PUCRS

FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS BRAHM

TÉCNICA DE JOURNALING PARA SISTEMAS DE ARQUIVOS BASEADOS EM MEMÓRIAS NÃO VOLÁTEIS

Porto Alegre
2018

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TÉCNICA DE JOURNALING PARA SISTEMAS DE ARQUIVOS BASEADOS EM MEMÓRIAS NÃO VOLÁTEIS

LUCAS BRAHM

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul

Orientador: Dr. Avelino Francisco Zorzo

Porto Alegre
2018

Ficha Catalográfica

B813t Brahm, Lucas

Técnica de journaling para sistemas de arquivos baseados em memórias não voláteis / Lucas Brahm . – 2018.
63 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Avelino Francisco Zorzo.

1. sistema de arquivos. 2. memória não volátil. 3. sistema operacional. I. Francisco Zorzo, Avelino. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Lucas Brahm

Técnica de journaling para sistemas de arquivos baseados em memórias não voláteis

Tese/Dissertação apresentada como requisito parcial para obtenção do grau de Doutor/Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciencia da Computação, Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul.

Aprovado em 28 de Agosto de 2018.

BANCA EXAMINADORA:

Prof. Dr. Rodrigo da Rosa Righi (UNISINOS)

Prof. Dr. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS)

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS - Orientador)

TÉCNICA DE *JOURNALING* PARA SISTEMAS DE ARQUIVOS BASEADOS EM MEMÓRIAS NÃO VOLÁTEIS

RESUMO

Tecnologias emergentes de memórias não voláteis prometem velocidades próximas as das memórias voláteis e serão conectadas no barramento de memória juntamente com a DRAM. Isto permite um acesso através de instruções de leitura e escrita em memória de forma muita rápida e simplificada. Com este novo modelo, surge a necessidade de novas formas de acesso otimizadas com alto desempenho e que consigam garantir a consistência e integridade da memória.

Este trabalho propõe utilizar um sistema de arquivos otimizado para trabalhar com estas memórias e aperfeiçoar seu sistema de *journaling*, sistema utilizado para garantir a consistência e atomicidade dos metadados. Neste trabalho, foram utilizadas instruções especificas do processador e reduzido as restrições de ordem de escrita, além de simplificando a lógica de gerenciamento do sistema de arquivos. Estas melhorias mostraram ganhos de desempenho e mantiveram a garantia de consistência e integridade que o sistema de arquivos garantia. Os resultados obtidos mostram que em algumas aplicações específicas é possível obter um ganho de 49% no desempenho em uma das soluções propostas.

Palavras Chave: sistema de arquivos, memória não volátil, sistema operacional

JOURNALING TECHNIQUE FOR NON-VOLATILE MEMORY-BASED FILE SYSTEM

ABSTRACT

Emerging non-volatile memories should reach latency as close as volatile memories and shall be connected to the memory bus alongside DRAM. This allows access through read and write instructions in memory in a very fast and simplified way. This new model brings a need to ensure consistency and integrity with high performance.

This work proposes to use an optimized file system to work with these memories and to improve its journaling system, one used to ensure consistency and atomicity of metadata. In this work, we use specific processor instructions and reduce write order restrictions, in addition to simplifying the file system management logic. These enhancements have shown performance gains and have maintained the consistency and integrity of file system. The results show that in some specific applications a gain of 49% is possible using one of the proposed solutions.

Keywords: file system, non-volatile memory, operational system

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

CD-ROM - Compact Disc Read-Only Memory
COW - copy-on-write
CPU - Central Processing Unity
CR - checkpoint region
DAX - Direct Access
DRAM - Dynamic Random Access Memory
FTL - Flash Translate Layer
HDD - hard disk drives
LAD - Laboratório de Alto Desempenho
LFS - Log-structured file system
NFS - Network File System
NTFS - New Technology File System
NVM - Non-Volatile Memory
PCM - Phase Change Memory
PMFS - Persistent Memory File System
POSIX - Portable Operating System Interface
PUCRS - Pontifícia Universidade Católica do Rio Grande do Sul
RAWL - raw word log
SSD - Solid-State Drive
STT-RAM - Spin-Torque Transfer RAM
TxB - Transaction Begin
TxE - Transaction End
VFS – Virtual File System
WB - write-back
WC - write-combining
WRL - wear rate level
WT - write-through
XIP - eXecute In Place

SUMÁRIO

1. INTRODUÇÃO

A diferença de desempenho entre processador e formas de armazenamento de dados cresce desde as últimas décadas. Esta diferença se tornou ainda mais perceptível recentemente, já que cada vez mais programas estão exigindo uma quantidade intensiva de dados, tanto na academia quanto na indústria [7]. Formas de armazenamento tradicionais, como *hard disk drives* (HDD), dominaram o mercado por muito tempo devido à sua mídia de baixo custo e tempo razoável de acesso aos dados. Entretanto, a natureza mecânica dos discos magnéticos torna difícil reduzir o seu tempo de acesso [34][37], por isso, não conseguem acompanhar o crescimento da velocidade dos processadores.

Uma forma de reduzir a diferença de desempenho entre unidade central de processamento (CPU - *Central Processing Unity*) e disco magnético foi a utilização de memórias de estado sólido, que possuem melhor desempenho e menor consumo de energia em relação ao disco. Entretanto, as memórias Flash, que são largamente utilizadas em SSDs (*Solid-State Drives*), ainda tem desempenho muito inferior em relação à memória DRAM (*Dynamic Random Access Memory*) [26][35] e elas são acessadas como um dispositivo de bloco e não são endereçadas a *byte* [49], fazendo que um bloco inteiro seja escrito mesmo se apenas uma pequena quantidade de *bytes* for modificada.

Novas tecnologias de memórias não voláteis (NVM - *Non-Volatile Memories*), que começam a surgir, permitem o acesso aos dados com granularidade de *byte* e prometem velocidades próximas a da DRAM. Além de poderem ser acessadas aleatoriamente sem penalidades, como a DRAM, também possuem a vantagem dos dados ficarem persistentes, como nos discos magnéticos. Memórias como Phase Change Memory (PCM) [1], Spin-Torque Transfer RAM (STT-RAM) [50], Resistive RAM [44] e 3D XPoint [40] são as tecnologias recentes da área de NVM e vêm sendo amplamente pesquisadas.

Entretanto, os sistemas atuais não estão preparados para trabalhar com estas novas memórias de forma eficiente [42]. Desde então, um enorme esforço por parte da indústria e academia está sendo feito para se obter o melhor desempenho destas memórias como, por exemplo, adição de novas instruções do processador, novas linguagens de programação e *frameworks* repensados e modificados.

Uma possível interface de acesso a estas memórias seria utilizar um sistema de arquivos. O sistema de arquivos controla como os dados são estruturados e como são salvos. Muitos sistemas de arquivos são feitos para um propósito específico ou para uma tecnologia específica [19]. Eles também são responsáveis por manter a consistência e atomicidade dos dados e metadados, características que possuem um alto custo envolvido e se tornam alguns dos principais desafios a serem tratados de maneira mais eficiente possível.

Alguns sistemas de arquivos desenvolvidos para estas NVMs como em [49], [29] utilizam mecanismos que serializam a execução e barreiras de escritas para garantir a consistência do sistema de arquivos. Esta dissertação apresenta uma proposta com uma nova técnica de *journaling* com o objetivo de melhorar o desempenho do sistema de arquivos relaxando os requisitos, aumentando o desempenho e garantindo a consistência e atomicidade. A implementação utilizada um sistema de arquivo otimizado para esta nova classe de memórias não voláteis. O objetivo principal é reduzir o número de barreiras de escrita existentes em um sistema de arquivos. Desta forma, acredita-se que a performance geral do sistema poderá melhorar em algumas situações.

Este trabalho está organizado da seguinte forma. A Seção 2 possui o referencial teórico do trabalho, onde na Seção 2.1 explica como funciona o sistema de arquivos do sistema operacional, na Seção 2.2 mostra algumas características destas nova classe de memórias, na Seção 2.3 discute os principais problemas e soluções para estas memórias e na Seção 2.4 faz uma análise dos problemas e soluções apresentados do estado da arte. A Seção 3 apresenta os problemas trabalhados e soluções desenvolvidas no trabalho. A Seção 4 mostra os resultados obtidos no trabalho. Por fim, a Seção 5 finaliza o trabalho.

2. REFERENCIAL TEÓRICO

Neste tópico será desenvolvido o conhecimento necessário para entender como funciona um sistema de arquivos no sistema operacional, ilustrar as características físicas desta nova classe de memórias não voláteis e seus principais desafios no sistema. Com isto, haverá uma melhor compreensão do funcionamento do sistema de arquivos utilizado para estas memórias junto com os problemas existentes que a solução proposta tenta mitigar.

2.1 Sistema de arquivos

Um programa de computador geralmente precisa armazenar informações necessárias para sua execução. Estas informações podem ficar na memória DRAM, onde são rapidamente acessadas e modificadas. Entretanto o programa fica restrito apenas ao tamanho máximo do espaço de endereçamento virtual. Também corre o risco de em caso de alguma falha do programa, ou um problema no sistema, as informações serem perdidas. Para isto utiliza-se uma forma de armazenamento persistente, onde as informações podem ficar retidas permanentemente, além de permitir o armazenamento de uma quantidade grande de dados.

Os discos magnéticos foram utilizados por muito tempo como forma de armazenamento devido ao seu baixo custo e tempo razoável de acesso. Eles são organizados em sequências de blocos de tamanho fixo. Os arquivos são abstrações criadas para um programa conseguir acessar um dado persistente do disco ou de qualquer outra forma de armazenamento. A parte do sistema operacional que cuida dos arquivos de um dispositivo é chamada de sistema de arquivos. O sistema de arquivo é responsável pelo gerenciamento de como os arquivos são estruturados, nomeados, acessados, usados, protegidos e implementados [5]. Sem um sistema de arquivos, as informações presentes em um dispositivo de armazenamento seria apenas uma grande quantidade de dados, onde não seria possível distinguir onde cada informação começa ou termina.

Há diversos tipos de sistema de arquivos, cada um com suas propriedades de flexibilidade, segurança, tamanho, velocidade, estruturas e lógica. Alguns sistemas de arquivos são desenvolvidos para alguns dispositivos ou aplicações específicas. Por exemplo, o sistema de arquivos ISO 9660 é muito utilizado em CD-ROMs e se aproveita da característica desta mídia: todos os tamanhos de arquivos são conhecidos com antecedência e nunca se alterarão durante o uso.

O sistema de arquivos também pode ser utilizado em outras mídias como discos rígidos, memórias flash, fitas magnética, discos ópticos. Também podem ser utilizado para um armazenamento temporário utilizando a DRAM como no caso do *tmpfs*. Outro exemplo seria o sistema de arquivos NFS, onde os arquivos podem ser acessados por protocolo de rede.

2.1.1 Arquivos e Diretórios

Arquivos e diretórios (ou pastas) são abstrações criadas pelo sistema de armazenamento. Um arquivo é um espaço linear de *bytes* que podem ser escritos ou lidos. Na maioria dos sistemas, o sistema operacional não sabe muito sobre o arquivo (por exemplo, não se sabe se é uma foto ou um arquivo de texto). O sistema de arquivos apenas possui a função de armazenar os dados no dispositivo de armazenamento e fazer a leitura deles quando solicitado. Já o diretório é uma abstração da qual contém uma coleção de arquivos e diretórios presentes nele. Colocando diretórios dentro de outros diretórios é possível construir uma árvore de diretórios onde todos os arquivos e diretórios são armazenados [45].

A Figura 1 exemplifica uma hierarquia de diretórios. Ela se inicia com um diretório raiz (no sistema operacionais baseados em Unix, o diretório raiz se inicia com “/”) e utiliza algum tipo de separador para nomear os diretórios subsequentes. Se um arquivo chamado “foo” for criado no diretório “tmp” presente no diretório raiz “/”, ele será referenciado por seu caminho absoluto, sendo neste caso “/tmp/foo”.

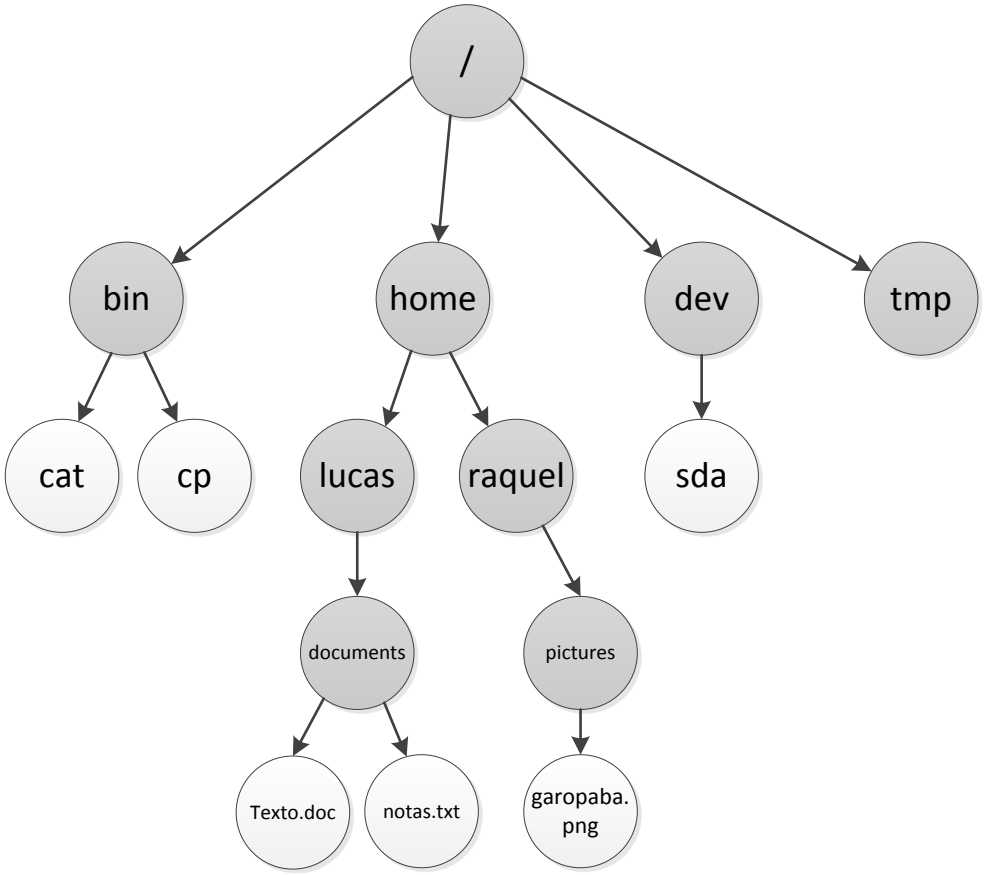


Figura 1. Exemplo de uma árvore hierárquica de arquivos. Os nodos cinza representam diretórios enquanto que os nodos brancos representam arquivos.

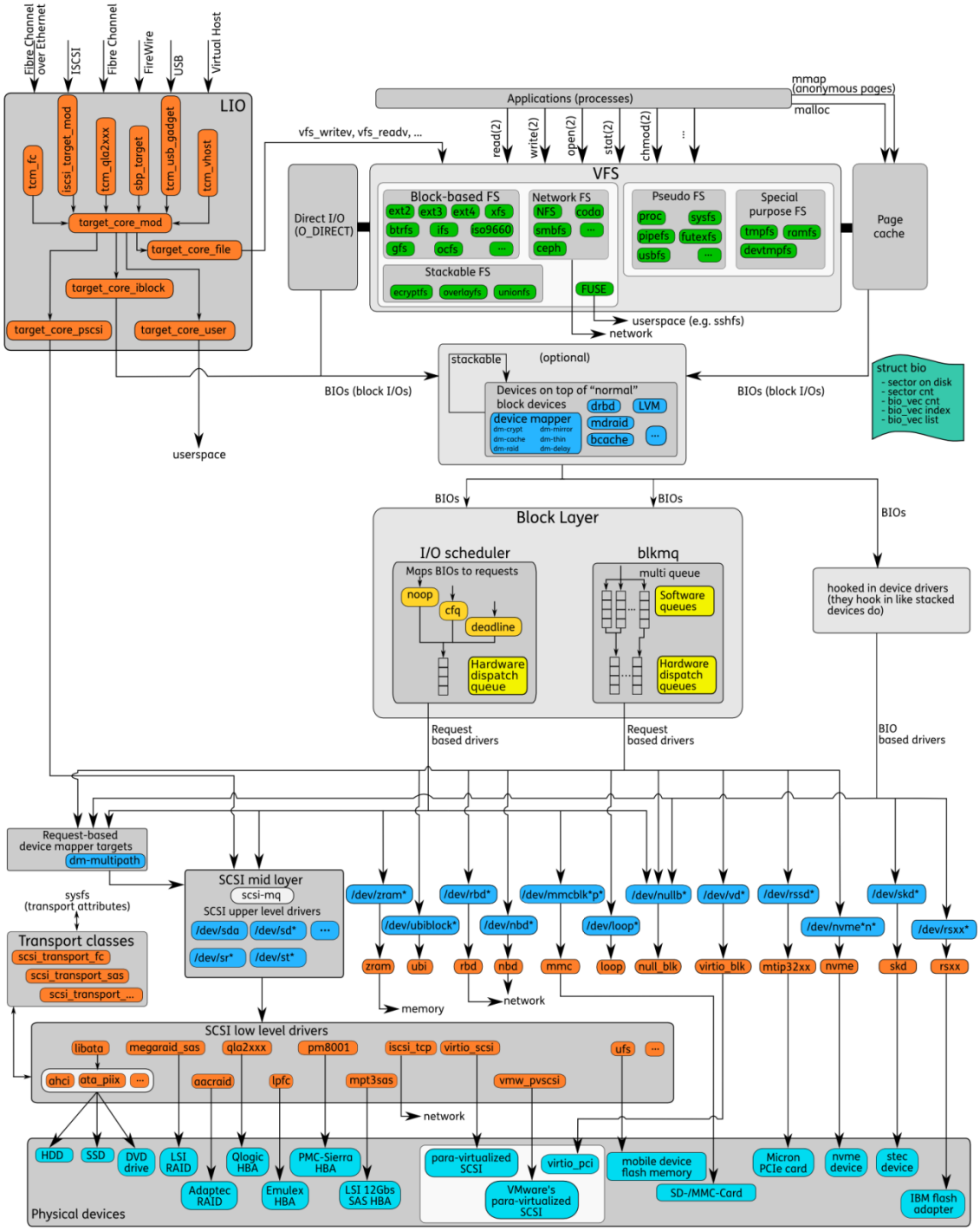


Figura 2. Camadas do Linux que tratam do armazenamento em dispositivos. [52]

2.1.2 Sistema de arquivo no Linux

O Linux possui compatibilidade com múltiplos tipos de sistemas de arquivos como Windows, Unix e outros. O sistema de arquivos virtual (VFS – *Virtual File System*) é uma camada de software do núcleo do sistema operacional que trata todas as chamadas do sistema relacionadas a arquivos através de uma

interface comum. Esta interface é utilizada tanto para sistema de arquivos baseados em disco, baseados em rede, entre outros [12].

Os programas possuem algumas operações que podem realizar com arquivos (algumas destas operações são listadas no item 2.1.3). Estas operações serão requisitadas para o VFS que as encaminhará para o sistema de arquivos especifico na qual as executará. As operações são solicitadas através de uma chamada de sistema. Os programas geralmente utilizam uma biblioteca ou API (biblioteca C - libc) na qual internamente irá realizar as chamadas de sistema. Na Figura 2, na parte mais superior das camadas, temos os programas (processos) fazendo chamadas ao sistema de arquivos virtual do Linux que serão tratados por um sistema de arquivos específico.

2.1.3 Operações com o sistema de arquivos

Uma das operações mais básicas e essenciais de um sistema de arquivos é a criação de arquivos. Através da chamada de sistema *open()* junto com a flag *O_CREAT*, um programa consegue criar um arquivo. Esta função recebe como entrada o caminho do arquivo, flags que indicam como será o comportamento do arquivo e um terceiro campo para especificar permissões. A chamada *open()* pode ser utilizada para abrir arquivos ou para cria-los, caso não existam. Outra chamada que possibilita a criação de um arquivo é a chamada *creat()*, que equivale chamar *open()* com as flags *O_CREAT* | *O_WRONLY* | *O_TRUNC*. A chamada *open()* retorna um descritor de arquivos (*file descriptor*). Um descritor de arquivos é um número inteiro, privado por processo e usado para acessar arquivos.

Uma vez com o arquivo aberto é possível efetuar leituras e escritas nele. Utilizando o descritor de arquivo, junto a um *buffer* de memória e a quantidade de *bytes*, a chamada de sistema *write()* realiza a cópia do *buffer* para o arquivo. Já para ler de um arquivo, a chamada de sistema *read()* realiza a cópia do arquivo para um *buffer* de memória e retorna a quantidade de *bytes* que foi possível ler, além de avançar a posição do arquivo. Desta maneira, o arquivo pode ser lido do início ao fim utilizando a chamada de sistema, pois sempre a posição do arquivo é avançada.

Estas chamadas descritas funcionam bem para realizar operações de leitura e escrita sequenciais, pois a posição do arquivo é sempre avançada. Como isto nem sempre é desejado, é possível manipular a posição do arquivo onde será realizada alguma operação. Fazendo o uso da chamada de sistema *lseek()* e passando como argumento o descritor do arquivo, o deslocamento e a configuração do deslocamento, é realizado a configuração do deslocamento de onde ocorrerá a escrita ou leitura de um arquivo aberto.

Mesmo após a chamada de sistema *write()* ter sido completada, não é garantido que a escrita será efetivada no dispositivo de armazenamento. O que ocorre no sistema, na maioria dos casos, é que a escrita é armazenada em um

buffer por questões de desempenho e após algum tempo ela será efetivamente escrita no dispositivo. Caso ocorra um problema no sistema neste meio tempo, a escrita não será realizada. Entretanto é possível forçar para que todas as escritas pendentes sejam efetivadas utilizando a chamada *fsync()*. Esta chamada recebe como parâmetro o descritor de arquivo e só retornará quando todas as escritas forem realizadas. Este é um recurso útil para sistemas mais críticos onde necessitam desta garantia para as escritas.

Para renomear um arquivo ou trocar ele de diretório, a chamada de sistema *rename()* pode ser usada. Esta chamada geralmente é implementada com a característica de ser atômica, ou seja, garantidamente será efetivada por completo ou ficará no estado antigo. Isto é útil para programas que criam arquivos temporários e depois os substituem pelo arquivo original.

O sistema de arquivos também mantém uma grande quantidade de informações do arquivo além dos seus dados. Informações como o dono e grupo do arquivo, tamanho, horário do último acesso e da última modificação são exemplo de informações que são geralmente chamadas de metadados. Estas informações ficam presentes nos *inodes* do sistema de arquivos do Linux. Para realizar a leitura destas informações, utiliza-se a chamada de sistema *stat()* ou *fstat()*.

A chamada de sistema *link()* possui como parâmetro o caminho para o arquivo novo e para o antigo, vinculando o novo arquivo com o antigo em um *link* criado, ou seja, uma nova forma de se referenciar ao arquivo. O arquivo não é copiado. Na prática os dois arquivos possuirão o mesmo número *inode*, referenciando o mesmo metadado. Quando um arquivo é criado, duas estruturas são criadas: o *inode* que mantém as informações referentes ao arquivo e um *link* no diretório que faz a referencia entre o nome dado ao arquivo e seu metadado.

Para remover um arquivo utiliza-se a chamada de sistema *unlink()*. O nome desta chamada remete a operação reversa da chamada *link()*. Quando é chamada, o sistema de arquivos verifica o contador de *links* que aquele metadado possui (presente no *inode*) e somente se não possuir mais referencias o arquivo é efetivamente apagado. Cada vez que a chamada *link()* é realizada, o contador de *links* é incrementado e, por seja vez, a chamada *unlink()* decrementa este contador. Há também outra forma de criar *links* chamada de *soft link*, onde é possível criar links entre diretórios e entre outros sistemas de arquivos. O *soft link* é um tipo especial de arquivo e é tratado de maneira diferente do *hard link*.

Já para criar um diretório utiliza-se a chama de sistema *mkdir()*. O sistema de arquivos cria o diretório com duas entradas: uma para o próprio diretório (com o nome de “.”) e uma referência para o diretório pai (com o nome de “..”). Para apagar um diretório, a chamada *rmdir()* é usada com o caminho do diretório. Para ser possível remover um diretório, é necessário que ele esteja vazio. Para fazer a leitura dos nomes de arquivos existentes em um diretório utiliza-se a chamada *readdir()*.

Para realizar todas as operações com arquivos necessitamos primeiramente que o sistema de arquivos esteja criado e montado. Para criar um sistema de arquivos, pode-se utilizar uma ferramenta chamada *mkfs*. Especificando o dispositivo e o sistema de arquivos desejado, é criado um sistema de arquivos vazio. Tendo um sistema de arquivos criado, é necessário montá-lo no sistema. Isto é possível com a chamada de sistema *mount()* especificando o dispositivo, o tipo de sistema de arquivos e o diretório destino. Esta chamada irá adicionar à árvore de diretórios existente uma sub-árvore com novos arquivos.

2.1.4 Organização do sistema de arquivos

Para tornar o gerenciamento mais prático, os dispositivos de armazenamento são divididos em blocos de tamanho fixo. O tamanho do bloco implica problemas se for mal dimensionado. Caso seja muito pequeno, um arquivo grande necessitaria de muitos blocos e demandaria múltiplas buscas e atrasos para ser lido, reduzindo o desempenho. Se o tamanho de bloco for muito grande ocorre desperdício de espaço. Se for muito pequena, desperdício de tempo. Por isto, blocos fixos de 4KB são geralmente usados [5].

Ao dividir o dispositivo de armazenamento em blocos, os dados dos arquivos ficam dispersos em um ou mais blocos espalhados pelo disco. É necessário armazenar quais destes blocos pertencem ao arquivo, além de outras informações como o seu tamanho, o seu dono, suas permissões, horário de acesso e modificação. Estas informações de metadados ficam armazenadas em uma estrutura chamada *inode*. Reserva-se uma região especial do disco (alguns blocos) para o armazenamento dos *inodes*. Esta região é conhecida como tabela de *inodes*. Os *inodes* geralmente são estruturas pequenas (em torno de 128 *bytes*) que não ocupam muito espaço. Logo, em um mesmo bloco, é possível armazenar diversos *inodes*.

Mesmo havendo como armazenar dados e metadados, é necessário haver uma forma de controlar quais posições estão livres ou ocupadas. Para isto, geralmente utiliza-se um mapa de bits (bitmap) ou uma lista que sempre aponta para o próximo bloco livre. No mapa de bits, cada bit é usado para indicar se o objeto/bloco está livre (bit 0) ou está ocupado (bit 1), por exemplo.

A Figura 3 exemplifica a disposição física das estruturas do sistema de arquivos no dispositivo de armazenamento. Estas estruturas são fundamentais para o gerenciamento, o controle e o acesso dos arquivos.

2.1.4.1 Superblock

Uma estrutura fundamental presente em sistema de arquivos é o superblock. O superblock contém informações essenciais do sistema de arquivos como seu nome, seu estado, quantidade de *inodes* e de blocos existentes no sistema, início da tabela de *inodes*. Quando um sistema de arquivo

é montado, o sistema operacional irá primeiramente ler do superblock para iniciar diversos parâmetros e para posteriormente saber onde procurar as informações solicitadas no dispositivo de armazenamento.

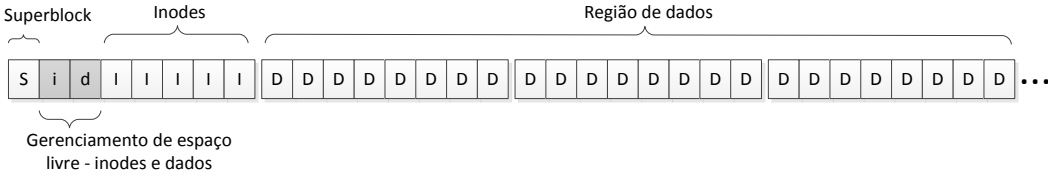


Figura 3. Exemplo da disposição das estruturas do sistema de arquivos no dispositivo de armazenamento dividido em blocos.

2.1.4.2 Index Node (*inode*)

O *inode* é uma das estruturas mais importantes do sistema de arquivos. Cada *inode* é referenciado por um número (chamado *inumber*). Na Tabela 1 é exemplificado algumas informações que das quais ele possui. Estas informações são também chamadas de metadados do arquivo. Uma informação muito importante contida é a referência da posição dos blocos alocados. O *inode* é geralmente uma estrutura pequena e não possui espaço para armazenar os ponteiros de tantos blocos. Para isto utilizam-se ponteiros indiretos que, ao invés de apontar para um bloco de dados do arquivo, ele aponta para outro bloco contendo mais ponteiros. O *inode* acaba armazenando ponteiros diretos para os blocos e, quando necessita mais espaço, aloca um bloco de dados para armazenar mais ponteiros e um ponteiro indireto é adicionado. Para o sistema de arquivos suportar arquivos grandes basta suportar ponteiros indiretos de vários níveis, como ponteiros indiretos duplos e triplos.

Tabela 1. Exemplificação do conteúdo de um *i-node*.

Tamanho	Nome	Descrição
2	mode	permissão de leitura/escrita/execução
2	uid	dono do arquivo
4	size	tamanho em *bytes* do arquivo
4	time	horário do último acesso ao arquivo
4	ctime	horário da criação do arquivo
4	mtime	horário da última modificação
4	dtime	horário que o *inode* foi apagado
2	gid	grupo que pertence o arquivo
2	link_count	quantidade de hard links
4	blocks	quantidade de blocos usados
4	flags	característica do *inode*
4	osd1	dependência do sistema operacional
60	block	conjunto de ponteiros aos blocos
4	generation	versão do arquivo (usado pelo NFS)

2.1.4.3 Entrada de Diretórios

Diretórios ou pastas basicamente são uma lista de nomes associadas a um *inode*. Diretórios são tratados de maneira semelhante a um arquivo. Ele possui seu *inode* e seu bloco de dados, onde irá armazenar a lista de arquivos ou diretórios presentes nele. Na Tabela 2 temos um exemplo de como são armazenadas as *dentries* (entradas de diretório). Além do número do *inode* e seu nome, há um campo para armazenar o tamanho do nome (e muitos sistemas de arquivos acabam limitando este tamanho máximo de 256 *bytes*) e um campo chamado *reclen* (*record length*). O campo *reclen* armazena a quantidade de espaço total que aquela entrada de diretório está ocupando (com todas as informações, inclusive o nome). Este campo acaba muitas vezes maior que o necessário para ficar alinhado com um número múltiplo de *bytes*, pois em algumas arquiteturas isto é necessário. Para apagar uma entrada no diretório, basta incrementar o *reclen* da última entrada, fazendo que a entrada "desaparecer" na hora for buscada.

Tabela 2. Exemplificação de um diretório com entradas.

inumber	reclen	namelen	name
4	12	2	.
5	12	3	..
77	12	4	meu
9	16	5	teste
14	20	11	nomegrande

2.1.5 Operações internas do sistema de arquivos

2.1.5.1 Abertura de arquivo

A chamada de sistema recebe como parâmetro o caminho do arquivo desejado. Por exemplo, caso seja solicitada para abrir o arquivo "/myfolder/myfile", o processo realizado pelo sistema de arquivo segue os passos da Tabela 3. É necessário encontrar o *inode* do arquivo "myfile" através do caminho do arquivo numa operação chamada de *tranverse*.

Esta operação começa no diretório raiz do sistema de arquivos. Para se encontrar o *inode* de um arquivo ou diretório, é buscado no diretório pai o *inumber* do mesmo. Entretanto o diretório raiz possui a particularidade de não possuir um diretório pai, então a maioria dos sistemas de arquivos possui um *inumber* pré-definido para ele (geralmente atribuído com *inumber* 2).

O sistema de arquivo faz a leitura do *inode* do diretório raiz. O *inode* possuirá os ponteiros para os blocos de dados que ele possui. Os blocos de dados do diretório raiz possuem as entradas de diretório, relação entre o nome dos arquivos e seu *inumber*. Será buscado nesta lista o próximo arquivo ou

diretório do caminho do arquivo especificado que será aberto. Para este exemplo será buscado o diretório “myfolder” no diretório raiz “/”. Ao encontrar seu *inumber*, será feita a leitura do seu *inode*. O *inode* do diretório “myfolder” conterá os ponteiros para os blocos de dados com a lista de arquivos e diretórios presentes do diretório. Ao encontrar o *inumber* do arquivo “myfile”, será realizada a leitura de seu *inode*. O último passo do tratamento da chamada *open()* será ler o *inode* de “myfile” para a memória, realizar uma verificação de permissões, alocar um descritor de arquivos para o processo e retornar para o usuário.

Tabela 3. Passos realizados durante a abertura de um arquivo.

1	Leitura do *inode* de “/”
2	Leitura dos dados de “/”
3	Leitura do *inode* de “myfolder”
4	Leitura dos dados de “myfolder”
5	Leitura do *inode* de “myfile”

2.1.5.2 Leitura de arquivo

Depois de aberto, o arquivo está apto a ser lido. A primeira leitura será na posição zero do arquivo, a menos que a chamada *lseek()* tenha configurado a posição desejada. Será lido o *inode* para encontrar o ponteiro do bloco, lido o bloco e atualizado o *inode* com o horário de último acesso. Será também atualizado o deslocamento de leitura do arquivo onde a próxima leitura lerá o próximo bloco, além da tabela de arquivos abertos para esse descritor de arquivo.

2.1.5.3 Escrita em arquivo

Após a abertura do arquivo, é possível realizar escritas nele. Diferente da leitura, o processo de escrita pode alocar blocos. O processo de escrita não é tão simples: é necessário primeiramente decidir quais dos blocos livres serão alocados, necessitando fazer uma leitura do gerenciamento de controle de blocos do sistema de arquivos. Ao ser decidido qual bloco será alocado, será necessário marca-lo como ocupado no gerenciamento de controle. O *inode* também precisará ser lido e atualizado com a informação do novo bloco alocado. Após todas estas operações, o dado do novo bloco alocado poderá finalmente ser escrito. Isto tudo acabando gerando um total de cinco operações de IO.

Para a criação de um arquivo é também necessário alocar um novo *inode* e uma nova entrada no diretório pai. Para alocar o novo *inode*, é necessário fazer a leitura do controle de *inodes* do sistema de arquivos, e marcar o novo *inode* como ocupado. O novo *inode* também deve ser inicializado. Já para adicionar a entrada de diretório, é necessário fazer a leitura do *inode* do diretório pai, escrever no bloco de dados do diretório, e atualizar o *inode* do diretório. Caso o

diretório não possua espaço e necessite de mais blocos, mais operações serão realizadas. Na Tabela 4 e Tabela 5 são exemplificadas as operações de criação e escrita do arquivo “/myfolder/myfile” respectivamente.

Tabela 4. Passos realizados na criação de um arquivo.

1	Leitura do *inode* de “/”
2	Leitura dos dados de “/”
3	Leitura do *inode* de “myfolder”
4	Leitura dos dados de “myfolder”
5	Leitura do bitmap de *inodes* (leitura dos *inodes* livres)
6	Escrita no bitmap de *inodes* (marcar o novo *inode* como ocupado)
7	Escrita de dados em “myfolder” (adicionar o novo nome do diretório)
8	Escrita do *inode* de “myfolder” (atualizar tamanho, etc)
9	Escrita do *inode* de “myfile” (inicialização do *inode*)

Tabela 5. Passos realizados durante a escrita de um arquivo.

1	Leitura do *inode* de “myfolder”
2	Leitura do bitmap de blocos (leitura dos blocos livres)
3	Escrita no bitmap de blocos (marcar bloco como ocupado)
4	Escrita no bloco de “myfolder” (escrita do dado solicitado)
5	Escrita no *inode* de “myfolder” (atualizar tamanho, etc)

2.1.6 Cache

Como visto acima, operações simples com arquivos podem gerar muitas operações de IO. Para dispositivos de armazenamento cuja latência é grande, o tempo para cada operação seria inviável. Por isto utiliza-se a memória DRAM como uma *cache* para acelerar as operações. Quando o dado é lido do disco, a informação permanecerá por um tempo na *cache*, acelerando a resposta se o mesmo dado for solicitado. Para as escritas, os dados são armazenados em um *buffer* e ficam atualizados apenas na memória DRAM, inicialmente. Isto é útil, pois é possível agrupar e reduzir a quantidade de IOs melhorando a performance.

No Linux utiliza-se a *cache* de páginas (*page cache*) para melhorar o desempenho. Quando um bloco é solicitado, primeiramente é procurado na cache de páginas, e, caso não seja encontrado, é solicitado para a camada de blocos do sistema operacional (ver Figura 2). Quando um dado é modificado, ele ficará sujo na *cache* de página e será sincronizado com dispositivo de armazenamento conforme a política adotada pelo sistema. Caso aconteça alguma falha do sistema neste meio tempo, os dados que ficarem na cache e não foram gravados serão perdidos. Por isto há a chamada *fsync()* que força as

escritas no dispositivo de armazenamento ou a opção de abrir o arquivo com a opção DIRECT IO, onde todos os acessos serão diretos ao dispositivo de armazenamento e não passarão pela *cache*.

2.1.7 Consistência

Como visto anteriormente, diversas estruturas do sistema de arquivos são atualizadas durante uma requisição de escrita do usuário. Entretanto, caso ocorra uma falha no sistema durante a atualização destas estruturas, o sistema de arquivos pode ficar em um estado inconsistente. Por exemplo, caso um usuário escreva em um arquivo existente, o gerenciador de blocos livres deve ser atualizado marcando um novo bloco alocado, o *inode* deverá ser atualizado com o ponteiro para este novo bloco, atualizado seu tamanho total e, finalmente, o bloco alocado será escrito seu dado. No caso de falha podemos ter algumas possíveis situações:

- Apenas atualizar o gerenciador de blocos livres: isto leva o sistema de arquivos a marcar um bloco como utilizado incorretamente, consumindo espaço sem utiliza-lo.
- Apenas atualizar o *inode*: o *inode* apontará para um bloco que está marcado como livre, além do bloco não possuir o dado escrito.

Este exemplo é um dos diversos cenários possíveis onde podem deixar inconsistente o sistema de arquivo caso nem todas as operações sejam completadas. Uma possível solução é utilizar uma ferramenta chamada *fsck*. Ela analisa e corrige inconsistências encontradas. Mas nem todas as inconsistências podem ser corrigidas, como no caso do bloco de dados não ter sido escrito, porém o sistema de arquivos estará consistente do ponto de vista dos metadados.

Uma solução mais ideal seria sair de um estado consistente do sistema de arquivos para outro estado consistente de forma atômica. Infelizmente isto não é algo simples, visto que as escritas são sequenciais e uma escrita será realizada antes da outra. Para isto, é possível utilizar algumas técnicas descritas abaixo.

2.1.7.1 Journaling

Um dos mecanismos mais conhecidos para garantia de consistência é chamado de *journaling* ou *write-ahead logging*. Esta técnica é usada em sistemas de arquivos como ext3, ext4, raiserfs, Windows NTFS. A ideia do *journaling* é escrever em um lugar reservado (chamado de *log*) o que se pretende alterar no sistema de arquivos antes de efetivamente fazê-lo. Em caso de uma falha durante uma operação, é possível consultar no *log* o que se pretendia fazer e tentar novamente. A recuperação do sistema de arquivos se torna mais rápida,

pois não é necessário procurar inconsistências por todo o sistema de arquivos por inconsistências. Basta procurar no *log* e verificar se há pendências não concluídas. Um sistema de arquivos com *journaling* possuirá a mesma estrutura tradicional semelhante ao da Figura 3 com a adição de uma área reservada para o *log* (*journal*) ilustrada na Figura 4.

