Design Issues for SCM-friendly Data Structure

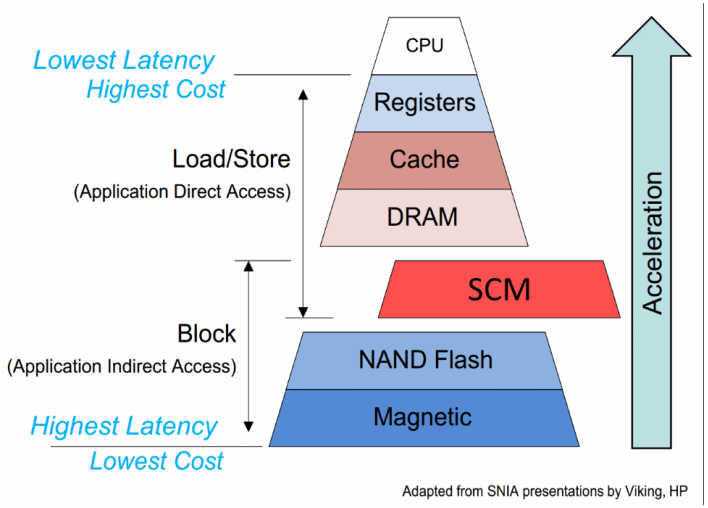
Outline

- **Introduction**
- **Persistent Data Structure Design Considerations**
- **Specific Examples of Microbenchmark Results**
- **Implementation Techniques**

SCM Overview

- **Storage Class Memory (SCM) technology summary:**
  - New layer in the storage hierarchy between DRAM & NAND flash
  - Price, capacity, endurance & perf are between those of DRAM & NAND flash
  - ***Persistent and byte-addressable memory – not block-based SSD!***
  - 128-512GB NVDIMMs announced (Intel/Micron **AEP**/3D XPoint), market availability in the next couple of years, other companies working on alternatives
  - Good for fine-grained, precious information, e.g. metadata, KVSes…

- **SCM challenges:**
  - SCM properties require **redesign of data structures & algorithms** to fully take advantage of SCM
  - Current system architectures (e.g. **volatile CPU caches**) require explicitly maintaining **consistency & durability** on SCM
  - SCM won't help all use cases, need to **choose workloads carefully**



A Note on Terminology

- Same solid-state media chips can be used for memory or storage:

	Storage Class Memory (SCM)	Solid State Disk (SSD)
Semantics	Load/store, fully random access	Block-sized IO
Granularity	CPU cache line	512-4096B or more
Connectivity	Directly to CPU memory bus (DDR-4/DDR-T)	Via peripheral and/or storage buses (PCIe/SAS/SATA)
Access overheads	DAX: direct access, bypasses storage stack	Must pass through storage stack + one or more bus & disk controllers
Latency	Lowest possible outside CPU (e.g. 0.5-2µs for 3DX in AEP)	Higher (e.g. 10-200µs for 3DX in Coldstream)
Examples	3D XPoint in Intel AEP / Apache Pass, DRAM+NAND Flash in NVDIMM-N by Micron/AgigA Tech/etc., NAND Flash in Diablo Memory1, STT-MRAM	3D XPoint in Intel Optane / Coldstream, DRAM+NAND Flash in most SSDs, DRAM in old TMS RamSan SSDs, STT-MRAM in SSDs by Mangstor

- If it's not using memory semantics, not connected to the memory bus, not byte-addressable and must traverse the storage stack – it's **not** a SCM (Storage Class **Memory**), it's something else and calls for a different solution.
  - Intel Optane/Coldstream is **not** SCM, it is a faster Solid State Disk **(SSD)**, even though it's using the same 3D XPoint chips as AEP/Apache Pass

Data Structure Design Considerations

Design considerations

Avoid Traditional Thinking

- When designing SCM-based data structures, avoid being locked in into the traditional system design thinking where access to persistent storage is considered to be so expensive that it makes sense to avoid doing it at almost all costs.

Transient vs Persistent State

- Organization of the data into “transient” and “persistent” portions in traditional RAM + block based storage systems. However, there is much less incentive for strict separation of representations of the persistent state in SCM-based systems .

