# Writes Hurt: Lessons in Cache Design for Optane NVRAM

Alexandra Fedorova
MongoDB and University of British Columbia
sasha.fedorova@mongodb.com

Keith A. Smith
MongoDB
keith.smith@mongodb.com

Keith Bostic
MongoDB
keith.bostic@mongodb.com

Susan LoVerso
MongoDB
Sue.LoVerso@mongodb.com

Michael Cahill
MongoDB
Michael.Cahill@mongodb.com

Alex Gorrod
MongoDB
alexander.gorrod@mongodb.com

## ABSTRACT

Intel® Optane™ DC Persistent Memory resides on the memory bus and approaches DRAM in access latency. One avenue for its adoption is to employ it in place of persistent storage; another is to use it as a cheaper and denser extension of DRAM. In pursuit of the latter goal, we present the design of a volatile Optane NVRAM cache as a component in a storage engine underlying MongoDB. The primary innovation in our design is a new cache admission policy. We discover that on Optane NVRAM, known for its limited write throughput, the presence of writes disproportionately affects the throughput of reads, much more so than on DRAM. Therefore, an admission policy that indiscriminately admits new data (and thus generates writes), severely limits the rate of data retrieval and results in exceedingly poor performance for the cache overall. We design an admission policy that balances the rate of admission with the rate of lookups using dynamically observed characteristics of the workload. Our implementation outperforms OpenCAS (an off-the-shelf Optane-based block cache) in all cases, and Intel Memory Mode in cases where the database size exceeds the available NVRAM. Our cache is decoupled from the rest of the storage engine and uses generic metrics to guide its admission policy; this design can be easily adopted in other systems.

**ACM Reference Format:**
Alexandra Fedorova, Keith A. Smith, Keith Bostic, Susan LoVerso, Michael Cahill, and Alex Gorrod. 2022. Writes Hurt: Lessons in Cache Design for Optane NVRAM. In *Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3542929.3563461

## 1 INTRODUCTION

Intel® Optane™ DC Persistent Memory is one of the first widely available non-volatile memory (NVRAM) products, released in 2019. At present the community is still grappling with the question of how to best use it in the storage stack. Although one way of adoption exploits its persistence (e.g., using it in place of another block storage device or turning applications' volatile memory into persistent), another avenue is to use it as a volatile extension to DRAM, a denser and cheaper one at that. Our study explores the second option.

We design and implement *NVCache*: an Optane NVRAM-resident volatile cache for WiredTiger [11] – the storage engine underlying MongoDB [10]. At the heart of any cache is an *admission policy*. An admission policy decides, upon a cache miss, whether the missing block should be *admitted*, i.e., kept in the cache after being retrieved from a lower level of storage. With few exceptions, caches indiscriminately admit data on read misses, differing only in whether they admit it on writes. We found that such a simplistic policy limits the throughput of write-heavy workloads to only 20% of the best achievable, that of read-only workloads to about 80%. Admitting new data into a cache generates writes: every newly inserted cache block must be written into the cache memory. Limited write throughput is a well known property of Optane NVRAM [39]. What was *not* previously known was that writes to Optane NVRAM disproportionately affect the throughput of simultaneously occurring *reads*. Although writes affect simultaneously occurring reads on any storage device, the effect is much larger on Optane NVRAM than on its counterpart DRAM (see §2). Overly eager cache admission

*SoCC '22, November 7–11, 2022, San Francisco, CA, USA*

ACM ISBN 978-1-4503-9414-7/22/11.
https://doi.org/10.1145/3542929.3563461

will thus limit the rate at which existing data can be retrieved, diminishing the utility of the cache. We confirm this claim experimentally (Table 1 in §3.2.3). ***Admission policy must, therefore, balance between the rate of admitting new data and the rate of accessing existing data***. Our main contribution is a new admission policy that embodies this principle.

Although our work is a case study exploring a specific point in a vast design space, our findings apply broadly to similar systems. NVCache is decoupled from the rest of the storage engine and our new admission policy relies only on the rates of data admission, removal and lookup for its decisions, so our design is easy to adopt in other storage engines or stand-alone caches. While our work addresses the idiosyncrasy of one specific storage technology, the lessons we learn apply for any caching device where writes disproportionately impact reads.

The rest of the paper is organized as follows: §2 demonstrates that writes disproportionately affect the throughput of reads on Optane NVRAM. That section also puts our work in the broader context of multi-tier caching systems, and provides relevant background on WiredTiger. §3 presents the basics of NVCache design, which relies on well-known methods, and then unveils the design of the new admission policy, backing its features with experimental data. §4 compares NVCache with off-the-shelf alternatives: Intel Memory Mode [3] and OpenCAS [5], and reports the effect on performance-per-$ of replacing part of system DRAM with Optane NVRAM. §5 describes related work and §6 summarizes our findings.

# 2 BACKGROUND AND MOTIVATION

## 2.1 Optane memory's Achilles' heel

Optane NVRAM has a superpower: read and write latency for small operations is competitive with DRAM, reads being only about 2× slower and writes being roughly the same latency as DRAM[1] (see [39], Fig.2). Read throughput is impressive: sequential reads reach 6GB/s per NVDIMM (see [39], Fig.4(a)), and with a single CPU supporting up to six NVDIMMs, the throughput can climb into double digits.

Optane also has an Achilles' heel: write throughput is sluggish and struggles with concurrency. Figure 1 shows sequential write throughput to Optane NVDIMMs (with one and two DIMMs) and to an Optane SSD P4800X (built with the same memory technology but packaged as an SSD). Writes to Optane memory are barely competitive with the SSD using one thread, but show negative scaling as we use more threads[2].

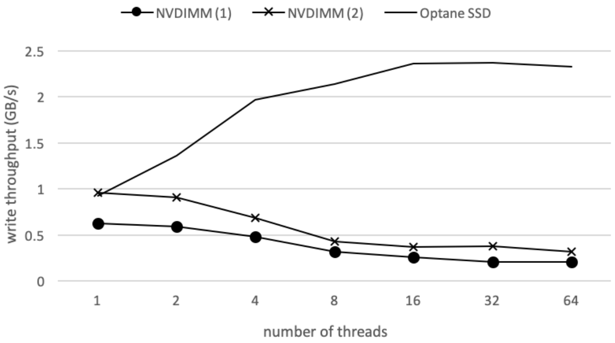


**Figure 1: Sequential write throughput to Optane persistent memory using one or two NVDIMMs, and to Optane SSD. Parameters of the experimental system are described in §3.2.1.**

Poor scalability of writes was reported before (see [39], Fig.4(b)). What was less known is that the presence of writes disproportionately affects the throughput of *reads*. Figure 2 shows the read throughput on Optane NVRAM dropping precipitously in the presence of concurrent writers. Only a single concurrent writer causes read throughput to drop from a solid 12GB/s to a unimpressive 3.4 GB/s (a 72% loss). With eight writer threads, reads proceed at only 0.8 GB/s (a 93% slowdown)[3]. The same experiment on DRAM produces a milder degradation in read throughput, with a loss of only 18% with one concurrent reader and of 35% with eight.

The implication of this finding for cache design on Optane NVRAM is that an admission policy that eagerly accepts new data (and thus generates writes) will disproportionately affect the speed of reads, i.e., cache lookups, severely limiting the effectiveness of the entire system. ***An admission policy, must therefore carefully balance the rate of cache admission relative to the rate of lookups***.

## 2.2 Multi-tier caching systems

We contribute a new design of a single-tier volatile cache in Optane NVRAM; since this cache co-exists with the DRAM

[1] Writes into NVRAM need only to reach the processor's ADR (Asynchronous DRAM refresh domain).

[2] Our data is for non-interleaved writes. Interleaved writes will achieve higher throughput (and also negative scaling with more threads, see [39], Fig.4(c)), but interleaving can only be used on NVDIMMs in the same NUMA node, which was not the case on our system configured according to manufacturer recommendations ([4], Table 17). NVRAM access was done via `memcpy` from a `mmaped` file residing on a DAX file system in NVRAM. This was the fastest method and it produced similar results as the fastest methods discovered by others [39].

[3] Our system has 16 cores, so CPU contention is not the issue.

