Bruno Chinelato Honorio

Melhorando o Desempenho de Dispositivos com Memória Persistente

Relatório de Pós-doutorado realizado na Universidade Estadual Paulista (UNESP), IGCE - DEMAC, Rio Claro.

Supervisor(a): Prof. Dr. Alexandro José Baldassin

FAPESP - 2023/04971-2

Rio Claro

2025

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), Brasil. Processo nº 2023/04971-2. As opiniões, hipóteses e conclusões ou recomendações expressas neste material são de responsabilidade do(s) autor(es) e não necessariamente refletem a visão da FAPESP.

1 Relatório Científico de Pós-Doutorado

1.1 Resumo

Memória persistente (PM) é uma tecnologia emergente que combina o acesso rápido aos dados com endereçamento por byte, além de oferecer grande capacidade de armazenamento persistente. Devido a essas características, a PM tem ganhado cada vez mais atenção, com diversos estudos sendo realizados para explorar seu uso e desempenho. Paralelamente, trabalhos sobre estruturas de dados (ED) têm evoluído para maximizar o aproveitamento das PMs. Isso inclui o desenvolvimento de EDs adaptadas para a persistência e transações atômicas.

Este trabalho focou em três pontos: explorar o impacto da alocação de memória em dispositivos persistentes, revisar os trabalhos mais relevantes na área, e adaptar um framework persistente para arquiteturas NUMA.

1.2 Realizações

Durante o período referente de dois anos, o beneficiário realizou uma série de atividades que serão descritas a seguir. Será usado o cronograma da proposta e o plano de desenvolvimento anexados no início do projeto como referência para o que se foi feito.

O beneficiário desenvolveu uma série de atividades com dispositivos persistentes. Em particular, como forma de fortalecer habilidades acadêmicas, o beneficiário supervisionou e acompanhou o trabalho de três alunos do Professor Alexandro, dois de graduação (Edgar e Otávio) e outro de mestrado (Lucas Bastelli). Os trabalhos de graduação focaram no desempenho de alocadores de memória persistente e seu impacto. Resultou em três artigos para duas edições da escola regional de alto desempenho do estado de São Paulo (ERAD-SP). Esses artigos estão em anexo deste relatório. A publicação destes artigos de graduação seguiram boa parte do cronograma de estudos e levantamentos previstos. (Pontos 1, 2, e 3 do cronograma).

Durante o acompanhamento do aluno de mestrado, publicou-se no SSCAD (Simpósio em Sistemas Computacionais de Alto Desempenho) um artigo fazendo uma revisão sistemática sobre estruturas de Dados em dispositivos persistentes contemporâneos, como estava previsto no cronograma. Este artigo está anexado no relatório. O beneficiário também participou da banca de estudos especiais do aluno Lucas Bastelli, avaliando seu progresso durante aquele período. Percebeu-se, durante esses estudos e levantamentos, um interessante ob-

jeto de pesquisa em dispositivos persistentes: a sua implementação em arquiteturas NUMA (Non-uniform memory access), em específico, como o desempenho de grafos dinâmicos é afetado em arquiteturas NUMA. O aluno de mestrado tem trabalhado com um framework chamado DGAP (Efficient Dynamic Graph Analysis on Persistent Memory) e implementou com sucesso uma versão NUMA do mesmo. Os resultados e experimentos estão sendo levantados no momento e serão publicados em um evento internacional relevante da área de programação paralela. (Pontos 4 e 5 do cronograma).

Além disso, durante a escola regional de alto desempenho do ano de 2024, o beneficiário ajudou como ”coach”o aluno de mestrado Lucas Bastelli na competição de programação paralela do evento, e obteve o primeiro lugar na competição.

Ainda como parte do desenvolvimento de atividades relacionadas à dispositivos persistentes, o beneficiário fez uma colaboração com o INESC-ID (Instituto Superior Técnico, Universidade de Lisboa, Portugal) para o desenvolvimento de um survey sobre o posicionamento de dados em memórias em camadas (tiered memory). Este trabalho visou aumentar sua experiência na pesquisa com memória persistente. Este survey está nos passos finais de desenvolvimento e também será submetido nos próximos meses para a ACM Computing Surveys. (Essa oportunidade surgiu com a experiência adquirida pelo beneficiário durante seu pós-doutorado.)

O beneficiário também participou como revisor de diversos eventos, tanto nacionais quanto internacionais, como parte de fortalecer suas atividades profissionais. Também serviu como assistente na disciplina de programação paralela da UNESP de Rio Claro para alunos do quarto ano, adaptando um sistema de submissão de atividades, assim como tirando dúvidas dos alunos e auxiliando o professor Alexandro no desenvolvimento das atividades práticas.

2 Plano de Gestão de Dados

O código-fonte dos trabalhos encontra-se no github, nos links:

1. **https://github.com/otaviosso/na-graph** para o trabalho de mestrado.
2. **https://github.com/otaviosso/ralloc** para os trabalhos de iniciação científica.

CERTIFICADO

ERAD-SP

A Sociedade Brasileira de Computação atesta a participação de

Bruno Chinelato Honorio

no(a) XIV Escola Regional de Alto Desempenho de São Paulo (ERAD-SP 2023),

realizado(a) entre 17 e 19 de Julho em São José dos Campos - SP, com carga horária de 19 horas.

Profa. Thais Vasconcelos Batista
Presidente da Sociedade Brasileira de Computação

SBC
Sociedade Brasileira de Computação

Prof. Denis Lima do Rosário
Diretor de Eventos e Comissões Especiais

CERTIFICADO

ERAD-SP

A Sociedade Brasileira de Computação atesta a participação de

Bruno Chinelato Honorio

no(a) XV Escola Regional de Alto Desempenho de São Paulo (ERAD-SP 2024),

realizado(a) entre 16 e 18 de Maio em Rio Claro - SP, com carga horária de 22 horas.

Profa. Thais Vasconcelos Batista
Presidente da Sociedade Brasileira de Computação

SBC
Sociedade Brasileira de Computação

Prof. Denis Lima do Rosário
Diretor de Eventos e Comissões Especiais

SSCAD 24

CERTIFICADO

A Sociedade Brasileira de Computação atesta a participação de

Bruno Chinelato Honorio

no(a) XXV Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD 2024),

realizado(a) entre 23 e 25 de Outubro em São Carlos - SP, com carga horária de 24 horas.

Profa. Thais Vasconcelos Batista
Presidente da Sociedade Brasileira de Computação

SBC
Sociedade Brasileira de Computação

Prof. Denis Lima do Rosário
Diretor de Eventos e Comissões Especiais

WILEY

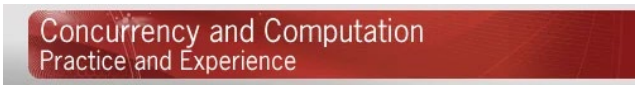
Reviewer Certificate

This certificate is awarded to

BRUNO CHINELATO HONORIO

for serving as a reviewer for

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE



Thank you for reviewing 1 manuscript in 2023

14 March 2024
Date

David W. Walker, Jinjun Chen, Nitin Auluck and Martin Berzins
Editors

An Initial Performance Analysis of Persistent Memory Allocators

Catalina Munoz[1], **Bruno Honorio**[1],
Lucas Bastelli[1], **Emilio Francesquini**[2], **Alexandro Baldassin**[1]

[1]Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP)
Rio Claro – SP – Brazil

[2]Universidade Federal do ABC (UFABC)
SP – Brazil

Abstract. *Persistent Memory (PM) has recently resurfaced with the advent of new technologies that make it a competitive option in terms of performance compared to traditional storage, adding the benefit of durability. Because it is byte-addressable, the use of PM is similar to that of volatile RAM (DRAM). However, there are some differences related to data consistency and read/write latency, which makes it necessary to have special memory allocators. This paper presents an initial performance analysis of the impact of two persistent memory allocators, PMDK and Ralloc, using three typical data structures, namely, i) Linked List, ii) Skip List, and iii) Hash Map. Although the study is in its initial phase, we could observe the impact of PM allocator in performance.*

1. Introduction

Persistent memory (PM) is a byte-addressable memory technology, similar to typical volatile RAM (DRAM) [Baldassin et al. 2021]. Its use has recently become popular since these memories share the same data bus as the DRAM. Moreover, Intel have recently commercialized persistent devices (Optane DC) [Tyson 2019] and the CXL (Compute Express Link) interconnect [Jung 2022] has added support for PM. Even though it is non-volatile storage, which is typically slower, current PM performs considerably better than storage devices such as solid-state drives, due to lower read/write latencies. For instance, PM read latency can be as low as 350 nanoseconds, compared to 10,000 nanoseconds for an SSD [Izraelevitz 2019]. This characteristic makes PM a viable option for high-performance computing that requires data durability with a relatively low cost in terms of performance.

Despite the benefits of persistent memory, optimally using it is a non-trivial task. Unlike DRAM, the storage space is much higher. However, the durability characteristic makes it necessary to guarantee data consistency in case of a power loss. Therefore, programmers are required to guarantee that the stored data can be adequately recovered and accessed following a system crash. Aside from typical store instructions, cache flushes must be regularly executed to ensure that persistent data is not lost during a sudden system failure. A persistent memory allocator must specially consider how allocated data will be recovered in case of system crashes (e.g., power failure). Several allocators have been proposed to date [Scargall 2020] [Bhandari et al. 2016] [Cai and Wen 2020], with the main differences in memory layout, metadata used for data recovery, and handling of memory deallocation and release (e.g., using garbage collectors). However, it is still

not clear how the different design choices affect the overall performance. In this short paper, we show initial performance results using two well-known persistent allocators: 1) PMDK (Persistent Memory Development Kit), an allocator implemented by Intel and part of its development kit for programming with PM [Scargall 2020]; and 2) Ralloc, a state-of-the-art allocator [Cai and Wen 2020]. In this initial report we make use of three simple data structures: a linked list, a skip list, and a hash set. We show how the allocator used plays an important role in performance, particularly during deallocation, with the number of operations per second going between 4x and 20x higher with Ralloc compared to PMDK.

2. Persistent Memory Allocators

Dynamic memory allocation is among the most expensive and common operations in software development. Studies conducted with a group of heap-intensive applications have shown that, on average, 30% of the total execution time is spent on dynamic memory management [Tiwari et al. 2010]. Compared to traditional ones, PM allocators have further requirements, such as the need for data consistency (allocator metadata needs to be persistent) and different design tradeoffs. For instance, some designs are optimized to reduce write endurance of PM [Yu et al. 2015]. In this initial study, we concentrate on two important persistent allocators: PMDK [Scargall 2020] and Ralloc [Cai and Wen 2020]. In the following we provide a short introduction, highlighting the main characteristics of each allocator.

