Implementação e Comparação de Ferramentas de Programação para Memória Persistente

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

Abstract. *This article covers the development and comparative analysis of the programming interface of two libraries. The fstream library widely used in C ++ and PMDK, a library created by Intel to facilitate the development of systems using persistent memory and files mapped in memory. The use of fstream was simple, the library was used to serialize the data in binary format. In contrast, development with the PMDK required more effort. The development was affected by the lack of support for complex types such as strings, which required the creation of an intermediate layer. In this article we report the experience of using these tools and highlight the difficulties encountered.*

Resumo. *Esse artigo aborda o desenvolvimento e análise comparativa da interface de programação de duas bibliotecas. A biblioteca fstream amplamente utilizada no C++ e PMDK, biblioteca criada pela Intel para facilitar o desenvolvimento de sistemas utilizando memória persistente e arquivos mapeados em memória. A utilização do fstream foi simples, a biblioteca foi utilizada para serializar os dados em formato binário. Em contrapartida o desenvolvimento com o PMDK exigiu mais esforço. O desenvolvimento foi afetado por falta de suporte a tipos complexos como string, que exigiu a criação de uma camada intermediária. Neste artigo relatamos a experiência de uso dessas ferramentas e evidenciamos as dificuldades encontradas.*

1. Introdução

Podemos dividir os sistemas de armazenamento em dois tipos: persistes e voláteis. A memória volátil é tipicamente de rápido acesso, contudo os dados se perdem quando ocorre uma interrupção no fornecimento de energia. A memória persistente é normalmente mais lenta, mas os dados são mantidos mesmo quando não há energia. Por muitos anos o desempenho das memórias persistentes tem sido um gargalo de desempenho nos computadores, e é um tópico de pesquisa bem ativo [Chen et al. 2009]. Atualmente, existem várias maneiras de trabalhar utilizando memória persistente. O modo mais simples para utilizar a memória persistente é utilizando arquivos. Um desenvolvedor comum está bem familiarizado com esse modelo de programação. Outra forma de desenvolver para memória persistente abordada nesse artigo é utilizando a biblioteca PMDK. O PMDK utiliza internamente um recurso do sistema operacional chamado arquivos mapeados em memória. Esse recurso pode ser utilizado também com arquivos comuns. A diferença

é que os arquivos mapeados em memória do PMDK têm por trás uma memória persistente e ele já tem implementado algumas funcionalidades como transações. Esse trabalho identifica e compara como a interface de programação do PMDK destoa em relação aos arquivos tradicionais com fstream.

2. Memória persistente

Ao longo dos anos a memória persistente foi disseminada e popularizada através dos discos rígidos (HD). Por sua natureza mecânica HDs são consideravelmente mais lentos quando comparados à *Random Access Memory* (RAM) e tem um maior gasto de energia [Chen et al. 2009]. Isto os torna uma fonte recorrente de limitação de desempenho. Recentemente a Intel anunciou uma tecnologia de memória persistente Intel Optane em julho de 2015 que foi lançada no mercado em abril de 2017. O lançamento foi acompanhado por uma biblioteca chamada PMDK. O PMDK auxilia o desenvolvimento de aplicações que utilizam memória persistente. A biblioteca foi escrita em C e posteriormente portada para C++. A tecnologia pode ser comparada com a RAM por ser endereçável por byte e é uma tecnologia emergente [Hu and Matheus 2018, Rudoff 2017]. Essa biblioteca foi escolhida para estudo neste artigo devido a sua popularidade e suporte da comunidade.

2.1. Acesso a arquivos

Um software frequentemente precisa persistir os dados para que na próxima execução ele consiga recuperar as informações. A maneira mais comum de implementar tal funcionalidade é utilizando arquivos. Virtualmente todas as linguagens de programação provêm uma camada de abstração para manipulação de arquivos. Em C++ utilizamos as classes fornecidas pelo `fstream`. A biblioteca `fstream` foi escolhida por ser amplamente utilizada e ter ampla documentação.

2.2. PMDK

O PMDK é uma coleção de bibliotecas escritas em C e C++. O PMDK abstrai parte da complexidade para trabalhar usando arquivos mapeados em memória. Com esse modelo, é possível trabalhar com memória persistente endereçáveis por bytes [Scargall 2020]. Para esse trabalho utilizamos, principalmente, o recurso de transações do PMDK. A biblioteca responsável pelas transações do PMDK é o `libpmemobj`.

3. DBKV

Desenvolvemos uma biblioteca chamada DBKV (KeyValue DataBase) que é semelhante a um banco de dados chave/valor com as operações básicas. São elas: busca, inserção, e remoção de valores. Tanto a chave como os valores são *strings*. A biblioteca DBKV fornece uma interface única com as operações supracitadas. No ato da compilação é possível escolher a implementação entre: arquivos comuns e PMDK. A tecnologia utilizada foi C++ devido a sua vasta documentação e ampla utilização.

Foram criadas duas implementações do DBVK, uma utilizando as classes fornecidas pelo `fstream` do C++ e outra utilizando o PMDK. Ambas as implementações utilizam o mesmo header (`KV.h`)) para garantir que interface seja exatamente a mesma.

