Programando para Memória Persistente: Dificuldades, Armadilhas e Desempenho

Lucas Bastelli[1], **Alexandro Baldassin**[1], **Emilio Francesquini**[2]

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

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

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

e.francesquini@ufabc.edu.br

Resumo. *A tecnologia de memória persistente (PM) provê endereçamento por byte com latência de acesso relativamente próxima às das memórias voláteis (DRAM). Contudo, programar com PM traz novos desafios não encontrados em ambientes típicos com memória volátil. Neste contexto, este artigo apresenta duas contribuições principais. Primeiramente, analisamos qualitativamente as dificuldades e armadilhas de se programar com a PMDK, a biblioteca oficial da Intel para programação do Optane DC (seu dispositivo de armazenamento persistente). Esta análise é o resultado das nossas experiências coletadas durante o desenvolvimento de um conjunto de estruturas de dados típicas de aplicações com persistência de dados. A segunda contribuição é uma análise de desempenho considerando implementações persistentes e voláteis dessas estruturas de dados para DRAM, Optane DC e SSD. Os resultados experimentais mostram que o Optane DC, apesar de em média ser 5,54× mais lento que o dispositivo volátil (DRAM), supera o SSD em 14× no pior caso.*

1. Introdução

Quando se trata de armazenamento de dados de forma persistente, ou seja, dispositivos que mantenham os dados mesmo após o desligamento da máquina, pensamos em HDs e SSDs, comumente utilizados atualmente. Porém, esses dispositivos possuem uma taxa de transferência muito baixa, com tempo de resposta alta; assim, não é possível realizar operações que precisam ser persistentes de forma rápida. Tradicionalmente, a tecnologia DRAM é utilizada como memória de trabalho, provendo acesso rápido porém os dados não são armazenados de forma persistente. Novas tecnologias de memória tem sido pesquisadas que possuem as principais qualidades de ambos os tipos previamente expostos: rapidez da DRAM e persistência dos HDs/SDs. Esse tipo de memória é geralmente conhecida como Memória Persistente ou PM (*Persistent Memory*) [Baldassin et al. 2021].

A PM é conectada diretamente no barramento do processador, possuindo além da alta taxa de transferência e tempo de resposta baixa, granularidade de byte, similar à memória DRAM. Porém, a programação com esse tipo de dispositivo traz problemas não comumente encontrados com estruturas armazenadas em memória volátil (DRAM). Em particular, como as alterações são persistentes, uma falha do sistema (como queda de energia, por exemplo) pode deixar o sistema em um estado inconsistente. Por exemplo, imagine uma operação que move um elemento do vetor V_1 para o vetor V_2: se a falha ocorrer depois que o elemento é retirado de A, mas antes de ser inserido em V_2,

na reinicialização o sistema estará em um estado inválido (um elemento do vetor V_1 simplesmente desapareceu). Um outro problema se deve ao fato de alguns componentes da hierarquia de memória, como as caches, ainda serem voláteis. Como o programador não controla quando os dados saem da cache, pode ser que um dado D_2, que foi escrito depois de um outro dado D_1, atinja a memória persistente antes de D_1. Se houver uma falha antes que D_1 se torne persistente, novamente o sistema pode ficar em um estado inconsistente.

Como resultado, a programação com PM exige cuidados adicionais quando comparada à programação com memória volátil. Vários modelos de programação para PM tem sido pesquisados recentemente [Baldassin et al. 2021]. A Intel, em particular, tem trabalhado em uma biblioteca para PM desde pelo menos 2014. Essa biblioteca é conhecida como PMDK (*Persistent Memory Development Kit*) [Scargall 2020]. Do ponto de vista comercial, a Intel também saiu na frente e está disponibilizando desde 2019 os dispositivos conhecidos como Intel Optane DC, baseados na tecnologia 3D XPoint [Jeongdong Choe 2017].

Neste contexto, são duas as principais contribuições deste artigo: 1) uma discussão das dificuldades e principais armadilhas ao se programar estruturas de dados típicas (lista encadeada, tabela de hash, skip-list) usando o PMDK da Intel; 2) uma análise inicial de desempenho de tais estruturas utilizando o Optane DC. A discussão das principais dificuldades encontradas tem como objetivo auxiliar programadores que estão começando a trabalhar com memória persistente e a interface PMDK da Intel. A análise experimental mostra o potencial ganho de desempenho ao se utilizar a nova tecnologia em contraste com a DRAM e SSDs [1].

O restante do artigo é organizado da seguinte forma. A Seção 2 descreve aspectos de programabilidade da PM e apresenta os principais conceitos utilizados na programação com o PMDK. Nesta seção também são apontados trabalhos que investigaram aspectos semelhantes aos tratados aqui. A Seção 3 apresenta a metodologia empregada para analisar os aspectos de programabilidade e discute as principais dificuldades encontradas, além de apontar as características essenciais que poderiam ser melhoradas. A Seção 4 apresenta os resultados experimentais e, finalmente, a Seção 5 conclui o trabalho.