Keeping Data in DRAM vs SCM

- Decision to maintain a DRAM cache of an SCM-based data structure or to maintain a derived/optimized transient subset of each entry of a persistent data structure in DRAM (e.g. index) depends almost entirely on the projected workload. The most important factors affecting it are: (1)The projected rate of updates of the data stored; (2)The amount of stores into SCM required to implement each update in a consistent and persistent manner.

Design considerations – Cont.

- The concept of "Update Amplification" (UA) in the context of SCM is analogous to that of Write Amplification (WA) in storage.
  - Notable difference from storage WA is that due to the CPU memory/cache subsystem organization, at the lowest level, *updates* of data in SCM often incur the following overheads, not all of which are strictly "writes":
    - First, fetching the data from SCM (*read*) – often causing a CPU cache miss stall and eviction of other useful data from the caches, as well as consuming bandwidth.
    - Then, modifying the data, while carefully maintaining update ordering (*modify*) – usually preventing the CPU from doing other useful work at the same time due to the coarse granularity of the CPU mechanisms available for this task.
    - Finally, requesting that the modified data reach the power-fail protected domain (*write*) – often causing further CPU stalls due to the limited throughput of the internal CPU subsystems responsible for doing that.
  - These overheads initially appear to be insignificant compared to the cost of traditional block storage IO operations. In practice, however, their combined cost is often many times higher than that of the actions the application is trying to carry out (e.g. increment an integer), and usually much higher than the corresponding RAM-only update overheads.

Design considerations – Cont.

- UA Effect on Data Structure Choice
  - Efficient data structures with inherently high internal UA are not necessarily unsuitable for use with SCM. However, the level of acceptable internal UA is *inversely proportional* to the expected rate of updates the data structure will undergo.

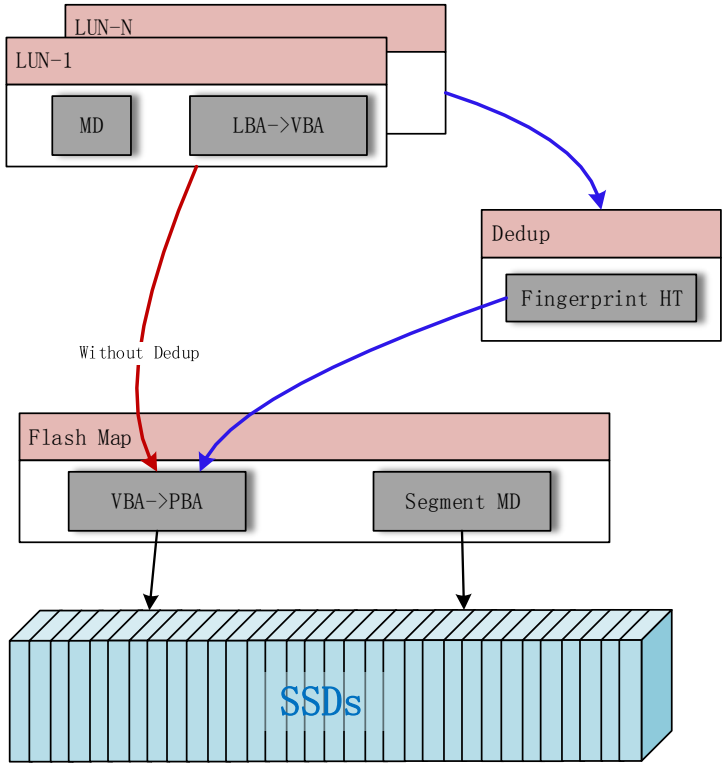
- Curtailing the SCM Accesses
  - Minimize the amount of cache lines mapped to SCM that are accessed per op by modifying the data structure designs, to reduce UA.

- DRAM / SCM Reads vs SCM Updates
  - Consider trading extra CPU cache and DRAM reads or writes, and even SCM reads for SCM updates.

