Unexpected Performance of Intel® Optane™ DC Persistent Memory

Tony Mason, *Student Member, IEEE,* Thaleia Dimitra Doudali, *Student Member, IEEE*, Margo Seltzer, and Ada Gavrilovska, *Member, IEEE*

Abstract—We evaluated Intel® Optane™ DC Persistent Memory and found that Intel's persistent memory is highly sensitive to data locality, size, and access patterns, which becomes clearer by optimizing both virtual memory page size and data layout for locality. Using the *Polybench* high-performance computing benchmark suite and controlling for mapped page size, we evaluate PMEM performance relative to DRAM. In particular, the Linux persistent memory (PMEM) support automatically maps persistent memory in large pages. The effect of large pages for PMEM and small pages for DRAM is enormous, dwarfing other effects discussed in the literature. We found PMEM performance comparable to DRAM performance for a number of tests with significant performance improvements when optimized for data locality.

Index Terms—Persistent Memory

1 Introduction

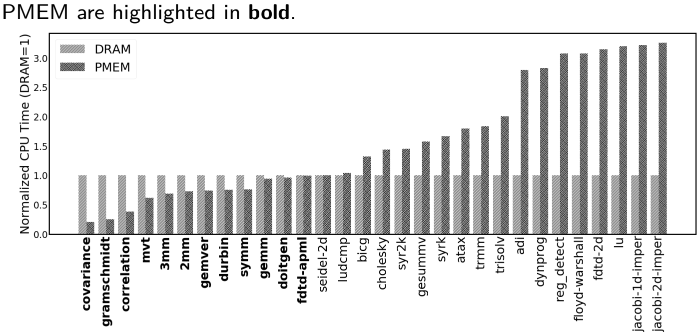
NON-VOLATILE memory research began decades ago [18]; recently it has accelerated with the April 2019 release of Intel Optane™ DC Persistent Memory (PMEM). Prior to availability, most work was done via emulation [3], [4], [11], [17], [18]. We expected PMEM to exhibit lower bandwidth and higher latency compared to DRAM; work with PMEM is consistent with these expectations [5], [9]. Yet we also note the actual performance behavior is more complex [13].

We evaluated PMEM using *Polybench*, a suite of well-known microbenchmarks used to evaluate memory locality optimization. Our initial results, shown in Figure 1, defied our prior emulation study [3]. We investigated this behavior to better understand the performance profile of PMEM. We confirmed that locality is critical and identified multiple ways in which locality manifests itself in current PMEM systems. We confirmed our observations on multiple different systems. We report our results on the last system that we evaluated.

We observe that the most significant effect of locality is related to default memory management policy, with secondary effects from the PMEM memory controller's use of striping, read-ahead, and write-behind caching. The Linux default memory management policy is to use the largest possible pages for PMEM, while it defaults to 4KB pages for DRAM. DAX-aware file systems (for application sharing, dynamic allocation, and security) interact with this policy as well. Because we observed the impact of this could be up to 5x, those publishing evaluations of PMEM should describe how they controlled for page size. PMEM striping primarily benefits workloads with good data locality (within the stripe size) and high bandwidth requirements. PMEM without striping often performs similarly when high bandwidth is less critical. PMEM caching in the memory controller and memory modules is more beneficial for CPUs with smaller caches. Once we controlled for page size and applied memory locality optimizing tools, we found performance for PMEM much closer to expectations. We achieved this performance from careful optimization, *not* by using default configurations.

T. Mason and M. Seltzer are with the Department of Computer Science, University of British Columbia, Vancouver, BC Canada e-mail: fsgeek@cs.ubc.ca, mseltzer@cs.ubc.ca.
T. Mason, T. Doudali, and A. Gavrilovska are with the College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail: fsgeek@gatech.edu, thdoudali@gatech.edu, ada@cc.gatech.edu.
Manuscript received August 13, 2019; revised April 7, 2020.

