A Thorough Analysis of Page Fault Handling in Persistent Memory Systems

André Libório[1], **Alexandro Baldassin**[1], **Daniel Castro**[2], **Paolo Romano**[2], **João Barreto**[2]

[1]Universidade Paulista Júlio de Mesquita Filho (UNESP) - Brazil

[2]INESC-ID & Instituto Superior Técnico - Portugal

{andre.lb.ferraz, alexandro.baldassin}@unesp.br

{daniel.castro, paolo.romano, joao.barreto}@tecnico.ulisboa.pt

Abstract. *The new technologies for building persistent devices have reached a point where these devices can be added to the processor bus and accessed via regular load/store instructions. Commonly know as Persistent Memory (PM), these devices have renewed the research interest in systems used to program them. One important implementation technique used by a class of these systems is the use of DRAM as shadow memory, which allows the use of contemporary hardware transactions. However, these systems have an important drawback: if DRAM is considerably smaller than PM, the performance can be degraded due to excessive paging. Despite this, few previous works have looked into that issue. We provide in this paper, for the first time, a thorough analysis of the performance of PM systems when the amount of DRAM is smaller than that of PM. We also present a user-level page handling mechanism that can be integrated in any current PM system. Whereas previous works have considered only synthetic workloads, our study uses a realistic benchmark. The experimental evaluation shows that the final performance under paging is heavily influenced by how often the transactions enters the Single Global Lock (SGL) mode, that is, the amount of conflicts caused by the paging mechanism.*

1. Introduction

Persistent Memory (PM) stands as a byte-addressable memory technology characterized by its non-volatile nature. Similar to SSDs and HDDs, PM can preserve the data when powered off, all while having performance comparable to DRAM but at lower power consumption [Patil et al. 2019]. PM has recently stood out due to Intel Optane DC PM (DCPMM), first released in 2019 in a partnership with Micron [Tyson 2019]. It achieved good performance while presenting much denser DIMMs compared to traditional DRAM available in the server market [Peng et al. 2019, Yang et al. 2020, Xiang et al. 2022]. Although Intel has recently discontinued the Optane DC devices, the industry has embraced the new Compute Express Link (CXL) standard [Jung 2022], which also has support for byte-addressable persistent memory.

Programming these new persistent devices is not straightforward due to numerous software challenges [Baldassin et al. 2021]. Most of the issues comes from the fact that cache memories are volatile and, therefore, when data is written it might still linger in the volatile realm before reaching the PM device. Hence, if some crash happens (such as a power failure) before the data is made durable, the system state might become inconsistent. Specific instructions must be used to force the data out of the cache (so called flush/fence instructions). Even then, crashes can still cause inconsistencies. Consider, for example, the insertion of an element to a persistent linked-list data structure. After the new node is allocated, it is necessary to: (i) make the previous node point to the new one; (ii) make the new node point to the next. If a crash happens after (i) but before (ii) is completed, the linked-list will become corrupted.

Transactions have been employed as one of the key components for programming with PM. Indeed, a number of systems have been recently proposed in the literature for that purpose [Liu et al. 2017, Genç et al. 2020, Castro et al. 2021]. Most of these systems rely on two key features to increase efficiency: (1) they rely on hardware transactions; (2) they use DRAM as shadow memory for execution and update PM via redo logs. In particular, using DRAM as shadow memory assumes that DRAM is as large as PM, which is not reasonable in the majority of the systems today. When DRAM is smaller than PM, a paging mechanism is required to swap shadow and persistent pages. However, with the exception of DudeTM [Liu et al. 2017], none of the current systems handle page swapping adequately. Even DudeTM limited itself to analyze the paging overhead due to shadow memory to very specific and artificial scenarios.

In this paper we aim to provide a thorough analysis of the overhead and behavior of the paging mechanism in current state-of-the-art PM systems. To achieve that, we implemented a flexible user-level page handling mechanism and integrated it into a state-of-the-art PM system, known as SPHT (Scalable Persistent Hardware Transactions) [Castro et al. 2021]. Although we use SPHT as a use case, our proposed page handling strategy could equally be adapted and employed by other PM systems. Furthermore, our characterization makes use of a realistic benchmark, Stanford Transactional Applications for Multi-Processing (STAMP) [Minh et al. 2008]. In particular, we make the following contributions:

- We describe a flexible user-level paging mechanism that can be used by current PM systems (existent ones either relies on specific hardware support or specific kernel versions) (Section 3);
- We provide, for the first time, a deep analysis of the performance of the paging overhead in PM systems that use shadow memory and hardware transactions (Section 4) using the SPHT system and the STAMP benchmark. Our experimental results show that the final performance under paging is heavily influenced by how often the transactions enters the SGL mode.