Inserção de elementos A inserção dos elementos, na implementação usando arquivos, foi feita criando uma simples *struct* cujos membros são duas *strings* (chave e valor). A *struct* é serializada em um arquivo binário para o salvamento. Já no caso do PMDK houve dificuldades com o desenvolvimento. O PMDK não oferece uma maneira simples de persistir vetores. Como o tipo `string` é um vetor de `char` então trabalhar com *strings* exige cuidados. Para resolver o problema criamos uma classe auxiliar chamada `pmem_array` que se comporta como uma lista encadeada de `char`. Com isso, o algoritmo para a inserção de elementos é similar aos algoritmos clássicos para listas encadeadas. A classe auxiliar funciona internamente utilizando a biblioteca `libpmemobj` que é especializada em transações.

Remoção de elementos A remoção de elementos dos arquivos foi implementada de maneira simples. Ela funciona através de cópias de elementos entre arquivos versionados. Após a cópia, o arquivo com a versão antiga da base de dados é descartado. Apesar de haver soluções com melhores desempenhos, a simplicidade desta implementação justifica o seu uso. Já no caso do PMDK, a remoção é feita com um algoritmo simples de remoção de nós em listas encadeadas: encontra-se o elemento a ser removido e os ponteiros apropriados são ajustados.

Busca de elementos A busca foi implementada usando o algoritmo de busca linear. A cada iteração do laço é comparado a chave que está sendo procurada a cada elemento do banco de dados. Esse é o caso tanto para a versão baseada em arquivos quanto a versão baseada no PMDK.

4. Discussão

Após a criação do DBKV ficaram evidentes as diferenças na utilização entre as bibliotecas de memória persistente. A principal diferença encontrada é a montagem de ambiente, sua configuração envolve mudar parâmetros do kernel, como criar uma partição para memória persistente, configurar o Direct Access Filesystems (DAX), entre outros [Scargall 2020]. Também há a opção de comprar o hardware necessário como Intel® Optane™ Persistent Memory. Arquivos comuns e o seu uso já são disseminados e contam com o suporte de linguagens e de sistemas operacionais por padrão. Assim, permitirem o início do desenvolvimento sem qualquer tipo de preparação do ambiente. Arquivos comuns já são uma ferramenta estabelecida com suporte a todos os compiladores, linguagens e sistemas operacionais.

Depois do ambiente simulado preparado para executar o primeiro programa utilizando o PMDK, também é necessário ter um arquivo mapeado em memória. Esse recurso pode ser criado via linha de comando ou via API. A interface para a criação é simples e exige apenas o caminho e layout do arquivo.

```
const char *file = "/pmem/myfile";
const std::string LAYOUT = "p";
pool<pmem_array> pop = pool<pmem_array>::open(file, LAYOUT);
```

Quando chamada, essa função cria um mapeamento do arquivo em memória. Caso já exista o mapeamento, irá ocorrer uma exceção que precisa ser tratada. Ao trabalhar com

arquivos tradicionais esse problema não existe. O comportamento padrão ao escrever em um arquivo é o arquivo ser criado caso não exista. O único problema que pode vir a ocorrer é em caso de leitura onde o arquivo não existe, nessa situação uma exceção irá ocorrer que precisa ser tratada.

Durante o desenvolvimento com PMDK também tivemos dificuldades para utilizar o tipo *string*. O PMDK não oferece uma maneira simples de persistir vetores, como a *string* é um vetor de *char* tivemos o mesmo problema com ela. O uso do PMDK para objetos complexos é simples de ser utilizado, basta encapsular os valores em uma struct e passar a referencia dela para o PMDK para torna-la persistente.

Na implementação que usa arquivos tradicionais foi empregada a serialização binária. Além disto, não encontramos uma maneira simples de remover apenas um registro. Um arquivo pode ser escrito com modo *append* no qual os dados são inseridos no final do arquivo ou *write* no qual o arquivo é reescrito completamente. Não existe um modo de abertura de arquivo para remoção de caracteres. Para deletar um registro em um arquivo é necessário implementar manualmente usando técnicas como aquelas usadas por sistemas gerenciadores de bancos de dados. Por outro lado, o PMDK oferece diretamente uma implementação baseada em transações de forma fácil e simples.

5. Conclusão

De maneira geral, utilizar a escrita em arquivos convencionais é mais simples devido a ampla documentação e o `fstream` ser um biblioteca madura com muitos anos de utilização. Em contrapartida, o PMDK oferece uma barreira de entrada mais alta, principalmente para a configuração do ambiente de desenvolvimento. Além disso, o PMDK ainda oferece um suporte limitado para tipos complexos como *strings*. Nestes casos, é necessária a implementação de uma camada intermediária para auxiliar no desenvolvimento. O PMDK, apesar de maduro o suficiente para experimentações e pesquisa, é uma ferramenta que ainda precisa evoluir para ser considerada como uma possível substituta ao tradicional uso de arquivos ou de bancos de dados.

Referências

Chen, F., Koufaty, D. A., and Zhang, X. (2009). Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):181–192.

Hu, X. and Matheus, A. (2018). Persistence parallelism optimization: A holistic approach from memory bus to rdma network. In *IEEE/ACM International Symposium on Microarchitecture*.

Rudoff, A. (2017). Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40.

Scargall, S. (2020). *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature.