PM.NET: Uma biblioteca de desenvolvimento para memória persistente com C#

Henrique Guirelli[1], **Emilio Francesquini**[1], **Alexandro Baldassin**[2]

[1]Universidade Federal do ABC (UFABC)
Santo André – SP – Brasil

[2]Universidade Estadual Paulista (UNESP)
Rio Claro – SP – Brasil

{henrique.guirelli,e.francesquini}@ufabc.edu.br, alexandro.baldassin@unesp.br

Resumo. *Este artigo apresenta a* `PM.NET`*, uma biblioteca em C# para desenvolvimento de aplicações que utilizam memória persistente. A biblioteca* `PM.NET` *oferece uma interface simples e orientada a objetos para o desenvolvimento em memória persistente, com gerenciamento automático da memória persistente (incluindo coleta de lixo). Uma vez identificados os objetos raiz a serem persistidos, a* `PM.NET` *transforma automaticamente todos os objetos relevantes em objetos persistentes. Adicionalmente, o* `PM.NET` *proporciona uma forma de envolver objetos voláteis em objetos persistentes com proxies, minimizando as alterações de código necessárias para o uso da memória persistente com desempenho comparável a banco de dados relacionais.*

1. Introdução

Ao discutir a arquitetura de computadores, é importante abordar a hierarquia de memória. Essa abordagem posiciona a memória mais rápida e de menor capacidade no topo, enquanto as memórias mais lentas e de maior capacidade estão nos níveis inferiores. A memória no topo da hierarquia é a primeira a ser acessada em uma requisição; caso ela não possa responder (por não conter o dado desejado, funcionando como uma cache), a responsabilidade é repassada para o nível inferior. Com essa estrutura, a hierarquia de memória atinge o objetivo de mascarar o desempenho inferior dos níveis mais baixos [Burks et al. 1982].

Tipicamente, os computadores utilizam quatro tecnologias para implementar a hierarquia de memória: DRAM (Dynamic Random Access Memory), SRAM (Static Random Access Memory), memória Flash e disco magnético. A DRAM representa a memória principal do computador, enquanto a SRAM é usada nos níveis mais próximos do processador, como as memórias cache. Embora a DRAM seja mais lenta que a SRAM, seu custo financeiro por *bit* é menor, e ela ocupa menos área por *bit* de memória, o que permite maior capacidade em relação à quantidade de silício [Hennessy and Patterson 2014, p. 331]. Além disso, também há a tecnologia de memória Flash, comumente utilizada em SSDs (Solid State Drives), e os discos magnéticos, que geralmente são os níveis mais baixos da hierarquia de memória e apresentam maior capacidade de armazenamento, embora sejam mais lentos [Hennessy and Patterson 2014].

As tecnologias de memória podem ser categorizadas em dois tipos principais: volátil e não volátil. A memória volátil é geralmente mais rápida, porém os dados são

perdidos quando há interrupção no fornecimento de energia. Por outro lado, a memória não volátil é geralmente mais lenta, mas possui características persistentes, mantendo os dados na ausência de energia [Han et al. 2013]. Dentro dessa categoria, surge um novo tipo de memória chamado memória persistente (PM). Essa memória se caracteriza pela junção de duas propriedades: endereçamento por *bytes* e armazenamento durável, além de oferecer um desempenho próximo à DRAM. Essa abordagem oferece novas possibilidades, mas também apresenta novos desafios. Com a mudança do comportamento da memória (volátil para não volátil), a maneira de se utilizar a máquina também precisa passar por adaptações. Por exemplo, chaves criptográficas não podem mais ser guardadas diretamente na memória sem qualquer cuidado (já que não são apagadas em eventual reinicialização da máquina), partes da memória corrompidas por *bugs* de software não são reinicializadas, entre outros.

Essa mudança no comportamento da memória exige uma abordagem mais cautelosa e cuidadosa no desenvolvimento de sistemas que utilizam memória persistente. Os desenvolvedores precisam implementar medidas de segurança adicionais para proteger dados sensíveis e garantir a integridade do sistema. Além disso, ao utilizar memória persistente, surge o desafio adicional da consistência de estruturas de dados. Um exemplo clássico é a lista ligada, onde a atualização de múltiplos ponteiros em sequência pode causar problemas de inconsistência entre a memória persistente e as caches do processador. Caso ocorram falhas do sistema durante esse processo, a estrutura da lista ligada pode ficar corrompida, tornando os dados inválidos e causando instabilidades no sistema. Portanto, é fundamental empregar técnicas como o uso de barreiras de memória para garantir a ordem correta das operações de escrita, assegurando assim a consistência dos dados em memória persistente e mantendo a estabilidade e confiabilidade do sistema.