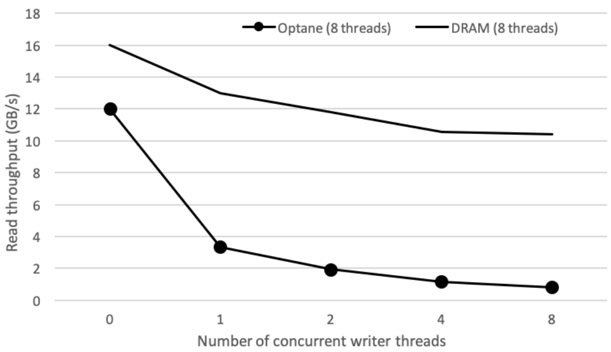


**Figure 2: Read throughput for Optane NVRAM (two NVDIMMs) and DRAM in with 8 reader threads and with increasing concurrent writers. Parameters of the experimental system are described in §3.2.1. This is a NUMA configuration; we observed similar results on a non-NUMA system. We made sure that file system or address space contention are not present.**

cache in our storage engine (see §3), it is helpful to discuss it in the broader context of multi-tier caches and tiered memory systems. Here we provide a broad overview of these areas, deferring the comparison with specific projects until §5.

A multi-tier caching system is comprised of multiple storage devices organized as a hierarchy or a pool of caches [14, 16–20, 25, 26, 28, 30, 31, 36, 40, 41]. Tiers might include DRAM and NVRAM in front of an SSD (as in our system), a SSD in front of an HDD, or any other combination thereof, but with faster, more expensive storage generally in front of slower, less expensive type. Studies of these systems investigate how to divide the data between the tiers to maximize performance. Broadly speaking, there are two design approaches: *cooperative* and *independent*. In a cooperative design the tiers are tightly coupled: one tier may evict data into another, and may inform it about the access patterns observed within its space. In an independent design each tier makes its own decisions about what data to admit and evict. There is also a middle ground, where one tier may take hints about data access characteristics from other tiers, but does not directly accept data or directives about what to cache. Independent caches are easier to design and maintain from software engineering perspective, because they are less coupled with the rest of the system, and for this reason they are easier to port to other systems. Our design falls into the independent category, as we explain in §3.

Multi-tier memory systems can be thought of as a subcategory of multi-tier caches, where one tier is DRAM and another is NVRAM or some other kind of slower memory [12, 13, 22, 27, 32, 35, 37**?** , 38]. These systems are typically implemented in the kernel or in a language runtime [13, 35**?** ] and are transparent to applications. The main challenge in building them is deciding which pages must go to the "fast" tier and which ones to the "slow" one – the same problem that must be addressed in cooperative caches.

Like all caches, multi-tier systems innovate on admission and eviction policies. An admission policy tells the cache when to insert new data; an eviction policy tells it which data to evict when the space becomes scarce. Typical caches always admit data on reads and vary as to whether they admit data on writes: i.e., *write-allocate* or not. Multi-tier caches may also admit data as it is evicted from another tier. While most caches tune their admission algorithms to maximize the hit rate, our algorithm takes into account the *rate of admission* for reasons explained in §2.1. So our main contribution is the admission policy that is based on a new principle. This principle will be relevant for cache storage media where the presence of writes disproportionately affects the throughput of reads.

## 2.3 WiredTiger

WiredTiger is a persistent transactional key-value store [11]. Internally it uses a B+-tree to organize the data. WiredTiger materializes data in memory (in its DRAM cache) in a different format than it is stored on disk. Data on disk contains efficiently encoded keys and values. The keys in each block are sorted, but not indexed. When WiredTiger reads a block from disk it decodes and indexes it, so that the data can be searched and updated efficiently. Furthermore, on-disk data may be optionally compressed and/or encrypted, and WiredTiger decompresses and decrypts it before placing it in DRAM.

The main advantages of this two-pronged approach to data representation is that it provides efficient space utilization for stored data and fast operations for cached data. It is also the reason we adopted the independent design for our NVRAM cache, as we explain in §3.1.

# 3 NVCACHE: A STEP-BY-STEP DESIGN

We first describe the baseline architecture of *NVCache*, which builds upon well-known techniques. Then we describe the evolution of the new admission policy design, beginning with a naïve architecture and presenting experiments that motivate the next feature.

## 3.1 NVCache basics

As explained in §2.3 WiredTiger uses different formats for data stored persistently on disk and for data materialized in memory. On-disk data is stored in *blocks*. In-memory data, which lives inside the engine's fixed-sized DRAM cache, is stored in *pages*. Blocks contain efficiently encoded keys and values. Pages additionally contain indexing and other structures to facilitate fast operations.

NVCache sits underneath the DRAM cache. Naturally we had to make a decision whether to use NVCache for caching pages, blocks or both. WiredTiger already has a DRAM cache for pages, so caching pages would amount to extending the existing cache to use both DRAM and NVRAM – a tiered cache similar to the recent one in Meta's RocksDB [28]. Caching blocks would entail creating a stand-alone block cache that sits between the DRAM cache and the block device. We decided to cache blocks, and not pages, for the following reasons.

WiredTiger's pages are organized in memory as a B+-tree for efficient searching and updating, and pages contain pointers to other pages. If a page were to be manually copied (at application level) from DRAM to NVRAM in a tiered cache, the virtual addresses would change and any pointers would have to be updated accordingly. Updating them is an error-prone process that would require locking or other form of synchronization. WiredTiger is lock-free on the read-path and mostly lock-free on the write path: adding synchronization would substantially compromise a core advantage of its original design.

An alternative to implementing a tiered cache manually would be to use transparent tiered memory implemented in the kernel, such as Nimble [38] or HeMem [32], or to build on top of CacheLib: Meta's library for building caches that provides support for tiered memory [15]. Kernel-based systems would require adopting an experimental kernel, which was not an option in a production deployment. CacheLib source became open on September 2, 2021 [1]; building upon it is one alternative we may consider in the future, but according to the authors, CacheLib is not the best option for building a database's internal page cache, and so it could not be used as the substrate for RocksDB's page cache (see [15], Section 6 and discussion in §5). Thus, for our current design we decided to use a stand-alone block cache, as it avoids the aforementioned problems, is simple to integrate in the existing storage engine and can be easily ported to other key-value stores. We did however compare with a configuration where the engine's page cache transparently expands into NVRAM configured to use Intel Memory Mode. This configuration (see §4.3) could be thought of as hardware-based tiered memory.

NVCache sits next to the *block manager* – the code responsible for reading/writing the data from/to disk (see Fig. 3). **Read path:** If the DRAM cache cannot locate searched-for data, it issues a read to the block manager ①. The block manager checks if the block is present in NVCache ②, accessing it from NVCache if it is ③ and reading it from disk if it is not ④. It then transforms the block into a page, decrypting and decompressing it if needed, and hands it over to the DRAM cache ⑤. If the block is not present in NVCache, NVCache has the discretion to admit it after the block manager has read it from disk ⑥. NVCache stores the blocks in the same format as they are stored on disk: compressed/encrypted if those configuration options were chosen. Storing compressed blocks increases NVRAM effective capacity.

**Write path:** The write path is not symmetrical to the read path, because WiredTiger does not modify disk blocks in place. Updates are written into in-memory specific data structures, and then formatted into blocks and written back to disk during a process called *reconciliation.* Reconciliation may occur when the DRAM cache evicts pages or as part of a database checkpoint. Reconciliation always writes a new page ⑦, which the block manager turns into a new block. When the block manager writes a new block ⑧, it notifies NVCache ⑨; NVCache has the discretion to admit it. Obsolete blocks are eventually freed, at which time the block manager instructs NVCache to invalidate cached copies of the freed blocks ⑩. The block manager always consults the cache on reads and writes, so it does not need to do extra work to keep the data consistent among NVCache and the storage.

Within a broader context of multi-tier caching systems, NVCache adopts an independent design (see §2). This is a natural consequence of our decision to cache blocks, as opposed to pages. The kernel buffer cache also caches blocks, so there is an opportunity for a cooperative design integrated with the kernel: we did not pursue this avenue, because adopting a custom kernel would not be practical in customer deployments. There are off-the-shelf NVRAM caching solutions implemented in the kernel: *device mapper write cache* [2] and OpenCAS [5]. We describe them, evaluate OpenCAS (the more advanced of the two) and present the results in §4.

We experimented in the middle ground between an independent and a co-operative design, where the DRAM cache informs the NVCache on evicting a clean page (so the NVCache could bump its priority) or informs NVCache about the reason for writing a dirty block (e.g., because of eviction or a checkpoint). Using this information did not improve NVCache effectiveness, and keeping track of it introduced overhead, so we retained a purely independent design. As a result, NVCache communicates with the block manager via a narrow API, allowing its codebase to evolve independently of the rest of the system.

Internally, NVCache is organized as a hash table with a fixed number of buckets. Upon collision, blocks mapping to the same bucket are chained in a linked list. A bucket is protected with a spinlock, but our measurements showed that the rate of collisions and the synchronization overhead were negligible (with 32K buckets for a 252GB NVCache).

