Accelerating Database Workloads with DM-WriteCache and Persistent Memory

Rajesh Tadakamadla
Hewlett Packard Enterprise
rajesh.tadakamadla@hpe.com

Mikulas Patocka
Redhat
mpatocka@redhat.com

Toshi Kani
Hewlett Packard Enterprise
toshi.kani@hpe.com

Scott J Norton
Hewlett Packard Enterprise
scott.norton@hpe.com

ABSTRACT

Businesses today need systems that provide faster access to critical and frequently used data. Digitization has led to a rapid explosion of this business data, and thereby an increase in the database footprint. In-memory computing is one possible solution to meet the performance needs of such large databases, but the rate of data growth far exceeds the amount of memory that can hold the data. The computer industry is striving to remain on the cutting edge of technologies that accelerate performance, guard against data loss, and minimize downtime. The evolution towards a memory-centric architecture is driving development of newer memory technologies such as Persistent Memory (aka Storage Class Memory or Non-Volatile Memory [1]), as an answer to these pressing needs. In this paper, we present the use cases of storage class memory (or persistent memory) as a write-back cache to accelerate commit-sensitive online transaction processing (OLTP) database workloads. We provide an overview of Persistent Memory, a new technology that offers current generation of high-performance solutions a low latency-storage option that is byte-addressable. We also introduce the Linux kernel's new feature "DM-WriteCache", a write-back cache decades the computing industry has been researching ways to reduce the performance gap implemented on top of persistent memory solutions. And finally we present data from our tests that demonstrate how this technology adoption can enable existing OLTP applications to scale their performance.

CCS CONCEPTS

• Information systems → Transaction logging • Information systems → Database performance evaluation • Hardware → Non-volatile memory

KEYWORDS

Persistent Memory; Storage Class Memory; Device Mapper; Write-back Cache; Write Latency Improvement; DM-WriteCache; Online Transaction Processing; Database

ACM Reference format:
Rajesh Tadakamadla, Mikulas Patocka, Toshi Kani and Scott Norton. 2019. Accelerating Database Workloads with DM-WriteCache and Persistent Memory. In *Proceedings of ACM/SPEC International Conference on Performance Engineering (ICPE'19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297663.3309669

1 INTRODUCTION

1.1 Persistent Memory and its types

Today's businesses demand real-time data to realize faster business outcomes. These businesses need systems that offer uncompromising performance that makes data available as quickly and reliably as possible. For between the low-latency processor and higher-(longer) latency storage devices. The progression of storage technology has thus evolved both in how data is accessed and how data is stored (magnetic to solid-state media). A recent trend has emerged of moving storage functionality to the memory bus [2], thus taking advantage of memory interconnects' low latency and fast performance. Placing storage devices on the memory bus offers something more: the prospect of byte-addressable storage, a new semantic that cuts through cumbersome software layers, and offers sub-microsecond device latencies. Multiple OEM vendors offer solutions in the space that are either performance optimized or capacity optimized or both.

Most of the performance optimized offerings fall under the NVDIMM-N category that deliver DRAM level performance but are in limited capacities. Intel Optane DC Persistent Memory [5] offers significantly higher capacities, but at lower than DRAM level performance. One unique offering that is optimized for both performance and capacity is HPE's Scalable Persistent Memory [6]. Each of these technologies provide native persistence or added persistence through platform enablement to regions of memory from DIMMs.

- NVDIMM-N devices get their persistence by backing up the content onto NAND flash memory located on the DIMM module during power loss by utilizing battery power. When a machine is powered back up, the firmware restores the backed-up contents back onto the DIMM module before resuming/beginning the normal system boot process. [3][4]

- Intel's Optane DC Persistent Memory devices employs a new technology termed 3D XPoint (pronounced 3D CrossPoint) in the DIMM form factor. This technology resembles NAND flash in terms of behavioral characteristics and offers native data persistence without additional hardware with a capability to scale to double digit terabyte capacities on a single server.
- HPE's Scalable Persistent Memory solution relies on server platform enablement to guarantee data persistence on memory regions by relying on external/internal battery backup and a set of NVMe drives as backup targets. This enables the solution to offer Persistent Memory that is as performant as DRAM and can also scale to offer multi-terabyte capacities.

1. 2 DM-WriteCache

It's an open source implementation by Mikulas Patocka of a write-back cache at the Device Mapper layer of Linux with the cache residing on low latency storage media. In its current implementation the feature supports hosting the cache on byte-addressable Persistent Memory or block-addressable devices like NVMe or SSD. DM-WriteCache attempts to cache only writes and relies on the application's caching or kernel's page cache for read caching. [7]

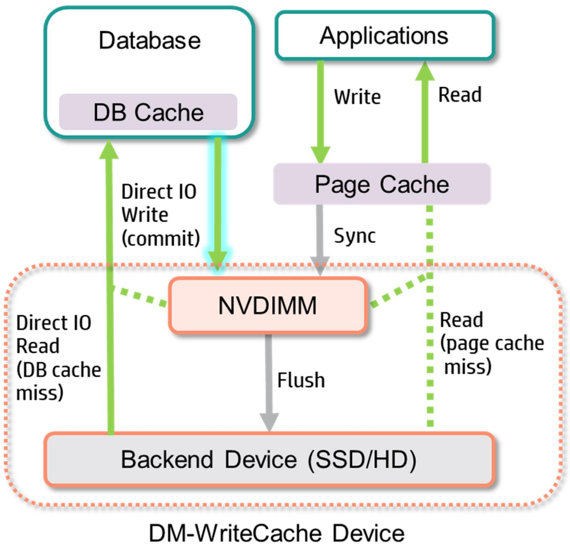


Figure 1: High level working of a DM-WriteCache device.