A utilização da memória persistente representa uma área promissora na busca por melhorias no desempenho e eficiência dos sistemas de armazenamento de dados. Entretanto, o desenvolvimento de aplicações que exploram esse tipo de memória tem sido desafiador, devido à complexidade associada. A utilização de ferramentas de baixo nível, como o `PMDK` (*Persistent Memory Development Kit*) [pmd 2023], pode ser extremamente difícil [Bastelli et al. 2022], apresentando diversos obstáculos que dificultam a criação de aplicações eficientes e confiáveis. Diante dessa problemática, torna-se evidente a necessidade de uma abordagem mais simplificada e amigável para o desenvolvimento de sistemas que utilizam memória persistente.

O `PM.NET` foi concebido como uma solução para esse desafio. Trata-se de uma biblioteca desenvolvida para abstrair as complexidades inerentes à utilização da memória persistente em aplicações desenvolvidas em C#. Ao adotar o `PM.NET`, os programadores podem usufruir dos benefícios da memória persistente sem se preocupar com detalhes de baixo nível, tornando o processo de desenvolvimento mais acessível, eficiente e produtivo.

Atualmente, a `Intel Optane DC Memory` (também conhecida como `Optane`) [Tyson 2019] talvez seja a opção mais amplamente disponível para a memória persistente. A `Optane` é um dispositivo endereçável por extitbytes e foi utilizado nos experimentos deste trabalho. Esta memória oferece alta velocidade de acesso e capacidade de armazenamento, tornando-a uma opção atrativa para aplicações que requerem acesso rápido a grandes volumes de dados. Neste trabalho, utilizamos a `Optane` como base para o desenvolvimento e testes da biblioteca `PM.NET`, aproveitando suas características

e abstrações de *hardware* para oferecer uma interface simples e orientada a objetos para o desenvolvimento em memória persistente. Apesar dos testes da PM.NET terem sido majoritariamente feitos com a Optane, a PM.NET tem como objetivo principal criar uma camada de abstração das complexidades da memória persistente, independentemente da tecnologia utilizada e, por isso, fornece uma interface genérica e que independe da escolha do *hardware*. Além disso, o uso da biblioteca para persistência de dados na PM mostrou um desempenho superior em relação a grandes bancos de dados amplamente utilizados, como o PostgreSQL e o SQLite, apresentando um ganho de aproximadamente 87% em cenários de uso com SSD e 44% em cenários com PM.

2. Trabalhos relacionados

Há trabalhos na literatura que se assemelham ao PM.NET ao proporem uma interface de programação mais amigável e de alto nível para o uso de memórias persistentes. Discutimos neste seção algumas destas propostas.

O Curundum [Hoseinzadeh and Swanson 2021] é uma interface de programação para memória persistente usando a linguagem Rust. Ele resolve problemas típicos que surgem ao usar a memória persistente, como ponteiros persistentes usados na memória volátil, bem como problemas de inconsistência da memória persistente. Para solucionar esses problemas, os autores adotaram a abordagem de realizar todas as alterações na memória persistente por meio de transações.

NV-Heaps [Coburn et al. 2011] é outra abordagem que também utiliza transações. Este é um sistema leve e de alto desempenho para acesso a memória persistente. Além do suporte a transações, ele implementa algumas estruturas de dados, como árvores, tabelas de *hash* e *arrays*, otimizadas para memória persistente.

Contemporâneo ao NV-Heaps, o Mnemosyne [Volos et al. 2011] é uma interface de programação para memória persistente. O trabalho possui dois focos principais: o gerenciamento da memória persistente e a garantia da consistência em casos de erros e falhas.

O trabalho de [Krauter et al. 2021] aborda o problema da memória persistente, onde, caso um software seja encerrado em um estado inconsistente, ele permanecerá inconsistente quando reiniciado. Para resolver esse problema, os autores, semelhantemente aos trabalhos anteriores, adotam transações. Em contraste com os trabalhos anteriores, esta abordagem é baseada em uma linguagem funcional pura (Haskell [Marlow et al. 2010]). Utilizando abstrações baseadas em mônadas para modelagem de transações e efeitos colaterais, esta abordagem é capaz de manter o estilo de programação praticamente inalterado quando comparado àquele que seria usado em uma aplicação com persistência utilizando dispositivos comuns. O PM.NET também oferece uma interface semelhante para os desenvolvedores que já estão acostumados a programar em C#, dessa maneira, diminui a curva de aprendizado para começar a utilizar a biblioteca.

