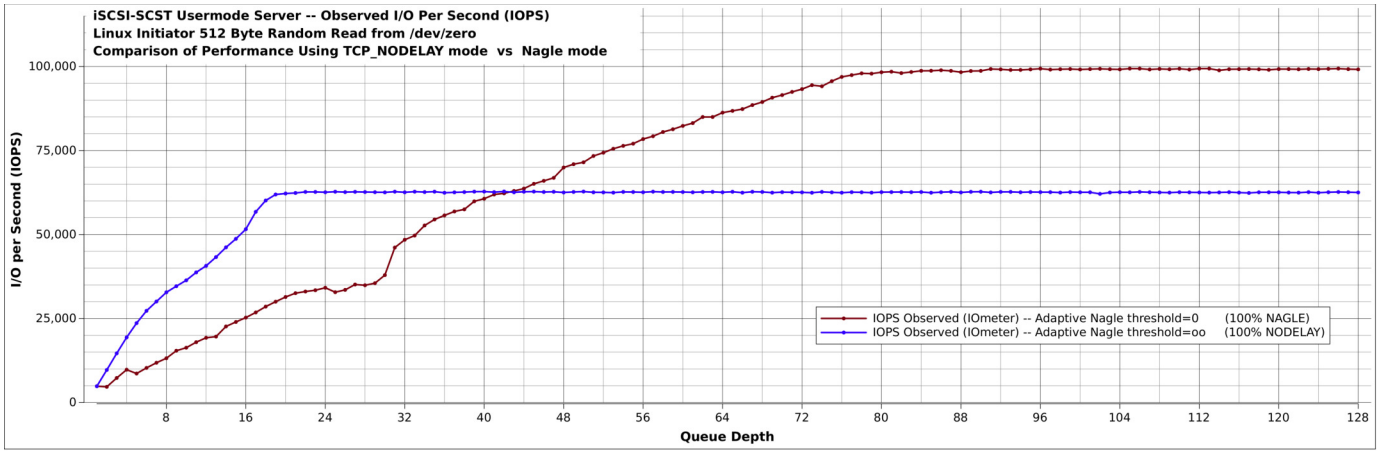
To keep latency low (particularly necessary at low queue depths), SCST disables Nagle's algorithm by setting TCP_NODELAY on the socket. That reduces latency as intended, but at a significant cost to CPU efficiency (on both the Server and the Initiator) due to more packets being processed; and a cost to network bandwidth efficiency due to smaller packets being carried for each instance of network protocol

overhead. In our small-I/O case here, the Session Thread is CPU bound, so Server CPU cost is the relevant limiting bottleneck. Since Nagle's algorithm improves CPU-efficiency, it is reasonable to expect it to improve performance of this CPU-bound workload.

On the other hand, enabling Nagle's algorithm can *degrade* performance at *low* queue depths — mainly due to an unfortunate interaction between the Nagle delay and scheduling latency on the Server (similar to the problem triggered by autocorking at low Queue Depths). More discussion appears in a later section on the *Performance Model* I developed to get a better understanding of the system behavior.

The next plot compares performance of 512-Byte Random Read from /dev/zero with Nagle's algorithm enabled *[red]* and disabled *[blue].* The difference between the red and blue curves represents the difference in performance between running in 100% Nagle mode as opposed to 100% TCP_NODELAY mode. Notice the substantial advantage to Nagle's algorithm starting at the crossover point around Queue Depth 43, across to the right side of the chart; and conversely the substantial disadvantage of Nagle's algorithm below the crossover point.

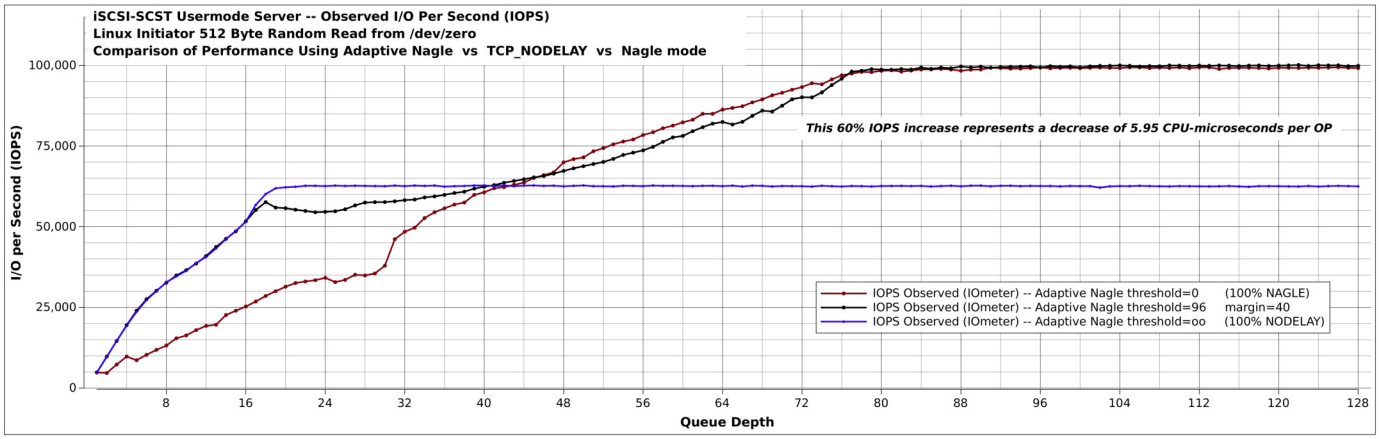
In this plot, the point where a curve flattens represents reaching 100% CPU usage. The *[red]* Nagle configuration takes a heavier workload (requires higher queue depth) to reach 100% than the *[blue]* NODELAY curve, but in doing so attains substantially higher IOPS performance, implying that it reflects a more CPU-efficient mode of operation, as expected. (Notice it also exhibits the same QD2 = QD1 behavior seen in the autocorking chart earlier.)



The intent of the #ifdef ADAPTIVE_NAGLE experimental optimization is to enable Nagle's algorithm during (at least some of) the time when it is beneficial to do so, and not (much) during times when it is detrimental to do so. The Server does not know the Queue Depth (which can vary randomly in real workloads). The criterion I've used for deciding whether to enable Nagle's algorithm is whether or not the **Session Thread is CPU bound**. The theory goes:

- If the Session Thread is CPU bound then network latency is not a problem, as evidenced by the continuous stream of work available to the (limiting bottleneck) Session Thread in its TCP receive buffer; so running with Nagle's algorithm enabled while CPU bound should not be detrimental.
- Moreover, if the Session Thread is CPU bound then it never sleeps, which means it never incurs the scheduling latency, which means the "unfortunate interaction" between scheduling latency and Nagle's algorithm does not occur; so running with Nagle's algorithm enabled while CPU bound should not be detrimental.
- Finally, if the Session Thread is CPU bound then the improved CPU-efficiency gained by enabling Nagle's algorithm (even part of the time) should appear as a beneficial effect on throughput.

This chart adds the Adaptive Nagle performance curve *[black]* for /dev/zero to the previous chart for comparison. The Adaptive Nagle optimization is attempting to capture the superior behavior of the other two modes across the entire range of Queue Depths — the NODELAY behavior *[blue]* below the crossover point, and the Nagle-mode behavior *[red]* above the crossover point. The black curve represents performance using the Adaptive Nagle optimization with its default parameter settings.