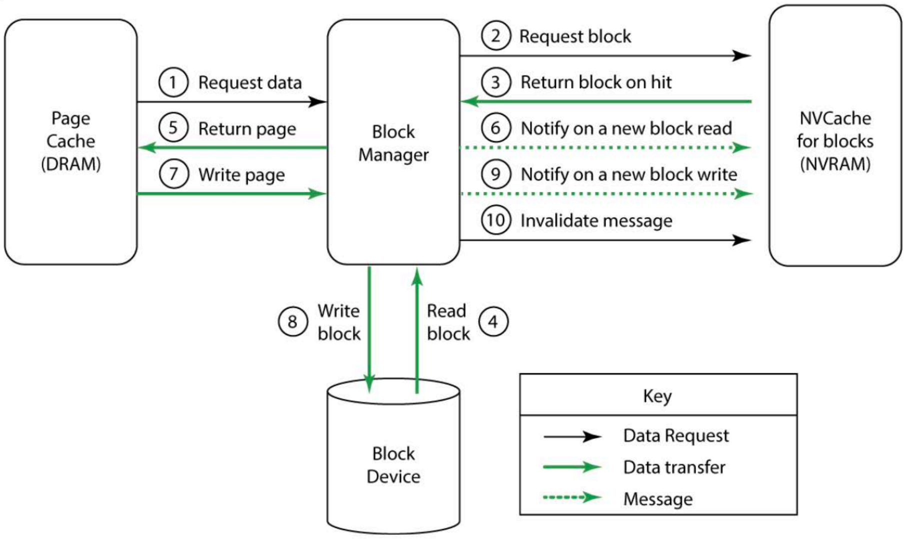


**Figure 3: Interaction of NVCache with the rest of the storage engine.**

We use PMDK's [7] allocator (based on jemalloc) to allocate NVRAM on admitting new blocks. NVCache metadata is in DRAM, but PMDK's jemalloc metadata is in the NVRAM. NVCache does not use NVRAM's persistent nature: upon exit it loses cached data. This decision simplified our design substantially, as we do not need to deal with crash consistency. The downside is that we pay the cost of re-warming the cache upon restart, and so we may revise our design in the future.

When NVCache runs out of space it cannot admit new blocks. *Eviction* is needed to purge blocks less likely to be used in order to make space for new ones. We use a simple LFRU eviction policy [29]. During eviction it targets blocks that were not reused within a fixed time window and evicts the least frequently used among those. We approximate tracking of the LRU and LFU information similarly to how the clock algorithm does it [9], so there is no need to maintain separate lists: the "clock hand" simply iterates over the buckets and lists in the hash table. There is an eviction thread that wakes up once a second and scans the cache for eviction candidates.

## 3.2 NVCache Admission Policy Design

The NVCache admission policy is rooted in experimental data; we therefore present the details of our experimental system and the workloads prior to exploring its design.

*3.2.1 Experimental system.* Our system is a Lenovo ThinkSystem SR360 built with two Intel Xeon Gold 5218 processors, each having 16 hyper-threaded cores.

**Memory:** There are two Optane NVRAM modules, 126GB each, for a total capacity of 252GB. The modules are placed in separate sockets as per manufacturer recommendation. There is 196GB of DRAM; we modulate the amount available for experiments either via software (by creating a large file in ramfs) or hardware (by physically removing DRAM) in cases where the experiments demand this. We used workloads with a variety of database sizes to study conditions when the working set fits into NVRAM and when it exceeds its capacity.

**Disk:** We use Intel Optane P4800X SSD, built with the same physical media as NVRAM DIMMs, but packaged as an SSD on the PCIe bus. This SSD provides up to 2.5GB/s sequential read bandwidth and up to 2.2GB/s sequential write bandwidth.

*3.2.2 Workloads.* While for the final evaluation (§4) we used the widely adopted YCSB [8, 21], during the *design* process we used our in-house benchmarks. The in-house benchmarks are configuration files for a WiredTiger-provided workload generator application, specifying parameters such as the number of records in the database, the sizes and distributions of keys and values, the mix of operations (read, update, insert, modify, scan), the number of threads, whether or not logging and transactions are enabled, the size of the DRAM cache, the total running time, etc. The benchmarks are designed to either emulate customer workloads or to stress a particular feature (e.g., checkpoints, eviction). When presenting the throughput for a benchmark we break it down by operation type: for example, if the benchmark *bm* performs a mix of reads, inserts and updates, we report the throughput as *bm.READ*, *bm.INS* and *bm.UPD*.

The workloads fall into two categories: **(1)** those that do not stand to benefit from NVCache (e.g., they use small data sets fitting entirely in DRAM, and/or they perform mostly writes) and **(2)** those that do (large data sets, read-dominant). We initially focus on benchmarks in the first category, many of which have small data sets. The database pages are cached in the engine's DRAM cache, and its blocks are cached in the kernel buffer cache[4]. So OS buffer cache would comfortably fit blocks of small workloads. Since NVRAM caching cannot benefit these workloads, they make for an easy demonstration of the implementation overhead and are excellent workloads for exploring how to minimize it.

*3.2.3 Lessons learned.* Our design rests on the three lessons that we learned in the process: (1) Bypass NVRAM for small workloads, (2) Throttle the admission rate, and (3) NVRAM cache benefit is limited to read-dominant workloads. Lesson #2 embodies our main contribution; the others, while not novel, were also crucial for building a well-performing cache.

*Lesson #1: Bypass NVRAM for small datasets.* Our simplest admission policy, *alloc-read-write*, was always admitting a block to the NVCache when it is read from or written to disk by the block manager. Figure 4 shows the performance *degradation* of running with 16GB DRAM and 252GB NVCache[5] for the first category of benchmarks that will not benefit from any additional caching. We disable eviction in these experiments to tease apart the sources of overhead; because we use benchmarks that don't benefit from caching the eviction policy is irrelevant to their performance. We enable eviction for the experiments, where caching matters (at the end of this section). We observe that performance penalty under this policy is substantial across the board, reaching 91% slowdown for *evict-btr-str-m*.

The trivial reason for overhead is that it is not useful to cache data for small workloads that fit into either the engine's cache or the OS buffer cache, both of which are in DRAM. So our first lesson is to ***bypass NVCache for datasets fitting into DRAM***. We call this feature *small-bypass*, and implement it by having the NVCache monitor the aggregate size of all database files used by the workload and abstain from admitting any blocks until the dataset size outgrows the available DRAM. The bar labelled *small-bypass* in Fig. 4 shows the overhead being significantly reduced by this feature.

*Small-bypass*, in a way, approximates cooperation with the OS buffer cache. NVCache cannot know which blocks the buffer cache holds, but it roughly approximates this information by juxtaposing the workload's data size and the amount of DRAM.

*Lesson #2: Throttle the admission rate.* The *small-bypass* feature all but eliminated the overhead for some workloads, but made only a small dent for others. To show why, Figure 5 presents the number of blocks removed from NVCache because they were outdated and freed by the block manager as a percent of all admitted blocks. We observe that the benchmarks whose overhead is still substantial after the introduction of *small-bypass* are those that overwrite many existing blocks.

When an application generates new data, either by inserting new key-value pairs or updating the old ones, the block manager generates new data blocks. The blocks containing old invalid data are eventually freed by the block manager and are removed from NVCache. Removing a block from NVCache frees its associated memory in NVRAM. Since the PMDK allocator keeps its metadata in NVRAM, freeing a block produces writes into NVRAM. Moreover, removing old blocks creates space for new blocks, and NVCache eagerly admits data in the freed space. That also generates writes. As we showed in §2 writes disproportionately affect the throughput of reads, i.e., of cache lookups.

One could simply disable the cache for write-intensive workloads, but even ***read-dominant workloads will suffer from the interference of writes if an overly eager eviction policy makes room for the admission of new blocks at a high rate***. Consider data in Table 1 for the three read-dominant workloads from Table 2 (this table contains workloads with large working sets, for which caching may be beneficial). Table 1 shows data for experiments with eviction configured to eagerly evict unused blocks and for experiments configured to run without any eviction at all. Even though the cache hit rate is higher with eager eviction (this is expected – eviction makes space for newer, frequently reused blocks), the throughput is substantially worse than without any eviction at all. That is because the number of cache writes produced with eviction is substantially greater than without it, and the writes slow down the reads.