This paper is organized as follows. Section 2 presents the background and related work. Section 3 presents our user-level paging mechanism and how it is integrated into SPHT, a state-of-the-art PM system. The experimental evaluation is presented in Section 4 and we conclude the paper in Section 5.

2. Background and Related Work

Most of the systems in the literature use transactions as the basic PM programming construct [Baldassin et al. 2021]. The ACID properties (Atomicity, Consistency, Isolation, and Durability) of a transaction [Rahmatian 2002] fit perfectly with the PM programming model, allowing all changes to be rolled back in case of crashes and making sure the data

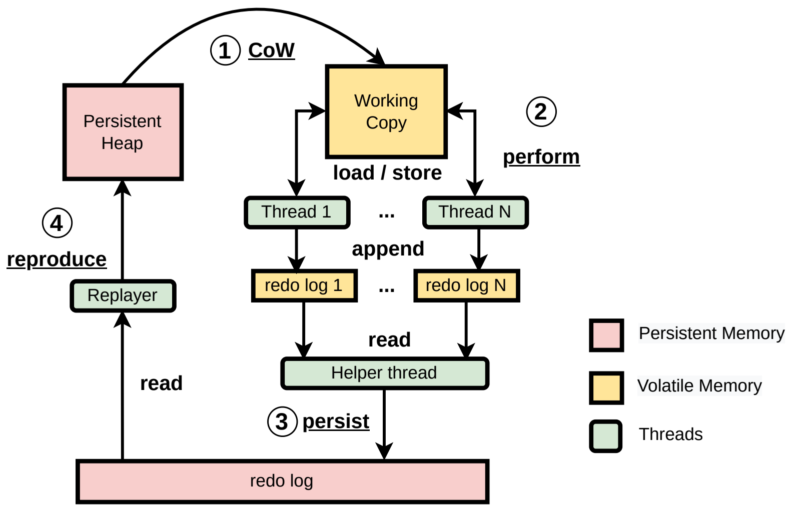


Figure 1: General scheme of PM systems with shadow memory: Copy-on-Write (CoW) is used to create a working copy of the persistent heap in DRAM – updates are first appended to redo logs and then persisted and reproduced in the persistent heap by background threads.

is durable once a transaction is committed. Initially, most PM systems were built upon the research on Software Transactional Memory (STM) [Shavit and Touitou 1995], which implemented transactions purely in software by providing data versioning and conflict detection. Mnemosyne [Volos et al. 2011] and NV-Heaps [Coburn et al. 2011] are examples of systems in this category.

With the introduction of hardware support for transactions in the 2010's by Intel and IBM [Intel 2020, Le et al. 2015], some new PM systems were proposed that benefited from this hardware acceleration [Liu et al. 2017, Castro et al. 2019, Giles et al. 2017, Castro et al. 2021]. One problem, though, is that hardware transactions do not provide durability and, as such, a software mechanism has to be implemented to make sure the changes are persisted. Another key mechanism that these systems employed is to map the persistent pages to DRAM (shadow memory) and execute transactions on them. Each transaction builds a persistent redo log which is later on applied to PM. Figure 1 shows the general architecture of a PM system that uses DRAM as shadow memory.

Firstly, a *working copy* of the persistent data is created usually via a copy-on-write mechanism provided by the OS ①. Then the execution of the system split into three fully asynchronous steps as initially proposed by DudeTM [Liu et al. 2017]. In the first step, called *perform* ②, the modifications are performed directly in the working copy. A volatile redo log is also kept locally by the transactions that run in the context of a thread. The working copy prevents the changes made by transactions from immediately affecting the persistent state. Only when a transaction commits successfully the volatile redo logs are persisted ③, usually in the background by a helper thread. This *persist* stage consists of permanently writing the redo logs in a separate memory region kept in PM. Notice that after this step the modifications are durable and will be replayed after recovery in case a failure occurs. Finally, in the third step, the redo logs are replayed in the persistent heap ④ by one or more replayer threads.