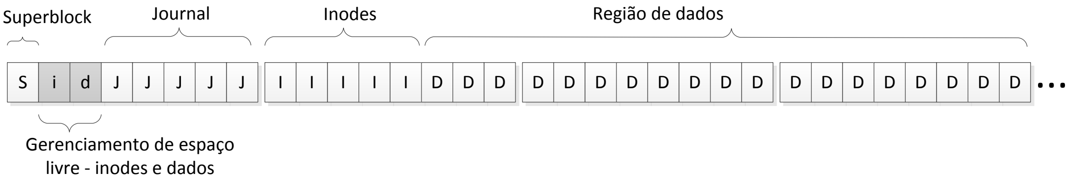


Figura 4. Exemplo da disposição das estruturas do sistema de arquivos no dispositivo de armazenamento com uma área reservada ao *journal*.

Na Figura 5 é ilustrado um exemplo de como fica armazenado os dados no *log* (*journal*) no caso de uma escrita em um arquivo existente for solicitada. O primeiro bloco a ser escrito é chamado de *Transaction Begin* (TxB) e ele marco o inicio da transação, além de possuir algumas informações das estruturas que serão alteradas (posição física onde a estrutura será atualizada, por exemplo) junto com um identificador da transação. Os três blocos do meio são o *inode* (I), o bitmap marcado com o novo bloco ocupado (B) e o bloco de dados a ser escrito (D). O último bloco chamado de *Transaction End* (TxE) marca o fim da transação. O ato de finalizar uma transação é chamado de *commit*.

Devido a questões de otimização do sistema, as escritas da transação podem ser reordenadas. Por isto utiliza-se mecanismos que garantem que a *Transaction End* só será escrita após as demais escritas da transação terem sido finalizadas.

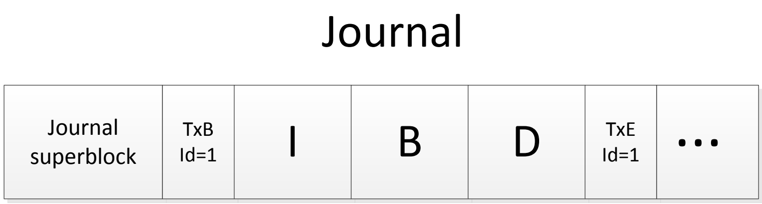


Figura 5. Exemplo da disposição das estruturas no *journal*.

Depois da transação ser escrita no disco, pode ocorrer a atualização das informações. Este processo é chamado de *checkpoint* e no exemplo será escrito I, B, D nas suas respectivas posições do dispositivo de armazenamento.

Como o espaço reservado para o *journal* é limitado, ele é geralmente utilizado como um *buffer* circular. Também é preciso haver uma forma de marcar onde começa e termina cada transação e saber quais estão ativas. Por isto, também há um controle (*journal* superblock) para gerenciar quais transações ainda estão ativas. Quando a transação é completamente finalizada, ela deve ser liberada do *journal*.

Conforme observado, existe uma quantidade de operações de IO desencadeada para tratar uma chamada *write()*. Caso o sistema de arquivos

garanta consistência com uma técnica de *journaling*, a quantidade de operações cresce consideralmente, dobrando a quantidade de escritas ao dispositivo. Por isto alguns sistemas de arquivos permitem realizar *journal* apenas de metadados. Desta forma os dados (bloco D da Figura 5) não farão parte da transação. No caso de utilizar a técnica de *journaling* apenas para metadados, os dados que precisam ser atualizados são previamente escritos para então ser realizado o *log* dos metadados e sua atualização.

2.7.1.2 Log Structured

A grande quantidade de escritas no disco geradas por cada chamada de *write()* foi um fator que levou a criação do mecanismo *log structured*. Por necessitar realizar escritas em diversas estruturas espalhadas aleatoriamente pelo disco, o desempenho das operações ficava lento. Foi observado que escritas sequenciais pelo disco são mais rápidas que escritas aleatórias, então, baseado neste princípio e para obter melhor desempenho com operações de escrita, o sistema de arquivos *log-structured* (LFS) foi criado. Quando é necessário realizar escritas no disco, o sistema as buferiza em um segmento de memória e, quando este segmento estiver cheio, é realizada a escrita no disco em um grande segmento sequencial. O LFS não sobrescreve o dado, mas sempre escreve segmentos em locais livres.

Encontrar o *inode* em um sistema de arquivos tradicional é geralmente uma tarefa simples, pois eles geralmente ficam em uma posição fixa no dispositivo de armazenamento. Dado um *inumber* do *inode*, o *inode* pode ser facilmente encontrado em um vetor de *inodes*. Entretanto no LFS os *inodes* ficam espalhados por todo o dispositivo de armazenamento e, como eles nunca são sobrescritos, a versão mais atualizada dos *inodes* permanece mudando.

Para lidar com este problema, uma estrutura chamada *inode map* (*imap*) existe para mapear um número de *inode* para a posição onde está armazenada a versão mais recente do *inode*. O *imap* é escrito juntamente com o segmento. Como o *imap* pode variar, uma região fixa chamada de *checkpoint region* (CR) existe para apontar para os últimos *imaps*. O CR é atualizado apenas periodicamente para evitar degradar o desempenho. Na Figura 6 é ilustrado um exemplo de como é disposta as estruturas no dispositivo de armazenamento de um LFS.

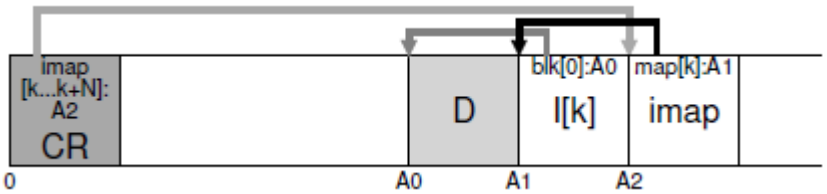


Figura 6. Disposição das estruturas para um sistema de arquivos *log structured* [45].

O processo de leitura de um arquivo no LFS se inicia pela leitura do *checkpoint region* e então o *inode map*. A leitura do *inode map* é mantido em

cache na memória. Após isso, a partir de um número de *inode* é possível ler a última versão do *inode* e então realizar a leitura de seu conteúdo.

No LFS, para manter as escritas eficientes, versões antigas dos arquivos permanecem espalhadas pelo dispositivo de armazenamento. Estas versões antigas são chamadas de *garbage* (lixo). É necessário periodicamente fazer uma limpeza estes dados antigos, tornando estes blocos livres novamente. O processo de limpeza é chamado de *garbage collection*, técnica semelhante usada em algumas linguagens de programação para liberar memória não usada pelos programas.

No começo de cada segmento há uma estrutura chamada *segment summary* para saber quais blocos estão ativos. Nela contém informações de cada bloco, o seu *inode* e sua localização. Para verificar se o bloco está ativo, basta conferir se estas informações estão coerentes com a leitura com *inode map*, *inode* e localização informada no *inode*.

Para manter redundância no sistema, o LFS possui duas cópias do *checkpoint region* localizadas em posições diferentes. O CR possui informação com o *timestamp* no início e fim de sua estrutura e devem estar coerentes para ser considerados válidos. No LFS será sempre considerado o CR que possuir o *timestamp* mais atual válido.

2.2 Nova classe de memorias não voláteis

Tecnologias emergentes na área de memórias não voláteis vêm surgindo nos últimos anos. O exemplo mais recente é de uma memória chamada 3D Xpoint desenvolvida pela Intel e Micron que promete ser mil vezes mais rápida que as flash NAND e 8 a 10 vezes mais densas que a DRAM [40]. Isto está gerando uma mudança enorme na indústria eletrônica e na informática.

Estas memórias possuem vantagens como serem endereçadas a *byte*, baixo consumo de energia quando não há atividade, alta densidade e alta velocidade de leitura. Elas também costumam possuir velocidades de escrita e leitura assimétricas, onde uma escrita consome mais tempo e energia. A Tabela 6 faz uma comparação de algumas destas memórias.

Tabela 6. Comparação das tecnologias de memórias [49].

Parâmetro	DRAM	Flash Nand	RRAM	PCM
Densidade	1X	4X	2X-4X	2X-4X
Latência de leitura	60ns	25µs	200-300ns	200-300ns
Velocidade de escrita	~1GB/s	2,4MB/s	140MB/s	~100MB/s
Durabilidade	10^{16}	10^{4}	10^{6}	10^{6} a 10^{8}
Endereçada a *byte*	Sim	Não	Sim	Sim

Este capítulo apresenta alguma das principais tecnologias de memórias emergentes com suas características físicas.

2.2.1 Phase Change Memory

A memória não volátil Phase Change (PCM) utiliza um material chamado de calcogeneto para armazenar seu estado. O calcogeneto possuiu dois estados estáveis: o cristalino e o amorfo. Para atingir o estado cristalino, a camada de calcogeneto é aquecida a uma temperatura superior a de cristalização e inferior a temperatura de derretimento, ficando com uma baixa resistência (estado lógico “1”). Ao aquecer a uma temperatura superior a temperatura de derretimento e esfriada rapidamente, é atingido o estado amorfo de alta resistência (estado lógico “0”) [8].

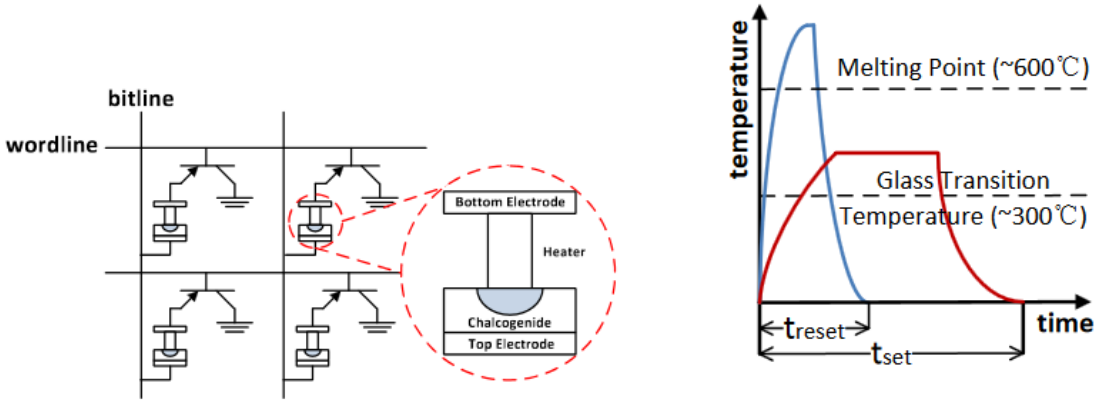


Figura 7. Célula de memória da PCM e operação de RESET (“0”) e SET (“1”) [8].

2.2.2 Spin-Transfer Torque RAM

A memória Spin-Transfer Torque (STT), ao invés de armazenar a informação com cargas elétricas, utiliza um túnel de junção magnética. Ele é composto por duas camadas ferromagnéticas e uma camada com um túnel de barreira. Uma das camadas ferromagnéticas (camada de referência) possui sua direção magnética fixa, enquanto a outra pode alterar sua posição magnética através de um campo eletromagnético. Se as duas camadas ferromagnéticas possuírem a mesma direção, o túnel terá uma resistência baixa indicando estado “0”. Se as duas camadas tiverem direções opostas o túnel apresentará uma alta resistência indicando estado “1” [8].

2.2.3 Memristor

Memristor é um dispositivo com dois terminais cuja resistência depende da magnitude e polaridade da voltagem aplicada além da duração que a voltagem é aplicada. Quando a voltagem é desligada, o memristor permanece com sua última resistência até ser submetida a uma nova voltagem [46].

2.3 Desafios ao utilizar NVM

Nesta seção serão apresentados alguns desafios de se trabalhar com NVM e soluções de alguns trabalhos. A divisão foi realizada em classe de problemas mais conhecidos e trabalhados.

2.3.1 Localidade da NVM

A primeira etapa antes de trabalharmos com esta nova classe de NVMs, é definir a forma de acessá-la com baixa latência, sendo possível extrair ao máximo seus benefícios. As formas sugeridas em diversos trabalhos são: acessá-la através de um barramento de I/O, através de um barramento de alta velocidade como PCI express [11], ou através do barramento de memória [49][26][20][32].

Entretanto, dispositivos de armazenamento tipicamente estão conectados a um controlador de barramento ou a um controlador de armazenamento. Como a maior parte do tempo para realizar leituras e escritas nestes dispositivos é o tempo de acesso deste dispositivo, o atraso destas arquiteturas não afetava significativamente o desempenho. O tempo de acesso do barramento é muito pequeno, mesmo quando comparado aos SSDs de flash NAND que possuem latências em dezenas de microsegundos [26].

Tecnologias de memórias promissoras, como PCM, possuem latência na faixa de centena de nanosegundos, que as torna apenas 2-5 vezes mais lentas que a DRAM. Logo, conectá-la a um barramento de I/O ou uma PCI não teria um bom desempenho, já que estes são baseados em acesso a bloco. Por isto, a melhor forma de acessá-la seria através do barramento de memória, estando lado a lado da DRAM. O acesso à NVM seria através de funções comuns de escrita e leitura em memória. Com isto, o acesso seria de baixa latência, além de se aproveitar da hierarquia de cache para melhorar o desempenho [26] e também possuir o acesso com granularidade de *byte*.

Acessar a NVM pelo barramento de memória parece ser a solução mais aceita em quase todos os trabalhos, como em [20][26][49], apesar de haver algumas desvantagens, como a possiblidade do tráfego gerado pela NVM interferir com os acessos à DRAM por utilizarem o mesmo barramento.

2.3.2 Cache e *buffers*

Como visto na Seção 2.3.1, colocando a NVM no barramento de memória, pode-se acessar a memória persistente com funções de leitura e escrita de forma semelhante à maneira feita na DRAM. Por ela estar no barramento de memória, é possível utilizar a memória cache para conseguir um melhor desempenho.

Entretanto, mesmo na presença de uma memória não volátil, existem *buffers* voláteis e caches na hierarquia de memória por causa das vantagens de desempenho que eles oferecem [32], como mostrado na Figura 8. Então, caso ocorra uma falha no sistema (devido à falta de energia ou *crash* no sistema),

todos os dados presentes nas estruturas voláteis do sistema que não conseguirem chegar à memória persistente serão perdidos.

Uma solução para este problema é a utilização de mecanismos que possibilitam à recuperação e mantenham a memória em um estado consistente. Outra solução seria a utilização de *buffers* e cache persistente, como em [15][31][37].

Para manter a consistência, técnicas como *copy-on-write* [33] (COW) ou *journaling* (ou *logging*) são usadas. Estes mecanismos aumentam a demanda por espaço na memória persistente e reduzem a performance do sistema ao aumentar o tráfego da memória com transferência extra de dados. E ainda outras soluções que trabalham com memória persistente usam instruções como *flush* (*clflush*) e *memory fence* (*mfence*) para garantir a consistência ao fazer *flush* das linhas sujas na cache na barreira de cada atualização na memória persistente. Todas estas técnicas podem acabar gerando um aumento de 120% no tráfego da memória ao se comparar com uma DRAM [31].

Por isso, em [31] foi proposta uma arquitetura chamada *Kiln*, que consiste de uma cache não volátil e uma memória não volátil. A DRAM e NVM compartilham o barramento de memória. O dado é transferido da cache volátil para a não volátil com uma transação. A cache não volátil sempre terá os dados mais atualizados, enquanto a NVM terá uma cópia antiga. Atualizações na memória podem ser feitas diretamente nos dados da cache não volátil, sem a necessidade de realizar *logging* ou COW. Em [37], é utilizado *buffer* não volátil para melhorar o desempenho das escritas e utilizado técnicas de COW.

A solução de utilizar cache e *buffers* não voláteis é simples para manter a consistência da memória e obter bom desempenho, porém a inerente baixa durabilidade destas memórias a torna proibitiva. Como no caso das memórias flash, cada bloco pode ser apagado um número limitado de vezes antes de perder a capacidade de armazenar o estado [9]. Além disso, a velocidade das memórias cache são superiores as destas novas memórias não voláteis.

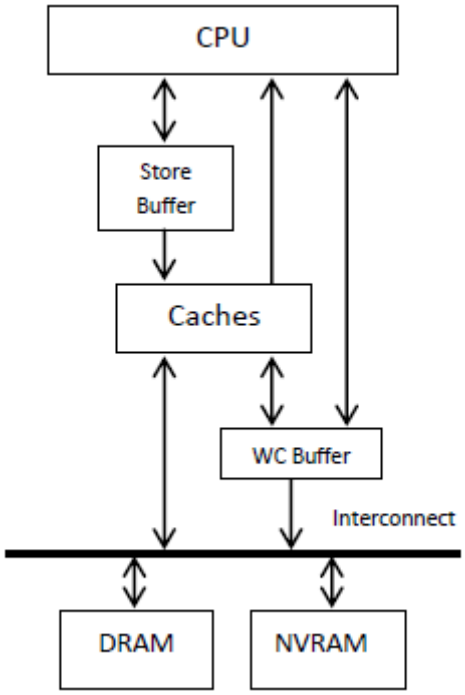


Figura 8. Modelo de arquitetura simplificado [32].

2.3.3 Ordem de escritas

Outro problema presente na utilização da cache é a falta de garantia da ordem que as escritas chegarão à memória. A hierarquia de cache e os controladores de memória foram desenvolvidos para a memória volátil e podem reordenar as escritas para otimizar o desempenho, não havendo como evitar esse reordenamento nas arquiteturas presentes [26].

Manter a ordem das escritas é um dos principais fatores para manter a consistência. Por exemplo, caso em que um nodo que precisa ser inserido em uma lista encadeada armazenada na memória persistente, o valor do nodo precisa ser escrito na memória persistente antes de atualizar o ponteiro da lista [31]. Em caso de uma falha do sistema, apenas uma das escritas pode chegar na memória persistente e, caso as escritas estivessem fora de ordem, após o reinicialização do sistema, a memória estará em um estado inconsistente.

Em uma política de cache *write-back* (WB), as escritas são feitas em alguma linha da memória cache. Quando esta linha precisa ser substituída, ela será gravada no próximo nível da hierarquia de memória, podendo em algum momento ser na memória persistente. Isto gera um grande indeterminismo, pois as escritas na memória são realizadas conforme a política de substituição da cache, podendo gerar escritas fora de ordem na memória persistente.

Uma solução para este problema é fazer um *flush* destas linhas de cache, forçando o dado a ser escrito na memória, garantindo durabilidade e consistência da mesma.

Por exemplo, Mnemosyne [20] utiliza-se de três primitivas de hardware: (i) *write-through stores*, que o dado é escrito diretamente para a memória em vez da cache. Na verdade, as escritas vão parar nos *buffers write-combining* (WC, ver Figura 8) através da operação *movntq*; (ii) *fences*, que previnem escritas subsequentes de serem completadas antes das escritas anteriores. Através da instrução *mfence*, a execução será atrasada até que ocorra o *flush* dos *buffers* WC; e (iii) *flushes*, que forçam a escrita de uma linha da cache para a memória.

Em [32] é utilizado a política *write-through* (WT) da cache, em que todas as escritas da CPU resultam em escritas em todos os níveis de cache e na memória persistente. A vantagem deste método é que as leituras continuam com o benefício do dado ainda estar presente na memória cache. Apesar de uma escrita da CPU com WT ser mais lenta que uma com WB, não é mais necessário *flushes* na cache cujo custo é elevado, além de não ser necessário fazer modificação no código para ser utilizado.