Generally the black Adaptive Nagle curve is expected to lie between the other two curves, because Adaptive Nagle is simply a repeating cycle of running in 100% NODELAY mode for some time period, followed by running in 100% Nagle-mode for some other time period (in a fairly constant ratio for a given I/O size, CPU speed, and network speed). The "trapping" of the black curve between the other two curves also implies that it crosses both of the other curves at the same point where they cross each other (within experimental variance).

The basic heuristic decides whether the Session Thread is CPU bound by counting the number of Requests it has processed since the last time it went to sleep after exhausting its TCP receive buffer of all Requests and finding it empty (IDLE). If the count of Requests processed since the last IDLE reaches a tunable *nagle_threshold*, the optimization logic considers its thread as CPU bound and enables Nagle's algorithm on the socket.

Nagle's algorithm then remains enabled on the socket until the next time the Session Thread exhausts its TCP receive buffer, at which time it resets the socket to TCP_NODELAY and the request counter to zero before going to sleep. **This reset to TCP_NODELAY mode triggers the immediate release out to the network of any accumulated partial packet awaiting transmission.**

The implementation (in nthread.c) adds some refinement to that basic model; mainly facilitation of continuous Nagle-mode operation for a generally-CPU-bound session experiencing periodic and/or sporadic interruptions. Under workloads that are approaching close to CPU bound, there will often be a pattern with most of the Queue Depth arriving at the Server in one IDLE-BUSY cycle, and one or two other OPs each arriving in a cycle by themselves; in such cases when Queue Depth is high enough it is still better to process all OPs in Nagle mode, rather than requiring the busy_count to count up to the threshold again after each IDLE.

So this refinement checks whether an ending cycle, *or any of the last "few" cycles* before it, survived at least *nagle_margin* OPs after Nagle's mode was enabled; if so, the algorithm reenables Nagle's algorithm immediately after disabling it, so that the next cycle will begin with it enabled from the outset. The busy_count is still reset to zero, to properly track the behavior of the next cycle. The disable/enable pair is done to ensure the release of any outgoing data (partial packets) accumulated in the TCP layer.

The "*few*" number of cycles of history considered is hard-coded to three, including the cycle just ending. It should be as small as possible while still covering typical observed patterns of "short" cycles interspersed among much longer ones.

The tunable *nagle_margin* is the number of OPs the Session Thread must process in a cycle *after* enabling Nagle's algorithm — representing its "CPU bound" status *with Nagle's algorithm enabled*. When going IDLE, the algorithm only retains the socket in Nagle mode if it has recently demonstrated the ability to remain BUSY through an additional minimum *nagle_margin* of OPs *while running with Nagle's algorithm enabled*. Otherwise it leaves the socket reset to TCP_NODELAY mode and the busy_count at zero as described above.

The intent of the above refinement is to make it easier (more likely) for a Session Thread to remain uninterrupted in Nagle mode if it has recently demonstrated a sufficient workload to sustain operation there for a minimum duration.

The intent of the next refinement is to make it more difficult (less likely) for a Session Thread to decide (non-optimally) to enter Nagle mode when it is far from able to sustain continuous operation there. The *nagle_threshold* is part of that control, but by itself is susceptible to being undesirably triggered by "spikes" of the busy_count seen in a small fraction of IDLE-BUSY cycles.

For mitigation, to enable Nagle's algorithm the optimization demands not only that the current IDLE-BUSY cycle reach the nagle_threshold, but also that the average of the last few cycles be at least (nagle_threshold + nagle_margin) / 2 . This filters out most spikes, requiring a certain overall level of busy_count over the past few cycles for the current cycle to be allowed into Nagle mode just at the threshold. The average includes the cycle in progress, so a single cycle can still enter Nagle-mode even after a string of very short cycles, if it counts high enough.

The number of cycles averaged (including the one in progress) is determined by the number of members in the conn->cpu_busy_count[] array, which is hard-coded to 4. It should be as small as possible while still providing effective spike filtering.

Referring to the chart: at low queue depths (here, QD < ~17) the session is not yet CPU bound; in this range the black and blue curves coincide. Below QD=14 Nagle's algorithm is never enabled (because the Session Thread always runs out of work and sleeps before reaching the threshold) and the session operates 100% in TCP_NODELAY mode, just as it does with the optimization absent. From QD=14 to QD=16 Nagle's algorithm begins to engage a small fraction of the time as the Session Thread approaches 100% CPU utilization.

At high queue depths (here, QD > ~76) the Session Thread is 100% CPU bound, and the black Adaptive Nagle curve reflects operation in Nagle mode virtually 100% of the time. In this range the black curve coincides closely with the red "100% Nagle" curve.

There is a fairly wide transitional range of queue depths between those two, where the Session Thread hovers around 90% to 99% CPU utilization — with the Adaptive Nagle IOPS performance slowly rising from a few percent below the NODELAY-mode peak until finally reaching up to the level of the Nagle-mode peak. This transitional range begins at the workload where running in TCP_NODELAY mode gets close to 100% CPU usage *[shortly before the blue IOPS curve flattens out].* When the Session thread gets this busy it soon counts up to the threshold to enable Nagle's algorithm.

With Nagle's algorithm enabled the Server becomes more CPU-efficient — able to process operations at that same queue depth using less than 100% CPU — so it soon runs out of pending work in the TCP receive buffer, resetting the counter to zero and returning to TCP_NODELAY mode before going to sleep (for a short time). Note that if the Session Thread had not switched to Nagle mode it would have remained CPU bound instead. When it wakes up it resumes the counting algorithm from zero.

This illustrates a subtlety in the mentioned "theory" — the difference between "*enabling Nagle's algorithm when reaching CPU bound*" and "*running with Nagle's algorithm enabled while CPU bound*" — in the transitional range these are different because as soon as Nagle's algorithm is enabled, the increased CPU-efficiency soon makes the Session Thread no longer CPU bound, and it will go IDLE after draining all Requests from its TCP receive buffer.

Note, however, that this drain is not complete until the Session Thread finds the TCP receive buffer empty when it looks there for another Request; so the drain ***includes the processing of any new Requests arriving*** at the receive buffer before it finally becomes empty. This drain can take quite some time to complete, with the Session Thread sometimes processing many multiples of the Queue Depth before finally catching up and exhausting the buffer. (For Read operations this time depends on the difference between the Response network transmission time and the operation's CPU processing time — the closer they are to each other, the longer it may take.)

