I Replaced an SSD with Storage Class Memory. Here is What I Learned.

On April 2, 2019 Intel Optane Persistent Memory became the first commercially available storage class memory (SCM). Like SSD, this memory is persistent, and like DRAM it sits on the memory bus. In the wake of this announcement, system architects began to ponder where it fits in the storage hierarchy. A straightforward question I wanted to answer is whether SCM could simply be a replacement for an SSD for my database application. Surely, cost and density of SCM is not comparable to that of SSD, but imagining the future where SCM becomes a commodity and its price significantly drops, will I reap significant performance benefits from making SCM my primary storage device?

The answer to this question certainly depends on what application I am running and *how* this application is accessing the SCM. SCM is byte-addressable, so it can be read and written byte-by-byte, just like DRAM. Alternatively, SCM can be configured to "look" like a block device, with file system on top and accessed just like any conventional SSD or a hard drive. The former method is very efficient, because nothing stands between the application and the hardware, but it is also very tricky to use, because most applications are not written to treat their memory as persistent. Accessing SCM via the block interface comes with OS and file system overhead, but there is no need to rewrite applications.

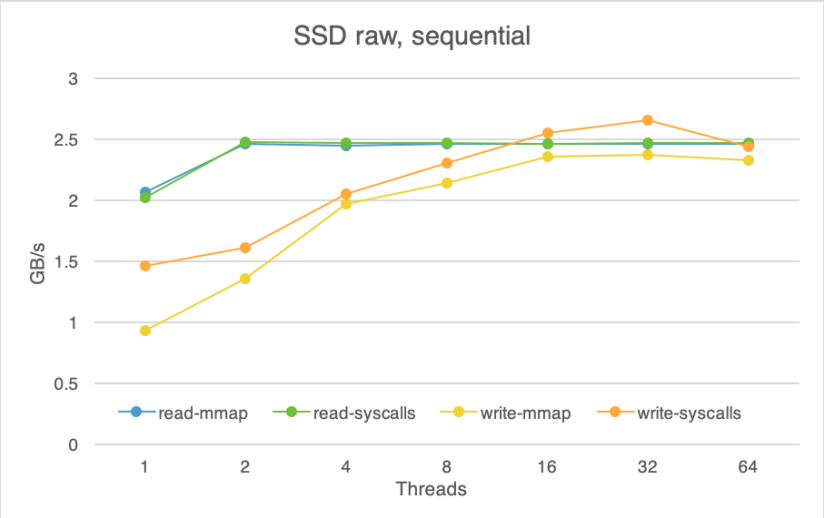
As my test application I used WiredTiger, MongoDB's storage engine. Like many storage engines, WiredTiger is throughput-oriented: it reads and writes data in large blocks and uses its own cache to mask storage device latency. My experiments, therefore, measured primarily throughput. As test hardware, I used Intel Optane DC Persistent Memory (PM) as SCM and Intel Optane SSD P4800X series as SSD. Both devices use the same kind of memory: Intel Optane 3D XPoint non-volatile memory (which is probably a form of ReRAM), while the former sits on the memory bus and the latter on the PCIe.

I chose to access the SCM via the block API, because this was straightforward, requiring no changes to WiredTiger. There is much academic work on adapting storage engines to use SCM via the byte-interface. The engineering effort and performance benefits of various adaptations vary, with one report even showing performance *degradation* relative to using the block interface. While the debate on how exactly to adapt storage engines to byte-addressable persistent memories is still ongoing, I wanted to measure the effects of simply upgrading the block storage to a faster SCM device with no changes to the software.

To begin with, I gauged raw device bandwidth with microbenchmarks that read or write a 32GB file using 8KB blocks. I can use threads to access the file, and if I use more than one, they each access their own chunk of the file. Threads can access the file

either via system calls (read/write) or via mmap; the latter method was reported to have less overhead.

SSD drive's raw performance meets the spec.
According to the spec, my P48000X drive is capable of up to 2.5GB/s sequential read bandwidth and up to 2.2GB/s sequential write bandwidth. Here are the numbers I observed via the Ubuntu Linux (5.3 kernel) raw device API, meaning that there should be no buffer caching.



The read bandwidth behaves according to the specification as long as I use at least two threads, regardless of whether I use mmap or system calls. The write bandwidth, unexpectedly, exceeds its specified upper bound when using multiple threads. I do not have an explanation, but suspect that this could be due to caching either at the OS or at the device.

The Optane P4800X SSD is faster than a typical SSD at the time of this writing, but not to the point of being incomparable. While the Optane SSD offers up to a 2.5GB/s of sequential read bandwidth, a typical NAND SSD (e.g., Intel SSD Pro 6000p series) offers up to 1.8GB/s. The difference is more noticeable with respect to writes. While the Optane drive can deliver up to 2.2 GB/s, the older drive can do no more than 0.56 GB/s.

Random reads and writes on the Optane SSD are not that much worse than sequential ones. The chart below shows the bandwidth for random access (using the mmap). We are able to achieve close to peak sequential throughput with reads and only 10% short of peak sequential throughput for writes. Near-identical performance of sequential and random access is a salient feature of these drives that other researchers have put to good use.