All read and write requests to a DM-WriteCache'ed device go through look-up in the cache before going to backing store. For applications that rely on the kernel's page cache, a major page fault will cause the data to be read from the backing store and on any subsequent sync/fsync events, the data is written to the cache and acknowledged. Similarly, in case of applications or databases that manage their own memory and perform Direct IO, all the writes are acknowledged from the cache while reads come from the backing store after a cache lookup.

In the following sections, we describe the usage of a DM-WriteCache device constructed using NVDIMM-N devices for caching and SAS SSD devices for the underlying storage with database workloads. We use a commit-sensitive TPC-C-like OLTP workload against an Oracle database to demonstrate the performance gains through the use of these write-back-cached devices. We evaluated the use of cached devices to host both database REDO logs and DATA files, but chose to focus REDO on the configuration where performance gains are significant.

This paper is organized in the following manner: Section (2) describes the test environment and configuration. Section (3) lists our performance observations and presents performance benefits of the cached configuration. Section (4) walks you through the statistics and discusses the factors and cached device inner workings that influence the performance gain. In Section (5), we discuss results from workload runs against the cached configuration with various environment modifications. Memory can be moved across the persistent and volatile pools as supported by HPE Scalable Persistent Memory or by simply swapping out NVDIMM-Ns in lieu of regular DIMMs. For such cases, we present our performance observations in Section (6) when a cached device is used for hosting database data files with an inadequately-sized Shared Global Area (SGA). Section (7) provides a conclusion on the effort.

2. TEST ENVIRONMENT

2.1 HARDWARE

For the server under test (SUT) running the database instance, we used a 2-socket HPE ProLiant DL380 Gen9 running Intel Xeon E5-2699 v4 processors each having 22 cores. Table 1 describes the core server configuration in detail while Figure 2 depicts the layout.

Table 1: Server configuration

Server	HPE ProLiant DL380 Gen9
CPUs	2 x Intel Xeon E5-2699 v4
Memory	16 x 32GB RDIMMs
Persistent Memory	8 x HPE 8G NVDIMM-N
Physical core count	44
Logical core count	88
RAM per CPU	256GB
Network	4 x 1Gbps LOM ports
Storage Controller	Smart Array P440ar Controller
Drives	2 x 800GB MU SAS SSD for OS 2 x 800GB MU SAS SSD for REDO 4 x 800GB MU SAS SSD for Data
RAID Configuration	1 x RAID1 for Operating System 2 x RAID0 for REDO 1 x RAID0 for DATA (4 disks)

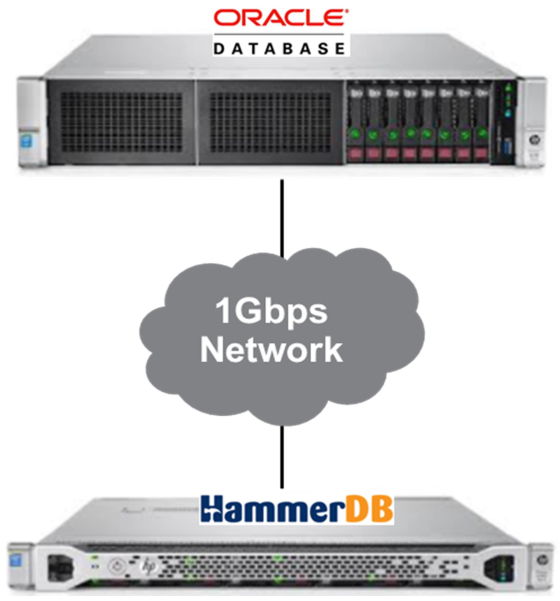


Figure 2: Overview of the hardware

2.2 OPERATING SYSTEM

The system under test (SUT) was running Red Hat Enterprise Linux 7.4 with a custom 4.14 kernel that was built to support Persistent Memory, DAX (direct access for files that removes unnecessary page-cache copying) [6] and DM-WriteCache. The DM-WriteCache feature was backported to multiple kernel versions for this effort.

We evaluated the performance of the database when REDO and DATA files were hosted on both the XFS file system and Oracle's ASM (Automatic Storage Management) [7], but chose to limit the content of the paper to XFS configuration.

2.3 DATABASE

A single instance Oracle 12cR2 non-container database was used for the evaluation. The COMMIT configuration of the database was left unaltered with following parameter settings:

COMMIT_LOGGING = IMMEDIATE
COMMIT_WAIT = WAIT

For Oracle 12cR2, the default log writer setting is ADAPTIVE mode.

_USE_SINGLE_LOG_WRITER = ADAPTIVE

By default, the database tends to use parallel log_writer slaves to write to the REDO logs and based on sampling count and intervals defined by "_adaptive_*_log_writer_*" parameters, the database switches to single log_writer master during a benchmark run. We chose to test with this default behavior as it demonstrated additional capabilities of DM-WriteCache feature.