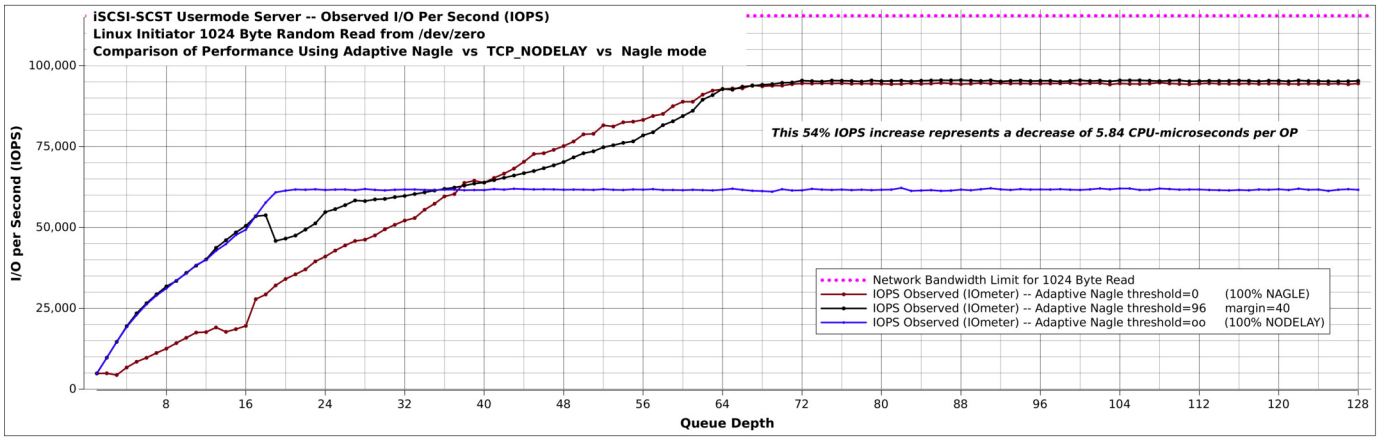
The fraction of time Nagle's algorithm spends enabled rises from nearly zero at the bottom of the transitional range to nearly 100% at the top of the range. The rise starts out slowly, with more of the increase occurring nearer the top of the range *[see black curve].*

As the queue depth approaches the top of the transitional range, the load eventually becomes sufficient to keep the CPU 100% busy, even *with* the added efficiency from Nagle's algorithm *[where the black curve merges with the red curve and flattens out].* At that queue depth, even after the Session Thread enables Nagle's algorithm it is still unable to drain its TCP receive buffer and remains awake and in Nagle mode CPU bound virtually 100% of the time.

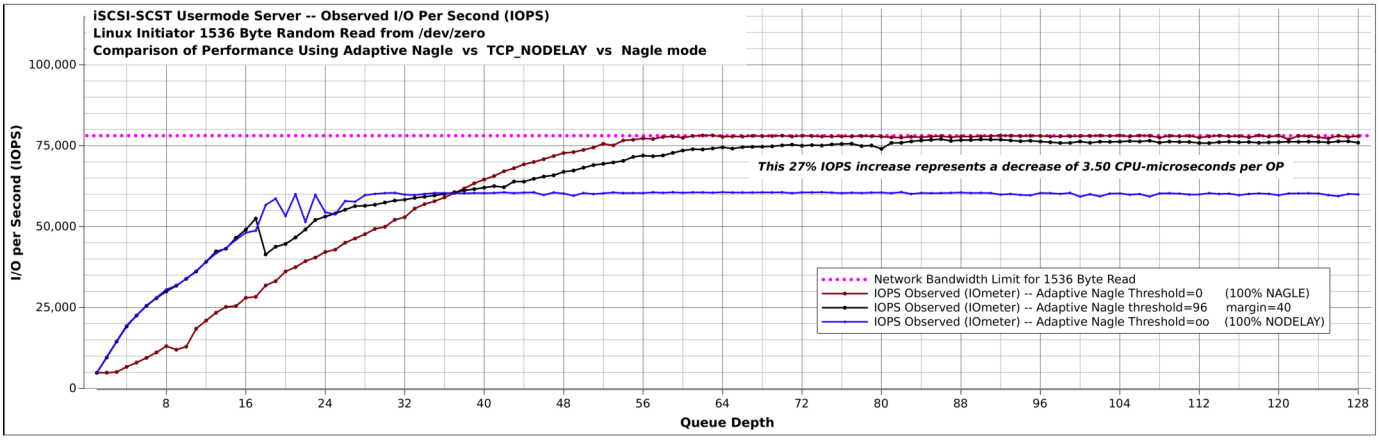
It seems like this approach ought to behave well at times when load gets high and many Session Threads are ganged up on few CPUs — a Session Thread should effectively see a fractional CPU, performing as though its CPU had been set to a lower clock speed. The per-session "CPU-bound" determination should trigger at a lower session Queue Depth in that case, resulting in an earlier switch to the more efficient operation in Nagle mode, at a time when it is most needed (total system CPU bound). (This remains to be tested.)

As seen in the chart *above*, showing 512-Byte Random Read from /dev/zero through a single session over 1 Gb Ethernet on a single 2.4 GHz CPU: the described *Adaptive Nagle* optimization *[black curve]* improves peak throughput performance from around 63,000 IOPS to **100,000 IOPS** — 60% increase as compared to running 100% in TCP_NODELAY mode — with no adverse impact below Queue Depth 17, and only a modest performance decrease of up to 13% in Queue Depth range 17 to 40. It still doesn't saturate the network, but now it peaks CPU bound at about 46% of network capacity instead of about 29%.

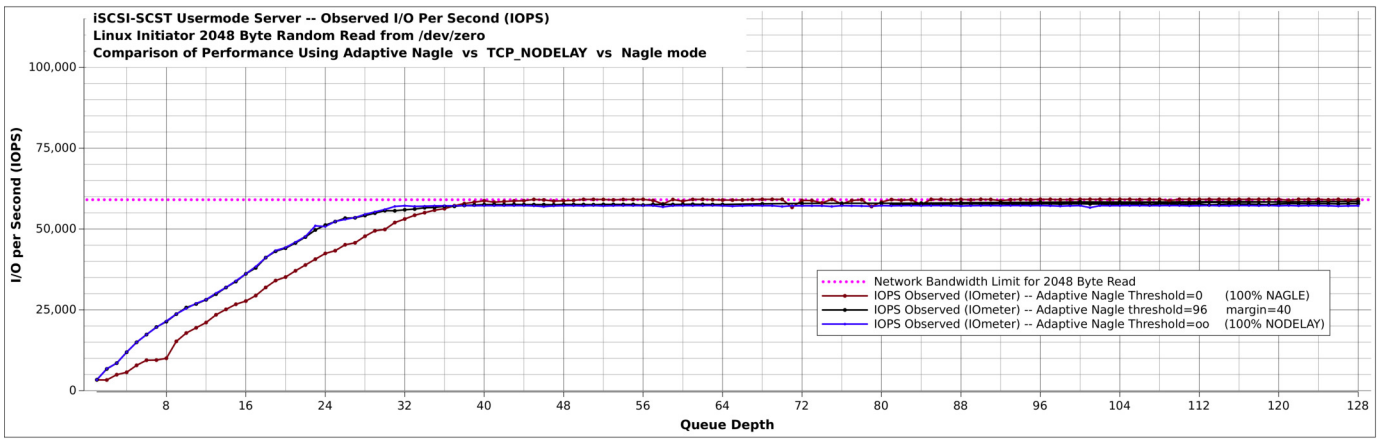
Similar to the 512-Byte chart above, these next charts compare Random Read IOPS of TCP_NODELAY *[blue]* versus Adaptive Nagle *[black]* for CPU-bound I/O sizes 1024, 1536, and 2048 Bytes, with the red curves showing for comparison the performance with Nagle's algorithm enabled 100% of the time. At 1024 Byte Read size neither TCP_NODELAY mode nor Adaptive Nagle were able to saturate the 1 Gb network *[dotted magenta line at very top of chart]*; but the Adaptive Nagle performance reached more than 50% higher, up to 95,500 IOPS as compared with 62,200 IOPS for TCP_NODELAY.



