Persistent Memory-Aware Scheduling for Serverless Workloads

Amit Samanta[1] Faraz Ahmed[2] Lianjie Cao[2] Ryan Stutsman[1] Puneet Sharma[2]
[1]University of Utah, USA [2]Hewlett Packard Labs, USA

***Abstract*—In the last few years, persistent memory (PM) has become widely commercially available. PM's direct load/store interface allows fine-grained storage access that fully bypasses all software I/O overheads. However, existing PM modules exhibit performance anomalies, particularly when they are concurrently accessed by many threads.**

PM is poised to change how all applications access storage, and in this paper we look at how PM might be best managed and used in serverless applications. We make the case that serverless presents a special opportunity for PM because serverless platforms can use the structure of serverless applications to gain more insight into their I/O behavior and because serverless platforms include specialized schedulers that can exploit the combined knowledge of application and PM behavior for managing concurrent workloads.

***Index Terms*—Serverless computing, persistent memory, scheduling, Intel Optane DCPMM.**

I. Introduction

Commodity persistent memory (PM) in the form of Intel's Optane DCPMM has recently transformed storage by providing a fast and direct load/store access (like DRAM in Table I) to persistent data. On its face, fine-grained random access to persistent structures with no software and no serialization/deserialization overheads could fundamentally change how many applications approach I/O. However, like all storage media, DCPMM has its idiosyncrasies that make it hard to simply use it as a large, persistent RAM. Specifically, DCPMM write bandwidth suffers substantially compared to its read bandwidth, and that mismatch widens when many concurrent threads write DCPMM simultaneously [1, 2, 3]. In fact, as we show in §II, total effective write bandwidth with DCPMM *decreases* with the number of concurrent writers. As a result of this, common advice is to limit concurrent writers to DCPMM to avoid write throughput bottlenecks and collapse [1]; however, this is difficult to do if the set of thread or applications to be run is not fixed in advance.

Instead, we argue that enforcing limits on concurrent DCPMM writers is a matter of better scheduling and should be handled by a scheduler rather than through fragile manual tuning. By performing admission control among processes that desire to write to DCPMM, it is possible to collectively improve job completion time (JCT) for all processes that use DCPMM. Such a scheduler can be implemented at different layers. For example, such scheduling can be done by a kernel CPU scheduler or in the user space by platform schedulers [4, 5, 6] for containerized and serverless workloads [7, 8, 9, 10]. A challenge is that such a scheduler would need to be aware of which tasks will perform DCPMM writes or reads, which tasks are sensitive to being scheduled while DCPMM is experiencing concurrent accesses, and the level of that sensitivity. In short, this means that such a scheduler needs some visibility into the structure of applications. Hence, in this paper, we propose a new PM-aware scheduler for serverless workloads, which expose sufficient structure for this type of profiling and optimization.

Memory Modules	Read Bw	Write Bw	Latency
DRAM	101 GB/s	79 GB/s	60-100ns
PM	37 GB/s	10.5 GB/s	300ns

Table I: DRAM & DCPMM performance characteristics.

Today's serverless platforms [11, 12, 13] do not expose PM to applications, so along with a motivation and design for a PM-aware serverless scheduler, we also include a first analysis of how serverless applications will interact with PM as it becomes available in the cloud. For our analysis, we ported a well-known serverless runtime, OpenFaas, to use PM and instrumented it to collect information about applications' PM use. First, to motivate our PM-aware scheduler and its design, we run microbenchmarks that show the effect of concurrent DCPMM writers and the resulting poor performance. We also demonstrate that constraining the number of concurrent writers to DCPMM improves average JCT for a synthetic workload. From those microbenchmarks, we outline a framework and design for a realistic scheduler that can optimize JCT for serverless workloads [7] when some set of the functions access DCPMM. Such a scheduler must overcome several challenges including: (1) extracting a model that determines when real workloads will read/write DCPMM, (2) profiling how sensitive functions are to interference from concurrent DCPMM access, (3) combining these models into a (non-work-conserving) scheduler that still results in overall efficiency and JCT improvements.

