
Introdução à Programação com Memória Persistente

Alexandro Baldassin (UNESP)
alexandro.baldassin@unesp.br

Emilio Francesquini (UFABC)
e.francesquini@ufabc.edu.br

Apresentação dos palestrantes

Alexandro Baldassin é professor associado no Departamento de Estatística, Matemática Aplicada e Computação (DEMAC) da Universidade Estadual Paulista (UNESP), Rio Claro. Recebeu o título de Doutor em Ciência da Computação pelo IC-UNICAMP em 2009. Atua na pesquisa de métodos para sincronização para programas paralelos e mais recentemente na otimização de sistemas com memória persistente.



Emilio Francesquini é professor adjunto no Centro de Matemática, Computação e Cognição (CMCC) da Universidade Federal do ABC (UFABC). Recebeu o seu doutorado em Ciência da Computação em dupla titulação pela Universidade de São Paulo (USP) e pela Université de Grenoble-Alpes, França (2014). Desde 2015 investiga o uso de memória persistente e o seu impacto no desenvolvimento de sistemas computacionais.

Repositório com código

- Para este curso criamos um repositório onde colocamos os fontes além de uma imagem do docker já configurada para os testes.
 - Ela contudo apenas finge ter memória não volátil baseada em disco
- Repositório Git:
 - <https://github.com/baldas/minicurso-memoria-persistente/tree/eradsp-2023>
 - git clone <git@github.com>:baldas/minicurso-memoria-persistente.git
 - cd minicurso-memoria-persistente/
 - git checkout eradsp-2023
 - Caso deseje, a imagem pré-construída (porém sem os scritps e códigos) está em:
 - https://hub.docker.com/r/francesquini/erad_pmem
 - Nele você encontra scripts para construir e executar a sua própria imagem do Docker
 - O diretório docker_home já é mapeado automaticamente quando o Docker é executado o que torna o processo de desenvolvimento mais tranquilo (já que permite o uso do seu editor favorito, etc...)

Smoke test

```
$ git clone git@github.com:baldas/minicurso-memoria-persistente.git  
...  
$ cd minicurso-memoria-persistente/  
$ git checkout eradsp-2023  
$ ./run_image.sh Vai levar um tempinho na primeira execução...  
...  
erad@docker-pmem:~$ cd pmdk/pmkv/  
erad@docker-pmem:pmkv$ make  
...  
erad@docker-pmem:~$ ./pmemkv  
key: key1 value: value1  
key: key3 value: value3  
key: key2 value: value2  
erad@docker-pmem:pmkv$
```

Conteúdo

- Motivação e importância
- Conceitos de persistência
- Desafios na programação
- Suporte do sistema operacional
- Intel PMDK
- Novas abordagens para programação

Parte 1 - Memória Persistente: O que é, onde vive e o que come?

O que é memória persistente?

- Também chamada de *storage class memory* ou *não-volátil*
 - Dados são mantidos mesmo em caso de falha de energia
- Alia atributos de dispositivos de armazenamento e memória
 - É **persistente**, como discos rígidos e SSDs
 - É **endereçável a byte**, como memória volátil (e.g., DRAM)
- Vantagens
 - Capacidade maior que DRAM
 - Velocidade maior que dispositivos baseados em blocos
- Mas por que apenas agora?
 - Novas tecnologias permitem um tempo de acesso direto próximo ao da DRAM
 - Disponível comercialmente!

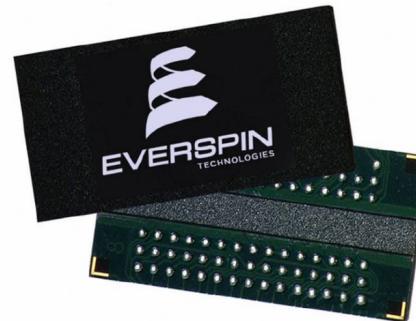
Um pouco de história...

- Tubo de raios catódicos (46) — Anos 40
- Core Memory (47) — Anos 50 e 60
- FE-RAM (52) — Desenvolveu-se no fim dos anos 80
- DRAM (68) SRAM (64) — Anos 70 até hoje
- PCM (69) — Apenas recentemente explorada
- FLASH (84) — Popularizou-se nos anos 90
- STT-RAM (96) — Início da exploração comercial nos anos 00
- 3D XPoint - (2019)

Ser volátil não é o padrão!

In the wild...

- Sandisk ULLtraDIMM SSDs - **Flash** (200-400GB)
- Everspin - **ST-RAM** (64MB)
- Viking ArxCis-NV - **DRAM + Bateria + Flash** (16GB)
- Netlist NVvault NVDIMM - **DRAM + Bateria + Flash** (8GB)



Intel Optane DC

Lançado em abril de 2019 pela Intel



Table 12-1. Latencies for Accessing Intel® Optane™ DC Persistent Memory Modules

Latency	Intel® Optane™ DC Persistent Memory Module	DRAM
Idle sequential read latency	~170ns	~75ns
Idle random read latency	~320ns	~80ns

Table 12-2. Bandwidths per DIMM for Intel® Optane™ DC Persistent Memory Modules and DRAM

Per DIMM Bandwidths	Intel® Optane™ DC Persistent Memory Module	DRAM
Sequential read	~7.6 GB/s	~15 GB/s
Random read	~2.4 GB/s	~15 GB/s
Sequential write	~2.3 GB/s	~15 GB/s
Random write	~0.5 GB/s	~15 GB/s

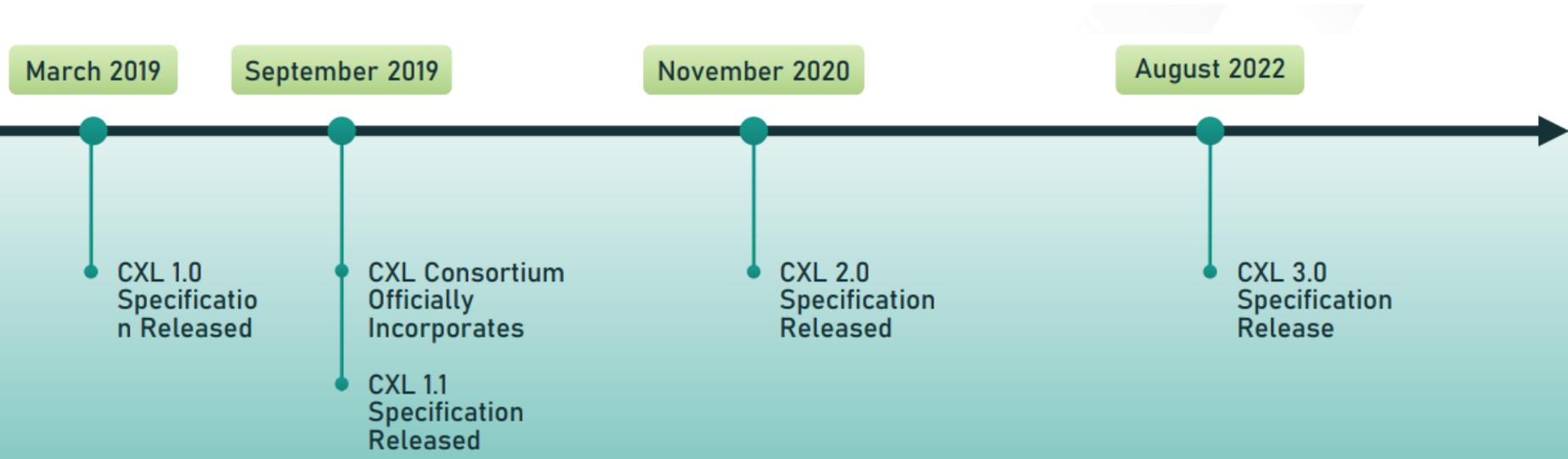
CXL - Compute Express Link

- Rede de interconexão aberta, padrão da indústria
 - Permite conectar CPU a dispositivos (E/S, aceleradores, memória) com uma alta largura de banda e baixa latência
- Alguns membros:



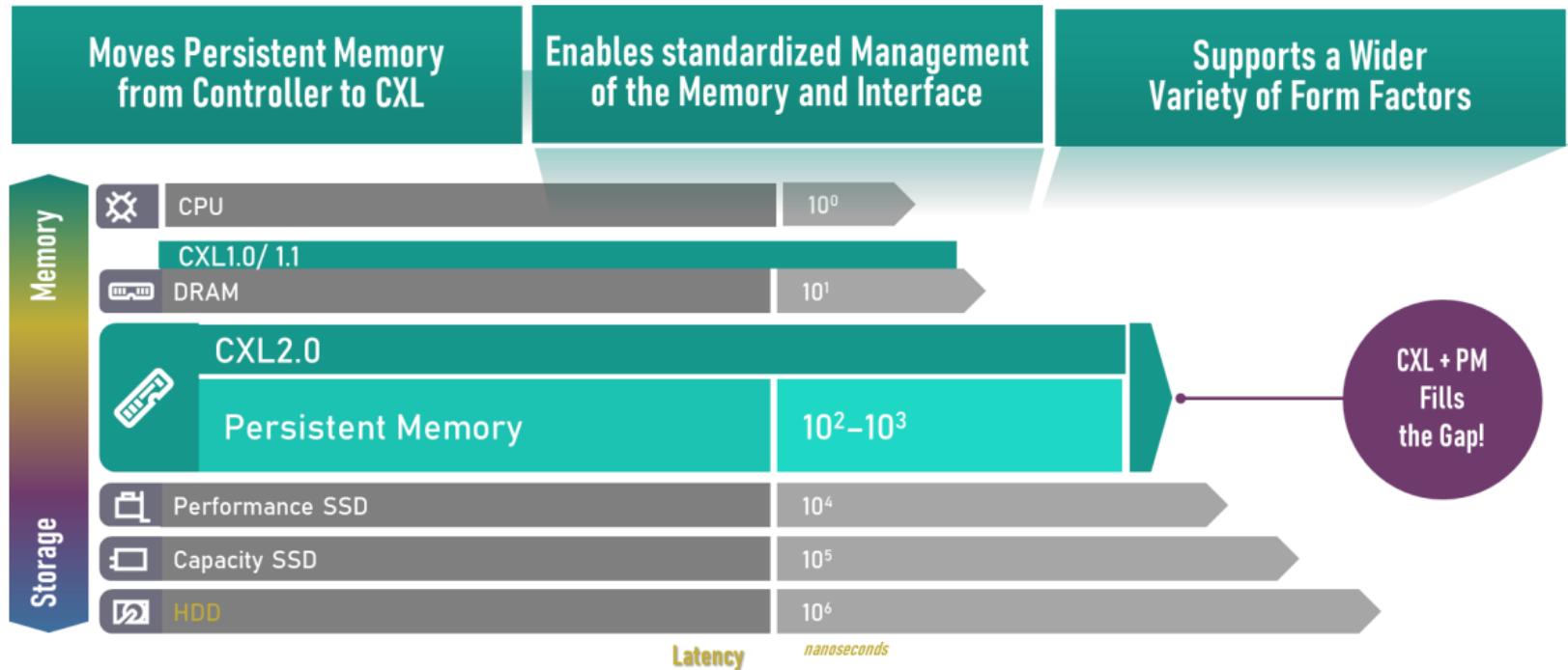
CXL - Timeline

- Evolução da tecnologia:

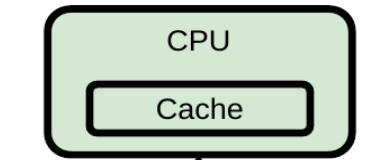
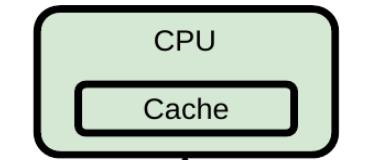
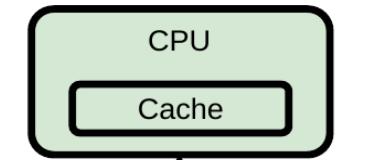
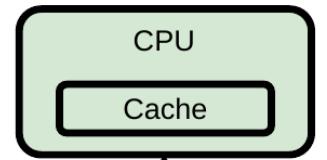