The question we then ask is: ***how to balance the rate of block admission and removal, which generate writes, with the rate of cache lookups, which produce reads?*** To address it, we introduce the *overhead bypass* ratio (OBP):

$$OBP = \frac{blocks\_inserted+blocks\_removed}{blocks\_looked\_up}$$

Intuitively, the quantity in the numerator captures the cost of using the cache: the write-generating insertions and removals. The quantity in the denominator captures the benefit: cache lookups. OBP thus expresses the balance between the cost and benefit of using the cache; we experimentally determined that a target ratio of 10% works best for our hardware. If OBP were to be ported and tuned for different hardware, the thresholds would be adjusted according to the degree to which concurrent writes affect the reads.

[4] §3.1 explains the difference between blocks and pages.

[5] We ran with larger DRAM sizes too, but reached the same conclusions.

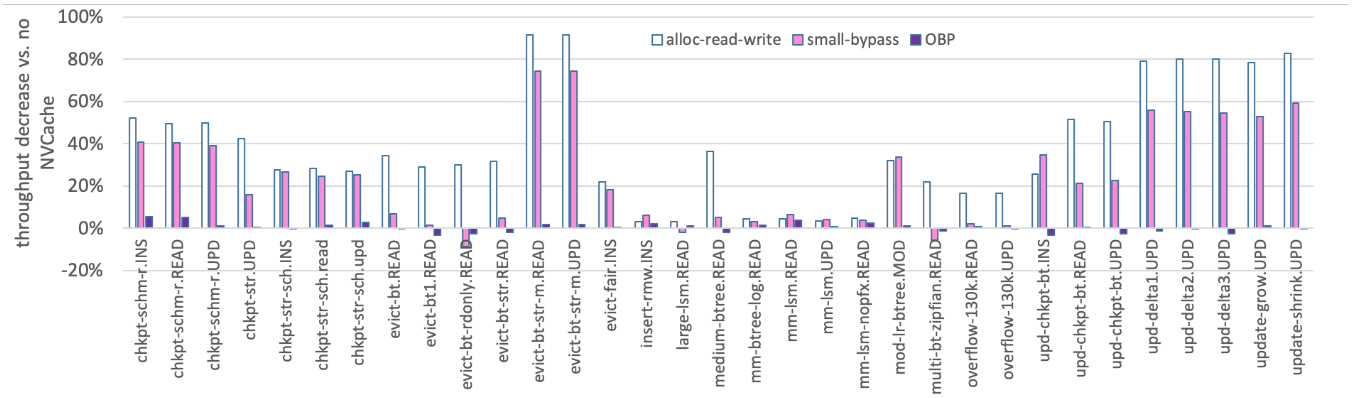


**Figure 4: Throughput degradation for workloads that do not stand to benefit from NVRAM caching. Lower numbers are better. Eviction is disabled during these experiments to simplify the analysis of the overhead.**

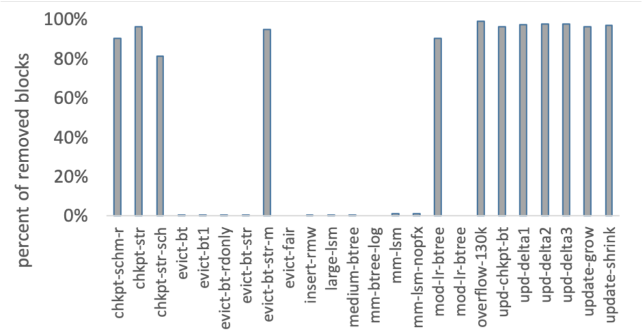


**Figure 5: Cached blocks that were outdated and freed. Data corresponds to the experiment in Fig. 4. These are aggregate data for the entire workload, so we do not show the breakdown by operation type.**

| | Eager eviction | | No eviction | |
|---|---|---|---|---|
| **WL** | **ops/sec** | **hit rate** | **ops/sec** | **hit rate** |
| *evict-btree-large* | 61,699 | **48%** | **162,690** | 44% |
| *evict-btree-scan.read* | 97,491 | **45%** | **134,404** | 36% |
| *medium-btree-large* | 62,012 | **48%** | **164,644** | 44% |

**Table 1: Throughput of read-dominant workloads suffers substantially with aggressive eviction despite it producing a higher cache hit rate. Aggressive eviction generates many writes that hurt the throughput of cache lookups (reads).**

E.g., on hardware where writes have a smaller effect on the performance of reads, acceptable OBP thresholds would be higher.

NVCache continuously updates OBP and abstains from admitting or evicting cache blocks if OBP exceeds its target (10%). The OBP metric proved remarkably stable across workloads and cache sizes. We also found OBP to work better than a simple no-write-allocate policy or OBP used in conjunction with the no-write-allocate policy. The *small-bypass+OBP* bar in Figure 4 shows that *small-bypass* and OBP completely eliminate the overhead for the benchmarks that do not stand to benefit from caching.

We did study the sensitivity of performance to the value of the OBP threshold and found that the values between 5% and 30% generated similar performance across all the workloads. Since OBP includes parameters that do capture the workload characteristics (the rate of insertions, removals and lookups will vary across workloads), OBP is workload-sensitive by design. So even though the OBP threshold is statically set, the resultant rate of cache admission will largely depend on the workload. E.g., we will admit very few blocks for write-intensive workloads, while filling up the cache for the read-intensive ones.

*Lesson #3: Only read-dominant workloads benefit.* To understand what workloads benefit from NVCache here we switch to workloads with large datasets that exceed the available DRAM, reconfiguring previously used 'small' workloads as necessary. They teach us the third lesson: NVRAM cache benefits only read-dominant workloads. A prior study of a custom NVRAM cache for Meta's RocksDB came to a similar conclusion [28].

Table 2 shows the large-sized workloads and their characteristics. The rate of operations marked with an asterisk (e.g.,

| Workload | Op mix (threads), data size | DRAM cache size | NVCache hit ratio | Removed / inserted ratio | Amount of data written to SSD | Amount of data admitted to cache |
|---|---|---|---|---|---|---|
| **500m-btree-50r50u** | 50% read, 50% update (20), 163GB | 28GB | 6% | 98% | 2190GB | 191GB |
| **chkpt-stress** | 100% update (6), 134GB | 28GB | 2% | 94% | 780GB | 36GB |
| **evict-bt-stress-multi** | 80% read, 20% update (100), 250GB | 1GB | 20% | 94% | 1740GB | 424GB |
| **evict-btree** | **100% read** (16), 120GB | 28GB | **97%** | 0% | 120GB | 115GB |
| **evict-btree-scan** | **95% read**, 4% insert*, 1% update* (430), 250GB | 28GB | **97%** | 47% | 400GB | 300GB |
| **medium-btree** | **100% read** (16), 120GB | 28GB | **97%** | 0% | 120GB | 115GB |
| **overflow-130k** | 50% read, 50% update (20), 253GB | 21GB | 6% | 95% | 2000GB | 127GB |
| **update-chkpt-btree** | 90% insert, 5% read, 5% update (5), 185GB | 25GB | 6% | 95% | 1720 GB | 137GB |
| **update-delta-mix1** | 100% updates (6), 125GB | 20GB | 2% | 98% | 2000GB | 93GB |
| **update-grow-stress** | 96% update, 4% inserts* (5), 190GB | 20GB | 2% | 97% | 2100GB | 98GB |

**Table 2: Properties of ‘large’ workloads.**

*insert*, *update* for *evict-btree-scan*) is kept constant by the workload generator, and so we do not report their throughput, because it is largely insensitive to the system configuration. The *data size* reported in the second column is the on-disk size of the database reported at the end of the run. The intermediate database size may be much larger at points when many new blocks were written to disk, but the outdated ones were not yet freed. Column six reports the total amount of data written to SSD during the run. This amount is non-zero even for read-only workloads, because it includes the data written to populate the database prior to the measured benchmark run. Although NVCache is enabled during the populate phase, it hardly admits any blocks, because OBP throttles the admission rate during this write-only phase. So when the measured run begins, NVCache is empty; it warms up during the measured run. All benchmarks run for 60 minutes, with the exception of *500m-btree-50r50u*, which runs for 120.

Figure 9 presents the throughput of large workloads with 32GB DRAM and 252GB of NVCache. (Data with other memory sizes leads to similar conclusions, so we omit it.)

Read-intensive workloads benefit from NVCache substantially, running over 3× faster with the cache than without it (e.g., *evict-btree-scan,READ*). But even a small proportion of writes substantially limits performance potential: *evict-bt-stress-m* performs 20% of update operations, but the performance boost it gets from NVCache is only 12%.