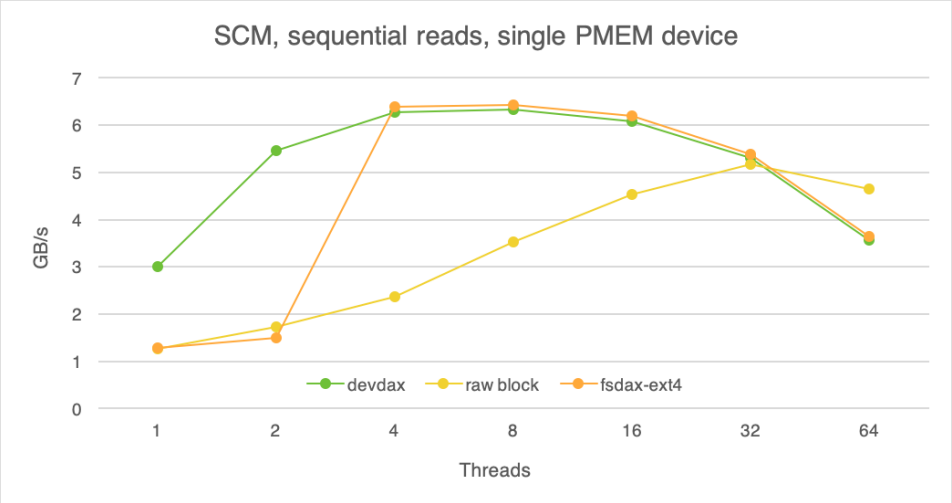
NVRAM offers high-bandwidth reads, but low-bandwidth writes.

Now let us look at the raw performance of SCM. Intel systems supporting Optane PM can fit up to six DIMMs; my experimental system had only two. So I was limited to benchmarking the throughput of a single DIMM and catch a glimpse of scaling across two DIMMs. Luckily I could rely on data from other researchers for scalability across a larger number of DIMMs.

There are three ways to obtain direct access to PM: (1) ***devdax*** -- a PM module is exposed as a character device, (2) **raw block device** – this is the same method that I used for SSD access, and (3) **fsdax** – in this case we have a file system on top of a PM module masquerading as a block device, but file accesses bypass the buffer cache via the Direct Access (DAX) mode. In my experiments I used the ext4 file system.

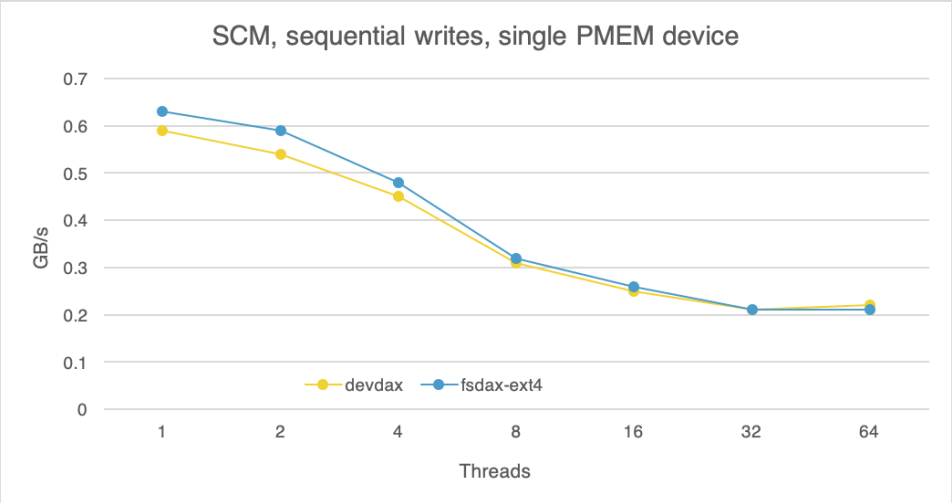
The following chart shows the throughput of sequential reads obtained via these three access methods. In all cases we use mmap, because that is the only method supported by devdax.

Sequential read bandwidth of a single PM module reaches about 6.4 GB/s; that is in the ballpark of what other researchers observed as well, and that was confirmed to be `about right' by an Intel expert. Random access behaves almost identically to sequential access, so I omit the charts.



Both devdax and fsdax methods are able to achieve the peak throughput, but not the raw block access method. There are a couple of reasons for that. First, raw block access performs a copy of the data from the kernel into the user space, even though it shouldn't be doing that, because we use mmap. \footnote{This makes sense: we use the PM module as a block device, and since the raw block access path doesn't know that it is capable of direct access (DAX) it has to put a data copy somewhere before mapping it; since the raw block method bypasses the buffer cache, the data is deposited somewhere in user space.} Second, unlike devdax and fsdax, the raw block access method does not give us access to huge pages. The raw block access method generates about 16 bln TLB loads and 6 bln TLB load misses; fsdax, in contrast, generates only 163 mln loads and 228K misses.

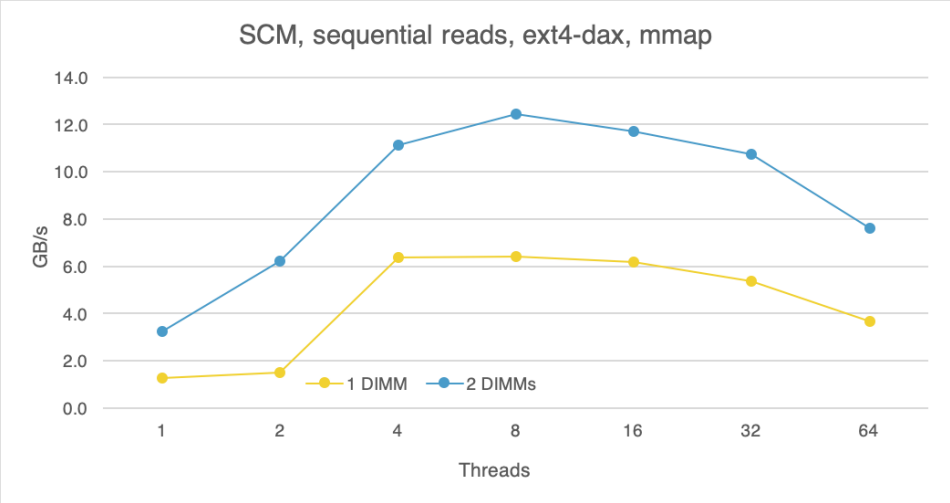
Writes are a very different story. The following chart shows sequential write throughput with devdax and fsdax methods, omitting the raw block method, because of its overhead. Random writes (not shown) show identical behavior.