O PMDK (*Persistent Memory Development Kit*) é uma biblioteca desenvolvida pela Intel, voltada para o desenvolvimento de aplicações que utilizam memória persistente em C e C++. Essa biblioteca oferece um conjunto de APIs que permitem o acesso eficiente e seguro à memória persistente, aproveitando as características únicas desse tipo

de memória. O PMDK facilita tarefas complexas, como gerenciamento de transações e recuperação de dados após falhas de energia, tornando mais acessível o desenvolvimento de aplicações que fazem uso dessa tecnologia. Apesar das vantagens oferecidas pelo PMDK, é importante destacar que seu uso pode ser desafiador e demanda um conhecimento detalhado sobre o funcionamento da biblioteca [Bastelli et al. 2022].

A LLPL (*Low-Level Persistence Library*) [llpl contributors 2022] é um projeto de código aberto que expõe acesso à memória persistente usando a linguagem Java. Dentre os trabalhos relacionados talvez seja o que mais se assemelha à nossa proposta já que em ambos os casos o objetivo é a criação de uma interface de programação de alto nível para memória persistente utilizando linguagens com gerenciamento automático de memória (Java e C#, respectivamente). No entanto, ao contrário do PM.NET, a LLPL não oferece uma interface amigável e é essencialmente uma camada superficial em Java que mapeia diretamente as funções do PMDK, não sendo orientada a objetos e carecendo de abstrações mais simplificadas e convenientes para o desenvolvimento de aplicações em memória persistente.

Esses trabalhos apresentam contribuições significativas no campo da memória persistente, com suas abordagens e focos específicos. No entanto, nosso trabalho se destaca ao desenvolver uma biblioteca em C#, aproveitando as abstrações de *hardware* fornecidas pelo PMDK e criando uma camada de abstração orientada a objetos para o uso da memória persistente. Tal abstração proporciona aos programadores uma forma mais simples e amigável de utilizar a memória persistente em suas aplicações desenvolvidas em C#, permitindo que se beneficiem das vantagens da memória persistente sem se preocupar com detalhes de baixo nível, facilitando o trabalho do desenvolvedor.

3. PM.NET

A biblioteca PM.NET foi desenvolvida para atender a falta de suporte para desenvolvimento para memória persistente em C# que, até onde pudemos verificar, não possui nenhuma biblioteca parecida. Ela oferece uma interface simplificada para lidar com objetos armazenados na memória persistente, logo seu principal objetivo é minimizar a quantidade de código necessário para o uso de memória persistente. Para atingir esse objetivo utilizamos uma abordagem que emprega o conceito de Programação Orientada a Aspectos (AOP, do inglês *Aspect-Oriented Programming*) [Elrad et al. 2001] para facilitar o uso da biblioteca. Explicamos essa abordagem em mais detalhes na seção a seguir.

3.1. AOP e proxies

Durante o desenvolvimento de uma nova aplicação, ou durante uma eventual migração de uma aplicação já existente para utilizar memória persistente, o desenvolvedor poderia ser obrigado a definir ou modificar os tipos dos seus objetos para que sejam compatíveis com esta tecnologia. Para minimizar o trabalho envolvido nessa transição, a biblioteca PM.NET emprega o conceito de AOP. Quando um método *set* de uma classe é chamado, a lógica de persistência é invocada automaticamente, sem interferência do desenvolvedor. Para atingir esse objetivo, utilizados *proxies*.

Um *proxy* é um objeto que substitui ou representa a localização de outro objeto para controlar o acesso a este [GAMMA et al. 2004]. Neste trabalho, utilizamos *proxies* para interceptar as chamadas aos objetos e alterar seu comportamento para acessar a

```
1  public class Interceptor : IInterceptor {
2    public void Intercept(IInvocation invocation) {
3      // Intercepta a chamada e redireciona a operação para a PM
4      if (invocation.Method.Name.StartsWith("set_")) {
5        invocation.Proceed(); // Continua a execução na memória
6      }
7      if (invocation.Method.Name.StartsWith("get_")) {
8        invocation.Proceed(); // Continua a execução para a memória
9      }
10   }
11 }
12
13 public class Pessoa {
14   public virtual int Idade { get; set; }
15 }
16
17 // Exemplo de uso do Proxy
18 var interceptor = new Interceptor();
19 var generator = new ProxyGenerator();
20 var proxy = generator.CreateClassProxy<Pessoa>(interceptor);
21
22 proxy.Idade = 10;
23 Console.WriteLine("Resultado do get: " + proxy.Idade);
```

Listagem 1. Propriedades virtuais no C#

memória persistente em vez da memória DRAM. Dessa forma, as chamadas aos métodos *set* e *get* são interceptadas e redirecionadas para à PM sem que o desenvolvedor precise escrever código específico algum[1].