In summary, this paper makes the following contributions:

1) First, we demonstrate that concurrent writers in DCPMM hurts performance, and then we show that using DCPMM-aware CPU schedule improves the average JCT and throughput.
2) Second, we outline the important challenges for an intelligent DCPMM-aware general thread scheduler that is practically implementable and optimizes JCT.

3) Finally, we outline a design for similar DCPMM-aware scheduler for serverless workloads.

II. Motivation

DCPMM write bandwidth collapses under concurrent writes [1, 2, 14]; this effect is easy to replicate. In Figure 1, we run an experiment where threads concurrently read or write to DCPMM. We vary the number of concurrent threads, and we plot the aggregate read/write bandwidth of the threads. Concurrent readers scale until the DCPMM bandwidth saturates as expected. However, with concurrent writers the effective bandwidth does not scale; in fact, it regresses.

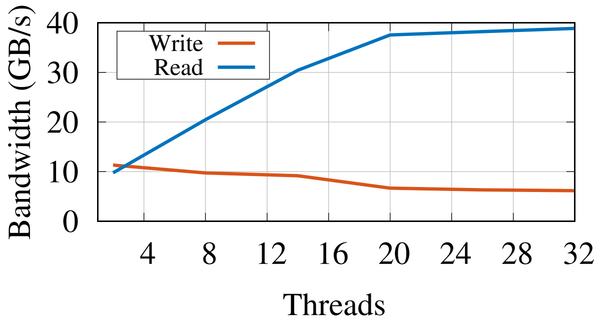

Figure 1: DCPMM bandwidth vs. number of threads. Effective BW under concurrency with interleaved Optane DCPMM.

The experiment machine has 32 cores, so this drop isn't a result of contention for CPU time; instead, it occurs because each DCPMM module's XP Buffer combines written back cache lines in 256 B XP lines when written sequentially. Under concurrent writes, XP lines are evicted from the XP Buffer before they are completely filled, creating write amplification and prematurely saturating DCPMM write bandwidth. Due to this, DCPMM effective write bandwidth drops to around 60% of its peak under many concurrent writers. Here, these 32 threads would be able to complete their work more quickly (by nearly 2×) if they were run in serial.

Figure 2 shows the results of exploiting this effect. Like before, this experiment is run with multiple threads concurrently reading and writing 4 GB each to DCPMM for an increasing number of threads. Each thread repeatedly runs tasks where the tasks write DCPMM (Only Write) or read/write DCPMM (Read+Write). The y-axis shows the average job completion time (JCT) of the tasks. Since the DCPMM write bandwidth slows with increasing thread counts, JCTs increase substantially as the count of concurrent write threads/tasks increases. When tasks spend some of their time reading DCPMM, things improve somewhat, but tasks are still suffer under this write bandwidth bottleneck. In other words, when the workload consists of both read and write tasks the average JCT drops; however, the write bandwidth bottleneck still worsens the JCT as thread concurrency increases.

The controlled schedule (green line) shows the average JCT with a simplistic scheduler that we developed that restricts the number of tasks that are allowed to concurrently write the DCPMM at a time. This leads to a 2× improvement in average JCT, showing that smarter scheduling can improve performance.

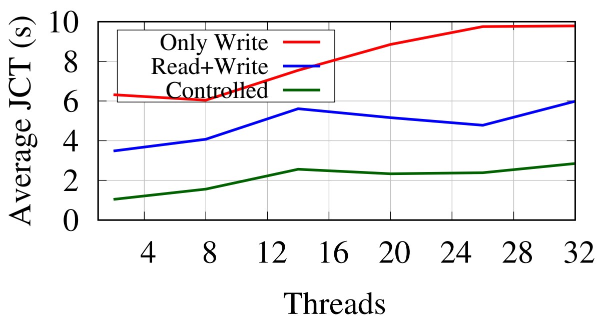

Figure 2: Average job completion time (JCT) for a set of tasks with an unrestricted (Only Write, Read+Write) and restricted (Controlled) number of concurrent writers.

