Capítulo 3

Introdução à Programação com Memória Persistente

Alexandro Baldassin
Universidade Estadual Paulista (UNESP)
Rio Claro, Brasil

Emilio Francesquini
Universidade Federal do ABC (UFABC)
Santo André, Brasil

Resumo

Este minicurso é uma introdução à programação com memória persistente (PM). Nele apresentaremos a motivação para uso dessa nova tecnologia; o suporte atual disponibilizado por processadores e sistemas operacionais; os problemas de consistências de dados que podem acontecer; e como utilizar abstrações de mais alto nível (como transações) para resolvê-los. O minicurso também apresenta uma série de exemplos práticos utilizando o Intel PMDK para programação de estruturas de dados persistentes.

3.1. Introdução

As tecnologias para memória persistente endereçada a byte (PM)[1] têm sido exaustivamente pesquisadas nos últimos anos [3]. Como resultado, em 2019, a Intel colocou no mercado o primeiro dispositivo dessa linha, conhecido como *Intel Optane DC* [33].

Quando comparadas à tradicional tecnologia de memória DRAM, essas novas memórias possuem maior densidade, menor custo por bit e um desempenho ligeiramente inferior (principalmente quanto às operações de escrita) [3]. O fato destas novas tecnlogias disponibilizarem acessos na granularidade de byte permite que programadores acessem tal memória diretamente, sem o uso de interfaces convencionais como sistemas de arquivos ou um banco de dados. No entanto, uma falha (como queda de energia) pode acontecer a qualquer momento e tornar os dados inconsistentes. Dessa forma, programar com PM requer cuidados extras quando comparado à forma usual empregada com

[1]Também conhecidas como memórias não voláteis (NVM) ou *Storage Class Memory* (SCM).

memória volátil. O grande desafio atual é como prover abstrações para a programação desses novos dispositivos persistentes [31].

Considerando a recente introdução de PM no mercado e os desafios relacionados à sua programação, argumentamos que esse tema é de extrema relevância e que o suporte atual para sua programação deva ser difundido, uma vez que tais dispositivos tendem a ser utilizados em sistemas de alto desempenho [26]. Estudos recentes mostram uma acentuada quantidade de trabalhos científicos com foco em modelos de programação para PM [2], ressaltando ainda mais a importância do tema. Apesar do minicurso ter um cunho bem prático, também serão mencionados tópicos de pesquisas mais recentes, o que pode abrir oportunidades para os interessados em começar algum estudo sobre o tema, tanto no nível de graduação quanto de pós-graduação.

Atualmente já é possível adquirir máquinas nas quais a PM divide espaço com a DRAM no papel de memória de trabalho. Neste minicurso, vamos explorar o uso de uma máquina com essas características, e nos concentraremos na tecnologia 3D Xpoint da Intel e no Intel PMDK (*Persistent Memory Development Kit*) que são o padrão *de facto* do mercado. O restante do documento está organizado como descrito a seguir. Um detalhamento mais aprofundado sobre os conceitos de persistência são descritos na Seção 3.2. Em seguida, na seção Seção 3.3, descrevemos alguns dos desafios envolvidos com o uso dessa tecnologia. Exemplos práticos do uso do Intel PMDK são mostrados na Seção 3.4, incluindo o desenvolvimento de uma lista ligada simples persistente. O minicurso é encerrado na Seção 3.5.

3.2. Conceitos de Persistência

Nesta seção descrevemos brevemente algumas tecnologias de memória para em seguida falarmos mais especificamente de conceitos de persistência relevantes à utilização de uma máquina com PM.

3.2.1. Tecnologias de memória

Static Random-access Memory (SRAM), *Dynamic Random-access Memory* (DRAM) e Flash são as tecnologias não mecânicas para armazenamento de memória que atualmente desfrutam das mais amplas aplicações no mercado. Essas tecnologias são empregadas desde dispositivos móveis, como telefones celulares, até computadores para processamento científico de alto desempenho. Além de serem amplamente utilizadas, são tecnologias muito maduras que já estão disponíveis há décadas (a tecnologia Flash, a mais recente dentre as três, surgiu há pelo menos 35 anos [20]). Cada uma destas tecnologias conta com certas vantagens e desvantagens dependendo da aplicação. Por exemplo, a tecnologia SRAM é rápida porém possui um gasto energético superior àquele da DRAM; Flash, ao contrário das demais, não é volátil; e DRAM permite maiores densidades do que SRAM (por utilizar apenas um transistor e um capacitor por célula enquanto SRAM utiliza ao menos seis). No entanto, em arquiteturas de hardware reais, usuários procuram por um bom desempenho, por um gasto de energia aceitável e também por um preço competitivo [1]. Portanto, em plataformas de hardware reais, há um compromisso entre as limitações e vantagens de cada tipo de memória aliado às necessidades práticas para a sua fabricação. Tal conjunto de restrições e compromissos motivou os projetistas de

hardware a utilizar soluções híbridas onde cada um destes tipos de memória é empregado para desempenhar papéis nos quais o seu desempenho é o melhor. Esta solução hibrida é claramente evidenciada pelos níveis hierárquicos de memória presentes nos atuais computadores.

Nas modernas arquiteturas de hardware pode ocorrer uma considerável variação no tempo de acesso aos dados dependendo da distância hierárquica entre os núcleos de processamento e a localização dos dados. Por esta razão memórias menores (mais rápidas porém com um sobrecusto energético significativo) são utilizadas para mascarar o tempo de acesso às memórias com maiores capacidades (com menor gasto energético porém maior tempo de acesso). Atualmente, processadores típicos possuem algumas dezenas de registradores baseados em SRAM, alguns níveis de memória cache localizadas no próprio processador (SRAM), a memória principal de trabalho baseada em DRAM e, finalmente, um dispositivo de armazenamento não volátil que normalmente se apresenta na forma de um disco rígido ou *Solid-state Drive* (SSD) baseado em memória Flash. Em algumas plataformas de hardware, *e.g. Non-Uniform Memory Access* (NUMA), pode existir ainda um nível hierárquico adicional entre os núcleos de processamento e algumas regiões da memória.

Não apenas os usuários destes sistemas computacionais estão interessados em desempenhos cada vez mais expressivos (que se traduzem, em parte, por tempos reduzidos de acesso à memória), como também estão interessados em memórias com capacidades cada vez maiores. Esta é uma tendência que não dá sinais de mudança a curto prazo [19]. No entanto, as tecnologias de hardware atuais para a memória principal baseadas em DRAM, por mais maduras que estejam, estão alcançando os seus limites em termos de escalabilidade [14, 15] por fatores intrínsecos à maneira pela qual operam. Os problemas enfrentados pelos engenheiros de hardware envolvem desde a criação de células de memória cada vez menores, até taxas de *refresh* que atualmente são impraticáveis.

A necessidade por maiores capacidades e desempenho dos mais modernos sistemas computacionais levaram os arquitetos de hardware a buscar alternativas para substituir as já consagradas tecnologias de armazenamento de memória. Entre as principais novas tecnologias alternativas competidoras estão versões mais modernas da memória Flash [5, 23, 32], *Ferroelectric RAM* (FeRAM) [4], *Spin-Torque Transfer RAM* (STT-RAM) [9, 24], *Phase-Change Memory* (PCM) [18, 27] e, mais recentemente, Intel 3D XPoint [11][2] lançada sob o nome comercial Intel Optane DC. Diferentemente da tecnologia DRAM, essas tecnologias alternativas têm como característica a não volatilidade dos dados, ou seja, não dependem de um fluxo contínuo de alimentação elétrica para a manutenção dos dados.