2. Programando com memória persistente

Esta seção apresenta os principais aspectos da programação com PM (Seção 2.1) e, em seguida, os conceitos mais relevantes utilizados pelo PMDK (Seção 2.2).

2.1. Por que é difícil programar para PM?

Do ponto de vista semântico, a principal diferença da programação com PM, quando comparada à tradicional com DRAM, é que as escritas são persistidas. Enquanto a re-inicialização do sistema reverteria os erros causados em uma versão para DRAM, o mesmo não pode ser dito no caso da PM; é como se os erros se tornassem persistentes também, deixando o sistema em um estado inconsistente de forma permanente. Esses erros em geral se manifestam em situações de *falhas* que seriam inócuas no caso de DRAM. Considere, por exemplo, uma queda de energia. No caso da PM, se a queda acontecer em

[1] Todo o código produzido e usado neste artigo pode ser encontrado em `https://github.com/LucasBastelli/WSCAD2022_PersistentMemory.git`.

algum momento intermediário de atualização de uma estrutura de dado composta, um problema de consistência pode se manifestar. O exemplo mais típico é quando um elemento deve ser movido de um conjunto para outro; essa operação não é atômica, no sentido que uma falha depois do elemento ser removido do conjunto, mas antes de ser adicionado ao outro, deixa a aplicação num estado inconsistente.

Assim, a programação com PM necessita de algum mecanismo que garanta segurança à falha no caso da atualização de estruturas de dados. Geralmente esta garantia é oferecida por meio do conceito de *transação*, já conhecido pela comunidade de Banco de Dados [Gray and Reuter 1992]. Uma transação garante ou que todas as atualizações são realizadas, ou então nenhuma é (propriedade conhecida também como *tudo ou nada*). É necessário que o programador saiba exatamente o trecho de código que contém as operações que serão realizadas para utilizar de forma adequada uma transação: se a atualização de um estado importante ficar fora da transação, pode-se ter um erro de consistência.

Dois trabalhos pioneiros investigaram aspectos de programabilidade com PM. O primeiro deles, conduzido por [Ren et al. 2017], reuniu um grupo de 30 participantes entre alunos de graduação e doutorandos, todos com conhecimento sólido em programação. Foi então fornecido aos participantes o código de uma aplicação convencional (codificada para memória DRAM) e pedido para que os participantes gerassem uma versão persistente. Eles poderiam usar um mecanismo similar ao da transação e também um alocador de memória persistente. Os pesquisadores então avaliaram o código entregue e analisaram os principais erros cometidos. Os erros em geral apontaram para a dificuldade de se usar adequadamente as transações: em alguns casos, os programadores não protegiam todos os dados persistentes de forma adequada; em outros casos, dados voláteis também eram protegidos.

Em outro trabalho pioneiro, [Marathe et al. 2017] discutiram as dificuldades em transformar uma versão da aplicação Memcached, otimizada para DRAM, para o uso em sistemas com PM. O resultado da pesquisa apontou para várias dificuldades encontradas durante o processo, entre elas a questão de determinar o que deve ser persistido e o uso de ponteiros persistentes. Este artigo procura complementar esses dois trabalhos anteriores ao expor nossa experiência com o uso da biblioteca PMDK.

2.2. PMDK

A programação para PM diverge de várias maneiras do modelo tradicional com DRAM. Como a tecnologia de PM é relativamente recente, a pilha de software ainda é incipiente e há uma grande variedade de modelos sendo propostos na Academia [Baldassin et al. 2021]. A abordagem para programação de PM da Intel está disponível no PMDK (*Persistent Memory Development Kit*) [Scargall 2020], desenvolvido no modelo de código aberto[2]. O projeto é atualizado com bastante frequência e é notável o interesse da Intel em prover essa infraestrutura de software dado que lançou recentemente no mercado dispositivos com PM (Intel Optane DC).

O PMDK é composto por várias camadas de abstração. Aqui nos concentraremos na camada `libpmemobj`. Esta é a camada mais básica e que provê os recursos essenciais

[2]Veja https://pmem.io/

para programação de estruturas de dados persistentes, em particular *transações*. São três os conceitos chaves empregados pelo PMDK para a programação com PM: i) acesso à PM; ii) atualização dos dados persistentes; e iii) gerenciamento de memória.

2.2.1. Acesso à PM

A PM é acessada por meio de um *pool de memória*, representado no PMDK pelo tipo `PMEMobjpool`. O pool de memória é uma abstração para a área de memória persistente que será gerenciada. Do ponto de vista do sistema operacional, o pool de memória é tratado como um arquivo convencional e pode ser criado pela ferramenta `pmemtool`, disponibilizada pelo PMDK (recomendado), ou programaticamente. Uma vez criado, programadores devem explicitamente abrir o pool por meio de uma chamada específica fornecida pelo PMDK antes de trabalhar com seu conteúdo. Essa chamada retornará um identificador, comumente chamado de *pool object pointer* (pop), utilizado para realizar diversas operações sobre o pool. Enquanto que na programação tradicional com memória volátil a memória é alocada a partir do *heap*, com PM os dados devem ser alocados no pool. Diferentemente do heap, os dados na PM são acessados a partir de um *objeto raiz*. Todo objeto armazenado em um pool persistente deve ser acessado a partir do objeto raiz. Todo pool tem exatamente um objeto raiz e ele sempre existe.