At 1536 Byte (3 sector) Read size, the TCP_NODELAY mode performance *[blue]* peaks at 60,700 IOPS due to reaching CPU bound; whereas 100% Nagle-mode *[red]* peaks at 78,200 IOPS due to reaching network bound *[dotted magenta]*. Adaptive Nagle performance almost reaches network bound, at 78,000 IOPS.



Reading 2048 Bytes (at standard CPU speed of 2.4 GHz), TCP_NODELAY mode was nearly able to saturate the network; so very little gain was seen from Adaptive Nagle at that size, as seen in this chart, and no gain at larger sizes.



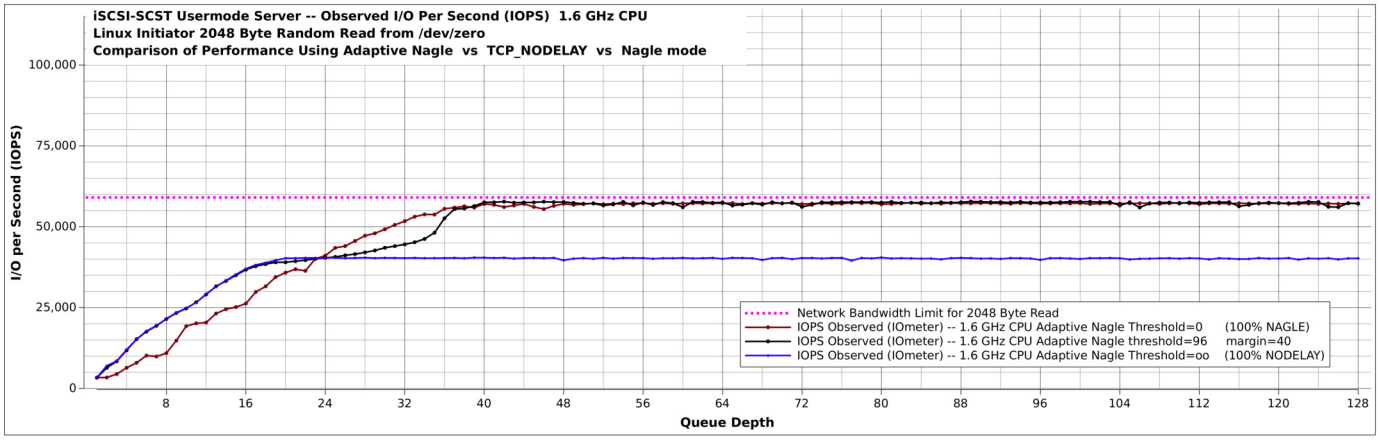
This table shows the peak IOPS reached at a selection of I/O sizes when running 100% in TCP_NODELAY mode (as in standard SCST), and when running with the Adaptive Nagle optimization enabled (default parameter settings). Above 2048 Byte size they are the same.

/dev/zero Peak Read Throughput — Single Session 1 Gb Ethernet, 2.4 GHz CPU

(All peak Read IOPS numbers for I/O sizes <= 16,384 Bytes were reached with response times under 1.25 milliseconds

Workload Type	I/O Size (Bytes)	I/O Size (Sectors)	Peak IOPS 100% NODELAY	Difference	Peak IOPS Adaptive Nagle	MBPS (10^6)	%net
CPU Bound	512	1	62,935	+60%	**100,565**	51.49	45.5%
	1,024	2	60,390	+54%	**93,271**	95.51	80.8%
Borderline	1,536	3	60,676	+27%	**77,053**	118.35	98.6%
	2,048	4	57,247	+2%	**58,505**	119.82	99.1%
				---- Peak IOPS (both) ----			
Network Bound	3,072	6		**39,015**		119.85	98.4%
	4,096	8		**29,577**		121.15	99.0%
	16,384	32		**7,544**		123.60	100%
	65,536	128		**1,891**		123.92	100%
	262,144	512		**473**		124.00	100%
	1,048,576	2048		**118**		124.00	100%

The next chart shows 2048 Byte Reads (as in the chart above), but this time running with the **CPU clock frequency reduced by 33%** to simulate a slower processor. After reducing the CPU speed from 2.4 GHz to 1.6 GHz, performance of TCP_NODELAY Reads from /dev/zero fell 29%, becoming CPU bound at about 40,500 IOPS; while the Adaptive Nagle optimization was able to maintain 98% network saturation at 57,800 IOPS — about 43% higher than the TCP_NODELAY throughput at this reduced CPU speed.



The busy-count threshold and margin for enabling Nagle's algorithm can be adjusted to tune performance. Setting the threshold to zero results in running with Nagle's algorithm enabled 100% of the time *[red curves]*, *without* a flush of outgoing packets upon transition to IDLE. Setting threshold to 1 or higher enables a release of pending partial packets prior to each sleep — if threshold is one and margin is zero then all writes are done with Nagle's algorithm enabled and flushes occur each time going IDLE.

Setting the threshold very large ("to infinity") results in running TCP_NODELAY mode 100% of the time *[blue curves, same as the behavior of a standard kernel-resident installation of SCST]*. All the Adaptive Nagle test runs discussed above used a busy-count threshold of 96 for enabling Nagle's algorithm *[black curves]*.

I think the nagle_margin should be set at or above the Queue Depth of the Nagle/NODELAY performance crossover point. The margin denotes the minimum operation count needed, *while running with Nagle's algorithm enabled*, to consider the Session Thread to be CPU bound. That information is used to decide whether to retain the Session Thread in Nagle mode across an IDLE. Since the crossover point is where it becomes advantageous to run with Nagle's algorithm enabled, it seems like the nagle_margin should be no less than that. (I set the default at 40, used in all my testing, which was around the crossover point for all CPU-bound workloads.)

Tuning of the threshold depends on multiple factors. The threshold somewhat influences the width and boundaries of the transitional range of Queue Depths. Below that range the socket is in TCP_NODELAY mode 100% of the time; and above the range the socket is in Nagle mode virtually 100% of the time. More pronounced is the threshold's influence on the *shape* of the performance curve within the transitional range. Within the transitional range the fraction of time spent operating in Nagle mode increases non-linearly from 0% to 100% as Queue Depth rises through the range. Non-optimally, the curve is not always monotonically increasing in all regions.

The model here is "probabilistic" because there is always some amount of variance in the timing of network transmission and thread scheduling, which (except at very low Queue Depths) leads to different IDLE-BUSY cycles reaching different maximum operation busy-counts during execution of the same workload; all of which get averaged out over many cycles to reach the measured results. Such variance would presumably be magnified when running over a network susceptible to traffic congestion (these tests were point-to-point).

Within the transitional range, *increasing* the threshold makes it harder (less probable) for the busy-count to reach it, thus making the tendency to switch to Nagle mode less aggressive — meaning that at a given workload the Session Thread is less likely to reach Nagle mode during any given IDLE-BUSY cycle than it would be with a lower threshold. Ultimately each of these probabilities translates into an average **fraction of time that Nagle's algorithm is enabled** during operation over a given workload.

A *decreased* (*more* aggressive) threshold results in the Session Thread more easily counting up to the threshold and switching to Nagle mode, reaching it sooner and in a larger percentage of IDLE-BUSY cycles. Above the Queue Depth of the crossover point this results in Nagle's algorithm being enabled a larger fraction of the time, where that is beneficial.

However, when the Queue Depth is below the crossover point a more aggressive threshold can be detrimental to performance: the scheduling latency sets a lower bound on the duration of an IDLE period, and switching to Nagle mode on a workload that too quickly returns the Session Thread to IDLE has the effect of limiting the ratio of BUSY/IDLE time.

So optimal performance depends on the threshold being set **"*high enough*"** to delay (much) activation of Nagle's algorithm until after the Queue Depth is large enough to overcome the scheduling latency. (See *Performance Model* in a later section.)

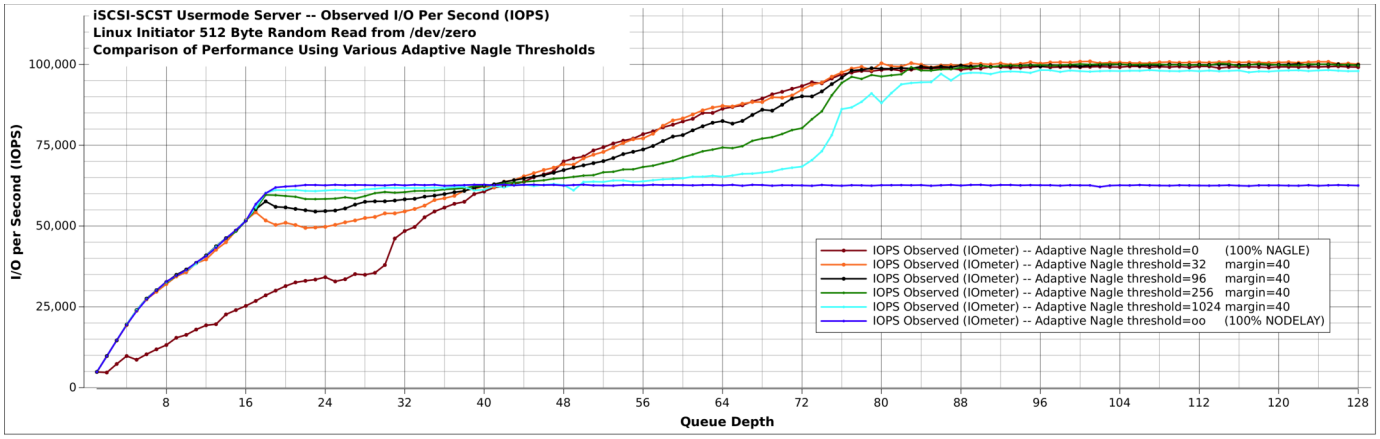
Above the crossover point, on the other hand, an increased (less-aggressive) threshold results in the Session Thread taking longer to count up to the threshold and return to operating in Nagle mode after falling back to TCP_NODELAY mode for whatever reason. That results in the socket spending a larger fraction of time operating in the less-efficient NODELAY mode, with accompanying performance decreases in that range of Queue Depths.

The mentioned refinements avoid returning to NODELAY mode for natural IDLE-BUSY cycles, but do not filter out all the sporadic spikes. So optimal performance depends on the threshold being set ***"not too high"***, so that CPU-bound sessions return to Nagle mode ASAP.

Adding to the earlier 512-Byte Adaptive Nagle comparison chart, the next chart overlays performance curves for three alternative settings of the Nagle-activation threshold. Recall the black curve represents Adaptive Nagle performance with the threshold set to 96.

The orange curve represents performance with the threshold reduced to 32 — in this case Nagle's algorithm is enabled more aggressively than with the other settings, leading to more of a falloff in performance *below* the crossover point because in that range the performance is better off staying away from Nagle's algorithm and its red curve down there halfway to the bottom.

On the other hand once the workload Queue Depth passes above the crossover point the orange curve looks more pleasing, closely following the target red Nagle-mode curve; again this is because this setting is more aggressive about enabling Nagle's algorithm; but when above the crossover point that has a *beneficial* effect.



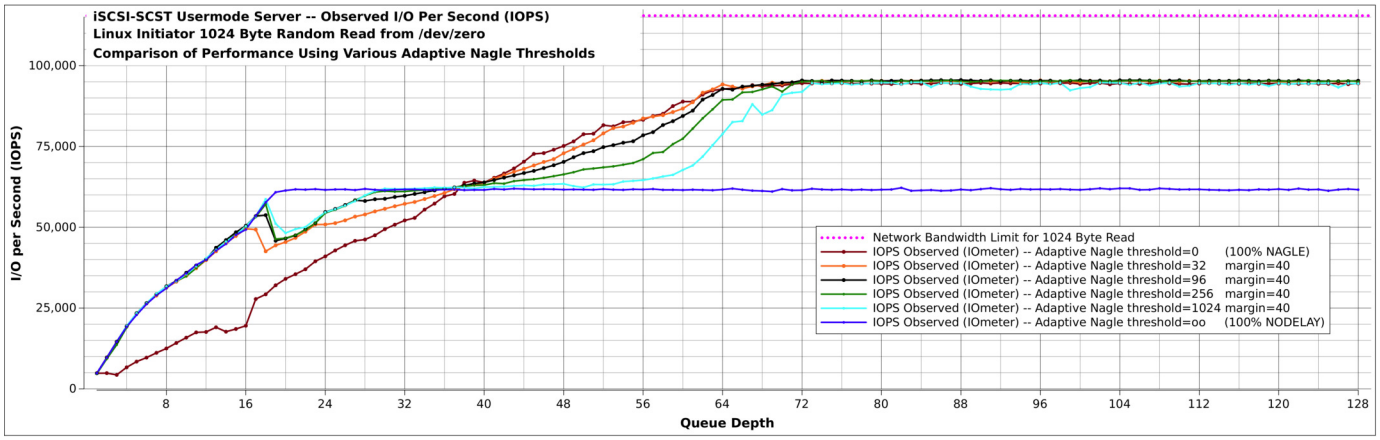
The green and cyan curves represent performance with the threshold raised to 256 and 1024, respectively, making activation of Nagle's algorithm *less* aggressive. Below the crossover point, raising the threshold narrows the gap of Adaptive Nagle from *[blue]* NODELAY performance; but this occurs at the expense of widening the gap to the Nagle-mode *[red]* target performance *above* the crossover point — and by a *much* wider margin than the narrowing benefit seen below the crossover point.

Although I settled on a threshold setting of 96 for most of my testing *[black curves]*, it is arguable that the orange curve for threshold 32 would be superior; someone working with a Queue Depth of 64 would probably think so, but maybe not someone working at a Queue Depth of 24. At or below Queue Depth 17 it makes no difference.