The database was configured to have 3 x 100GB 4K REDO log groups with single log members for most of the evaluation runs. However, to meet the minimum durability requirements, we upgraded the configuration to support 3 x REDO log groups with 2 x log members where each log member was hosted on two separate devices to assess the performance impact, if any, due to REDO multiplexing.

2.4 WORKLOAD

We used HammerDB v2.23 to simulate TPC-C like OLTP workload against the SUT. We employed two distinct workloads for evaluation:

(a) REDO log IO intensive workload: For this type, we configured an SGA that is significantly larger than the data disk footprint. By the end of ramp-up phases of a benchmark run, we have data cache at 100%. During the timed benchmark interval, only IO is to the REDO logs.
(b) DATA file IO intensive workload: We sized the SGA to be smaller than the data disk footprint. During a benchmark run against this configuration, we constantly see data blocks being read into the buffer and modified blocks being written back to the disk due to insufficient buffers in the cache.

The database was seeded with data for 3200 warehouses and supported HASH clusters. For the benchmark runs, a user count of 112 was used to generate the workload. The users were configured not to have any think-time to be able to generate significant stress on the SUT.

To have comparable data, we used the same SSD device in both cached and non-cached configurations. Statistics are compared from two back-to-back runs across both configurations to keep the data set size delta to the minimum. Each run had a ramp-up phase of 2 minutes and a timed benchmark run of 5 minutes. An Automatic Workload Repository (AWR) snapshot is triggered before and after the timed benchmark run using which AWR reports are generated.

2.5 CACHE DEVICE CONSTRUCTION

For most of the initial evaluation, we used three REDO log groups with single member each and all of them hosted on a single SSD device. For caching, we used an 8GB x 4 interleaved 32GB device from the first socket. Cached device was constructed as follow:

```
# dmsetup create dm_log --table "0 1562758832 writecache p
/dev/sdb /dev/pmem0 4096 4 high_watermark 10 low_watermark
5“
```

The second value within the quotation marks is the block size of the backend device or in other words the target for caching. The fourth value describes whether the caching tier would be a persistent memory device "p" or a SSD/NVMe device "s". One key advantage of Persistent Memory usage is that the feature capitalizes on DAX for improved performance. The fifth and sixth arguments specify the backend device to be cached and the device used for cache respectively. Seventh argument defines the block size of the cache entries. DM-WriteCache supports a number of optional parameters that control how the cache behaves, so the eighth parameter from the command represents the count of optional arguments passed to the command.

The "high_watermark" option defines the percentage of the used cached entries that would trigger flushing or de-staging of cached data to its backing store. Once the flushing is triggered, the "low_watermark" option defines the percentage of used cached entries at which the flushing of data has to stop. However, the development version of the feature that we used did not have the "low_watermark" threshold implemented and this value was silently ignored.

2.6 METRICS

We used one client-side metric and one database-side metric to gauge the overall performance of each configuration. Transactions per Minute (TPM) is the primary client-side metric that reports average transaction rate per minute for the timed benchmark run.

From the database standpoint, for REDO-intensive workloads, we looked at average wait time per "log file sync" wait event as reported by the Automatic Workload Repository (AWR) report. A "log file sync" event is triggered when a user session issues a commit (or a rollback) and concludes with a signal/post on successful flush of log buffer entry to redo log file back to the user session. Reported values are in microseconds or milliseconds. For the DATA-file-intensive workloads, the second metric that we used is the average wait per "free buffer waits" event. This event gets triggered when a session needs to load a block of data from the disk or needs to clone a read-consistent buffer, but is unable to find a free buffer. The wait event is timed until some of the dirty buffers are flushed to the disk and free buffers are made available.

3. PERFORMNCE OBSERVTIONS

All observations discussed in this section are from performance runs with a 4.14 kernel configuration where we have observed optimistic results when compared to other configurations. Comparing the client-side TPM metric across cached and non-cached configurations, we see a 22% improvement with cached configuration.

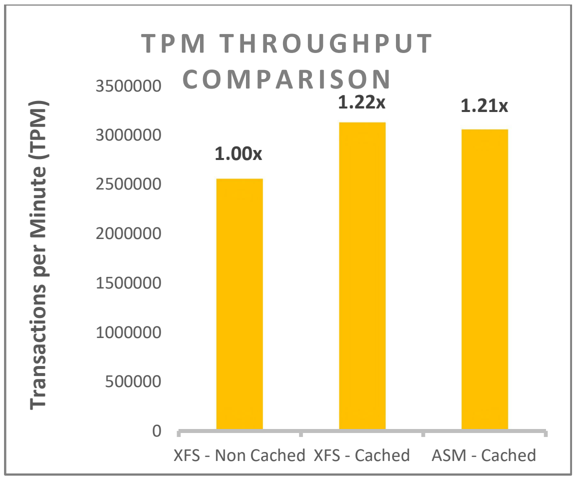


Figure 3: TPM Throughput comparison between cached and non-cached configuration

Comparing the database-side of the metrics, we see an increase in count of "log file sync" proportionate to the overall increase in TPM. We see the average wait per event drop to 390 µs from 893 µs which is a 40+% drop.