Uma vez o pool aberto, o objeto raiz pode ser recuperado por meio de uma chamada oferecida pelo PMDK que devolve um ponteiro para esse objeto. Esse ponteiro é diferente dos ponteiros típicos empregados em linguagens convencionais como C, por exemplo, já que aponta para um objeto persistente. Usa-se o termo *ponteiro persistente* para diferenciar os ponteiros para PM dos ponteiros convencionais para memória volátil. No PMDK, um ponteiro persistente é representado pelo tipo `PMEMoid` de 128 bits, composto por um identificador único de pool (64 bits), e um deslocamento (64 bits).

2.2.2. Atualização dos dados persistentes

O conceito principal utilizado pelo PMDK para proporcionar alteração do estado persistente é o de *transação*. Uma transação garante que todas as alterações realizadas dentro dela ou são efetivadas (*committed*) e consideradas persistentes, ou então abortadas (*aborted*) e revertidas (*rollback*) – nesse caso é como se nada tivesse acontecido. Desta forma, se uma queda de energia ou falha do sistema acontece enquanto uma transação está sendo executada, todas as alterações realizadas até aquele ponto serão revertidas. As transações fornecidas pelo PMDK não garantem atomicidade entre threads, ou seja, as modificações realizadas por uma transação de uma determinada thread são visíveis para todas as outras. Portanto é necessário a utilização de um mecanismo de sincronização específico no caso de aplicações multithreading.

No PMDK, uma transação pode ser especificada por meio do conjunto de macros `TX_BEGIN` e `TX_END`. Ao começar a transação é necessário especificar o pool de memória sobre o qual ela atuará. A maneira como as transações do PMDK garantem atomicidade a falhas é por meio do *versionamento de dados*, similar ao que acontece em sistemas de Banco de Dados. A técnica mais comum é conhecida como *Write-Ahead Logging*, ou simplesmente WAL [Mohan et al. 1992]. Apesar do programador não precisar conhecer exatamente como o esquema de logging é implementado, ele precisa informar

para o PMDK, por meio de chamadas específicas da biblioteca (e.g., `TX_ADD`), quais os blocos de memória persistente serão alterados.

2.2.3. Alocação de memória

A interface para alocação de memória persistente também se difere da interface convencional para memória volátil, já que é necessário efetuar a alocação e a atribuição do ponteiro persistente de forma atômica. A forma mais comum de efetuar operações de gerenciamento de memória é por meio de uma transação. Pelo fato da chamada estar sendo realizada dentro de uma transação, uma falha antes da atribuição do ponteiro faz com que todo o trecho de código (inclusive a memória persistente alocada) seja revertido, evitando o vazamento de memória.

3. Metodologia e dificuldades encontradas

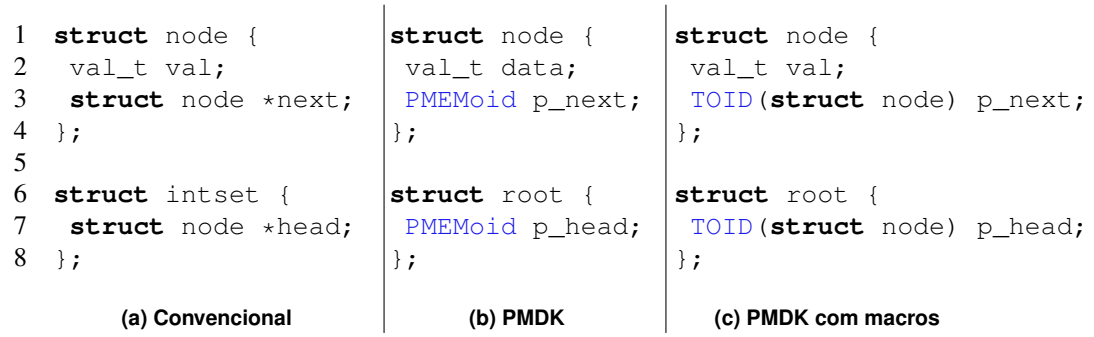
Para avaliar a questão da programabilidade do PMDK, partiu-se de uma versão já codificada de uma aplicação sintética que continuamente executa operações de inserção, remoção e busca em 3 estruturas de dados típicas, a saber: i) lista encadeada, ii) tabela de hash e iii) skip-list. Esta aplicação tem sido bastante usada na pesquisa com memória transacional e é comumente conhecida como `intset` [Felber et al. 2008]. Uma versão persistente do `intset` foi desenvolvida com base na versão volátil para as três estruturas de dados mencionadas. O procedimento espelha o que aconteceria em um caso típico de uso da PM, ou seja, o porte da versão volátil de uma aplicação para o domínio persistente, similar aos trabalhos de [Ren et al. 2017] e [Marathe et al. 2017]. No restante desta seção organizamos a análise por meio de tópicos que descrevem as principais dificuldades e armadilhas encontradas, bem como sugerem algumas melhorias, quando cabíveis.