- A partir da versão 2.0, PM é suportada

CXL 2.0 - Suporte para PM



Papéis de PM na hierarquia de memória



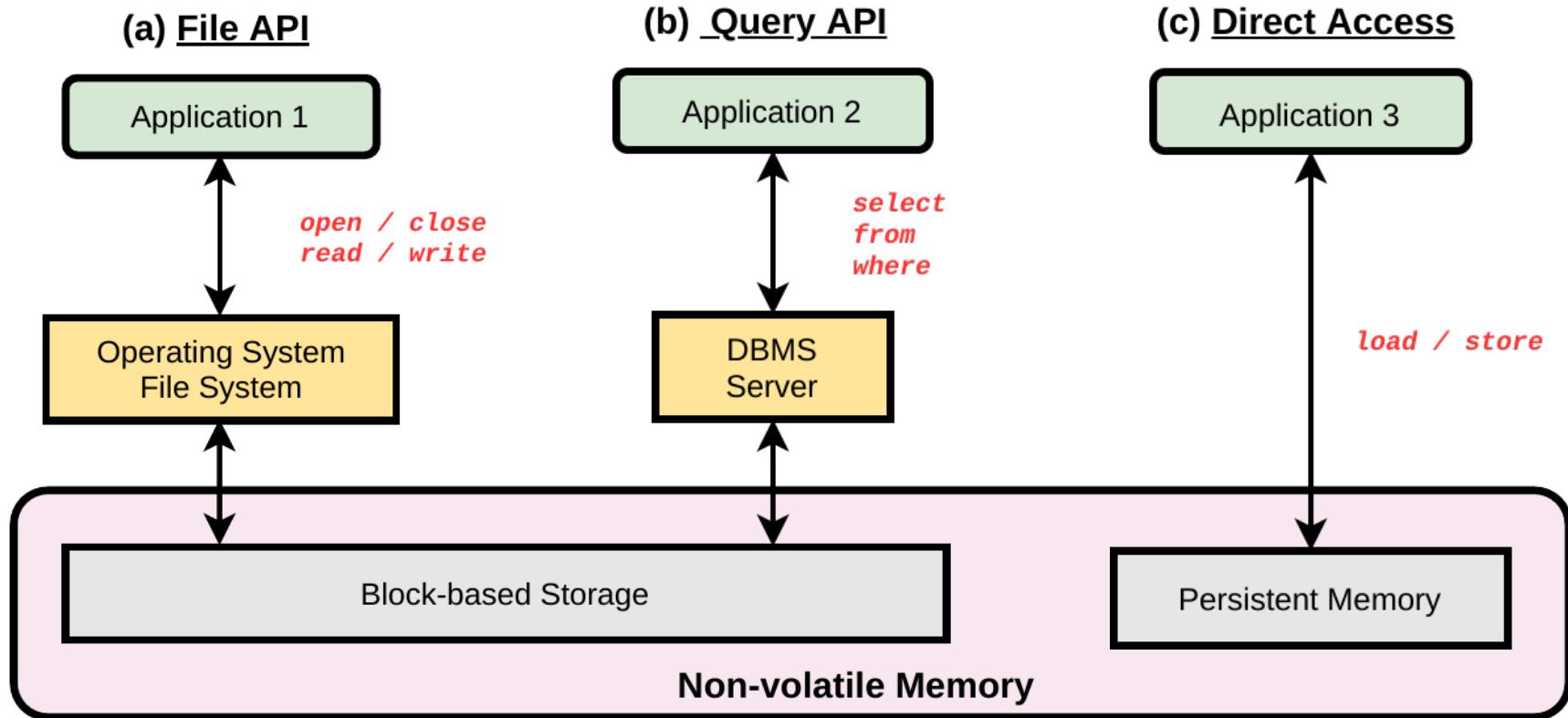
(a) Traditional Hierarchy

(b) Replacing DRAM

(c) DRAM as a cache

(d) DRAM and PM

Interfaces para programação



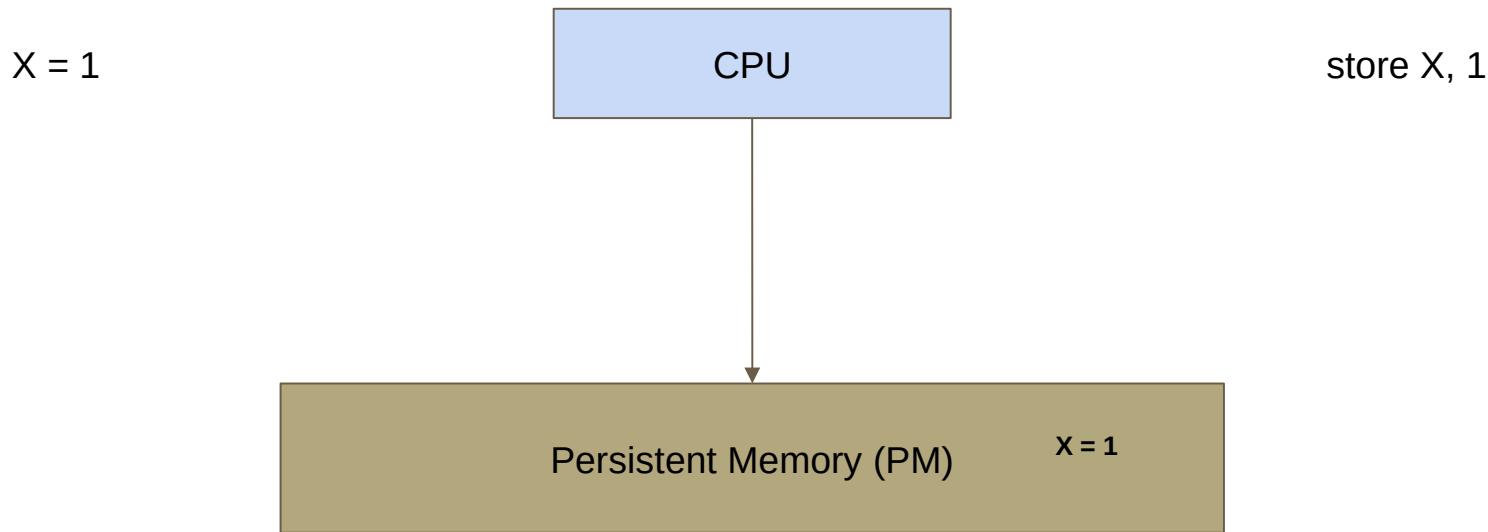
Desafios para programação com PM

Assuma que queremos apenas escrever um valor de forma persistente na PM

X = 1

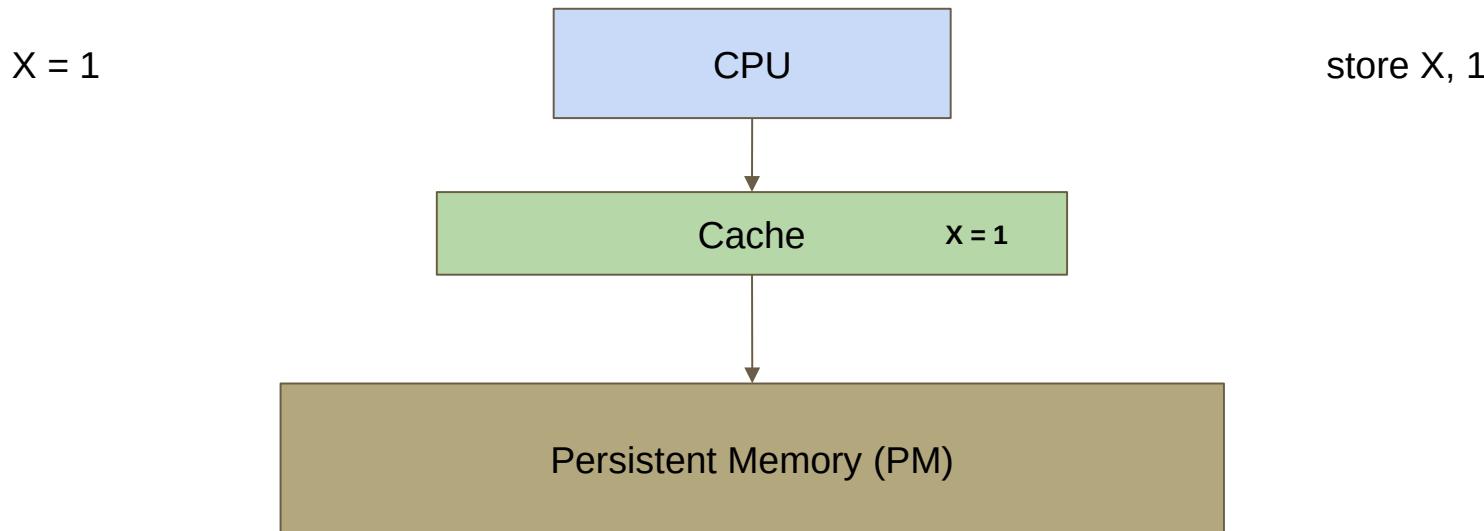
Desafios para programação com PM

Assuma que queremos apenas escrever um valor de forma persistente na PM



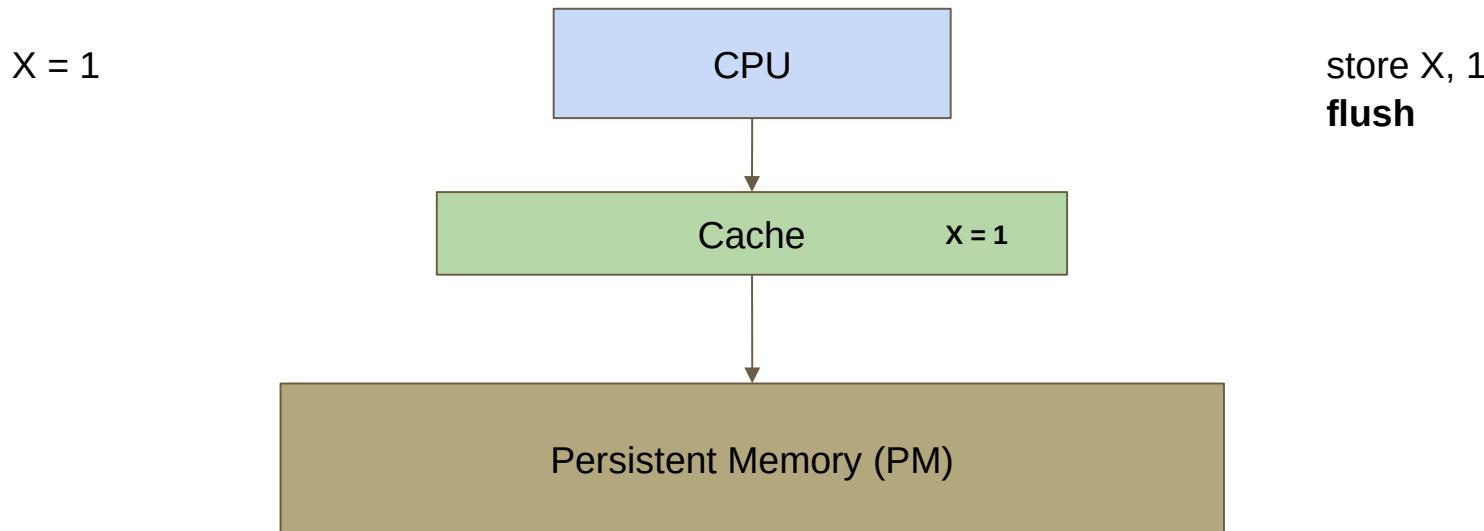
Desafios para programação com PM

Assuma que queremos apenas escrever um valor de forma persistente na PM



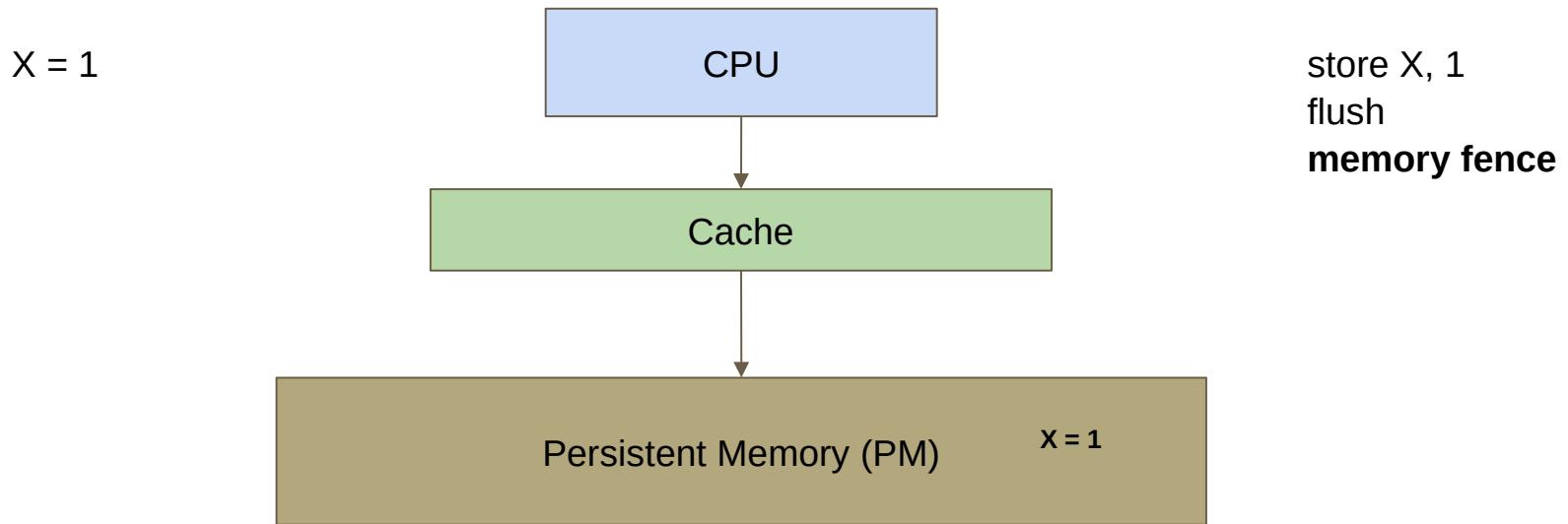
Desafios para programação com PM

Assuma que queremos apenas escrever um valor de forma persistente na PM



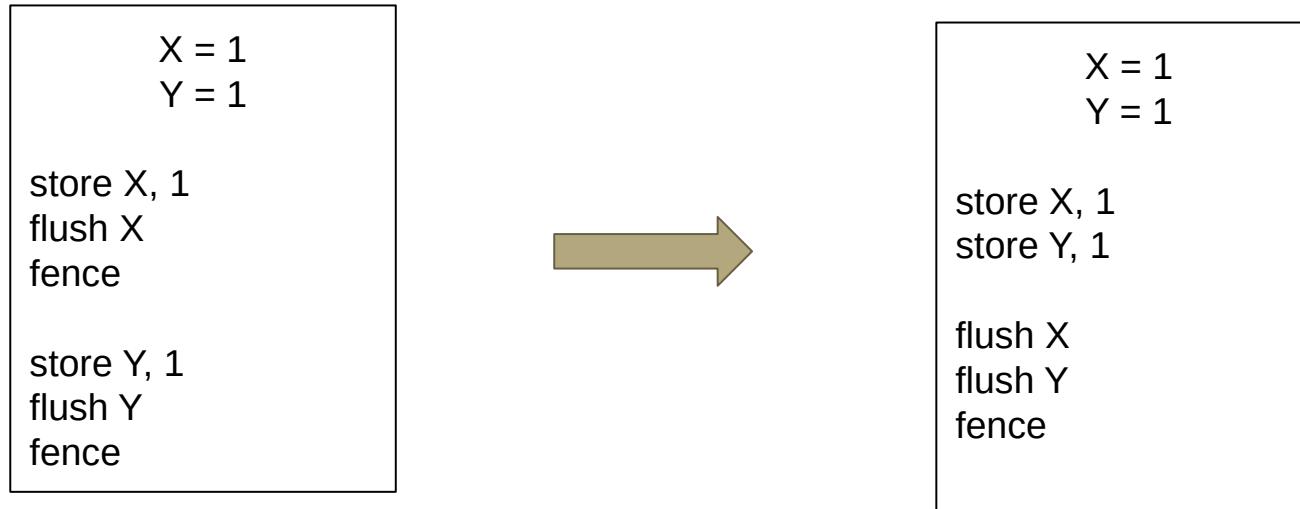
Desafios para programação com PM

Assuma que queremos apenas escrever um valor de forma persistente na PM



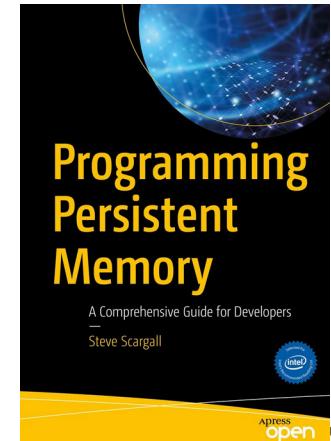
Desafios para programação com PM

- Adicionar instruções extras (e.g., flush e barreira) para cada store torna o programa lento
- Tentativa de otimizar o código pode torná-lo errado!