One key aspect of the shadow memory mechanism is that it requires the amount of

DRAM memory to be equivalent to PM, which is not expected to be the case since PM is much denser than DRAM. When much more PM pages exist than DRAM ones, a paging mechanism is required to handle the page faults. In theory, one could use the operating system (OS) paging mechanisms to deal with this, but constantly moving pages to and from the swap area might cause a performance degradation. All systems in the literature, except DudeTM [Liu et al. 2017], actually do not deal with this issue at all. DudeTM does provide some basic mechanism to handle the page faults, by ignoring pages that are removed (page-out) and appropriately replaying the corresponding redo logs when a page is brought from PM (page-in). Notice that ignoring a page-out event is safe because the changes are already persistent in the redo log; therefore one only needs to make sure the changes made to a specific page were already applied before mapping it back to DRAM.

Dealing with shadow paging requires intercepting and changing the OS memory management routines. DudeTM implementation provides two different ways to handle this: (i) by using the Dune library [Belay et al. 2012], an approach that uses Intel's VT-X virtualization technique to enable user-space code to manage their own page tables; (ii) by implementing its own software-based paging. The first solution is specific to Intel processors and requires changes to specific Linux kernels. The second solution is specific to DudeTM and is not flexible enough to be applied to other PM systems. Another option to deal with page faults in Linux kernels (starting from version 4.3) is to use the `userfaultfd` library [kernel development community 2024]. It allows user space code to be notified of page faults in designated virtual address ranges. There is no way, however, to evict a specific page using userfaultfd. There was an attempt to extend `userfaultfd` with that feature [Caldwell et al. 2017], but it is not part of the open source distribution and therefore it does not support all kernel versions. Our solution (described in Section 3.2) relies on Linux signals to catch the page faults and the memory-map system calls to map and unmap the pages.

3. Implementing Paging Handling in SPHT

In order to characterize the overhead of page swapping in PM systems and show the flexibility of our page handling scheme, we adopted SPHT [Castro et al. 2021] as a use case since it is a state-of-the-art implementation of a system that follows the shadow memory with Hardware Transaction Memory (HTM) transactions, as depicted by Figure 1. In this section, we start by showing the basic workings of SPHT, followed by our user-level paging mechanism and its integration to SPHT.

3.1. SPHT

SPHT is one of the more recent PM systems published. It follows the shadow memory approach with hardware transactions as discussed previously. Recall that with the shadow memory technique, the data is first written to volatile working memory. A copy of this data is also saved temporarily in a volatile redo log. Since SPHT uses transactions, the modifications are automatically rolled back in case of a failure or conflicts with other transactions in the system. In case of high contention scenarios, the system grabs a Single Global Lock (SGL) and all transactions are serialized. Hardware transactions do not provide durability and, therefore, the commit protocol implemented by SPHT is more elaborated, being divided into two main stages: (i) logical commit – the hardware transaction is committed but the data is not yet written back do PM; (ii) durable commit – the system checks whether the written data is safe to be persisted (i.e., there is no older transaction that has not yet flushed its data to PM) and finally writes the redo logs to PM. One important novelty of SPHT is that the commit protocol is designed in such a way that it allows grouping concurrent commits (for more details, please check the paper [Castro et al. 2021]).

The most important aspect of SPHT with regard to page handling is how the redo logs are stored and the replayer thread. Each thread has its own private persistent redo log area. The replayer is a background thread that periodically scans through this log area and applies the changes to PM. As we mentioned earlier, SPHT does not handle persistent page faults, meaning it relies on the OS' default page handling when the DRAM space is full. As discussed before, current OS paging implementation is oblivious of PM systems and therefore unnecessary overhead is added. In order to improve that, we need to know which changes were made to each evicted page, so that the corresponding redo logs can be replayed before the page is brought back in. We discuss how we implemented this behavior after introducing our page handling mechanism in the next subsection.

3.2. User-level Page Handling

After trying several options for handling page faults we decided to implement our own mechanism. Recall that available options either required specific hardware (such as Dune [Belay et al. 2012]) or relied on specific Linux kernel versions. As such, we devised a solution that relies only on Linux signals and the memory map system calls. More concretely, we defined a custom `SIGSEGV` handler and leveraged the `mmap`/`munmap` functions. To verify whether a given page is mapped or not, we keep a one-level page table in DRAM as a bitmap. When the page handling module is initialized, it is necessary to inform the total size of the persistent heap and the working copy. These values do not need to match the real values present in the machine and therefore we are able to simulate scenarios where DRAM is much more scarce than PM.

