Comparação Qualitativa das Interfaces de Programação em Memória Persistente

André Morato Pupin[1], **Emilio Francesquini**[2], **Alexandro Baldassin**[1]

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

{andre.pupin, alexandro.baldassin}@unesp.br

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

e.francesquini@ufabc.edu.br

Abstract. *Persistent Memories (PM) are fast, byte-addressable, and data-persistent. This new technology requires caution when being utilized. Due to its persistence, ensuring the integrity of the data structures after system failures is necessary. To use it efficiently and safely, several programming interfaces have been created with the aim of assisting the programmer in the task of building solutions that make use of persistent memory. In this article, we will qualitatively analyze some popular interfaces, observe how they choose to address current issues in persistent memory programming, and assess the degree of effectiveness in performing this task.*

Resumo. *Memórias Persistentes (PM) são rápidas, endereçáveis a byte e com persistência dos dados. Essa nova tecnologia requer cautela ao ser utilizada. Devido a sua persistência é necessário garantir a integridade das estruturas lá presentes após falhas do sistema. Para utilizá-la de modo eficiente e seguro, foram criadas várias interfaces de programação com o objetivo de auxiliar o programador na tarefa de construir soluções que se utilizam da memória persistente. Neste artigo é analisado de forma qualitativa algumas interfaces populares, observado como elas abordam os problemas atuais na programação com memória persistente e qual é o grau de eficácia na execução dessa tarefa.*

1. Introdução

As memórias persistentes são dispositivos que se encontram entre as memórias voláteis (e.g., DRAM) e os dispositivos não voláteis (e.g., SSD), possuindo endereçamento em bytes, alta velocidade de resposta e com garantia de persistência dos dados [Baldassin et al. 2021]. Com a chegada dos dispositivos persistentes no mercado, foi criada a necessidade de interfaces para a programação persistente, visto sua dificuldade de se programar [Jinglei Ren 2017].

Existem artigos que compararam essas interfaces principalmente pelo seu desempenho, porém o desempenho não é necessariamente o fator predominante e, normalmente, o equilíbrio entre a usabilidade, desempenho e portabilidade determina o sucesso de uma solução. Esse artigo tem o objetivo de comparar de forma qualitativa três interfaces de programação: i) Intel PMDK (libpmemobj); ii) Go-Pmem, visto em [Jerrin Shaji George 2020]; e iii) Corundum, visto em [Morteza Hoseinzadeh 2021].

```
1  TOID(struct Node) createNewNode(PMEMobjpool *pop, int data) {
2    TOID(struct Node) p_newNode;
3
4    TX_BEGIN(pop) {
5      p_newNode = TX_NEW(struct Node);
6      D_RW(p_newNode)->data = data;
7      D_RW(p_newNode)->p_next = TOID_NULL(struct Node);
8    } TX_END
9    return p_newNode;
10 }
```

Figura 1. Função de criação de nós de uma lista ligada em libpmemobj usando macros.

Neste artigo é analisado como cada uma das interfaces aproveita dos pontos positivos de suas respectivas linguagens para facilitar a programação para os dispositivos persistentes, além de analisar a documentação existente. Em geral observamos que o nível de abstração das interfaces é baixo e exige um domínio muito grande do programador.

2. Apresentação das Interfaces

Nesta seção serão apresentadas brevemente as 3 interfaces estudadas neste artigo: PMDK, Go-Pmem e Corundum. Para facilitar a descrição, é apresentado o código para a criação de um elemento de uma lista encadeada em cada interface.

Persistent Memory Development Kit (PMDK) é um conjunto de bibliotecas desenvolvidas pela Intel para lidar com memórias persistentes. A principal delas, *libpmemobj*, utiliza um sistema de *pool* de memória persistente representado como um arquivo. Uma vez inicializado esse pool, é definido sua estrutura e alocado um objeto raiz que é utilizado como o ponto de entrada para as outras estruturas no pool. A *libpmemobj* não possui um sistema de checagem de ponteiros ou operações, sendo uma extensão da linguagem C baseada em macros. Por exemplo, é possível criar um ponteiro não volátil para uma área volátil da memória. Note que, neste caso, o ponteiro apontará para uma área de memória que não existirá mais quando o objeto for acessado depois de uma queda do sistema. A Figura 1 mostra como um novo nodo é criado no PMDK. A macro `TOID` (linhas 1 e 2) serve para especificar ponteiros persistentes. A alocação em si deve ser feita dentro de uma transação (linhas 4-8). Para escrever um ponteiro persistente é utilizado a macro `D_RW` (linhas 6 e 7).

Go-Pmem é um híbrido entre uma nova linguagem, baseada em *Go*, e uma biblioteca. É utilizada uma estrutura de *Pool* com objeto raiz. Foram adotados ponteiros diretos, não precisando de nenhuma macro ou função adicional para a conversão de ponteiros. O coletor de lixo da linguagem Go foi revisado e expandido, conseguindo realizar sua operação na memória persistente, porém não é garantido o funcionamento ao realizar operações entre memória não volátil e a volátil. A Figura 2 mostra a criação do nó numa lista ligada. O alocador principal da linguagem é o `pnew()` (linha 3) que retorna uma referência para a variável. O Sistema de transações é realizado por blocos de funções lambda (linhas 5-8), não sendo necessário especificar o que será salvo. Ele suporta apenas o *undo log* apesar de aparentar ter suporte para escolha do formato de log (i.e., redo log).

```
1  func createNewNode(data int) Node* {
2      var newNode *Node
3      newNode = pnew(Node)
4
5      txn("undo") {
6          newNode.data = data
7          newNode.next = nil
8      }
9      return newNode
10 }
```

Figura 2. Função de criação de nós de uma lista ligada em Go-Pmem.