Estas tecnologias são genericamente classificadas como *Storage Class Memory* (SCM), *Non-Volatile Memory* (NVM), ou simplesmente *Persistent Memory* (PM). PMs, principalmente aquelas baseadas em Flash, têm sido utilizadas para a fabricação de dispositivos de armazenamento secundários, um território (ainda) dominado por dispositivos de armazenamento mecânicos como discos rígidos. A principal vantagem vinculada ao

[2]A tecnologia 3D XPoint é, de fato, uma tecnologia PCM. Contudo ela tem diversas características como ter várias camadas empilhadas [16] (daí o nome 3D) que justificam a sua classificação em uma categoria diferente das PCMs tradicionais.

Características	DRAM	XPoint	FeRAM	STT-RAM	PCM	NOR Flash	NAND Flash
Tamanho da célula (F^2)	6–8	4	15–32	36	4–12	10	4
Tamanho do bloco/latência	N.A.	N.A.	N.A.	N.A.	N.A.	64–256KB 900ms	8–64KB 2ms
Tamanho da página	64b	N.A.	64b	64b	64b	64b	512b–4KB
Latência de leitura	10’s ns	10’s ns	10’s ns	10’s ns	10’s ns	100’s ns	10’s μs
Latência de escrita	10’s ns	10’s μs	10’s ns	10’s ns	10’s μs	10’s μs	100’s μs
Não-volatilidade	não	sim	sim	sim	sim	sim	sim
Durabilidade (Nº. escritas)	10^{16}	10^8	10^{14}	10^{15}	10^8–10^9	10^5	10^5

Tabela 3.1: Comparação entre DRAM [22, 13, 14], XPoint [11, 16], FeRAM [6, 7, 17], STT-RAM [9, 24, 35], PCM [18, 27] e Flash [5, 23, 32].

uso de PMs é uma capacidade potencialmente muito maior do que as que são possiveis com o uso de DRAM (apesar de possuírem velocidades de acesso inferiores) e velocidades muito maiores de acesso do que dispositívos de bloco (apesar de ter uma capacidade de armazenamento inferior). Em outras palavras, PMs se encaixam como uma tecnologia intermediária entre os atuais dispositivos de bloco e a as memórias de trabalho que são tipicamente representadas pela tecnologia DRAM.

Algumas dessas novas tecnologias, como por exemplo PCM, já possuem desempenhos comparáveis à DRAM. A Tabela 3.1 apresenta um resumo das características de algumas dessas tecnologias de memória. Apesar de Flash estar entre as tecnologias de memória com o maior amadurecimento atualmente, ela tem uma durabilidade limitada. No entanto, algumas projeções [14, 15] já indicam que as mais novas tecnologias alternativas de memória superarão o desempenho e escalabilidade de tecnologias tradicionais como Flash em alguns anos, como já é, inclusive, o caso da 3D XPoint.

O ritmo de desenvolvimento destas novas tecnologias indica que o desempenho dos dispositivos de armazenamento baseados nessas novas tecnologias cada vez mais deve se aproximar do desempenho das memórias de trabalho, hoje baseadas quase que exclusivamente[3] em DRAM. Quando levamos em consideração a natureza hierárquica dos computadores atuais e o desempenho das mais novas tecnologias de memória, nos damos conta que um passo natural após a substituição da tecnologia dos dispositivos de armazenamento secundários por memórias persistentes é a utilização de tecnologias de memória com maior escalabilidade no próximo nível hierárquico: a memória de trabalho. Com exceção da memória Flash[4], novas tecnologias de memória persistente já podem ser consideradas como possíveis substitutas à tecnologia tradicional.

[3]Em alguns dispositivos embarcados o uso de memória persistente como memória de trabalho já é uma realidade.

[4]Sua baixa durabilidade a torna inadequada para uma memória ativa de trabalho.

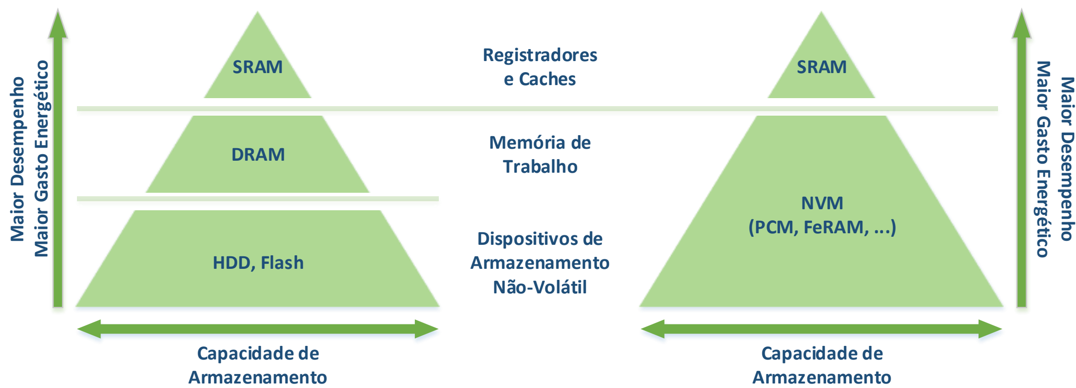


Figura 3.1: Diagrama simplificado dos níveis hierárquicos de memória. Nesta figura são comparados os níveis hierárquicos de memória e suas funções entre uma arquitetura tradicional (esquerda) com uma arquitetura baseada em memória universal (direita). Em uma arquitetura com memória universal, a memória de trabalho é a mesma utilizada para o armazenamento permanente de dados.

3.2.2. O papel das PMs na hierarquia de memória

Pesquisas sobre os estágios de adoção de PM têm sido feitas considerando tanto substituições parciais [27, 36, 29, 28] quanto totais [18, 35, 37] da DRAM por PMs. Uma arquitetura com *memória universal* é, simplificadamente, uma arquitetura na qual os níveis de memória de armazenamento secundário e de trabalho foram condensados em apenas um [1]. A Figura 3.1 ilustra simplificadamente as diferentes funções de cada um dos níveis hierárquicos de memória comparando uma arquitetura tradicional a uma arquitetura com memória universal.

Ainda assim, é muito provável que leve alguns anos até que uma máquina comercial com estas características chegue ao mercado [21]. Em 2014 a Hewlett-Packard anunciou [10, 34, 8] ter dado início um a um projeto de pesquisa para a construção de um protótipo com tal arquitetura (apesar de, até hoje, não existir um produto amplamente disponível no mercado com essas características).

As atuais limitações de desempenho, durabilidade, disponibilidade e custo das PMs quando comparadas à DRAM (uma tecnologia madura e já produzida em larga escala) fazem com que a adoção dessas novas tecnologias seja gradual, primeiramente com PMs lado a lado com DRAM, em seguida substituindo-a completamente para finalmente estar presente em todos os níveis hierárquicos de memória.