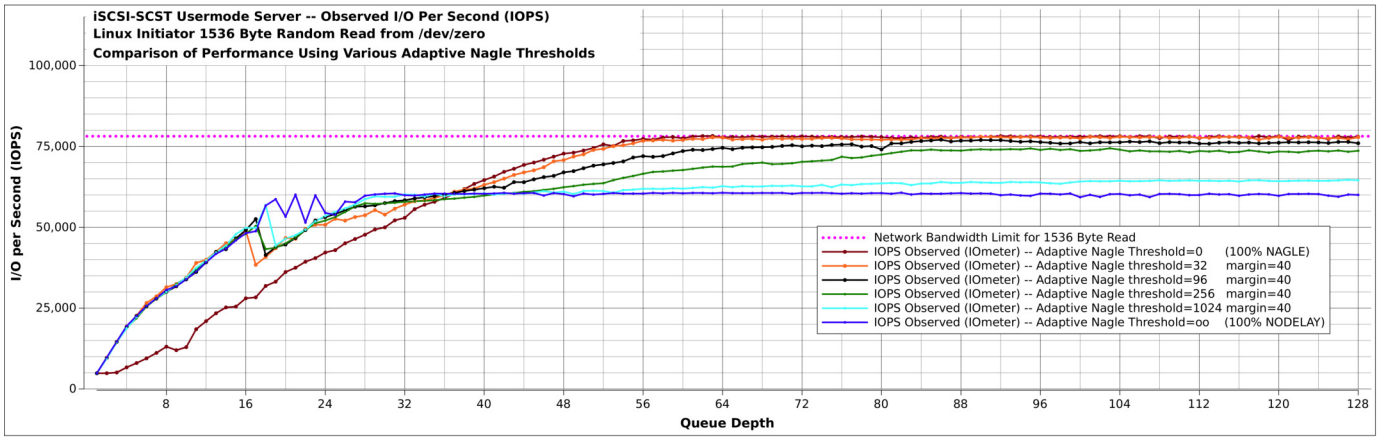
[Referring to the chart above, and recapitulating an earlier description more generally] Within the transitional range and below the crossover point: performance of the *lower* threshold settings *[red, orange]* falls off from the *[blue]* TCP_NODELAY performance more than that of the higher threshold settings *[green, cyan]*, because Nagle's algorithm is enabled more frequently and Queue Depth isn't yet high enough to completely overcome the scheduling latency. The *higher* threshold settings delay activation of Nagle's algorithm longer, keeping the Adaptive Nagle performance closer to the TCP_NODELAY performance below the crossover point.

Above the crossover point: higher threshold settings *[green, cyan]* delay activation of Nagle's algorithm, raising the Queue Depth necessary for the Adaptive Nagle algorithm to reach the performance of 100% Nagle-mode. Higher thresholds also appear to tend toward less smooth curves above the crossover point. Provided the threshold is not *too* high, the same performance level is attained as with a lower threshold, but a higher Queue Depth is required to reach it. (In the plot above, the cyan curve does not reach quite all the way to the other curves, indicating its threshold of 1024 is "too high".) Lower threshold settings *[orange]* enable Nagle's algorithm at lower Queue Depths, keeping the Adaptive Nagle performance closer to the "100% Nagle" performance *[red curve]* above the crossover point.

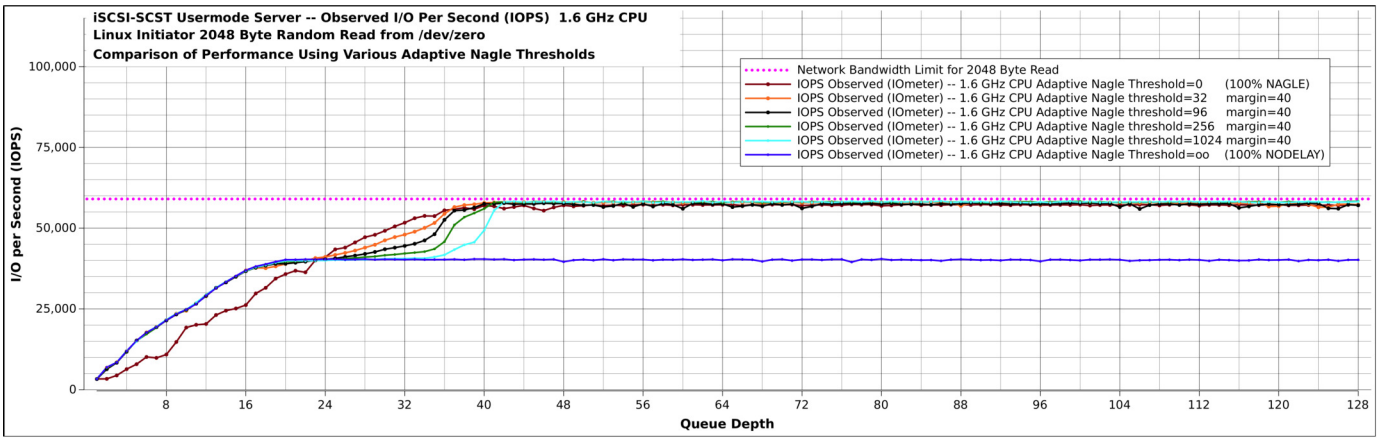
Below are similar charts for 1024 and 1536 Byte Reads. Notice with larger I/O sizes (later charts) the Adaptive Nagle IOPS flatten out at lower Queue Depths, peaking about 16 OPs lower at 1536 Bytes than at 512 Bytes; but the Queue Depth of the crossover point only decreases by five OPs between those two I/O sizes.



The network bandwidth limit for a 1536-Byte (3 sector) Read workload is a little over 78,000 IOPS *[dotted magenta below]*. At this size the 100% TCP_NODELAY *[blue]* performance becomes CPU bound at around 61,000 IOPS. At threshold=32 *[orange]* the performance reaches the maximum and becomes network bound; at threshold=96 *[black]* it reaches about 77,000 IOPS, 99% of maximum. A threshold setting of 1024 *[cyan]* is clearly "too high" at this I/O size, seeing its curve reaching less than 65,000 IOPS, not much better than 100% TCP_NODELAY mode. Even the *[green]* threshold=256 curve never reaches all the way up to the *[red]* 100% Nagle-mode performance level, peaking at about 95% of network bandwidth capacity. (The jitter in the *[blue]* 100% TCP_NODELAY curve at Queue Depth 16-24 occurs consistently.)



The next chart shows observed 2048-Byte IOPS performance at various Adaptive Nagle threshold settings, with the **CPU clock frequency reduced from standard 2.4 GHz to 1.6 GHz**: Recall the crossover point didn't move much when changing I/O sizes at 2.4 GHz, staying in a range around 36 to 42; but here with the reduced CPU speed the crossover point jumps down to about 23.



Although I've mentioned qualitatively some of the influences on performance of raising or lowering the threshold, I have no calculation or quantitative theory to predict an optimal value for it — my tuning here was done empirically. Indeed the tradeoff implicit in these performance curves suggests that this is a situation where "optimal" is not entirely objective, existing at least partly in the mind of the beholder.

In the absence of a fully autotuning algorithm, the threshold and margin should probably be runtime settable through procfs/sysfs. It is possible that a suitable default threshold depends on factors such as CPU speed or Network line speed, but I think those influences should already be factored into the dynamic "CPU-bound" assessment, along with system load. More testing would be appropriate here.