Na solução do BPFS [26], foi proposto um novo mecanismo que possibilita, via software, definir a ordem das escritas ao hardware. Por software, é possível definir barreiras de escrita especiais que delimitam um conjunto de escritas chamas de *epoch*, e o hardware garantirá que cada *epoch* será escrita na memória em ordem, mesmo se escritas individuais forem reordenadas dentro de uma *epoch*. Isto permite desacoplar ordem da durabilidade. Esta solução necessita mudanças de hardware.

Já em [49] é criticado a utilização de WT (como a usada em [32]) por apresentar um desempenho pior em relação ao WB e pelo fato que as memórias persistentes possuem uma largura de banda limitada. Também é criticado a utilização de instruções não-temporais (como a usada em [20]) por tornar a programação mais difícil. Os autores defendem que utilizar WB e realizar o *flush* das linhas sujas da cache (usando *clflush*) e forçando a sua finalização (usando *sfence*) ainda é a solução mais eficiente. Apesar de tudo isto, não é garantida a durabilidade, devido aos controladores de memória. A fim de garantir durabilidade em tal arquitetura, foi proposta uma nova primitiva de hardware (pm_barrier) que garante a durabilidade da escrita na memória não volátil que já tiver realizado o *flush* das caches da CPU.

2.3.4 Consistência e Atomicidade

Garantir apenas a ordem das escritas, não garante a consistência do ponto de visto de um sistema de arquivos. Um sistema de arquivos deve permanecer consistente mesmo quando ocorrer um *crash* ou falta de energia no sistema. Para garantir isto, o sistema de arquivos deve possuir um mecanismo recuperável e consistente para metadados e, opcionalmente, para dados. Uma operação é chamada de atômica se e somente se modificações feitas pela operação são efetivadas em uma espécie de “tudo ou nada”, ou seja, ou a

modificação é totalmente completada ou falha completamente, deixando o dado sem ser modificado [31][49].

Para garantir a atomicidade e manter o sistema de arquivos consistente, técnicas como *journaling*, *shadow paging* e *log-structured* são utilizadas. Apesar de estas técnicas serem de extrema importância, elas acabam gerando um grande tráfego adicional à NVM.

Journaling (ou *logging*) pode ser implementado de duas maneiras: *redo* e *undo* [49]. Com o método *redo* (do inglês refazer), o novo dado a ser escrito é primeiramente salvo em um *log* e posteriormente o dado é atualizado. Em caso de uma falha no sistema, a operação é refeita. No caso do método *undo* (do inglês desfazer), o dado antigo é salvo primeiro, antes do dado novo ser escrito e, em caso de falha, é revertido com o dado antigo que havia sido salvo.

Com *journaling*, cada escrita no sistema de arquivos resulta em duas escritas: uma para o *journal* e outra para o sistema de arquivos. O custo destas escritas é muito elevado, fazendo que a maioria dos sistemas de arquivo que utilizam esta técnica apenas façam *log* dos metadados [26].

Shadow Paging **Erro! Fonte de referência não encontrada.** utiliza COW (*copy-on-write*) ara realizar todas as modificações de dados, de tal forma que o dado original permaneça inalterado enquanto o dado novo é escrito na NVM [26]. Os dados são armazenados em uma árvore, e o novo dado é escrito por COW, onde o bloco pai precisa realizar COW, pois irá apontar para o novo bloco de dado. Este processo é repetido até ser propagado para o topo da árvore, onde uma única escrita na raiz persistirá todos os novos blocos criados através do COW.

Sistemas de arquivo *log-structured* (LFSs, do inglês *log-structured file systems*) foram originalmente desenvolvidos para explorar a alta performance de acessos sequenciais em HDDs. LFS fazem a “buferização” das escritas na memória e convertem elas em grandes escritas sequenciais ao disco [29]. Como a última versão do arquivo é sempre escrita em novas posições da memória, blocos e metadados antigos acabam permanecendo nela [45], necessitando constante limpeza e compactação dos *logs*, gerando uma degradação na performance [29].

Devido ao elevado custo existente na técnica de *journaling*, no BPFS [26] foi criada uma nova técnica que aprimora o *shadow padding* chamada de *short-circuit shadow paging* (SCSP). No BFPS, as estruturas de dados são organizadas em uma árvore composta por blocos de tamanho fixo e a modificação dos dados se baseia na atomicidade em escritas de 64 bits que o hardware garante. Devido a esta garantia é possível modificar diretamente caso a mudança ocorra em até 64 bits do arquivo. Também é possível diretamente modificar metadados e fazer adições de dados a arquivos devido à garantia de atomicidade. Para outros casos, onde são necessário grandes mudanças no sistema de arquivo, é utilizado um *copy-on-write* parcial, fazendo que as cópias fiquem restritas a uma pequena sub-árvore do sistema de arquivo, copiando apenas aquelas porções de velhos dados para uma pequena sub-árvore do sistema de arquivos.

De acordo com [49], sistemas de arquivos que utilizam técnicas como COW ou *log-structured* realizam cópias ou fazem *logs* em granularidade de bloco ou de segmento. Estas técnicas usadas para garantir a consistência geram muitas escritas, especialmente para garantir a consistência de metadados que geralmente requer apenas uma pequena modificação no metadado. Conforme uma análise sobre a consistência de metadados foi mostrada que utilizar *journaling* com uma granularidade de linha de cache ou de 64 *bytes* (chamada de *logging* de grão fino) possui o melhor desempenho comparado com COW e *log-structured*. Devido às desvantagens do *journaling*, ele só é usado com atualizações atômicas nos metadados, utilizando a implementação *undo journaling*. Para a atualização dos dados de arquivos, é utilizado COW.

Já em Kiln [31], é apresentado uma solução que utiliza cache não volátil. Os dados são escritos diretamente nesta cache persistente, sem mecanismos de consistência como *journaling* ou *copy-on-write*. Em caso de falha, apenas a cache ficará inconsistente enquanto a memória não volátil permanecerá com os dados inalterados em um estado consistente.

Também é possível utilizar um sistema de arquivos comum e permitir o acesso direto a NVM e ainda assim garantir a consistência de metadados e dados. Ext4-DAX [36] utiliza o sistema de arquivos Ext4 com DAX (*Direct Access*) para acessar diretamente a NVM sem passar pela cache de páginas do sistema operacional e mantem a consistência dos metadados utilizando *journaling*.

SCMFS [54] utiliza o módulo de gerenciamento de memória virtual do sistema operacional e mapeia os arquivos em regiões de endereço virtual, entretanto não garante a consistência dos metadados e dados.

NOVA [29] é um sistema de arquivos que maximiza o acesso a NVM. Foi utilizado *log-structured*, pois possui melhor resultado para atualizações atômicas quando comparado com *journaling ou shadow paging*. Para atualizações atômicas complexas (como operação de mover entre diretórios), é utilizado *journaling* para atomicamente atualizar os diversos *logs*, já que em NOVA cada *inode* possui seu próprio *log*. Uma vez que cada *inode* possui seu próprio log, isto possibilita atualizações concorrentes entre arquivos sem sincronização. Os *logs* são mantidos na NVM e implementados como uma única lista encadeada para não necessitar alocar grandes regiões contíguas reservadas, enquanto os índices são mantidos na DRAM por questões de desempenho. Para a consistência de dados, é utilizado *copy-on-write*, não sendo necessário realizar o *log* deles.

2.3.5 Durabilidade

A memória Flash possui um número limitado de vezes que seus blocos podem ser apagados [41]. Com estas limitações uma camada de software chamada FTL (*Flash Translate Layer*) cria uma abstração de páginas contínuas que podem ser lidas, escritas e sobre-escritas, além de realizar o gerenciamento dos dados para garantir que as escritas fiquem distribuídas de forma uniforme pela memória [14].

Apesar desta nova classe de memórias não voláteis possuir durabilidade maior que a Flash NAND [49][26][22], ainda há uma preocupação com o desgaste das células de memória após uma certa quantidade de escritas. Ao colocar estas memórias no barramento de memória, elas acabam sendo expostas a uma grande quantidade de escritas. Para lidar com isto, sistemas de arquivos tentam minimizar a quantidade de escritas e adaptam técnicas de *wear-leveling*, que distribuem as escritas igualmente pelas células [26] de forma semelhante às técnicas usadas na memória Flash.

Técnicas de *wear-leveling* podem se basear na quantidade de vezes que cada bloco foi escrito e armazenar esta quantidade, e trocar os blocos “quentes” (com mais escritas) por blocos “frios” (com menos escritas) para conseguir uma escrita balanceada [16]. Em [28] foi proposto um mecanismo chamado WRL (*wear rate level*), que leva em consideração a durabilidade diferente que cada célula de memória possui. WRL usa a taxa de quantidade de escritas pela durabilidade como métrica e troca blocos com taxa alta por blocos com taxa baixa. Em [18] é adicionado ao controlador de memória uma tabela que mapeia o endereço das páginas e a quantidade de escritas feitas a cada página. Quando a quantidade de escritas chega a um valor limite, o controlador gera uma interrupção de *page swap* ao processador. O sistema operacional é responsável por tratar a interrupção e realizar uma troca de página. Em [26] são propostas duas técnicas: (i) tratar dentro de cada página, rotacionando os bits a nível de controlador de memória; (ii) tratar entre páginas, onde é periodicamente trocado o mapeamento virtual-físico das páginas.

Muitos trabalhos assumem que este problema será resolvido por hardware e uma solução por software seria muito complicada e custosa. Outros trabalhos assumem que este problema tende a desaparecer conforme as tecnologias de memórias evoluem.

2.3.6 Interface de acesso

Tradicionalmente, o sistema operacional separa o gerenciamento da memória volátil (através do gerenciador de memória virtual) de dispositivos de armazenamento (através de um sistema de arquivos ou driver). Já que a NVM é endereçada a *byte* (como a DRAM) e persistente (como HDDs e SSDs) [49], ela poderia ser gerenciada de diversas maneiras como: (i) estender o gerenciador de memória virtual para gerenciar a NVM [54]; (ii) implementar um dispositivo de bloco para a NVM e usar como um sistema de arquivos existente (como Ext4) [36]; (iii) utilizar sistema de arquivos existentes modificados para trabalhar melhor

com a NVM [42]; (iv) implementar um sistema de arquivos otimizado para a NVM sem passar pela camada de blocos [49]; (v) utilizar novos modelos de programação que simplificam o acesso à NVM [20][25].

SCMFS [54] é um sistema de arquivos que utiliza o módulo de gerenciamento de memória do sistema operacional para o gerenciamento dos blocos e para manter o espaço sempre contíguo para cada arquivo. Funções como *nvmalloc* e *nvfree* foram adicionadas ao kernel para permitir ao SCMFS o gerenciamento da memória persistente. PMFS [49] é um sistema de arquivos POSIX com otimizações para a NVM. Possui um mmap otimizado que não necessita copiar as páginas acessadas para a DRAM. O mapeamento das páginas é feito diretamente no espaço de endereço da aplicação. A vantagem destes sistemas de arquivos e de outros (como [26]), é que utilizam uma interface POSIX, que é usada por grande parte das aplicações. Esta interface possui o I/O de arquivo (open, read, write) e o I/O de memória mapeada (open, mmap, load/store).

A interface de sistema de arquivos POSIX possui desvantagens como necessitar trocar de modo usuário para modo núcleo e o custo de possuir uma interface genérica que abstrai cada recurso do sistema como um arquivo. Considerando este atraso, Aerie [21] apresenta uma interface de sistema de arquivos flexível que expõe a memória para as aplicações para que elas possam acessar arquivos sem passar pelo núcleo do sistema operacional. As aplicações são ligadas a uma biblioteca de sistema de arquivo que possibilitará o acesso aos dados e se comunica com um serviço que coordenada o acesso concorrente e a atualização de metadados. É possível utilizar um sistema de arquivo do tipo POSIX ou um sistema de arquivos otimizado para determinas aplicações, já que cada um possuirá a sua biblioteca. Em [4], Moneta-D também utiliza uma biblioteca no espaço de usuário para evitar passar pelo núcleo do sistema operacional. Um hardware especializado cuida da proteção dos arquivos e disponibiliza canais para acessar os dados na memória. Os canais e aplicações são mapeados pelo sistema operacional. A biblioteca em espaço de usuário funciona como um driver que abstrai todo o acesso aos dados e funciona com interface POSIX.

Novos modelos de programação facilitam o acesso da aplicação à memória persistente como em Mnemosyne [20], NV-Heaps [25], NVM Library [38]. Em Mnemosyne [20], as aplicações pode acessar à memoria não volátil através de uma interface de programação de baixo nível. Mnemosyne também disponibiliza regiões de memória persistente, que são segmentos de memória virtual armazenadas na NVM, que podem ser criadas automaticamente utilizando a palavra *pstatic* na definição de uma variável ou alocadas dinamicamente. É possível utilizar funções como *pmalloc* e *pfree* para uma *heap* persistente e também são disponibilizadas primitivas que garantem atualizações consistentes na NVM.

Uma forma eficiente de acessar arquivos é fazer o mapeamento do arquivo com a função *mmap*. Este processo mapeia uma região do espaço de

endereçamento da aplicação para um arquivo e copia os blocos do arquivo em uma cache de páginas conforme a demanda [19]. Com sistema de arquivos que suportam *eXecute In Place* (XIP) ou *Direct Access* (DAX) é possível mapear diretamente para o endereço físico da memória, evitando a cópia da cache de páginas [19][29].

2.4 Análise sobre NVMs

Em 2013, a Linux Foundation Collaboration Summit [27] em sua seção “Preparando o Linux para dispositivos de memória não volátil” propôs uma abordagem de adaptação do sistema operacional em três etapas: (i) o sistema de arquivos irá acessar a NVM utilizando *drivers* tradicionais orientados a bloco; (ii) sistema de arquivos serão adaptados para acessar a NVM diretamente, de maneira mais eficiente. Esta etapa garante que o sistema de arquivos será modelado para NVM, continuando a tradicional abstração do sistema de arquivos para manter a compatibilidade; (iii) criar APIs de IO com granularidade de *byte* para novas aplicações. Esta etapa pode acabar com a compatibilidade existente dos sistemas de arquivo.

Na Seção 2.3 foram enumeradas algumas das principais dificuldades e algumas soluções propostas. Alguns trabalhos como [11] e [4] apresentarem soluções interessantes ao otimizar software e hardware. Estas soluções se encaixam na etapa (i) sugerida pela Linux Foundation. Entretanto, a NVM é acessada como um dispositivo de bloco, não sendo uma solução ótima para estas memórias. Mesmo eliminando o escalonador de I/O do sistema operacional e camadas de software, ainda haverá o atraso da controladora de DMA tornando uma latência alta para acessar a NVM.

Alguns trabalhos como [37][18][47][55] defendem a utilização de uma memória única persistente e principal, entretanto estas novas tecnologias de memórias ainda apresentam latência maior e possuem menor durabilidade em relação a DRAM. Conforme as tecnologias avançam isto pode se tornar uma solução viável, caso atinjam desempenho superior ao atraso na realização de cópias entre a memória volátil e não volátil. Entretanto esta solução acaba acentuando novos problemas como a proteção e segurança desta memória.

A solução mais simples para acessar a NVM é através do barramento de memória junto à DRAM, como explicado na Seção 2.3. Com base nessa arquitetura muitos sistemas de arquivos otimizados para esta nova classe de memória não volátil foram desenvolvidos. Eles visam minimizar os problemas anteriormente citados tentando obter o máximo de desempenho. A maioria dos trabalhos utiliza politica de *write-back* na cache e realizando o *flush* da linha cache para forçar o dado a ser escrito na NVM e utilizando uma barreira de memória para evitar reordenamento de escritas. Alguns trabalhos sugerem alterações no hardware para aprimorar este processo, o que são sugestões bem vindas, mas são mais prováveis de aparecem em um futuro mais distante

conforme as tecnologias evoluem e pesquisadores convergem para uma solução mais ideal.

Para garantir a consistência e atomicidade técnicas como *copy-on-write* e *log-structured* são usadas, entretanto essas técnicas possuem granularidade de bloco ou de segmento, o que acaba gerando uma quantidade grande de escritas, especialmente para a consistência de metadados onde geralmente uma pequena quantidade de dados é atualizada. *Log-structured* também necessita de grandes regiões contínuas de memória e de um *garbage collector* para ficar constantemente limpando e compactando dados apagados para liberar espaço. Já a técnica de *journaling* é uma solução simples que possui granularidade fina e que funciona muito bem para atualização de metadados.

Apesar de novas interfaces de acesso e APIs especializadas e otimizadas terem surgido para se trabalhar com NVM, o sistema de arquivos continuará existindo devido a sua interface flexível e para manter compatibilidade com os softwares legados existente, conforme o item da etapa (ii) sugerida na Linux Foundation.

Em resumo, a NVM estará conectada lado a lado da DRAM no mesmo barramento e compartilhando a cache (Figura 9), visto que estas novas memórias chegaram a um desempenho próximo ao da memória DRAM e possuem uma alta densidade. Imagina-se que algumas soluções por hardware irão surgir, pois seria demasiado custoso trata-las por software, como no caso o problema da durabilidade da NVM. Os sistemas de arquivos serão a principal interface de baixo nível com a memória não volátil, onde garantirão atomicidade e consistência de dados e metadados. Soluções simples como técnica de *journaling*, seções críticas pequenas e reduzir as barreiras de escrita aumentam o desempenho e o paralelismo. A principal preocupação em relação a esta nova classe de memórias não voláteis é obter o máximo de desempenho levando em consideração os problemas existentes. Este trabalho apresenta novas técnicas que visam melhorar o desempenho do sistema de arquivos quando é necessário atomicidade e consistência dos metadados.

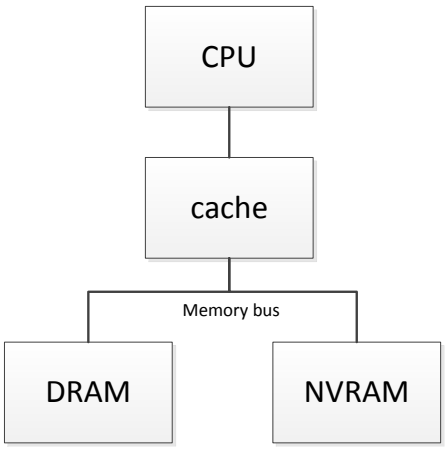


Figura 9. Arquitetura alvo.

3. SOLUÇÕES: ONEFLUSH E MANUALFLUSH

3.1 Contextualização

Considerando a análise feita na Seção 2.4 e visto que a NVM possui maior latência que a DRAM, este trabalho propõe mitigar os problemas que são considerados mais críticos: garantir a ordem de escritas (Seção 2.3.3) e garantir a consistência e atomicidade (Seção 2.3.4).

A ordem das escritas pode ser reordenada por questões de desempenho ou devido à política adotada na cache. Na Figura 10, é exemplificado este problema. As escritas da cache para NVM foram reordenadas. Durante a escrita de um dos dados, ocorre uma falha, fazendo que a memória persistente fique com um dado errado enquanto ele é considerado válido.

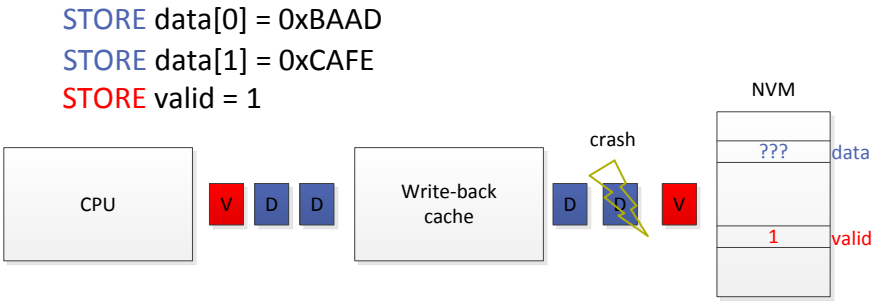


Figura 10 - Inconsistência na memória após falha com escritas desordenadas.