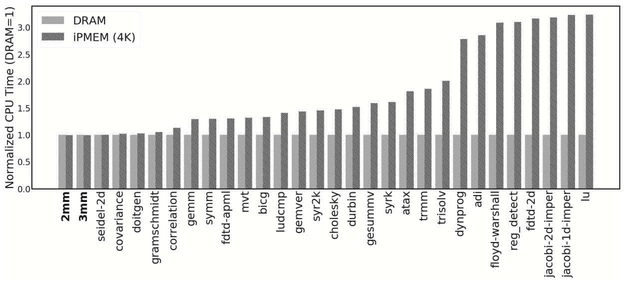
Fig. 1. Polybench: Performance of Striped PMEM relative to DRAM, Linux 5.0 kernel. Surprising results showing better performance on PMEM are highlighted in **bold**.



2 Background

PMEM's AppDirect mode permits two types of direct-access (DAX) usage: *devdax* access, which provides raw PMEM that is memory mapped by an application for exclusive use. Devdax mode requires pre-selecting memory allocation units (4KB, 2MB, and 1GB), static partitioning of the overall memory, using the "ndctl" utility, and is restricted to privileged applications; and *fsdax* access, which uses a DAX-aware file system to provide support for sharing between multiple applications, dynamic memory allocation — including page alignment and allocation unit size, and multi-user security. When an application memory maps a file on a DAX-aware file system the operating system configures the process page

Fig. 2. Polybench: DRAM vs Interleaved PMEM (both 4K pages). Results closer to expectations but still better than expected. Caching masks lower bandwidth of PMEM; memory usage optimization improves cache benefits which improves performance and decreases benefit of PMEM interleaving.



tables to directly reference the physical addresses of the underlying PMEM. Preferred practice is to use a DAX-aware file system [16].

The Persistent Memory Development Kit (PMDK) includes libraries for transparently converting standard memory allocation calls (e.g., *malloc*) to a memory allocator using arenas backed directly by files on a DAX filesystem device. This is similar to the approach used by *hugetlbfs*, a Linux file sytem that exposes DRAM to applications for allocation against large pages — the standard malloc calls in that case are implemented using arenas backed directly by files on hugetlbfs. We use both of these mechanisms in our evaluation of the Polybench tests.

Our PMEM hardware uses two dedicated memory controllers per CPU; each memory controller has three channels; each channel can manage two PMEM modules, which are equivalent to DRAM DIMM packages. The memory controllers include a small cache memory and provide the ability to transparently stripe across the PMEM modules although non-interleaved access is also supported. Each PMEM module also contains memory for converting 64 byte cache line sized load/store operations into 256 byte PMEM block size load/store operations.

The Linux operating system includes native support for PMEM. However, the policies for PMEM differ from the ones for DRAM. The Linux kernel transparently uses large memory page mappings for PMEM whenever possible, as dictated by the alignment and length of the page mappings. Linux version 5.3 uses 1GB, 2MB, or 4KB page sizes for PMEM when setting up page tables for running processes. Earlier Linux versions only used 2MB or 4KB pages for PMEM. The default for DRAM remains 4KB. We note that large page performance impact is well-described in the literature [1], [12]. While Linux behavior can be tuned, because default page sizes differ for PMEM and DRAM, it can create significant performance differences — we observed up to 5x difference.

We evaluated three different DAX-aware file systems: ext and xfs, which are standard Linux file systems incorporating experimental DAX support, and NOVA. The NOVA file system was the first purpose-built file system for PMEM [19]. However, we found NOVA unsuitable for use as a DAX file system, because its storage allocation policy does not preserve the 2MB page alignment the OS requires to provide large page support; this is not reported in even recent literature [9], [20]. Initially, we had not expected our choice of DAX-aware file systems to be relevant, as once memory mapped, the PMEM is directly accessed by the application. As a result of our discovery, our evaluation in this paper uses ext4, which did not exhibit the behavior observed with NOVA.

The Polybench test suite is a set of 30 computational kernels drawn from several different domains and is commonly used for evaluating the impact of memory performance with respect to those tasks. We previously used Polybench [3], and Polybench continues to be actively used for evaluating memory optimizer performance.