A Figura 3.2 mostra algumas das possíveis combinações. No caso *(a)*, vemos o uso tradicional de memória DRAM como memória de trabalho exclusiva, sem a presença de PMs e com dispositivos de armazenamento secundários tradicionais. Em *(b)*, contudo, vemos a presença de PM em total substituição à memória volátil. Este cenário é o mais próximo de uma máquina com arquitetura universal de memória. No entanto, tal máquina ainda possui as caches e os registradores do processador utilizando tecnologias de memória voláteis como, por exemplo, SRAM. Este cenário, onde há a substituição completa de DRAM por PM, deve ainda levar alguns anos para ocorrer de maneira ampla

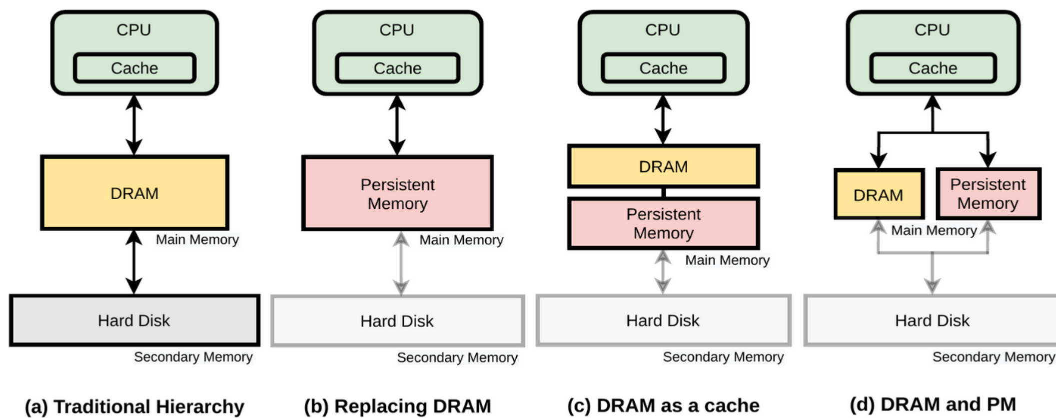


Figura 3.2: Possíveis papeis da PM na hierarquia da memória principal. (a) Arquitetura tradicional. (b) Como uma substituta para DRAM (c) Utilizando DRAM como uma cache (d) DRAM e PM lado a lado. Adaptado de [30].

simplesmente porque a tecnologia de DRAM ainda possui um melhor desempenho do que as PMs. O caso *(c)* tenta combinar o desempenho superior de DRAM com a capacidade superior das PMs, assim servindo como uma cache para acelerar os acessos. De fato, neste cenário pode-se pensar na DRAM como um nível adicional de cache (externo) ao processador. Finalmente, no caso *(d)*, a PM trabalha lado a lado com a DRAM. Logo, a escolha de qual memória deve ser utilizada (e quando) fica a cargo do software (do sistema operacional e das aplicações). Embora as máquinas atuais equipadas com Optane DC possam trabalhar nas configurações *(c)* e *(d)*, as pesquisas tendem a se concentrar na opção *(d)*, já que essa configuração expõe a PM diretamente ao programador.

3.3. Desafios na Programação

A Figura 3.3 apresenta os modos de acesso a dispositivos não-voláteis. O primeiro modo de uso (a) emprega uma API de arquivos, ou seja, o dispositivo é exposto para as aplicações como se fosse um arquivo (tipicamente localizado em `/dev/pmem` em sistemas Linux). A segunda interface de programação *(b)* é através de uma API de consultas, por exemplo fornecida por um banco de dados, que enxerga a dispositivo persistente como um arquivo armazenado em um dispositivo de armazenamento em blocos. Finalmente, a terceira interface *(c)* enxerga a PM como uma área de memória, a ser utilizada diretamente como se fosse memória de trabalho convencional, fazendo acessos de leitura e escrita diretamente da aplicação.

Os modos de acesso *(a)* e *(b)*, embora tradicionais, foram construídos para dispositivos acessados em blocos. Eles requerem duas representações dos dados para uma estrutura de dados: uma para a versão em DRAM e outra serializada para ser armazenada no dispositivo de armazenamento em bloco. Por esta razão é necessário converter entre um formato e outro durante o carregamento (de-serialização) e salvamento (serialização). A utilização desses dois modos não possui um mapeamento direto e óbvio com o uso da DRAM pelas aplicações. O modo *(c)*, que é o foco desde minicurso, acessa a PM diretamente, como se fosse a memória DRAM que estamos acostumados. Esta maneira

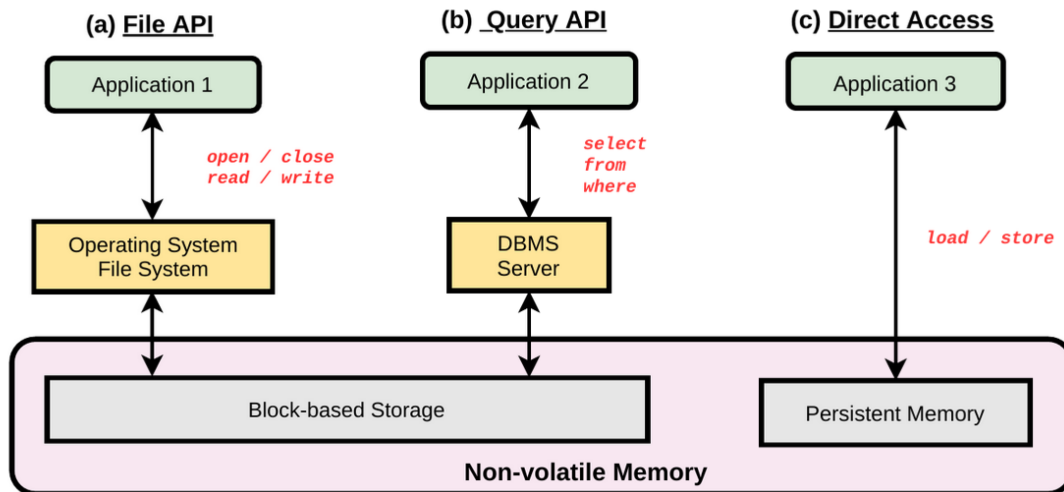


Figura 3.3: Principais modos de acesso à PM. (a) através de uma API de arquivos. (b) através de uma API fornecida, por exemplo, por um banco de dados que acessa a PM como um dispositivo de armazenamento de blocos. (c) Através de acesso direto, usando operações de load/store diretamente da aplicação. Adaptado de [30].

de acesso evita gargalos impostos pela pilha de software presente nos modos *(a)* e *(b)* (sistema de arquivos, sistema gerenciador do banco de dados, paginação do sistema operacional, etc). Enquanto os overheads associados com este tipo de acesso, em geral, são desprezíveis quando estamos lidando com dispositivos lentos como um disco rígido ou com um SSD, eles passam a ser um gargalo importante de desempenho quando utilizamos uma PM cujo desempenho supera muito (em termos de latência e banda) aqueles de SSDs e HDs típicos.

O uso de PM como uma memória de trabalho traz consigo desafios, alguns novos e outros bem conhecidos, para os desenvolvedores. O primeiro e mais óbvio é que torna-se preciso assegurar que acessos concorrentes aos dados sejam sincronizados e, assim, sejam evitadas condições de corrida. Outros, contudo, são desafios que aparecem devido às interações da PM com os demais níveis de memória. Nós os exploramos na próxima seção.

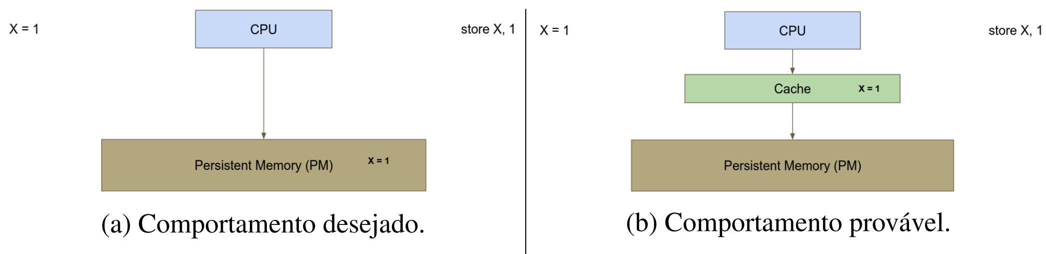
3.3.1. Interações com as *caches*

Suponha que tenhamos um programa simples, que apenas quer armazenar um valor em uma posição de memória. Um possível código que efetua esta tarefa é mostrado abaixo (à esquerda) e o que será efetivamente executado pelo processador será algo semelhante ao que está à direita:

```
x = 1;
```

```
STORE X, 1
```

Após a execução deste código, desejamos que como resultado o valor `1` seja armazenado na PM, conforme a Figura 3.5a. Porém, na verdade, o que provavelmente ocorrerá é o cenário mostrado na Figura 3.5b.