Uma solução simples seria desativar a cache ou utilizar a política de *write-through* (WT) da cache. Estas soluções não apresentam bom desempenho, por isto, muitos trabalhos fazem *flush* das linhas da cache para forçar o dado ser escrito na memória persistente na ordem correta. Após o *flush*, uma instrução do tipo *fence* garante que todas as escritas subsequentes só ocorrerão após as escritas anteriores terminarem. Isto age como uma barreira de escrita. Como exemplificado na Figura 11, o programador consegue definir de maneira correta as escritas na NVM.

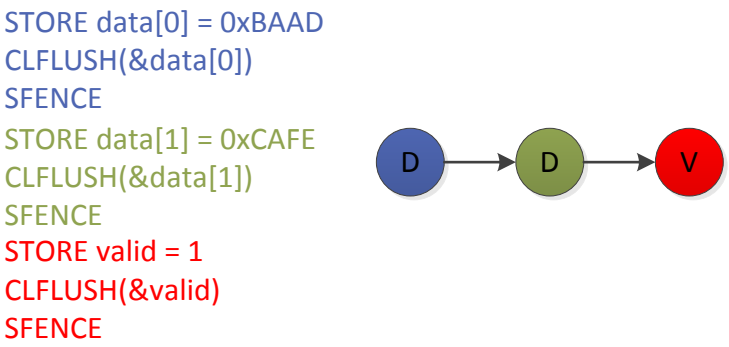


Figura 11 – Exemplo de cada escrita sendo persistida na NVM sequencialmente.

Na Figura 11, pode-se observar que há restrições de ordem desnecessárias. A única restrição que realmente existe é que os dados (data[0] e data[1]) sejam escritos antes que colocados como válidos (valid = 1). Então a ordem dos dados não importa entre si, podendo até mesmo ser atualizados simultaneamente. Entretanto, conforme [23], a instrução *clflush* é serializada entre si, entre escritas e entre *fences*. Um exemplo é mostrado na Figura 13a. O *pipeline* da CPU acaba trancando e a execução é serializada.

A Intel propôs novas instruções para resolver estes problemas, como *clflushopt* (*clflush* otimizado), *clwb* (forçar a escrita da linha da cache sem fazer o *flush*) e *pcommit* (força as escritas para a NVM) [23][24]. Utilizando a instrução *clflushopt*, podemos ver suas melhorias na Figura 12 e Figura 13b.

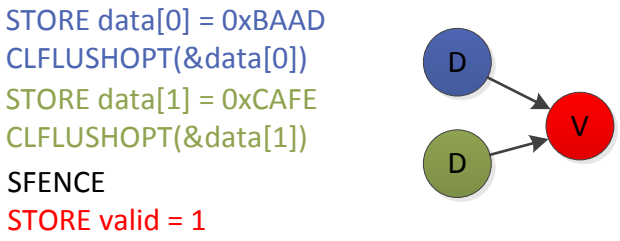


Figura 12 – Persistindo dados utilizando a instrução *clflushopt*.

Alternativa para a instrução *clflushopt* é utilizar a instrução *clwb* que possui o mesmo efeito prático, entretanto não invalida a linha da cache. A cache foi desenvolvida para melhorar o desempenho do sistema e invalidá-la como forma de forçar escritas degrada o desempenho do sistema e reduz sua funcionalidade. A instrução *clwb* é uma ótima alternativa de escrever na NVM sem fazer *flush* da linha da cache, mantendo as operações de leituras com o benefício de poder estar presente na cache.

Após passar pela cache, a escrita ainda não será efetivada, pois estará em uma fila de escrita no controlador de memória. Com a instrução *pcommit* é garantido que a escrita chegou a NVM. Estas novas instruções são usadas por trabalhos mais recentes como [2][29]. Entretanto, a Intel anunciou que a instrução *pcommit* será descontinua [13], já que as plataformas que iriam suportar o módulo de memória da Intel já possuem uma solução em hardware que automaticamente escreve na NVM quando ocorre falha ou desligamento do sistema. Isto é bom, pois delega uma função ao hardware e não necessita mais usar a instrução *pcommit*, que serializava a execução de instruções.

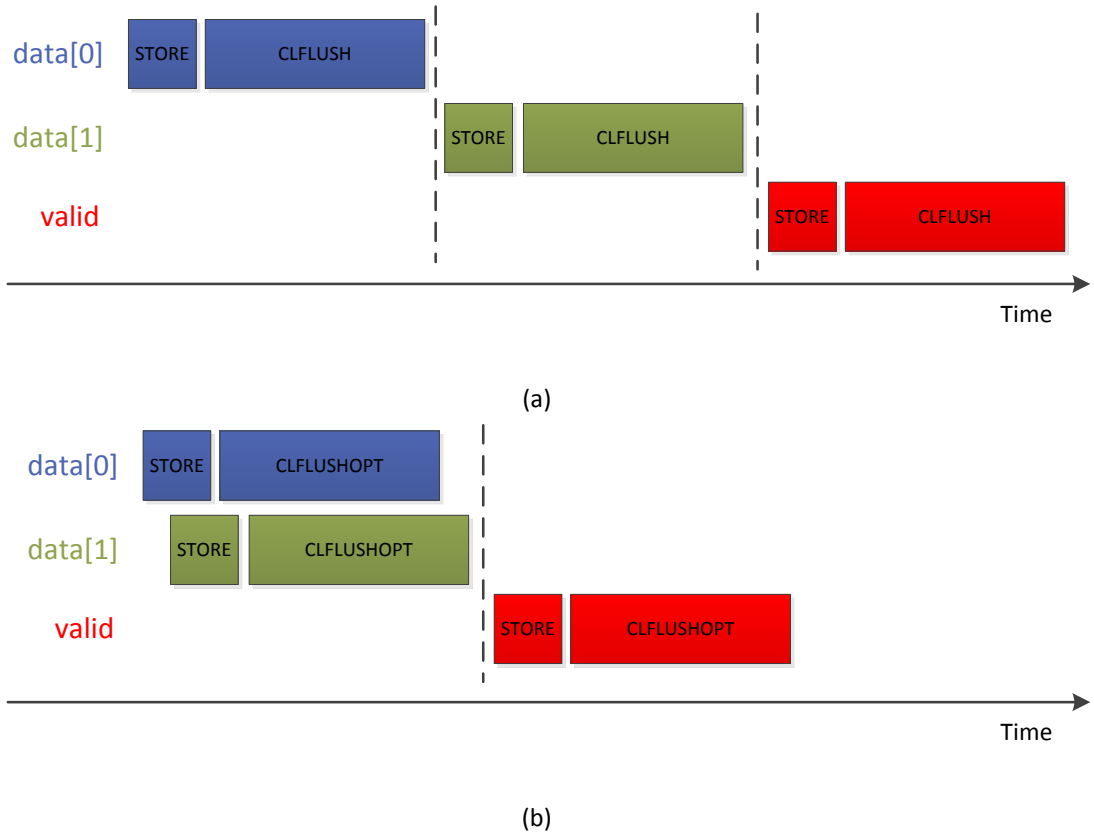


Figura 13 – Comparação de execução com instrução *clflush* (a) e *clflushopt* (b)

Apesar desta nova classe de memórias não voláteis serem muito mais rápidas que o disco rígido ou SSDs, elas ainda são e provavelmente continuarão a ser mais lentas que a DRAM. Técnicas como *batching*, reordenamento e paralelismo ajudam a conseguir um desempenho melhor nestas memórias [2][30].

Em [48], foi mostrado que minimizar as dependências de escrita na NVM pode melhorar a performance até 30 vezes. Na Figura 11 é mostrado um exemplo de dependências desnecessárias e na Figura 12 são mostradas as dependências minimizadas. Entretanto, não era possível paralelizar as escritas, já que a instrução *clflush* serializada a execução. Com as novas instruções disponibilizadas pela Intel (*clwb*, *clflushopt*) foi possível paralelizar a execução.

Através do paralelismo, poderia ser possível alterar múltiplos arquivos simultaneamente. Isto não é algo simples, já que é necessário garantir que uma alteração não deixe a NVM em um estado inconsistente. Para este problema, a técnica de *journaling* poderia ser usada. Permitir múltiplas transações simultâneas necessita poder permitir adicionar simultaneamente diversas entradas de *journals*. Outro problema é verificar se as entradas no *journal* são válidas, já que falhas podem acontecer durante a escrita no *journal*.

Para identificar que as entradas são válidas, uma possibilidade é de adicionar as entradas atomicamente, entretanto isto é difícil, já os processadores apenas garante uma atomicidade de 8 *bytes*. Uma possível alternativa é adicionar uma entrada e depois escrever um bit de validade para validá-la. Outras

possibilidades poderiam ser adicionar um *checksum* no cabeçalho de entrada do *journal* ou usar técnicas como *tornbit RAWL* [49].

3.2 PMFS

Persistent Memory File System (PMFS) [49] é um sistema de arquivos POSIX otimizado para trabalhar com memórias não voláteis, onde os acessos à memória podem ser feitos sem passar pela camada de blocos do sistema operacional, eliminando cópias adicionais desnecessárias. A Figura 14 mostra uma comparação do PMFS com um sistema de arquivos tradicional. O PMFS também é capaz de utilizar o memory-mapped IO para mapear as páginas da NVM diretamente para o espaço de endereçamento da aplicação.

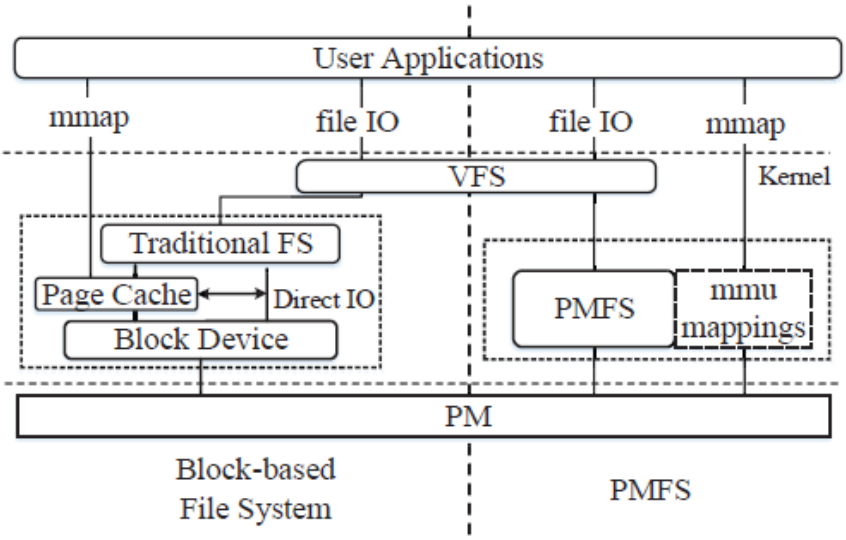


Figura 14. PMFS vs Sistema de arquivo tradicional [49].

No PMFS, toda a memória não volátil é mapeada no espaço de endereçamento virtual do núcleo do sistema operacional quando o sistema de arquivos é montado. Ao expor toda a memória, uma escrita incorreta poderia comprometer o sistema de arquivos. Por isto as páginas da NVM são mapeadas como apenas leitura e são temporariamente atualizadas durante uma escrita.

O layout da estrutura de dados do PMFS é mostra na Figura 15. O superblock possui uma cópia de redundância, seguido pelo *journal* (PMFS-Log) e as páginas dinamicamente alocadas. Os metadados no PMFS são organizados usando uma B-tree. A B-tree é usada para a tabela de *inode* e para os dados nos *inodes*.

Cada alocação na NVM é baseada em páginas com suporte a diversos tamanhos de páginas (4K, 2MB, 1GB). Por padrão é utilizada páginas de 4K para nodos (internos) de metadados, mas os nodos de dados podem ser de 4KB, 2MB, 1GB.

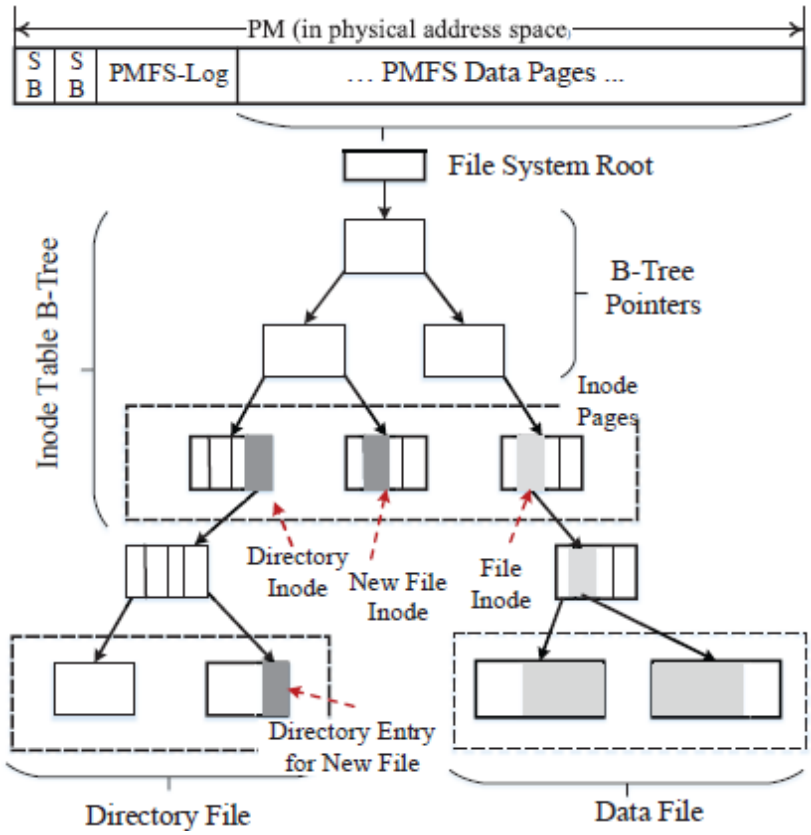


Figura 15. Layout da estrutura dados do PMFS Erro! Fonte de referência não encontrada..

No PMFS é usada a política de cache WB (Write-Back) e é feito *flush* nas linhas de cache com dados modificados através da instrução *clflush*. E para manter a ordem das escritas é utilizada a instrução *sfence*. Para garantir a consistência dos metadados é utilizado *redo journaling* e para a atualização dos dados é utilizado *copy-on-write*. Atualizações atômicas de 64 *bytes* (granularidade de linha de cache) são suportadas.

Para o controle do *log*, há os ponteiros *head* e *tail* que marcam o início e o fim dos *logs* ativos, funcionando como uma janela deslizante marcando os *logs* ativos (ver Figura 16). Para cada transação há um identificador único (*transaction id*) e para cada entrada do *log* há um *header* e uma porção de dados. Há também um identificar especial (*generation id*) que é usado para identificar se a entrada de *log* está válida. Este identificador deve ser o mesmo presente no metadado do *journal* do PMFS. A cada vez que o *log* “gira” (por ser um *buffer* circular) ou depois de se recuperar, este identificador especial é incrementado e isto acaba automaticamente invalidando todas as entradas de *log* presente.

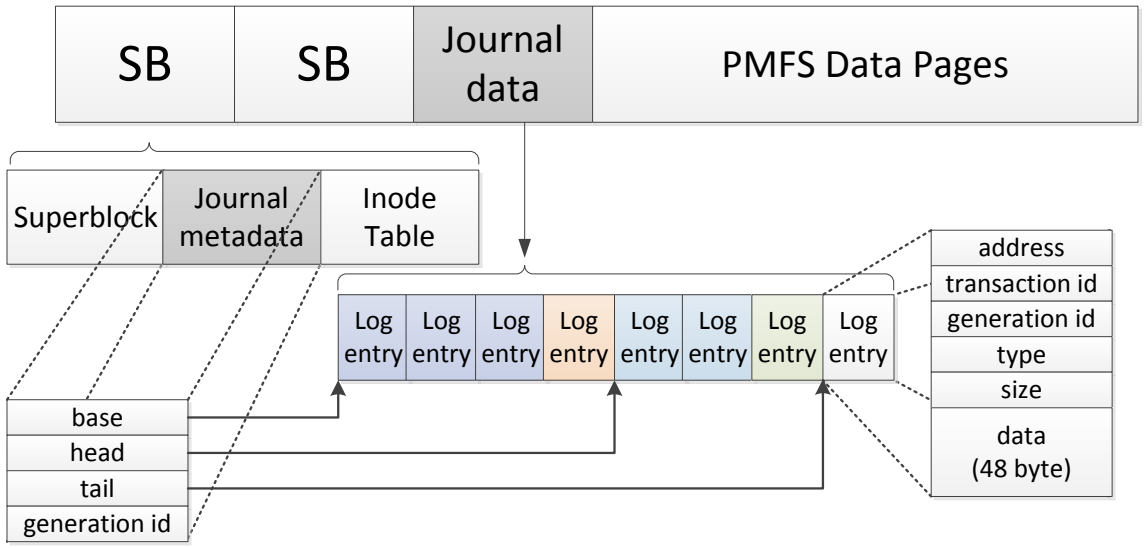


Figura 16. Layout do *journal* do PMFS.

Cada entrada de *log* consegue armazenar 48 *bytes* de dados. O *inode* do PMFS possui 96 *bytes* intencionalmente para ser possível de ser armazenado em duas entradas de *log*.

No caso de uma escrita em um arquivo, o sistema de arquivos primeiramente irá analisar se será necessário alocar mais um bloco de dados. Caso não seja necessário, os metadados podem ser atualizados atomicamente sem a necessidade de técnicas de *journaling*. Caso contrário, será alocado o número máximo de entradas de *logs* requerido ao atomicamente incrementar o ponteiro *tail*. Os metadados são então salvos em uma ou mais entradas de *log* na NVM no *journal*. Após isto, os metadados podem efetivamente ser salvos na NVM. Com isto feito, é feito o *commit* da transação.

Durante o processo de uma transação, depois que todas as entradas de *log* forem escritas, será efetuada uma barreira de escrita para garantir que os *logs* foram escritos, e então é adicionado uma entrada de *log* especial de *commit* seguida de uma barreira de escrita para completar a transação. O processo de limpeza dos *logs* é realizada por uma *thread* que periodicamente libera *logs* que tenham sido feito *commit* e após isto é atualizado o ponteiro *head* seguido por uma barreira de escrita.

3.3 Implementação do *oneflush* e *manualflush*

Foi escolhido o sistema de arquivos PMFS para a base deste trabalho por possuir código aberto e por trabalhar com técnica de *journaling* para garantir consistência e atomicidade de metadados. Visto que as NVMs terão velocidades próximas das memórias DRAM, empregar uma técnica de baixa complexidade e que não necessita um *garbage collector. Journaling* é uma técnica simples com um bom desempenho para atualização geralmente pequena de metadados.

Apesar de ser um sistema de arquivos completo e otimizado para NVMs, ele sofre de alguns problemas: utiliza técnica de *undo journaling*, que garante apenas a integridade do metadado anterior à atualização e um alto custo para gerir e verificar as entradas no *journal*.