However, our demonstration is contrived. The scheduler is omniscient. It knows precisely which tasks read and write DCPMM, when they read/write it, and how much. The tasks are all uniform as well; for example, it doesn't make sense to limit concurrency if tasks run code that are CPU-bound. It also doesn't make sense to limit concurrency even when tasks write DCPMM if their JCT isn't sensitive to the bottleneck. A realistic PM-aware scheduler would need to make these decisions while adding little overhead to collect the information it needs and in selecting tasks/threads to run.

III. Why Serverless Workloads?

Though these performance issues apply to many workloads [2, 15], serverless workloads create a unique and special opportunity to address them. Serverless applications are composed of individual functions that we can understand individually (rather than trying to infer "phases" of opaque programs or their internals). Similarly, by watching throughput and response time, we can tell we are improving application-level performance, something that is much harder to infer from opaque applications.

Subsequently, recent work [16] has pointed out that the serverless workloads will have improved performance by using faster and higher bandwidth persistent storage (i.e., DCPMM). Similarly, Katsakioris [17] *et al.* proposed a snapshotting approach to avoid the cold start latency of serverless functions by exploiting DCPMM. It brings cold start performance close to warm for serverless workloads. The study also shows that DCPMM is beneficial for serverless as the direct storage access eliminates the data movement to DRAM, providing 2× performance benefit.

Herein we focus mainly on serverless workloads, but for general applications and workloads, it is possible to take a more general approach. For example, it may be possible to do similar scheduling as part of an OS scheduler or by building PM-contention-aware scheduling on ghOSt [18], which is a user-level scheduler framework. However, building such a general scheduler precludes taking advantage of the extra structure of serverless workloads; later, we plan to extend our approach for general applications.

IV. Challenges in PM-Aware Scheduling

Though DCPMM creates a new opportunity for scheduling to improve application-level performance, there are several challenges in doing so for serverless applications. Those challenges include maintaining high throughput with low write-amplification, inexpensive application profiling, and ensuring application fairness. In the next section, we outline a design that addresses these issues that maximizes application performance while ensuring fairness without interposing on application PM accesses.

Challenge 1: High Throughput with Low Write-Amplification. Some serverless functions will aggressively mutate stored state; as Section II explains running such functions concurrently will create write amplification which, in turn, limits both read and write throughput. A smart scheduler needs to understand which functions access and mutate state, and it must limit concurrent writers to maintain high throughput and low write amplification. This will also minimize wear on DCPMM modules. One key challenge is that simply serializing access to DCPMM modules doesn't always result in the best performance either, so an efficient scheduler should be calibrated to understand precisely when threads should be co-scheduled and when they should be serialized in order to optimize overall JCT.

Another way to eliminate bottlenecks due to concurrent writers is to partition the state that functions operate over across DCPMM modules such that no two concurrently scheduled threads write to the same module at the same time. Ideally, the serverless platform should control both thread scheduling and data placement to minimize interference.

Challenge 2: Low-Overhead Profiling. PM performance is highly workload dependent, but applications are mostly opaque. To schedule workloads that use PM effectively, the scheduler must understand how applications interact with storage. Serverless helps with this because it breaks applications into functions that represent some common functionality that can be profiled individually. This breaks the phases of monolithic applications down into more predictable chunks that can be profiled and scheduled individually.

A related challenge with PM is that, unlike workloads that use traditional I/O interfaces, no software component interposes on each PM storage access. To be effective, the scheduler must be able to efficiently extract application access patterns and adapt when workloads change without interposing on every PM access.