Um código que demonstra criação de *proxies* para interfaces e classes concretas pode ser visto na Listagem 1. No início da execução, que ocorre na linha 18 até a linha 20, é criado um *proxy*. Na linha 22, é chamado o método `set` da propriedade `Idade`. Nesse momento, o *proxy* é ativado e o método `Intercept`, na linha 2, é chamado. O método `Intercept` verifica se o método é um *getter* ou *setter*[2]. Isso permite realizar modificações na execução convencional, que, no contexto deste trabalho, direcionam as chamadas para a PM.

3.2. Interface de uso

A biblioteca visa prover uma interface simplificada ao desenvolvedor, de modo que este nem precise estar ciente do uso de *proxies*, uma vez que a própria biblioteca disponibiliza uma implementação de uma *factory* [GAMMA et al. 2004] que abstrai a criação dos objetos *root* (raiz) persistentes. Um objeto *root* refere-se a um objeto na memória persistente que é diretamente acessível pelo programa e a partir do qual todas as outras referências de objetos persistentes podem ser alcançadas. Quando o `PM.NET` cria tais objetos e os armazena na memória persistente, ele constrói uma árvore para acompanhar as relações de dependência entre esses objetos. Os objetos *root* são o ponto de partida dessa árvore de referência, a partir do qual todos os outros objetos podem ser alcançados. Em resumo,

[1]Usamos a biblioteca Castle DynamicProxy [Castle contributors 2022] para criação de *proxies* em C#.

[2]Por padrão, o CLR (Common Language Runtime), no qual o C# se baseia, adiciona um prefixo “get_” e “set_” antes do nome da propriedade.

```
class Pessoa {
  public virtual int Idade { get; set; }
  public virtual string Nome { get; set; }
  public virtual char Sexo { get; set; }
}

IPersistentFactory factory = new PersistentFactory();
var pessoa = factory.CreateRootObject<Pessoa>("/pm/PessoaPM");
// A partir desse ponto, todas as chamadas para propriedades virtuais
// a esse objeto serão redirecionadas para a memória persistente.
pessoa.Idade = 42;
pessoa.Nome = "Bob";
pessoa.Sexo = 'M';
Console.WriteLine(pessoa.Idade);
```

Listagem 2. Código utilizando `PM.NET`

para que a criação ocorra com sucesso é preciso realizar duas alterações no código fonte do desenvolvedor:

- **Propriedades:** Declarar as propriedades como *virtuais* para que a interceptação seja possível.
- **Instanciação:** Para criar um objeto persistente, a instanciação deve ocorrer de forma diferente do tradicional uso da palavra reservada *new*. Para facilitar esse processo, é oferecida uma *factory* que permite que o desenvolvedor crie seus objetos por meio do método `CreateRootObject<T>(string pmFile)`. Esse método cria um objeto do tipo `T` e o envolve com um *proxy* para interceptar as chamadas às propriedades e armazenar os valores no arquivo PM definido pelo argumento `pmFile`. Objetos internos não precisam ser criados via *factory*, uma vez que todo objeto referenciado por um objeto persistente é automaticamente convertido em um objeto persistente.

O uso do `PM.NET` se dá então através da criação de objetos persistentes com a *factory* (que isola o desenvolvedor da aplicação dos detalhes de *hardware*). Para isso, é preciso instanciar a *factory* para criação dos objetos *root*. O `PM.NET` já fornece uma implementação padrão chamada `PersistentFactory`. A interface expõe o método `CreateRootObject<T>(string pmemFile)` que recebe um tipo `T` como *generics* e um argumento `string pmFile`, o método se encarrega de criar o *proxy* do tipo `T`, criar o arquivo especificado no argumento `pmFile` e o mapear em memória como demonstrado na Listagem 2. Após esse processo, o objeto estará criado e pronto para ser utilizado. Todas as operações de leitura e escrita das propriedades serão direcionadas para a memória persistente.

3.3. Transações

O `PM.NET` também oferece suporte a transações para garantir a consistência dos dados em memória persistente. Para isso, utiliza a técnica de *redo-log*, em que todas as alterações nos objetos persistentes são registradas em um arquivo de *log*. Esse arquivo de *log* contém informações sobre as modificações realizadas, como os novos valores de cada atributo dos objetos.

```
IPersistentFactory factory = new PersistentFactory();
var contasCorrentes =
  factory.CreateRootObject<ContasCorrentes>("/pm/ContaCorrente");

void TransacaoBancaria(decimal valor) {
  contasCorrentes.Transaction(() => {
    var contaBob = contasCorrentes.BuscarContaCorrente("Bob");
    var contaAlice = contasCorrentes.BuscarContaCorrente("Alice");

    contaBob.Saldo -= valor;
    contaAlice.Saldo += valor;
  });
}
```

Listagem 3. Exemplo de uso de transações no PM.NET

Antes que a transação seja considerada completa, um *flag* no arquivo de *log* precisa ser escrito indicando que todas as alterações foram registradas. Após essa marcação, os valores registrados no *log* são copiados para o arquivo original, efetivando as mudanças. Caso ocorra alguma falha durante a escrita do arquivo de *log*, ele é considerado inválido e a transação não é concluída, evitando que dados incorretos sejam persistidos. Além disso, caso haja uma falha durante a cópia dos valores do *log* para o arquivo original, o *log* ainda permanece válido, e na próxima inicialização do processo, a cópia é refeita para garantir a consistência dos dados. Testes práticos foram realizados para confirmar a eficácia desse mecanismo de segurança e recuperação em caso de falhas.