Write-intensive workloads do not benefit from NVCache, or from any other block caching, because they make most of the cached content obsolete (remember that WiredTiger does not overwrite blocks). Table 2 shows the NVCache hit ratio

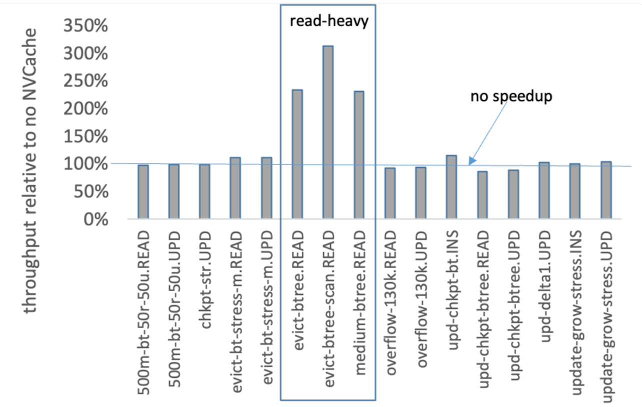


**Figure 6: Workloads with large datasets. 32GB DRAM and 252GB NVCache. Workload names are abbreviated.**

and the fraction of removed blocks relative to those inserted. The data tells us two things: (1) workloads that don't benefit from the cache have a very low hit ratio, (2) the low hit ratio is because they make obsolete most of the blocks they insert. They write terabytes of data throughout the run (Column 6), even though their database size at the end of the run is no larger than a couple hundred gigabytes (Column 2).

These data suggest that admitting zero blocks for write-dominant workloads would be the most practical strategy, but since the degree of write-intensity is not always known *a priori*, we rely on the OBP feature to limit the damage. As Figure 9 shows, OBP effectively prevents performance overhead for write-dominant workloads, and columns 6 and 7 of Table 2 show that OBP filters the majority of the write traffic to NVRAM.

WiredTiger does not update existing blocks in place. A storage engine that does may be less sensitive to the phenomenon described in this section, but given a limited write throughput of Optane NVRAM we expect the lesson learned here to be broadly applicable.

3.2.4 *Summary.* We presented three lessons in design Optane NVRAM-resident caches:

(1) Detect workloads that fit into the OS buffer cache and do not admit their blocks.
(2) Admitting blocks into Optane NVRAM produces writes, which slow down the reads, i.e., cache lookups. The admission policy must balance the cost of admitting data into the cache against the benefit of using it later.
(3) Optane NVRAM caches benefit read-dominant workloads. For write-dominant workloads, the admission policy must minimize the number of admitted blocks.

Our admission policy uses the *small-bypass* feature to embody the first lesson, and the OBP feature to embody the second and third.

# 4 EVALUATION

We evaluate NVCache primarily using the YCSB benchmarks [8, 21]. Since we tuned the algorithms and the parameters of the NVCache using our in-house benchmarks (e.g., a "training set", to use an analogy from statistical modeling), we had to make sure that the algorithms perform well on previously unseen workloads (i.e., the "test set"), and we chose YCSB to fulfill that purpose. We provide supplementary figures showing our in-house benchmarks to add nuance to the results.

We ran experiments on the system described in §3.2.1, varying the amount of DRAM and NVRAM. Parameters of the YCSB benchmarks are shown in Table 3[6]. The DRAM cache size was set to half of the available DRAM[7], but capped at 40GB, except in the experiments where the goal was to observe the variation in engine's cache size (§4.3).

| Workload | Op mix, threads | Dataset |
|---|---|---|
| YCSB-A | 50% read, 50% update, 20 | 130GB |
| YCSB-B | 50% read, 50% update, 20 | 194GB |
| YCSB-C | 100% read, 20 | 259GB |
| YCSB-D | 95% read, 5% insert, 100 | 219GB |
| YCSB-E | 95% scan, 5% insert, 20 | 210GB |

**Table 3: YCSB characteristics**

Our evaluation asks three questions:

(1) How does NVCache compare to off-the-shelf solutions pursuing similar goals ?
(2) What is the effect of using an NVRAM cache on performance-per-$?
(3) Does using NVRAM block cache perform better than using hardware tiered memory and a larger page cache?

## 4.1 Comparison with off-the-shelf solutions

4.1.1 *Baselines used for comparison.* We compare with two solutions that permit using NVRAM as an extensions of DRAM, available in off-the shelf Optane systems: *Intel Memory Mode* (MM) [3] and *Intel Open Cache Acceleration Software* (OpenCAS) [3]. For potential future deployment of NVRAM in the field, it was important for us that these alternatives were available in standard Linux servers and did not require custom unsupported kernels.

*Intel Memory Mode* is a hardware configuration that presents Optane NVRAM to the rest of the system as regular volatile

[6] We did not include YCSB-F: it is modify-heavy, and modify operations in our storage engine were designed to trade performance for smaller cache footprint and smaller log records. Therefore, the overall throughput in modify operations was very low and insensitive to memory configurations.
[7] The engine's cache and the OS buffer cache share the available DRAM, so this setting gives each an equal share.

memory, and uses DRAM transparently as its cache, with data transferred between the two in units of cache lines. This is an attractive alternative, because it permits using NVRAM as an extension to DRAM without requiring any code changes, and makes it available for all data structures, in user space and kernel alike. In contrast, NVCache makes NVRAM available only for caching database file blocks.

Memory Mode can be enabled only in specific hardware configurations ([4], Table 17). We were able to successfully configure MM such that each NVDIMM was "paired" with a DRAM DIMM, meaning that it must be placed in the unused slot of the same channel of the same iMC (integrated memory controller) as the NVDIMM. Using additional DRAM DIMMs that were not paired with NVDIMMs produced configuration errors on our system, so we could only use the configuration with two NVDIMMs and two DRAM DIMMs. Our DRAM DIMMs were 16GB in size, so that restricted us to a configuration with 32GB of DRAM. Fortunately, MM could be configured to use all or part of the NVRAM, so we were able to vary the amount of NVRAM in the experiments.

In MM, the amount of total system memory is reported to be the same as the size of the NVRAM dedicated to MM. To answer questions (1) and (2) the WiredTiger's DRAM cache is configured to be half the size of the *physical* DRAM (see the beginning of §4). In that case, the kernel buffer cache will dynamically expand to use more plentiful system memory as the NVRAM size grows, using NVRAM for caching file blocks, just like NVCache. To answer question (3), we will vary the size of the WiredTiger page cache to use additional system memory.

*OpenCAS* is an open-source software project supported by Intel that allows using a fast block device as a cache for a slow block device, and it can be configured so that NVRAM acts as a block cache for the SSD – same idea as NVCache. OpenCAS can be configured in several modes [6]: *write-back*, *write-through*, *write-around*, *pass-through* (disabled) and *write-only* (allocate blocks only on write). Based on the lessons learned during admission policy design, *write-around* seemed the most appropriate configuration option: "*In write-around mode, the caching software writes data to the flash device if and only if that block already exists in the cache and [...] further optimizes the cache to avoid cache pollution in cases where data is written and not often subsequently re-read.*" [6]

**Alternative baselines not pursued:** Other alternatives to compare would be device mapper write cache (*dm-wc*) [2] and First Responder [33] – both OS-level block caches, and tiered memory systems, such as Nimble [38] and HeMem [32]. We considered comparing to *dm-wc* (the source code for First Responder is not available at the time of the writing), but upon analysing its properties we discovered that *dm-wc* admits blocks only on writes and does not throttle the admission rate. These properties contradict the lessons learned in this work. For example, *dm-wc* would admit zero blocks for read-only workloads, depriving the workloads that could benefit the most from a NVRAM cache. OpenCAS, in contrast, can be configured with flexible admission policies, superseding *dm-wc* in that regard.

Nimble [38] and HeMem [32] are tiered memory systems that transparently move application pages between DRAM and NVRAM depending on how the pages are accessed. We did not compare against them, because they both required custom kernels, which would be impractical to adopt in production. Furthermore, HeMem uses the NVRAM tier only for large allocations exceeding 1GB (HeMem specifically targets "big data" systems), so it would not use NVRAM for our engine's pages or blocks, whose size is on the order of a dozen kilobytes. In §4.3 we evaluate using a larger page cache with MM, which in many respects is similar to a transparent tiered-memory system. Strictly speaking, Intel Memory Mode is *not* a tiered system, because DRAM acts as a cache for NVRAM, but it comes close.