Specific Examples of Microbenchmark Results

Metadata Management

- Design SCM-based data structures for AFA/SDS storage system under realistic storage workload scenarios:
  - Deduplication fingerprint indexing
  - Scalable block mapping mechanisms, ref-counting, GC metadata tracking, et
  - SCM write caching/aggregation for WA reduction



Address Mapping of SSD Pool

Disclaimers

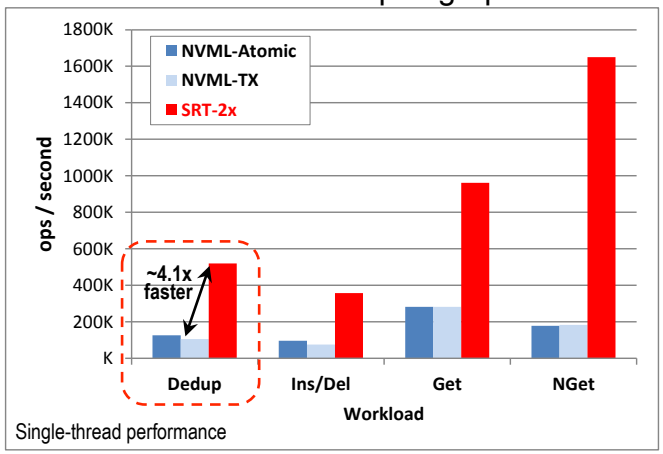
- Please keep in mind:
  - All the data structures mentioned in this section are *tunable* designs. Their performance can vary significantly based on the stored data properties (e.g. K/V sizes), workloads (R:W ratio, specific nature of updates), and characteristics of the SCM (emulated, in this case)
  - The results provided here are just *specific examples* of performance, relevant for a specific tuning of a specific data structure measured using a specific test harness. Changing any of those will affect the results, often quite dramatically
  - Since the results presented are from microbenchmarks, the *absolute* numbers are of little value – they are largely dictated by the memory subsystem of the CPUs used, number and clock frequency of the memory channels and DIMMs used, and SCM emulation target settings. But the *order of magnitude* of the results and their relation to one another is indicative of the SCM potential
  - Most of these results were obtained at a specific point in time during the development process of the individual data structures. Most of the data structures were subsequently re-tuned for different use cases inside SSTD, often providing significantly *higher* results
  - Unless otherwise specified, the results are for a single CPU core

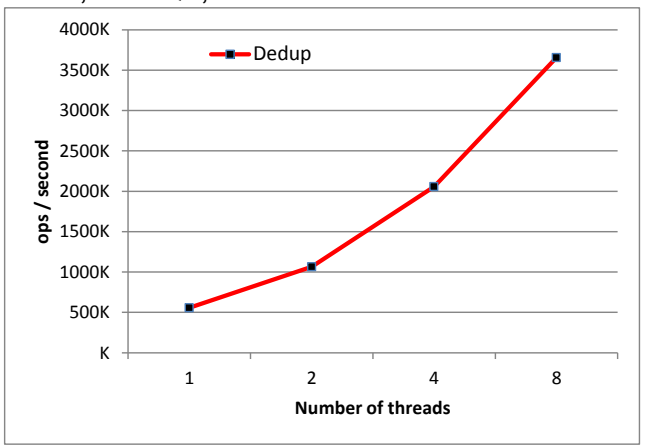
Persistent Hash Table

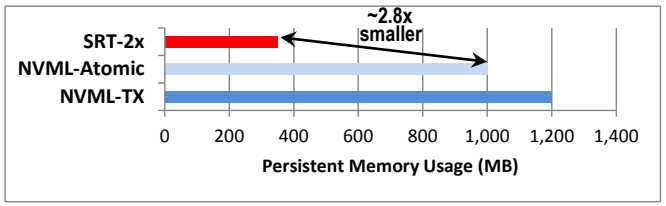
- Our first design was a family of persistent hash tables (HTs)
  - Non-resizable, fixed K/V size HT, aimed primarily at various indexing tasks, and in particular AFA dedup fingerprint indexing. The design is tuned to maintain both high insert and high lookup performance at high load factors (90-95%) and O(1) recovery.
  - The actual solution is a dual HT design, with both tables based on bucketized Cuckoo Hashing
    - Primary HT is typically tuned to occupy ~97-99.5% of the total capacity, with its max relocation path length set to ~1, to ensure high average update performance.
    - Secondary HT occupies the rest of the total capacity, with its max relocation path length is set to a fairly high value to delay premature table “overflow” at the expense of slightly reduced update rate
    - Inserts are always directed first to the primary table, only falling back to the secondary if the primary “overflows”. Lookups can be done in parallel on both tables, or sequentially with prefetching – depending on the workload.
    - The balance between the two table capacities, bucket sizes of the tables and the max relocation path of the primary HT are the main tunable parameters that can – and should! – be adjusted to match the expected workload characteristics (R/W skew).