Our solution does not perform any address translation at all (unlike DudeTM's approach). Instead, we register a callback function with the Linux kernel that signals our library when a not present page is accessed. This will happen when a transaction accesses a memory address whose page was not yet mapped into the working copy memory. After receiving a page fault signal, our library first checks whether we still have space in the working copy memory. If that is not the case, then it is necessary first to evict a page to make room for the new one. We use a simple eviction algorithm based on a random number to decide which page to evict, but more elaborated ones (such as LRU) can be easily added. This page is then unmapped and the faulted one is mapped. The user-level page table is then updated accordingly.

All that is required to use our library is to call an initialization routine with the desired configuration (size of working copy memory and persistent heap, as well as the page size). The signal handler is then registered with the kernel and paging tasks are performed automatically. Adding other behaviors to the basic one explained above is also very easy, as all that is required is to extend the page-in and page-out actions. We show how that can be done with SPHT in the next subsection.

3.3. Integration with SPHT

Recall that, when the working copy is much smaller than the persistent heap, the system must handle paging. Although the OS could theoretically manage this scenario, the overhead of moving the evicted pages to and from the swap space (usually stored in HD/SSD) could significantly hurt performance. Therefore, we added the page handling ability to SPHT by using our previously described user-level paging mechanism.

We follow the technique first described by DudeTM for page handling of shadow memory: on a page-out, nothing is done (i.e., the page contents are not written back to PM). This approach is correct because the persistent redo logs already keep the newest state. On a page-in, however, we need to make sure that all the updates relative to the page being swapped into the shadow memory are already applied.

In order to implement this feature, we use an extra data structure, a hashmap, that stores the timestamp of the last transaction that made changes to a given page. This timestamp is collected (usually via the `rdtsc` instruction) right before a transaction's logical commit. The redo log is then scanned and, for each written address, the `<page-number,timestamp>` tuple is inserted into the hashmap. Since the redo logs in SPHT also contain the timestamp of the transaction that generated it, the replayer thread can keep the timestamp of the most recent replayed redo log. In that case, all that is required to implement the shadow paging scheme is to extend the page-in action to: (i) retrieve the timestamp of the page that is being swapped to shadow memory via the hashmap; (ii) compare the page's timestamp to that of the replayer thread; and (iii) if the page's timestamp is lower, it means that the replayer already applied the changes to PM and it is safe to continue; otherwise (i.e., replayer's timestamp is lower), we need to wait for the replayer to advance until the replayer's timestamp is greater than the page's timestamp.

Although this waiting phase during the page-in event could potentially hamper performance, our experimental results show that in the majority of the scenarios the replayer thread is fast enough. Moreover, we noticed that waiting for the whole working copy memory to be full to start ejecting pages could potentially stall the system (all transactions will be waiting for the page-in events to complete). Therefore, we added another background thread that wakes up when the paging system detects that the working copy memory is becoming full. This thread then starts to remove pages until the shadow memory occupation reaches a watermark.

4. Experimental Evaluation

In this section we first present the configuration of the experiments followed by the performance analysis.

4.1. Setup

In order to measure and compare the performance of the paging mechanism implemented on top of SPHT, we make use of the STAMP benchmark suite [Minh et al. 2008] with the largest suggested configuration for each benchmark. Contrary to previous studies that considered only synthetic workloads, using STAMP allows us to analyze workloads that present a high diversity and focus on real use cases. Table 1 shows the STAMP

Table 1: STAMP benchmark suite.

Application	**Domain**	**Arguments**
Genome	bioinformatics	-g16384 -s64 -n16777216
Intruder	security	-a10 -l128 -n262144 -s1
Kmeans Low	data mining	-m1000 -n1000 -t0.00001 -i random-n65536-d32-c16
Kmeans VLow	data mining	-m40 -n40 -t0.00001 -i random-n65536-d32-c16
Labyrinth	engineering	-i random-x48-y48-z3-n64
SSCA2	scientific	-s20 -i1.0 -u1.0 -l3 -p3
Vacation Low	transaction processing	-n2 -q90 -u98 -r1048576 -t4194304
Yada	Delaunay triangulation	-a15 -i ttimeu1000000.2

Table 2: Working copy size for each application used in scenario `PEN`

Application	Genome	Intruder	KLow	KVlow	Lab.	SSCA2	Vacation	Yada
Size (KB)	1472884	713032	1363	819	131	3094090	614466	2680507

applications and configurations used in our experiments. The only application left out of the tests was Bayes due to its characteristic instability [Dragojevic and Guerraoui 2010].