3.1. Ponteiros persistentes

O primeiro aspecto importante encontrado na codificação para PM envolve o uso de ponteiros persistentes. No PMDK, o tipo `PMEMoid` é usado. Nas definições da estruturas de dados, os ponteiros convencionais devem então ser substituídos pelos persistentes para as estruturas que serão persistidas. A Figura 1 mostra um exemplo para um lista encadeada simples. Na parte (a) da figura, pode-se notar o ponteiro para a lista na linha 7 e o ponteiro que encadeia os elementos da estrutura na linha 3. Uma primeira versão persistente é mostrada na parte (b), usando agora o tipo `PMEMoid`. Um aspecto complicador é que o tipo `PMEMoid` não é um tipo nativo; toda de-referenciação do ponteiro deve ser feita por meio de uma chamada PMDK (`pmemobj_direct`). Assim, para atribuir um valor val para um nodo apontado por um ponteiro $pnode$, é necessário escrever algo como:

```
((struct node *)pmemobj_direct(pnode))->val = val;
```

Note que `PMEMoid` especifica um endereço de um objeto persistente, mas não o tipo do objeto que está sendo apontado. Assim, é necessário também fazer um coerção explícita para `struct node`. Esta forma de codificação tende a provocar facilmente vários erros que podem passar despercebidos, já que o tipo do valor sendo associado à variável não é verificado pelo compilador. Para sanar parcialmente o problema, os desenvolvedores do PMDK passaram a fornecer uma alternativa baseada em macros para se trabalhar com os ponteiros e estruturas persistentes em geral. No código exemplo da Figura 1(c), a macro `TOID(t)` é usada para declarar um ponteiro persistente que aponta para um objeto do tipo t. Neste caso, a atribuição anterior poderia ser feita assim:



```
1 struct node {
2  val_t val;
3  struct node *next;
4 };
5
6 struct intset {
7  struct node *head;
8 };
```

(a) Convencional

```
struct node {
 val_t data;
 PMEMoid p_next;
};

struct root {
 PMEMoid p_head;
};
```

(b) PMDK

```
struct node {
 val_t val;
 TOID(struct node) p_next;
};

struct root {
 TOID(struct node) p_head;
};
```

(c) PMDK com macros

Figura 1. Definição da estrutura para lista ligada simples: (a) versão típica para memória volátil; (b) ponteiros persistentes (sem tipagem); (c) ponteiros persistentes (com tipagem através de macros).

```
D_RW(pnode)->val = val;
```

A macro `D_RW` é usada para de-referenciar o ponteiro persistente para escrita. Por meio das macros, o compilador pode detectar estaticamente erros relacionados a tipos. Mesmo assim, observamos que grande parte dos erros cometidos estiveram relacionados aos ponteiros. Um estilo de codificação que nos ajudou foi usar o prefixo `p_` antes dos nomes dos ponteiros persistentes para separá-los dos voláteis. Outro aspecto importante é que o conteúdo de um dado persistente deve ser acessado por meio de um objeto raiz. Assim, por convenção, sempre declaramos uma estrutura que contém este objeto (linha 6 das partes (b) e (c)).

O exemplo anterior pode ter dado a falsa impressão de que a conversão de ponteiros voláteis para persistentes pode ser feita de forma automática. Infelizmente este não é o caso. Considere agora um outro exemplo para uma implementação do `intset` que usa uma estrutura baseado em tabela de hash, como mostrado na Figura 2. Novamente, o lado (a) da figura mostra a implementação original para memória volátil. A estrutura é implementada por meio de um ponteiro para um vetor de ponteiros (linha 7), no qual cada ponteiro aponta para uma lista com elementos da tabela (linhas 1–4). Note que simplesmente substituir a linha 7 por `TOID(struct bucket_t) *buckets` não funcionaria, pois o ponteiro para o vetor de ponteiros persistentes não seria persistente. A solução foi dividir de forma explícita os dois níveis de ponteiros como mostra a Figura 2(b). O objeto raiz (linha 12) aponta para uma estrutura que armazena um vetor de ponteiros persistentes (linha 8). Para isso foi usado *membros de array flexível*, um recurso da linguagem C disponível no padrão C99. Além do vetor (linha 8) é necessário que a estrutura tenha pelo menos outro membro (linha 7). Algo análogo também precisou ser feito para a implementação da estrutura skip-list. Ponteiros, em geral, tendem a ser um fator comum de erros na linguagem C. Notamos que o uso do PMDK para memória persistente torna ainda mais complexo o uso de ponteiros. [Marathe et al. 2017] chegaram a uma conclusão similar com o Memcached.