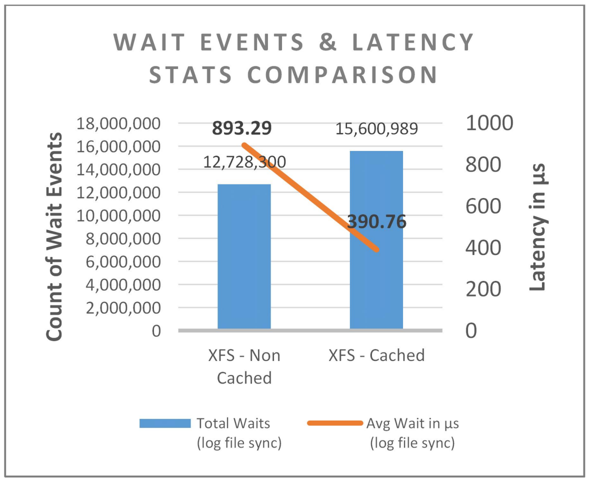


Figure 4: Wait counts and average wait value comparison.

To validate that both runs against cached and non-cached configurations were similar, we took a look at the top foreground waits listed in AWR reports.

Tables 2 and 3: Top foreground wait event comparison between non-cached (top table) and cached (bottom table) configuration.

Event	Waits	Total Wait Time (sec)	Avg Wait	% DB time	Wait Class
DB CPU		13.7K		49.1	
log file sync	12,728,300	11.4K	893.29us	40.6	Commit
library cache: mutex X	329,764	1106.4	3.36ms	4.0	Concurrency
db file sequential read	1,081,898	324.2	299.64us	1.2	User I/O
db file scattered read	774,201	275.2	355.46us	1.0	User I/O
latch: In memory undo latch	498,940	84	168.28us	.3	Concurrency
cursor: mutex X	7,189	48.3	6.72ms	.2	Concurrency
buffer busy waits	152,875	33.4	218.36us	.1	Concurrency
PGA memory operation	1,051,070	11.5	10.91us	.0	Other
SQL*Net message to client	9,789,645	7.6	779.99ns	.0	Network

Event	Waits	Total Wait Time (sec)	Avg Wait	% DB time	Wait Class
DB CPU		16.6K		61.3	
log file sync	15,600,989	6096.2	390.76us	22.5	Commit
library cache: mutex X	481,126	2192.7	4.56ms	8.1	Concurrency
db file sequential read	1,153,516	357.6	309.99us	1.3	User I/O
db file scattered read	803,353	306.4	381.38us	1.1	User I/O
latch: In memory undo latch	683,051	112.6	164.91us	.4	Concurrency
cursor: mutex X	9,245	68.2	7.38ms	.3	Concurrency
buffer busy waits	222,435	61.8	277.75us	.2	Concurrency
PGA memory operation	1,117,515	12.1	10.83us	.0	Other
latch: undo global data	5,638	9.1	1.62ms	.0	Other

Wait events from AWR reports clearly show that the only significant wait event reported is "log file sync". Again, DB_CPU might not indicate a productive CPU utilization, but tuning to improve the CPU efficiency would benefit both configurations.

We will discuss a few more performance observations with configuration changes in subsequent sections where the TPM throughput gains take an impact.

We see higher CPU utilization rates with runs against the cached configuration where we achieve improved TPM. On an average, we see about 10% higher CPU utilization rates in cached-configurations compared to the non-cached configurations.

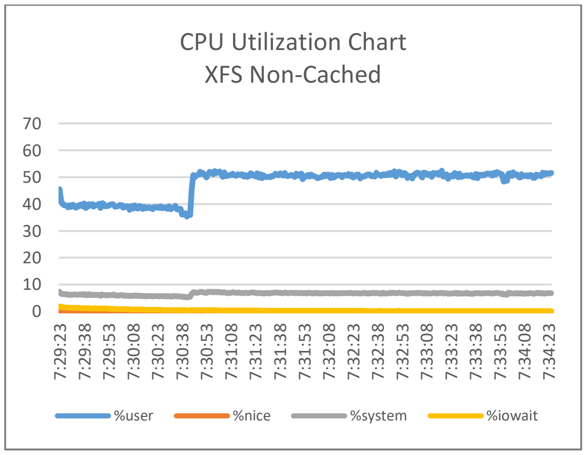
4. IO & CPU STATISTICS ANALYSIS

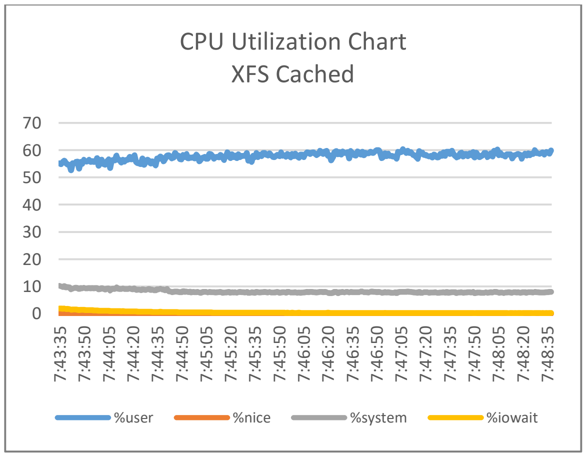
In this section, we dive into IO & CPU statistics gathered from both workload benchmark runs for which we have discussed the performance numbers in the previous section. This helps us understand the inner working of DM-WriteCache feature which would further help us in identifying other workloads where the feature can provide a performance benefit.