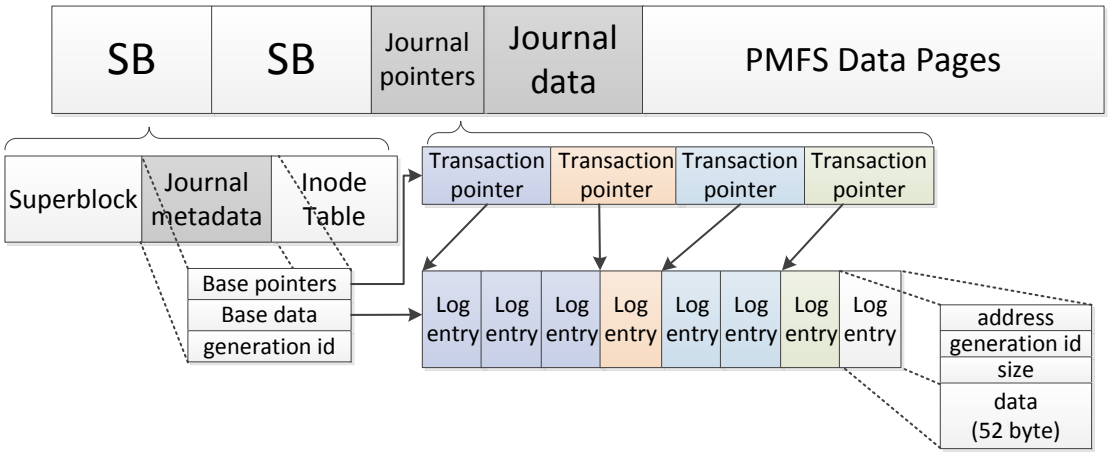


Figura 17. Layout proposto para o PMFS.

O PMFS utiliza o *undo journaling* junto com uma técnica onde uma janela deslizante marca as transações ativas. Este trabalho se propõe a modificar este mecanismo. Será utilizado o *redo journaling* que possui a vantagem de sempre armazenar no *journal* os metadados mais atualizados. Com esta técnica, novas melhorias podem ser feitas no sistema.

Neste trabalho introduzimos uma técnica baseada em ponteiros com o objetivo de reduzir a quantidade de barreiras de escrita (Figura 17). Em caso de necessidade de utilizar *journaling*, os passos realizados por nosso sistema de arquivos são:

Uma nova transação precisa ser alocada. Para isto será verificado se possui espaço para adicionar um novo ponteiro ou se possui espaço para adicionar a quantidade de entradas de log necessária. Caso não possua, deve-se iniciar o processo de limpeza do *journal*. Isto tudo corre de forma mutualmente exclusiva, ou seja, o processo de verificação, limpeza e alocação é realizada de forma única e serializada.

Como a região crítica não pode ser acessada de forma paralela, a lógica para fazer o gerenciamento deve ser pequena e com baixa latência. Para obter o máximo de desempenho há apenas um contador global com a quantidade de entradas de *log* e a quantidade de ponteiros livres. A cada nova transação estes contadores são decrementados, além de ser designado o ponteiro da transação.

O ponteiro (Transaction pointer) é composto de três informações: os primeiros 8 bits serão para o *generation id*, os próximos 24 bits serão para a quantidade de entradas de *log* presente na transação, e os últimos 32 para o *offset* de onde está o início das entradas de *log*.

Logo o ponteiro contém todas as informações necessárias para a transação. Com o espaço alocado no *journal*, os metadados podem ser escritos nas entradas de *log*. Depois de terminado, é realizada uma barreira de escrita e então é escrito o ponteiro da transação. A barreira garante que o ponteiro será escrito após todas as entradas de *log*. O ponteiro age como um *commit* da transação e, como sempre é escrito após todas as entradas de *log*, não há possibilidade de apontar para uma transação incompleta.

Nesta primeira implementação chamada de *oneflush,* o objetivo era minimizar a quantidade de barreiras de escrita. Para isto deixamos a escrita dos metadados apenas no *journal* e a atualização dos metadados é adiada. A atualização dos metadados na memória só ocorrerá quando o *journal* estiver cheio ou quando desmontar o sistema de arquivos, o que acaba gerando a limpeza do *journal*. O processo de limpeza copia todos os dados do *journal* para a posição real dos metadados na NVM, realiza uma barreira de escrita, apaga todos os ponteiros e realiza outra barreira de escrita. A vantagem desta técnica é que cada transação necessita apenas de uma barreira de escrita e apenas quando estiver lotado o *journal* será realizada a limpeza com mais duas barreiras de escrita. A desvantagem é que para requisições de leitura, precisaria ser verificado se o metadado não está presente no *journal* e então ser lido de sua real localização na NVM. Esta implementação pode se assemelhar a técnica *log-structured*, onde ocorre uma buferização das escritas, porém o foco dela é reduzir a quantidade de barreiras de escrita, mas ainda sofrerá com a “escrita duplicada” do que técnicas de *journaling* geram e a área de *log* estará restrita ao tamanho do *journal*. A Figura 18 apresenta a solução para o *oneflush* em formato de algoritmo.

```
Início
  Se journal não cheio então
    Copia dados para o journal
    Barreira de escrita
    Escreve ponteiro
  Senão
    Copia todos dados do journal para suas posições
    Barreira de escrita
    Incrementa generation id (apagar ponteiros)
    Barreira de escrita
Fim
```

Figura 18. Algoritmo para o *oneflush*.

Uma segunda implementação chamada de *manualflush,* foi desenvolvida para ser comparada com a *oneflush.* Ela funciona de forma similar com a *oneflush*, exceto que após a escrita dos metadados no *journal*, barreira de escrita, escrita do ponteiro da transação, uma segunda barreira de escrita será realizada e então atualizado os metadados na sua real localização na NVM. E então o ponteiro da transação pode ser apagado e uma barreira de escrita posterior foi opcionalmente não realizada. A barreira de escrita que foi opcionalmente omitida não é necessária, pois, no pior caso, o ponteiro da transação não será apagado e a atualização do metadado será considerada como não realizada e feita

novamente durante a montagem do sistema de arquivo. Isto é algo que pode ocorrer na última transação realizada pelo sistema de arquivos, já que barreiras de escrita realizadas em novas transações acabam servindo como as barreiras omitidas nas transações anteriores. Já a segunda barreira de escrita é necessária, pois é necessário garantir que o ponteiro da transação seja escrito antes da atualização dos metadados. Desta forma, caso ocorra uma falha do sistema, como o ponteiro da transação foi escrito antes da atualização dos metadados, a recuperação poderá ocorrer. A

Figura 19 apresenta a solução para o *manualflush* em formato de algoritmo.

```
Início
  Se journal não cheio então
    Copia dados para o journal
    Barreira de escrita
    Escreve ponteiro
    Barreira de escrita
    Copia dados da transação no journal para suas posições
    Apaga ponteiro
  Senão
    Espera transações pendentes terminarem
    Barreira de escrita
    Incrementa generation id (apagar ponteiros)
    Barreira de escrita
Fim
```

Figura 19. Algoritmo para *manualflush*.

Para estas implementações foi construído um modelo simulado no PMFS. A estrutura com ponteiro foi implementada, porém o *undo-journaling* permaneceu. Já que o modelo proposto necessita do *redo-journaling*, assume-se que o desempenho do *redo-journaling* é igual ao do *undo-journaling*. Para gerar a mesma quantidade de cópias, é forçado cópias do metadado para o *journal*, ao invés de ser do *journal* para o metadado.

4. AVALIAÇÃO

As simulações feitas foram realizadas em um computador usando Linux kernel 4.10 com Intel Core i5 com 8GB de DRAM, onde 4GB foram reservadas para emular a NVM e no Laboratório de Alto Desempenho (LAD) da PUCRS com um computador Intel Xeon Gold 5118 – Skylake com 32GB de DRAM, onde 8GB foram reservados para emular a NVM. A forma de emular a NVM foi utilizar o *block device* PMEM presente no kernel, que reserva parte da DRAM de forma contígua, da mesma forma que [19][29][49]. Também foi colocado um atraso adicional nas instruções de *clflush* (de forma semelhante feita em [29] para memórias do tipo STT-RAM) já que estas memórias possuem uma latencia maior que a DRAM.

Todas as simulações foram realizadas em três variações do PMFS (no PMFS original, na solução *oneflush* e *manualflush*). Como o PMFS foi originalmente projetado para a versão de kernel 3.9, algums adaptações foram necessárias para funcionar na versão 4.10.

Para a medir o desempenho, foram utilizados os *benchmarks* FIO [17] e FILEBENCH [51]. Estes benchmarks também foram usados em trabalhos relacionados como [29][49]. FIO (flexible IO tester) é uma ferramenta que permite criar diversas tarefas realizando operações de IO conforme configurado pelo usuário. Já o FILEBENCH é um *benchmark* também flexível onde é possível criar cargas de trabalho ou utilizar algumas pré existentes da ferramenta.

A simulação utilizando FIO consiste em realizar diversas escritas consecutivas durante cinco minutos com o tamanho de bloco designado em um único arquivo. As escritas realizadas eram de tal forma que sempre necessitavam realizar *journaling*, configurando-se um cenário de pior caso que poderia ocorrer. As simulações realizadas com o FILEBENCH foram utilizando algumas de suas cargas de trabalho nativas da ferramenta: servidor de arquivos, servidor de e-mail, servidor de webproxy e servidor web.

Todas as simulações foram executadas dez vezes nas duas máquinas com um índice de confiança de 95% com uma margem de erro de 2% no pior caso. Nos gráficos apresentados, são exibidos os valores médios calculados.

4.1 Resultados com Intel Core i5

A Figura 20 mostra a primeira simulação realizada utilizando o *benchmark* FIO. Os resultados mostram que para blocos de escrita de até 256 *bytes* o *manualflush* obteve os melhores resultados (15% a mais que o PMFS original) enquanto, para blocos de escrita maiores, o *oneflush* se saiu melhor. Com tamanho de bloco de escrita de 1KB, o *manualflush* teve 17% e o oneflush 14% de melhoria em relação ao PMFS original.

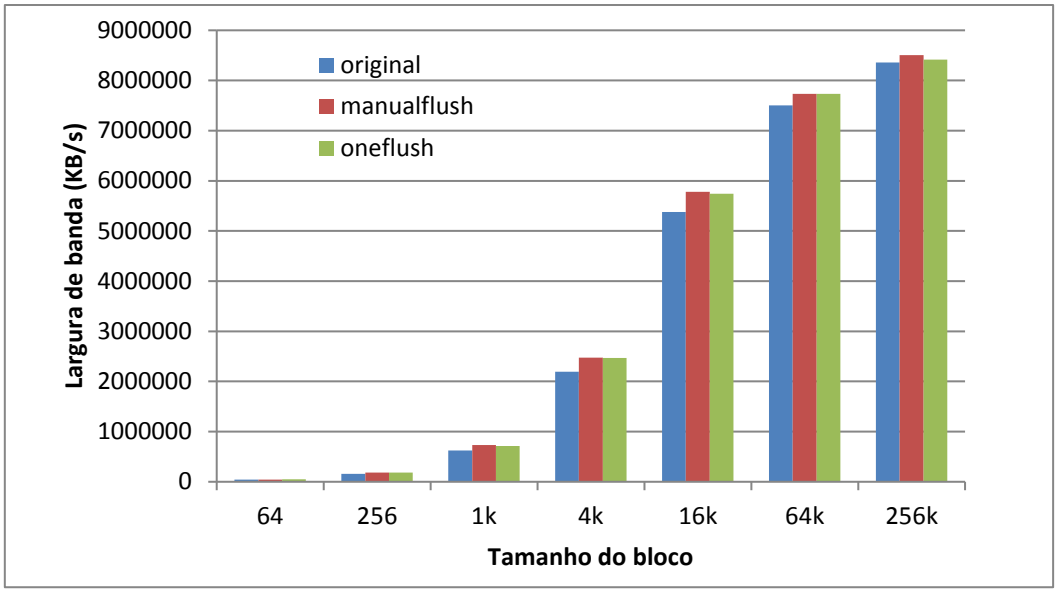


Figura 20. Resultados usando o benchmark FIO. Comparação da largura de banda (em KB/s) entre as implementações com diversos tamanhos de blocos de escrita.

Já na Figura 21 é exibido os resultados obtidos com o benchmark FILEBENCH. Obteve-se um ganho de 5% utilizando a solução *manualflush* para a carga de trabalho *varmail*, enquanto para as demais cargas de trabalho obteve-se resultados muito próximos (diferença menor que 1%) em relação ao PMFS original. Para o *oneflush*, houve uma redução próxima de 9% no *varmail*, e em torno de 3% para as demais cargas de trabalho em relação ao original.

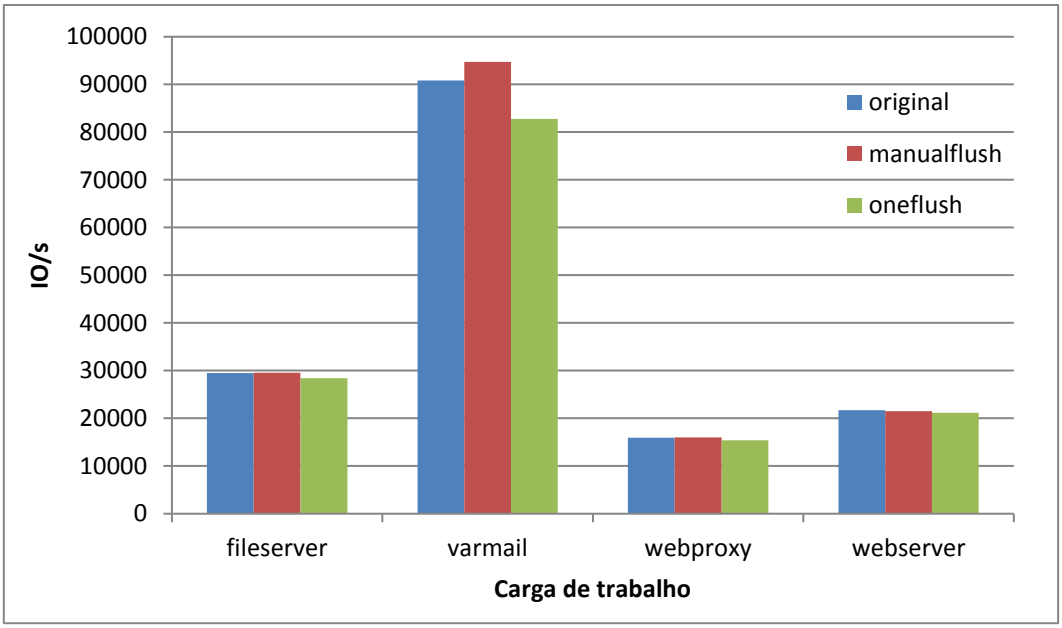


Figura 21. Resultados usando o benchmarks do FILEBENCH. Comparação de IOPS de escrita entre as implementações.

Na Figura 22 temos a quantidade de barreiras de escrita e na Figura 23 o tempo total gasto por elas. Como esperado, as implementações propostas reduziram a quantidade de barreiras de escrita e consequentemente o tempo

gasto por elas. Entretanto reduzir a quantidade de barreiras de escrita não se mostrou ser o fator mais impactante no desempenho, já que *oneflush* mesmo com menos barreiras de escrita obteve os piores resultados com o *benchmark* FILEBENCH.

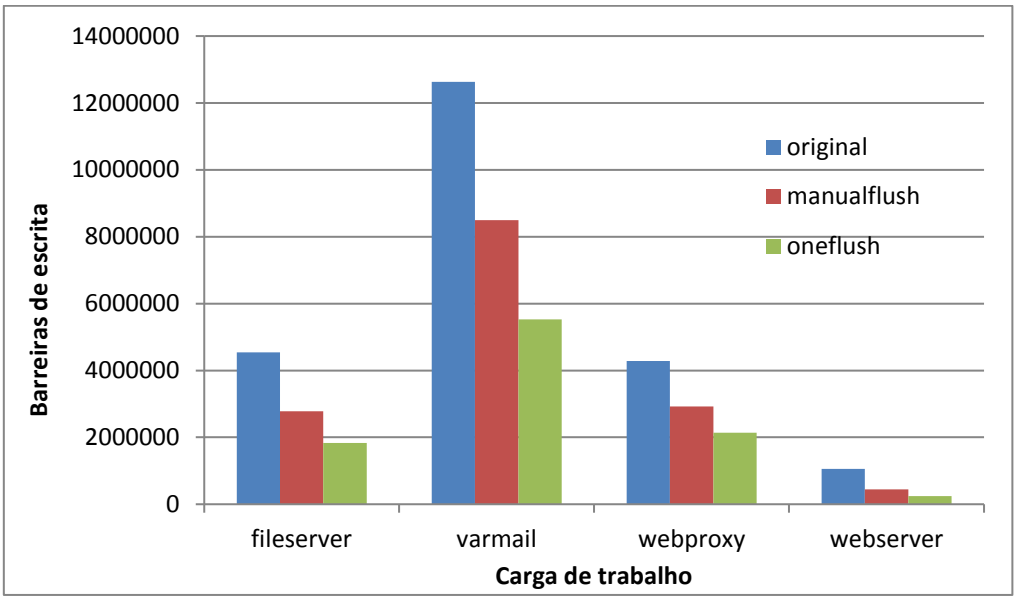


Figura 22. Comparação da quantidade de barreiras de escrita entre as implementações.

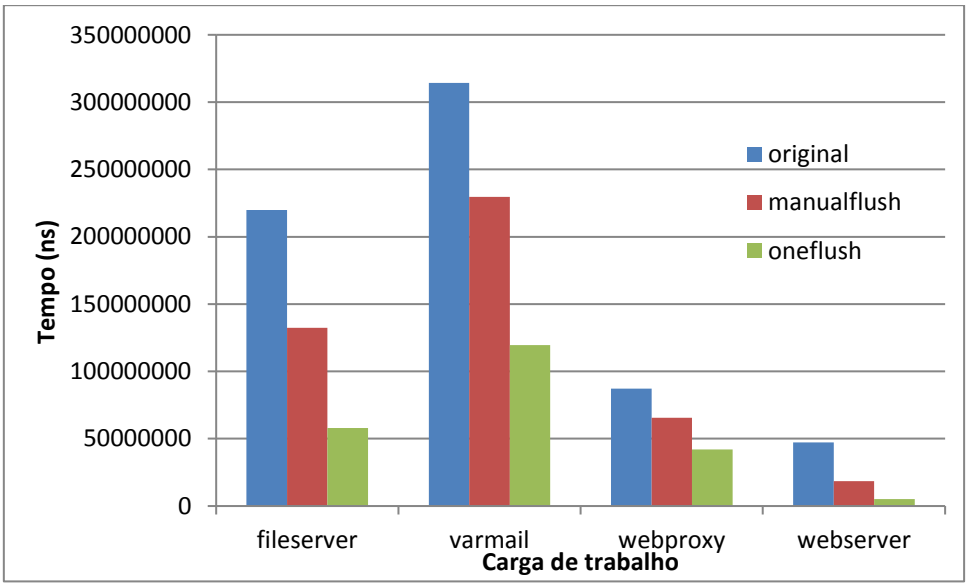


Figura 23. Comparação do tempo gasto (em nanosegundos) nas barreiras de escrita.

Ao investigar a razão deste comportamente foi constatado que isto ocorre devido ao tempo gasto na seção crítica. Na Figura 24 é mostrado o tempo total gasto na seção crítica em cada uma das implementações. No *manualflush*, cada operação que necessita de *journaling* realiza duas barreiras de escrita. Já no *oneflush*, cada operação realiza apenas uma barreira. Analisando isoladamente, o *oneflush* consegue uma latência menor para as operações de escrita. Porém os metadados não ficam atualizados no *oneflush*, já que permanem no *journal*

até ser feito a limpeza. Quando uma operação necessita utilizar o journal e não possui espaço sobrando, é necessário realizar a limpeza do *journal*. Como a operação de limpeza é custosa em tempo, as diversas outras operações (*threads*) que também necessitam do *journal* ficam aguardando a operação de limpeza terminar. Já no *manualflush*, cada operação apaga seu registro no *journal,* não necessitando de uma grande operação de limpeza que bloqueia as demais *threads* por um longo período. Como no *benchmark* FILEBENCH é utilizado centenas de *threads* realizando operações de IO, esta diferença de desempenho se mostrou mais evidente.