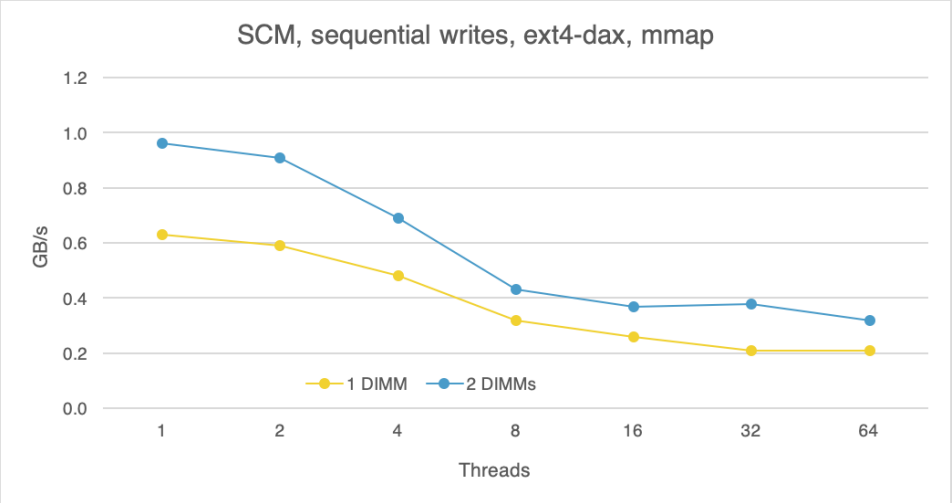
The single-module write throughput achieves a mere 0.6 GB/s. This measurement does not agree with the data from the UCSD researchers who observed around 2.3GB/s write bandwidth on a single device. Their tests wrote to persistent memory using byte interface (via Intel SDK) with non-temporal store instructions or cache-line write-back (clwb) instructions. The methods available via devdax and fsdax are less efficient, even though devdax should not be subject to journaling overhead that might slow down fsdax. Anyhow, my use case involved using PM without any changes to the operating system stack, so I have to pay the price of using the file system. With these constraints taken into consideration, ***a single PM module achieves write throughput comparable only to a NAND SSD.***

Next, let's look at scaling: if my tests use two modules and spread the load evenly across them, will my bandwidth double? If that is the case, then it is reasonable to assume that if I fill all six slots with PM, my bandwidth will increase six-fold.

The following figure shows the measurements for sequential reads, using mmap over fsdax. I used the Linux striped device mapper to spread the load across two DIMMs. With two DIMMs we can almost double the peak read bandwidth, from 6.4 GB/s with one DIMM to 12.4 GB/s with two. Similarly, researchers at UCSD observed nearly linear scaling across six DIMMs.



The following chart shows the same measurements for writes.



We achieve nearly 1 GB/s of write throughput with two DIMMs relative to 0.6 GB/s with one, but the scaling is less than linear if we can extrapolate from a single data point. The USCD researchers observed that bandwidth with six DIMMs improved by 5.6x relative to using a single DIMM, which is in line with my observation. Extrapolating from these data points, if my system had six DIMMs, I'd observe around 3.4 GB/s of peak write bandwidth, which is about 50% greater than what I can get from the Optane SSD.

In summary, with bare device access I see about 2.5 GB/s of peak read bandwidth on the Optane SSD and about 6.4 GB/s on a single Optane PM module. With six modules, the throughput would be ~38GB/s. Write throughput was only 0.6 GB/s on a single PM module, projected to reach 3.4 GB/s with six, while the Optane SSD reached 2.2 GB/s write bandwidth.

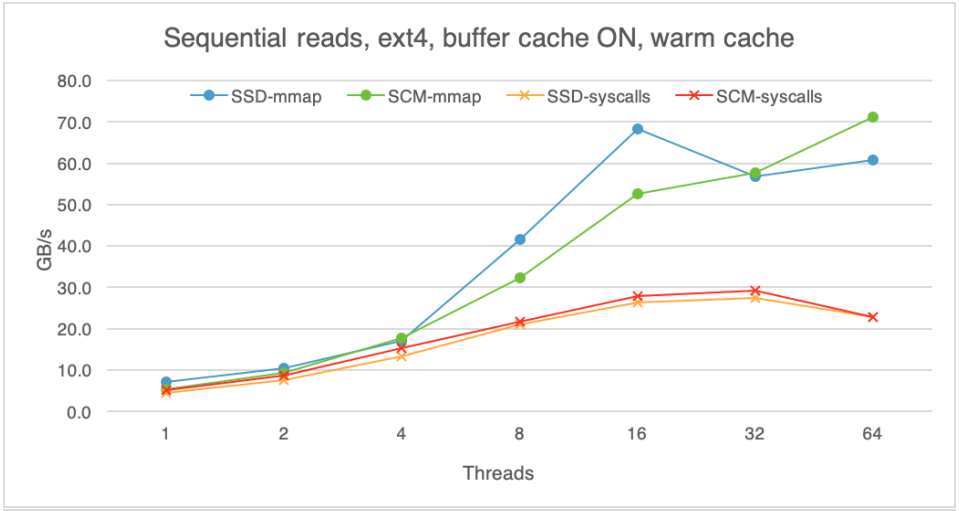
Optane SCM has a significant edge over the SSD with respect to reads, and a small advantage in writes, provided you can afford six PM modules; otherwise, an SSD will deliver a higher write throughput.