3.2. Uso de macros

Como explicado anteriormente, o PMDK passou a disponibilizar a opção de usar macros principalmente para reduzir os erros cometidos com a manipulação dos ponteiros persistentes. Para que as macros de acesso (leitura e escrita) aos objetos possam ser utilizadas, é necessário antes criar um *layout* para a área de memória persistente que será utilizada.

```
1  struct bucket {
2    val_t val;
3    struct bucket *next;
4  };
5
6  struct intset {
7    struct bucket_t **buckets;
8  } intset_t;
9
10
11
12
13
```

(a) Convencional

```
struct bucket {
  val_t val;
  TOID(struct bucket) p_next;
};

struct hashmap {
  size_t size;
  TOID(struct bucket) bucketList[];
};

struct root {
  TOID(struct hashmap) buckets;
};
```

(b) PMDK

Figura 2. Definição da estrutura para uma tabela de hash: (a) versão típica para memória volátil; (b) versão para PM.

```
1  POBJ_LAYOUT_BEGIN(intset);
2  POBJ_LAYOUT_ROOT(intset, struct root);
3  POBJ_LAYOUT_TOID(intset, struct bucket);
4  POBJ_LAYOUT_TOID(intset, struct hashmap);
5  POBJ_LAYOUT_END(intset);
```

Figura 3. Definição do layout de memória para o exemplo com a tabela de hash usando o mecanismo de macros do PMDK.

Este layout consiste das declarações do tipos que serão utilizados para a manipulação do pool persistente. A Figura 3 mostra o layout definido com o PMDK para o `intset` implementado com a tabela de hash apresentada anteriormente na Figura 2(b). A linha 1 indica o começo da declaração e atribui um nome (`intset`) para o layout. De forma análoga, a linha 5 termina a declaração do layout. A declaração consiste de uma lista de tipos. É sempre necessário ter um objeto raiz, como declarado na linha 2. As linhas 3 e 4 definem tipos auxiliares.

Apesar do objetivo importante de facilitar a manipulação de ponteiros persistentes, as macros trazem consigo problemas novos. O primeiro deles é que erros de compilação causados pelas macros podem ser confusos e de difícil diagnóstico. Porém, a principal inconveniência que encontramos foi manter a declaração do layout sincronizada com as declarações das estruturas que fazem parte do pool persistente. Em um cenário, precisamos definir uma nova estrutura para o problema que estávamos resolvendo, mas esquecemos de colocá-la na declaração do layout. Só depois de muito tempo este problema foi diagnosticado, pois as mensagens de erro de compilação não permitiam o seu diagnóstico exato. Tal dificuldade seria amenizada ao se adotar um estilo de programação que sempre coloque próximas, no código, as estruturas de dados e a definição do respectivo layout.

3.3. Transações

Alterações em uma estrutura de dados persistente devem ser colocadas dentro de uma transação. Apesar de transações serem comuns em banco de dados, elas são relativamente desconhecidas pela maioria dos programadores. O estudo conduzido por [Ren et al. 2017]

```
1 TX_BEGIN(pop){
2  TOID(struct node) p_newN = createNewNode(pop, data);
3  D_RW(p_newN)->p_next = *head;
4  TX_ADD_DIRECT(head);
5  *head = p_newN;
6 }TX_END
```

Figura 4. Exemplo de um transação para a operação de inserção de um elemento em uma lista encadeada simples.

destaca que o uso correto de transações foi uma das principais dificuldades encontradas pelos programadores. Um dos principais desafios é determinar exatamente quais instruções devem fazer parte da transação.

O suporte para transação disponível no PMDK adiciona novos desafios. O principal deles é que o programador deve especificar explicitamente quais dados devem ser versionados. O versionamento é a principal forma usada pela transação para garantir durabilidade, pois é o mecanismo usado para criar os logs. A Figura 4 mostra um exemplo de uma transação usada para inserir um novo elemento na cabeça de uma lista encadeada simples. Ao iniciar a transação (linha 1) é necessário especificar o pool persistente que ela está atrelada (`pop`). Um novo nodo é criado (linha 2) e então ligado com o próximo elemento: a cabeça da lista (linha 3). Antes da cabeça da lista ser alterada para apontar para o novo elemento criado (linha 5) é necessário adicioná-la no log da transação, o que pode ser feito por meio da chamada `TX_ADD_DIRECT` (linha 4). Note que, se isso não for feito, e houver alguma falha depois da alteração da cabeça (linha 5), mas antes da transação ser finalizada (linha 6), a estrutura ficará em um estado inconsistente.