Notice the important distinction between Adaptive Nagle's mechanism for improving performance (enabling Nagle's algorithm), and the optimization decision of when to activate it. For the latter I have used "*Session Thread is CPU bound*" as the decision criterion, with a particular tunable method of deciding *[really, defining]* whether or not that is the current state (counter, threshold, and margin). I use this particular criterion because it is the first one that occurred to me as simple to implement and having a plausible theory of operation to support belief in its reasonable behavior under general conditions; and after a little refinement it has worked well enough that I have never gone searching for another.

The point is that an "Adaptive Nagle" algorithm could be based on entirely different decision criteria, some of which might result in a superior Adaptive Nagle curve more closely conforming (than any of the ones shown above) to the blue NODELAY curve between Queue Depth 16 and the crossover point, where that performs better; and/or closer to the red "100% Nagle" curve above the crossover point.

Perhaps nearly optimal activation of Nagle's algorithm would be solved by an oracle that could answer whether the current workload is in the region of Queue Depths above or below the NODELAY/Nagle performance crossover point; and enabling Nagle's algorithm (or not) based on the oracle's advice. Then I think the Adaptive Nagle curve should match perfectly with the blue curve below the crossover point, and with the red curve above the crossover point.

In the meantime I think I would consider the *orange* curve's threshold of 32 a better *default* than the black curve's threshold of 96. Where this makes the most difference is at 512 Byte I/O size. I chose the less-aggressive 96 for these tests due mainly to being "conservative" about degrading performance at any Queue Depth as compared with the *status quo* TCP_NODELAY version; and in the range QD17 to QD40 the 512-Byte orange Adaptive Nagle (threshold=32) curve falls off much farther (from the blue NODELAY curve) than the black Adaptive Nagle (threshold=96) curve does.

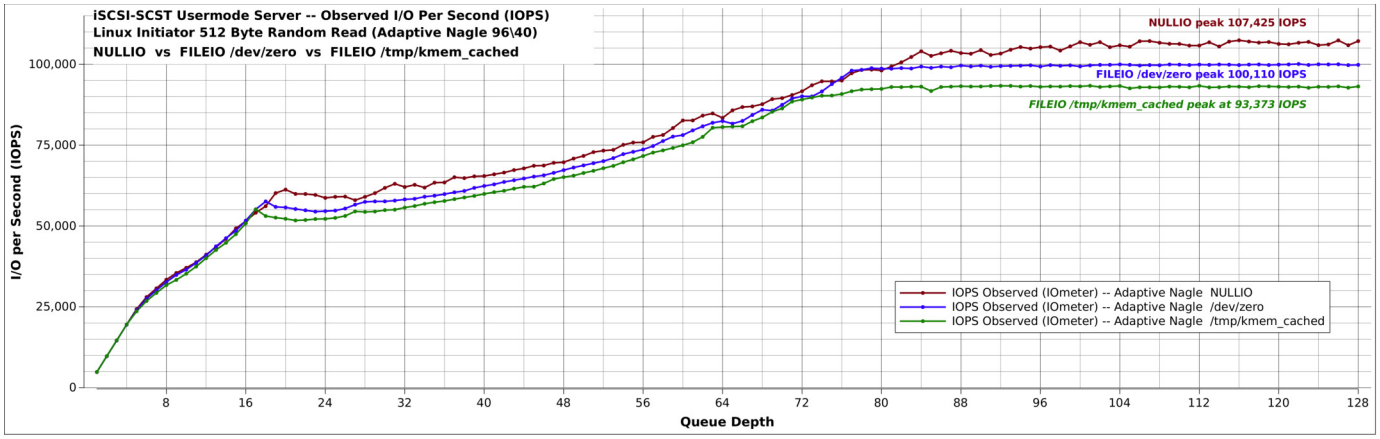
On the other hand, *above* the crossover point the orange curve stays much closer than the black curve to the target red curve, virtually on top of it at 512-Byte I/O size. The orange curve also shows at least as much superiority over the black curve at the other CPU-bound I/O sizes, visible in the previous few charts. At size 1536 Bytes the black curve takes much longer to reach peak (never actually quite reaching the dotted magenta limit), whereas the orange curve catches up with the red 100% Nagle curve and reaches maximum somewhere around Queue Depth 56.

Considering all affected (CPU-bound) I/O sizes, comparing the orange (threshold=32) curves with the black (threshold=96) curves: it seems to me that the increased performance shown by the orange curves above the crossover point would usually be considered to outweigh their decreased performance below the crossover point. Possibly some other value of the threshold between 32 and 96 would be superior to either. I did not search that space.

The next chart compares performance of **512-Byte Read of NULLIO, /dev/zero, and /tmp/kmem_cached FILEIO**, when the Adaptive Nagle optimization is active. In the similar chart for TCP_NODELAY mode seen earlier, the two FILEIO curves (/tmp/kmem_cached, /dev/zero) showed the same performance; but here with Adaptive Nagle enabled the speed is high enough to be able to distinguish them.

The gap between peak NULLIO performance *[red]* and peak /dev/zero performance *[blue]* corresponds to a difference of about 0.68 microseconds per 512-Byte Read operation. That gives a reduced upper bound for the CPU time consumed to enter the kernel and read a block from /dev/zero. (Again, some of that time is I/O setup in the SCST logic.)

The gap between /dev/zero *[blue]* and /tmp/kmem_cached *[green]* corresponds to about 0.72 microseconds per OP. That represents the difference in the length of two different pathways through the kernel during the read system call.



The next charts compare IOPS performance of (CPU bound) 512-byte Random Read from /dev/zero, depending on **which CPU runs the Session Thread**. Each of the four colored curves in a chart represents IOPS observed at various Queue Depths on one of the CPUs.

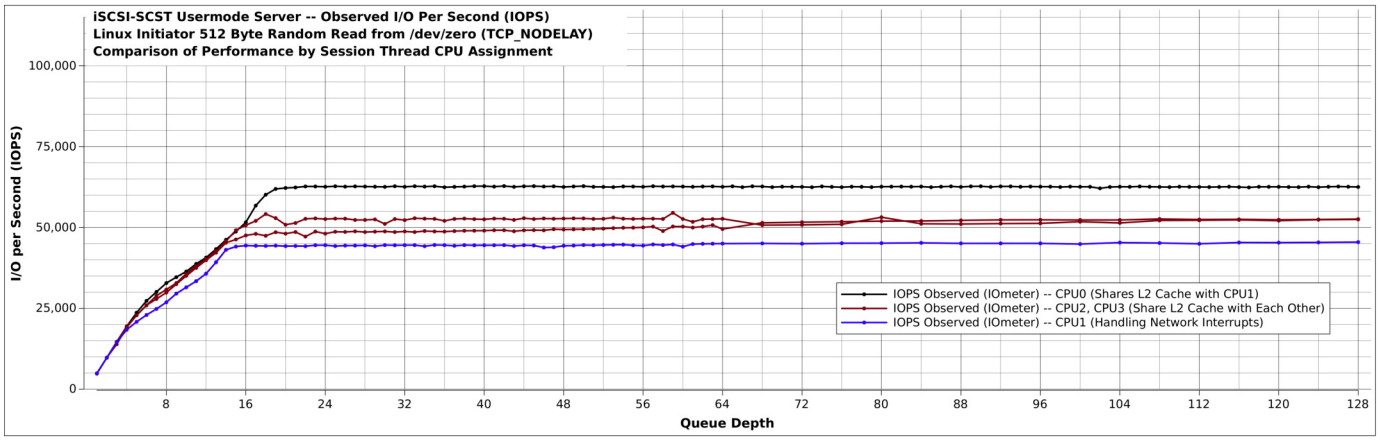
The first chart below shows these variations when running in the standard, TCP_NODELAY mode; whereas the second chart shows measurements taken with the Adaptive Nagle algorithm enabled (configured at its default threshold=96, margin=40). Both scenarios show a marked performance difference depending on CPU assignment.