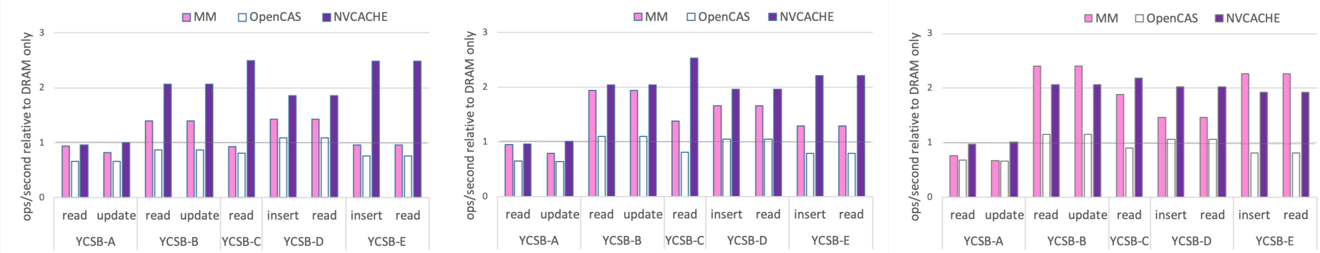
*4.1.2 Results.* Figure 7 shows the throughput of the memory mode (MM), OpenCAS and NVCache with 32GB DRAM and 64GB, 128GB and 252GB of NVRAM relative to using no NVRAM at all. We make the following observations:

**Observation 1:** OpenCAS cache derives no performance benefit from NVRAM. This occurs because OpenCAS does not throttle the admission rate. OpenCAS delivers similar or better read hit rate as the NVCache (numbers not shown), but also makes two orders of magnitude more writes to NVRAM. Even for read-only workloads, admitting data into cache produces writes. ***Failing to throttle the admission rate to NVRAM is the main reason why OpenCAS fails to perform***.

**Observation 2:** Memory mode outperforms or performs comparably to NVCache when NVRAM is ample, as shown in Figure 7(c). In that configuration, the size of the NVRAM is 252GB, and the marginal utility of NVCache is small after the memory size reaches 128GB. For example, increasing the NVRAM size from 64GB to 128GB, NVCache hit rate grows by about 20%, but going from 128GB to 252GB, it grows by only another 5%. So as NVRAM grows beyond 128GB, NVCache brings little extra benefit, but MM, being able to cache not only file blocks but other data structures (including the kernel buffer cache), brings further improvements.

On the other hand, we observe that Memory Mode hurts performance of the write-intensive YCSB-A (by about 30%), while NVCache keeps it unchanged.

**Observation 3:** When the dataset size exceeds NVRAM capacity, NVCache provides substantially better performance than Memory Mode. As shown in Fig. 7(a), NVCache outperforms the memory mode by between 30% (for YCSB-B) and 169% (YCSB-C). Further, the memory mode hurts YCSB-A's



**(a) 64GB NVRAM**

**(b) 128GB NVRAM**

**(c) 252GB NVRAM**

**Figure 7: YCSB throughput under memory mode, OpenCAS and NVCache relative to 32GB DRAM and zero NVRAM.**

update throughput by about 18% relative to the DRAM-only baseline, while NVCache doesn't. We conclude that a cache tailored to throttle the admission rate can be superior to Memory Mode when the dataset size substantially exceeds the available NVRAM.

*4.1.3 Combining Memory Mode and NVCache.* We also experimented with configurations where part of the NVRAM is dedicated to MM and the remainder is used for NVCache, reasoning that we could size NVCache such that its marginal utility is highest ( 128GB), and the rest of the NVRAM could be used as MM's system memory for the benefit of other data structures. Unfortunately, we observed orders of magnitude worse throughput than with either MM or NVCache alone, and did not pursue this avenue further.

## 4.2 Performance vs. cost

In this experiment we take a fixed memory budget of 96GB and vary the fraction used by DRAM and NVRAM as shown in Table 4[8]. We perform the experiments in this section using only NVCache, as we are unable to vary the amount of DRAM used in MM (see §4.1.1) and OpenCAS proved to be not competitive.

| NVRAM | DRAM | Relative cost |
|---|---|---|
| 0GB | 96GB | 1 |
| 16GB | 80GB | 0.90 |
| 32GB | 64GB | 0.79 |
| 48GB | 48GB | 0.69 |
| 64GB | 32GB | 0.59 |

**Table 4: NVRAM and DRAM amounts and the cost of all system memory relative to an all-DRAM setup.**

We use the NVRAM/DRAM per-byte cost ratio of 0.38, same as in a recent study with Optane memory [28]. As the amount of NVRAM increases and the amount of DRAM decreases, the total cost of system memory also decreases, as shown in Column 3.

Figure 8(a) shows the performance of YCSB normalized to the 96GB DRAM configuration and divided by the cost ratio in Column 3. In other words, these are performance/$ numbers relative to the DRAM-only configuration. Positive numbers mean that the performance decreased less than the memory cost. Read-only or read-mostly workloads that benefit from the NVCache (see cache hit ratios in Fig. 8(b)) experience a positive gain, as expected.

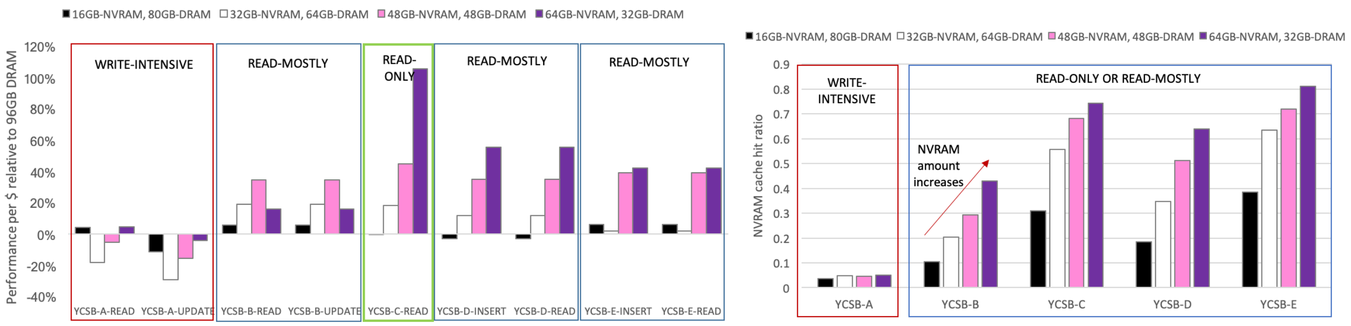
While in most cases performance predictably drops as the amount of DRAM decreases, YCSB-C in configuration with 64GB NVRAM and 32GB DRAM actually performs better than it does with 96GB DRAM – so we decrease the system cost *and* improve performance in absolute terms. This occurs because beyond 32GB DRAM the utility of additional DRAM (and a larger page cache) is smaller than the loss in performance due to a smaller NVCache.

YCSB-A, whose write intensity makes caching futile, suffers the overall loss in terms of performance/$. Its performance drops at a steeper rate than the memory cost as we decrease the amount of DRAM.

We also performed identical experiments using our in-house benchmarks (see Figure 9). As most of them are write-intensive and their performance does not change with the introduction of NVCache (see §3), here we present read-intensive benchmarks (with *large* datasets): *evict-btree* (read-only), *evict-btree-scan* (95% reads), *medium-btree* (read-only), and *evict-btree-stress-multi* (80% reads, 20% updates). See Table 2 for their characteristics. The results show the same pattern as with YCSB. Performance/$ increases favourably as the NVRAM size increases for benchmarks that perform only or mostly reads. Although 80% of the operations in *evict-btree-stress-multi* are reads, the prominent presence of writes means that trading DRAM for NVRAM is not cost-effective.

We conclude that NVRAM is a cost-effective method of reducing memory cost while balancing the impact on performance for read-dominant workloads, where in some cases we

[8] We do not use the configuration with 16GB DRAM, because a scarce DRAM amount triggered a known kernel bug in the DAX code (at fs/inode.c:530).



**(a) Performance per $**

**(b) NVCache hit ratio**

**Figure 8: YCSB performance per dollar and NVCache hit ratio under increasing NVRAM and decreasing DRAM.**

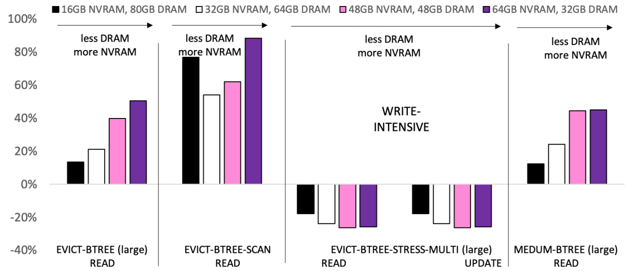


**Figure 9: Performance per dollar under increasing NVRAM and decreasing DRAM for several in-house benchmarks.**

can both reduce cost *and* improve performance. At the same time, even a modest presence of writes can render NVRAM unprofitable relative to DRAM – an observation shared by the authors of the Meta's RocksDB study [28]. .