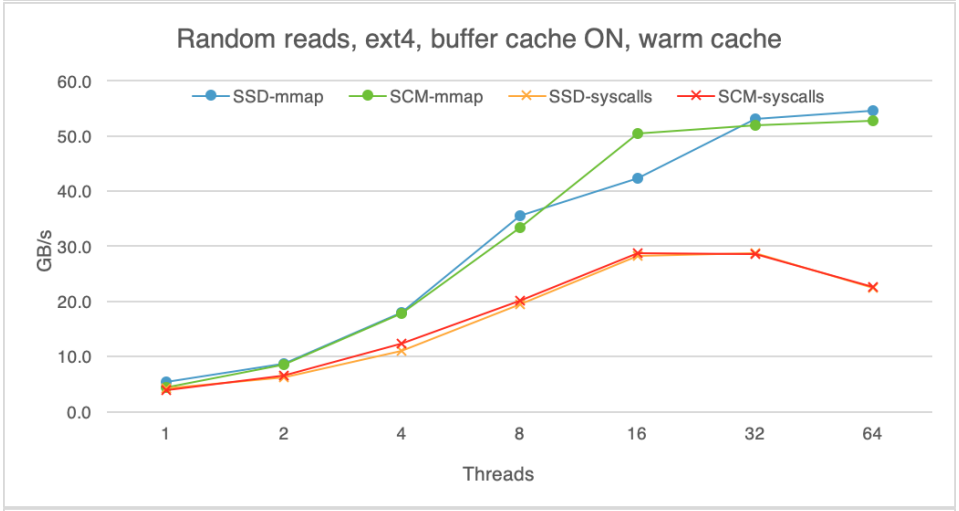
OS and ext4 file system in direct-access mode impose little overhead on reads, but a noticeable overhead on writes on the Optane PM device.

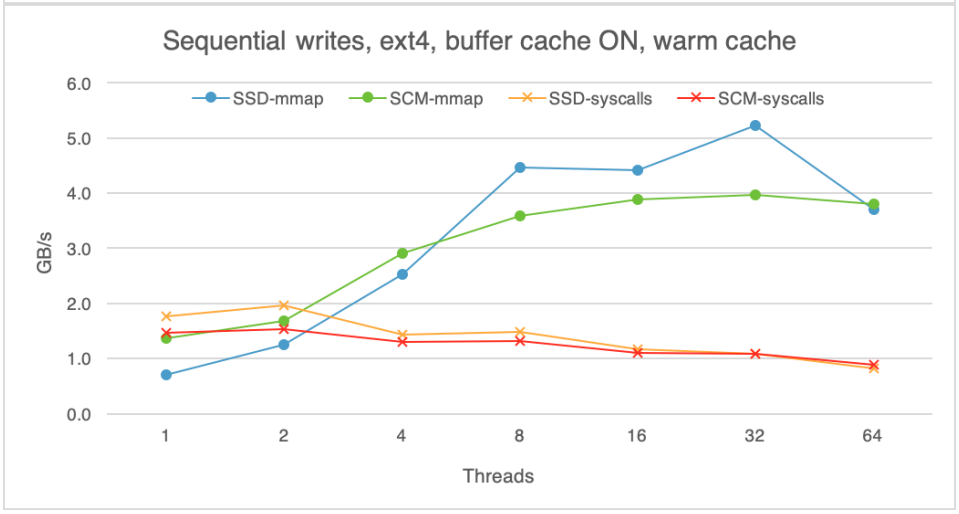
Software caching attenuates SCM performance advantage

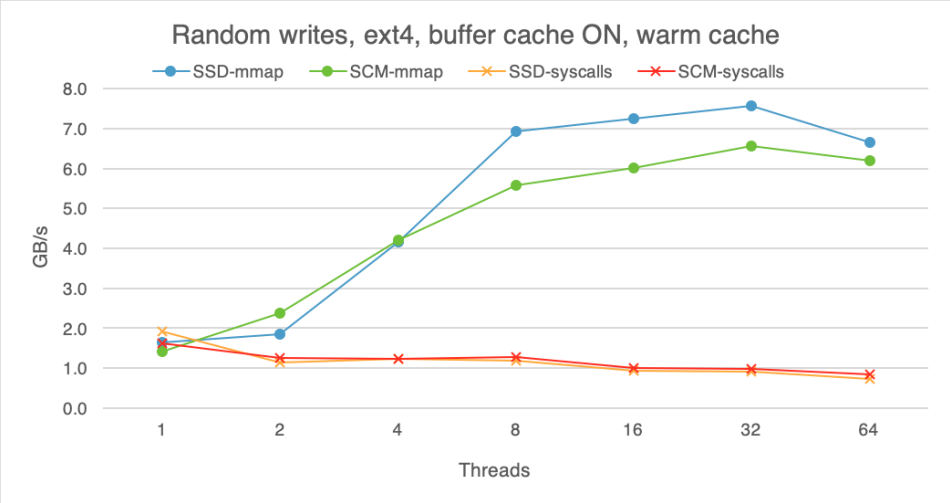
Performance of bare hardware is only part of the story. While SCM is closer to the speed of DRAM than traditional storage media, DRAM is still faster, so advantages of DRAM caching, both in the application and in the OS buffer cache are difficult to overlook. The following charts shows that with the buffer cache on (here I am using ext4 *without* the DAX option), all devices perform roughly the same, regardless of whether we are doing reads or writes, random or sequential access. These experiments were done with the *warm* buffer cache, i.e., the file was already in the buffer cache before I began the experiment, so what we are measuring here is essentially DRAM

performance and various software overheads. (That is the reason why the advantage of using mmap, as opposed to system calls, is quite clear here.)