3 Evaluation

Our evaluation system is a dual socket NUMA architecture system, using two Intel® second generation Xeon® Scalable processors (codename *Cascade Lake*) running between 1.0GHz and 3.9GHz (variable) with 32KB L1 instruction and data caches, 1MB L2 caches, and a 55MB L3 cache, 20 cores per processor, and 12 memory slots per processor, with six 32GB DRAM modules and six 256GB PMEM modules, all running at 2666 Memory Transactions per second (MT/s). Each processor has two persistent memory controllers, with three channels per controller. Each channel manages two PMEM modules; Intel refers to this as the 2-2-2 configuration [8]. We found similar results on the same base system with different model Intel CPUs: 1-3.7GHz (variable), 32KB L1, 1MB L2, 32MB L3 caches.

We used Fedora 31 with Linux kernel 5.0.0, which includes native PMEM support. We configured the PMEM as six-way interleaved memory for one processor (iPMEM), and six single non-interleaved memories for the other processor (PMEM). We used the SNIA standard programming model, which is a DAX-aware file system providing dynamic PMEM management and security; while it is possible to use devdax mode, which provides raw PMEM access, it requires static allocation and privileged (“root”) access. We also used the PMDK libraries for transparently converting standard malloc calls into corresponding mmap calls for direct access PMEM, which only works with a DAX-aware file system [15].

We used Polybench/C Version 3.2 [14], [21], [22] compiled with clang version 10.0.0, choosing compile time constants to report execution time and the Linux real-time scheduler, and using maximum (`-O3`) optimization. We tested both with (`--llvm -polly`) and without polyhedral optimization. We chose our dataset sizes to match our prior work [3]. We ran the single-threaded Polybench 3.2 tests serially, bound to a

Fig. 3. Polybench: Cases where PMEM (non-interleaved PMEM) faster than iPMEM (interleaved PMEM) — interleaving does *not* provide substantial benefit in these cases; surprising because we expect the 2MB page size to dominate.

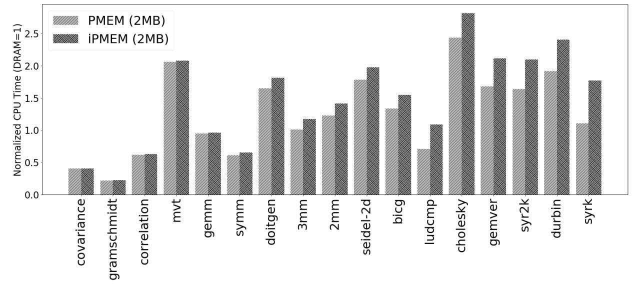


TABLE 1
Polybench Test Results: data size, by memory type (DRAM, PMEM, iPMEM), page size (4K, 2M), and optimization (`-O3` versus `-poly`). Execution time in seconds; sorted by 4K DRAM time improvement (optimized versus poly). Only shows test where polyhedral optimization benefitted at least one memory configuration (before rounding); other results omitted. Memory locality impact differs across memory types and configurations. CPU and memory allocation bound to one NUMA node. Ratio > 1 (pre-rounding) shows polyhedral optimization benefits.