PMDK: PMDK is a set of libraries developed by Intel [Scargall 2020] for C and C++ that enable persistent memory usage, including allocation and deallocation. In order to use the persistent memory space with PMDK, the user must first allocate a root pointer, from which the applications' memory region will be reachable. PMDK separates the entire memory region into zones of specific size, and each zone into chunks of variable size. For allocation, a blocks' size is rounded up to match the next size of a free list. The malloc method provided by PMDK finds a free block and attaches it to previously allocated blocks. This operation occurs atomically to avoid possible memory leaks during a system crash. Additionally, redo and undo logs are kept for allocation atomicity and transactional snapshotting of a memory region, respectively. For deallocation, the free method gets the pointer to the block and puts it back into the free list. To manage the free address space of the application, PMDK maintains in the DRAM a vector of pointers to persistent memory blocks of specified size.

Ralloc: Ralloc is a persistent memory allocator developed by Wentao et al. [Cai and Wen 2020], which reorganizes the typical memory layout to enable recoverability. Ralloc separates memory in superblocks of the same size, mapped in a single region to keep the applications' data. To guarantee consistency and recoverability, two more regions are created in the address space to maintain metadata of superblocks size, state, and allocated data types. In the main memory of the application, superblocks are divided in an array of subblocks, where a free/allocated identifier is kept, and a free subblock points to the next free subblock on the list. Each superblock holds a state that indicates if such superblock is free, fully or partially allocated. Inside the actual memory region, objects are arranged according to their sizes by keeping blocks of equivalent sizes inside the same superblock. Therefore, during the allocation of an object of a specific size, a superblock for such a size class is selected, and a pointer for the next free block inside the superblock is

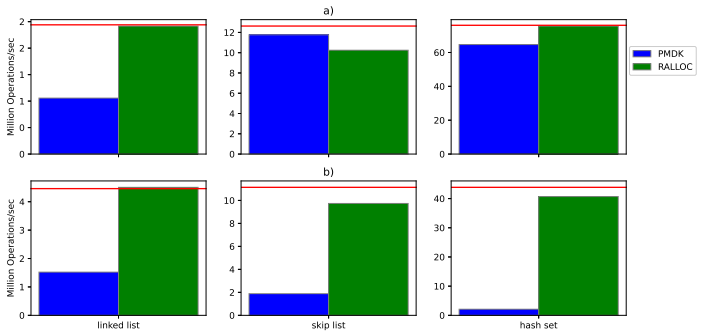


Figure 1. Add (a) and Update (b) operations per second for linked list, skip list, and hashset.

returned. For deallocation, a simple change from "full" to "empty" in the state descriptor of the block is required. The memory reclaim process consists of pushing the block from a "full" or "partial" list to a "partial" or "free list", according to the state transition of the block. Superblock reclamation is delayed during a malloc operation when no superblock is available for allocation. This causes the deallocation process to be much faster.

3. Experimental Setting and Performance Results

To carry out the performance analysis, we use three data structures on which `delete` and `add` operations require allocation and deallocation, and a `search` operation only reads from persistent memory. The memory access patterns depend on each data structure used as follows. A **Linked List** is a linear structure in which each element points to the next and/or the previous one. For this experiment, we use a simple linked list; in a **Skip List**, similar to linked lists, each of the elements points to the next one. However, these lists tend to be more efficient search operations since the list is arranged in levels that allow ordering the data; a **Hash Set** is a structure that facilitates the search for non-repeating elements by arranging them in semi-ordered groups (called buckets) using a hashing mechanism.

The used data structures are allocated exclusively in PM. Other metadata (volatile) used during execution is allocated in DRAM. Applications were executed in a machine powered by an Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz processor with 196GB of RAM, and 128GB of Intel Optane DC Persistent Memory. We compare the PMDK and Ralloc allocators and used LRmalloc [Leite and Rocha 2019], a DRAM allocator, as a performance reference (red line in the figures). Performance results are presented in terms of the number of operations per second executed by each application for the respective data structures. All sets are initialized with 256 randomly generated elements of integer type. Each application is executed 10 times with a 10 seconds duration each time. Results of each run are averaged to obtain the number of ops per second (ops/sec).

The top row of Figure 1 shows the number of `add` operations per second performed for each set. Each element is randomly generated and added if the value does not previously exist in the set. As can be seen in the figure, the difference between Ralloc and PMDK is large, particularly for the linked list, where 128% more set add operations

are performed per second using Ralloc than PMDK. For the hash set, 16% more operations are executed using Ralloc, compared to PMDK. However, a particular case occurs with the skip list, wherein PMDK outperforms Ralloc. PMDK performs 16% more `add` operations than Ralloc. This is possibly a consequence of the memory layout in PMDK, which benefits skip list lookup over Ralloc.

The bottom part of Figure 1 shows the number of `update` operations performed per second. This operation is composed of additions and removals of an element in the set (with equal probability). A wider performance difference can be seen between PMDK and Ralloc, with Ralloc performing better for all the studied sets. In the case of Ralloc, the garbage collector allows delayed memory recovery. As Rallocs' implementation is based on LRmalloc, the difference in performance is expected to be solely given by the differences in latency between PM and DRAM. In most of the tests, Ralloc behaves relatively similarly to LRmalloc, making it evident that PM is competitive for high-performance computing.

4. Conclusions

Our initial results show that persistent memory allocators play an important part in performance. Although in general Ralloc displayed much better results, we also spotted a scenario in which the PMDK provided better results. We are now checking more precisely the causes for the performance gain of PMDK with the skip list (`add` operations) and conducting an evaluation on more representative benchmarks (real cases), as well as adding comparisons with other available PM allocators.

5. Acknowledgments

This research has been supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grant numbers: 2018/15519-5, 2023/04969-8, 2023/04971-2.

References

Baldassin, A., Barreto, J., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7).

Bhandari, K., Chakrabarti, D. R., and Boehm, H.-J. (2016). Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694.

Cai, W. and Wen, e. (2020). Understanding and optimizing persistent memory allocation. In *Proc. of the ISMM*, page 60–73.

Izraelevitz, J. et al. (2019). Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714.

Jung, M. (2022). Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proc. of the 14th ACM Hotstorage*, page 45–51.

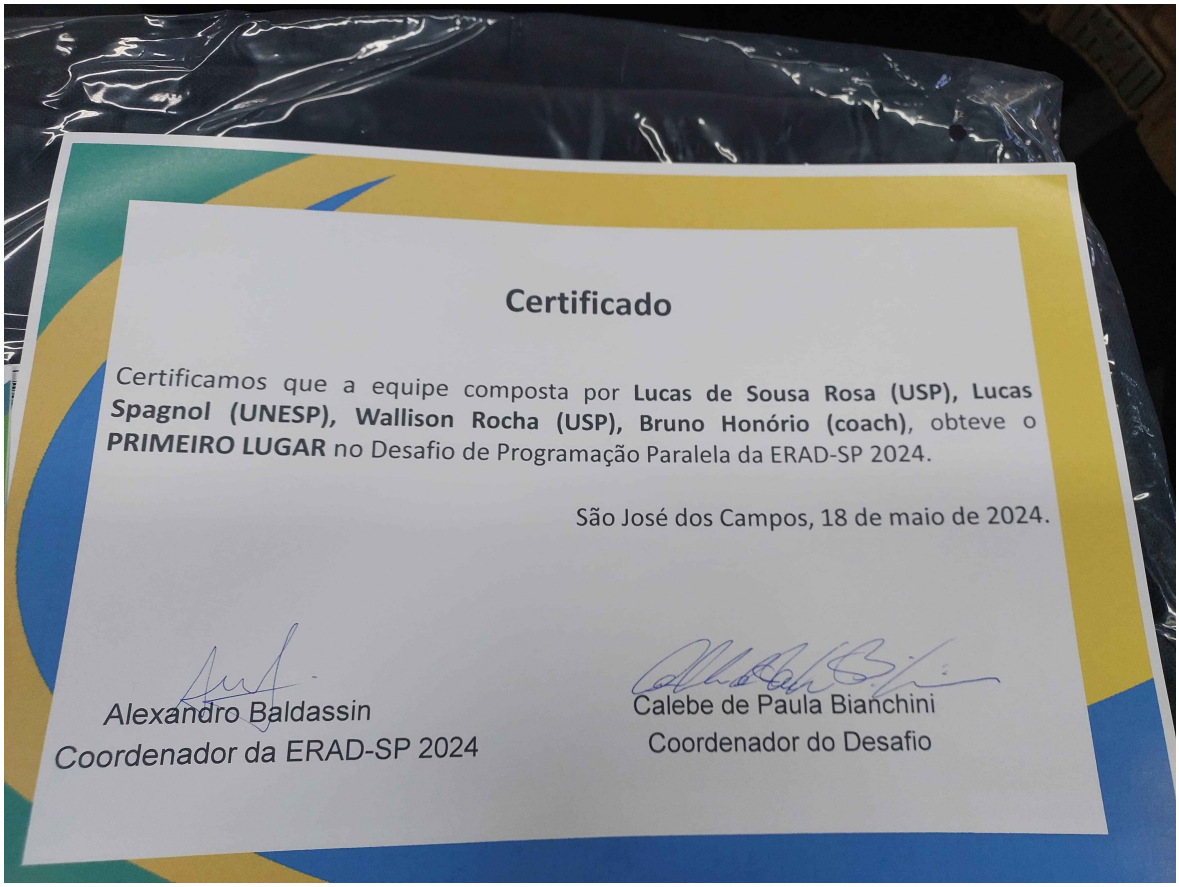
Leite, R. and Rocha, R. (2019). Lrmalloc: A modern and competitive lock-free dynamic memory allocator. In *Proc. of the 13th VECPAR*, pages 230–243.

Scargall, S. (2020). *Programming Persistent Memory - A Comprehensive Guide for Developers*. Apress, 1st edition.

Tiwari, D., Lee, S., Tuck, J., and Solihin, D. (2010). MMT:exploiting fine-grained parallelism in dynamic memory management. In *Proc. of IPDPS*, pages 1–12.

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

Yu, S., Xiao et al. (2015). WAlloc: An efficient wear-aware allocator for non-volatile main memory. In *Proc. of the 2015 IEEE IPCCC*, pages 1–8.