All the results presented here were collected using a system equipped with a dual-socket Intel(R) Xeon(R) Gold 5317 CPU (total of 24 physical cores and 48 threads), with a total of 256GB DRAM, and 512GB of Intel Optane PM DC (200 Series). In the experiments we do not use the second socket in order to exclude the non-uniform memory access (NUMA) impact, and pin each software thread (max of 24), with physical cores as a priority. The Linux operating system with kernel 5.4.0 was used. The applications were compiled using GCC 9.4.0 with the `-O2` flag. The reported results represent the average of 10 runs; a 95% confidence interval bar is also shown for the throughput plots.

In order to better understand the impact of the paging mechanism, the experiments consider three scenarios:

- No paging (`NP`) – The working copy size is large enough and therefore no page faults happen during execution. This configuration does not present any overhead due to the paging mechanism;
- No paging but the hashmap is enabled (`NPHASH`) – The working copy size is large enough and no page faults are generated, but the hashmap structure is used to track the updated pages. This configuration aims to measure the overhead of tracking the updated pages only;
- Paging mechanism enabled (`PEN`) – The working copy size is smaller than what is necessary to execute the application, so that page faults are forced to happen. The memory size used, as shown in Table 2, was based on the memory sizes presented in a previous work [Baldassin et al. 2015] that analyzed the allocation behavior of the STAMP applications. Table 3 shows the number of page-outs generated by each application for a given number of threads.