## 4.3 A larger page cache with memory mode

We explored using NVRAM for a block cache, but another alternative is to use it as a transparent extension of system volatile memory to enable a larger WiredTiger page cache. In the experiments described in this section we use NVRAM in the Memory Mode, and increase the size of the WiredTiger page cache. We compare this configuration to the NVCache configuration from the previous section.

Figure 10 shows the throughput of YCSB with 32GB of DRAM and 64GB, 126GB and 252GB of NVRAM. The NVCache configuration uses a 16GB WiredTiger page cache, as in the previous section. For the configurations without NVCache, we vary the size of the WiredTiger page cache as much as the NVRAM capacity allows: 16GB, 32GB, 40GB, 80GB and 160GB.

Our conclusions are similar to those in the previous section: When the amount of NVRAM is small relative to the dataset size, using NVCache is superior to using the Memory Mode, even if we allow the WiredTiger page cache to expand. When NVRAM is plentiful (the 252GB configuration), using Memory Mode is often (though not always) advantageous to using the block cache, in some cases by a large margin. In-between those extremes using NVCache is preferable for some workloads, but not for others.

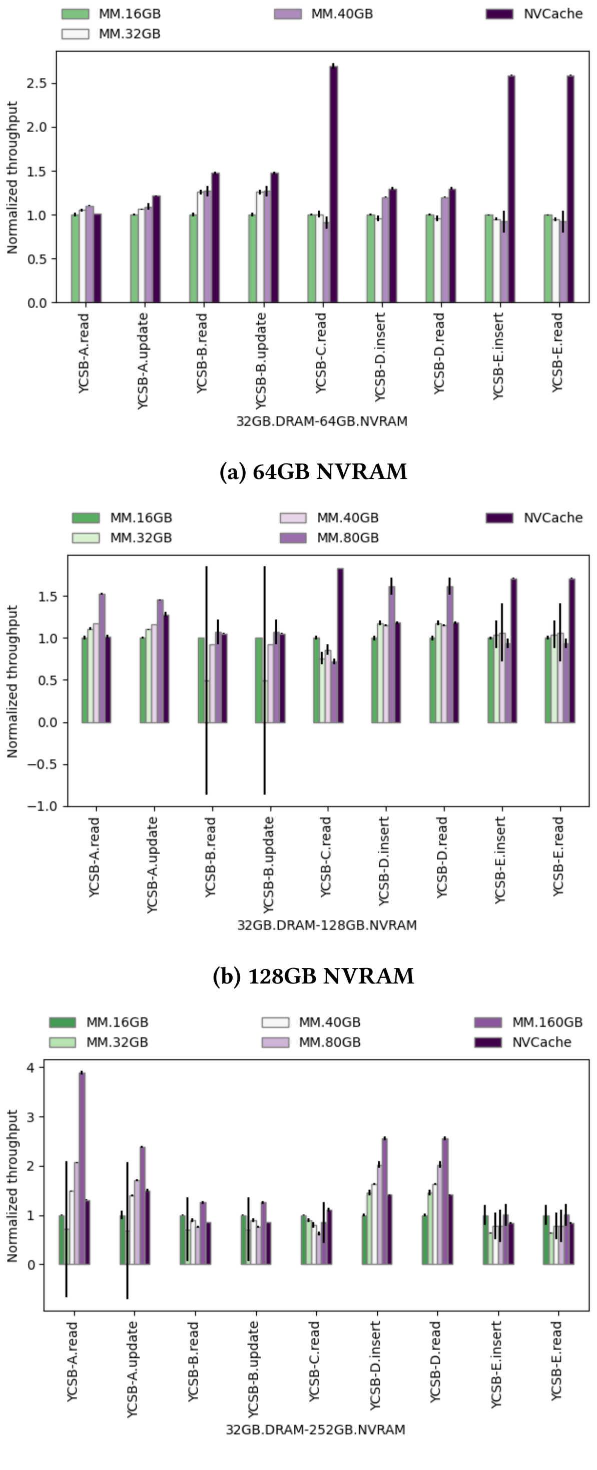
One anomaly evident in Figure 10 is a very large variation in running times that we often observed in the Memory Mode, but never with the NVCache[9]. We don't fully understand this phenomenon, but the variability suggests that Memory Mode is not consistently keeping the hottest WiredTiger data in DRAM. If Memory Mode used direct mapping when caching NVRAM-resident data in DRAM, conflict misses could be the reason for performance variability, but we could not confirm this hypothesis. In any case, a more predictable performance of NVCache positions it as a better candidate for production deployment.

## 4.4 Summary

Our evaluation revealed that the memory mode is a competitive off-the-shelf alternative to a custom NVRAM cache when the amount of NVRAM is ample, but when it is scarce a custom cache solution such as NVCache will deliver better performance. OpenCAS is not competitive with either NVCache or the memory mode.

NVRAM is a cost-effective method of reducing memory cost while balancing the impact on performance for read-dominant workloads, where in some cases we can both reduce cost *and* improve performance as DRAM is swapped in favour of NVRAM. For write-intensive workloads, however, replacing part of DRAM with NVRAM is not a cost-effective option.

[9] We did not include error bars in our other figures, because the variance was always small.



**(a) 64GB NVRAM**

**(b) 128GB NVRAM**

**(c) 252GB NVRAM**

**Figure 10: YCSB throughput in memory mode and with NVCache. Bars labeled *MM.NGB* correspond to configurations in memory mode with varying sizes of the WiredTiger page cache, where *N* stands for the size of the page cache. The NVCache configuration uses a 16GB DRAM page cache and the NVRAM block cache, which has the size of the available NVRAM. The numbers are normalized to *MM.16GB*.**

# 5 RELATED WORK

The most similar and recent counterpart to our study is a volatile Optane-resident cache for Meta’s RocksDB [28]. That work takes RocksDB’s DRAM block cache and turns it into a two-tiered cache of DRAM and NVRAM, making it similar to tiered memory systems. Like other tiered memory systems, it addresses the question of how to split the cached data between the DRAM and the NVRAM tiers. We present a different design, one that uses a stand-alone block cache interposed between the DRAM cache and the block device. Although the RocksDB study also used Optane NVRAM as the cache media, it does emphasize the impact of writes on simultaneously occurring reads – a new finding in our work – and does not factor this phenomenon into the admission policy.

HeuristicDB [40] is a cooperative block layer cache that uses a fast Optane SSD as a caching tier in front of a slower drive. HeuristicDB admits all blocks read from and written to the block device, except those part of sequential access pattern. While this generous admission policy might work for Optane SSDs, we demonstrated that it is unacceptably costly for Optane NVRAM.

MyNVM is another key-value store based on RocksDB that uses Optane SSD as the medium for an internal block cache [25]. Similarly to NVCache, MyNVM caters its admission policy to the properties of the Optane device, but pursues different goals: (1) to extend its endurance MyNVM admits only carefully selected keys, and (2) to maximize its bandwidth it accumulates keys into relatively large 128KB blocks before writing them to the device. While we did not focus on improving endurance, write throttling performed in NVCache should increase it. The second goal is potentially applicable to our NVRAM device too, but prior experiments ([39], Fig. 5) showed that writing into Optane NVRAM blocks larger than those that we already write (e.g., 16KB+) does not improve its bandwidth.

Flashield [24] admission policy aims to predict which objects will be frequently read, but not updated and caches those objects by writing them in large chunks. (CacheLib’s ML policy [1, 15] is similar, modeled on that of Flashield.) So for read-only workloads, Flashield could potentially admit objects at a high rate, generating writes into the cache. In contrast, we found that admission throttling is crucial even for read-only workloads (see §3.2.3, *Lesson 2*). Admission rate can become high even for workloads without updates, if popularity of cached read-only blocks fades rapidly and new blocks are admitted in their place. If that occures, our policy will throttle admission and limit the harmful effect of writes, while Flashield would not. In summary, Flashield offers a useful approach for predicting which blocks will be most useful to cache, but for caches built on Optane NVRAM admission throttling proposed in our work is a crucial design feature.

CacheLib, by default, admits objects into a cache with a probability $p$. Although setting $p$ to a low number could accomplish a similar effect to that of OBP, $p$ in CacheLib is a statically configured parameter. We found that admission throttling rate must cater to the workload properties. E.g., NVCache would admit hardly any blocks for write-intensive workloads, while filling up the cache for read-intensive ones.