Certificado

Certificamos que a equipe composta por **Lucas de Sousa Rosa (USP), Lucas Spagnol (UNESP), Wallison Rocha (USP), Bruno Honório (coach)**, obteve o **PRIMEIRO LUGAR** no Desafio de Programação Paralela da ERAD-SP 2024.

São José dos Campos, 18 de maio de 2024.

Alexandro Baldassin
Coordenador da ERAD-SP 2024

Calebe de Paula Bianchini
Coordenador do Desafio

Papers You Are Reviewing: CTD

To accept the review, click on ✔, to decline click on 🚫. To fill out the review, click on 'Review' or 'TPC'. If review access is blocked the review link won't be available.

Type (Review)	#	Authors	Title (link to paper)	Assigned by	Discussion messages	Rebuttal	Status		Files		Due
Review	244316	João Pedro da Costa, Mario Dantas, José Maria David	*Uma Arquitetura de Edge-Fog-Cloud para Conciliar Interoperabilidade e Redução de Dados Focada em E-Health*	Ricardo Ferreira					Paper manuscript 1.8 MB		2024-09-06

Legend: notified confirmed declined reminded late delegated completed

Papers You Are Reviewing: Fórum de Iniciação Científica

To assign reviewers or delegate a paper review, you must first accept to review the paper, then click on review to assign/delegate. To accept the review, click on ✔, to decline click on 🚫. To fill out the review, click on 'Review' or 'TPC'. If review access is blocked the review link won't be available.

Type (Review)	#	Authors	Title (link to paper)	Assigned by	Discussion messages	Rebuttal	Status		Files		Due
Review	239679	Guilherme Arcencio, Helio Guardia	*Study and analysis of parallel N-body simulations*	Aleardo Manacero Jr.					Paper manuscript 0.2 MB	[Bibtex]	2024-04-26
Review	239939	Vitor Silva, Rodrigo Nunes, Helio Guardia	*Programação paralela com Unity: um estudo de caso usando threads em CPU e em GPU*	Aleardo Manacero Jr.					Paper manuscript 3.5 MB	[Bibtex]	2024-04-26

Legend: notified confirmed declined reminded late delegated completed

Papers You Are Reviewing: Iniciação Científica

To accept the review, click on ✔, to decline click on 🚫. To fill out the review, click on 'Review' or 'TPC'. If review access is blocked the review link won't be available.

Type (Review)	#	Authors	Title (link to paper)	Assigned by	Discussion messages	Rebuttal	Status		Files		Due
Review	231895	Yuri Freire, Hermes Senger	*Integrating CUDA memory management mechanisms for domain decomposition of an acoustic wave kernel implemented in OpenMP*	Alexandro Baldassin					Paper manuscript 0.3 MB	[Bibtex]	2023-06-22
Review	232017	Letícia Farias Machado, Claude Todonki, Hermes Senger	*A tool for profiling memory accesses locality on NUMA architectures*	Alexandro Baldassin					Paper manuscript 0.1 MB	[Bibtex]	2023-06-22

Legend: notified confirmed declined reminded late delegated completed

Investigando Fatores de Desempenho em Dispositivos Persistentes

Lucas Spagnol[1], **Bruno Honorio**[1], **Edgar Galvão**[1], **Emilio Francesquini**[2], **Alexandro Baldassin**[1]

[1]Universidade Estadual Paulista (UNESP) – Rio Claro, SP

{lucas.b.spagnol, bruno.honorio, edgar.galvao, alexandro.baldassin}@unesp.br

[2]Universidade Federal do ABC (UFABC) – Santo André, SP

e.francesquini@ufabc.edu.br

Abstract. *Persistent Memory is an emerging technology that combines the characteristics of DRAM and SSD, enabling its application in a variety of tasks. However, some issues may arise regarding its functionality and performance when tested on a machine equipped with it, especially with respect to NUMA devices and cache operations. In light of this, this paper presents some insights that seek to uncover how these factors impact the performance of persistent devices.*

Resumo. *A Memória Persistente é uma tecnologia emergente que combina as características da DRAM e do SSD, possibilitando sua aplicação em uma variedade de tarefas. No entanto, alguns reveses relacionados ao seu funcionamento e desempenho quando testada em uma máquina equipada com ela podem surgir, especialmente no que diz respeito às máquinas NUMA e às operações em cache. Diante disso, apresentamos algumas observações que buscam revelar como esses fatores impactam o desempenho de dispositivos persistentes.*

1. Introdução

A Memória Persistente (PM) representa uma inovação tecnológica que almeja combinar as propriedades da memória volátil, tais como a baixa latência e o endereçamento a byte, com os benefícios da persistência da memória não-volátil. Em 2018, a Intel lançou comercialmente a Intel Optane, fundamentada na tecnologia 3D Xpoint [Seltzer et al. 2018]. A partir disso, uma série de estudos foram conduzidos com o objetivo de analisar tanto o desempenho quanto a funcionalidade desta tecnologia.

Ao executar os *benchmarks* descritos no artigo [Islam and Dai 2023], um *framework* estado-da-arte para análise de grafos dinâmicos, observou-se que o SSD (*Solid State Drive*) exibia um desempenho superior ao da PM. Os resultados obtidos por nós mostraram o SSD alcançando um tempo de execução equivalente à metade do tempo registrado para a PM. Notou-se, posteriormente, que o principal fator por trás deste comportamento é o sistema de cache empregado pelo sistema operacional. Outro ponto importante nos experimentos refere-se ao uso de uma arquitetura NUMA (Non-Uniform Memory Access) com dois nós. Nossos resultados experimentais mostram que há uma diferença de desempenho de 15% a depender se o acesso é intra- ou inter-nós.

Este artigo procura elucidar alguns fatores chaves que devem ser considerados na análise de desempenho com memórias persistentes. O primeiro deles refere-se ao fator NUMA, já conhecido no âmbito de memórias voláteis, mas ainda não explorado em

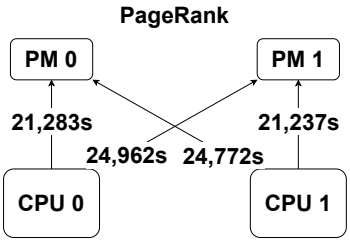


Figura 1. Tempo necessário para a execução do *Page Rank*, em segundos.

memórias persistentes. O segundo fator diz respeito ao esquema de cache utilizado pelo sistema operacional, o que pode mascarar o desempenho observado nos experimentos. Assim, pretende-se com esse breve estudo chamar a atenção dos pesquisadores que fazem experimentos com dispositivos persistentes sobre a importância de configurar corretamente o ambiente experimental.

2. Efeito NUMA em Memórias Persistentes

Antes de abordar o problema em questão, é essencial compreender o funcionamento da memória persistente. A máquina empregada para os testes é equipada com dois processadores da *Intel* e oito módulos de memória persistente *Optane DC*, também da *Intel*. A memória persistente é conectada ao barramento do processador, o mesmo utilizado para a DRAM. Dessa forma, os módulos de memória persistente estão distribuídos entre os barramentos dos processadores, com quatro módulos alocados para cada processador. Na *BIOS*, esses módulos são configurados para formar uma única unidade de memória, resultando em dois nós de memória persistente para o sistema operacional, os quais chamamos de $PM0$ e $PM1$.

Durante experimentos com o DGAP [Islam and Dai 2023], um framework para processamento de grafos dinâmicos que utiliza memória persistente, se notou um comportamento anômalo de acesso a essa memória e se levantou a suspeita que pudesse ser devido ao fator NUMA. Para verificar essa suposição, foi empregada a ferramenta *Taskset*[1], que permite selecionar quais *threads* do processador estarão disponíveis para o *benchmark*. Dessa forma, o teste foi realizado na CPU0, acessando ambas as memórias, e posteriormente na CPU1, também acessando ambas as memórias $PM0$ e $PM1$. Com essa abordagem, foi possível observar que, ao acessar a memória conectada ao barramento do outro processador, o desempenho diminuiu aproximadamente 15%, conforme ilustrado na Figura 1. Os testes foram feitos utilizando o *benchmark* de análise de grafos *Page Rank*. Essas observações revelam a importância do desenvolvimento de métodos eficientes para alocação de memória em máquinas NUMA com PM.

3. Desempenho do SSD vs. PM

Para abordar o problema do SSD estar mais rápido que a PM, a primeira ação executada consistiu na desativação do *DAX* na memória persistente. O *DAX* possibilita o acesso direto à memória persistente (PM).A desativação do DAX fez com o que sistema operacional utilizasse o mesmo esquema de bufferização (page caching) utilizado no SSD, aumentando o desempenho da PM. É importante observar que ao utilizar o esquema de

[1]https://man7.org/linux/man-pages/man1/taskset.1.html

bufferização, a propriedade de *durabilidade imediata* [Baldassin et al. 2021] da PM é perdida, ou seja, os últimos dados escritos não necessariamente terão sido persistidos.

No caso do SSD, o acesso geralmente é feito por uma API (*Application Programming Interface*) de arquivos. Essa API comumente realiza algum tipo de bufferização para acelerar o acesso aos dados, uma vez que este acesso é notavelmente mais lento do que o acesso a DRAM. Ou seja, para garantir a durabilidade imediata dos dados é necessário forçar que os dados armazenados no buffer sejam escritos no dispositivo.

Assim, para uma comparação justa entre os dispositivos, é necessário evitar que o SSD utilize a bufferização. Uma primeira alternativa empregada foi montar o sistema de arquivos da SSD com a opção *flag -sync*. Isso realmente fez com que as escritas fossem persistidas diretamente no disco, mas não alterou o comportamento das leituras (o tempo de execução com o DGAP permaneceu o mesmo, visto que ele basicamente só fazia leituras). Outra opção que adotada foi alterar a configuração do sistema operacional (Linux) para eliminar a bufferização de escritas, modificando as variáveis *vm.dirty_background_ratio* e *vm.dirty_ratio* do arquivo */etc/sysctl.conf*[2]. Essas variáveis definem a quantidade de memória DRAM que será destinada para a utilização da *page cache*.