Challenge 3: Fairness. Maximizing job throughput is an important goal, but read bandwidth is greater than write bandwidth on PM. In optimizing overall performance, the scheduler must ensure fairness so that, for example, read-intensive applications don't starve write-intensive applications. With PM, this is more difficult than it is with traditional I/O interfaces for two reasons. First, Linux's CFQ block I/O scheduler helps ensure fairness between applications with traditional I/O accesses, but PM has no such scheduler since the kernel cannot (efficiently) interpose on PM accesses. Second, concurrent writers with other writers or readers cause slowdowns that must be accounted for. Ideally, a PM-aware scheduler should typically give all applications better performance than they would receive if they are run concurrently rather than just maximizing system throughput.

Even ignoring the challenge of interposing on I/O accesses, such a scheduler cannot use the same approach as Linux's CFQ to enforce fairness, since CFQ results in scheduling a single process's I/Os at a time (according to a processes weighted fair share). While such a schedule might minimize cross-application interference, it fails to exploit the high I/O parallelism that DCPMM provides, especially when the number of writers is limited.

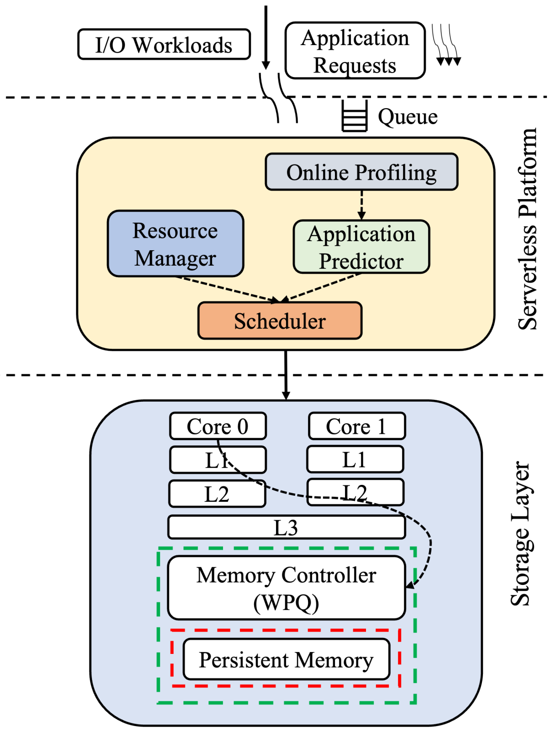

Figure 3: High-level architectural view of PM-aware scheduler for serverless workloads/applications.

V. PM-Aware Serverless Scheduling

To address these challenges, we are developing a new serverless platform built on top of OpenFaas [19]. It runs standard serverless applications which are composed out of chained functions triggered by events such as incoming HTTP requests. The functions for each triggered chain are scheduled on worker nodes by a centralized scheduler. In our platform, each serverless worker node is equipped with PM, giving functions access to a few terabytes of local DCPMM.

We assume that any function can be run on any worker node. PM between the worker nodes is neither shared nor coherent; if an invocation of a function stores persistent state on PM, it may be the case that future invocations of that function don't see the persisted data since they may be mapped to other workers. This is typical for serverless functions, which assume they must coordinate through shared storage in order to bootstrap their state or to forward output to one another. Approaches that provide shared memory semantics between invocations are interesting, but today they all require software interposition on storage accesses which negates direct load/store performance benefit of PM. Although, CXL and other emerging coherent interconnects that can support remote shared memory through a load/store interface could change that [20].

Figure 3 shows the overall design of the platform with the three new components we add to OpenFaas for PM-aware scheduling. The *Online Profiler* ① profiles serverless applications as they run to determine how each function of each application interacts with PM. The *Per-Function I/O Prediction Model* ② predicts the PM I/O demands and patterns of functions that enter into the scheduling queue based on their past behavior. The *PM Performance Model* ③ is constructed using microbenchmarks similar to Section II; it predicts how given PM I/O patterns will affect one another if run concurrently. The *PM-aware Scheduler* ④ consults the I/O prediction model, the I/O performance model, and per-application resource consumption statistics from the resource manager to place and schedule function invocations among the set of available worker nodes in the cluster, attempting to maximize throughput while respecting fairness. Note, this does not imply minimizing interference, since serializing execution per worker generally does not maximize throughput. Here, we discuss how this PM-aware Scheduler addresses the challenges laid out in §IV.