The work by Arulraj et al. [14] establishes a broad framework for reasoning about multi-tiered caching systems comprised of DRAM, persistent memory and SSDs. The authors propose an algorithm for data placement that dynamically tunes the following probabilities: the probability of bypassing DRAM on reads and writes (where data would be read/written directly from/to NVRAM) and the probability of bypassing the NVRAM on reads and writes. Bypassing DRAM is not applicable in our engine, because our DRAM cache stores data in a different format than our NVRAM cache, but bypassing NVRAM is exactly the question we investigated in the design of our admission policy. Within Arulraj's framework, our admission policy proposes a solution for tuning the variables $N_r$ (the probability of bypassing the NVRAM cache during reads) and $N_w$ (the probability of bypassing the NVRAM cache during writes) – see Sections 3.3 and 3.4 in [14]. Similar to our work, Arulraj et al. express the optimization function as the combination of cost (writes) and benefit (throughput). In our system, with the OBP metric we approximate the cost via insertions and removals (writes), and the benefit via lookups, which also correlate with the throughput of our storage engine. The difference of Arulraj's work is that their system is tuned using simulated annealing, while we use a single metric that dynamically controls the admission rate and is robust across different workloads. Using a simple metric is advantageous for a large production codebase.

Estro et al. explored the relationship of performance and cost and the effects of different cache settings (such as write-through vs. writeback) in multi-tier caching configurations on real hardware [26]. Performing similar analysis would be a natural extension of our work, but can only be done after understanding the idiosyncrasies of cache design using recently adopted memory technology, contributed by our study.

Dulong et al. built an eponymous system (NVCache) that acts as a user-level write cache for the file system [23]. Despite sharing the name, that system is orthogonal to ours: it uses NVRAM solely for caching writes. It is very similar to the device mapper write cache (discussed in §4), except it gains efficiency from running at user level.

The design of Orthus [36] was driven by an observation similar to ours: a seemingly faster device (Optane SSD, in their case) outperforms a slower device (a flash-based SSD) in general, but lags behind it under high concurrency. Orthus embraces a hybrid design: initially, a faster device acts as a cache for a slower device, admitting all blocks until a desired hit rate is accomplished. Then Orthus switches to a "tiered mode", where the load is distributed among both devices to maximize the overall throughput. Our OBP feature accomplishes a somewhat similar effect when it begins throttling the admission rate to NVCache, and as a result more reads are being sent to the storage device over time. In contrast to Orthus, NVCache throttles the admission rate based on the observed cost/benefit metric, and not as a consequence of achieving a certain hit rate. In fact, we observed (§3.2, *Lesson 2*) that it may be beneficial for overall performance to throttle the admission rate at the expense of the reduced hit rate.

Multi-tiered memory systems focus primarily on policies for selecting the right tier for a memory page, and (to that end) efficiently tracking page access patterns [12, 13, 22, 27, 32, 35, 37**?** , 38]. Our decision to make NVCache independent from the DRAM cache makes these techniques complementary. We evaluated Intel Memory Mode as a hardware based tiered memory-like system, where the WiredTiger page cache can expand into the additional memory provided by NVRAM, and saw mixed results. We did not evaluate software solutions, such as Nimble [38] and HeMem [32]: they required custom kernels that were impractical do adopt in the field.

# 6 CONCLUSION

Although it was well known that Optane NVRAM delivers limited write throughput, it was not known that writes disproportionately affect the throughput of reads. We discovered that in the presence of a single writer thread, the throughput of reads drops almost by a factor of 4×. In contrast, with DRAM used in the same experiment the impact on read throughput was only 18%. This discovery led us to propose a new admission policy for Optane-resident caches. Our policy throttles the rate of writes to the cache (generated by the admission of new data, removal of invalid data and eviction), with the rate of reads, i.e., cache lookups. The metric capturing this principle, the Overhead Bypass Threshold, is generic and can be applied in any cache residing on hardware with similar properties. Our implementation outperforms an off-the-shelf cache from OpenCAS across the board, and the hardware tiered memory system (Intel Memory Mode) in all cases where the dataset size exceeds the amount of NVRAM.

# 7 AVAILABILITY

The WiredTiger source code, including NVCache, is available as open source software [11].

# REFERENCES

[1] 2021. CacheLib, Facebook's open source caching engine for web-scale services. https://engineering.fb.com/2021/09/02/open-source/cachelib.

[2] 2021. Device Mapper Write Cache. www.kernel.org/doc/html/latest/admin-guide/device-mapper/cache.html.

[3] 2021. Intel Memory Mode. https://software.intel.com/content/www/us/en/develop/articles/qsg-intro-to-provisioning-pmem.html.

[4] 2021. Intel® Server Board S2600WF Product Family. Technical Product Specification. https://www.intel.com/content/dam/support/us/en/documents/server-products/server-boards/S2600WF_TPS.pdf.

[5] 2021. Open Cache Acceleration Software. https://open-cas.github.io.

[6] 2021. Open Cache Acceleration Software: Admin Guide. https://open-cas.github.io/guide_configuring.html.

[7] 2021. Persistent Memory Development Kit. https://pmem.io/pmdk.

[8] 2021. Yahoo! Cloud Serving Benchmark, Git Repo. https://github.com/brianfrankcooper/YCSB.

[9] 2022. Clock Page Replacement Algorithm. Wikipedia. https://en.wikipedia.org/wiki/Page_replacement_algorithm#Clock.

[10] 2022. MongoDB. https://www.mongodb.com.

[11] 2022. WiredTiger Storage Engine. Block cache. https://github.com/wiredtiger/wiredtiger/blob/develop/src/block_cache/block_cache.c.

[12] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. *SIGPLAN Not.* 52, 4 (April 2017), 631–644.

[13] Shoaib Akram, Kathryn S. McKinley, Jennifer B. Sartor, and Lieven Eeckhout. 2018. Managing Hybrid Memories by Predicting Object Write Intensity. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 75–80.

[14] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. 2019. Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory. *CoRR* abs/1901.10938 (2019). arXiv:1901.10938 http://arxiv.org/abs/1901.10938

[15] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768.

[16] Muhammad Bilal and Shin-Gak Kang. 2017. A Cache Management Scheme for Efficient Content Eviction and Replication in Cache Networks. *IEEE Access* 5 (2017), 1692–1701.

[17] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD Bufferpool Extensions for Database Systems. *Proceedings of the VLDB Endowment* 3, 2 (2010), 1435–1446.

[18] Yuxia Cheng, Wenzhi Chen, Zonghui Wang, Xinjie Yu, and Yang Xiang. 2015. AMC: an adaptive multi-level cache algorithm in hybrid storage systems. *Concurrency and Computation: Practice and Experience* 27, 16 (2015), 4230–4246.

[19] Yuxia Cheng, Yang Xiang, Wenzhi Chen, Houcine Hassan, and Abdulhameed Alelaiwi. 2018. Efficient cache resource aggregation using adaptive multi-level exclusive caching policies. *Future Gener. Comput. Syst.* 86 (2018), 964–974.

[20] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic Cloud Caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, Santa Clara, CA.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. 143–154.

[22] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 16 pages.

[23] Rémi Dulong, Rafael Pires, Andreia Correia, Valerio Schiavoni, Pedro Ramalhete, Pascal Felber, and Gaël Thomas. 2021. NVCache: A Plug-and-Play NVMM-based I/O Booster for Legacy Systems. In *51th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 21)*.

[24] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[25] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 13 pages.

[26] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. 2020. Desperately Seeking ... Optimal Multi-Tier Cache Configurations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association.

[27] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 521–534.

[28] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 821–837.

[29] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 1999. On the Existence of a Spectrum of Policies That Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Atlanta, Georgia, USA) *(SIGMETRICS '99)*. Association for Computing Machinery, New York, NY, USA, 134–143.

[30] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. 2014. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 133–144.

[31] Sundaresan Rajasekaran, Shaohua Duan, Wei Zhang, and Timothy Wood. 2016. Multi-cache: Dynamic, Efficient Partitioning for Multi-tier Caches in Consolidated VM Environments. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 182–191.

[32] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *28th ACM Symposium on Operating Systems Principles (SOSP'21)*.

[33] Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh. 2021. First Responder: Persistent Memory Simultaneously as

High Performance Buffer Cache and Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 839–853.

[34] ]panthera Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. [n. d.]. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories.

[35] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting Program Semantics to Place Data in Hybrid Memory. In *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*. IEEE Computer Society, 163–173.

[36] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 307–323.

[37] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime Data Managementon Non-Volatile Memory-Based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 58, 14 pages.

[38] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 331–345.

[39] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182.

[40] Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2021. *HeuristicDB: A Hybrid Storage Database System Using a Non-Volatile Memory Block Device*. Association for Computing Machinery, New York, NY, USA.

[41] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. 2017. AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In *36th IEEE International Performance Computing and Communications Conference, IPCCC 2017, San Diego, CA, USA, December 10-12, 2017*. IEEE Computer Society, 1–8.