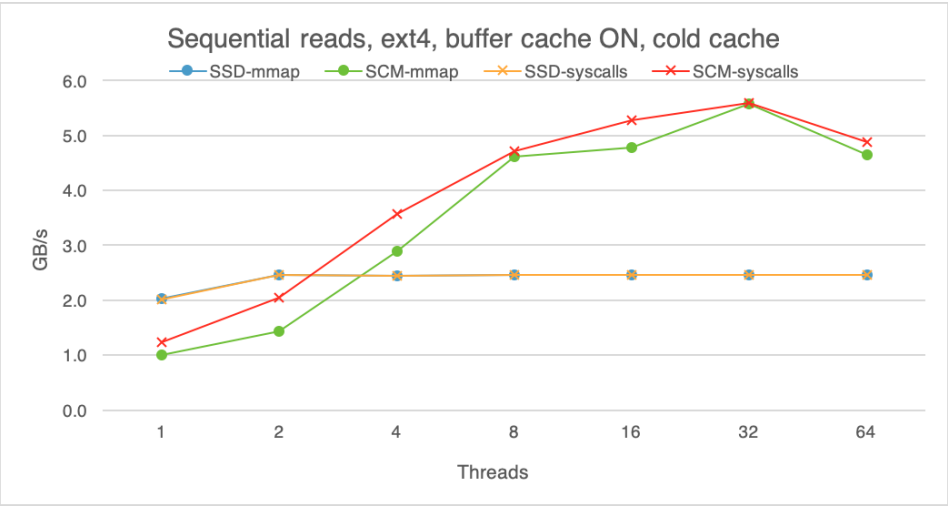


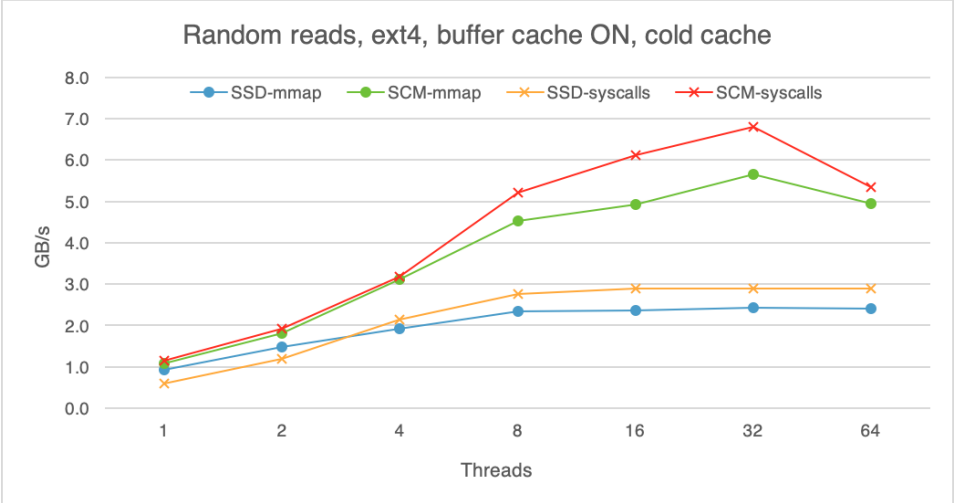


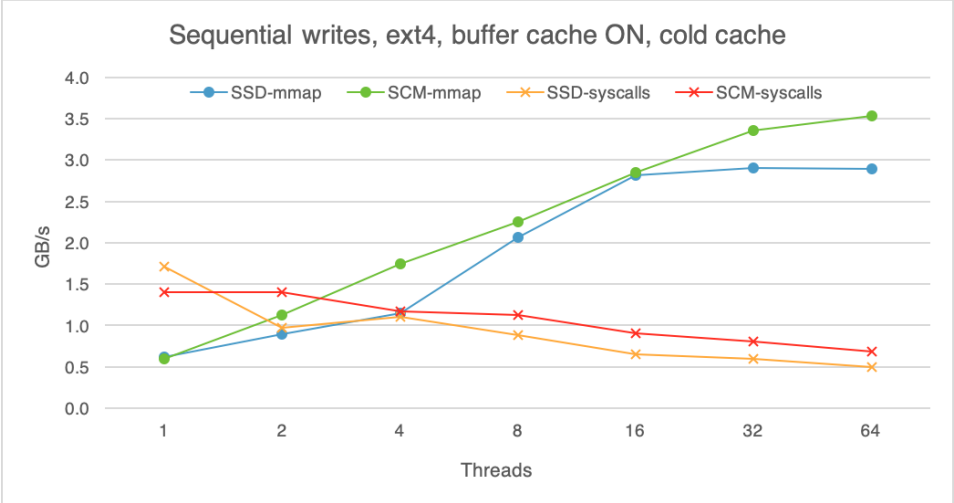


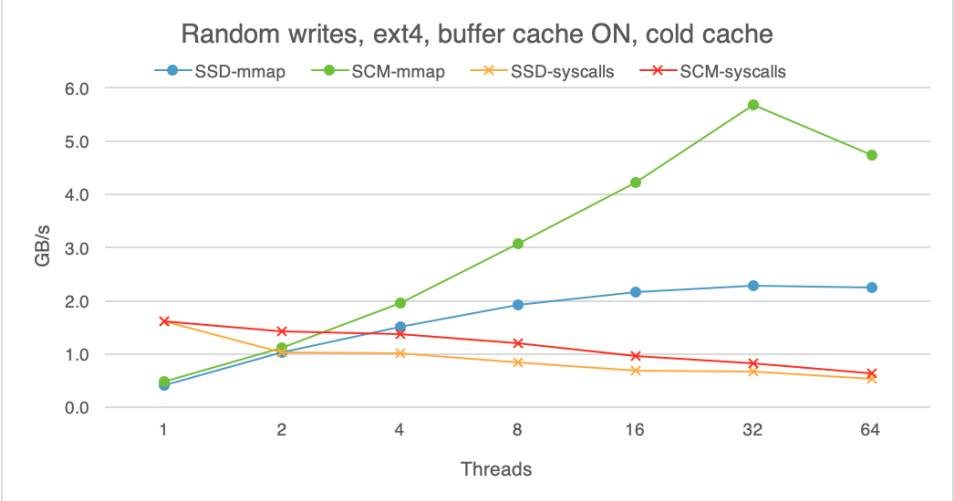
If I begin each experiment with a *cold* cache, the difference between the devices is still visible, but less apparent than with raw access. With cold buffer cache, on the read path the OS has to copy the data from the storage device into the buffer cache before making it available to the application (hence extra software overhead). Furthermore, with buffer cache on, the OS is not using huge pages. *These factors dampen the raw read advantage of SCM.*

