

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

Otávio Scarparo Souza, Bruno Honorio, Alexandro Baldassin

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

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

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

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

1. Introdução

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

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

2. Alocadores Persistentes

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

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

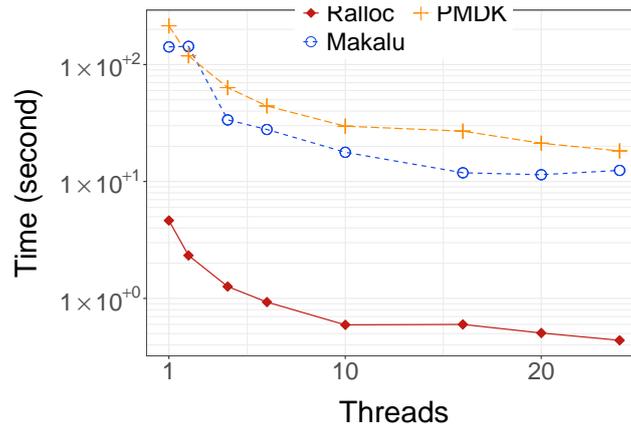
O alocador deve, primeiramente, alocar uma região de memória persistente, atualizar seus metadados (também persistentes) para refletir a nova alocação, e então retornar o endereço alocado para a aplicação. Considere, no entanto, que haja um *crash* (e.g., queda de energia) antes da aplicação armazenar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

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

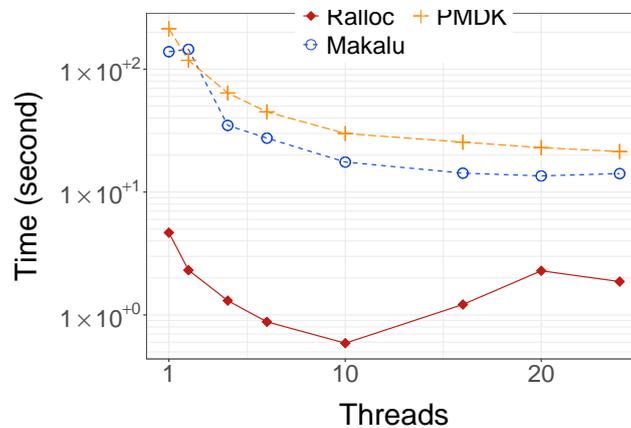
Makalu: É um alocador que utiliza coleta de lixo para prevenir vazamentos de memória, garantindo a integridade da estrutura da heap persistente. A coleta ocorre em dois momentos: i) quando uma aplicação está ativa e pode modificar metadados e ii) quando, após uma falha, os dados são restaurados e voltam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

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

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



(a) Limitado a um único NUMA node.



(b) Distribuído entre os 2 NUMA nodes.

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

3. Avaliação Experimental

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

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

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

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

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

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

4. Conclusão

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

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

Referências

- Baldassin, A., Barreto, J. a., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7).
- Bhandari, K., Chakrabarti, D. R., and Boehm, H.-J. (2016). Makalu: fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694.
- Cai, W., Wen, H., Beadle, H. A., Kjellqvist, C., Hedayati, M., and Scott, M. L. (2020). Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management, ISMM 2020*, page 60–73, New York, NY, USA. Association for Computing Machinery.
- Scargall, S. (2020). *Programming Persistent Memory - A Comprehensive Guide for Developers*. Apress, 1st edition.