High Throughput with Low Write-Amplification. The Per-Function Model informs the PM-aware scheduler about how aggressively each function reads and writes state along with details on access granularity and information on how much the function is influenced by slowdowns to PM access (this depends, for example, on how many data dependencies there are between a function's PM accesses, which the model can infer from collected CPU performance counters). Combining this with the PM Performance Model, we plan for our scheduler to implement a *read-prefer-write-limit* policy. In the presence of functions that both read and write PM, the *read-prefer-write-limit* policy prioritizes read-intensive functions while concurrently scheduling write-intensive functions up to the point that overall throughput worsens. Additional writers would be delayed until the reads are completed.

We are also considering letting the scheduler decide which DCPMM DIMMs should be used to store state for specific applications or functions. In this model, the scheduler would try to avoid interference by partitioning state intelligently across the DIMMs. For example, if a function mainly read state and was generally CPU-bound, then a single DIMM might be sufficient to prevent bottlenecks in it. In this case, the scheduler could pack the function's state on one DIMM where it is expected that concurrent writes to be rare. Another possibility is to let the scheduler choose which DIMMs a function invocation writes to, letting the scheduler avoid interference by segregating reads/writes. The scheduler has flexibility since it need not guarantee that all invocations of a function see the results produced by prior invocations (this can already happen when invocations run on separate worker nodes). We plan to experiment with an indirection layer that tracks where data is written to PM, which lets the schedule control how writes are mapped to DIMMs.

Low-Overhead Profiling. Past behavior of functions help predict their future accesses, but monitoring PM accesses of functions is challenging since the I/O path isn't exposed to software. To solve this, our platform will use performance counters to track the amount of data read from and written to PM. By tracking the total memory accesses, we believe it should be possible to make inferences about typical access granularities. We plan to update the Per-Function I/O Prediction Model after each invocation of a function completes using performance counters, which effectively have no runtime overhead. We can also use performance counters that track CPU stalled cycles combined with counts of accesses to PM to score functions on how sensitive they are to PM latency as compared to PM bandwidth.

Fairness. Finally, since read-prefer-write-limit could lead to starvation, the schedulers policy will also employ multi-resource fair queuing [21, 22] (with CPU, PM read bandwidth, and PM write bandwidth resources separated out, accounting for the dependency between PM read/write bandwidth) with a form of aging that lets reads be prioritized over writes, but which bounds delay of writers by boosting priority for functions that remained enqueued too long.

VI. Related Work

The increase in data size, the complexity of serverless workloads, and heterogeneity of storage infrastructure has created new challenges for serverless platforms. The serverless paradigm provides fine-grained resource orchestration and scalability, but data transfer between serverless functions can become a scaling bottleneck. This is because parallel I/O operations result from running multiple functions concurrently.

To resolve this bottleneck, existing systems (such as Pocket [23], SONIC [24], Locus [25], and Jiffy [26]) store the data in virtual machines (VMs) for functions with few dependencies [24], and they use remote storage for functions with wide dependencies [23, 26]. Jiffy [26] is an elastic far-memory system for serverless analytics that meets the memory demand of serverless tasks. Pocket and Jiffy address the problem of resource scaling for stateful serverless workloads. Wukong [27] also addresses the problem of scheduling serverless functions to reduce data movement. However, none of these approaches consider the impact of the idiosyncrasies of the storage medium on application performance in their scheduler designs (e.g., read-prefer-write-limit policy in §V). Further, they do not consider fairness when running concurrent serverless jobs. Locus [25] is one such effort that uses fast and slow storage to deal with such bottlenecks while considering storage costs. However, there is no prior work (as per our knowledge) that explores the potential of PM (i.e., DCPMM) for serverless workloads. Further, no existing serverless platform supports PM for persistent storage along with other storage options. Thus, there is a scope for designing PM-aware scheduler for serverless workloads which improves efficiency and scales with growing data size while providing higher throughput and lower JCT.

OdinFS is a specialized file system designed for PM [28]. It is able to carefully "shape" accesses to PM modules (via delegation of writes to a small number of threads). However, it can only do this because it is not a DAX file system: all data access in OdinFS requires software interposition (as explicit read/write syscalls). A key idea in our approach is that we still support direct load-store access to PM data by CPU cores. This eliminates all software overhead for storage access, but it leaves us with the drawback that we can only control the scheduling of loads/stores at a coarser grain and only indirectly by controlling thread or function scheduling. How the software and CPU overheads of an approach like OdinFS compare with an indirect, DAX-like approach like ours is an interesting question that we plan to examine.

VII. Discussions

Use Cases. We target intermediate storage between functions and cases where instances of a function produce PM files where they might share concurrent access to memory-mapped state. Serverless applications often use key-value/object stores and message brokers to coordinate communication between functions. There has been a push toward shared-memory architectures for serverless function communication. Using shared memory significantly reduces the overheads associated with data access for serverless functions [29]. With shared memory, we can imagine a function that generates a B-tree index (if it doesn't exist) on PM that other invocations can access and even update — leading to a shared memory data structure. In our current design, we wouldn't guarantee invocations run where the B-tree is, but then if functions run on other machines, they could similarly construct a local index and do the same on a different worker.

Profiling Overhead. Our current plan for profiling is purely online/dynamic. Static profiling is possible (by distinguishing loads and stores), but things like function inputs could cause wildly different access patterns making it hard to infer the quantity of loads/stores statically. Instead, we try to limit the overhead of runtime profiling by only repeating heavy-weight profiling when our statistics seem to be inaccurate for a specific function. Once we have a handle on how much overhead profiling adds overall in realistic workloads, we'll be in a better position to understand how much static profiling might help.

Prediction Accuracy. The predictions for a specific function need to vary based on expected contention with concurrent writers. Our plan is that each function's performance model will be a function of the expected amount of data written during its scheduled interval, but when we update the model we'll do so only based on how much data was actually written (by other concurrent functions) when it ran. In some cases, we may desire to run a function with more or less write contention than we have seen in the past; in that case, our best bet is to just use the prediction with the closest amount of contention that we've seen in the past – initially these predictions won't be accurate, but if that scenario arises repeatedly and the function's behavior is stable it should converge quickly. We're considering simple models like exponentially weighted moving averages for tracking how much data functions read and write, but we're interested in seeing how more sophisticated models work as well (e.g. using reinforcement learning).

Persistence. PM can be used to recover from application-level faults since it is persistent with additional PM capacity over DRAM. PM data persists across host crashes, but this aspect of PM is unlikely to help the applications we target since on a host crash functions will need to be re-executed on another machine where the data isn't resident in PM. This isn't a drawback specific to our paper; PM generally doesn't help improve performance (over other durable media) when high availability is needed since remote replication is needed for availability and remote access costs dominate. Remote replication generally puts software back in the data access path, which would make our scheduling optimizations less important (techniques like those used in OdinFS can be used to mitigate contention if the software is on the access path). We have begun to explore how we can get OdinFS-like optimizations that shape PM accesses without adding software on the access path (by relying on hardware accelerators), but that work is beyond the scope of this paper. Finally, concurrent invocations of the same function can concurrently access and modify PM files, providing a benefit of shared memory within functions. We plan to extend this to also allow different functions and different tenants to share concurrent access to PM files, though we need to define an access control model before that is safe.