Para utilizar uma transação, basta chamar o método *Transaction* (Listagem 3), que simula uma transferência bancária onde o cliente fictício *Bob* envia um valor para a conta bancária de *Alice*. Neste exemplo, o valor da transferência é removido da conta de *Bob* juntamente em que o saldo na conta bancária de *Alice* é acrescido. É importante garantir que, em caso de algum erro durante esse processo, nem a conta de *Bob* tenha o valor do saldo reduzido sem que *Alice* tenha recebido a *Alice*, nem *Alice* tenha o saldo aumentado sem que o saldo de *Bob* tenha sido devidamente reduzido. Aqui uma transação é utilizada para assegurar as propriedades ACID durante a operação.

3.4. Coleta de lixo

Ao longo da vida de um sistema, o desenvolvedor cria múltiplos objetos na memória do computador. Alguns objetos podem ser mantidos na memória por um curto período, como um objeto criado e descartado dentro de um laço de repetição, enquanto outros podem permanecer na memória durante toda a vida do processo, sendo descartados apenas quando o processo é finalizado. Em linguagens de programação com gerenciamento manual de memória, como C++, o desenvolvedor é responsável por liberar manualmente a memória utilizada pelos objetos, o que pode levar a erros, como vazamentos de memória (*memory leaks*). No entanto, ambientes de desenvolvimento com gerenciamento automático de memória adotam um recurso conhecido como coletor de lixo (*garbage collector*).

O coletor de lixo é um recurso que gerencia automaticamente a memória do processo. De maneira geral, quando o desenvolvedor cria um objeto o coletor de lixo mantém um contador de quantas referências aquele objeto possui. Enquanto o objeto tiver referências, o coletor de lixo não toma nenhuma ação. Assim que o objeto perde todas as

referências, o coletor de lixo o remove da memória. O coletor também trata de algumas situações especiais como quando o contador de referência do objeto não chega a zero, mas o objeto deve ser coletado, como em casos de autorreferência e referências cíclicas. Isso elimina a responsabilidade do desenvolvedor de liberar manualmente a memória dos objetos, reduzindo o risco de erros humanos na programação.

A `PM.NET` gera um arquivo por objeto persistente. Ao longo da execução do programa quando não há mais referências aos objetos persistentes, esses arquivos são excluídos. O gerenciamento da memória persistente é realizado aproveitando o coletor de lixo nativo do .NET, que é altamente otimizado. Assim, a `PM.NET` não implementa um coletor de lixo próprio, mas sim estende o funcionamento daquele do .NET para incorporar as funcionalidades para o tratamento dos objetos persistentes.

Durante a execução do processo que utiliza o `PM.NET`, são realizadas duas abordagens distintas para realizar a coleta de lixo e remover os arquivos não utilizados na memória persistente. A primeira abordagem ocorre na inicialização do sistema. O `PM.NET` realiza uma varredura no diretório onde os arquivos são armazenados e constrói árvores de referência em memória. Cada arquivo *root* gera sua própria árvore de referência, permitindo mapear todas as referências dos objetos persistentes. Após esse mapeamento, obtem-se o número de referências de cada objeto persistente, que é armazenado para uso posterior. Com as árvores de referência construídas, é verificado se existem objetos persistentes sem referências. Nesse caso, os arquivos associados são excluídos.

A segunda abordagem para a coleta de lixo ocorre ao longo da execução do sistema. Antes de criar um objeto persistente a biblioteca `PM.NET` o envolve em outro objeto volátil, um *wrapper* criado pela classe chamado *CacheItem*. Quando o coletor de lixo do .NET determina que esse objeto volátil deve ser coletado, seu destrutor é chamado. Nesse momento, o `PM.NET` realiza uma segunda verificação para decidir se o arquivo associado ao objeto deve ser apagado. Essa segunda verificação é importante, pois nem todos os objetos descartados na memória devem ter o arquivo removido. Alguns objetos podem estar presentes na memória persistente sem terem sido carregados na memória principal (volátil) e podem ser referenciados por outros objetos. Nesse caso, o objeto em memória deve ser descartado, mas sem a exclusão do arquivo associado a ele, para evitar inconsistências no sistema.