Tradicionalmente, as transações também fornecem atomicidade e isolamento da execução relativo a outras transações. Assim, é possível ter várias transações executando em paralelo. Contudo, transações no PMDK não fornecem esta propriedade. Logo, é necessário ainda utilizar um outro mecanismo de sincronização (como locks, por exemplo) para garantir atomicidade. Por questões práticas, aspectos de execução concorrente não foram explorados neste texto, mas é importante ter em mente que este é ainda outro fator que deve ser considerado no uso das transações com o PMDK.

3.4. Corretude

Testar se um código escrito para PM está correto traz novos desafios. Grande parte das falhas que podem corromper as estruturas de dados não acontecem com frequência, o que torna difícil sua detecção e até mesmo a confecção de testes de corretude.

Para ilustrar o problema, considere o versionamento realizado no exemplo da transação (linha 4 da Figura 4) e suponha que o programador o tenha omitido. O compilador não apresentará nenhum erro e a operação funcionará corretamente na maioria dos casos. O problema se manifestará apenas quando uma falha (e.g., queda de energia) ocorrer depois da modificação da cabeça da lista (linha 5) mas antes do final da transação (linha 6). Ou seja, há uma janela de tempo muito estreita para que o problema aconteça.

O problema de corretude foi detectado várias vezes durante a codificação das estruturas de dados que usamos neste artigo. Em geral esses erros não provocaram nenhum

problema de execução porque as falhas não aconteceram durante nossos testes. Para detectá-los tivemos de checar todo o código diversas vezes. Este é um problema importante e diversos artigos recentes tem explorado novas técnicas para facilitar a detecção, correção e geração de testes para persistência [Liu et al. 2020, Gorjiara et al. 2022, Chen et al. 2022].

4. Análise de desempenho

3D XPoint é o nome da tecnologia de memória usada na PM oferecida pela Intel sob o nome comercial Optane DC. Essa tecnologia é uma implementação de PCM que emprega uma estrutura tridimensional visando o aumento da densidade [J.Choe 2017], chegando a alcançar $\sim 16\times$ a densidade típica de uma memória DRAM. Quando comparada a outras tecnologias não-voláteis de memória como Flash, essa tecnologia oferece uma ótima durabilidade com 10^8 ciclos de leitura e escrita. Tais características também são acompanhadas de um preço competitivo, mais barato que a memória DRAM. Por outro lado, quando comparada à mesma DRAM, tem uma latência maior (~ 350ns) e larguras de banda $2,7\times$ e $12,9\times$ menores para a leitura e escrita, respectivamente [Patil et al. 2019].

Nesta parte do artigo concentraremo-nos no impacto que o uso da PM tem no desempenho de aplicações. De fato, ficou demonstrado que as caches dos processadores modernos são capazes de mascarar o aumento do custo de acesso à memória de trabalho para boa parte dos benchmarks do SPEC2006 [Palma et al. 2016]. Contudo, diferentemente de trabalhos anteriores [Palma et al. 2016, Kannan et al. 2016, Ren et al. 2015, Liu et al. 2014] que utilizam simulações, nossa avaliação apresenta dados coletados por experimentos feitos em um hardware real. Outro aspecto que diferencia este trabalho é o uso do PMDK que, efetivamente, combina o uso de DRAM com PM pelas aplicações; um cenário cujo desempenho esperado pode ser bem diferente daquele previsto pelos trabalhos relacionados, seja por que eles preveem a substituição total de DRAM por PM ou por usarem acessos diretos à PM muitas vezes sem o devido controle para garantir a consistência e a durabilidade dos dados em caso de falhas.

4.1. Ambiente experimental

A avaliação experimental do desempenho foi feita através do `intset`, implementado com as três estruturas de dados já mencionadas: lista encadeada, tabela de hash e skip-list. O benchmark `intset` começa com a estrutura de dados pré-populada. O valor "Tamanho" relatado nos resultados da Seção 4.2 é o número de elementos nesta estrutura. A execução do benchmark prossegue por um tempo pré-determinado efetuando uma sequência de operações escolhidas aleatoriamente dentre busca, inserção e remoção. Nos nossos experimentos utilizamos 50% de probabilidade para buscas, e 50% de atualizações (divididos igualmente entre inserções e remoções). Como o objetivo é a medição da vazão média, após o período pré-determinado de tempo o número de operações efetuadas é coletado e a vazão média calculada. Durante a execução, os elementos na lista encadeada e da skip list são mantidos em ordem. Isso garante que buscas, inserções e remoções na skip-list possam ser feitas em tempo esperado logarítmico, enquanto para listas encadeadas o tempo de cada operação é linear e para a tabela de hash o tempo esperado é constante.

Para avaliar a diferença de desempenho que o uso da PM tem quando comparado à DRAM, implementamos cada uma das estruturas de dados usando o PMDK e, durante

a execução, escolhemos o dispositivo a ser usado para o armazenamento dos dados entre PM, DRAM (*sem persistência*, ou seja, é útil para base de comparação) e arquivo tradicional salvo em um SSD. Cada experimento foi executado 10 vezes e as barras de erro representam o intervalo de confiança de 95% calculado utilizando o método bootstrap com 10.000 reamostragens.