Houve bastante dificuldade para evitar a bufferização das leituras realizadas pelo sistema operacional, uma vez que não há um procedimento bem documentado para isso. Uma alternativa encontrada foi a utilização do seguinte comando:

```
sudo sync; echo 1 > /proc/sys/vm/drop_caches
```

forçando o sistema operacional a limpar toda a bufferização realizada. Portanto este comando é necessário caso seja necessário também levar em consideração o tempo de acesso de leituras ao SSD. Note, no entanto, que essa alternativa não é uma solução completa, pois ela exige executar o comando para limpar a bufferização regularmente. Para fins práticos, é suficiente desabilitar a bufferização apenas de escrita.

4. Experimentos utilizando a API de arquivos

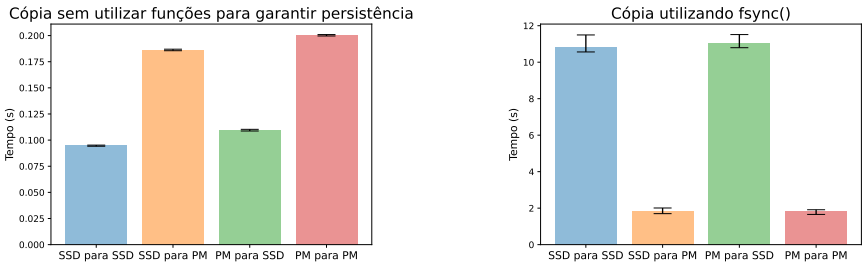
Para investigar a discussão anterior em um cenário específico, foi desenvolvido um *benchmark* que copia um arquivo de um diretório para outro em blocos. Este *benchmark* foi implementado utilizando as funções de sistema de arquivos da linguagem C, especificamente, `fopen(), fread(), fwrite()`. Durante a cópia do arquivo, o programa registra o tempo decorrido e exibe este valor ao final do processo.

Ao empregar exclusivamente as funções mencionadas, observou-se (veja Figura 2a) que o SSD apresentou um desempenho superior ao da PM. Esta superioridade do SSD pode ser atribuída ao fato de o sistema operacional realizar a bufferização de escrita na DRAM e, posteriormente, salvar os arquivos no SSD. Em contraste, na PM, devido à ativação do DAX, as operações de escrita são executadas diretamente na PM.

Para investigar métodos de evitar o uso da bufferização sem a necessidade de alterar configurações na máquina, foram testadas três chamadas da API: `fflush()` e `fsync()`[3]. O objetivo era determinar se essas funções influenciariam os resultados obtidos. A função `fflush()` limpa o *buffer* da aplicação (no nível da `(glibc)`), enviando para o sistema operacional os dados que serão escritos na memória. Por outro lado,

[2]https://man7.org/linux/man-pages/man5/sysctl.conf.5.html

[3]É possível encontrar os *benchmarks* no repositório:https://github.com/LucasBastelli/block-copy.git



(a) Cópia comum. **(b) fsync().**

Figura 2. Tempo(s) de cópia de um arquivo de 100MB entre os dispositivos.

`fsync()` faz uma chamada ao sistema operacional para forçar a escrita dos dados no dispositivo.

São apresentados os resultados experimentais da comparação dos tempos necessários para a cópia de arquivos de 100MB entre os dispositivos de armazenamento utilizando a função `fsync()`. Cada experimento foi repetido 25 vezes para o cálculo da média. O cálculo do intervalo de confiança de 95% foi efetuado com a técnica de Bootstrap com 10.000 reamostragens.

Em primeiro lugar, nossos experimentos mostraram que a chamada `fflush()` não alterou os resultados. Ou seja, a bufferização em nível de aplicação não parece estar influenciando no ganho de desempenho. No entanto, a chamada `fsync()` resultou em um aumento no tempo de execução, tornando o SSD significativamente mais lento quando comparado à memória persistente (PM), como é possível observar na Figura 2b. Também foram realizados testes similares usando a API de mapeamento de memória (`mmap`) e a chamada `msync()`, com resultados similares ao apresentado na Figura 2b. Estes resultados mostram que é necessário garantir a escrita dos dados na SSD quando o desempenho é comparado com a PM.

5. Conclusão

Observou-se que o efeito da bufferização no SSD resulta em um desempenho aparentemente superior ao real, o que pode levar à percepção inicial de que a Memória Persistente é "lenta". Portanto, é essencial adotar métodos que evitem o uso da bufferização, uma vez que ela impede a persistência dos dados escritos. Além disso, notou-se que o efeito NUMA influencia o desempenho da Memória Persistente, tornando-se necessário minimizar seu uso quando o objetivo é otimizar o desempenho.

Agradecimentos. Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos nº 2023/04969-8, 2023/04971-2, 2018/15519-5 e 2023/10128-6 pelo apoio a este trabalho.

Referências

Baldassin, A., Barreto, J. a., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7).

Islam, A. A. R. and Dai, D. (2023). Dgap: Efficient dynamic graph analysis on persistent memory. In *Proc. of the SC'23*.

Seltzer, M., Marathe, V., and Byan, S. (2018). An NVM Carol: Visions of NVM Past, Present, and Future. In *Proc. of the ICDE'18*, pages 15–23.

Avaliando o Impacto da Alocação de Memória em Sistemas com Memória Persistente

Otávio Scarparo Souza, Bruno Honorio, Alexandro Baldassin

[1]Universidade Estadual Paulista (UNESP) – Rio Claro, SP

{otavio.scarparo-souza, bruno.honorio, alexandro.baldassin}@unesp.br

Abstract. *Persistent Memory allocators can improve the performance of applications that use them. However, due to the differences between Persistent Memory and traditional, volatile memory, they must employ error handling and mitigation techniques. Features such as the presence of NUMA and the techniques aforementioned may impact the performance of those allocators. In this paper we are going to present how those factors may impact memory allocation.*

Resumo. *Alocadores de Memória Persistente podem aumentar o desempenho de aplicações que os utilizam. Porém, devido às diferenças entre a Memória Persistente e a convencional, eles devem apresentar mecanismos de tratamento e redução de erros. Fatores como a presença do NUMA e os mecanismos citados podem causar grandes impactos no desempenho dos alocadores. Neste artigo, será mostrado como esses fatores vêm a impactar a alocação de memória.*

1. Introdução

A Memória Persistente (PM) permite que aplicações possam manter estruturas de dados diretamente na memória. Com isso, uma aplicação pode acessar diretamente dados sem a necessidade de um sistema de arquivos, como acontece atualmente com dispositivos de armazenamento externo, como SSD, aumentando assim a eficiência e o seu desempenho [Baldassin et al. 2021]. Porém, devem haver mecanismos de manutenção da integridade delas caso ocorram falhas, que podem causar problemas como inconsistências, metadados irrecuperáveis, recuperações demoradas e dealocação insegura. Em particular, as estruturas de dados internas do alocador de memória devem ser mantidas intactas, preferencialmente como elas estavam logo antes da ocorrência de um *crash*. Assim, alocadores de memória para PM funcionam de maneiras diferentes dos convencionais. Estratégias como coleta de lixo podem ser aplicadas para a mitigação de erros. Neste trabalho, será mostrado, por meio de testes, que diferentes estratégias e o NUMA do processador afetam o desempenho da alocação de memória em Memória Persistente.

Devido a diferentes estratégias de alocação e dealocação, alocadores de memória persistente podem apresentar resultados variados que podem vir a impactar o desempenho de suas operações. Este artigo faz uma avaliação preliminar do impacto no desempenho causado por três alocadores persistentes: Ralloc [Cai et al. 2020], Makalu [Bhandari et al. 2016] e PMDK [Scargall 2020]. Em particular, notamos que o alocador Ralloc possui o melhor desempenho mas pode sofrer com o fator NUMA, ao contrário dos outros dois.

2. Alocadores Persistentes

Ao contrário dos alocadores convencionais (para memória volátil), os alocadores persistentes enfrentam outros tipos de problemas. Como exemplo, assuma que a aplicação fez uma chamada para alocar memória persistente como se segue:

```
PersistentPointer = p_malloc(4092);
```

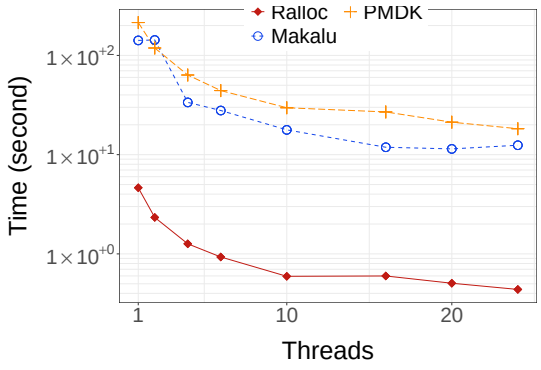
O alocador deve, primeiramente, alocar uma região de memória persistente, atualizar seus metadados (também persistentes) para refletir a nova alocação, e então retornar o endereço alocado para a aplicação. Considere, no entanto, que haja um *crash* (e.g., queda de energia) antes da aplicação armazenar o endereço do bloco alocado. Neste caso, teríamos um vazamento de memória persistente, ou seja, a região de memória aparece como alocada para o alocador, embora não tenha sido utilizada pela aplicação.

Uma das formas de resolver o problema é executar o exemplo acima dentro de uma transação. Nesse caso, a alocação e a atribuição do ponteiro vão ser executados de forma atômica. A outra forma é deixar o vazamento acontecer e o alocador usar algum algoritmo de coleta de lixo para recuperar a área alocada, mas não usada. No resto desta seção estão brevemente descritos os principais alocadores que utilizamos neste artigo.

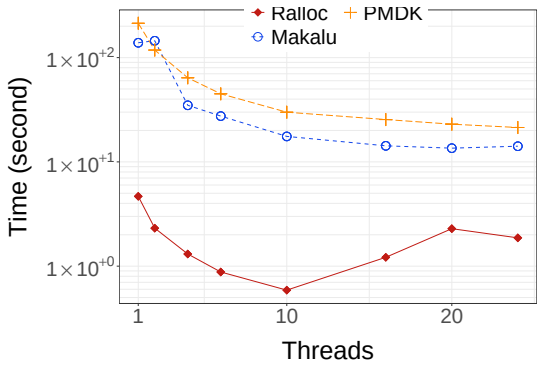
Makalu: É um alocador que utiliza coleta de lixo para prevenir vazamentos de memória, garantindo a integridade da estrutura da heap persistente. A coleta ocorre em dois momentos: i) quando uma aplicação está ativa e pode modificar metadados e ii) quando, após uma falha, os dados são restaurados e voltam para um estado consistente antes da ocorrência do problema. Cada bloco no Makalu tem um cebeçalho na memória persistente que possui seu endereço, tamanho e uma flag que indica se está em uso ou não. Cada objeto em um bloco possui um bit correspondente que indica se está alocado ou se foi adicionado na *free list* de alguma thread com a intenção de alocá-lo. O alocador considera como lixo tudo aquilo que não pode ser acessado pelas *persistent roots* da PM. Depois de uma falha, os metadados são restaurados usando *logging* e os bits indicadores com a coleta de lixo, além de serem desmarcados.