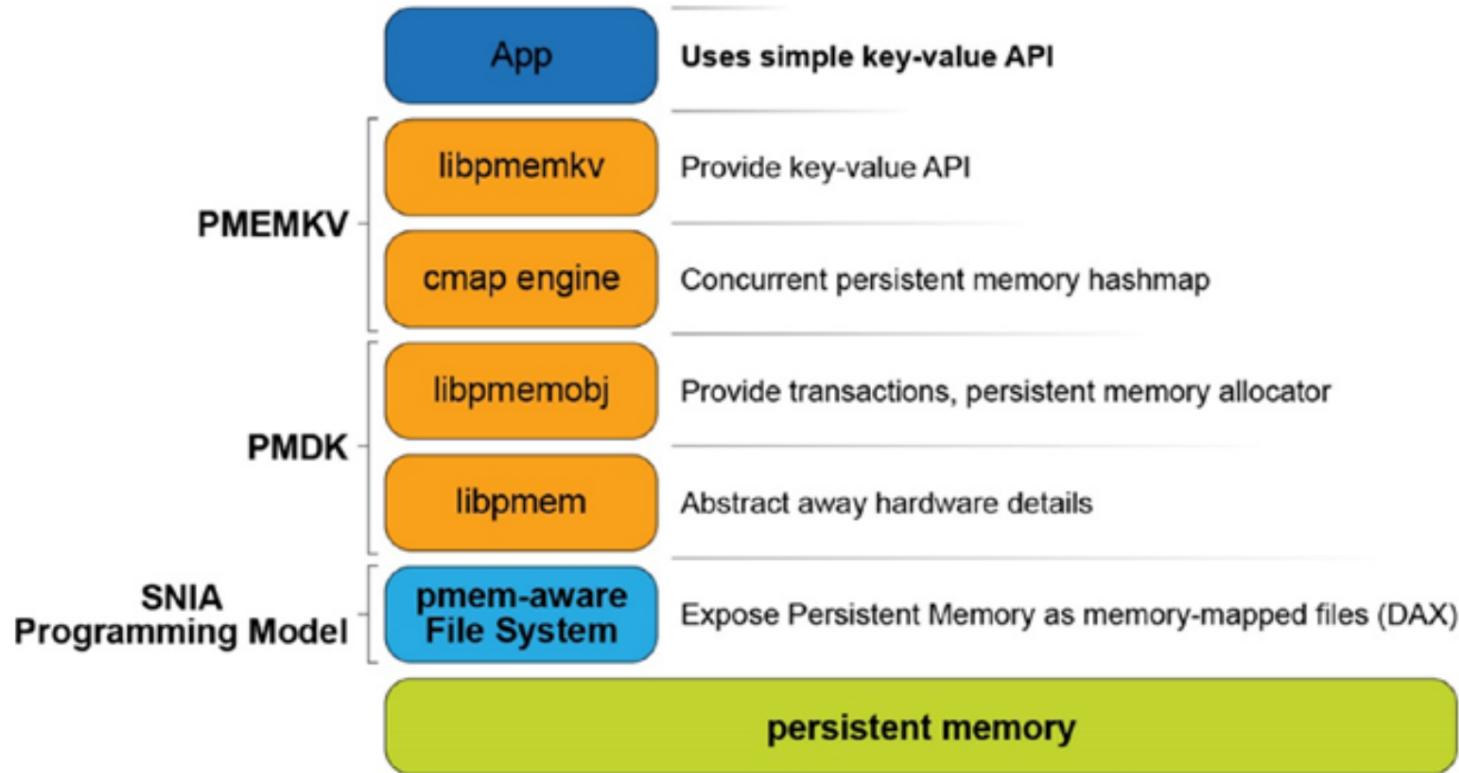


Objetivo e estrutura do mini-curso

- Mostrar como programar para sistemas com PM utilizando bibliotecas disponíveis
 - HP: Atlas (obsoleta) - <https://github.com/HewlettPackard/Atlas>
 - famus: failure-atomic msync() - <https://web.eecs.umich.edu/~tpkelly/famus/>
 - Intel: **PMDK (Persistent Memory Development Kit)** - <https://pmem.io/pmdk/>
- Seguiremos o Intel PMDK (livro disponível gratuitamente)
 - <https://pmem.io/book/>
 - Código fonte do livro:
 - <https://github.com/Apress/programming-persistent-memory>
 - Código PMDK:
 - <https://github.com/pmem/pmdk/>



Software stack com o PMDK



Parte 2 - Arquitetura de Sistemas com Memória Persistente e Suporte do Sistema Operacional

Arquitetura de memória persistente

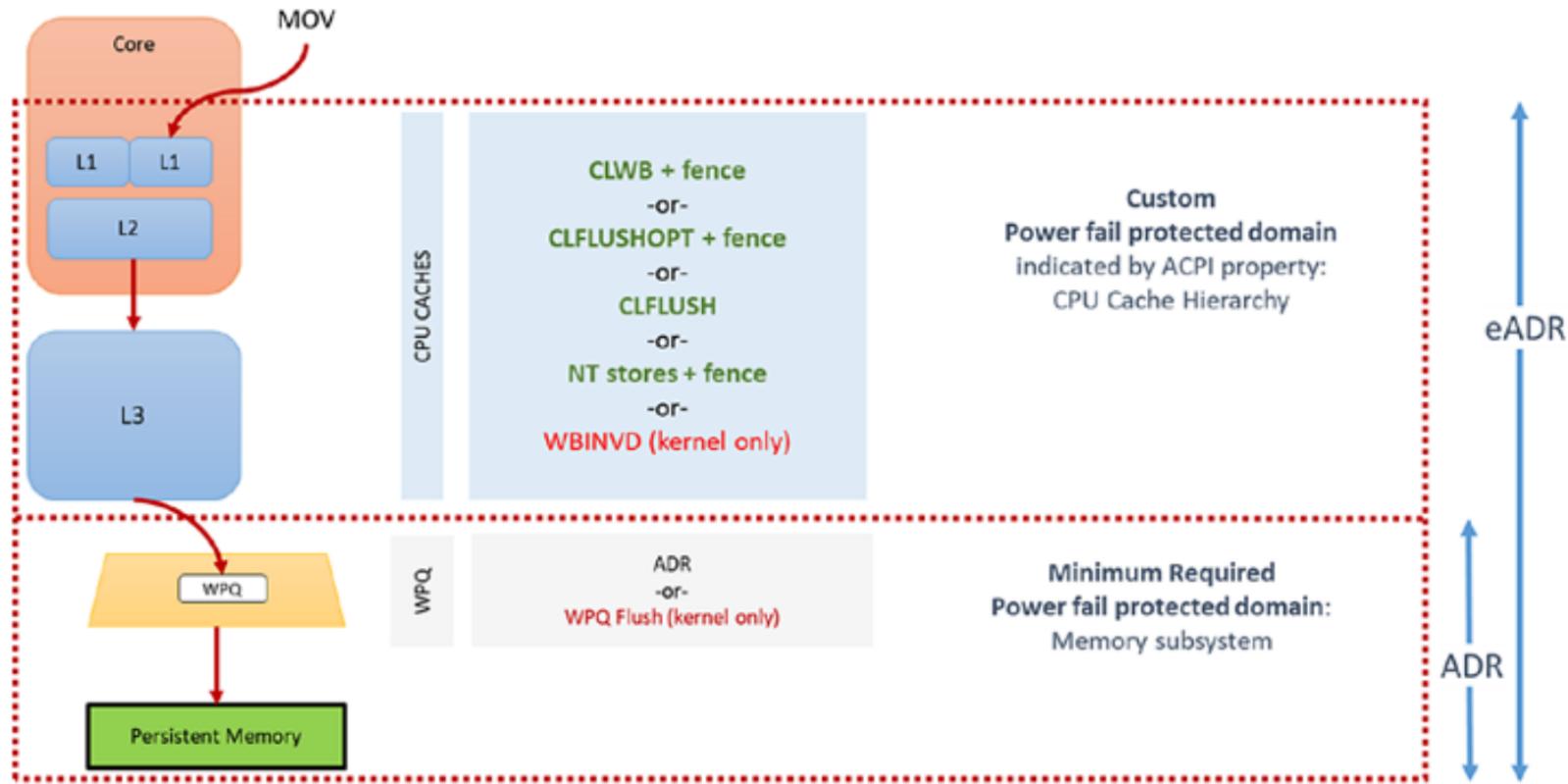
Resumo do que falamos até agora:

- Desempenho > SSDs, HDs, ... < DRAM, SRAM
- Durabilidade > FLASH < DRAM
- Capacidade > DRAM < HD ou fitas
- Atualização de dados in-place
 - Byte addressable
- Pode ser usado com DMA ou RDMA
- Persistente
- Após verificações iniciais, todas os acessos podem ser feitos sem interferência do kernel, sistema de arquivos, etc...