(a) Comportamento desejado.

(b) Comportamento provável.

Figura 3.5: Interações da cache com a PM. Uma operação de escrita não altera imediatamente a PM, mas sim a cache do processador.

A situação mostrada na Figura 3.5b pode ser perigosa pois, apesar da visão dos processos em execução que têm acesso à essa posição de memória ser consistente e todos verem o valor `1` (garantido pelo modelo de memória do processador/protocolos de coerência de cache), uma possível falha no fornecimento de energia neste ponto da execução implicaria a perda do valor armazenado já que as caches são voláteis.

De fato existem instruções específicas que ignoram a cache e que permitem fazer *loads* e *stores* diretamente na memória de trabalho. Em arquiteturas Intel tais instruções são chamadas de *Non-temporal*. Essas instruções, contudo, trazem consigo uma penalidade considerável de desempenho além de forçarem o programador a trocar um programa relativamente simples, como uma atribuição a uma variável, pela chamada de uma função especial do compilador para a geração das instruções relevantes, incluindo barreiras de memória para tornar os resultados visíveis para todos os processadores.

Uma possível correção para o problema acima usando apenas barreiras seria a seguinte:

```
x = 1;
```

```
STORE X, 1
FLUSH
```

A instrução do tipo *flush* força o dado a sair da cache e ir para o próximo nível de memória. Há diferentes versões de flushes, como veremos adiante, mas é possível que o flush não envie o dado imediatamente para o próximo nível. Podemos imaginar que o flush age como se fosse uma chamada não bloqueante. O que significa que mesmo após executar o flush, em caso de falhas, pode ocorrer de alguns dados serem perdidos. Assim, para garantir que tudo está corretamente salvo, precisamos de uma barreira de memória (*fence*) após o flush.

Outro cuidado a ser tomado é durante otimizações do código que podem inadvertidamente introduzir bugs. Por exemplo, o uso de flushes e barreiras a cada operação de store pode deixar o programa lento. Logo é razoável supor que sua consolidação seja feita como uma possível otimização:

```
// x = 1;
// y = 1;

store X, 1
flush X
fence

store Y, 1
flush Y
fence
```

(a) Original

```
// x = 1;
// y = 1;

store X, 1
store Y, 1

flush X
flush Y
fence
```

(b) Otimizado

Contudo o código otimizado acima pode não ser válido caso exista uma necessidade (definida pela aplicação em questão) de que o valor de `X` seja armazenado antes do valor de `Y`. Em outras palavras, o código acima pode não obedecer a ordem dos stores em caso de falhas e é possível que apenas o valor de `Y` seja escrito, e portanto salvo, na PM.

A lista abaixo apresenta algumas instruções de flush e acesso não-temporal à memória disponíveis na ISA x86. Note que nem todas as instruções podem estar disponíveis em todos os processadores e que o comportamento de algumas pode variar dependendo da versão do processador.

- `CLFLUSH` - Faz o flush e invalida uma linha de cache. Comportamento 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 de instruções deste tipo.
- `CLWB` - Comportamento semelhante ao CLFLUSHOPT, contudo não invalida (obrigatoriamente) a linha da cache. Em alguns modelos de processadores a linha é invalidada e em outros não. Exige uso de uma barreira.
- `Non-Temporal 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.

3.3.2. Domínios de persistência

Para tentar especificar (um pouco) melhor o modelo de memória oferecido pelas diferentes máquinas com suporte a PM e facilitar o trabalho dos desenvolvedores, a Intel divide as máquinas com este suporte em duas categorias chamadas domínios de persistência (*persistence domains*) ou domínios de proteção contra falhas de alimentação (*power-fail protection domain*). Esses domínios são mostrados na Figura 3.8.

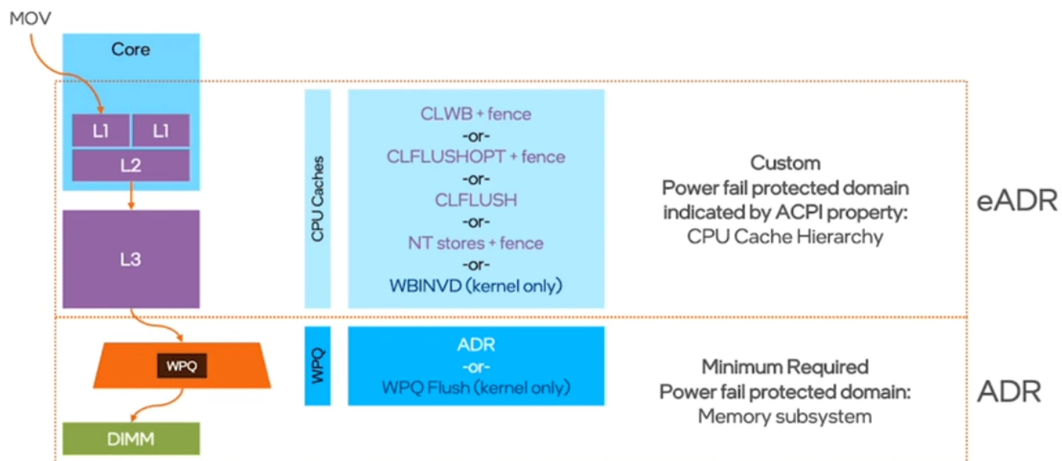


Figura 3.8: Domínios de persistência. Com ADR, os dados são persistentes assim que atingirem a WPQ. A eADR garante que os dados se tornem persistentes assim que atingirem as caches. Adaptado de [12].

O primeiro domínio de persistência chamado de *Asynchronous DRAM Refresh* (ADR) garante que dados que já chegaram até a PM, ou ainda, que estejam no controlador de memória (representado pela fila de escritas pendentes, *Write Pending Queue* - WPQ, na Figura 3.8) serão persistidos, ainda que haja uma falha. Para garantir que esses dados estejam em uma dessas localizações podemos utilizar uma combinação de flushes e barreiras de memória. As combinações corretas de quais instruções de flush e de barreira devem ser utilizadas dependem da máquina em questão.

O segundo domínio de persistência, chamado *Enhanced Asynchronous DRAM Refresh* (eADR), oferece todas as garantias oferecidas pelo domínio ADR, além de oferecer garantias que dados presentes nas caches dos processadores também serão escritos para a PM. Neste caso apenas os dados que ainda estejam nos registradores do processador seriam perdidos em caso de falhas. Essa garantia exige, contudo, uma fonte de energia auxiliar que seja capaz de manter o processador funcionando por alguns instantes para que as linhas sujas das caches possam ser enviadas para o controlador de memória e, em seguida, para a PM. Apesar deste domínio de persistência já ter suporte oficial de alguns processadores e placas-mães [12], ainda não está claro como eADR funcionaria já que não existem, até onde pudemos averiguar, soluções disponíveis no mercado para fornecer a energia que seria necessária para essa operação.