Test	Data Size MB	DRAM						PMEM						iPMEM					
		4K			2M			4K			2M			4K			2M		
		Opt	Poly	Ratio	Opt	Poly	Ratio	Opt	Poly	Ratio	Opt	Poly	Ratio	Opt	Poly	Ratio	Opt	Poly	Ratio
gemm	91.6	29.3	**1.5**	19.1	14.1	**1.5**	9.3	39.0	**1.7**	22.6	32.0	**1.7**	18.7	37.8	**1.7**	22.9	27.5	**1.7**	16.6
3mm	120.2	17.1	**1.5**	11.8	12.1	**1.4**	8.4	16.6	**1.8**	9.5	11.9	**1.7**	6.8	17.1	**1.7**	10.3	11.7	**1.6**	7.2
2mm	85.8	5.4	**0.5**	11.3	3.8	**0.5**	7.7	5.2	**0.6**	8.6	4.1	**0.6**	6.9	5.4	**0.6**	9.7	3.9	**0.6**	6.9
correlation	137.4	105.1	**22.4**	4.7	23.2	**20.5**	1.1	123.8	**23.9**	5.2	45.5	**23.3**	2.0	118.8	**23.6**	5.0	39.9	**22.9**	1.7
covariance	49.5	19.7	**4.7**	4.2	3.3	4.4	0.8	20.3	**5.0**	4.1	4.4	4.9	0.9	20.0	**4.9**	4.1	4.1	4.8	0.9
mvt	1717.1	3.4	**1.0**	3.5	1.1	**1.0**	1.0	4.7	**2.0**	2.3	2.3	**2.0**	1.1	4.5	**1.9**	2.4	2.1	**1.9**	1.1
gemver	1717.4	3.7	**1.5**	2.4	1.2	1.7	0.8	7.9	**4.9**	1.6	5.3	**4.9**	1.1	5.2	**3.2**	1.6	2.7	3.2	0.9
jacobi-1d	152.6	27.6	**12.0**	2.3	26.0	**11.0**	2.4	261.8	**70.0**	3.7	260.0	**70.1**	3.7	89.0	**70.0**	1.3	88.6	**69.9**	1.3
doitgen	512.5	13.0	**6.2**	2.1	8.6	**6.1**	1.4	13.3	**10.2**	1.3	12.4	**9.8**	1.3	13.5	**7.2**	1.4	12.5	**7.1**	1.8
gramschmidt	91.6	67.0	**38.8**	1.7	11.3	**9.75**	1.2	73.1	**50.6**	1.4	24.5	28.7	0.9	70.5	**44.8**	1.6	16.6	19.1	0.9
fdtd-2d	1464.8	33.0	**19.3**	1.7	34.8	**17.0**	2.0	275.5	**57.4**	4.8	273.6	**56.9**	4.8	104.29	**52.2**	2.0	103.8	**52.5**	2.0
syrk	61.0	9.1	**5.4**	1.6	9.0	**5.4**	2.0	15.1	**6.0**	2.5	15.0	**5.9**	2.5	14.6	**6.1**	2.4	15.1	**6.1**	2.5
jacobi-2d	976.6	18.7	**12.1**	1.5	18.1	**9.0**	2.0	179.6	**62.3**	2.8	179.6	**62.1**	2.9	59.6	60.1	1.0	61.0	**60.0**	1.0
syr2k	91.6	17.9	**12.0**	1.5	18.0	**11.8**	1.5	31.0	**14.6**	2.1	30.0	**14.6**	2.1	26.0	**14.3**	2.8	25.9	**14.2**	2.8
symm	91.6	49.6	**34.2**	1.4	9.8	11.5	0.9	65.1	**44.5**	1.5	41.4	**38.9**	1.1	64.4	**44.0**	1.5	37.8	**23.5**	1.6
dynprog	245.4	11.3	**11.3**	1.0	11.5	11.5	1.0	71.4	73.0	1.0	74.7	75.3	1.0	31.4	31.5	1.0	32.0	**31.8**	1.0
durbin	1526.2	1.7	**1.7**	1.0	0.5	**0.5**	1.0	4.0	4.0	1.0	2.8	**2.8**	1.0	2.6	**2.6**	1.0	1.3	1.3	1.0
seidel-2d	190.7	25.1	**25.1**	1.0	24.5	**24.5**	1.0	31.3	**31.2**	1.0	31.0	31.1	1.0	25.1	**25.1**	1.0	25.1	**25.1**	1.0
trisolv	1716.8	0.2	0.2	1.0	0.2	**0.2**	1.0	0.3	**0.3**	1.0	0.3	**0.3**	1.0	0.3	0.3	1.0	0.3	**0.3**	1.0
cholesky	122.1	12.0	13.0	0.9	11.9	**10.8**	1.1	18.8	20.5	0.9	18.5	19.4	1.0	17.7	20.4	0.9	17.2	18.5	0.9
lu	122.1	17.0	25.2	0.7	16.0	28.4	0.6	209.8	**192.2**	1.1	219.2	**192.8**	1.1	55.1	78.2	0.7	54.4	80.5	0.7
reg_detect	381.6	30.1	52.2	0.6	28.7	51.0	0.6	331.2	**119.0**	2.8	327.7	**118.9**	2.8	93.3	118.6	0.8	92.6	118.4	0.8

single core, and all memory, including PMEM, was allocated from memory local to the NUMA node of that core. We used the PMDK allocator to redirect standard memory allocation calls to use memory mapped PMEM, via the DAX aware file system. The PMDK uses a jemalloc-based memory allocator.