Utilizando as duas abordagens em conjunto, a gestão eficiente da memória persistente é alcançada por meio das abordagens complementares mencionadas. Essa técnica assegura que a memória persistente permaneça livre de objetos não referenciados.

4. Avaliação experimental

Nesta seção, realizaremos uma análise detalhada do desempenho da biblioteca `PM.NET` em vários cenários e configurações. Essa análise é dividida em duas subseções: a primeira trata do ambiente experimental, descrevendo as condições dos testes, e a segunda foca nos resultados de desempenho, permitindo avaliar a biblioteca `PM.NET`.

4.1. Ambiente experimental

A máquina onde os experimentos foram executados utilizou o sistema operacional GNU/Linux com kernel versão 5.4.0, GCC 9.4.0, `PMDK` 1.8, e .NET 7.0. O *hardware*

da máquina inclui um processador Intel Xeon Gold 5317, com frequência base de funcionamento de 3.0 GHz (até 3.6GHzcom Turbo Boost), e 128GiB de DRAM, dividida em 4 módulos de 32GiB funcionando a 3.2GHz. O dispositivo de armazenamento empregado foi um SSD da Intel, o DC S4600 Series de 960GB.

O *hardware* utilizado para memória persistente é o *Intel Optane DC Memory*. Esse dispositivo de memória persistente permite acesso rápido aos dados e alta capacidade de armazenamento. Além disso, a Intel disponibiliza o `PMDK`, um conjunto de bibliotecas voltadas para o desenvolvimento em memória persistente, que foi utilizado como base para o desenvolvimento do `PM.NET`. Com isso, os testes foram conduzidos utilizando a memória persistente *Intel Optane DC*, num total de 512GiB, dividida em 4 módulos de de 128GiB funcionando a 3.2 GHz.

Cabe mencionar que a Intel anunciou no segundo trimestre de 2022 o abandono do *Intel Optane Memory*, mas isso não impacta diretamente esta pesquisa, uma vez que o objetivo principal do `PM.NET` é criar uma camada de abstração para as complexidades da memória persistente, independente da tecnologia subjacente utilizada. Portanto, a biblioteca `PM.NET` pode ser adaptada para outras tecnologias de memória persistente (como CXL [Jung 2022]), mantendo sua funcionalidade e utilidade.

4.2. Avaliação de desempenho

Para avaliar o desempenho e a eficiência da biblioteca `PM.NET`, além de alguns testes sintéticos, adaptamos um sistema de *Order Management System (OMS)*, que é comumente utilizado em corretoras de investimentos como um sistema responsável por gerenciar e executar ordens de compra e venda de ativos financeiros. Como é uma aplicação crítica para o funcionamento das corretoras, ela precisa lidar com eventuais falhas e manter a consistência do sistema. Por possuir fluxos que refletem as demandas de uma aplicação crítica que exige alto desempenho em persistência de dados, o OMS foi escolhido como uma base relevante para a avaliação da biblioteca `PM.NET`.

O OMS foi desenvolvido com a capacidade de executar com diferentes alvos para persistência, sendo eles os bancos de dados `PostgreSQL`, `SQLite`, `PM.NET` usando o `PMDK` e `PM.NET` usando arquivos mapeados em memória tradicionais. O sistema de arquivos foi montado em duas principais configurações, `ext4` que é uma das mais populares e amplamente utilizadas entre as opções disponíveis, e `DAX` (que é o acrônimo para Direct Access eXtensions). O `ext4` é uma evolução do `ext3` e é projetado para sistemas Linux, oferecendo maior desempenho, suporte a arquivos grandes e melhor confiabilidade. Por outro lado, o `DAX` é uma extensão que permite o acesso direto à memória de armazenamento, otimizando o acesso a dispositivos de armazenamento não volátil, como memória flash e SSDs (Solid-State Drives) e a própria PM. Com base nessas configurações, os testes foram executados com as seguintes opções:

1. `PostgreSQL` em um sistema de arquivos montado (com `DAX`) sobre a PM
2. `PostgreSQL` em um sistema de arquivos tradicional (ext4) sobre o SSD
3. `SQLite` em um sistema de arquivos montado (com `DAX`) sobre a PM
4. `SQLite` em um sistema de arquivos tradicional (`ext4`) sobre o SSD
5. `PM.NET` com arquivos mapeados em memória tradicional em um sistema de arquivos montado (com `DAX`) sobre a PM