Atualmente há, de fato, apenas suporte de processadores da Intel para o uso de PM. A biblioteca PMDK, apoiada pela Intel, foi criada como uma solução para que os desenvolvedores que desejam utilizar PM não tenham que, repetidamente, lidar com problemas semelhantes aos descritos acima além de não precisarem estar, necessariamente, cientes das idiossincrasias de cada processador. Em particular, determinar que tipos de operações de memória (barreiras, flushing, logging, ...) estão disponíveis em cada processador é algo que foge do domínio da maior parte das aplicações que empregam PM. Some-se a isso o fato de que alguns comportamentos de instruções não são exatamente especificados pela documentação e que variam de acordo com a versão de processadores

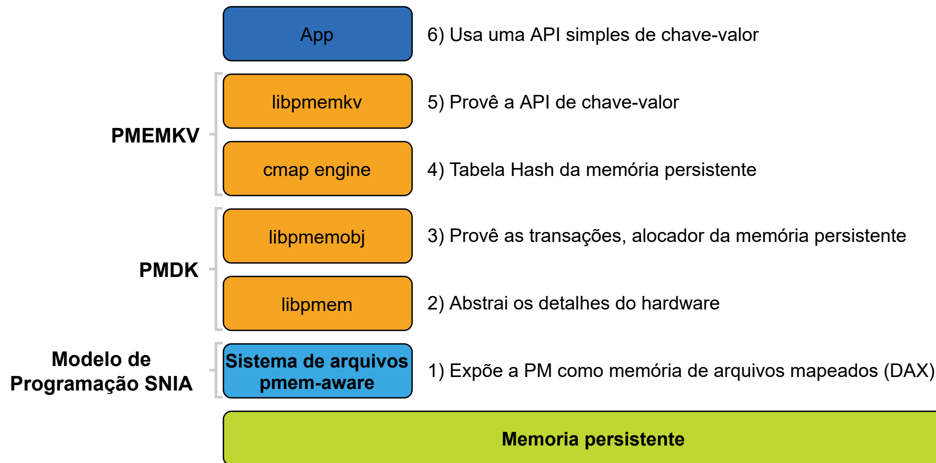


Figura 3.9: Pilha de software para programação com PM fornecida pela Intel. Este Adaptado de [30].

com a mesma ISA! Por estas razões recomendamos o uso do PMDK (ou de uma biblioteca semelhante). Neste curso concentraremo-nos no uso do PMDK apenas.

3.4. Programando com o Intel PMDK

Como apresentado nas seções anteriores, a programação para PM diverge de várias maneiras do modelo tradicional com DRAM. Esta seção apresenta de forma prática os principais pontos necessários para garantir a persistência dos dados usando o Intel PMDK (*Persistent Memory Development Kit*) [30]. Como a tecnologia de PM é relativamente recente, a pilha de software ainda é incipiente e há uma grande variedade de modelos sendo propostos na Academia (veja [2]). O Intel PMDK tem sido desenvolvido há uma década e a Intel optou por abrir o código fonte do projeto [5]. O projeto é atualizado com bastante frequência e é notável o interesse da Intel em prover essa infraestrutura de software dado que lançou recentemente no mercado dispositivos com PM (Intel Optane DC).

A Figura 3.9 mostra as camadas típicas fornecidas pela Intel para a programação com PM, separadas em vários níveis de abstração. Como o propósito desta seção é apresentar uma visão prática dos conceitos já discutidos, optou-se por focar na camada conhecida como `libpmemobj` (camada 3). Essa é a camada mais básica que provê os recursos essenciais para programação de estruturas de dados persistentes, em particular *transações*. O `libpmemobj` é uma biblioteca para linguagem C mas atualmente há alguns *bindings* para outras linguagens, como C++ e Java. Os exemplos utilizados neste texto usam a sintaxe original para linguagem C.

Nesta parte serão discutidos os conceitos chaves empregados pelo PMDK para a programação com PM. Estes podem ser divididos em: i) acesso à PM; ii) atualização dos dados persistentes; e iii) gerenciamento de memória. As próximas seções apresentam com um maior detalhe esses conceitos, seguido de um exemplo prático com a implementação

[5]Veja https://pmem.io/

de um estrutura de lista ligada persistente.

3.4.1. Acesso à PM

A PM é acessada por meio de um *pool de memória*, representado no PMDK pelo tipo `PMEMobjpool`. O pool de memória é uma abstração para a área de memória persistente que será gerenciada. Do ponto de vista do sistema operacional, o pool de memória é tratado como um arquivo convencional e pode ser criado pela ferramenta `pmemtool` disponibilizada pelo PMDK (recomendado) ou programaticamente. Uma vez criado, programadores devem explicitamente abrir o pool através de uma chamada específica fornecida pelo PMDK antes de trabalhar com seu conteúdo. Essa chamada retornará um identificador, comumente chamado de *pool object pointer* (pop), utilizado para realizar diversas operações sobre o pool. Enquanto que na programação tradicional com memória volátil a memória é alocada a partir do *heap*, com PM os dados devem ser alocados no pool. Diferentemente do heap, os dados na PM são acessados a partir de um *objeto raiz*. Todo objeto armazenado em um pool persistente deve ser acessado a partir do objeto raiz. Todo pool tem exatamente um objeto raiz e ele sempre existe.

Uma vez o pool aberto, o objeto raiz pode ser recuperado através de uma chamada oferecida pelo PMDK, tipicamente a `pmemobj_root()`. Essa chamada retorna um ponteiro para o objeto raiz. É importante ressaltar que esse ponteiro aponta para uma área de memória persistente e portanto requer alguns cuidados especiais. Quando um pool é aberto, ele é mapeado para o espaço de endereçamento do processo. Geralmente não é possível prever em qual faixa desse espaço o pool será mapeado. Ou seja, usar um ponteiro convencional não é suficiente para endereçar um objeto persistente porque os endereços podem mudar de uma execução para outra. Usa-se o termo *ponteiro persistente* para diferenciar os ponteiros para PM dos ponteiros convencionais para memória volátil. No PMDK, um ponteiro persistente é representado pelo tipo `PMEMoid` de 128 bits, sendo composto por um identificador único de pool, de 64 bits, e um deslocamento, também de 64 bits. Para encontrar o endereço de um objeto no pool persistente é necessário somar o endereço virtual base, no qual o pool foi mapeado, com o deslocamento. Isso geralmente é feito por meio da chamada `pmemobj_direct()`.

Como um resumo, eis os passos necessários para acessar a PM: 1) criar o pool (se ainda não existir); 2) abrir o pool; 3) recuperar o objeto raiz; 4) efetuar operações na PM a partir deste objeto.

3.4.2. Atualização dos dados persistentes

O conceito principal utilizado pelo PMDK para proporcionar alteração do estado persistente é o de *transação*. Uma transação garante que todas as alterações realizadas dentro dela ou são efetivadas (*committed*) e consideradas persistentes, ou então abortadas (*aborted*) e revertidas (*rollback*) – nesse caso é como se nada tivesse acontecido. Essa propriedade é geralmente conhecida como *tudo ou nada* (*all or nothing*) e é a principal razão de se usar transações no contexto de PM. Desta forma, se uma queda de energia ou falha do sistema acontece enquanto uma transação está sendo executada, todas as alterações realizadas até aquele ponto serão revertidas. As transações fornecidas pelo PMDK podem ser aninhadas, mas é importante ressaltar que não garantem atomicidade entre threads,

ou seja, as modificações realizadas por uma transação de uma determinada thread são visíveis para todas as outras. Portanto é necessário a utilização de um mecanismo de sincronização específico no caso de aplicações multithreading. Por simplicidade, este texto não discute o suporte da PMDK para programação multithreading.

Toda operação efetuada sobre a PM que possa causar inconsistências deve ser colocada dentro de uma transação. No PMDK, uma transação pode ser especificada através do conjunto de macros `TX_BEGIN` e `TX_END`. Ao começar a transação é necessário especificar o pool de memória sobre o qual ela atuará. A maneira como as transações do PMDK garantem atomicidade a falhas é através do *versionamento de dados*, similar ao que acontece em sistemas de Banco de Dados. A técnica mais comum é conhecida como *Write-Ahead Logging*, ou simplesmente WAL [25]. Por exemplo, até recentemente o PMDK utilizava o *undo-logging* [6], no qual uma cópia do dado é salvo em um log interno antes do valor original ser alterado na PM. Apesar do programador não precisar conhecer exatamente como o esquema de logging é implementado, ele precisa informar para o PMDK, por meio de chamadas específicas da biblioteca (e.g., `TX_ADD`), quais os blocos de memória persistente serão alterados. Discutimos com mais detalhes como isso é feito ao implementar uma estrutura de dados persistente na Seção 3.4.4.