Suporte atual das plataformas

- Modelo de programação tem sido desenvolvido pela SNIA (Advancing Storage and Information Technology)
 - Última versão: 1.2 (novembro de 2017)
 - https://www.snia.org/tech_activities/standards/curr_standards/npm
 - Utilizado pelo PMDK e os dispositivos Optane DC da Intel
- A programação exige o uso de uma biblioteca (recomendável) ou o emprego de barreiras de memória, cache flushing, logging, etc...
 - Logging usado para garantir a atomicidade
 - Cache flushing cuidadoso garante que escritas foram retiradas das caches e enviadas para a memória persistente
 - Barreiras de memória (como SFENCE no x86) evitam o reordenamento de escritas

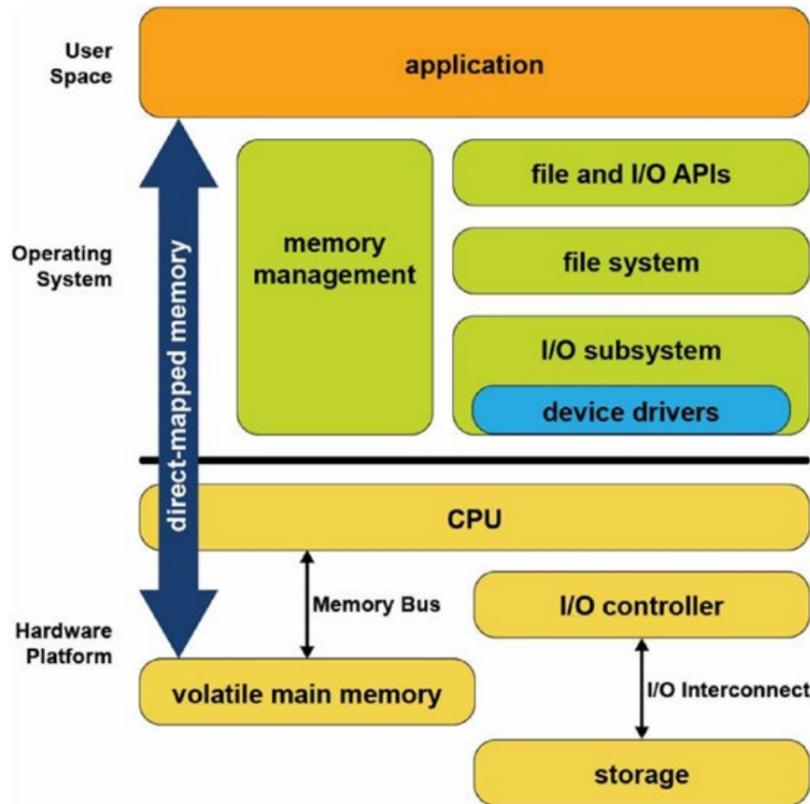
Persistence Domains / Power-Fail Protection Domains



Instruções x86

- **CLFLUSH** - Faz o flush e invalida uma linha de cache (serializado)
- **CLFLUSHOPT** - O mesmo que CLFLUSH mas não serializa (o que permite um certo nível de concorrência). Exige o uso de uma barreira após uma sequência.
- **CLWB** - Comportamento semelhante ao CLFLUSHOPT, contudo não invalida (obrigatoriamente) a linha da cache. Exige uso de uma barreira.
- **NT Stores** - Ignoram as caches e escrevem diretamente na PM. Exigem uso de barreira.
- **SFENCE** - Barreira de escrita. Garante que todas as operações que apareceram antes da barreira (*program order*) têm seus efeitos globalmente visíveis antes de efeitos de qualquer outra instrução de store após a barreira.
- **WBINVD** - (kernel mode) Faz o flush de todas as linhas de todas as caches e as invalida.

Suporte dos Sistemas Operacionais



Problemas com SOs não preparados para PM

- Ainda usam interfaces tradicionais de arquivos ou...
- ...quando mapeados em memória ainda fazem a transferências dos dados da memória persistente para a RAM e da RAM para a memória persistente (page cache)

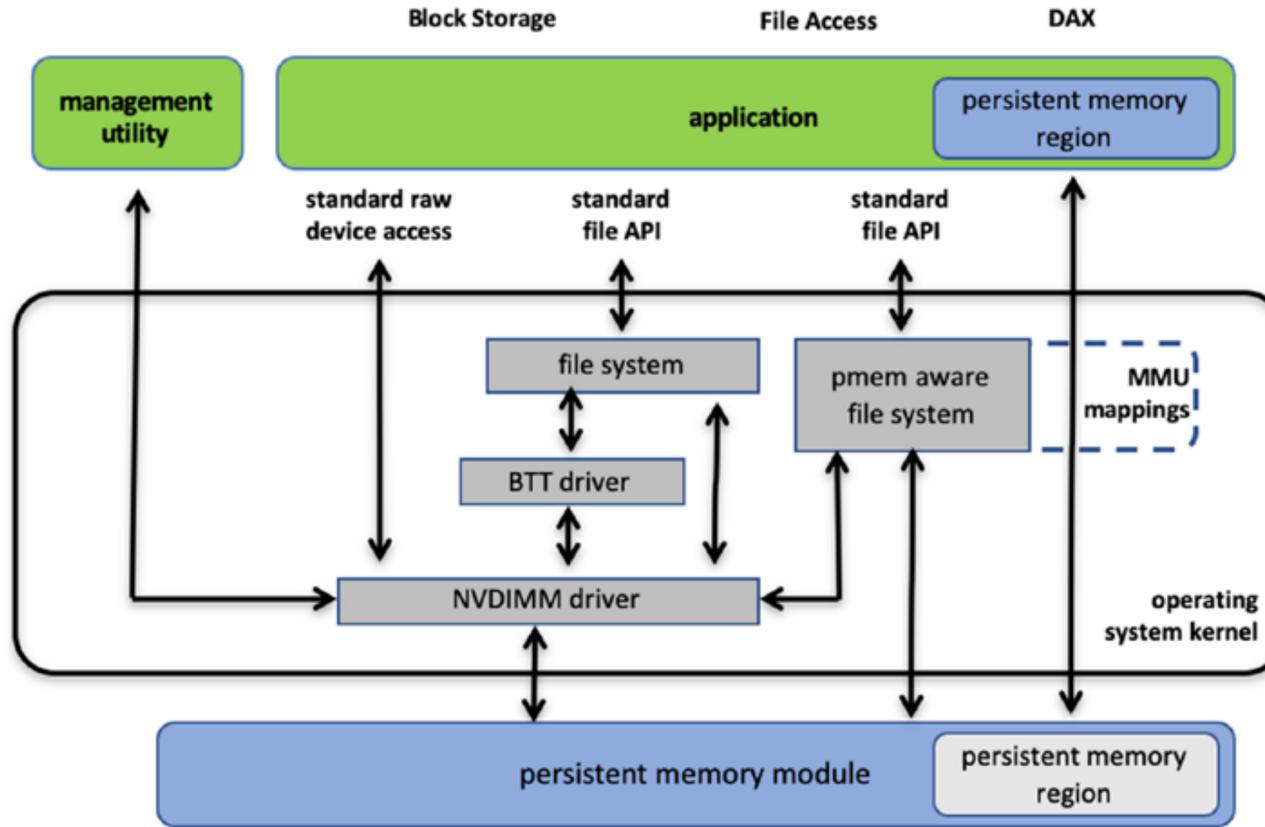
Queremos trabalhar, efetivamente, com:

- Loads e stores
- Acesso direto à PM, sem caching de páginas feitos pelo SO

DAX

- DAX -> Direct Access
 - NTFS no Windows
 - XFS e ext4 no Linux
- Evita o buffering em RAM
- Mapeia as páginas diretamente para o dispositivo de PM
- No Linux, dispositivos aparecem como um device em /dev/pmemXX
- Mapear os dispositivos de PM como arquivos tem várias vantagens:
 - Aproveitar os controles de acesso já presentes no FS
 - Ferramentas de backup funcionam como antes
 - É possível usar todas as ferramentas que já estavam disponíveis (e eventualmente adaptar códigos já existentes) para usar PM de maneira relativamente direta