Ralloc: O alocador, assim como o Makalu, utiliza coleta de lixo após erros de execução. Ele busca garantir que, após um erro, só blocos que estavam sendo utilizados são alocados. Ao alocar e dealocar, só há sincronização entre caches da thread sendo utilizada quando é necessário. A maioria dos metadados importantes só está presente na memória não permanente, sendo reconstruídos após um *crash* total do sistema. Depois de um *crash* parcial, a memória que foi vazada é recuperada com coleta de lixo. Para melhorar o desempenho, *filter functions* são necessárias para enumerar as referências presentes em cada bloco, para serem usadas durante a fase de *trace* da coleta de lixo. Para recuperar metadados após a coleta de lixo, o tamanho de cada bloco é salvo na memória persistente, sendo possível saber o quanto de memória é alcançável.

PMDK: Este alocador, diferentemente dos descritos anteriormente, utiliza transações para redução de erros. As operações de alocação e dealocação de memória ocorrrem atomicamente: O bloco alocado é adicionado persistentemente a um endereço específico. Na dealocação, o ponteiro persistente que aponta para um bloco é destruído e o bloco volta para a lista de blocos livres. Estas operações devem ser feitas usando transações para garantir a atomicidade.



(a) Limitado a um único NUMA node.



(b) Distribuído entre os 2 NUMA nodes.

Figura 1. Resultados considerando duas configurações NUMA com o benchmark `shbench`.

3. Avaliação Experimental

Para a avaliação quantitativa mostrada nesta seção, utilizamos um sistema com o Linux 5.4.0 (Ubuntu 20.04.6 LTS), uma máquina com dois processadores Intel Xeon Gold 5317 que possui 24 cores físicos e 48 threads. É importante levar em consideração que pelo fato da máquina ser NUMA, tendo dois nodos diferentes, o desempenho das aplicações que utilizam a memória persistente pode apresentar variações de acordo com a distância entre o core emitindo a instrução de acesso à memória e o módulo físico dessa memória.

O *benchmark* utilizado foi o `shbench`, em que *threads* alocam e dealocam objetos que variam de 64 a 400 bytes múltiplas vezes. Os experimentos foram executados 3 vezes, e o resultado apresentado na Figura 1 é a média desses valores. Foram utilizados os alocadores de memória persistente Makalu, Ralloc e PMDK, como apresentados anteriormente.

A Figura 1a mostra o tempo de execução conforme o número de threads é aumentado. Neste cenário, as 24 threads estão todas alocadas em um único nodo NUMA. Nota-se claramente que o Ralloc apresenta um desempenho bem superior quando compa-

rado aos outros dois alocadores. No melhor caso, com 24 threads, ficou 27x mais rápido que o Makalu e 40x mais rápido que o PMDK. Isso mostra a efetividade da estratégia adotada pelo Ralloc com o uso do coletor de lixo, embora deve-se ter em mente que o efeito do coletor não está sendo medido explicitamente nesse gráfico (deixamos para trabalhos futuros).

Já a Figura 1b mostra o resultado das execuções ao espalhar as 24 threads entre os dois nodos NUMA, primeiro preenchendo todos os cores de um nodo NUMA (12) e depois o outro nodo. Neste caso nota-se que, apesar do Ralloc ainda continuar com melhor desempenho, ele sofre com o fator NUMA pois seu desempenho piora a partir de 12 threads. Em particular, com 20 threads seu desempenho fica 4.8x mais lento quando comparado à melhor versão com apenas um nodo NUMA.

Os resultados apontam para a necessidade de levar em consideração à organização da memória e a alocação das threads quando uma máquina NUMA é utilizada com o alocador Ralloc. Não foi notado o mesmo comportamento com os outros dois alocadores.

4. Conclusão

Neste artigo foram apresentados problemas que podem ocorrer durante a alocação e dealocação na PM e como diferentes alocadores de memória tentam mitigar erros usando diferentes métodos, como coleta de lixo e transações para garantir a atomicidade, dado que a Memória Persistente possui um funcionamento diferente da convencional. Estas estratégias podem vir a ter impactos significativos no desempenho da alocação e dealocação, com a coleta de lixo possuindo a tendência de apresentar um desempenho melhor que as transações. Por outro lado, o fator NUMA da máquina utilizada causou diferenças no desempenho. Em geral, usando dois nodos NUMA, o desempenho foi pior que ao usar apenas um. Conclui-se, por fim, que o alocador Ralloc é mais sensível ao NUMA que os outros, que apresentaram resultados levemente piores, enquanto o Ralloc apresentou pioras significativas de desempenho.

Agradecimentos. Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos nº 2018/15519-5, 2023/05032-0 pelo apoio a este trabalho.

Referências

Baldassin, A., Barreto, J. a., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7).

Bhandari, K., Chakrabarti, D. R., and Boehm, H.-J. (2016). Makalu: fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694.

Cai, W., Wen, H., Beadle, H. A., Kjellqvist, C., Hedayati, M., and Scott, M. L. (2020). Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2020, page 60–73, New York, NY, USA. Association for Computing Machinery.

Scargall, S. (2020). *Programming Persistent Memory - A Comprehensive Guide for Developers*. Apress, 1st edition.

Uma Revisão Sistemática sobre Estruturas de Dados em Dispositivos Persistentes Contemporâneos

Lucas Spagnol[1], **Bruno Honorio**[1], **Alexandro Baldassin**[1], **Emilio Francesquini**[2]

[1]Universidade Estadual Paulista (UNESP) – Rio Claro, SP

{lucas.b.spagnol, bruno.honorio, alexandro.baldassin}@unesp.br

[2]Universidade Federal do ABC (UFABC) – Santo André, SP

e.francesquini@ufabc.edu.br

Resumo. *Memória persistente (PM) é uma tecnologia emergente que combina o acesso rápido aos dados com endereçamento por byte, além de oferecer grande capacidade de armazenamento persistente. Devido a essas características, a PM tem ganhado cada vez mais atenção, com diversos estudos sendo realizados para explorar seu uso e desempenho. Paralelamente, trabalhos sobre estruturas de dados (ED) têm evoluído para maximizar o aproveitamento das PMs. Isso inclui o desenvolvimento de EDs adaptadas para a persistência e transações atômicas. Este artigo revisa os trabalhos mais relevantes na área de memória persistente com foco em estruturas de dados, destacando aqueles que desenvolvem ou adaptam tais estruturas para uso na PM. Nossos estudos identificaram as principais características necessárias para o desenvolvimento de uma estrutura de dados voltada para PM, visando aprimorar o desempenho ao reduzir o número de operações de escrita por meio de otimizações de código e do uso de memória DRAM como cache. Além disso, enfatizamos a importância da garantia de persistência, uma vez que a memória persistente mantém os dados mesmo após o desligamento do sistema.*

1. Introdução

Computadores convencionalmente são divididos em armazenamento primário (memória DRAM) e secundário (HDs e SSDs). O armazenamento primário permite ao computador acessar rapidamente os dados, porém com pouco espaço de armazenamento e de forma volátil, ou seja, quando o computador é desligado, seja por falta de energia, erros, ou por comando do usuário, todos os dados armazenados nele são perdidos. Já no armazenamento secundário, é possível a retenção de dados mesmo após o desligamento do sistema. Em contrapartida, ao contrário da DRAM, esses dispositivos apresentam largura de banda pequena e tempos de resposta elevados quando comparados a ela, tornando-os inadequados para operações que exigem rapidez, o que os torna mais apropriados para armazenamento de dados [8].

Os recentes avanços na indústria de semicondutores viabilizaram o desenvolvimento de tecnologias inovadoras para armazenamento persistente [1]. Essas tecnologias unem as características da DRAM e dos HDs e SSDs, como altas taxas de transferência de dados e granularidade de byte, juntamente com a grande capacidade de armazenamento e persistência dos dados. Isso possibilita a realização de operações na Memória Persistente (PM) de forma rápida e persistente.

As estruturas de dados (ED) desempenham um papel fundamental nos sistemas computacionais. Elas permitem armazenar e processar informações de maneira eficiente

e organizada, sendo utilizadas desde caches para acelerar o acesso a dados e arquivos até para armazenar informações em bancos de dados. Com o avanço da memória persistente, torna-se necessário desenvolver ou adaptar as estruturas de dados para que elas possam aproveitar a melhoria de desempenho além, é claro, da persistência.

Contudo, ao integrar estruturas de dados com persistência, é imprescindível adotar medidas preventivas para assegurar a consistência dos dados, evitando assim a perda de dados e vazamentos de memória. Comumente, empregam-se transações para garantir a correta gravação dos dados [7]. Este trabalho tem como objetivo apresentar um levantamento atual da pesquisa sobre o desenvolvimento das estruturas de dados voltadas para a memória persistente. Busca-se responder questões sobre essas estruturas de dados em relação ao seu uso e funcionalidade, por meio de uma revisão sistemática de artigos publicados na área nos últimos 5 anos.

A análise dos trabalhos encontrados possibilita a elaboração de uma lista de requisitos essenciais que uma estrutura deve possuir para operar de maneira otimizada na memória persistente. Essas informações se mostram valiosas ao se considerar o desenvolvimento de uma nova estrutura de dados voltadas à PM. O objetivo principal deste artigo é identificar e empregar os requisitos essenciais na elaboração de estruturas de dados para memória persistente. Como contribuição, este artigo apresenta as estruturas de dados mais utilizadas na área de PM, bem como as otimizações aplicadas para melhorar o desempenho neste tipo de dispositivo. Além disso, são determinados os principais fatores que ainda não foram explorados na PM, destacando áreas potenciais para futuras pesquisas e desenvolvimento.

2. Contexto

As estruturas de dados exercem um papel crucial na computação, uma vez que possibilitam a representação organizada das informações na memória dos computadores. Algumas dessas estruturas, como é o caso da *hash table*, proporcionam ainda uma maior eficiência na busca de dados. Com o progresso das memórias persistentes que possuem endereçamento a *byte*, torna-se imprescindível a adaptação das estruturas de dados para essas memórias, de modo que possam ser utilizadas de maneira eficaz e com a capacidade de persistência. Diferentemente das estruturas empregadas na memória DRAM, as estruturas adaptadas para as memórias persistentes devem ser modificadas para serem resistentes a falhas.