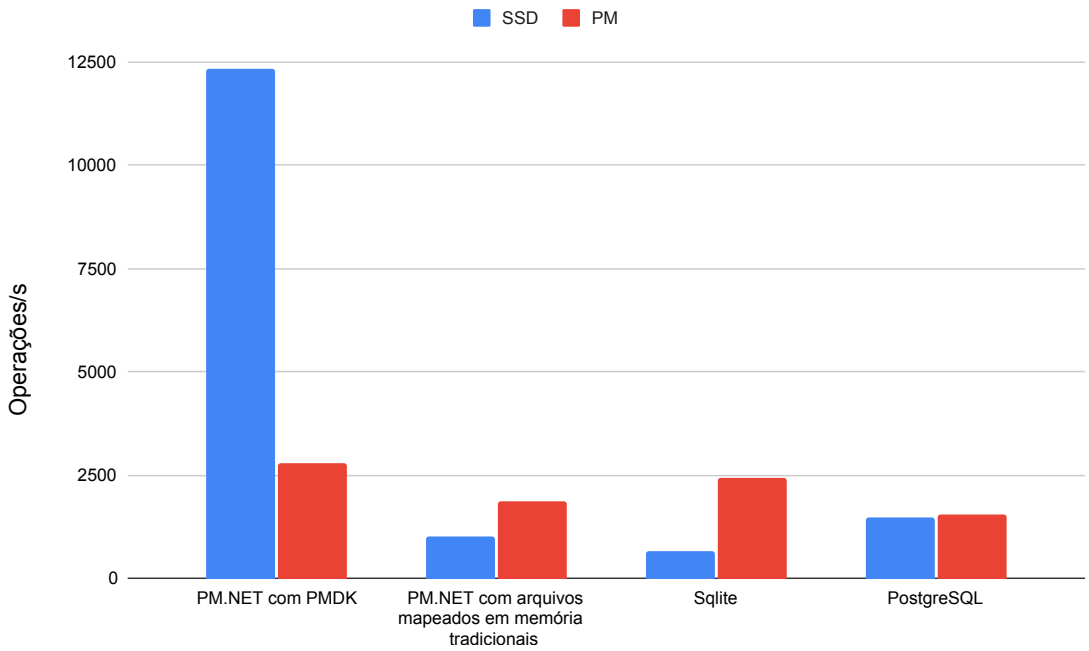

Figura 1. Desempenho OMS

6. `PM.NET` com arquivos mapeados em memória tradicional em um sistema de arquivos tradicional (`ext4`) sobre o SSD
7. `PM.NET` com `PMDK` em um sistema de arquivos montado (com `DAX`) sobre a PM
8. `PM.NET` com `PMDK` em um sistema de arquivos tradicional (`ext4`) sobre o SSD[3]

Os resultados foram medidos em termos do número total de *execution reports* processados por segundo (*i.e.*, vazão) como mostra a Figura 1. O termo "*execution report*" refere-se a uma mensagem que contém informações sobre a execução de uma ordem de compra ou venda de ativos financeiros no OMS. Essas mensagens são essenciais para o funcionamento das corretoras de investimentos, pois fornecem dados detalhados sobre cada transação realizada. Cada *execution report* inclui informações como o tipo de ordem (compra ou venda), o preço do ativo, a quantidade transacionada, a hora e data da execução e outros dados relevantes. Dessa forma, o número total de *execution reports* processados por segundo é uma medida direta da eficiência do sistema OMS em lidar com as transações. Valores mais altos nessa métrica indicam um melhor desempenho do sistema, pois ele é capaz de processar um maior volume de transações em um curto período de tempo. Adicionalmente, para cada nova ordem, foram executadas 6 atualizações, totalizando uma razão de 6/1.

Embora os bancos de dados `PostgreSQL` e `SQLite` tenham apresentado um aumento de desempenho ligeiramente superior quando executados sobre a memória persistente, esse ganho não foi significativo. Essa diferença pode ser atribuída ao fato de que esses bancos de dados não foram originalmente desenvolvidos para operar diretamente na PM.

Notavelmente, o desempenho do `PM.NET` ao utilizar arquivos mapeados em memória tradicionais não apresentou uma diferença significativa em relação aos bancos

[3]O `PMDK` pode identificar quando está sendo executado em um sistema que não possui PM e ainda ser capaz de funcionar corretamente em um SSD.

de dados consolidados. Nesse cenário, a biblioteca demonstrou um desempenho similar aos sistemas já estabelecidos. No entanto, quando foi empregado o PMDK, a situação se modificou. O PM.NET revelou um desempenho superior em relação aos bancos de dados.

Surpreendentemente, ao executar o PM.NET com o PMDK em um SSD, observamos um desempenho superior ao da execução diretamente na PM. Essa constatação contrária ao esperado foi atribuída à natureza da biblioteca PM.NET, que gera múltiplos arquivos, um para cada objeto persistente. Isso pode ter contribuído para que a criação dos arquivos no PMDK, na PM, fosse mais lenta, impactando o desempenho geral.