For writes, whereas SCM used to deliver lower bandwidth than SSD with raw access, now SCM outpaces SSD, possibly because buffer cache absorbs and batches some of them, instead of flushing each write to the device immediately.









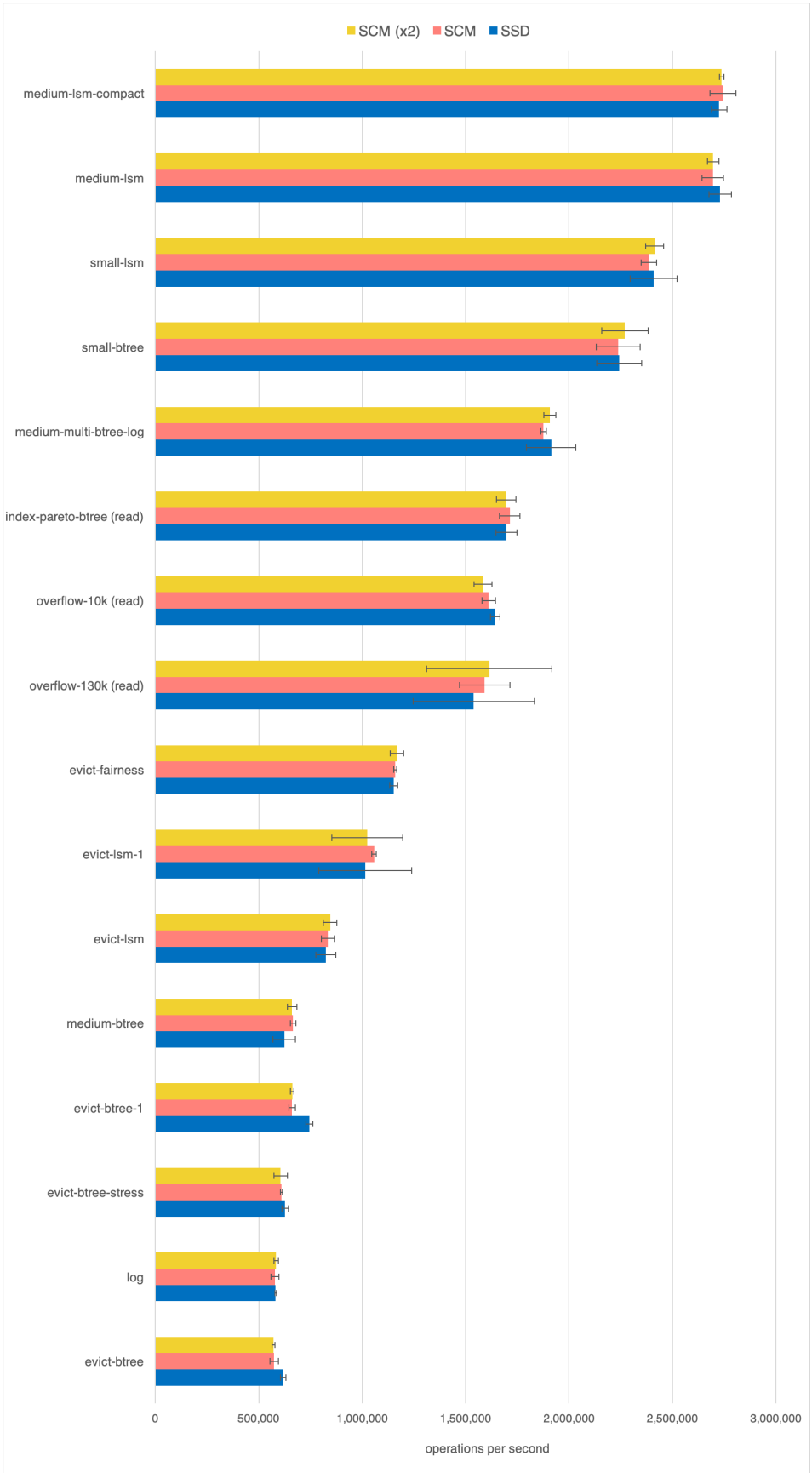
WiredTiger strorage engine internally uses a B-tree data structure, thus many workloads show a high degree of locality (e.g., internal pages are used and reused a lot more often than leaf pages). So it is not surprising that WiredTiger greatly benefits from a DRAM