A máquina onde os experimentos foram executados possui um processador Intel Xeon Gold 5317, com frequência base de funcionamento de 3.0 GHz (até 3.6GHz com Turbo Boost), e 128GiB de DRAM, dividida em 4 módulos de 32GiB funcionando a 3.2GHz. A PM utilizada é a Intel Optane DC, num total de 512GiB, dividida em 4 módulos de de 128GiB funcionando a 3.2 GHz, e o SSD é o Intel DC S4600 Series de 960GB. O ambiente de software é composto do sistema operacional Linux com kernel versão 5.4.0, GCC 9.4.0, e PMDK 1.8.

4.2. Resultados experimentais

A Figura 5 reúne os resultados experimentais. Nas Figuras *5(a), 5 (b) e 5 (c)* o eixo vertical mostra, para cada uma das estruturas de dados, a vazão em operações por segundo (atente-se à escala logarítmica). Já no eixo vertical (também em escala logarítmica) está indicado o tamanho, ou seja, o número de elementos na estrutura de dados. As barras vermelhas trazem os resultados para o salvamento em DRAM, em azul os resultados com o uso da PM e, em verde, os resultados do salvamento dos dados em um SSD.

Considerando, a princípio, apenas as operações em DRAM e PM, fica claro o impacto negativo que a complexidade linear das operações na lista encadeada (Figura 5 *(a)*) têm na vazão conforme avaliamos tamanhos maiores de conjuntos. Como esperado, este comportamento não é visível na tabela de hash (Figura 5 *(b)*), que tem complexidade constante, e apenas ligeiramente observável em skip lists (Figura 5 *(c)*) que tem uma complexidade esperada logarítmica. Contudo, este comportamento não é tão visível (apesar de estar presente) quando avaliamos a vazão para SSDs (barras verdes). A explicação, neste caso, é que o tempo de acesso aos dados no SSD é tão maior que aquele da DRAM e da PM, que a vazão já começa em um patamar tão baixo a ponto do *overhead* de utilizarmos uma estrutura menos eficiente, como listas encadeadas, ser apenas notado para os maiores tamanhos de entrada.

A Figura 5 *(d)* mostra os resultados da comparação direta do desempenho tomando como *baseline* o desempenho da DRAM (atenção à escala logarítmica). Cabe ressaltar que tal comparação deve ser considerada com parcimônia já que, enquanto os tempos para PM e SSD incluem persistência, o mesmo não pode ser dito a respeito de DRAM. Enquanto para PM os slowdowns estão entre 1,6 e 11,7 com média 5,54 (médias 3,2 para lista encadeada, 11,2 para tabela de hash, e 4,7 para skip-list), eles são muito maiores para SSD, variando de 2,3 a 633,4 com média 129,3 (médias 47,0 para lista encadeada, 274,8 para tabela de hash, e 167,4 para skip-list). Ainda que os resultados mostrem que, claramente, DRAM é de fato mais rápida do que PM, o uso de PM para persistência de estruturas de dados comumente utilizadas pode reduzir, no pior caso, em 14$\times$ o tempo de salvamento quando comparado ao padrão de ouro atual baseado em SSDs.

5. Conclusão

Neste artigo apresentamos duas contribuições principais. Primeiro, discutimos e apontamos as principais dificuldades e armadilhas ao se programar com PM utilizando-se o