Corundum é uma biblioteca para *Rust*. Também utiliza um sistema de pool com objeto raiz. Usando o ambiente Rust temos o *borrow checker* que é utilizado para impedir vazamentos de memória e o sistema de *Traits* que permite a restrição de operações com dados que seriam consideradas não seguras. Algumas operações restringem a alteração de valor de dados fora de transações. A Figura 3 mostra uma função de criação de nós. O alocador utilizado é o `Pbox::new()`; o resultado da alocação está sendo empacotado numa enumeração. O Corundum possuiu vários alocadores que garantem algumas proteções cada. Não é possível observar o bloco de alocação, pois ele ocorre pela chamada da função `Allocator::transaction()`, que recebe a função a ser executada em transação.

```
1  fn createNewNode(data: T, j: &Journal) {
2          let Some(ref node) = Some(Pbox::new(
3              Node {
4                  data,
5                  next: None,
6              }, j, ));
7      }
```

Figura 3. Função de criação de nós de uma lista ligada no Corundum.

3. Comparação entre as Interfaces

São usados quatro aspectos para comparar as interfaces de programação: i) legibilidade e Escrita, ii) segurança, iii) curva de aprendizado e iv) suporte da comunidade. Em cada tópico falaremos como cada uma das interfaces é avaliada e as razões para tal avaliação.

A *libpmemobj* é bem verbosa, mesmo com o uso de macros. Apesar da leitura não ser algo difícil, a escrita é. A exceção é a implementação dos blocos transacionais que são bem claros. Há um aumento de 31% em relação ao mesmo código volátil em *C*. O *Go-Pmem* possui uma legibilidade excelente. Um código em *Go* e *Go-Pmem* não são muito distintos, devido ao fato de se usar ponteiros diretos. Isso pode se tornar um aspecto negativo caso o código não seja bem documentado. No caso do *Corundum*, ao comparar o código não persistente com o persistente, é possivel observar que parte da dificuldade da legibilidade e escrita vem da linguagem adotada. O sistema de *borrow checker* necessita de um programador experiente.

A *libpmemobj* herda todos os problemas de segurança da linguagem *C*. Segurança fica totalmente ao encargo da capacidade do programador. O *Go-Pmem* possuiu o coletor de lixo que resolve os problemas de vazamento de memória. Porém, há reclamações em relação a falta de atomicidade de transação em algumas funções [1] . No caso do *Corundum*, podemos afirmar que o código ao passar pelo compilador terá uma garantia forte que o código será seguro.

Na *libpmemobj* temos uma dificuldade normal a usar a biblioteca, porém a maior dificuldade está em escrever códigos seguros. No *Go-Pmem* há a mesma dificuldade de escrever códigos seguros. A linguagem *Rust* possuiu uma curva de aprendizado muito elevada de acordo com [Kelsey R. Fulton 2021], logo é difícil de julgar quanto o *Corundum* interfere nessa curva de aprendizado.

Com relação ao suporte, o conjunto de bibliotecas *PMDK* continua a ser desenvolvido ativamente no *Github*. O repositório do *Go-Pmem* foi arquivado em 07/03/2023. O respositório do *Corundum* teve seu última atualização no dia 28/04/2022.

4. Conclusão

Cada interface de programação usa os recursos de suas linguagens com sucesso, porém em cada uma delas há pontos negativos que impedem que as interfaces deem um passo além para o uso de um público maior que esteja fora da circulo de pesquisa de memória persistente. Na *libpmemobj* temos códigos muito sujeitos a erros, devido a reponsabilidade que é colocada ao programador, sendo ainda muito baixo nível para uma aplicação grande. No *Go-Pmem* temos falta de suporte que desencoraja o uso em projetos maiores que necessitem de maior confiabilidade, numa interface que se destaca em vários outros pontos. Por fim, no *Corundum* temos uma curva de aprendizado elevada principalmente devido a linguagem adotada, apesar da melhor segurança entre as interfaces mais populares.

Agradecimentos. Os autores agradecem à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos nº 2018/15519-5, 2023/05019-3 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).

Jerrin Shaji George, Mohit Verma, R. V. P. S. V. (2020). go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference.* USENIX.

Jinglei Ren, Qingda Hu, S. K. T. M. (2017). Programming for non-volatile main memory is hard. In *Proceedings of APSys '17.*

Kelsey R. Fulton, Anna Chan, D. V. M. H. M. L. M. (2021). Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Proceedings of the Seventeenth Symposium on Usable Privacy and Security.* Usenix.

Morteza Hoseinzadeh, S. S. (2021). Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ASPLOS.*

[1] Exemplo : https://github.com/jerrinsg/go-pmem/issues/17