3.4.3. Alocação de memória

A interface para alocação de memória persistente também se difere da interface convencional para memória volátil. Para entender o porquê, considere que exista um alocador de memória persistente e que ela pudesse ser alocada através da seguinte chamada:

```
persistent_ptr = persistent_malloc(100);
```

Agora pense nos problemas que podem acontecer durante a execução desse trecho de código. Em particular, o que acontece se a memória é reservada internamente pelo alocador mas uma falha qualquer (e.g., queda de energia) acontece antes da atribuição para o ponteiro persistente? Qual é o estado do sistema depois que a falha for tratada (e.g., o sistema for religado)? Do ponto de vista do alocador, a alocação aconteceu. Porém, do ponto de vista da aplicação, o ponteiro ainda é inválido. O resultado da falha é um vazamento persistente de memória!

A forma mais comum e recomendada para resolver essa inconsistência é realizar a alocação de memória persistente dentro de uma transação. Pelo fato da chamada estar sendo realizada dentro de uma transação, uma falha antes da atribuição faz com que todo o trecho de código (inclusive a memória persistente alocada) seja revertido. Ou seja, o vazamento de memória não pode mais acontecer. O PMDK fornece ainda outras duas formas para alocação de memória persistente: alocação atômica e a interface *reserve/-publish*. Pelo fato da usabilidade dessas interfaces ser restrita a casos específicos, este texto se concentra no uso de alocação através de transações. O leitor pode encontrar mais detalhes sobre esses métodos alternativos no livro da Intel [30].

[6] Outra forma comum é o *redo-logging*, no qual as alterações são salvas em um log interno e aplicadas durante a fase de efetivação da transação.

```
1 struct Node {
2   int data;
3   struct Node *next;
4 };
5
6 typedef struct Node NODE;
7
8
```

(a) Convencional

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

struct my_root {
  PMEMoid p_head;
};
```

(b) PMDK

Figura 3.10: Definição da estrutura para lista ligada simples. A versão com o PMDK (direita) utiliza ponteiros persistentes (`PMEMoid`) e declara explicitamente o objeto raiz.

3.4.4. Uma lista ligada persistente

Esta seção discute como os principais conceitos do PMDK descritos anteriormente podem ser usados para implementar uma estrutura de dados do tipo lista ligada persistente. Para fins de comparação, uma versão convencional (memória volátil) dessa estrutura de dados também é usada[7]. A Figura 3.10a mostra a forma convencional, enquanto a Figura 3.10b apresenta uma versão utilizando o PMDK. Há duas diferenças principais. Primeiro, note que os ponteiros precisam usar o tipo `PMEMoid`, já que agora a estrutura residirá na PM. Em segundo lugar, é comum declarar uma estrutura para representar o objeto raiz, no caso do exemplo a estrutura `my_root` (linhas 6–8). Neste exemplo, o objeto raiz é apenas um ponteiro persistente para o primeiro elemento da lista.

Há diversos procedimentos que operam sobre a lista ligada, entre eles: inserção, remoção, busca e impressão. Para começar a operar com a estrutura, a versão convencional provavelmente vai declarar um ponteiro (volátil) que será passado para os diferentes procedimentos. A versão persistente necessita primeiramente abrir o pool e recuperar o objeto raiz. A Figura 3.11 mostra um pool já existente (`exemplo.pool`) sendo aberto (linha 2)[8]. O pool é simplesmente um arquivo localizado na PM. Para que a aplicação possa confirmar se o conteúdo do pool é o esperado, geralmente um *layout* (basicamente, uma string) pode ser associado ao pool. Desta forma, no momento da abertura do pool, a aplicação pode exigir que o pool sendo aberto tenha um layout específico. No exemplo, o layout `pm-linkedlist` é usado. Assim, se o pool possuir um outro layout, a rotina vai falhar. Quando o pool persistente é aberto, um ponteiro para o objeto do tipo `PMEMobjpool` é retornado. Este ponteiro deve ser usado em outras chamadas do PMDK que lidam com o pool.

Uma vez o pool aberto, é possível recuperar o ponteiro para o objeto raiz (linha 5). Lembre-se que todo pool tem um único objeto raiz e ele é sempre válido. Caso seja a primeira vez que o objeto esteja sendo recuperado, a chamada `pmemobj_root` cria um novo objeto (vazio, por padrão), cujo tamanho é dado pelo segundo parâmetro. Para acessar o conteúdo do objeto raiz (no caso, um ponteiro para a lista ligada persistente) é

[7]O código fonte deste exemplo está disponível em `https://github.com/baldas/minicurso-memoria-persistente`

[8]O código para checagem de erro é omitido para simplificar a exposição do tema.