Considere, por exemplo, uma lista encadeada que contém os valores 1, 3, 7 e 9, conforme ilustrado na Figura 1. Suponha que desejamos inserir o valor 5 nessa lista. Durante esse processo de inserção, se alguma falha ocorresse (por exemplo, queda de energia), poderíamos enfrentar duas situações problemáticas. A primeira situação (A) é a possibilidade do valor ser alocado na memória mas, por alguma falha (como falta de energia), não ser efetivamente inserido na lista. Isso resultaria em um vazamento de memória, onde um espaço ficará ocupado mas sem uso real. A segunda situação (B) ocorre quando o nó contendo o valor 3 é atualizado para apontar para o novo nó 5, mas o ponteiro do novo nó 5 não é ajustado para apontar para o nó com o valor 7 (novamente, alguma falha pode acontecer antes desta última escrita). Isso resultaria na perda de todos os dados subsequentes na lista, uma vez que o encadeamento dos nós é interrompido. Essas situações destacam a importância de garantir a integridade dos dados durante as operações.

A presente pesquisa não identificou revisões sistemáticas anteriores que abordas-

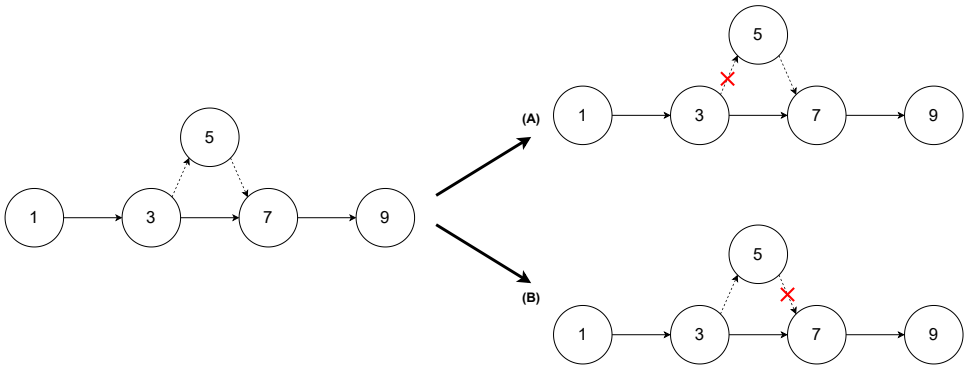


Figura 1. Casos de falhas na inserção de um elemento em uma lista encadeada.

sem questões como: quais são as estruturas de dados mais utilizadas, quantos trabalhos disponibilizam código aberto e quais otimizações foram implementadas para melhorar o desempenho na memória persistente. Essas questões são detalhadas na Seção 4. Este artigo analisa os aspectos mais recorrentes em trabalhos sobre estruturas de dados na memória persistente, identificando e organizando aqueles menos explorados. Dessa forma, são reveladas áreas com potencial significativo para estudos futuros.

3. Metodologia

Foi utilizado neste trabalho o protocolo desenvolvido por Barbara Kitchenham [13], que enfatiza a precisão metodológica na criação de uma revisão. A revisão se concentra em investigar trabalhos publicados nos últimos 5 anos sobre estruturas de dados aplicadas à memória persistente com endereçamento por *byte*. Esses trabalhos buscam criar uma nova estrutura ou adaptar uma já existente para maximizar o desempenho da PM. Para isso, a pesquisa se baseou em buscar os trabalhos nas principais bases científicas: ACM, IEEE e Scopus. A escolha dessas bases de dados deve-se à sua ampla utilização em pesquisas nas áreas de ciência da computação e engenharia, além de possuírem um extenso acervo de trabalhos. A seguir é apresentada a metodologia utilizada para a pesquisa.

3.1. Pergunta Chave da pesquisa

A principal questão que essa pesquisa pretende responder é:

Qual o estado atual da pesquisa de estruturas de dados para memória persistente?

Assim, procura-se conhecer mais sobre as estruturas de dados em desenvolvimento para memória persistente, realizando uma análise aprofundada de informações relevantes sobre elas. Como resultado, foi possível identificar fatores ainda não explorados ou pouco explorados, como estruturas de dados menos utilizadas e técnicas que podem melhorar o desempenho na memória persistente, visando maximizar o aproveitamento da memória persistente, combinando alto desempenho com persistência.

3.2. Estratégia de Busca

O primeiro passo para o processo de revisão consistiu na escolha das palavras-chave que retornariam estudos sobre o tema desejado. Assim, foi utilizada a seguinte expressão:

```
("data structure") AND ("persistent memory" OR "Storage Class
   Memory" OR "Non-Volatile Memory" OR "Non-Volatile RAM" OR "
   Byte-addressable Persistent RAM" OR "Non-Volatile Main Memory
   " OR "Non-Volatile DIMM" OR "Non-Volatile Byte-addressable
   Memory")
```

Inicialmente, a busca foi realizada por palavras-chave em todo o corpo do texto. No entanto, isso resultou em resultados que não estavam relacionados ao nosso tema, onde as estruturas de dados eram utilizadas em contextos diversos que não se alinhavam ao tema pesquisado. Em virtude disso, a busca foi restringida apenas ao resumo dos artigos. Essa busca foi feita nas seguintes bases, com os respectivos resultados:

- ACM - total de 24 trabalhos
- IEEE - total de 16 trabalhos
- Scopus - total de 146 trabalhos

A pesquisa foi realizada utilizando o sistema de busca integrado à própria base, restringindo os resultados a artigos publicados nos últimos 5 anos. Tal restrição visa entender as tendências atuais de pesquisa, identificando as áreas que estão recebendo maior atenção e esforço.

3.3. Critérios de Inclusão e Exclusão

Para refinar a pesquisa e alcançar os resultados desejados, é necessário estabelecer critérios para a inclusão e exclusão de estudos. A seguir são apresentados ambos critérios.

Tabela 1. Critérios de inclusão e exclusão.

Critério	ID	Descrição
Inclusão	I1	O trabalho deve desenvolver ou adaptar uma estrutura de dados para memória persistente.
	I2	O trabalho deve estar em português ou inglês.
Exclusão	E1	O trabalho apenas analisou o desempenho de uma estrutura de dados.
	E2	O trabalho se concentrou na criação de um arcabouço que utiliza estrutura de dados.
	E3	O trabalho é uma revisão literária.
	E4	O trabalho não foca em memória persistente com endereçamento por *byte*.
	E5	O trabalho não toma cuidado com a persistência dos dados.
	E6	O trabalho deve possuir DOI.

3.3.1. Explicação dos critérios

- **I1 e E1:** Para inclusão no trabalho, é necessário que o trabalho adapte ou desenvolva uma estrutura de dados específica para uso em memória persistente. Trabalhos que apenas testam o desempenho de estruturas de dados, previamente desenvolvidas em outros estudos, na memória persistente, não são de nosso interesse.

- **I2:** Para ser considerado neste trabalho, o trabalho deve estar redigido em inglês ou português, a fim de garantir sua compreensão.
- **E2:** Para ser considerado neste trabalho, é imprescindível que o trabalho tenha como foco principal as estruturas de dados. Trabalhos que se concentram em arcabouços que modificam as estruturas de dados, colocando estas últimas a um papel secundário, não são o objetivo deste artigo.
- **E3:** Artigos que se limitam a revisar trabalhos sobre um determinado tema não são aceitos neste trabalho, uma vez que não contribuem para o desenvolvimento de estruturas de dados para memória persistente.
- **E4:** Para ser considerado neste trabalho, é essencial que o trabalho desenvolva estruturas de dados especificamente para memórias persistentes com endereçamento por *byte*, em vez de memórias persistentes em geral, como SSDs e HDs.
- **E5:** Ao programar para memória persistente é preciso adotar precauções específicas para evitar a perda de dados ou a geração de arquivos corrompidos. No caso de uma falha da máquina, há o risco de perder todos os dados da memória. No exemplo da lista encadeada já visto anteriormente, a perda de um nó pode tornar todos os nós subsequentes inacessíveis, eliminando assim as vantagens do uso da PM. Existem bibliotecas, como a *Persistent Memory Development Kit* (PMDK), que oferecem transações para garantir a persistência dos dados. Caso o estudo em questão não disponha de um sistema que assegure a persistência dos dados, este será descartado.
- **E6:** A necessidade do DOI se deve ao fato de facilitar a localização dos trabalhos, além de permitir a filtragem de *conference reviews* da pesquisa, que não são os trabalhos originais.

3.4. Seleção dos Estudos

A busca inicial realizada em Outubro de 2023 resultou em um total de 186 artigos. No entanto, 15 desses artigos foram removidos por não possuírem um Identificador de Objeto Digital (DOI), restando assim 171 artigos.

Devido ao grande volume de artigos, foi necessário adotar medidas adicionais para restringir a seleção. A primeira etapa foi a remoção de artigos duplicados, utilizando o DOI como referência, o que resultou na exclusão de 40 artigos. Após essa etapa inicial de filtragem, todos os títulos e resumos foram cuidadosamente analisados para refinar ainda mais a seleção. Optou-se por selecionar apenas os estudos que estavam diretamente alinhados ao tema principal da pesquisa. Com isso, o número total de artigos selecionados foi reduzido para 77. Na etapa final, procedeu-se à leitura integral dos artigos restantes. Aqueles que não se enquadravam no tema proposto foram excluídos. Após a conclusão desta etapa, restaram 25 trabalhos para análise. Os passos citados estão ilustrados na Figura 2.

4. Extração de Dados

Subquestões foram elaboradas para aprofundar a compreensão do cenário atual relativo ao tema. Essas perguntas buscam detalhar características importantes que impactam tanto o desempenho quanto o funcionamento das mesmas. A leitura completa dos artigos permitiu responder a essas subquestões. As subquestões formuladas foram:

- **Q1 - Qual a linguagem mais utilizada na implementação das estruturas de dados?**
  A identificação da linguagem de programação mais frequentemente utilizada na

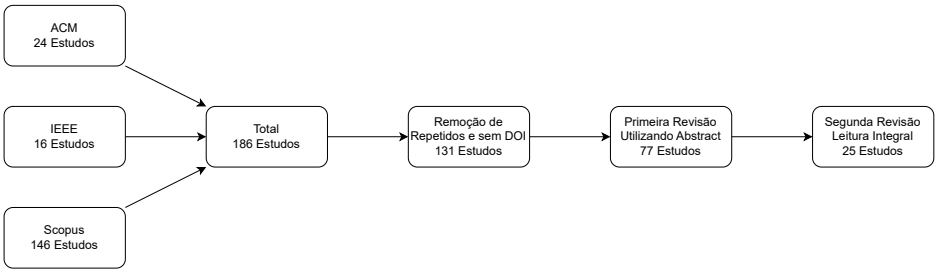


Figura 2. Etapas de Seleção.

construção de estruturas de dados para a memória persistente é fundamental, pois permite determinar qual linguagem é mais adequada para essa finalidade, oferecendo maior controle sobre a PM.