In these two charts, the blue curves represent performance running the Session Thread on the CPU handling network interrupts; these are the lowest curves because the Session Thread does not receive 100% of the CPU time as it does on other CPUs. The network software interrupts consume some portion of the interrupt CPU, 25% or more for 512-Byte Read.

The black curves represent performance running on the L2-twin of the CPU handling network interrupts — that is, the CPU sharing L2 cache with it. This is the best-performing Session Thread CPU.

The red curves represent performance running on the other two CPUs in the quad package — those *not* sharing L2 cache with the interrupt CPU. Those curves are not as high as the black curve, the difference presumably due to the difference in L2 cache sharing.

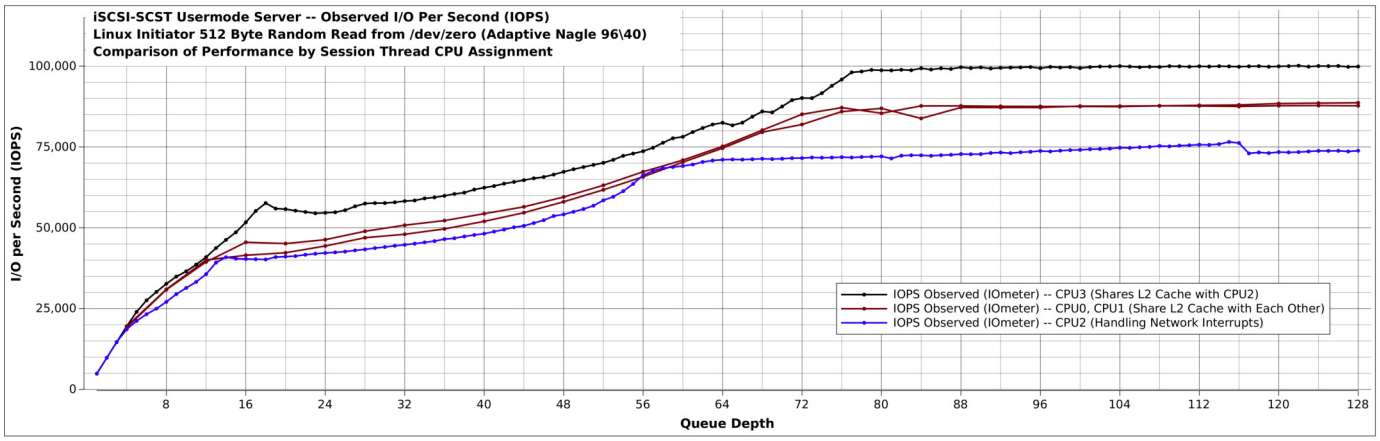
All the measurements discussed earlier were done with the session thread assigned to the L2-twin of the CPU handling the network interrupts — that scenario corresponding to the best of the performance curves shown in the charts here. (Different interrupt CPUs were in use during the measurements for these next two charts.)



In these two charts, the gap between the black and red curves represents the performance difference — for 512-Byte Random Read from /dev/zero — between running the Session Thread on the L2-twin of the CPU servicing network interrupts *[black]*, and running it on one of the other processors not sharing L2 cache with the interrupt CPU *[red]*.

In the TCP_NODELAY chart (above), the gap in performance between these two curves represents a difference of about 2.42 CPU-micro-seconds per operation, with the L2-twin producing 15% more IOPS than the non-L2-twin processors.

Below is a similar chart of performance with the Adaptive Nagle optimization present. Here the gap in performance between the L2-twin and the non-L2-twins represents about 1.72 CPU-microseconds per operation, yielding about 13% more IOPS on the L2-twin. Within experimental variation this most likely represents the same percentage gain as the 15% seen when using TCP_NODELAY mode.



This table lists the Minimum Throughput Time (TPTmin) for Adaptive Nagle Reads of various I/O sizes, calculated as the reciprocal of the peak IOPS observed (best sampled Queue Depth) at each I/O size. Alongside for comparison are the corresponding estimated Response network transmission times computed based on the I/O size, assuming 1 Gb network bandwidth and network MTU=9000 efficiently packed.

At larger I/O sizes the workload is network bound and the observed TPT follows the calculated Response transmission time based on network bandwidth *[green]*.

At smaller I/O sizes the workload is CPU bound and Session Thread CPU time per operation is *computed* as (1 / IOPS) (so the values shown for the Adaptive Nagle CPU time are taken from the TPT column). The TCP_NODELAY per-OP CPU consumption is shown for its CPU-bound workloads at the far right.

The CPU-microseconds listed can be scaled linearly from the assumed 2.4 GHz CPU clock frequency to other Server CPU_GHz speeds by multiplying the CPU-microsecond numbers seen in the table by (2.4 / CPU_GHz). If the result is smaller than the corresponding Network Response time, then the load will be network bound at that CPU speed on the target CPU_GHz processor.

iSCSI READ — Throughput Time (1 / IOPS), Network Reply Time, and Server CPU Time (Microseconds)

	I/O Size (Bytes)	**I/O Size (Sectors)**	**(Peak IOPS) TPTmin µsec (10^6/IOPS)**	**Calculated 1 Gb Net Reply µsec**	**Per-OP 2.4 GHz CPU-µsec (Adaptive Nagle)**	Per-OP 2.4 GHz CPU-µsec (100% NODELAY)
CPU Bound	512	1	**9.94**	4.52	**9.94**	15.89
	1,024	2	**10.55**	8.66	**10.55**	16.08
Borderline	1,536	3	**12.78**	**12.80**		16.48
	2,048	4	**16.90**	**16.94**		17.47
Network Bound	3,072	6	**25.63**	**25.21**		
	4,096	8	**33.81**	**33.48**		
	16,384	32	**132.56**	**132.77**		
	65,536	128	**528.84**	**529.92**		
	262,144	512	**2114.00**	**2118.51**		
	1,048,576	2048	**8456.26**	**8472.88**		

Performance Model (Introduction)

The whole picture has too many moving parts to explain in one pass. We start with a basic model and then add to it in steps. Although the analysis is complicated, it doesn't use any math beyond middle-school algebra, and the diagrams are intended to provide some intuition.

The model should provide a good basis for understanding and predicting qualitative behavior, but any quantitative predictions drawn from it must be validated with measurements in the target running environment, because there are many factors in my configuration which were never varied over the course of the testing, and some of those could be hidden implicitly in the models developed in part from analyzing the test results. I should also mention that this analysis has yet to be reviewed by another software engineer or mathematician, and could contain errors. It *definitely* contains approximations.