```
1  /* Abre o pool persistente e retorna um identificador */
2  PMEMobjpool *pop = pmemobj_open("exemplo.pool", "pm-linkedlist");
3
4  /* Recupera um ponteiro persistente para o objeto raiz */
5  PMEMoid p_root = pmemobj_root(pop, sizeof(struct my_root));
6
7  /* Transforma em um ponteiro convencional para acesso ao objeto raiz */
8  struct my_root *root = pmemobj_direct(p_root);
9
10 /* Imprime o conteudo da lista */
11 display(root->p_head);
```

Figura 3.11: Trecho de código necessário na versão persistente: abertura do pool de memória e acesso ao objeto raiz.

necessário primeiro transformar o ponteiro persistente (`PMEMoid`) para um ponteiro convencional. Isto pode ser feito adicionando ao endereço base do pool persistente (endereço no qual o pool for mapeado no espaço de endereçamento do processo) o deslocamento do objeto raiz. A chamada `pmemobj_direct` desempenha essa função (linha 8). Em geral, essa transformação é necessária toda vez que o conteúdo de um objeto apontado por um ponteiro persistente (PMEMoid) precisar ser acessado. Uma dica para evitar a confusão entre ponteiros convencionais e persistentes é sempre nomear as variáveis do tipo `PMEMoid` com o prefixo "`p_`". Finalmente, um exemplo com a chamada que imprime o conteúdo da lista ligada persistente pode ser visto na linha 11.

Para compreender melhor a diferença da versão volátil da lista ligada para a persistente, considere a Figura 3.12. O lado esquerdo mostra o espaço de endereçamento (endereços de memória aumentam de baixo para cima) de um processo que usa apenas memória volátil. A alocação dinâmica usa memória do heap (área verde). No exemplo, o ponteiro para o começo da lista é armazenado na seção de dados. Agora compare com a versão persistente (lado direito). Neste caso, a memória persistente também foi mapeada para o espaço de endereçamento do processo (área azul). É possível, portanto, acessar tanto a memória DRAM quanto PM. Os ponteiros persistentes (`PMEMoid`) estão representados graficamente por hexágonos. A figura assume que um pool persistente (azul mais escuro) já foi aberto. O ponteiro `root` (alocado em DRAM) contém o endereço de 128 bits (`PMEMoid`) para o objeto raiz. É importante que os ponteiros persistentes sempre apontem para algo dentro do pool.

Um fator complicador para quem está migrando do modelo convencional para o persistente é trabalhar de forma consistente com os ponteiros persistentes. Considere a implementação do procedimento para impressão da lista apresentado na Figura 3.13. A Figura 3.13a mostra uma implementação típica para a versão volátil da estrutura de dados. Comparando com a versão persistente, da Figura 3.13b, é possível notar que o ponteiro convencional foi substituído pelo persistente (`PMEMoid`). Qualquer operação não trivial sobre esse ponteiro requer algum tipo de chamada para o PMDK. Por exemplo, para checar se o ponteiro é nulo, geralmente a macro `OID_IS_NULL` é usada (linha 3). Como já dito, sempre que o conteúdo do objeto apontado por um ponteiro persistente deve ser acessado, primeiro é necessário convertê-lo para a versão convencional através

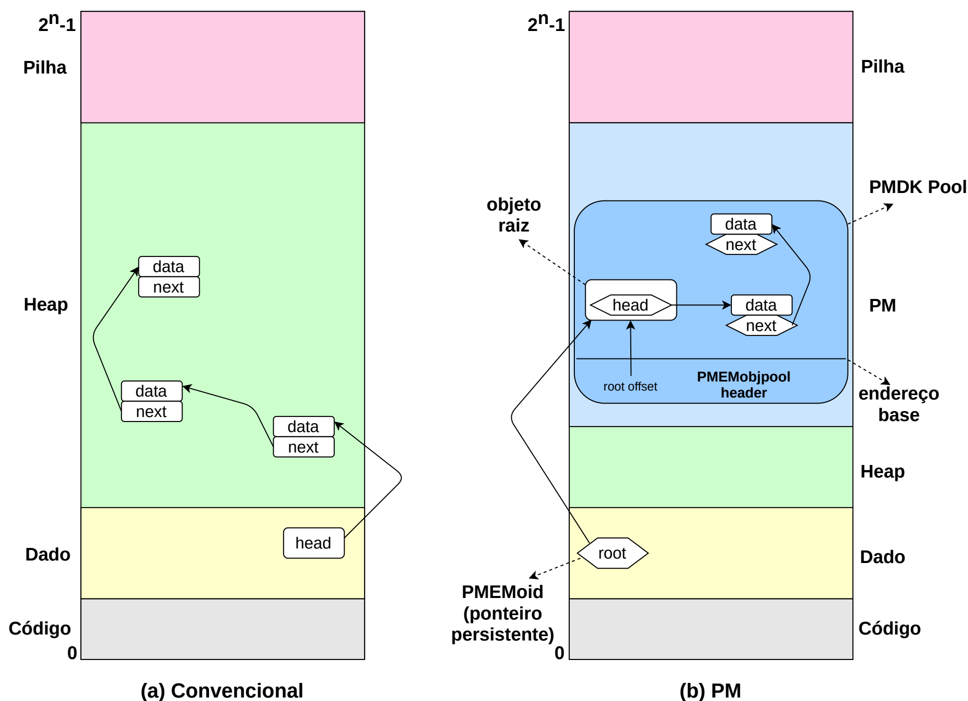


Figura 3.12: Uma visão do espaço de endereçamento com a lista ligada volátil (esquerda) e persistente (direita). A PM é mapeada no espaço de endereçamento do processo e pode ser acessada assim que um pool (`PMEMobjpool`) é aberto.

da chamada `pmemobj_direct` (linhas 4 e 5).

As operações realizadas diretamente sobre os objetos `PMEMoid` podem ser confusas e suscetíveis a erros. Por exemplo, a chamada `pmemobj_direct` retorna ponteiros sem tipo (void *). Portanto, é necessário realizar uma coerção para o tipo adequado. Os desenvolvedores do PMDK logo notaram que isso poderia ser um problema e implementaram uma abordagem baseada em macros para facilitar o manuseio dos ponteiros persistentes. Esta estratégia provê checagem estática de tipo, ou seja, o compilador pode acusar erros se um ponteiro para um objeto de um tipo for atribuído para um ponteiro de um tipo diferente.

Uma explicação completa sobre o suporte para checagem de tipos do PMDK está fora do escopo deste texto. No entanto, será apresentado o básico para a implementação da lista ligada persistente. A primeira etapa consiste da declaração do layout do pool, como mostrado na Figura 3.14. O layout é declarado nas linhas 1 a 4. A primeira linha marca o início da declaração e, a quarta linha, seu final. É necessário informar um nome para o layout (no caso, `linkedlist`). A declaração é composta do tipo para o objeto raiz (linha 2) e de todas as estruturas auxiliares; no caso do exemplo, apenas o tipo do nodo da lista ligada (linha 3). As linhas 6–13 mostram a declaração das estruturas, similar àquelas já apresentadas na Figura 3.10b. Porém, note que agora o tipo `PMEMoid` é substituído pela macro `TOID`. O parâmetro da macro é o tipo do ponteiro persistente.

A Figura 3.15 mostra como o procedimento para impressão da lista persistente

```
1 void display(NODE *head) {
2   NODE *current = head;
3   while (current != NULL) {
4     printf("␣%d␣", current->data);
5     current = current->next;
6   }
7 }
```

(a) Convencional

```
1 void display(PMEMoid p_head) {
2   PMEMoid p_current = p_head;
3   while (!OID_IS_NULL(p_current)) {
4     printf("␣%d␣", ((NODE *)pmemobj_direct(p_current))->data);
5     p_current = ((NODE *)pmemobj_direct(p_current))->p_next;
6   }
7 }
```

(b) PMDK

Figura 3.13: Implementação do procedimento para impressão da lista ligada.

pode ser codificada usando as macros. Compare essa versão com a apresentada anteriormente na Figura 3.13b. Note que, novamente, o tipo `PMEMoid` é substituído pela macro `TOID` (linhas 1 e 2). Além disso, ao invés de usar a chamada `pmemobj_direct`, é possível usar a macro `D_RO` para acessar o ponteiro direto para o objeto persistente (linhas 4 e 5). Quando o conteúdo do objeto é somente lido, em geral a macro `D_RO` é usada. Para os casos onde o conteúdo é escrito, necessita-se usar a macro `D_RW`.

O procedimento para a criação de um novo nodo para a lista ligada persistente é apresentado na Figura 3.16. Como discutido previamente, a forma mais comum de alocar memória do pool persistente é através de uma transação. A transação é iniciada na linha 5 e termina na linha 10. O parâmetro `pop` passado para a transação é o identificador do pool aberto. Um novo objeto é alocado através de `TX_NEW`, na linha 6. As linhas 8

```
1  POBJ_LAYOUT_BEGIN(linkedlist);
2  POBJ_LAYOUT_ROOT(linkedlist, struct my_root);
3  POBJ_LAYOUT_TOID(linkedlist, struct Node);
4  POBJ_LAYOUT_END(linkedlist);
5
6  struct Node {
7    int data;
8    TOID(struct Node) p_next;
9  };
10
11 struct my_root {
12   TOID(struct Node) p_head;
13 };
```

Figura 3.14: Lista ligada usando o suporte para declaração de layout do PMDK.

```
1 void display(TOID(struct Node) head) {
2   TOID(struct Node) p_current = head;
3   while (!TOID_IS_NULL(p_current)) {
4     printf("␣%d␣", D_RO(p_current)->data);
5     p_current = D_RO(p_current)->p_next;
6   }
7 }
```

Figura 3.15: Procedimento para impressão usando o suporte de macros do PMDK.

e 9 inicializam os campos do novo nodo (dado e ponteiro). Note que aqui é necessário utilizar a macro D_RW ao invés da D_RO, já que os campos estão sendo alterados. Se a transação for abortada por alguma falha em qualquer ponto antes de ser finalizada, todas as operações realizadas até aquele ponto serão descartadas.

Finalmente, a operação para inserir um novo elemento na cabeça da lista persistente é apresentada na Figura 3.17. A primeira grande diferença para uma versão volátil do mesmo procedimento é que é necessário o uso de uma transação (linhas 3–9). Isto ocorre porque as operações de alocação do nodo (linha 4, código da Figura 3.16) e atualização dos ponteiros (linhas 5 e 8) devem ser atômicas, já que uma falha em qualquer desses pontos pode deixar a estrutura de dados em um estado inconsistente. Note também que o código que cria o novo nodo é executado dentro de uma transação (Figura 3.16). Nesse caso, o PMDK faz um "achatamento" (*flattening*) das transações, ou seja, é como se existisse apenas uma. Uma outra diferença para a versão volátil é que, como discutido anteriormente, é necessário explicitamente versionar todos os dados que são alterados dentro da transação para que eles possam ser salvos no undo-log e a transação consiga se recuperar corretamente em caso de falhas. Isso é feito através da chamada da linha 7. Note que o único dado que precisa ser versionado é a cabeça da lista, já que o nodo em si (alterado na linha 5) já é versionando automaticamente pelo alocador de memória.