To begin, we examine the CPU utilization rates recorded from both runs using the following two graphs:

We can make two distinct observations. (1) There is roughly 10% higher CPU utilization in the XFS configuration like we noted earlier, (2) for the first one third of the benchmark run on non-cached configuration, there is an anomaly where CPU is being underutilized.

Explanation of the anomaly lies in the REDO log IO pattern generated by the database instance's log writer master process and its slaves. The following two graphs were generated based on "pidstat –d" stats gathered from log writer master and its slave processes:

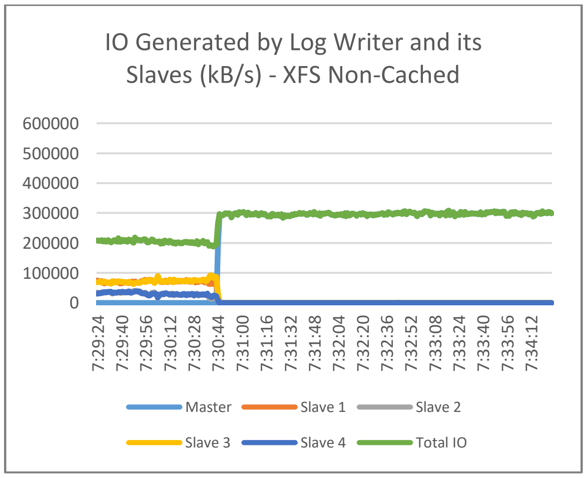


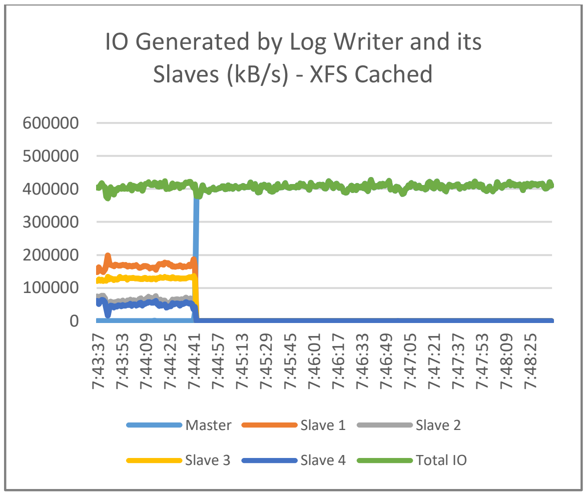


Figures 5 and 6: Comparison of CPU utilization rates between cached and non-cached configuration.

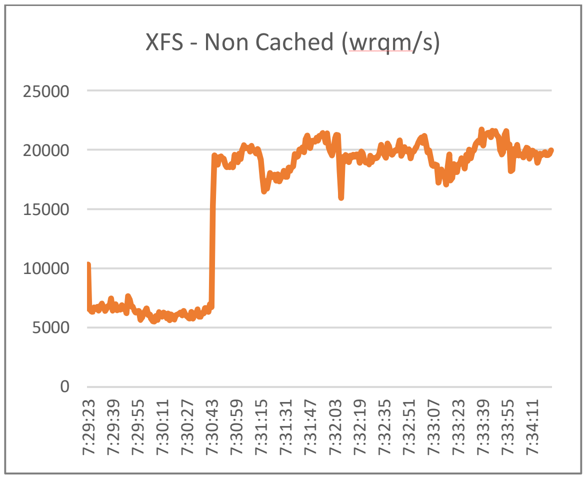
In an earlier section describing database configuration, we discussed the log writer configuration to be in ADAPTIVE mode which is the default. In both the benchmark runs, for the first 1/3rd of the execution period, we see that log writer slaves are actively flushing the log buffer to the REDO logs. For the remaining period, we see the database instance switch to a single process where the master does the flushing while slaves go dormant.

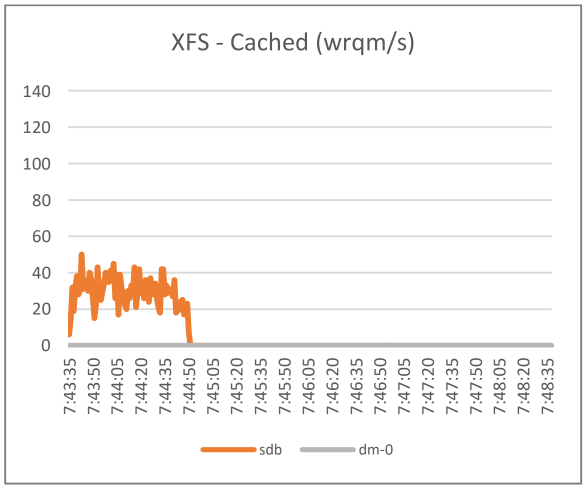
We can clearly see that multi-process sequential IO to a SSD device is not as efficient as a single process doing the IO. On the contrary, with the cached device, we see that there is no variation in throughput between single and multi-process environments. Beyond the current workload, we could benefit by using DM-WriteCache where multiple processes or threads are issuing IOs to a single SSD/HDD device.





Figures 7 and 8: IO rates to REDO logs between cached and non-cached configuration.