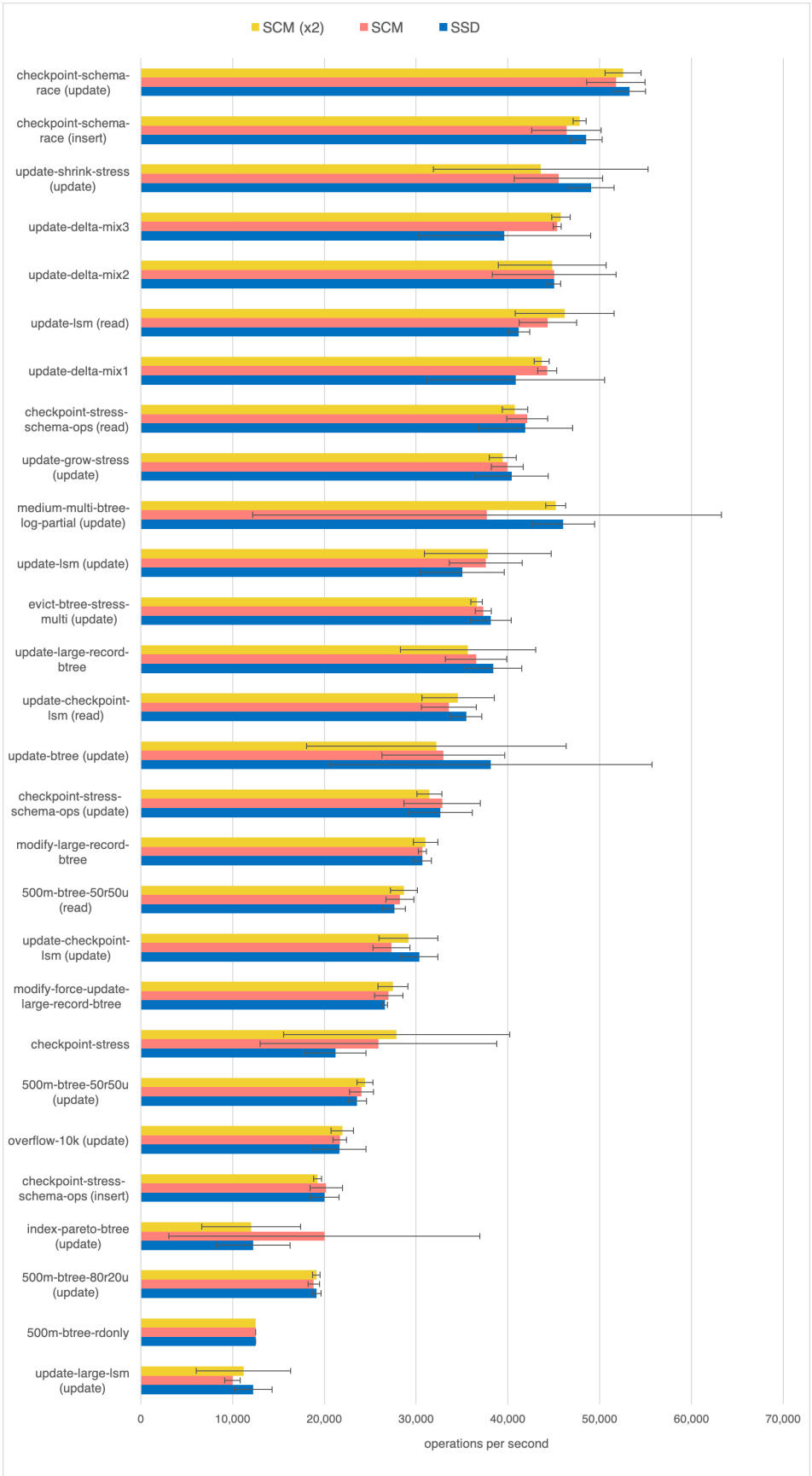
cache. When I ran WiredTiger over SCM with buffer cache on, I observed up to 30x performance improvement relative to bypassing the buffer cache (ext4-dax), and none of the workloads ran faster when it bypassed the buffer cache. That said, my dual socket Intel Gold (Skylake) system has plenty of RAM – 192GB, which is similar to how AWS would configure a system with 64 virtual CPUs. Therefore, DRAM resources for caching where plentiful. If my system did not have that much DRAM for caching, perhaps the speed of the SCM device would be more pronounced.

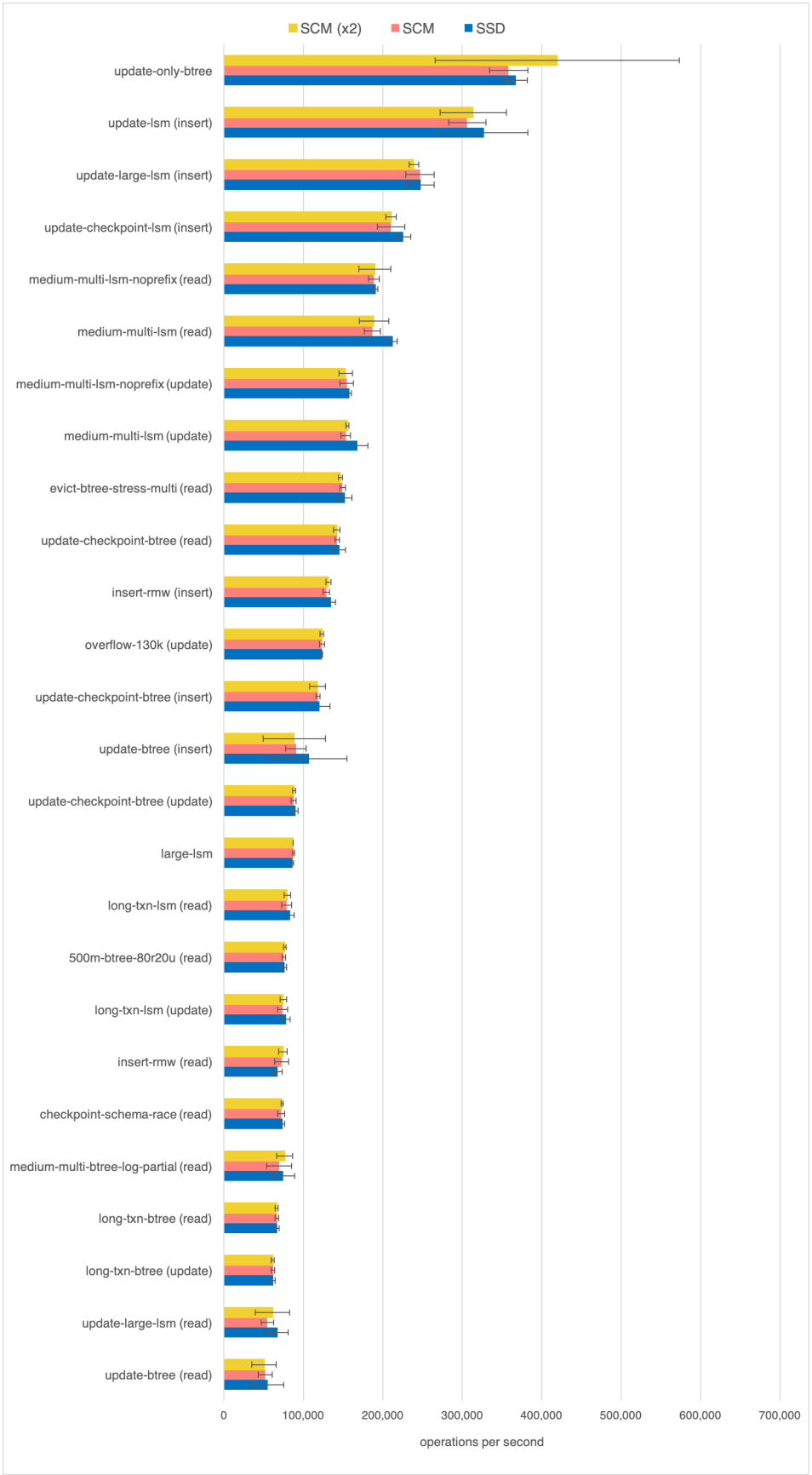
So for my use case, the OS buffer cache is a keeper, and that means that the differences between SCM and SSD will be much smaller than with raw access. On top of that, WiredTiger has its own cache that has extra knowledge about which pages to keep around in memory.