Resource Idleness. Resource idleness may arise on worker nodes if the PM resources are overallocated to applications. To handle the resource idleness problem, the number and size of PMs in each worker node need to be utilized appropriately. However, all the functions are scheduled through a centralized scheduler, which may introduce the performance bottleneck. We plan to mitigate this with our two-level scheduling approach: the scheduler assigns work to workers, which then perform admission control to avoid too many concurrent writers. This approach helps us avoid the resource idleness as a bottleneck up to some point since the effort to enforce scheduling is subdivided among workers.

VIII. Is Commercial PM Dead?

After just a few years on the commercial market, Intel recently announced the end of development and manufacture of these 3D XPoint-based Persistent Memory Modules [30].

This is a major setback in the roadmap to pervasive use of PM in systems, since Optane DCPMM was the first PM with a direct load/store interface that was widely commercially available, had a lower price per bit than DRAM (though it was unclear whether its *cost to produce* was fundamentally lower), and had access latencies that approached DRAM's. Nonetheless, there are several reasons why PM and PM research is still practical and important.

First, it took more than a decade of work to bring practical PM to modern CPU architectures, and all that hard work in refining interfaces and CPU architectures is now embedded and deployed most data center systems with today's architectures and operating systems. For example, today Intel [31], ARM [32], RISC-V [33], and POWER [34] CPUs all now ship with ISA-level support for interacting with PM safely; most major server operating system kernels include support for it as well. JEDEC has also recently standardized persistent main memories [35] that complement longer-standing approaches that combine DRAM and Flash for NVDIMM support.

Second, there are still commercial PM offerings [36, 37] (often pairing DRAM, NAND Flash, and supercapacitors), and there are more still on track for commercial release [38, 39], with new media that span from magnetoresistive RAM to carbon nanotube-based RAM.

Third, CXL/PCIExpress attached devices now emerging that implement the CXL.cache and CXL.mem interfaces which will allow direct load/store access to massive memories on one host [20], creating pressure for more high-capacity PM devices and DRAM+NAND Flash-based PM solutions. For example, Samsung already commercially offers devices that can provide 16 TB of volatile DRAM per machine on CXL in this fashion today [40].

Finally, by exposing such devices over the network it is possible to build PM out of large volatile memories [41, 42, 43, 44]. All of the PM support in CPUs today are key to making use of such distributed PMs through hardware interfaces.

In short, despite the demise of Intel's commercial PM offering, the foundations for using PM in applications and building new PM solutions still remain in place, making this a rich and important topic for research. Optimistically, it is even possible that Intel Optane's demise will help prevent the systems community from focusing too heavily on one implementation of PM and its specific idiosyncrasies.

IX. Status & Roadmap

Our initial benchmarks using PM have shown us the potential benefits of PM-aware scheduling for serverless workloads, and they have led us to our initial PM-aware scheduler design. In the short term, our plan is to demonstrate this design on OpenFaas so that we can determine what practical gains PM-aware policies can offer. As we gain experience with workloads, our plan is to find ways to incorporate strong notions of fairness while preserving PM's performance benefits and to determine where serverless application storage access patterns are easier and harder to predict.

Overall, we believe that PM-aware scheduling will be an important step forward for serverless workloads in extracting value out of new PM technologies.

References

[1] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, 2020, pp. 169–182.

[2] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, "Maximizing persistent memory bandwidth utilization for OLAP workloads," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2021, pp. 339–351.

[3] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, 2020, p. 105–119.

[4] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021, pp. 653–669.

[5] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proceedings of the International Middleware Conference (Middleware)*, 2020, pp. 280–295.

[6] L. Zhao, Y. Yang, Y. Li, X. Zhou, and K. Li, "Understanding, predicting and scheduling serverless workloads under partial interference," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021, pp. 1–15.

[7] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2020, pp. 205–218.

[8] Y. Zhang, Í. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 724–739.

[9] Z. Guo, Z. Blanco, M. Shahrad, Z. Wei, B. Dong, J. Li, I. Pota, H. Xu, and Y. Zhang, "Resource-centric serverless computing," *arXiv preprint arXiv:2206.13444*, 2022.