```
1  TOID(struct Node) createNewNode(PMEMobjpool *pop, int data)
2  {
3    TOID(struct Node) p_newNode;
4
5    TX_BEGIN(pop) {
6      p_newNode = TX_NEW(struct Node);
7
8      D_RW(p_newNode)->data = data;
9      D_RW(p_newNode)->p_next = TOID_NULL(struct Node);
10   } TX_END
11
12   return p_newNode;
13 }
```

Figura 3.16: Procedimento para criação de um novo nodo usando o pool persistente.

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

Figura 3.17: Procedimento para inserção de novo nodo na lista persistente.

3.5. Conclusão

Neste minicurso apresentamos os principais conceitos relacionados à memória persistente, suas tecnologias e seu uso. Para exemplificar o uso de PM, utilizamos a biblioteca Intel PMDK e mostramos como uma estrutura de dados simples, como uma lista ligada persistente, pode ser implementada para fazer uso das vantagens da PM em relação à memória de trabalho tradicional. Já neste exemplo simples é fácil de perceber que ainda há um longo caminho de pesquisa a ser percorrido para que o uso de PM seja tão simples, direto e menos propenso a falhas. Em particular, eventuais erros na programação utilizando PM frequentemente só se manifestam quando há efetivamente uma falha no sistema computacional em questão, o que dificulta a localização e a eventual correção de erros de implementação. Neste minicurso nosso propósito também foi mostrar que dispositivos com memória persistentes já são uma realidade e que a programação de tais dispositivos vem ganhando bastante interesse tanto por parte da academia quanto pela iniciativa privada.

Agradecimentos

Este minicurso teve apoio dos processos nº 2019/26702-8 e 2018/15519-5 da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

Referências

[Åkerman 2005] Åkerman, J. (2005). Toward a universal memory. *Science*, 308(5721):508–510.

[Baldassin et al. 2021] 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).

[Boukhobza et al. 2018] Boukhobza, J., Rubini, S., Chen, R., and Shao, Z. (2018). Emerging NVM: A Survey on Architectural Integration and Research Challenges. *ACM Trans. Des. Autom. Electron. Syst.*, 23(2):1–32.

[Burr et al. 2008] Burr, G., Kurdi, B., Scott, J., Lam, C., Gopalakrishnan, K., and Shenoy, R. (2008). Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464.

[Chen and Yen 2009] Chen, E. and Yen, T. (2009). Comparing SLC and MLC Flash Technologies and Structure.

[Doh et al. 2010] Doh, I. H., Kim, Y. J., Park, J. S., Kim, E., Choi, J., Lee, D., and Noh, S. H. (2010). Towards Greener Data Centers With Storage Class Memory: Minimizing Idle Power Waste Through Coarse-Grain Management In Fine-Grain Scale. In *Conference On Computing Frontiers*.

[Eshraghian 2010] Eshraghian, K. (2010). Evolution of Nonvolatile Resistive Switching Memory Technologies: The Related Influence on Hetrogeneous Nanoarchitectures. In *Transactions on Electrical and Electronic Materials*.

[Firth 2014] Firth, S. (2014). The machine – HP labs launches a bold new research initiative to transform the future of computing.

[Guo et al. 2010] Guo, X., Ipek, E., and Soyata, T. (2010). Resistive Computation: Avoiding the Power Wall with Low-Leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)*.

[HP Labs 2014] HP Labs (2014). The Machine: A New Kind of Computer.

[Intel 2018] Intel (2018). Intel optane technology, press kit.

[Intel 2021] Intel (2021). eadr: New opportunities for persistent memory applications.

[International Technology Roadmap for Semiconductors (ITRS) 2007] International Technology Roadmap for Semiconductors (ITRS) (2007). Emerging Research Devices.

[International Technology Roadmap for Semiconductors (ITRS) 2009] International Technology Roadmap for Semiconductors (ITRS) (2009). Emerging Research Devices.

[International Technology Roadmap for Semiconductors (ITRS) 2010] International Technology Roadmap for Semiconductors (ITRS) (2010). 2010 Update Overview.

[Jeongdong Choe 2017] Jeongdong Choe, T. I. (2017). Intel 3d xpoint memory die removed from intel optane™ pcm (phase change memory).

[Kim et al. 2007] Kim, E.-k., Shin, H., Jeon, B.-g., Han, S., Jung, J., and Won, Y. (2007). FRASH: hierarchical file system for FRAM and flash. In *International Conference on Computational Science and Its Applications (ICCSA)*.

[Lee et al. 2009] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. (2009). Architecting Phase Change Memory as a Scalable DRAM Architecture. In *International Symposium on Computer Architecture (ISCA)*.

[Lim et al. 2009] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F. (2009). Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA. ACM.

[Masuoka et al. 1984] Masuoka, F., Asano, M., Iwahashi, H., Komuro, T., and Tanaka, S. (1984). A new flash e2prom cell using triple polysilicon technology. In *Electron Devices Meeting, 1984 International*, volume 30, pages 464–467.

[McGrath 2008] McGrath, D. (2008). 'universal memory' race still on the starting block. EE Times.

[Micron 2006] Micron (2006). 1Gb: x4, x8, x16 DDR3 SDRAM Features.

[Micron 2006] Micron (2006). TN-29-19: NAND Flash 101 NAND vs. NOR Comparison.

[Mishra et al. 2011] Mishra, A. K., Dong, X., Sun, G., Xie, Y., Vijaykrishnan, N., and Das, C. R. (2011). Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs. In *International Symposium on Computer Architecture (ISCA)*.

[Mohan et al. 1992] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162.

[Peng et al. 2020] Peng, I., Wu, K., Ren, J., Li, D., and Gokhale, M. (2020). Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 916–925.

[Qureshi et al. 2009] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. (2009). Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *International Symposium on Computer Architecture (ISCA)*.

[Ramos and Bianchini 2012] Ramos, L. and Bianchini, R. (2012). Exploiting phase-change memory in cooperative caches. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 227–234.

[Ramos et al. 2011] Ramos, L. E., Gorbatov, E., and Bianchini, R. (2011). Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA. ACM.

[Scargall 2020] Scargall, S. (2020). *Programming Persistent Memory - A Comprehensive Guide for Developers*. Apress, 1st edition.

[Solihin 2019] Solihin, Y. (2019). Persistent Memory: Abstractions, Abstractions, and Abstractions. *IEEE Micro*, 39(1):65–66.

[Toshiba 2006] Toshiba (2006). NAND vs. NOR Flash memory.

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

[Vance 2014] Vance, A. (2014). With 'the machine,' HP may have invented a new kind of computer. *BusinessWeek: technology*.

[Wu et al. 2009] Wu, X., Li, J., Zhang, L., Speight, E., Rajamony, R., and Xie, Y. (2009). Hybrid Cache Architecture with Disparate Memory Technologies. In *International Symposium on Computer Architecture (ISCA)*.

[Zhang and Li 2009] Zhang, W. and Li, T. (2009). Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[Zhou et al. 2009] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. (2009). A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *International Symposium on Computer Architecture (ISCA)*.