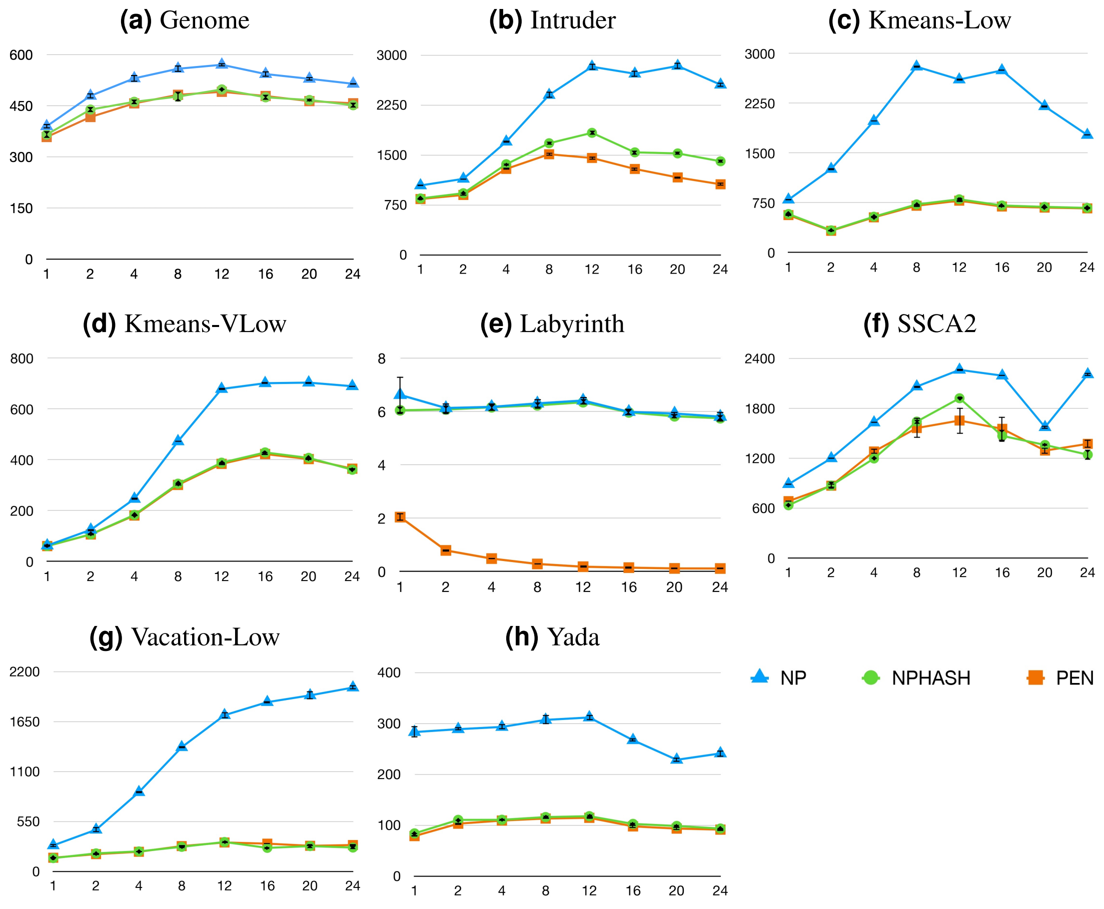
4.2. Results

Figure 2 shows the throughput of the STAMP applications for each of the scenarios. As expected, the scenario with no paging (`NP`) always displays the best performance.

Table 3: Number of page-outs generated for each application

Application	1T	2T	4T	8T	12T	16T	20T	24T
Genome	5102	5103	5105	5106	5107	5109	5112	5115
Intruder	18405	24805	26843	43461	56808	70072	84154	96175
KmeansLow	11	11	11	11	11	11	11	11
KmeansVLow	760	760	760	744	728	720	719	717
Labyrinth	526	1743	2374	2607	2885	3140	3375	3377
SSCA2	1852	1852	1853	1856	1858	1859	1862	1865
VacationLow	5912	5906	5910	5916	5935	5943	5931	5939
Yada	4266	5924	7746	9049	9714	9994	10256	10558

Figure 2: STAMP throughput (x1000) results for each evaluated scenario.



Recall that in the NPHASH scenario, it is necessary to scan the transaction redo log and insert the respective `<page-number,timestamp>` tuple in the hashmap. Since this needs to be done before the hardware transaction is committed, there are two new situations that can degrade performance. Firstly, the transaction duration is longer, increasing the chance of conflicts against other concurrent transactions. Secondly, since all transactions must update the hashmap structure, there is another source of conflict that did not exist in the NP scenario.

As for the PEN scheme, further overhead may be caused because system calls to memory map and unmap the new and old pages, respectively, are required. These calls will further abort the hardware transactions and force the concurrent transactions to enter the SGL path, serializing the execution. Moreover, when a new page is brought to the working copy, we need to wait for the replayer to apply the changes made to this page. In our experiments we noticed that this waiting time is almost always zero, indicating that the replayer thread is fast enough.

We also use the abort rate illustrated in Figure 3 to complement the explanation. Hardware transactions can abort due to: i) a *conflict* with other transactions; ii) *capacity* limitations (cache size was not sufficient to hold the transaction state); and iii) other reasons, like unsupported instruction such as the `syscall`. This figure shows the abort rate for each scenario evaluated in Figure 2.

With the considerations above, we can more clearly understand the performance behavior of the different scenarios. Firstly notice that, in general, the abort rate increases with the addition of the hashmap (`NPHASH`) but stays mostly the same when paging is added (`PEN`), as can be observed in Figure 3. This difference is less significant for Intruder (Figure 3b) and Yada (Figure 3h), as their abort rate with `NP` is already relatively high. Labyrinth is an exception and will be explained later. This increase in the abort rate is explained by the fact that, with the hashmap, transactions are more likely to conflict with each other, as we have pointed out earlier.

The increased abort rate has different implications for the applications throughput as seen in Figure 2. With the exception of Labyrinth (more on that later), the performance of `NPHASH` and `PEN` are very similar. This indicates that the extra overhead of using the memory map syscalls for paging is minimal. Genome and SSCA2 presented the best overall results, with both `NPHASH` and `PEN` following the same trends of `NP`. With Intruder and Kmeans-VLow, we can notice that there was some performance drop for `NPHASH` and `PEN`, but their scalability still resembles `NP`. For these applications, our results show that the induced conflicts caused by the hashmap did not severely increased the number of transactions that entered the SGL mode and, therefore, the final performance was not critically affect as the remaining applications.

The `NPHASH` and `PEN` schemes did not produce good results for Kmeans-Low (Figure 2c) and Vacation-Low (Figure 2g). We noticed that for these applications, the number of transactions that committed with SGL increased a lot: around 52x in the worst case for Kmeans-Low (from 109,000 to 5,600,000 transactions) and 17x for Vacation-Low (from 248,000 to 4,100,000 transactions). Recall that the system enters in SGL mode when a transaction aborts repeatedly. When paging is enabled, this scenario is more likely to happen. With SGL, only one transaction is running and therefore the execution is serialized, explained the flat scalability lines for these applications with paging. Something similar happened with Yada (Figure 2h), but this application did not show any scalability even without paging (`NP`).