[10] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in *Proceedings of the 19th International Middleware Conference Industry*, 2018, pp. 1–8.

[11] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, pp. 76–84, 2021.

[12] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *Proceedings of the VLDB Endowment*, vol. 13, no. 11, 2020.

[13] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: warming serverless functions better with heterogeneity," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 753–767.

[14] Y. Yang, Q. Cao, J. Yao, Y. Dong, and W. Kong, "Spmfs: A scalable persistent memory file system on optane persistent memory," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, 2021.

[15] D. Koutsoukos, R. Bhartia, A. Klimovic, and G. Alonso, "How to use persistent memory in your database," *arXiv preprint arXiv:2112.00425*, 2021.

[16] A. Uyar, S. Akkas, J. Li, and J. Fox, "Intel optane dcpmm and serverless computing," *arXiv preprint arXiv:2109.11021*, 2021.

[17] C. Katsakioris, C. Alverti, V. Karakostas, K. Nikas, G. Goumas, and N. Koziris, "Faas in the age of (sub-) μs i/o: a performance analysis of snapshotting," in *Proceedings of the 15th ACM International Conference on Systems and Storage*, 2022, pp. 13–25.

[18] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghOSt: Fast & Flexible User-space Delegation of Linux Scheduling," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 588–604.

[19] OpenFaas, "https://github.com/openfaas/faas," Open Source Serverless Platform.

[20] "HOME — Compute Express Link," https://www.computeexpresslink.org/, 2023, [Online; accessed 03-Mar-2023].

[21] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012, pp. 1–12.

[22] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 349–362.

[23] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Oct. 2018, pp. 427–444.

[24] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2021.

[25] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Feb. 2019, pp. 193–206.

[26] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: elastic far-memory for stateful serverless analytics," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2022, pp. 697–713.

[27] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 1–15.

[28] D. Zhou, Y. Qian, V. Gupta, Z. Yang, C. Min, and S. Kashyap, "ODINFS: Scaling PM Performance with Opportunistic Delegation," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022, pp. 179–193.

[29] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. Ramakrishnan, "SPRIGHT: Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-memory," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022, pp. 780–794.

[30] R. Smith, "Intel To Wind Down Optane Memory Business," https://www.anandtech.com/show/17515/intel-to-wind-down-optane-memory-business, 2022.

[31] "Brief: Intel optane persistent memory," https://www.intel.la/content/www/xl/es/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[32] W. Wang, "Architectural support for persistent memory programming," https://community.arm.com/arm-research/b/articles/posts/simplifying-persistent-programming-with-microarchitectural-support, 2021.

[33] M. Larabel, "Risc-v adds support for persistent memory devices," https://www.phoronix.com/news/Linux-6.2-RISC-V, 2022.

[34] A. Khan, "Persistent memory enhancements on ibm power systems," https://www.ibm.com/blogs/systems/persistent-memory-related-enhancements-on-ibm-power-systems-for-sap-hana/, 2019.

[35] "Non-volatile Dual In-line Memory Module," https://www.intel.com/content/www/us/en/developer/articles/technical/enabling-persistent-memory-in-the-storage-performance-development-kit-spdk.html, 2019.

[36] "Agigaram NVDIMM," http://agigatech.com/products/agigaram-nvdimms/.

[37] "HPE NVM," https://www.hpe.com/us/en/storage/hpe-memory-driven-flash.html.

[38] "Nantero NRAM," https://www.nantero.com/.

[39] "Emerging Memory Report," https://thememoryguy.com/emerging-memory-report-updated/.

[40] "Samsung CXL Module," https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module, 2022.

[41] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum *et al.*, "The ramcloud storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–55, 2015.

[42] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[43] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414.

[44] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No Compromises: Distributed Transactions with Consistency, Availability, and Performance," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 54–70.