Persistent Hash Table – Cont.

- Evaluation with dedup workload
  - Achieved much better performance and space utilization at high load factors than the best of Intel NVML persistent hash tables – even *before* any major optimizations
  - Additional use cases: dedup fingerprint indexes, KVSes, NoSQL, caches







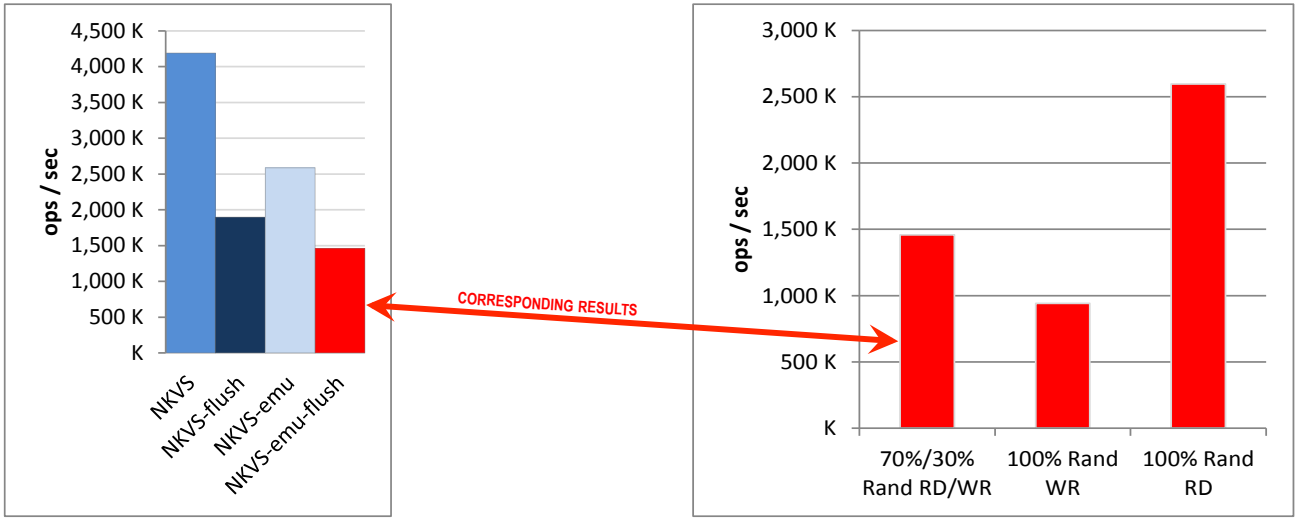
- All HTs measured at **90% load factor**, while emulating SCM with projected performance characteristics of Intel AEP (**not** DRAM at full speed!)
- **SRT-2x**: SCM-based persistent HT designed to maintain high performance at load factors of 90-95%+, for these tests configured for dedup-like app
- *"**Dedup**"*: mix of Ins/Del/Get/NGet ops representative of AFA deduplication use case, *"Ins/Del"*: 50%/50%, *"Get"/ "NGet"*: 100% hit/miss lookups
- All benchmarks done on large datasets to eliminate caching effects
- Different benchmarks used to measure different performance aspects, but using similar workloads. `pmembench` used as the test harness on the left 2 graphs. Results above are averages of multiple runs. All results are preliminary, measured early on, before significant performance tuning
- Intel Xeon E5- 2680 v4 CPU used, CPU was not the bottleneck above

NKVS: Persistent Linear K/V Store

- NKVS is a simple but powerful linear mapping data structure
  - It is a “nameless” K/V store, i.e. inserts do not specify a key, only the value to be inserted. The key is “returned” to the caller/invoker upon successful insertion, and can subsequently be used for lookups/updates
    - This obviates the need for an external “free slot” lookup mechanism (e.g. O(n) bitmap or O($\log_2$(n)) B$^+$Tree that are often used in storage systems) on the caller’s side
    - Keys are unsigned integers of appropriate width (proportional to the number of elems)
    - This solution is uniquely suited to SCM-based environments where the caller’s own data structure can be *cheaply* updated post-factum with the actual key allocated. In traditional block-based persistence environments this update would trigger potentially an avalanche of updates, but in SCM it can be done trivially due to its byte-addressability
  - It has array-like direct O(1) lookup and update complexity and O(1) recovery
  - The allocator used is an internal dual-mode arena/free-list design that incurs small O(1) overheads for all the operations, including system initialization
  - Vector interfaces allow amortizing crash-consistency overheads over a large number of operations
  - Use cases: block mapping, segment tracking, storage GC, ref-counting, etc.