Para averiguarmos essa hipótese, realizamos um teste adicional focando em um cenário com um número significativamente maior de atualizações em relação à criação de arquivos. Nessa situação, com um fator de atualizações de 1000/1 por arquivo (em vez de 6/1 usado no experimento anterior), conseguimos diminuir o overhead do sistema de arquivos e, assim, aumentamos o desempenho em aproximadamente 137 vezes (de 2.784/s para 382.064/s). Esse ganho de desempenho fez com que o PM.NET com PMDK superasse o SSD em 20,65% (antes ele era 77,41% mais lento, na razão 6/1, descritos na Figura 1). Essa constatação indica a necessidade de aprimorar o PM.NET, priorizando o uso de um único arquivo. Mapear diversos objetos em um único arquivo, embora solucione o problema da multiplicidade de arquivos, pode introduzir alguns pequenos overheads ao buscar objetos distintos no mesmo arquivo, que não devem ser significativos.

5. Conclusão

A biblioteca PM.NET oferece uma solução viável e eficiente para o desenvolvimento em memória persistente com C#. Ao simplificar o acesso e o gerenciamento da memória persistente, a PM.NET permite que os desenvolvedores aproveitem o potencial da memória persistente em suas aplicações sem a necessidade de grandes modificações no seu estilo de programação ou para a adaptação de aplicações já existentes.

Resultados experimentais mostram que a abordagem utilizando o PMDK proporcionou um desempenho superior em relação aos bancos de dados relacionais PostgreSQL e SQLite. A inesperada superioridade do PM.NET no SSD, em certas condições, demonstra sua capacidade de evolução no futuro. Como atualmente, o PM.NET gera arquivos individuais para cada objeto há uma sobrecarga do sistema de arquivos. Como melhoria é preciso avaliar o uso de um único arquivo para múltiplos objetos persistentes, melhorando o desempenho.

Trabalhos futuros envolvem implementar recursos de migração para versionamento e evolução de objetos persistentes ao longo do tempo. Isso permitirá que os desenvolvedores atualizem estruturas de objetos persistentes de forma organizada, mantendo a compatibilidade com versões anteriores e facilitando a evolução das aplicações. Além disso, planeja-se realizar mais testes com *benchmarks* da literatura para medir o *overhead* do PM.NET.

Agradecimentos

Esta pesquisa é financiada por recursos do Projeto Temático FAPESP (processo Nº 2019/26702-8) e do Projeto de Pesquisa Regular FAPESP (processo Nº 2021/06867-2).

Referências

(2023). Persistent memory development kit (pmdk). https://github.com/pmem/pmdk. Acessado em: 06-Sep-23.

Bastelli, L., Baldassin, A., and Francesquini, E. (2022). Programando para memória persistente: Dificuldades, armadilhas e desempenho. In *Anais do XXIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 133–144, Porto Alegre, RS, Brasil. SBC.

Burks, A. W., Goldstine, H. H., and Neumann, J. v. (1982). Preliminary discussion of the logical design of an electronic computing instrument. In *The Origins of Digital Computers*, pages 399–413. Springer.

Castle contributors (2022). Dynamicproxy. http://www.castleproject.org/projects/dynamicproxy/. [Online; acessado em 21-Jul-2022].

Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. (2011). Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118.

Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32.

GAMMA, E., Helm, R., Johnson, R., and Vlissides, J. (2004). Padrões de projeto–soluções reutilizáveis de software orientado a objetos, 2004, ed.

Han, S.-T., Zhou, Y., and Roy, V. (2013). Towards the development of flexible non-volatile memories. *Advanced Materials*, 25(38):5425–5449.

Hennessy, J. L. and Patterson, D. A. (2014). *Organização e Projeto de Computadores: a interface hardware/software*, volume 4. Elsevier Brasil.

Hoseinzadeh, M. and Swanson, S. (2021). Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 429–442.

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

Krauter, N., Raaf, P., Braam, P., Salkhordeh, R., Erdweg, S., and Brinkmann, A. (2021). Persistent software transactional memory in haskell. *Proc. ACM Program. Lang.*, 5(ICFP):1–29.

llpl contributors (2022). Low-level persistence library. https://github.com/pmem/llpl. [Online; accessed 22-Jun-2022].

Marlow, S. et al. (2010). Haskell 2010 language report. *Available online http://www.haskell. org/(May 2011)*.

Tyson, M. (2019). Intel Optane DC Persistent Memory launched. Retrieved from https://hexus.net/tech/news/storage/129143-intel-optane-dc-persistent-memory-launched/. [Online; acessado em 01-Ago-2023].

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