Figure 1 reproduces our initial results, using the ext4 file system properly configured to use aligned 2MB pages. These initial results surprised us because they showed better performance than we expected. We determined that while the default behavior for Linux is to use 4KB TLB mappings for DRAM, it used 2MB TLB mappings for PMEM when possible. Indeed, this behavior of Linux has changed; as of Linux 5.3 it now uses the largest possible TLB mapping for PMEM between 1GB, 2MB, and 4KB, based upon the alignment and length of the memory region being accessed. **This behavior has not previously been reported in the literature, despite its substantial impact on performance.** We counted the number of TLB and last level cache misses when forcing different page sizes; it accounts for all of the performance difference. Recent work has alluded to the 2MB page impact, but does not explain why this occurs [10], while other recent work does not address it, despite the up to 5x impact we have observed on performance [2], [13]. We note that NOVA, which was designed for PMEM and is the *de facto* standard for PMEM-optimized file systems interacts poorly with the page-size needs of applications converted to use PMEM, because it mixes 4KB and 2MB page allocations internally. Over time this leads to fragmentation, which causes the performance degradation we observed. Thus, we subsequently switched to using ext4, configured to ensure 2MB aligned allocation. This did not exhibit the performance degradation.

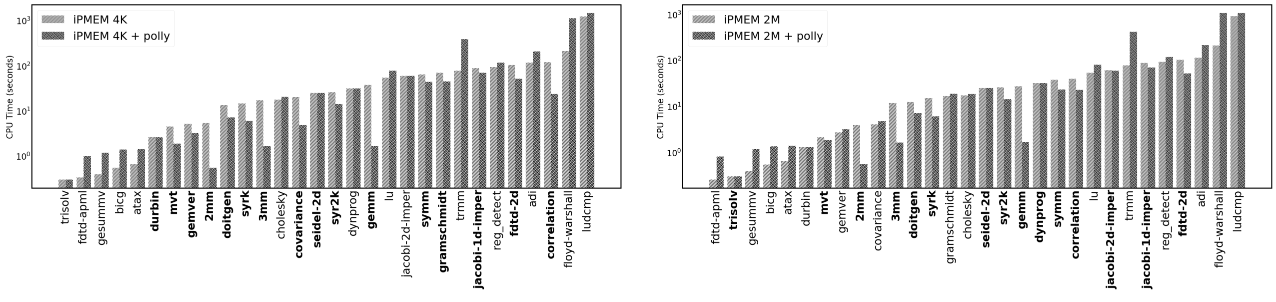
We used `perf`, a standard Linux performance utility, for measuring processor performance. From perf, we identified dramatically different data TLB miss rates, which in turn led us to finding both the default behavior of Linux and the sensitivity of page locality to the DAX file system allocation policy. **OS developers make fundamental decisions we must understand to achieve good performance.** Further, these choices do change over time.

We used `AEPWatch`, an open source Intel utility [7], for measuring the performance behavior of the persistent memory controllers. Once we controlled for page size, this provided us with greater insight into the impact of read caching, write combining, and striping in the persistent memory controllers. We found that **no single configuration produced consistently best performance.**

We hypothesized that software memory locality techniques would also yield improved performance. To test this, we used the LLVM compiler tools, which include a polyhedral memory optimizer. The memory optimizer optimizes memory layout and code generation to improve data locality; Polybench was constructed to evaluate the effectiveness of polyhedral optimization. In Table 1 we show our results across DRAM, (non-interleaved) PMEM, and (interleaved) iPMEM, for both 4KB and 2MB page sizes. Most PMEM configurations benefitted from polyhedral optimization, which reflects the sensitivity of the system to locality. We found that non-interleaved PMEM benefitted most from polyhedral optimization, often exhibiting comparable performance to interleaved PMEM. In only two cases did PMEM exhibit better performance than DRAM, likely due to the additional caching in the PMEM system itself. We omit results for eight tests where no memory configuration benefitted from polyhedral optimization.