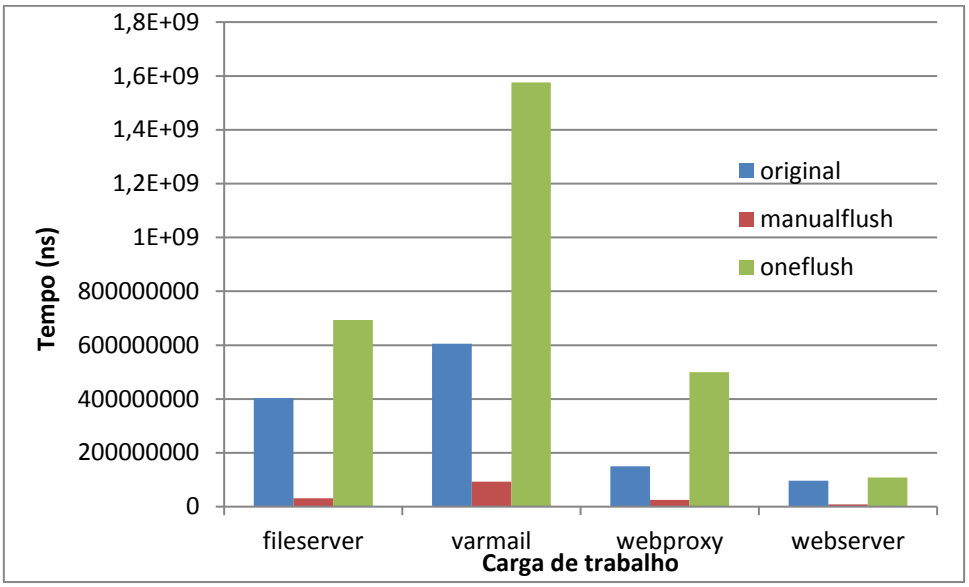


Figura 24. Comparação do tempo gasto (em nanosegundos) em sessão crítica.

4.2 Resultados com Intel Xeon Gold

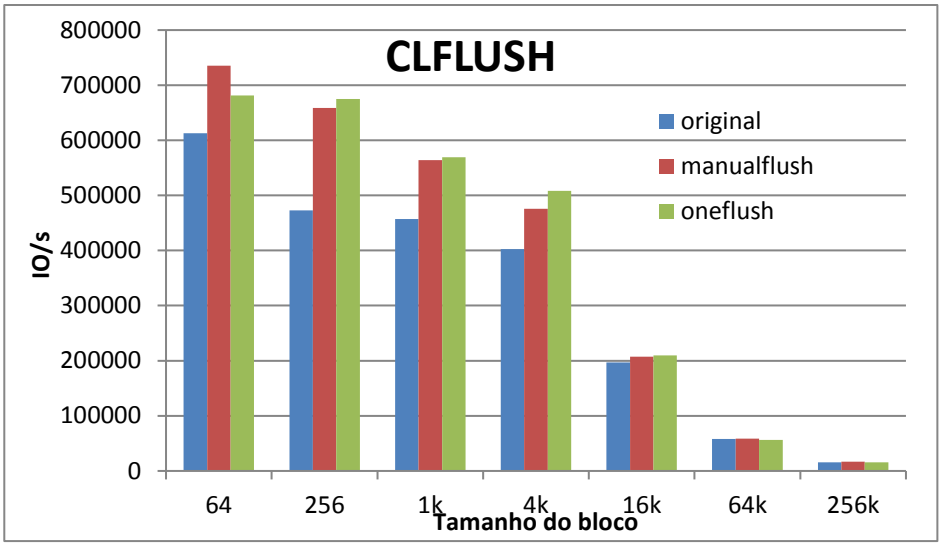
Os mesmos testes foram realizados em computador com processador Intel Xeon Gold, microarquitetura Skylake, que suporta novas instruções em nível de cache como *clflushopt* e *clwb*. Na Figura 25 mostra os resultados obtidos utilizando o benchmark FIO usando as diferentes instruções de cache. A implementação *manualflush* obteve em todos os testes desempenho superior ao PMFS original chegando em 39% de melhoria usando *clflush*, 48% usando *clflushopt* e 19% usando *clwb* usando um tamanho de bloco de 256 *bytes*. Já a implementação *oneflush* obteve melhora em quase todos os casos ao se comparar com o PMFS original, obtendo uma melhoria de 42% com *clflush*, 49% com *clflushopt* e 32% com *clwb* usando um tamanho de bloco de 256 *bytes*.

Utilizar a instrução *clflushopt* obteve uma melhora de desempenho de 7% para tamanho de bloco de 256 e 1k *bytes*, e este ganho chega a 11% para tamanho de bloco de 16k na implementação *manualflush*. Utilizar tamanho de bloco pequeno (256 *bytes* e menor) possui uma granularidade muito fina e gera uma quantidade de IOs acima do máximo suportado pela arquitetura, não

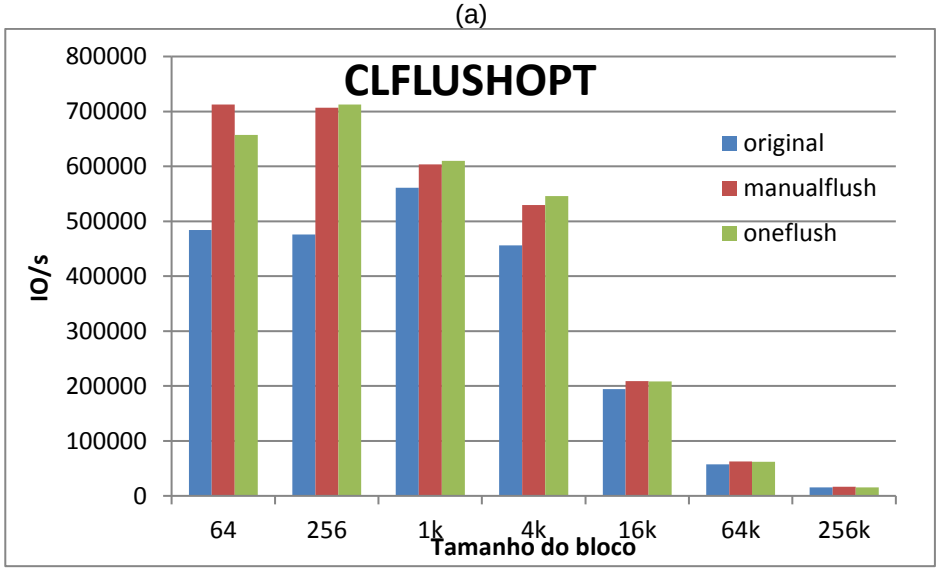
conseguindo acompanhar uma melhora significativa no desempenho ao ir reduzindo o tamanho no bloco. Ao utilizar tamanho de bloco superior a 16k *bytes* acabou não impactando muito significativamente no desempenho devido a uma baixa quantidade de IOs gerada pelo tráfego.

Já utilizando a instrução *clwb*, houve uma piora nos resultados a partir de 256 *bytes*. Como há muitas escritas no *journal* e elas são apenas usadas como área temporária, não existe a necessidade de manter estes dados ativos ainda na cache, fato que não ocorre ao utilizar as instruções *clflush* e *clflushopt*, e manter estes dados não reutilizados acaba reduzindo o desempenho.

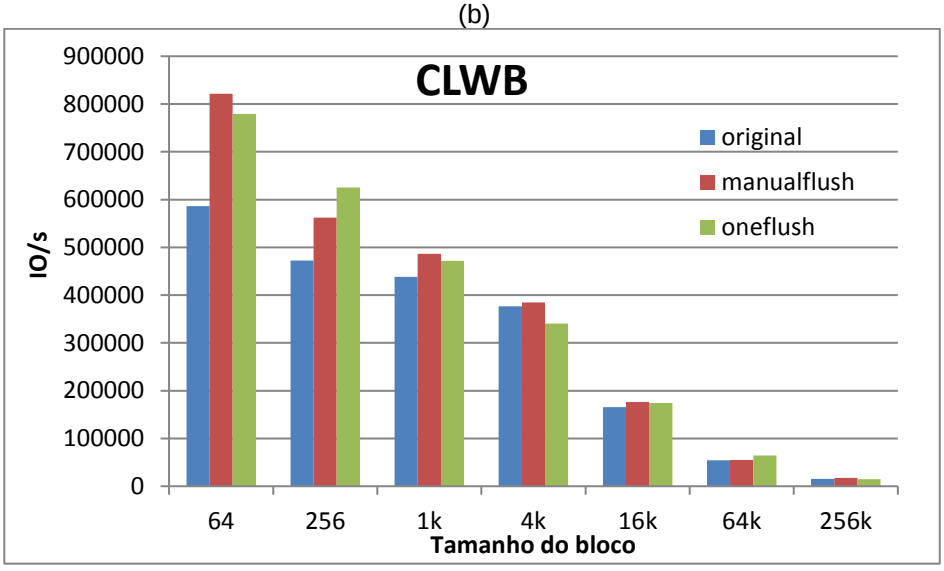
Na Figura 26 temos os resultados obtidos ao utilizar o benchmark FILEBENCH. Houve uma redução de 2% no desempenho com a carga de trabalho *fileserver*, 1% com a carga de trabalho *webproxy* e um aumento de 1% com a carga *varmail*. Estes resultados são explicados analisando o elevado tempo que as barreiras de escrita levaram (Figura 27) devido a uma grande quantidade de núcleos realizarem escritas nas caches em seus diversos níveis e com custo para manter a coerência entre elas junto com um grande tempo gasto nas sessões críticas (Figura 28). Devido a esta saturação, utilizar as instruções *clflushopt* e *clwb* não agregou no resultado final de desempenho do sistema de arquivos.



(a)

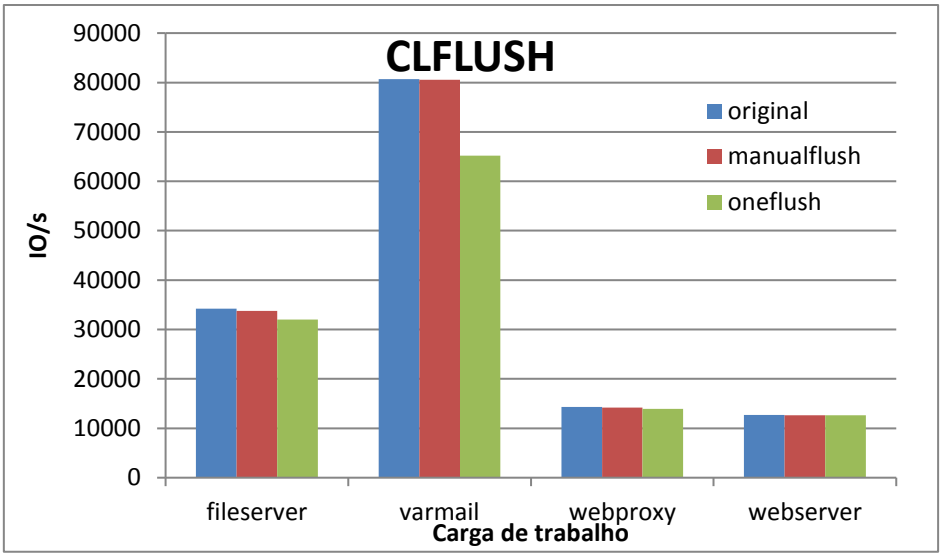


(b)

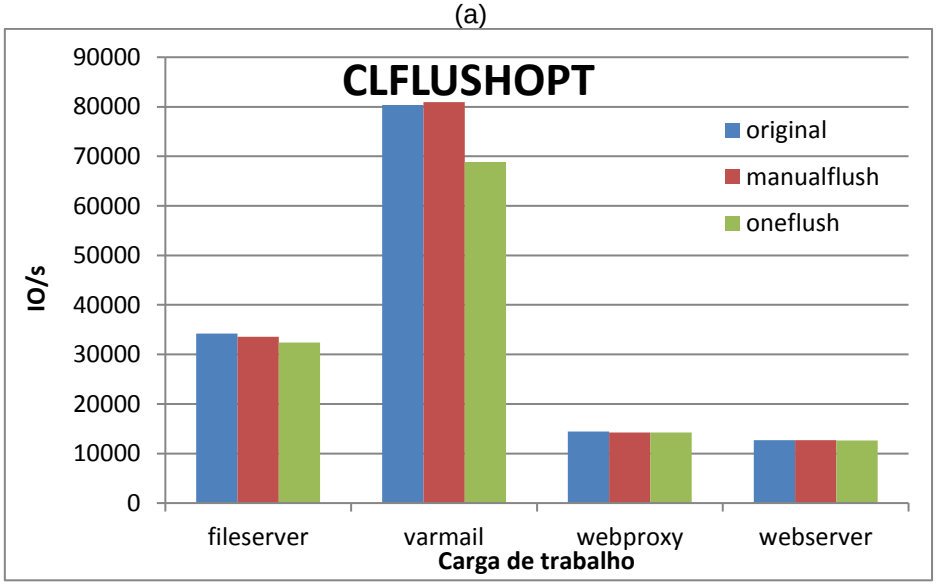


(c)

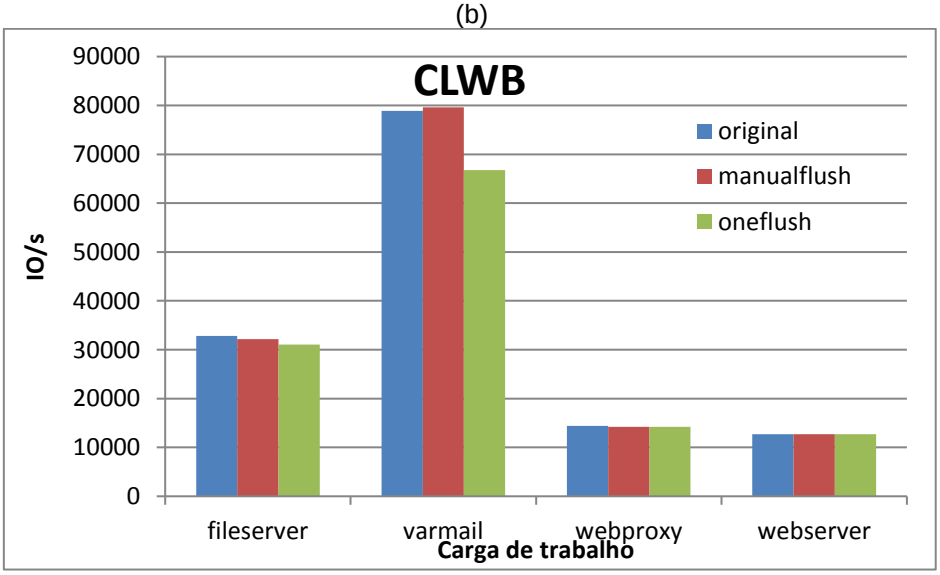
Figura 25. Resultados usando o benchmark FIO. Comparação de IO/s entre as implementações com diversos tamanhos de blocos de escrita. (a) utilizando a instrução *clflush*; (b) utilizando a instrução *clflushopt*; (c) utilizando a instrução *clwb*.



(a)

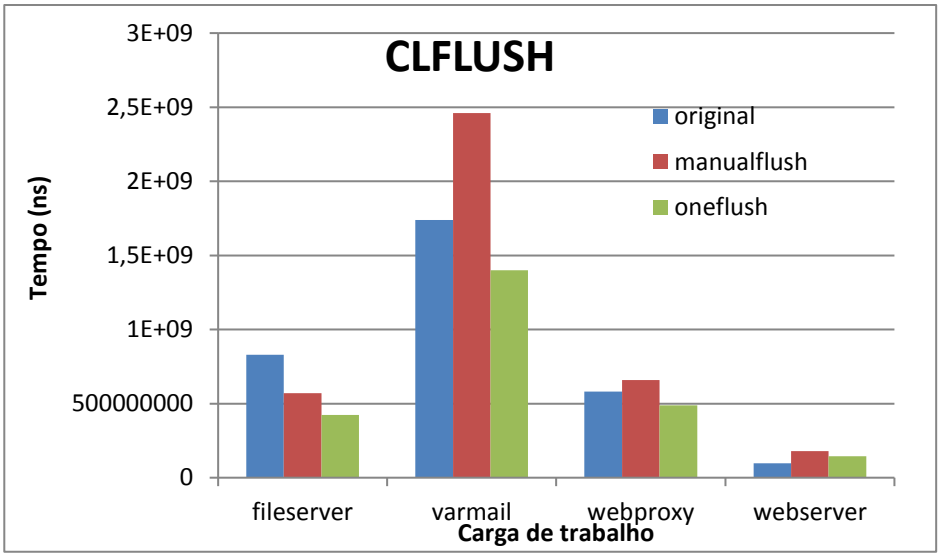


(b)

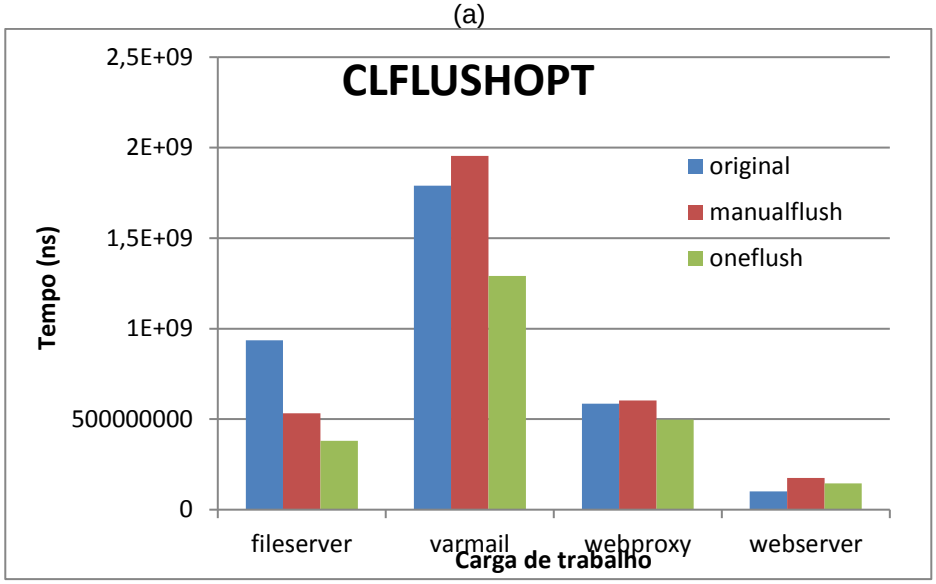


(c)

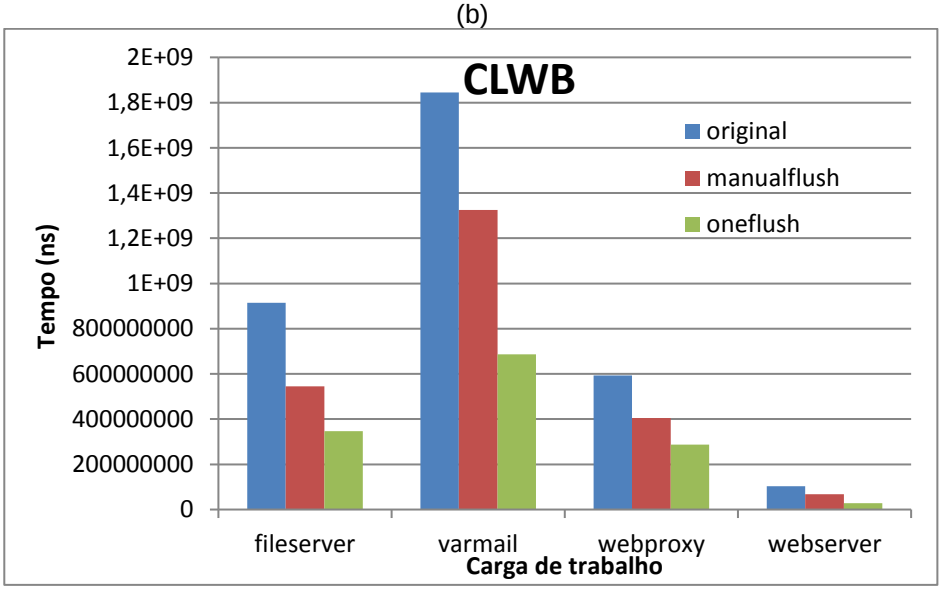
Figura 26. Resultados usando o benchmarks do FILEBENCH. Comparação de IOPS de escrita entre as implementações. (a) utilizando a instrução *clflush*; (b) utilizando a instrução *clflushopt*; (c) utilizando a instrução *clwb*.