Labyrinth (Figure 2e) is characterized by very long transactions that usually exceeds the hardware buffering limits, causing a lot of capacity aborts (Figure 3e). As such, even the `NP` scheme does not scale and the overhead added by the hashmap (`NPHASH`) is almost negligible. However, we noticed that when page faults start to happen, 85% of all the transactions enters the SGL (compared to 50% without the page faults) that, added

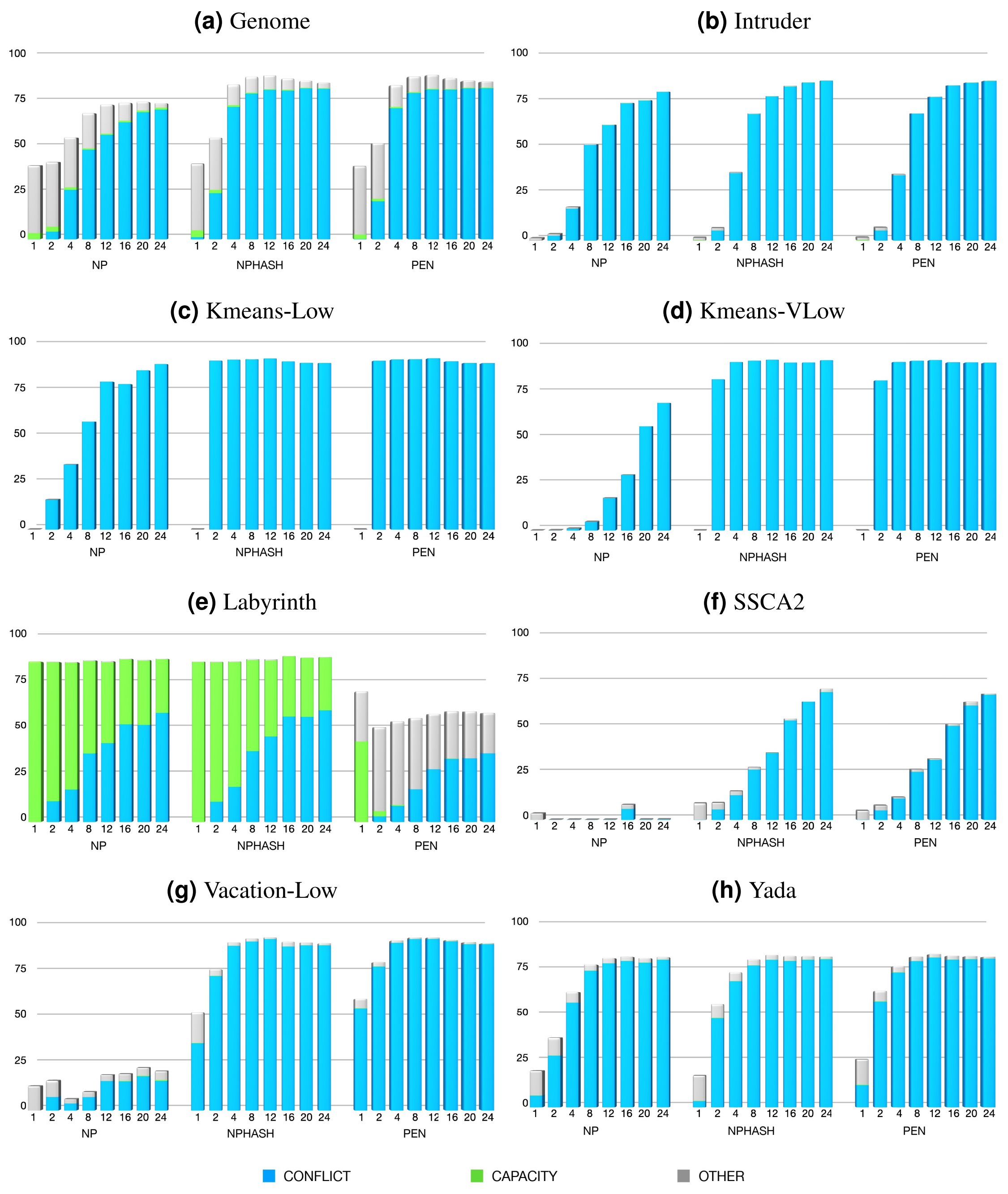


Figure 3: STAMP abort rate (%) for each of the evaluated scenarios

with extra cost of the memory map system calls, causes a slow-down with `PEN`.