End-to-end, there is no difference between SCM and SSD

The following chart shows the performance of the WiredTiger wtperf benchmark suite on Intel Optane SCM (one and two modules) and on the Optane SSD. We see no significant performance difference between SCM and SSD. Apart from one write-intensive benchmark evict-btree-1, which is *faster* on SSD, there are no statistically significant differences between the two.


SCM (x2)
SCM
SSD
medium-lsm-compact
medium-lsm
small-lsm
small-btree
medium-multi-btree-log
index-pareto-btree (read)
overflow-10k (read)
overflow-130k (read)
evict-fairness
evict-lsm-1
evict-lsm
medium-btree
evict-btree-1
evict-btree-stress
log
evict-btree
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
operations per second


SCM (x2)
SCM
SSD
checkpoint-schema-race (update)
checkpoint-schema-race (insert)
update-shrink-stress (update)
update-delta-mix3
update-delta-mix2
update-lsm (read)
update-delta-mix1
checkpoint-stress-schema-ops (read)
update-grow-stress (update)
medium-multi-btree-log-partial (update)
update-lsm (update)
evict-btree-stress-multi (update)
update-large-record-btree
update-checkpoint-lsm (read)
update-btree (update)
checkpoint-stress-schema-ops (update)
modify-large-record-btree
500m-btree-50r50u (read)
update-checkpoint-lsm (update)
modify-force-update-large-record-btree
checkpoint-stress
500m-btree-50r50u (update)
overflow-10k (update)
checkpoint-stress-schema-ops (insert)
index-pareto-btree (update)
500m-btree-80r20u (update)
500m-btree-rdonly
update-large-lsm (update)
0
10,000
20,000
30,000
40,000
50,000
60,000
70,000
operations per second


SCM (x2)
SCM
SSD
update-only-btree
update-lsm (insert)
update-large-lsm (insert)
update-checkpoint-lsm (insert)
medium-multi-lsm-noprefix (read)
medium-multi-lsm (read)
medium-multi-lsm-noprefix (update)
medium-multi-lsm (update)
evict-btree-stress-multi (read)
update-checkpoint-btree (read)
insert-rmw (insert)
overflow-130k (update)
update-checkpoint-btree (insert)
update-btree (insert)
update-checkpoint-btree (update)
large-lsm
long-txn-lsm (read)
500m-btree-80r20u (read)
long-txn-lsm (update)
insert-rmw (read)
checkpoint-schema-race (read)
medium-multi-btree-log-partial (read)
long-txn-btree (read)
long-txn-btree (update)
update-large-lsm (read)
update-btree (read)
0
100,000
200,000
300,000
400,000
500,000
600,000
700,000
operations per second

Comparing the performance between a single-module SCM and dual-module SCM (SCMx2) we observe that an extra 2x scale in bandwidth capacity of the hardware has no effect on the performance for the reasons discussed earlier. Clearly, I cannot extrapolate from these numbers that using all six modules will not give any extra performance edge, but the data makes one doubtful. If the OS overhead (to some extent) and DRAM caching (for the most part) completely masked the ~6x read performance advantage that a two-modules SCM system had over SSD, how much extra benefit will we see with an extra 3x scale?

Conclusion

While the raw performance of SCM has a clear advantage over a memory-equivalent SSD (at least when it comes to reads), using SCM as a block device will not necessarily give the same performance advantages. In my case, two layers of effective DRAM caching (OS buffer cache and application cache) masked the performance advantages of SCM. OS and file system overhead did play a role in diminishing raw device advantage, but to a much lesser extent than DRAM caching.

For the time being, I won't be replacing my SSD with an SCM block device. Instead, I will be looking for ways to leverage SCM's lower (compared to SSD) latency in places where DRAM caching cannot help. So, next on my list is to investigate how I can use SCM to extend either the application cache or OS buffer cache, so the effective caching area becomes larger.