NKVS – Cont.

- Evaluation with block-mapping workload
  - The chart on the right reflects the performance of NKVS in its full crash-consistent configuration *with* SCM emulation on different workloads.
  - The chart on the left demonstrates the effect on performance of `CLFLUSH`-es used for the crash-consistency mechanism and of SCM emulation.



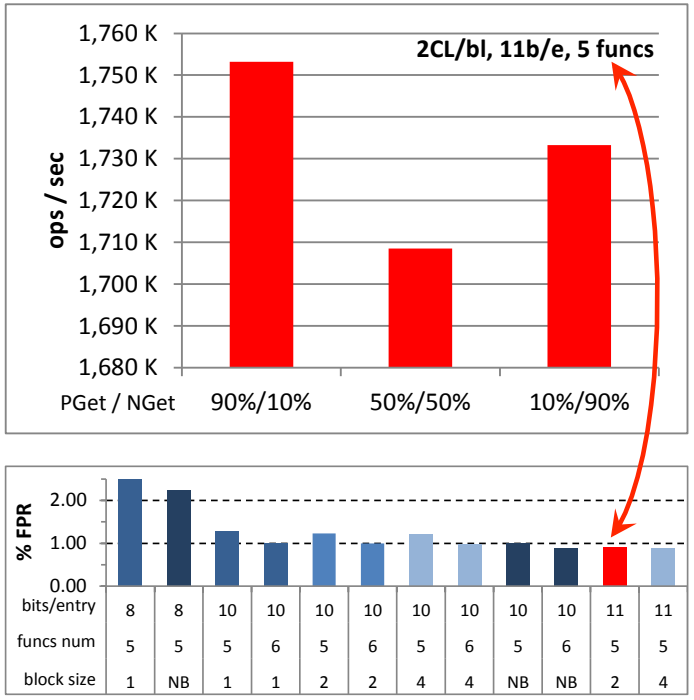
- Measured at 90% load factor, with (suffix **"-emu"** in the left chart, all of the right chart above) or without SCM emulation
- *"**70%/30% Rand RD/WR**"* storage workload: a mix of Get/Ins/Del/Upd ops representative of SDS/AFA block mapping with GC use case. This translates to about **3:2** lookups:updates ratio at the NKVS level
- All benchmarks were run on Intel Xeon E5- 2680 v4 CPU, on large datasets to limit caching effects. `pmembench` used as the test harness. Results displayed are averages of multiple runs.

Persistent Bloom Filter

- Another good example of effects of different SCM-specific tunings and optimizations on a data structure is a Bloom Filter (BF)
  - Classic/naive implementation of BF has very poor cache locality. The impact of this is amplified by the increased latency of SCM
  - By changing the design of BF from basic to blocked and applying some basic optimizations, it is possible to significantly improve the overall performance without degrading the false positives rate (FPR)
    - The optimal design of choice for the given target parameters turned out to be 2 CLs per block, where the 1st of the hash functions selects the block while the rest of the functions turn on the corresponding bits as in the regular design. This significantly improves cache locality
    - Once the cache locality issue is fixed, it becomes possible to apply additional optimizations, like prefetching, which further improves performance
  - Use cases: I/O elision for dedup SSD/disk lookups, slow/complex data structure lookup elision, cross-node communication elision, etc.

Persistent Bloom Filter – Cont.