- **Q2 - Quais estruturas de dados são mais utilizadas?**
  Estruturas como árvores e *hash tables* possibilitam buscas significativamente mais rápidas, uma vez que apresentam menor complexidade, quando comparadas a estruturas como filas e listas. A compreensão da estrutura de dados mais empregada possibilita a identificação daquela que otimiza o desempenho da memória persistente.
- **Q3 - Possui código aberto?**
  O código aberto nos oferece a possibilidade de estudar a maneira como a estrutura de dados foi implementada na memória persistente, além de permitir a identificação das bibliotecas utilizadas. Isso facilita a replicação dos experimentos e aumenta a confiabilidade do trabalho.
- **Q4 - Usou sistema simulado para fazer avaliação?**
  Nem todos os estudos dispõem de memória persistente com endereçamento por *byte* para testar as estruturas de dados. Ao utilizar sistemas simulados, pode haver uma perda de precisão nos resultados, mesmo que seja adicionada latência na memória durante a simulação. Quando se simula um disco na DRAM sem a inclusão de latência, o desempenho tende a ser significativamente superior em comparação com qualquer tipo de memória persistente. Portanto, seria relevante determinar a quantidade desses que recorreram a um sistema simulado para avaliar o desempenho.
- **Q5 - Utiliza DRAM como mecanismo para otimização?**
  Para otimizar o acesso à memória persistente, alguns estudos adotam uma abordagem híbrida de armazenamento, estabelecendo um arquivo de cache na memória DRAM. Isso contribui para a redução do número de leituras na PM, aumentando o desempenho. É crucial determinar se foi empregada alguma otimização desse tipo devido às características específicas da memória persistente, como a latência elevada em operações de escrita. Assim, é possível compreender se os estudos utilizam a DRAM para aprimorar algumas dessas características.
- **Q6 - Levou em conta alguma característica específica do dispositivo para otimização?**
  Alguns estudos otimizam as estruturas para minimizar o número de leituras ou escritas realizadas na memória, em comparação com a estrutura convencional implementada na memória DRAM. Ao analisar essas otimizações, torna-se possível examinar as características específicas do dispositivo que devem ser consideradas

durante a programação para esse dispositivo. Essa consideração é imprescindível ao buscar desenvolver uma ED que maximize o desempenho da PM.

5. Resultados

Esta seção apresenta os resultados obtidos a partir das perguntas anteriores, após uma análise detalhada dos artigos selecionados.

5.1. Q1 - Qual a linguagem mais utilizada na implementação das estruturas de dados?

Em nossa análise dos artigos, observamos que 14 deles empregaram C++, uma linguagem de programação conhecida por oferecer um controle melhor sobre a memória. Isso sugere uma preferência potencial por essa linguagem em pesquisas que exigem um gerenciamento de memória mais refinado. No entanto, é importante notar que a linguagem de programação adotada não foi declarada em outros 9 artigos analisados, o que pode indicar uma variedade de abordagens, visto que os mesmos não disponibilizaram o código fonte. Além disso, dois artigos utilizaram C, uma linguagem que, assim como o C++, permite um maior controle da memória. Isso reforça a ideia de que o controle preciso da memória é um fator importante considerado pelos pesquisadores. Os trabalhos foram classificados de acordo com a linguagem de programação utilizada na Tabela 2.

Linguagem	**Trabalhos**
C++	[16, 21, 2, 29, 26, 17, 24, 18, 11, 14, 15, 5, 4, 23]
C	[20, 30]
Não declarado	[6, 9, 19, 22, 27, 28, 10, 25, 12]

Tabela 2. Linguagens utilizadas nos artigos (Q1).

5.2. Q2 - Quais estruturas de dados são mais utilizadas?

Em nossa análise, descobrimos que as estruturas mais empregadas foram o *hash table* e a árvore, cada uma sendo mencionada em 10 artigos. Isso pode indicar uma preferência por essas estruturas devido à sua eficiência em operações comuns como adicionar, remover ou procurar um dado, resultando em um desempenho superior. No entanto, é importante considerar que a escolha da estrutura de dados pode variar dependendo do contexto específico do problema que está sendo resolvido. Além dessas, o grafo foi utilizado em dois artigos, o que pode refletir seu uso comum em aplicações de inteligência artificial. Outras estruturas como lista encadeada, fila e *skip-List* foram mencionadas em um artigo cada, sugerindo que elas podem ser aplicadas em contextos mais específicos ou nichos de pesquisa. Essa variedade de estruturas utilizadas destaca a importância de escolher a estrutura de dados mais adequada para cada aplicação específica. A Tabela 3 agrupa os trabalhos de acordo com a estrutura de dados utilizada.

5.3. Q3 - Possui código aberto?

Durante a análise dos artigos selecionados para este artigo, observou-se que a disponibilização do código-fonte não é uma prática comum. De fato, dos 25 artigos analisados, constatou-se que 15 deles não forneceram acesso direto ao código-fonte. No entanto, é importante destacar que alguns desses artigos compartilharam suas metodologias por meio da apresentação de trechos de códigos na forma de pseudo-código.

Estruturas	**Trabalhos**
Grafo	[17], [4]
Hash table	[16], [27], [30], [19], [28], [20], [29], [9], [14], [10]
Lista encadeada	[11]
Fila	[6]
Skip-list	[5]
Árvore	[21], [2], [26], [24], [18], [22], [15], [25], [12], [23]

Tabela 3. Artigos por estrutura de dados (Q2).

A disponibilização do código-fonte é uma prática que traz inúmeros benefícios para a comunidade científica. Ela permite uma análise mais aprofundada das técnicas empregadas, oferece a possibilidade de testar e adaptar essas técnicas para outros programas e aumenta a credibilidade do estudo ao permitir a repetição dos experimentos. Para os propósitos deste trabalho, consideramos como código aberto apenas os artigos que incluíram um link direto para o repositório no corpo do texto. Optamos por não realizar buscas adicionais por repositórios em plataformas externas, como o Google, para manter o foco na disposição direta dos autores em compartilhar seus códigos. Na Tabela 4, é possível consultar quais trabalhos possuem código aberto.

Código Fechado	[18, 22, 15, 25, 12, 23, 16, 27, 19, 28, 29, 9, 10, 6, 11]
Código Aberto	[30, 21, 2, 20, 26, 17, 24, 14, 5, 4]

Tabela 4. Artigos com código aberto e código fechado (Q3).

5.4. Q4 - Usou sistema simulado para fazer avaliação?

Entre os estudos selecionados para esta análise, observou-se que 15 optaram por testar o desempenho das estruturas de dados diretamente na memória persistente, utilizando a tecnologia Intel Optane. Dos 10 estudos restantes, dois recorreram ao programa Quartz [20, 23] para simular o ambiente de teste e um utilizou o Gem 5 [30]. Essas informações estão detalhadas na Tabela 5.

Alguns estudos adotaram uma abordagem diferente, optando por gerar um disco virtual na DRAM. Para isso, utilizaram uma ferramenta disponível no próprio sistema operacional Linux, sem adicionar qualquer latência à memória. No entanto, é importante ressaltar que essa abordagem tem suas limitações. Ao criar um disco na memória DRAM sem considerar a latência, torna-se inviável comparar o desempenho com outros testes realizados na memória persistente real. Isso se deve ao fato de que o desempenho da DRAM é superior. Portanto, esse tipo de teste permite apenas a verificação do funcionamento da estrutura de dados, e não uma comparação justa do desempenho.

5.5. Q5 - Utiliza DRAM como mecanismo para otimização?

A memória DRAM, com sua versatilidade e velocidade, oferece uma gama de possibilidades para otimização de desempenho. Uma estratégia comum é carregar dados completos na DRAM, permitindo um acesso mais rápido em comparação com a memória

Não simulam	[21], [26], [24], [14], [5], [4], [15], [16], [19], [28], [29], [9], [10], [6], [11]
Não nomeados	[27], [2], [17], [18], [22], [25], [12]
Quartz	[20], [23]
Gen 5	[30]

Tabela 5. Artigos que utilizaram simulação (Q4).

persistente. Alternativamente, pode-se optar por armazenar apenas as partes mais frequentemente acessadas ou um índice da estrutura de dados na DRAM. Essas abordagens, embora distintas, compartilham o objetivo comum de maximizar a eficiência e o desempenho do sistema.

Observou-se que a maioria dos artigos selecionados para este trabalho empregou a DRAM como uma estratégia para otimizar o desempenho. Eles utilizaram parte das estruturas de dados na DRAM como cache, acelerando assim o processo de leitura. Dos artigos analisados, 19 optaram por essa abordagem, enquanto 6 decidiram não utilizar a DRAM, conforme demonstrado na Tabela 6. A maioria dos estudos que empregaram árvores e criaram *caches* optou por uma estratégia específica: armazenar os dados na memória persistente, enquanto a estrutura da árvore era gerada na memória DRAM. Essa abordagem tem uma vantagem significativa, pois elimina a necessidade de acessar a memória persistente para a busca de um nó. Isso acelera a maioria dos processos, já que o acesso à memória persistente se torna necessário apenas para a remoção ou adição de um valor.

Não Utilizada	[27], [21], [9], [11], [14], [23]
Utiliza	[30], [2], [20], [26], [17], [24], [5], [4], [18], [22], [15], [25], [12], [16], [19], [28], [29], [10], [6]

Tabela 6. Artigos que utilizaram DRAM para otimização (Q5).

5.6. Q6 - Levaram em conta alguma característica específica do dispositivo para otimização?

A maioria dos estudos analisados neste artigo emprega algum tipo de otimização com o objetivo de aprimorar o desempenho das estruturas de dados, minimizando a quantidade de operações de leitura ou escrita. Muitos desses estudos focam especificamente na redução do número de operações de escrita, uma vez que a performance da memória persistente tende a ser inferior durante essas operações, apresentando uma latência superior à das operações de leitura.