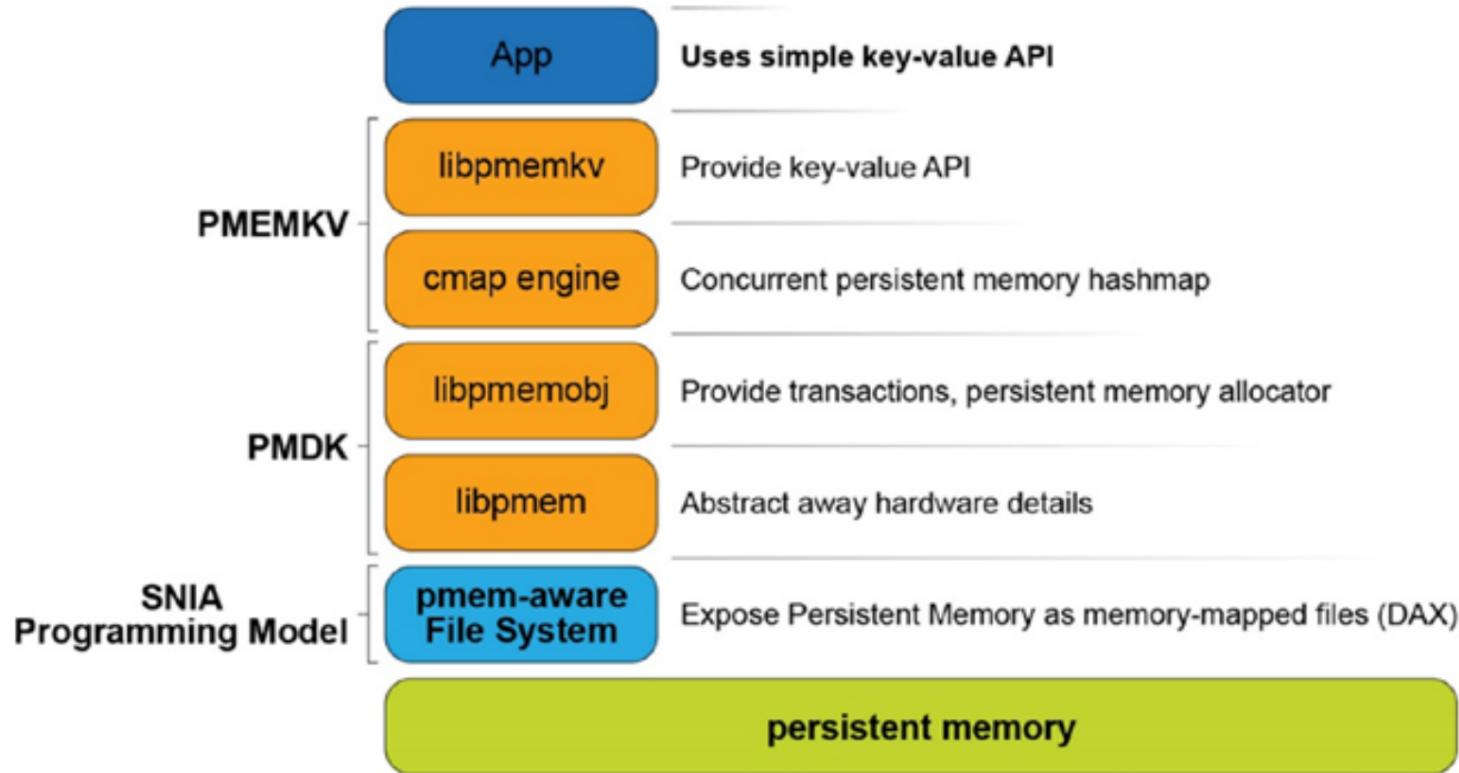
Visão geral



Parte 3 - Libpmemobj

Interface transacional e para alocação dinâmica de memória

Recapitulando: PMDK stack



libpmemobj

- Permite utilizar arquivos mapeados em memória como um *object store*
 - DAX é utilizado para acesso direto
- Principais conceitos
 - *Memory pool*
 - Ponteiros persistentes
 - Objeto raiz
 - Transações
 - Alocação de memória persistente

Memory Pool

- É a abstração utilizada pela `libpmemobj` para trabalhar com memória persistente
 - Usa o esquema de arquivo mapeado + DAX mencionado anteriormente
- Duas formas de criar:
 - Utilitário `pmempool`
Ex: `pmempool create --layout my_layout --size 20M obj exemplo.obj`
 - Programação (API)
`PMEMObjpool *pmemobj_create(const char *path, const char *layout, size_t poolsize, mode_t mode);`

Ponteiros persistentes

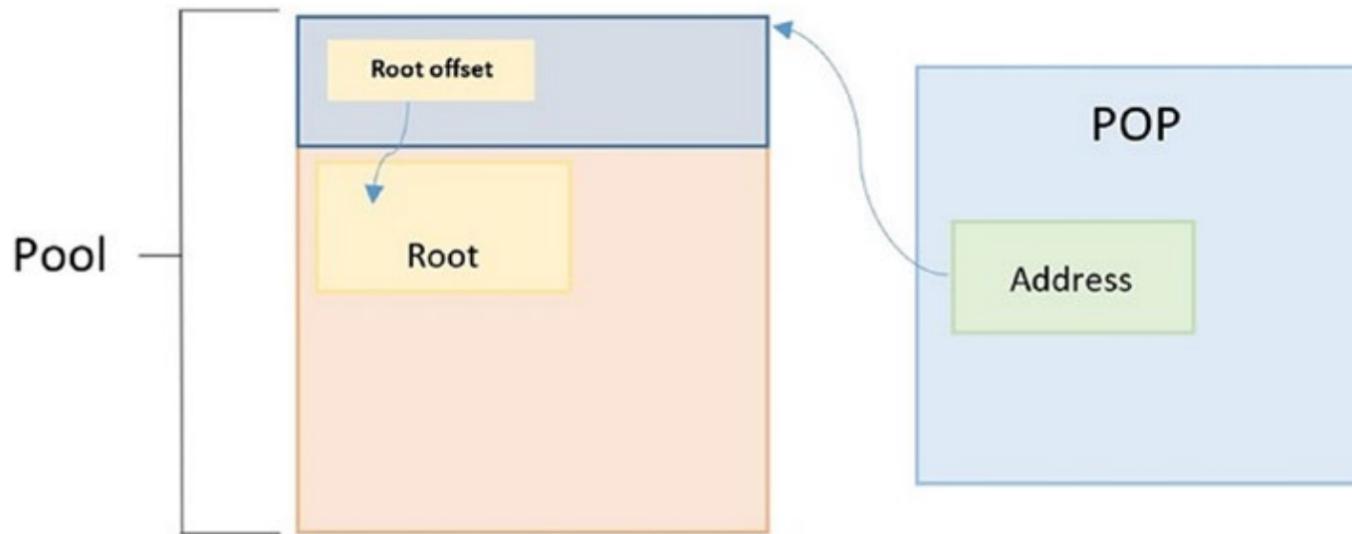
- Uma vez criado, como o memory pool é acessado?
- Ponteiros “convencionais” não são usados
 - Quando aberto, o memory pool pode ser mapeado para endereços virtuais diferentes (ASLR - *Address Space Layout Randomization*)
- PMDK utiliza os chamados *fat pointers*

```
typedef struct pmemoid {  
    uint64_t pool_uuid_lo;  
    uint64_t off;  
} PMEMoid;
```
- Um *fat pointer* precisa ser transformado em um ponteiro regular para que seja dereferenciado
 - Cálculo: `(void *)((uint64_t)pool + oid.off)` // pool é o endereço virtual base
 - Há uma chamada na API do PMDK para fazer a conversão:
`void *pmemobj_direct(PMEMoid oid);`

Objeto raiz

- Vimos até agora
 - Como criar/abrir memory pools
 - O tipo de ponteiro utilizado para representar um objeto no pool
- Mas como encontrar um objeto particular dentro do pool?
 - Todo pool contém um *objeto raiz* através do qual todos os objetos dentro do pool podem ser acessados
- Objeto raiz
 - Funciona como um ponto de entrada para acesso ao memory pool
 - Acessado através da chamada `pmemobj_root`
 - `PMEMoid pmemobj_root(PMEMobjpool *pop, size_t size);`

Pool Object Pointer (POP) e o Objeto raiz



Passos

1. Abrir (ou criar) memory pool (retorna o POP) -
2. Acessar objeto raiz (retorna ponteiro persistente) -
3. Transformar em ponteiro regular antes de dereferenciar -

`pmemobj_open(...)`
`pmemobj_root(...)`
`pmemobj_direct(...)`

Exemplo: implementação de uma lista-ligada persistente

- Ponto de partida: versão volátil

```
struct Node {  
    int data;  
    struct Node *next;  
};  
  
NODE *sll = NULL;
```

Código original obtido de:
<https://gist.github.com/hilsonshrestha/8545379>

- Principais operações

```
void display(NODE *head);  
void insertAtHead(NODE **head, int data);  
void insertAtEnd(NODE **head, int data);  
void deleteByValue(NODE **head, int data);  
int searchByValue(NODE *head, int data);
```

Definições para versão persistente

volátil

```
struct Node {  
    int data;  
    struct Node *next;  
};
```

```
NODE *sll = NULL;
```



persistente

```
struct Node {  
    int data;  
    PMEMoid p_next;  
};
```

```
struct my_root {  
    PMEMoid p_head;  
};
```

```
void display(NODE *head);  
void insertAtHead(NODE **head, int data);  
void insertAtEnd(NODE **head, int data);  
void deleteByValue(NODE **head, int data);  
int searchByValue(NODE *head, int data);
```

```
void display(PMEMoid p_head);  
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data);  
void insertAtEnd(PMEMobjpool *pop, PMEMoid *head, int data);  
void deleteByValue(PMEMobjpool *pop, PMEMoid *head, int data);  
int searchByValue(PMEMoid p_head, int data);
```

Inicialização

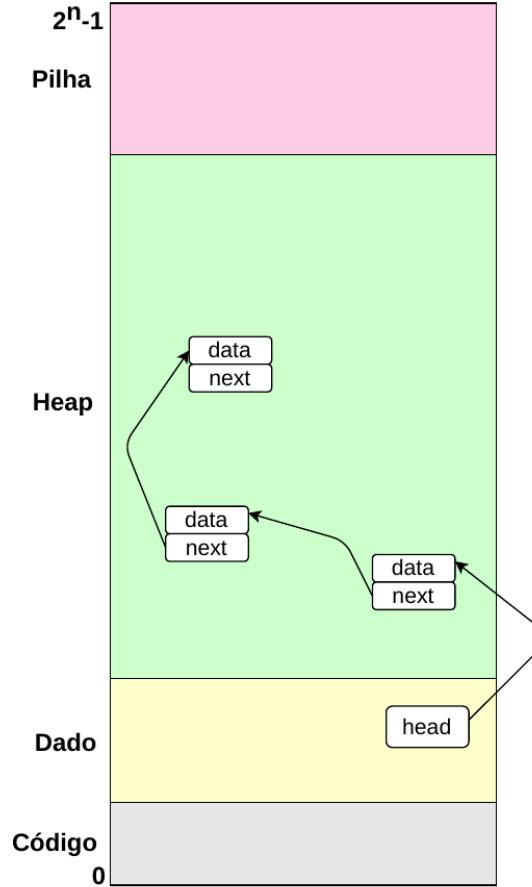
Arquivo criado com **pmempool**

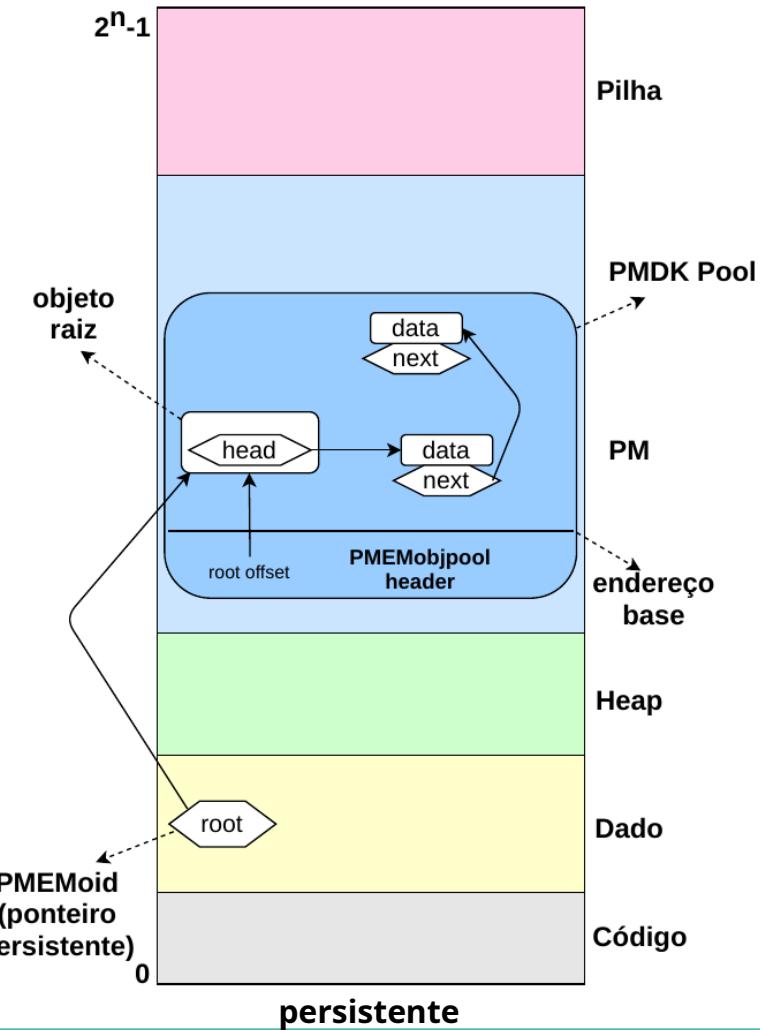
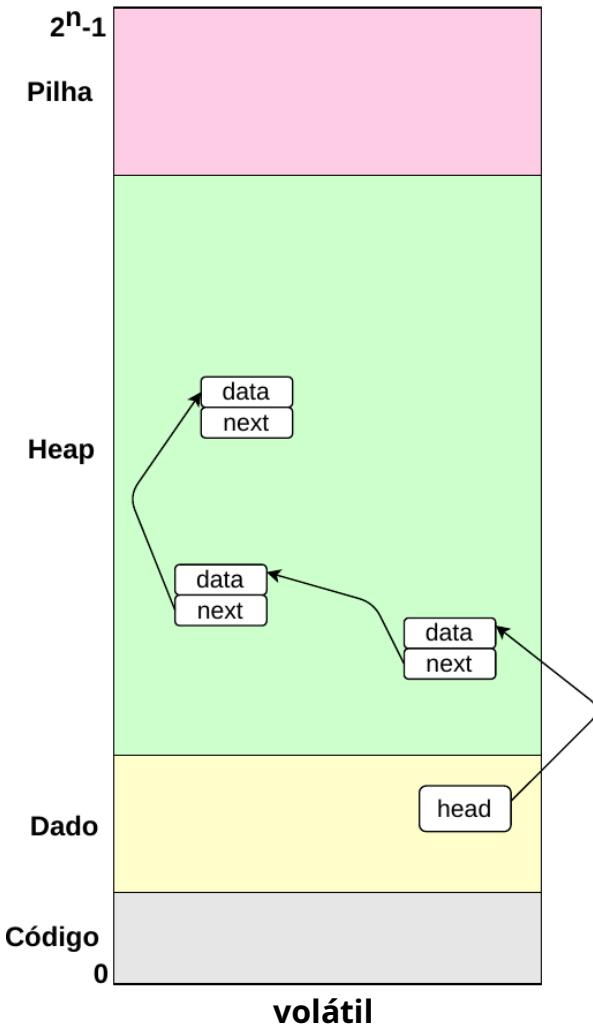
```
/* Open the pool and return a "pool object pointer" */
PMEMobjpool *pop = pmemobj_open(<file>, LAYOUT_NAME);

/* Retrieve a persistent pointer to the root object */
PMEMoid p_root = pmemobj_root(pop, sizeof(struct my_root));

/* Get a "conventional" pointer to the root object */
struct my_root *root = pmemobj_direct(p_root);
```

Layout de memória volátil





Alocação de memória persistente

- Memória é alocada no contexto de algum *memory pool* já aberto
- Problema com alocação
 - Interface convencional para alocação:
 - `x = persistent_malloc(100);`
 - Quando a memória persistente sendo alocada deve ser marcada como alocada?
 - O que acontece se a memória for alocada, mas antes da atribuição acontecer uma falha?
- A alocação de memória persistente precisa de uma interface diferente
 - A modificação do ponteiro destino e a alocação deve ser feita de forma atômica

Métodos para alocação

- Usar métodos atômicos

```
int pmemobj_alloc(PMEMobjpool *pop, PMEMoid *oidp, size_t size, uint64_t type_num,  
pmemobj_constr constructor, void *arg);
```

- Usar transações

```
PMEMoid pmemobj_tx_alloc(size_t size, uint64_t type_num);
```

```
TX_BEGIN(pop) {  
    ...  
    = pmemobj_tx_alloc(...);  
} TX_END
```

Lista ligada: alocação de memória

```
volátil
NODE *createNewNode(int data) {
    NODE *newNode = (NODE *) malloc(sizeof(NODE));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

```
persistente
PMEMoid createNewNode(PMEMobjpool *pop, int data)
{
    PMEMoid p_newNode;

    TX_BEGIN(pop) {
        p_newNode = pmemobj_tx_alloc(sizeof(NODE), 1);
        ((NODE *)pmemobj_direct(p_newNode))->data = data;
        ((NODE *)pmemobj_direct(p_newNode))->p_next = OID_NULL;
    } TX_END

    return p_newNode;
}
```

Lista ligada: inserção na cabeça

volátil

```
void insertAtHead(NODE **head, int data) {
    NODE *newNode = createNewNode(data);
    newNode->next = *head;
    *head = newNode;
}
```

persistente

```
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data)
{
    PMEMoid p_newNode = createNewNode(pop, data);
    ((NODE *)pmemobj_direct(p_newNode))->p_next = *head;
    *head = p_newNode;
}
```

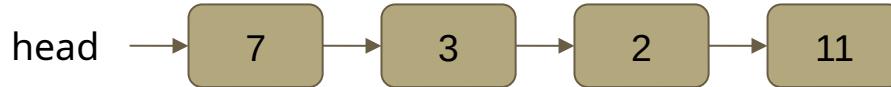
Versão problemática! Qual o erro?

Problemas na inserção

```
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data)
{
    PMEMoid p_newNode = createNewNode(pop, data);

    ((NODE *)pmemobj_direct(p_newNode))->p_next = *head;
    *head = p_newNode;

}
```



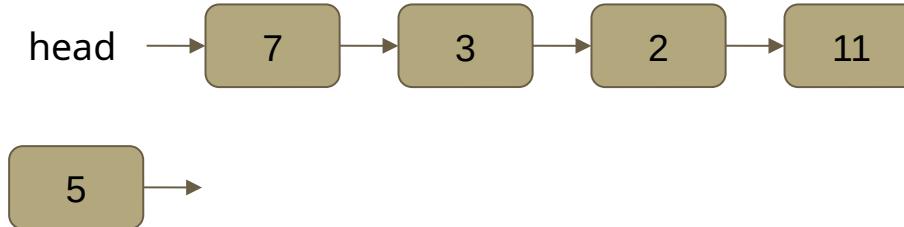
inserir novo elemento “5”

Problemas na inserção

```
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data)
{
    PMEMoid p_newNode = createNewNode(pop, data);

    ((NODE *)pmemobj_direct(p_newNode))->p_next = *head;
    *head = p_newNode;

}
```



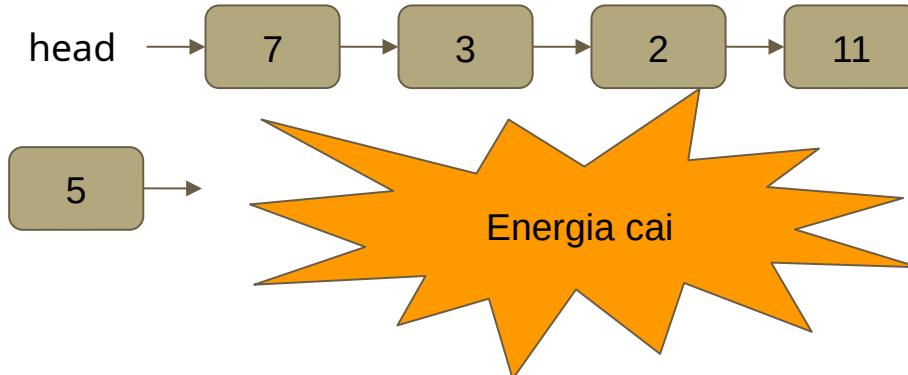
cria nó

Problemas na inserção

```
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data)
{
    PMEMoid p_newNode = createNewNode(pop, data);