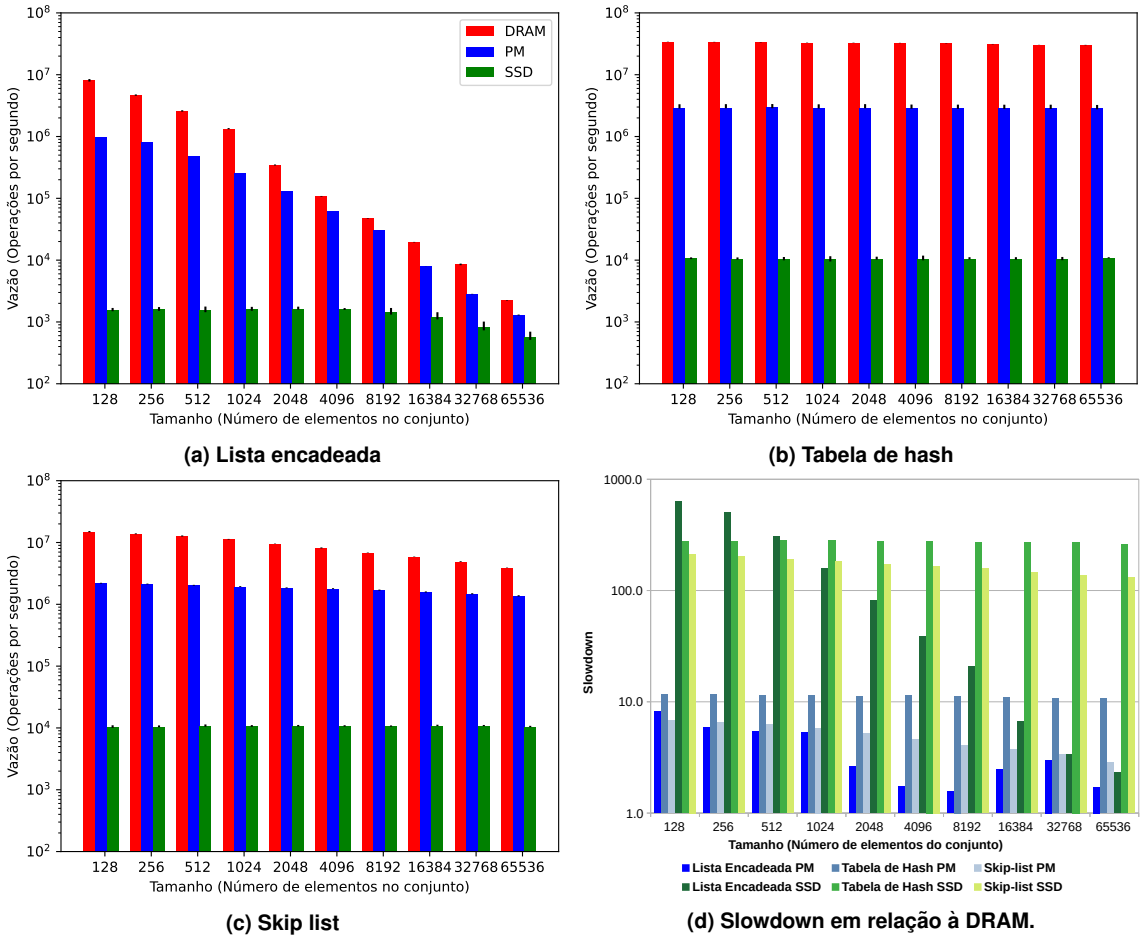


Figura 5. Comparação entre os desempenhos das memórias.

Intel PMDK. Posteriormente, mostramos resultados experimentais nos quais comparamos o desempenho de implementações de estruturas de dados típicas (lista encadeada, tabela de hash e skip-list) para PM, DRAM e SSD. Os resultados experimentais mostram que o dispositivo persistente baseado no Intel Optane DC, apesar de em média ser 5,54× mais lento que o dispositivo volátil (DRAM), supera o SDD em 14× no pior caso.

Agradecimentos. Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos nº 2018/15519-5, 2019/26702-8 e 2021/05440-5 pelo apoio a este trabalho.

Referências

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

Chen, Z., Hua, Y., Zhang, Y., and Ding, L. (2022). Efficiently detecting concurrency bugs in persistent memory programs. In *ASPLOS'22*, pages 873–887.

Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *PPoPP'08*, pages 237–246.

Gorjiara, H., Xu, G. H., and Demsky, B. (2022). Yashme: detecting persistency races. In *ASPLOS'22*, pages 830–845.

Gray, J. and Reuter, A. (1992). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1st edition.

J.Choe (2017). Intel 3d xpoint memory die removed from intel optane™ pcm (phase change memory). urlhttps://www.techinsights.com/blog/intel-3d-xpoint-memory-die-removed-intel-optanetm-pcm-phase-change-memory.

Jeongdong Choe, T. I. (2017). Intel 3D XPoint memory die removed from Intel Optane™ PCM (phase change memory).

Kannan, S., Gavrilovska, A., and Schwan, K. (2016). pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *EuroSys'16*, pages 1–16.

Liu, R.-S., Shen, D.-Y., Yang, C.-L., Yu, S.-C., and Wang, C.-Y. M. (2014). NVM Duet: Unified Working Memory and Persistent Store Architecture. In *ASPLOS'14*, pages 455–470.

Liu, S., Seemakhupt, K., Wei, Y., Wenisch, T., Kolli, A., and Khan, S. (2020). Cross-Failure Bug Detection in Persistent Memory Programs. In *ASPLOS'20*, ASPLOS '20, pages 1187–1202.

Marathe, V. J., Seltzer, M., Byan, S., and Harris, T. (2017). Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *USENIX HotStorage' 17*.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162.

Palma, M., Francesquini, E., and Azevedo, R. (2016). Simulação de arquiteturas de hardware com memórias não-voláteis. In *Anais do XVII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 121–132.

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 pptane DC persistent memory modules. In *ISMM'19*, pages 288–303.

Ren, J., Hu, Q., Khan, S., and Moscibroda, T. (2017). Programming for Non-Volatile Main Memory Is Hard. In *APSys'17*, pages 1–8.

Ren, J., Zhao, J., Khan, S., Choi, J., Wu, Y., and Mutlu, O. (2015). ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *MICRO'15*, pages 672–685.

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