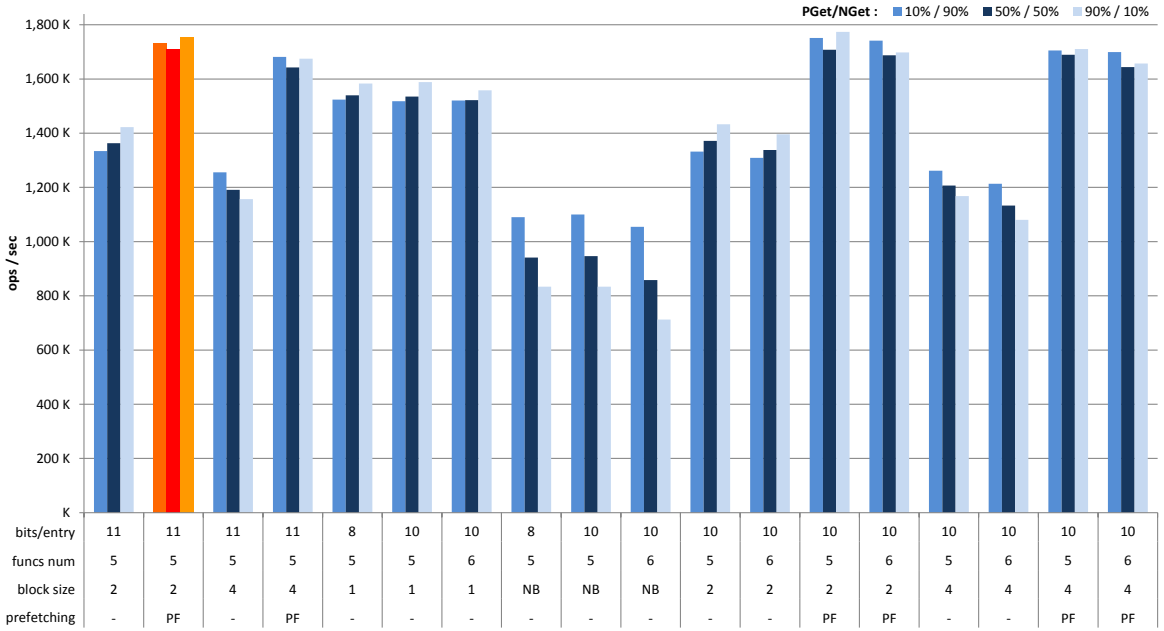
- Evaluation with different workloads
  - The design of choice was 2 CLs per block, 11 bits/entry using 5 hash functions, which gives FPR of 0.9% – though these were derived from an arbitrary goal
  - Due to the nature of BFs, the variance between positive and negative lookups is very minor
  - As expected, other target BF parameters resulted in significantly varying performance, but the 2 CL blocked design remains the leader
  - Impact of the different design choices on performance can be seen on the next slide



- "NB": non-blocked BF, otherwise block sizes specified in CLs
- Mixes of PGet/NGet ops representative of dedup segment content lookups for high/low dedup ratio workloads
- Measured while emulating SCM with projected performance characteristics of Intel AEP (**not** DRAM at full speed!)
- All benchmarks done on large datasets (120M) to eliminate caching effects
- Intel Xeon E5- 2680 v4 CPU used, CPU was not the bottleneck above
- Results above are averages of multiple runs. All results are preliminary

Persistent Bloom Filter – Cont.

- Impact of BF design and implementation optimizations on performance
  - FPRs are as per previous slide. Configuration of choice - highlighted



bits/entry	11	11	11	11	8	10	10	8	10	10	10	10	10	10	10	10	10	10	10
funcs num	5	5	5	5	5	5	6	5	5	6	5	6	5	6	5	6	5	6	
block size	2	2	4	4	1	1	1	NB	NB	NB	2	2	2	2	4	4	4	4	
prefetching	-	PF	-	PF	-	-	-	-	-	-	-	-	PF	PF	-	-	PF	PF	

Implementation Techniques

Custom-made Persistence

- Consider using custom-made, optimized persistence implementations matched to the corresponding data structures instead of relying on a single general-purpose library
  - Libraries like NVML, NVM-Direct, Mnemosyne, etc. have the appeal of using a uniform approach to persistence that allegedly can be (or was, by a 3rd party) implemented and debugged only once. However, in practice, for the foreseeable future their usage will probably not yield the expected hassle-free benefits, at least not until a lot of hands-on SCM-based data structure design and implementation experience is gained by the teams.
  - Leveraging system-, workload- and data structure-specific semantics can lead to great improvements to performance by allowing a lot of runtime work to be eschewed. This is typically impossible to do with cookie-cutter generic implementation approaches

Data-dependent Implementations

- Consider creating “families” of data structures based on the common code base, but each tuned to different requirements (e.g. size of data being stored)
  - In many cases a data structure designed to store small values (that can be atomically modified) can be made to be significantly faster (e.g. 2x-4x faster) than the same design that needs to store moderately small values (e.g. 24-64B in size), just by using a different crash-consistency technique
  - If possible, structure the code so that the persistence strategy is independent of the data structure maintenance algorithms. Careful decomposition, macros and/or C++ templates can be of help in this regard
  - Since usage of general-purpose persistent allocators (as well as multiple internal indirections in general) has significant negative performance impact, crash-consistency requirement amplifies the need for smart data layout. E.g. fixed-size value data structure can often be made much faster than variable-sized one by cutting a level of indirection, without altering the other data structure algorithms

Picking a Crash-consistency Method

- Consider using atomic updates, µLogs, CoW and WAL for different data structures as necessary and where possible, in that order of preference
  - In simpler systems using just SCM as the only persistent media in the system, where the entire "client" request can be processed synchronously and sequentially, it might be possible – and desirable – to use a single crash-consistency solution (e.g. literally a single common WAL or CoW + WAL for all the data structures – though typically this will be a per-core mechanism)
  - However, in more complex systems with asynchronous and potentially high-latency stages of processing (e.g. involving network accesses or SSD/HDD-based storage), processing the entire "client" request as a single persistent transaction will often not be possible from latency/throughput standpoint. In these cases, using different data structure-specific crash-consistency techniques will lead to a higher-performance system than just trying to flood a single generic common WAL with many mini-transactions at different stages of the processing

Avoiding General-purpose Allocators

- If possible, try avoiding usage of general-purpose external persistent allocators in favor of intrusive data structure implementations and internal allocators
  - High-performance multi-threaded RAM allocators (`jemalloc`, `tcmalloc`, TBBmalloc, Hoard, etc.) achieve their performance at the expense of significant internal complexity. Trying to make all these smarts work in a fully crash-consistent manner in conjunction with the calling code will (and does, see e.g. NVML) result in a fairly slow application level performance on SCM
  - Simpler general-purpose SCM-friendly allocators that don't require quite as many modifications to their persistent state are not currently available, or result in degraded performance
  - For the foreseeable future, sticking with static pre-allocation, per-data structure free-lists, internal arena-based allocators, etc. appears to be a good way to achieve high performance. While this may come at the expense of dynamicity of SCM capacity allocation to different data structures / caches, given the projected total SCM capacity this seems to be a good trade-off.

Locking vs Persistency

- When possible, avoid mixing SCM accesses (especially persistence) with usage of blocking locks (e.g. mutexes / spinlocks)
  - Many lock-heavy algorithms implicitly rely on the fact that *storing* a value into memory as part of a critical section is an almost “free” operation (a CPU register update followed by an asynchronous – as far as program flow is concerned – writeback to L1 cache, around 4 cycles if lucky)
  - However, performing a crash-consistently persisted store into SCM could involve 2-8 very expensive and serializing (in the memory model sense) CL flush/writeback operations, taking potentially 10x-1000x longer than L1 store
  - Therefore, holding a spinlock while crash-consistently updating several discrete SCM-backed memory locations may turn out to be much slower than expected, system wide. Even SCM loads should be minimized *if* they will usually result in cache misses (q.v. prefetching)
  - Lock-free designs, especially those avoiding *mutable* shared state altogether will yield superior throughput with SCM, and in many cases lower latency too
  - Where applicable, usage of oplocking should be evaluated

SCM Pointers vs Indexes

- At present, in common commercial/FOSS general-purpose OSes, there is no 100% reliable way of ensuring that the various regions of SCM will get mapped to the same addresses across system restarts
  - On the one hand, the common push in OS development now is for address space randomization to improve the security of the applications. On the other hand, once SCM becomes widely available, presumably a combination of libraries and some modest tweaks to OS syscalls can be introduced to work around this, either upstream or in project-specific fashion
- One currently popular workaround is to use a "*base + offset"* approach instead of storing actual process address-space pointers to SCM
  - The "base" is typically a pointer to the start of the data structure. While it can change across reboots, it is typically easy to discover it as part of the recovery phase. It can then be stored just once
  - The "offset" is usually an offset within the data structure or region of SCM. While it alone is not enough to access the data, it is guaranteed to be correct across system restarts. Typically, it also occupies less space than a pointer
  - To dereference a pointer into SCM, one combines "base" + "offset" on demand

Thank you

Questions?