Dos 25 estudos analisados, apenas 7 não implementaram otimizações específicas nas estruturas de dados, limitando-se a adaptá-las para uso com memória persistente. No entanto, é importante ressaltar que esses sete artigos propuseram a criação de uma nova estrutura de dados. Isso sugere que a inovação e a adaptação são considerações importantes na pesquisa e desenvolvimento de estruturas de dados para memória persistente. Os trabalhos estão agrupados na Tabela 7.

É importante ressaltar que, além dos métodos empregados nos estudos em questão, há uma gama de outros não explorados. Entre eles, destacamos os padrões de acesso, uma

estratégia potencialmente benéfica para certos tipos de memória que se favorecem de acessos sequenciais. Também existe a prática de evitar escritas repetidas no mesmo local, que pode ser uma medida eficaz para prevenir o desgaste desproporcional da memória. Essas abordagens, embora não tenham sido objeto de nossa análise, merecem ser consideradas em futuras investigações. A inclusão dessas estratégias pode enriquecer o escopo do trabalho e fornecer informações adicionais sobre a otimização do uso da memória.

Implementaram Otimizações	[18], [22], [15], [23], [27], [19], [28], [29], [10], [6], [30], [2], [26], [17], [24], [14], [5], [4]
Não Implementaram	[16], [21], [20], [9], [11], [25], [12]

Tabela 7. Artigos que implentaram otimizações para a PM (Q6).

6. Discussão

Uma análise dos estudos revela um padrão recorrente em suas abordagens. A linguagem C++ é frequentemente escolhida devido ao controle mais granular que oferece ao programador sobre a memória. Além disso, a DRAM é comumente utilizada como uma estratégia de otimização, mantendo a estrutura de dados nela para evitar o acesso à PM, assim, melhorar o desempenho. As implementações de otimizações, como a redução da quantidade de escritas na memória persistente, são outra característica comum nos trabalhos. Essas práticas, observadas na maioria dos estudos analisados, refletem o interesse crescente em aprimorar o desempenho da memória persistente em diversos cenários.

Quando se trata da estrutura de dados mais utilizada, as *hash tables* e as árvores se destacam pelo seu desempenho superior em operações de busca e inserção de dados. Foram apresentadas várias versões dessas estruturas de dados, algumas focando no paralelismo, enquanto outras buscam trazer persistência sem comprometer o desempenho. Essas tendências evidenciam a busca contínua por soluções que equilibram eficiência e desempenho na manipulação de dados em memória persistente. Um ponto importante ao comentar sobre os estudos que não simularam seus testes, a maioria utilizou a memória persistente da Intel, Intel Optane 3D, para executar os testes.

Infelizmente a maioria dos trabalhos não disponibiliza os repositórios de códigos fonte em seus artigos. Esta prática limita a replicação dos testes e a adaptação potencial dessas estruturas em outras pesquisas. Acreditamos firmemente que a disponibilidade do código promove o crescimento coletivo da comunidade científica, permitindo a todos contribuir para o desenvolvimento e aprimoramento das tecnologias existentes.

7. Conclusão

Neste artigo, observou-se uma tendência recorrente entre as pesquisas, onde a maioria concentra-se em objetivos similares: aprimorar o desempenho das estruturas de dados ao minimizar o número de gravações na memória persistente, empregando estruturas de dados que priorizam a eficiência, como *hash table* e árvores. Constatamos que a quantidade de pesquisas dedicadas ao estudo de grafos, uma estrutura de dados muito utilizada atualmente, é comparativamente menor do que o esperado, um achado que nos instigou a investigar mais a fundo essa área. Com o fim do suporte para os dispositivos Optane DC da Intel, esperamos que novos trabalhos comecem a focar na padrão Compute Express Link (CXL) [3], que também provê suporte para PM. No entanto, ainda não observarmos esta tendência refletida nos artigos pesquisados.

Agradecimentos. Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos nº 2018/15519-5, 2019/26702-8, 2023/04971-2 e 2023/04969-8 pelo apoio a este trabalho.

Referências

[1] BALDASSIN, A., BARRETO, J., CASTRO, D., AND ROMANO, P. Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv. 54*, 7 (July 2021).

[2] COHEN, N., AKSUN, D. T., AVNI, H., AND LARUS, J. R. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the 24th ASPLOS* (2019), p. 441–454.

[3] DESNOYERS, P., ADAMS, I., ESTRO, T., GANDHI, A., KUENNING, G., MESNIER, M., WALDSPURGER, C., WILDANI, A., AND ZADOK, E. Persistent memory research in the post-Optane era. In *Proceedings of the 1st DIMES* (2023), p. 23–30.

[4] DHULIPALA, L., MCGUFFEY, C., KANG, H., GU, Y., BLELLOCH, G. E., GIBBONS, P. B., AND SHUN, J. Sage: Parallel semi-asymmetric graph algorithms for NVRAMs. *Proceedings of the VLDB Endowment 13*, 9 (2020), 1598 – 1613.

[5] DUAN, Z., YAO, J., LIU, H., LIAO, X., JIN, H., AND ZHANG, Y. Revisiting log-structured merging for kv stores in hybrid memory systems. In *Proceedings of the 28th ASPLOS* (2023), p. 674–687.

[6] FRIEDMAN, M., HERLIHY, M., MARATHE, V., AND PETRANK, E. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd PPoPP* (2018), p. 28–40.

[7] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann, 1992.

[8] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, Nov. 2017.

[9] HU, J., CHEN, J., ZHU, Y., YANG, Q., PENG, Z., AND YU, Y. Parallel multi-split extendible hashing for persistent memory. In *Proceedings of the 50th ICPP* (2021).

[10] HUANG, K., YAN, Y., AND HUANG, L. Revisiting persistent hash table design for commercial non-volatile memory. *Proceedings of the 2020 Design, Automation and Test in Europe Conference and Exhibition, DATE 2020* (2020), 708 – 713.

[11] IZADPANAH, R., PETERSON, C., SOLIHIN, Y., AND DECHEV, D. Petra: Persistent transactional non-blocking linked data structures. *ACM Transactions on Architecture and Code Optimization 18*, 2 (2021).

[12] JAMIL, S., KHAN, A., BURASTALLER, B., AND KIM, Y. Towards scalable manycore-aware persistent b+- trees for efficient indexing in cloud environments. In *2021 ACSOS-C* (2021), pp. 44–49.

[13] KITCHENHAM, B. Procedures for performing systematic reviews. *Keele, UK, Keele Univ. 33* (08 2004).

[14] LAMAR, K., PETERSON, C., DECHEV, D., PEARCE, R., IWABUCHI, K., AND PIRKELBAUER, P. PMap: A non-volatile lock-free hash map with open addressing. In *2021 NVMSA* (2021), pp. 1–7.

[15] LAVINSKY, B., AND ZHANG, X. PM-Rtree: A highly-efficient crash-consistent r-tree for persistent memory. *Proceedings of the 34th International Conference on Scientific and Statistical Database Management* (2022).

[16] LI, Y., ZENG, L., CHEN, G., GU, C., LUO, F., DING, W., SHI, Z., AND FUENTES, J. A multi-hashing index for hybrid dram-nvm memory systems. *Journal of Systems Architecture 128* (2022).

[17] LIM, S., COY, T., LU, Z., REN, B., AND ZHANG, X. NVGraph: Enforcing crash consistency of evolving network analytics in nvmm systems. *IEEE Transactions on Parallel and Distributed Systems 31*, 6 (2020), 1255 – 1269.

[18] NGUYEN, B., TAN, H., DAVIS, K., AND ZHANG, X. Persistent octrees for parallel mesh refinement through non-volatile byte-addressable memory. *IEEE Transactions on Parallel and Distributed Systems 30*, 3 (2019), 677 – 691.

[19] OH, H., CHO, B., KIM, C., PARK, H., AND SEO, J. Anifilter: Parallel and failure-atomic cuckoo filter for non-volatile memories. *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020* (2020).

[20] PAN, W., XIE, T., AND SONG, X. Hart: A concurrent hash-assisted radix tree for dram-pm hybrid memory systems. *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019* (2019), 921 – 931.

[21] SRIVASTAVA, A., AND BROWN, T. Elimination (a,b)-trees with fast, durable updates. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* (2022), 416 – 430.

[22] WANG, C., WEI, Q., WU, L., WANG, S., CHEN, C., XIAO, X., YANG, J., XUE, M., AND YANG, Y. Persisting rb-tree into nvm in a consistency perspective. *ACM Transactions on Storage 14*, 1 (2018).

[23] WANG, H., LI, Z., ZHANG, X., ZHAO, X., AND JIANG, S. Wobtree: a write-optimized b+-tree for non-volatile memory. *Frontiers of Computer Science 15*, 5 (2021).

[24] YAO, Z., ZHANG, J., AND FENG, J. NV-QALSH: An NVM-optimized implementation of query-aware locality-sensitive hashing. *Lecture Notes in Computer Science 12924 LNCS* (2021), 58 – 69.

[25] YU, S., XIAO, N., DENG, M., XING, Y., LIU, F., AND CHEN, W. Rio: Fast b+-tree based on remote accessible non-volatile memory. In *2017 IEEE ISPA/IUCC* (2017), pp. 494–496.

[26] ZHANG, B., ZHENG, S., QI, Z., AND HUANG, L. Nbtree: a lock-free pm-friendly persistent b+-tree for eadr-enabled pm systems. *Contemporary Mathematics 15*, 6 (2022), 1187 – 1200.

[27] ZHANG, X., FENG, D., HUA, Y., CHEN, J., AND FU, M. A write-efficient and consistent hashing scheme for non-volatile memory. *ACM International Conference Proceeding Series* (2018).

[28] ZHONG, C., CHALLA, P., ZHAO, X., AND JIANG, S. Buffered hash table: Leveraging dram to enhance hash indexes in the persistent memory. *Proceedings - 2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium, NVMSA 2022* (2022), 8 – 13.

[29] ZHU, J., HUANG, K., ZOU, X., HUANG, C., XU, N., AND FANG, L. Hdnh: A read-efficient and write-optimized hashing scheme for hybrid dram-nvm memory. *ACM International Conference Proceeding Series* (2021).

[30] ZUO, P., AND HUA, Y. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems 29*, 5 (2018), 985 – 996.