    ((NODE *)pmemobj_direct(p_newNode))->p_next = *head;
    *head = p_newNode;

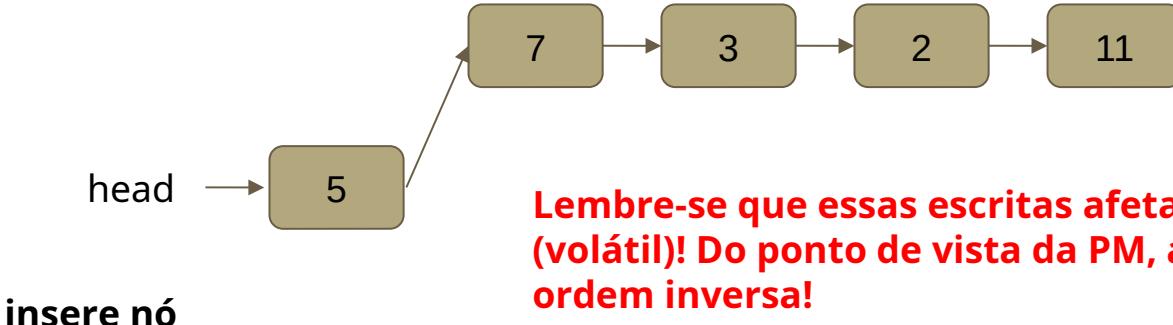
}
```



Vazamento persistente de memória!

Problemas na inserção

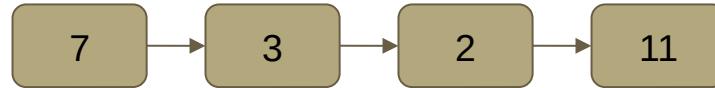
```
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data)
{
    PMEMoid p_newNode = createNewNode(pop, data);
    ((NODE *)pmemobj_direct(p_newNode))->p_next = *head;
    *head = p_newNode;
}
```



Lembre-se que essas escritas afetam primeiramente a cache (volátil)! Do ponto de vista da PM, as escritas podem ocorrer em ordem inversa!

Problemas na inserção

```
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data)
{
    PMEMoid p_newNode = createNewNode(pop, data);
    ((NODE *)pmemobj_direct(p_newNode))->p_next = *head;
    *head = p_newNode;
}
```



head → 5
insere nó

Se a escrita de “head” atingir a PM primeiro e houver uma falha antes do ponteiro “next” ser alterado, teremos um problema de consistência! (a lista foi corrompida)

Lista ligada persistente: inserção atômica

```
void insertAtHead(PMEMobjpool *pop, PMEMoid *head, int data)
{
    TX_BEGIN(pop) {
        PMEMoid p_newNode = createNewNode(pop, data);
        ((NODE *)pmemobj_direct(p_newNode))->p_next = *head;
        pmemobj_tx_add_range_direct(head, sizeof(PMEMoid));
        *head = p_newNode;
    } TX_END
}
```



Versionamento manual das posições que são alteradas pela transação

Uso de macros

- Usar as chamadas diretas fornecidas pelo `libpmemobj` é propenso a erros
 - Não oferece checagem de tipo
- Para remediar o problema, uma série de macros é disponibilizada
- O primeiro passo é declarar o layout das estruturas utilizadas

```
POBJ_LAYOUT_BEGIN(linkedlist);

POBJ_LAYOUT_ROOT(linkedlist, struct my_root);
POBJ_LAYOUT_TOID(linkedlist, struct Node);

POBJ_LAYOUT_END(linkedlist);
```

Uso de macros

- As macros possibilitam checagem de tipo em tempo de compilação

```
/* Retrieve a persistent pointer to the root object */  
PMEMoid p_root = pmemobj_root(pop, sizeof(struct my_root));
```

Para declarar ponteiros persistentes



```
TOID(struct my_root) p_root = POBJ_ROOT(pop, struct my_root);
```

```
/* Get a "conventional" pointer to the root object */  
struct my_root *root = pmemobj_direct(p_root);
```



```
struct my_root *root = D_RW(p_root);
```

Exemplo com as macros: busca

sem
macro

```
int searchByValue(PMEMoid p_head, int data) {
    PMEMoid p_current = p_head;

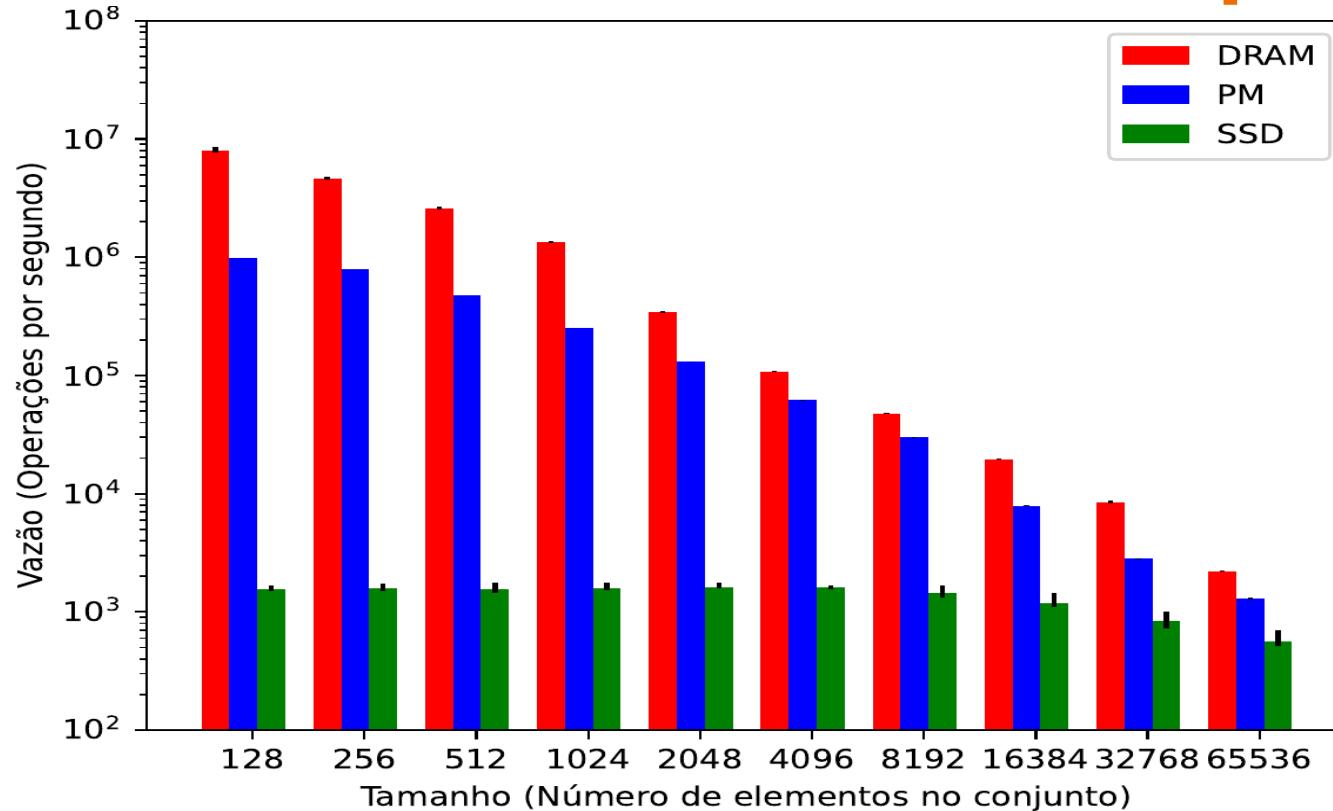
    while (!OID_IS_NULL(p_current)) {
        if (((NODE *)pmemobj_direct(p_current))->data == data)
            return 1;
        p_current = ((NODE *)pmemobj_direct(p_current))->p_next;
    }
    return 0;
}
```

com
macro

```
int searchByValue(TOID(struct Node) head, int data) {
    TOID(struct Node) p_current = head;

    while (!TOID_IS_NULL(p_current)) {
        if (D_RO(p_current)->data == data)
            return 1;
        p_current = D_RO(p_current)->p_next;
    }
    return 0;
}
```

Resultados com uso do Intel Optane



Finalizando libpmemobj

- Há bindings do libpmemobj para C++ e Javascript
- C++, em particular, usa metaprogramação para facilitar a escrita de aplicações persistentes
 - Também é fornecida uma versão persistente da STL
- Links:
 - C++:
 - <https://github.com/pmem/libpmemobj-cpp>
 - Javascript:
 - <https://github.com/pmem/libpmemobj-js>

Outras abordagens

go-pmem

Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, Pratap Subrahmanyam: go-pmem: Native Support for Programming Persistent Memory in Go. USENIX Annual Technical Conference 2020: 859-872

```
// add new node to tail; return new tail
func.addNode(tail *node) *node {
    n := pnew(node)           // <-
    txn("undo") {             // <-
        mutex.Lock()
        n.prev = tail
        updateTail(tail, n)
        mutex.Unlock()
    }                         // <-
    return n
}

func.updateTail(tail, n *node) {
    txn("undo") {             // <-
        tail.next = n
    }                         // <-
}
```

Corundum (Rust)

Morteza Hoseinzadeh, Steven Swanson: Corundum: statically-enforced persistent memory safety. ASPLOS 2021: 429-442

PM.NET (C#)

Trabalho de mestrado em desenvolvimento

Obrigado!!!

Se interessou pelo assunto e quer fazer pesquisa na área?

Nos contacte!

Alexandro Baldassin - alexandro.baldassin@unesp.br

Emilio Francesquini - e.francesquini@ufabc.edu.br

Slides backup