Figure 2 was a fairer comparison of memory performance and a clear indication on the importance of locality. We evaluated the impact of polyhedral memory optimization to corroborate our theory on the importance of data locality. Figure 3 suggested that locality was a more important factor than memory striping. Figure 4 demonstrated that in many cases polyhedral memory optimization improved data locality by as much as 80%. Table 1 provides the specific timings for the 22 tests where polyhedral optimization improved performance for at least one memory type. However, we were unable to identify any single factor that explained these exact results: we considered both data and instruction TLB miss rates, last level cache misses, data set sizes, and instruction counts. We suspect this is due to the generic nature of the polyhedral optimizer, which has no specific knowledge of the processor or memory characteristics on our test system; it is also possible there are secondary features within the PMEM itself that, while not exposed, impact this.

Fig. 4. Polyhedral Memory optimizations for interleaved PMEM (iPMEM), 4K and 2M pages. Varying cache layers impact performance differently. Results sorted by execution time without polyhedral optimization. Boldface indicates improvement with poly optimization.



4 Conclusion

We found PMEM is more sensitive to memory locality than DRAM; this is due to differences in the hardware implementing these memory technologies. The decision to default PMEM to large pages provides a substantial boost in page locality. As datasets grow in size, using large pages yields better TLB locality because more actual PMEM can be described with a fixed number of TLB entries. PMEM's support for interleaving is most beneficial for data with both good locality and high bandwidth requirements; non-interleaved PMEM works comparably well to interleaved PMEM with good data locality and moderate bandwidth requirements, such as with CPUs employing larger caches.

We caution those evaluating PMEM to ensure the evaluation properly controls for page size because it's impact may lead to incorrect results. Those using PMEM should consider optimizing data locality as it can lead to substantial performance gains. However, those gains are clearly sensitive to the behavior of the underlying memory and thus are dependent upon the workload. Not optimizing risks sacrificing performance when using PMEM.

While our evaluation was done with the first generation of commercially available PMEM, we expect these insights will generalize to future implementations of PMEM, particularly those related to the impact of data locality, OS behavior, and caching/striping (interleaving) of PMEM.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback, as well as Vaastav Anand, Swati Goswami, Nodir Kodirov, Joel Nider, Ranjan Sarpangala Venkatesh, and Brian Veraa for their assistance in reviewing drafts. This research was partially funded under NSF award 1822972, US Dept. of Energy UNITY, Intel Corporation, and Exascale Computing Project SICM (Simplified Interface to Complex Memory).

References

[1] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 631–644.

[2] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 2019, pp. 478–493.

[3] T. D. Doudali and A. Gavrilovska, "Comerge: toward efficient data placement in shared heterogeneous memory systems," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2017, pp. 251–261.

[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 15.

[5] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," *arXiv preprint arXiv:1904.07162*, 2019.

[6] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[7] Intel, "Memory optimizer," 2019. [Online]. Available: https://github.com/intel/memory-optimizer

[8] ——, "Quick start guide: Provision intel optane dc persistent memory," https://tinyurl.com/wcznyhf, 9 2019.

[9] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[10] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 2019, pp. 494–508.

[11] Maciejewski, "Persistent memory programming: How to emulate persistent memory," December 2017, (accessed January 1, 2019). [Online]. Available: https://tinyurl.com/y53saf9j

[12] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 679–692.

[13] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable nvm," in *Symposium on Memory Systems (MEMSYS'19)*, 2019.

[14] L. Pouchet, "Polybench/c: the polyhedral benchmark suite@MISC," March 2017. [Online]. Available: https://tinyurl.com/y442hthg

[15] A. Rudoff, "Persistent memory development kit," 2018. [Online]. Available: http://pmem.io/pmdk

[16] ——, "Impact on application development: Snia nvm programming model in the real world," in *SNIA Persistent Memory Summit*. SNIA, 1 2019.

[17] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, "Quartz: A lightweight performance emulator for persistent memory software," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 37–49.

[18] M. Wu and W. Zwaenepoel, "envy: a non-volatile, main memory storage system," in *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5. ACM, 1994, pp. 86–97.

[19] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories." in *FAST*, 2016, pp. 323–338.

[20] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," *arXiv preprint arXiv:1908.03583*, 2019.

[21] T. Yuki, "Understanding polybench/c 3.2 kernels," in *International workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014, pp. 1–5.

[22] T. Yuki and L.-N. Pouchet, "Polybench 4.0," https://bit.ly/2LQbJQY, 2 2015.