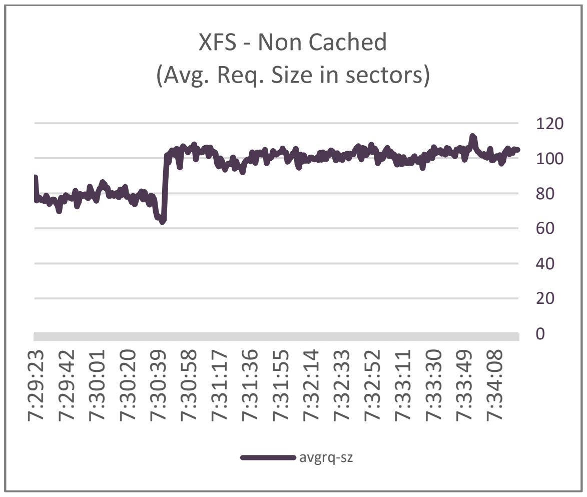
Figures 9 and 10: Count of write requests that get merged while in queue for cached and non-cached configuration.

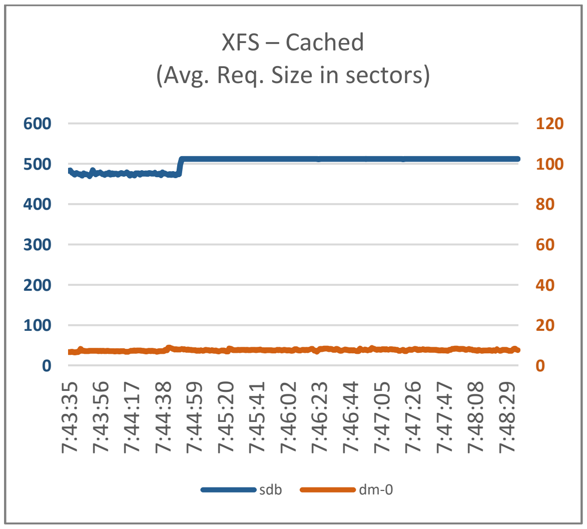
Now we look at the counts of write requests that get merged while in the IO queue (wrqm/s metric from "iostat –x" output) from both configurations:

We see that non-cached configuration heavily relies on the Linux IO stack's ability to identify and merge IO requests to achieve efficiency. While the single-process log write does it more efficiently (by reducing the number of IO requests issued to the backing SSD device) we see multiple-log-writer slaves suffer.

For the cached configuration, we see a zero to negligible count of IO requests being merged while in queue. This is true for both the cached device and then subsequently to the backend device when the data is flushed. This clearly shows that DM-WriteCache is much better at identifying IOs that can be merged and reducing the count of IOs issued to the backend device.

To substantiate the observation, we look at the average size of IO requests from both configurations. Average request size to the SSD device should be significantly higher in case of cached configuration as compared to that of non-cached configuration.





Figures 11 and 12: Average IO request size comparison between cached and non-cached configuration.

The graphs above clearly show that the average request size to the SSD device is about 5x more sectors with cached configuration as compared to that of a non-cached one. While we see an average of around 100 sectors per write request in non-cached configuration, we see 500 sectors per write request in the cached one.

Looking at the average-writes/sec metric from the cached configuration shows small bursts of IOs received by the cache getting converted to a few large IOs to the backing SSD device helps us conclude how the efficiency is realized.

Around 100k-120k of IOs with small request sizes received by the cache are fused into a few large IOs with counts in the range of 1.6k-1.8k before being flushed to the backend SSD device.

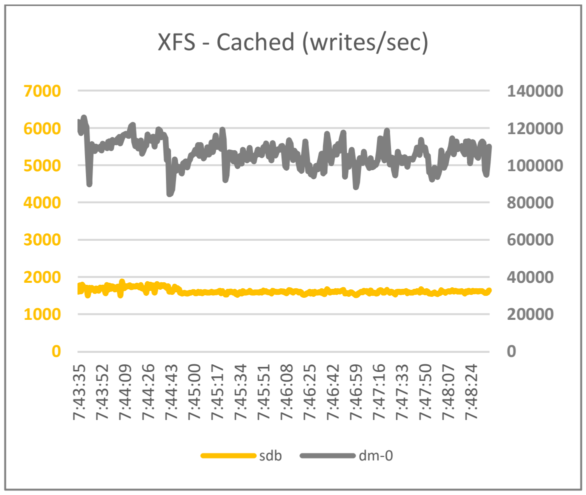


Figure 13: Average write requests per second to the cached device and its backing SSD device.

Before we conclude this section, let's take a look at the cache usage statistics. We can query the cache usage statistics with "dmsetup status" command as follows:

```
# dmsetup status /dev/mapper/dm_log
0 1562758832 writecache 0 8224895 7402836 0
#
```

The last three values are of interest in the above command's output. The last value represents the number of IO jobs in progress that are flushing data to the backend device. The second to last value represents available free cache entries and third from the last value represents the total count of cache entries in the configuration. Since we did not have the low_watermark threshold implemented in the feature yet, we see the cache utilization hover around the 10% mark:

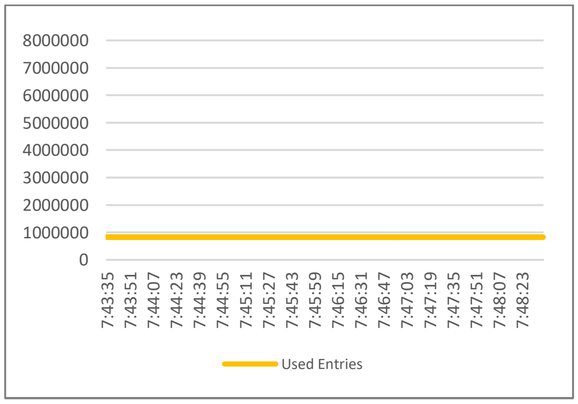


Figure 14: Count of used cache entries.

At its peak consumption, we see a 10.05% cache utilization which is a mere 0.05% above the high_watermark threshold. As for the count of active jobs trying to flush the data to backend, the pattern was as follows:

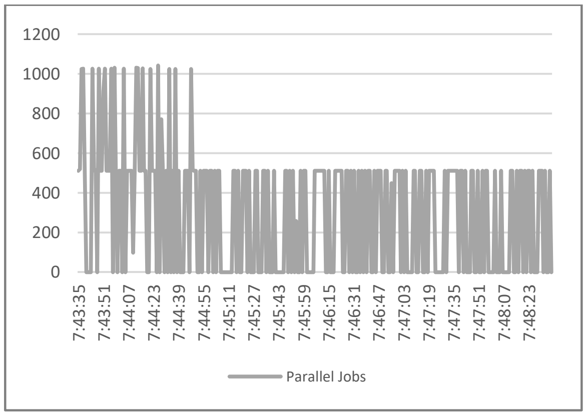


Figure 15: Count of jobs that are actively flushing cached data to backing SSD device.

The average REDO IO throughput generated against cached configuration is around 450MB/sec and as per the product specifications of the backend SSD device, it is capable of sustaining 1075MB/sec. With abundant free cache entries and the backend devices ability to sustain higher throughputs, the configuration is capable of sustaining workloads at a much larger scale than described in this paper.

5. OTHER CONFIGURATIONS

We evaluated an enhanced version of the DM-WriteCache feature to validate non-performance aspects to gauge how production-capable the feature was. The key differences to note are, (1) a low_watermark threshold implementation, (2) a switch to a 4.12 kernel release which is now mainstream with some of the commercial Linux distributions, (3) locking onto single process log_write configuration as it performs best in non-cached configuration, and (4) Multiplexing of REDO logs to meet durability requirements. We will go over observations related to these in subsequent subsections.

5.1 DURABILITY

High availability of REDO logs is one of the key factors to meet the durability requirements of a database deployment. We could either use a RAID device to host REDO logs or use Oracle's REDO log multiplexing feature to meet these availability requirements.

With current generation Persistent Memory offerings, there is no hardware/firmware level support for implementing RAID levels. Software-based RAID solutions like MD or LVM can be used, but these solutions lack support for DAX which in turn is a requirement for DM-WriteCache feature to be performant with Persistent Memory. This leaves REDO log multiplexing as the only option to meet the availability requirement.

We evaluated a configuration where each of the three REDO groups had two log members each. Each of these log members were hosted on two different cached devices. Persistent Memory devices used to cache the SSDs were from two DIMM slots that belonged to two different CPU sockets. Both these Persistent Memory devices were non-interleaved (i.e. each NVDIMM-N is presented as one Persistent Memory device to the OS) and were of 8GB in capacity. The cached devices were created as follows:

```
# dmsetup create logvol_1 --table "0 1562758832 writecache p /dev/sdb /dev/pmem0 4096 4 high_watermark 6 low_watermark 3"
# dmsetup create logvol_2 --table "0 1562758832 writecache p /dev/sdf /dev/pmem4 4096 4 high_watermark 6 low_watermark 3"
```

Key things to note in the above are the drop from a 32GB cache to 8GB and the changed high/low watermark thresholds.

From the benchmark runs, we see similar deltas with average wait per "log file sync" event. For this iteration of workload runs, the database was configured to use single log_writer process, so we see non-cached configuration perform much better than the previous iteration. Across multiple runs, we noted a difference of 14% with TPM metric between both configurations. With a run-to-run variance of 2-3%, TPM scores from previously discussed configuration and current configuration for cached device workload runs are nearly equal. These results clearly prove that REDO multiplexing to meet availability requirements does not have a significant impact on performance.

5.2 LOW_WATERMARK THRESHOLD

With low_watermark implemented and configured, we see that used cache entries drop to the set 3% mark once it hits the high_watermark threshold set at 6%.

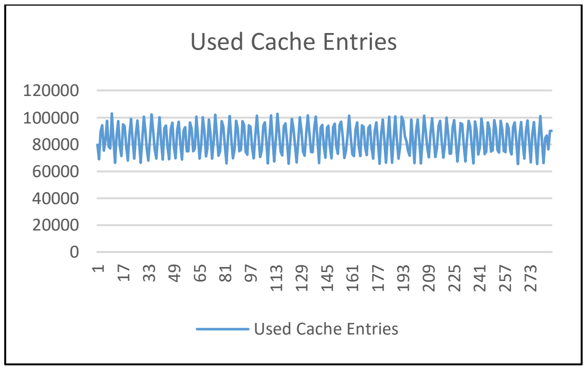


Figure 16: Pattern of used cache entries when both high and low watermarks are configured.

With larger chunks of data to be flushed, the expectation was to have larger IO request sizes being issued to the backend SSD device. However, we see a slight drop in average request size to around 450 sectors per write as compared to previous configuration's average of 500 sectors per write request.

5.3 DM-WriteCache with ASM

We compared workload run results between two ASM configurations where one configuration consumed SSDs as ASM Disks while the other consumed DM-WriteCache'ed devices as ASM disks. Some of the noteworthy observations are as follows:

(1) Between XFS non-cached vs. ASM non-cached, we saw ASM configuration outperform its XFS peer by about 8%.
(2) ASM cached vs. ASM non-cached, we noted a performance delta of about 7% i.e. TPM gains were mere 7% higher in cached configuration compared to that of non-cached one.

The key takeaway from this subsection is that the DM-WriteCache can push a database/application with data hosted on a file system to be nearly as performant as when their data is stored on raw devices. For deployments demanding the flexibility of file systems along with the performance of raw devices, DM-WriteCache is an indispensable feature.

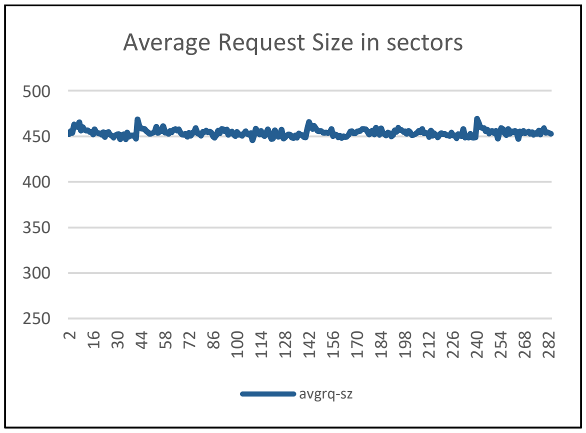


Figure 17: Average request size to the backend SSD device when both high and low watermarks are configured.

6 DM-WriteCache for DATA files

An Oracle database deployment where IO latencies and throughput play a critical role is a situation where the instance runs out of free buffers and is actively flushing dirty buffers back to the disk. One way to address this situation is to increase the amount of memory to be used for buffer cache.

Each server offers limited number of DIMM slots that have to be shared between regular DIMMs and NVDIMM-Ns. We evaluated a database workload starving for free buffers by swapping out a part of the SGA with an equal amount of Persistent Memory to be used for caching the SSD device where the data files are hosted.

Across multiple runs with varying cache sizes ranging from 8GB to 64GB, we observed that cached configurations underperformed against the non-cached configuration. For a given workload strength, a non-cached configuration with 192GB SGA outperformed a cached configuration of 128GB SGA + 64 GB DM-WriteCache by 10% for TPM metric. Comparing the average wait per "free buffer waits" event, we see 110% increase in average waits with cached configuration. Results from our tests suggest that DM-WriteCache for devices hosting data files will be beneficial only when SGA is adequately sized.

7 CONCLUSION

Existing applications lack native support to take full advantage of low latency and high bandwidth capabilities of Persistent Memory devices that are offered today. To facilitate the adoption of these Persistent Memory devices for existing workloads before full-fledged support gets into applications, there is a need for intermediate solutions. The DM-WriteCache is one such solution which enables consumption of these superfast devices in a cost effective way without requiring application changes.

Results from our experiments show that any workload that is latency sensitive and does IOs in bursts can gain a significant performance improvement by adopting DM-WriteCache feature using any form of Persistent Memory currently available.

For OLTP workloads, we can realize a performance gain anywhere between 7% and 22% using DM-WriteCache for devices hosting transaction logs. Our experiments reveal that a mere 8GB cache is sufficient to accelerate the performance of an OLTP workload against a database scale of 1TB. An 8GB NVDIMM-N costs significantly less as compared to an enterprise class NVMe write-intensive drive. Most of the enterprise database products follow CPU core based licensing schemes and each such license could cost thousands of dollars. Use of DM-WriteCache with Persistent Memory can significantly boost the throughputs by driving higher CPU utilization rates while reducing overall $/transaction cost.

With substantiations presented in this paper, the first version of DM-WriteCache as a feature got accepted to 4.17 RC1 kernel. Support for Persistent Memory has been available since the 4.5 kernel version. We expect that any commercial distribution adopting a 4.18 or later kernel revision will have the capability to support the DM-WriteCache feature for production deployments.

8 ACKNOWLEDGEMENTS

All the hardware used for DM-WriteCache feature evaluation and qualification was sponsored by Server Performance Engineering lab at Hewlett Packard Enterprise.

Would like to thank Varadarajan Sahasranamam, Srinivasan, Klaus Lange, Tom Vaden, Lou Gagliardi, Karen Dorhamer, and Paul Cao for their guidance and extensive reviews of the document. Special thanks to Steven Gary for the hardware provisioning and support. Thanks to Ronald Smith & P K Unnikrishnan for encouraging and supporting the effort.

9 REFERENCES

[1] Storage Networking Industry Association - Persistent Memory https://www.snia.org/PM
[2] Persistent Memory Overview https://docs.pmem.io/getting-started-guide/introduction
[3] HPE Persistent Memory ttps://www.hpe.com/in/en/servers/persistent-memory.html
[4] Dell EMC NVDIMM-N Persistent Memory User Guide https://topics-cdn.dell.com/pdf/poweredge-r740_users-guide3_en-us.pdf
[5] Intel® Optane™ DC Persistent Memory https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html
[6] HPE Scalable Persistent Memory https://supprt.hpe.com/hpsc/doc/public/display?docId=emr_na-a00038934en_us&docLocale=en_US
[7] DM-WriteCache https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/writecache.txt