5. Conclusion

Transactions have been used as the main abstraction for programming the new Persistent Memory (PM) systems. One important implementation feature of these systems is the use of DRAM as shadow memory, allowing the efficient use of current hardware transactions. However, these systems have an important drawback: if DRAM is considerably smaller than PM, the performance can be degraded due to excessive paging. Nonetheless, few previous works have looked into that issue. In this paper, we provided the first thorough analysis of the performance of PM systems under the paging condition using a realistic benchmark (STAMP). We also introduced a flexible user-level paging mechanism and integrated it into SPHT, a state-of-the-art PM system.

The experimental evaluation revealed that the final performance under paging is heavily influenced by how often the transactions enters the SGL mode. Since the paging mechanism requires annotating which pages have been updated inside transactions, the likelihood of that happening is increased. For the eight workloads studied under paging, two displayed very low overhead, two low overhead (the workload still scaled), three did not scaled at all, and one workload even presented a slow-down. Our results point to the need of improving the execution of transactions under SGL, maybe using a combination of hardware and software transactions.

Acknowledgments. The authors would like to thank the São Paulo Research Foundation (FAPESP), processes nº 2018/15519-5, 2023/12624-0 for supporting this work.

References

Baldassin, A., Barreto, J., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Computing Surveys (CSUR)*, 54(7):1–37.

Baldassin, A., Borin, E., and Araujo, G. (2015). Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 87–96.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. (2012). Dune: safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, page 335–348. USENIX Association.

Caldwell, B., Im, Y., Ha, S., Han, R., and Keller, E. (2017). Fluidmem: Memory as a service for the datacenter.

Castro, D., Baldassin, A., Barreto, J., and Romano, P. (2021). SPHT: Scalable Persistent Hardware transactions. In *FAST'21*, pages 155–169.

Castro, D., Romano, P., and Barreto, J. (2019). Hardware transactional memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79.

Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. (2011). NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *ASPLOS'11*, pages 105–118.

Dragojevic, A. and Guerraoui, R. (2010). Predicting the scalability of an STM: A pragmatic approach. In *5th ACM SIGPLAN Workshop on Transactional Computing*.

Genç, K., Bond, M. D., and Xu, G. H. (2020). Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–74.

Giles, E., Doshi, K., and Varman, P. (2017). Continuous Checkpointing of HTM Transactions in NVM. In *ISMM'17*, pages 70–81.

Intel (2020). Intel® Architecture Instruction Set Extensions Programming Reference.

Jung, M. (2022). Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 45–51.

kernel development community, T. (2024). Userfaultfd. [Online; Access in July, 10 2024]. Available in: https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html .

Le, H., Guthrie, G., Williams, D., Michael, M., Frey, B., Starke, W., May, C., Odaira, R., and Nakaike, T. (2015). Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8–1.

Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., and Ren, J. (2017). DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *ASPLOS 17*, pages 329–343.

Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE.

Patil, O., Ionkov, L., Lee, J., Mueller, F., and Lang, M. (2019). Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 288–303, New York, NY, USA. Association for Computing Machinery. Available in: ¡https://doi.org/10.1145/3357526.3357541¿.

Peng, I. B., Gokhale, M. B., and Green, E. W. (2019). System Evaluation of the Intel Optane Byte-Addressable NVM. In *MEMSYS'19*, pages 304–315.

Rahmatian, S. (2002). Transaction processing systems. *Encyclopedia of Information Systems*, 4:479.

Shavit, N. and Touitou, D. (1995). Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213.

Tyson, M. (2019). Intel Optane DC Persistent Memory launched. Retrieved from https://hexus.net/tech/news/storage/129143-intel-optane-dc-persistent-memory-launched/.

Volos, H., Tack, A. J., and Swift, M. M. (2011). Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104.

Xiang, L., Zhao, X., Rao, J., Jiang, S., and Jiang, H. (2022). Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, page 488–505, New York, NY, USA.

Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., and Swanson, S. (2020). An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST'20*, pages 169–182.