(a)

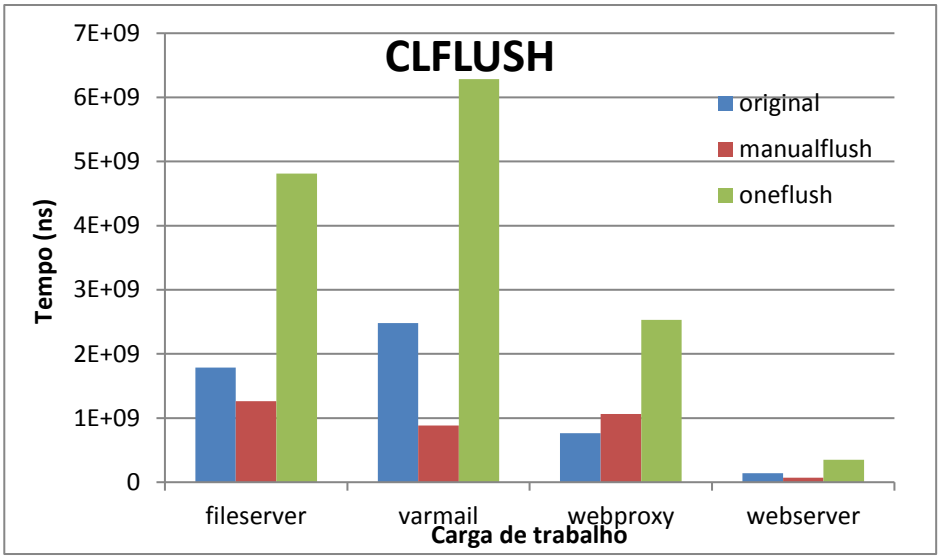


(b)

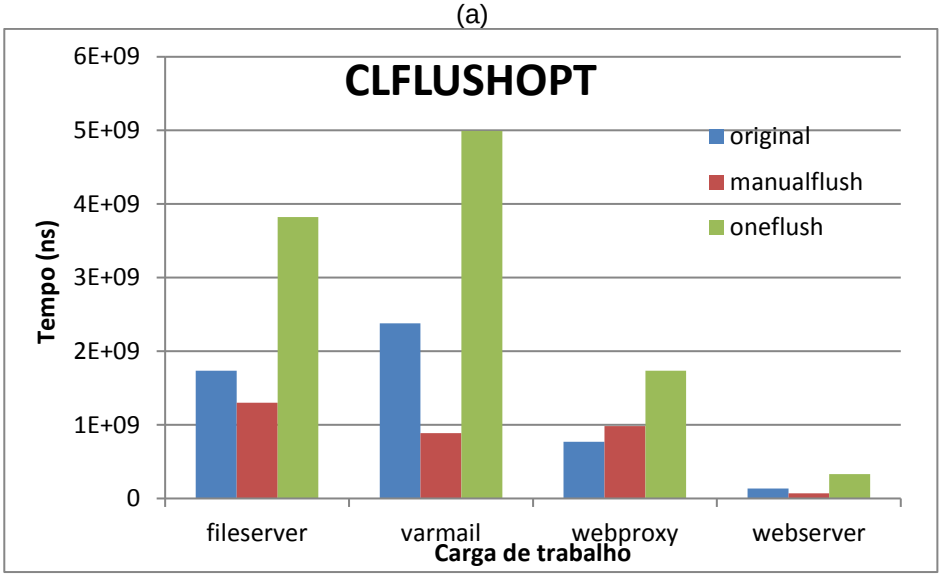


(c)

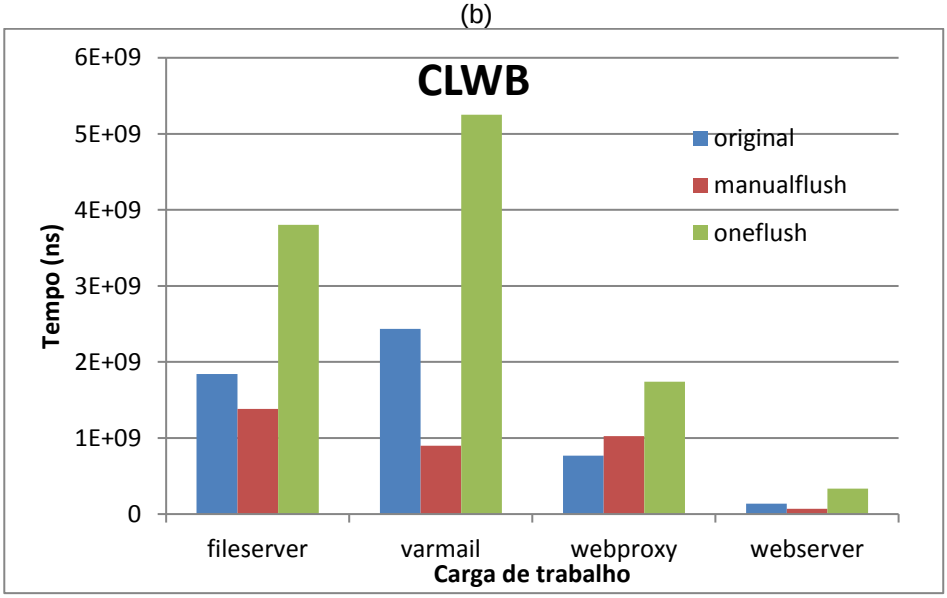
Figura 27. Comparação do tempo gasto (em nanosegundos) nas barreiras de escrita. (a) utilizando a instrução *clflush*; (b) utilizando a instrução *clflushopt*; (c) utilizando a instrução *clwb*.



(a)



(b)



(c)

Figura 28. Comparação do tempo gasto (em nanosegundos) em sessão crítica. (a) utilizando a instrução *clflush*; (b) utilizando a instrução *clflushopt*; (c) utilizando a instrução *clwb*.

5. CONCLUSÃO

Esta dissertação apresentou uma visão geral sobre problemas e soluções para as emergentes novas tecnologias de memórias não voláteis. Também apresentou duas implementações chamadas *oneflush* e *manualflush* sobre o sistema de arquivos PMFS.

Apesar de uma implementação teórica que degradaria o desempenho de requisições de leitura, o *oneflush* também se demostrou inferior às demais implementações na maioria das simulações. Já a implementação *manualflush* conseguiu ganhos consideráveis em alguns casos, sendo uma melhoria viável de ser aplicada ao PMFS.

Embora estes ganhos nem sempre são visíveis, como em algumas cargas de trabalho usando o *benchmark* FILEBENCH, a solução *manualflush* não chegou a impactar muito negativamente o desempenho do sistema de arquivos. Como é realizado *journaling* apenas para atualização de metadados, se torna difícil notar alguma diferença no desempenho em grande parte de aplicações comuns, visto que atualização em metadados que necessitam realizar *journaling* é uma porção muito pequena em relação à quantidade de largura de banda total utilizada na NVM. Para aplicações específicas que precisam realizar operações que necessitam constantemente atualizar metadados através da técnica de *journaling*, o ganho de desempenho pode chegar até 49%, como mostrado pelo *benchmark* FIO.

No benchmark FIO, além de necessitar constantemente realizar *journaling*, as operações eram serializadas. Já no benchmark FILEBENCH há diversas tarefas concorrentes, cada uma realizando barreiras de escrita e *flush* na cache. Esta disputa por recursos compartilhados acaba degradando o desempenho do sistema. Logo, as soluções propostas não mostram ganhos consideráveis para aplicações altamente concorrentes.

Utilizar a instrução *clflushopt* se mostrou de maneira geral que a instrução *clflush* e *clwb*. No benchmark FIO os ganhos são mais expressivos, enquanto no benchmark FILEBENCH os ganhos por utilizar a instrução são mínimos, devido ao cenário com grande concorrente. A instrução *clflushopt* é uma otimização da instrução *clflush* com relaxamento de ordenamento e não serializada o pipeline de execução da CPU, obtendo maior desempenho.

Por fim, conclui-se que utilizar uma lógica simples na seção crítica para torná-la menor pode trazer um maior desempenho ao sistema do que reduzir a quantidade de barreiras de escrita em si. Reduzir a quantidade de barreiras de escrita não impacta muito significativamente um sistema muito concorrente. Esta conclusão é similar a trabalhos relacionados, como em [53], que também reduziu a quantidade de *flushes* e restrições de escrita em um mecanismo de *journaling* para banco de dados SQLite e o ganhos obtidos variaram de 0,8% a 4,6%.

Um aspecto que poderia ainda ser investigado é relacionado a quais casos seria mais vantajoso usar a instrução *clwb* ou a *clflushopt*. Outro fator importante a se investigar é tentar reduzir a quantidade de barreiras de escrita geradas por

requisições de leitura, já que elas acabam gerando a necessidade de atualizar o *access time* do *inode*.

REFERÊNCIAS

[1] Akel, A.; Caulfield, A. M.; Mollov, T. I.; Gupta, R. K.; Swanson, S. “Onyx: A Prototype phase change memory storage array”. In: Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems, 2011, pp. 2-2.

[2] Kolli, A.; Pelley, S., Saidi, A.; Chen, P. M.; Wenisch, T. F. “High-Performance Transactions for Persistent Memories”. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, 2016, pp. 399-411.

[3] Caulfield, A. M.; De, A.; Coburn, J.; Mollov, T. I.; Gupta, R. K.; Swanson, S. “Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories”. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, 2010, pp. 385-395.

[4] Caulfield, A. M.; Mollov, T. I.; Eisner, L. A.; De, A.; Coburn, J.; Swanson, S. “Providing Safe, User Space Access to Fast, Solid State Disks”. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2012, pp. 387-400.

[5] A. S. Tanenbaum. “Sistemas Operacionais Modernos”. Prentice Hall Brasil, 2009, 672p.

[6] Lee, B. C.; Ipek, E.; Mutlu, O.; Burger, D. “Architecting Phase Change Memory As a Scalable Dram Alternative”. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 2-13.

[7] Feng B.; Liu, N.; He, S.; Sun, X. “HPIS3: Towards a High-performance Simulator for Hybrid Parallel I/O and Storage Systems”. In: Proceedings of the 9th Parallel Data Storage Workshop, 2014, pp. 37-42.

[8] Xue, C. J.; Zhang, Y.; Chen, Y.; Sun, G.; Yang, J. J.; Li, H. “Emerging Non-volatile Memories: Opportunities and Challenges”. In: Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2011, pp. 325-334.

[9] Wu, C.-H.; Chang, W.-Y.; Hong, Z.-W. “A Reliable Non-volatile Memory System: Exploiting File-System Characteristics”. In: 15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009, pp 202-207.

[10] Hitz, D.; Lau, J.; Malcolm, M. “File System Design for an NFS File Server Appliance”. In: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, 1994, pp. 19-19.

[11] Vučinić, D.; Wang, Q.; Guyot, C.; Mateescu, R.; Blagojević, F., Franca-Neto L.; Moal, D. L.; Bunker, T.; Xu, J.; Swanson, S.; Bandić, Z. “DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express”. In: Proceedings of the 12th USENIX Conference on File and Storage Technologies, 2014, pp. 309-315.

[12] D. P. Bovet, M. Cesati. “Understanding the Linux Kernel”, O'Reilly Media, 2005, 944p.

[13] Intel Corporation, “Deprecating the PCOMMIT Instruction”. Recuperado de: https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, Maio 2017.

[14] Budilovsky, E.; Toledo, S., “Kernel Based Mechanisms for High Performance I/O”, Dissertação de Mestrado, Tel Aviv University, 2013, 48p.

[15] Lee, E.; Bahn, H.; Noh, S. H. “A Unified Buffer Cache Architecture That Subsumes Journaling Functionality via Nonvolatile Memory”. *ACM Transactions on Storage (TOS)*, vol. 10, Janeiro 2014, pp 1-17.

[16] Xia, F.; Jiang, D.-J.; Xiong, J.; Sun, N.-H. “A Survey of Phase Change Memory Systems”. *Journal of Computer Science and Technology*, vol. 30-1, Janeiro 2015, pp. 121-144.

[17] A. Carroll, J. Axboe. “fio - flexible I/O tester”. Recuperado de: https://linux.die.net/man/1/fio, Maio 2017.

[18] Dhiman, G.; Ayoub, R.; Rosing, T. “PDRAM: A Hybrid PRAM and DRAM Main Memory System”. In: Proceedings of the 46th Annual Design Automation Conference, 2009, pp. 664-469.

[19] Puglia, G.; Zorzo, A. F. “Exploring Atomicity on File Mappings Based on Non-Volatile Memory File systems”, Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2017, 105p.

[20] Volos, H.; Tack, A. J.; Swift, M. M. “Mnemosyne: Lightweight Persistent Memory”. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 91-104.

[21] Volos. H.; Nalli, S.; Panneerselvam, S.; Varadarajan, V.; Saxena, P.; Swift, M. M. “Aerie: Flexible File-system Interfaces to Storage-class Memory”. In: Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 1-14.

[22] Intel Corporation, “Intel and Micron Produce Breakthrough Memory Technology”. Recuperado de: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, Maio 2017.

[23] Intel Corporation, “Intel® 64 and IA-32 Architectures Software Developer’s Manual”. Recuperado de: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, Maio 2017.

[24] Intel Corporation. “Intel® Architecture Instruction Set Extensions Programming Reference”. Recuperado de: https://software.intel.com/sites/default/files/managed/26/40/319433-026.pdf, Maio 2017.

[25] Coburn, J.; Caulfield, A. M.; Akel, A.; Grupp, L. M.; Gupta, R. K.; Jhala, R.; Swanson, S. “NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories”. In: Proceedings of the Sixteenth

International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 105-118.

[26] Condit, J.; Nightingale, E. B.; Frost, C.; Ipek, E.; Lee, B.; Burger, D.; Coetzee, D. “Better I/O Through Byte-addressable, Persistent Memory”. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, 2009, pp. 133-146.

[27] J. Corbet, “LFCS: Preparing Linux for nonvolatile memory devices”. Recuperado de: https://lwn.net/Articles/547903/,Maio 2017.

[28] Dong, J.; Zhang, L.; Han, Y.; Wang, Y.; Li, X. “Wear Rate Leveling: Lifetime Enhancement of PRAM with Endurance Variation”. In: Proceedings of the 48th Design Automation Conference, 2011, pp. 972-977.

[29] Xu, J.; Swanson, S. “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories”, In: Proceedings of the 14th Usenix Conference on File and Storage Technologies, 2016, pp. 323-338.

[30] Zhao, J.; Mutlu, O.; Xie, Y. “FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems”. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014, pp. 153-165.

[31] Zhao, J.; Li, S.; Yoon, D. H.; Xie, Y.; Jouppi, N. P., “Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support”. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, 2013, pp. 421-432.

[32] K. Bhandari, D. R. Chakrabarti, H.-J. Boehm. “Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming”. Recuperado de: http://www.hpl.hp.com/techreports/2012/HPL-2012-236.pdf, Maio 2017.

[33] Sun, L.; Lu, Y.; Shu, J., “DP2: Reducing Transaction Overhead with Differential and Dual Persistency in Persistent Memory”. In: Proceedings of the 12th ACM International Conference on Computing Frontiers, 2015, pp. 1-8.

[34] Bjørling, M.; Axboe, J.; Nellans, D.; Bonnet, P. “Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems”. In: Proceedings of the 6th International Systems and Storage Conference, 2013, pp. 1-10.

[35] Qureshi, M. K.; Srinivasan, V.; Rivers, J. A. “Scalable High Performance Main Memory System Using Phase-change Memory Technology. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 24-33.

[36] M. Wilcox, “Add support for NV-DIMMs to ext4”. Recuperado de: https://lwn.net/Articles/613384/, Maio 2017.

[37] Wu, M.; Zwaenepoel, W.; “eNVy: A Non-volatile, Main Memory Storage System”, Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1994, pp. 86-97.

[38] Autor Desconhecido. “Non-Volatile Memory Library”. Recuperado de: https://github.com/pmem/nvml/, Maio 2017.

[39] Oracle Corporation, “Oracle Solaris ZFS File System”. Recuperado de: https://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/, Maio 2017.

[40] P. Clarke. “Intel, Micron Launch ‘Bulk-Switching’ ReRAM”. Recuperado de: https://www.eetimes.com/document.asp?doc_id=1327289, Maio 2017.

[41] Olivier, P.; Boukhobza, J.; Senn, E. “Micro-benchmarking Flash Memory File-System Wear Leveling and Garbage Collection: A Focus on Initial State Impact”. In: 15th IEEE International Conference on Computational Science and Engineering, 2012, pp. 437-444.

[42] Sehgal, P.; Basu, S.; Srinivasan, K., Voruganti, K. “An empirical study of file systems on NVM”. In: 31st Symposium on Mass Storage Systems and Technologies (MSST), 2015, pp. 1-14.

[43] Zhou, P.; Zhao, B.; Yang, J.; Zhang, Y. “A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology”. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, 2009, pp. 14-23.

[44] Fackenthal, R.; Kitagawa, M.; Otsuka, W.; Prall, K.; Mills, D.; Tsutsui, K.; Javanifard, J.; Tedrow, K.; Tsushima, T.; Shibahara, Y.; Hush, G. “A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology”. In: IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014, pp. 338-339.

[45] R. H. Arpaci.-Dusseau, A. C. Arpaci.-Dusseau, “Operating Systems: Three Easy Pieces”, Arpaci-Dusseau Books, 2015, 675p.

[46] Williams, R. S. “How We Found The Missing Memristor”, *IEEE Spectrum*, vol. 45-12, Dezembro 2018, pp. 28-35.

[47] Liu, R.-S.; Shen D.-Y.; Yang, C.-L.; Yu, S.-C.; Wang, C.-Y. M. “NVM Duet: Unified Working Memory and Persistent Store Architecture”. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014, pp. 455-470.

[48] Pelley, S.; Chen, P. M.; Wenisch, T. F. “Memory Persistency”. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture, 2014, pp. 265-276.

[49] Dulloor, S. R.; Kumar, S.; Keshavamurthy, A.; Lantz, P.; Reddy, D.; Sankaran, R.; Jackson, J. “System Software for Persistent Memory”. In: Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 1-15.

[50] Kawahara, T. “Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing”. *IEEE Design & Test of Computers*, vol. 28-1, Jan-Fev 2011, pp. 52-63.

[51] Tarasov, V.; Zadok, E.; Shepler, S. “Filebench: A flexible framework for file system benchmarking”. *;login: The USENIX Magazine*, vol. 41-1, 2016.

[52] Fischer, W.; Schonberger, G.; Krenn, T. “The Linux Stack Diagram”. Recuperado de: https://www.thomas-

krenn.com/de/wikiDE/images/e/e0/Linux-storage-stack-diagram_v4.10.png, Maio 2017.

[53] Kim, W.-H.; Kim, J.; Baek, W.; Nam, B.; Won, Y. “NVWAL: Exploiting NVRAM in Write-Ahead Logging”. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, 2016, pp. 385-398.

[54] Wu, X.; Reddy, A. L. N. “SCMFS: A File System for Storage Class Memory”. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1-11.

[55] Lu, Y.; Shu, J.; Sun, L. “Blurred Persistence: Efficient Transactions in Persistent Memory”. *ACM Transactions on Storage (TOS) - Special Issue on Massive Storage Systems and Technologies (MSST 2015)*, vol 12-1, Janeiro 2016, pp 1-29.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br