LEONARDO ANTÔNIO DOS SANTOS

AVALIAÇÃO DE MECANISMOS DE CACHING COM DISCOS DE ESTADO SÓLIDO EM ESTRUTURAS DE ARMAZENAMENTO SECUNDÁRIO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Carlos Alberto Maziero
Coorientador: Prof. Dr. Luis Carlos Erpen de Bona

CURITIBA

2015

LEONARDO ANTÔNIO DOS SANTOS

AVALIAÇÃO DE MECANISMOS DE CACHING COM DISCOS DE ESTADO SÓLIDO EM ESTRUTURAS DE ARMAZENAMENTO SECUNDÁRIO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Carlos Alberto Maziero
Coorientador: Prof. Dr. Luis Carlos Erpen de Bona

CURITIBA

2015

S237a Santos, Leonardo Antônio dos
Avaliação de mecanismos de caching com discos de estado sólido em estruturas de armazenamento secundário/ Leonardo Antônio dos Santos. – Curitiba, 2015.
72 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática, 2015.

Orientador: Carlos Alberto Maziero – Co-orientador: Luis Carlos Erpen de Bona.
Bibliografia: p. 68-72.

1. Armazenamento de dados. 2. Linux (Sistema operacional de computador). 3. Gerenciamento de memória (Computação). I. Universidade Federal do Paraná. II.Maziero, Carlos Alberto. III. Bona, Luis Carlos Erpen de . IV. Título.

CDD: 004.568

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Leonardo Antônio dos Santos, avaliamos o trabalho intitulado, "Avaliação de mecanismos de caching com discos de estado sólido em estruturas de armazenamento secundário", cuja defesa foi realizada no dia 31 de agosto de 2015, às 14:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela:
(X)**aprovação** do(a) candidato(a). ( )**reprovação** do(a) candidato(a).

Curitiba, 31 de agosto de 2015.

Prof. Dr. Carlos Alberto Maziero
PPGInf - Orientador

Prof. Dr. Luis Carlos Erpen de Bona
PPGInf – Co-orientador

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

Prof. Dr. Douglas Dyllon Jeronimo de Macedo
IFSC – Membro Externo

Prof. Dr. Carlos Alberto Martins de Carvalho
UFPR – Membro Externo

AGRADECIMENTOS

Agradeço primeiramente a Deus por ser o meu sustento e conforto presente em qualquer momento de dificuldade. Aquele que me acompanha desde sempre e é a principal razão da minha existência. Sem Ele este trabalho não seria possível.

Agradeço à minha esposa Vania pela compreensão nos momentos em que tive que me dedicar aos estudos e dessa forma abrir mão de momentos preciosos na nossa vida. Obrigado por ser compreensiva e companheira, por ter suportado cada dia sem a minha presença.

Agradeço aos meus pais por serem aqueles que me projetaram, não só quanto ao meu nascimento, mas quanto o homem que me tornei. Obrigado pelo exemplo que sempre foram.

Agradeço ao amigo Pedro que sempre foi um apoio ao ingresso no mestrado e alguém que sempre esteve me incentivando. Da mesma forma agradeço aos colegas do TRE/PR e TRT9 que sempre estiveram na torcida.

Agradeço ao JONG (Jovens da Nova Geração), pela força que sempre me deram e por suportarem a minha ausência física e intelectual. Por me aguentarem como líder mesmo sabendo que às vezes eu precisava gastar mais energia no mestrado do que com vocês. Agradeço a cada um com quem não pude sair para conversar ou por projetos que não pude dar continuidade. Agora me aguentem!

Por fim, agradeço aos professores Maziero e Bona pelo apoio desde o primeiro dia de mestrado. Ao professor Maziero por confiar no meu trabalho e potencial mesmo antes do mestrado começar, quando ainda era professor da UTFPR. Ao professor Bona pelos conselhos, correções e por confiar o seu tempo na [c]orientação. Levarei os conselhos e ensinos nas nossas conversas durante a minha carreira profissional e vida acadêmica.

Agradeço aos brasileiros que patrocinaram tudo de forma direta ou indireta. Tentarei ao máximo esforço retribui-los.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Recentemente, os discos de estado sólido (SSDs – *Solid State Disks*) elevaram muito o desempenho no acesso ao armazenamento secundário. Contudo, seu custo e baixa capacidade inviabilizam a substituição integral dos discos rígidos (HDDs – *Hard Disk Drives*) por SSDs a curto prazo, sobretudo em instalações de maior porte. Por outro lado, é possível aliar o desempenho dos SSDs à capacidade e baixo custo dos HDDs, usando SSDs como cache dos HDDs para os dados mais acessados, de forma transparente às aplicações. Essa abordagem é usada nos discos híbridos, que são HDDs com um pequeno cache interno em estado sólido, geralmente gerenciado pelo *firmware* do próprio disco. Também é possível usar SSDs independentes como cache de HDDs subjacentes, com o gerenciamento feito pelo sistema operacional. O núcleo Linux oferece dois subsistemas de gerenciamento de caches em SSD, *DMCache* e *BCache*, que usam abordagens e algoritmos distintos. Este trabalho avalia estes dois subsistemas em diversas configurações de SSDs, HDDs e RAID, sob diversas cargas de trabalho, com o objetivo de compreender seu funcionamento e definir diretrizes para a configuração de tais subsistemas em ambientes computacionais de médio/grande porte. Dentre os resultados apontados neste trabalho, foi verificado que em *workloads* mais sequenciais, como em servidores de arquivos, o uso de cache pode alcançar até 72% a mais de desempenho se comparado aos RAIDs de HDDs. Em *workloads* aleatórios, como em bancos de dados, o uso de caching SSD pode apresentar pouco desempenho diante de custos elevados por GB, chegando a 79%.

ABSTRACT

Recently, Solid State Disks (SSDs) has elevated performance in secondary storage access. However, its high cost and low capacity make it impossible to fully replace Hard Disk Drives (HDDs) for SSDs in a short-term, especially in larger environments. On the other hand, it is possible to combine the performance of SSDs and low cost of HDDs, using SSDs as cache of HDDs for the most accessed data, transparently to applications. This approach is used in hybrid drives, which are HDDs with a small internal solid state cache, usually managed by device firmware itself. You can also use independent SSDs as underlying HDDs cache with the management assigned to operating system. Linux kernel offers two subsystems caches management in SSD, DMCache and BCache, using different approaches and algorithms. This work evaluates these two subsystems in various configurations of SSDs, HDDs and RAID over various arranges and workloads, in order to understand its operation in specific scenarios and set guidelines for setting up such subsystems in computing environments of medium and large scales. Among the presented results, it was found that in most sequential workloads, such as file servers, the use of cache can achieve up to 72% more performance compared to RAID of HDDs. In random workloads, such as databases, the use of SSD caching may have little performance before high costs per GB, which could reach up to 79%.

CAPÍTULO 1

INTRODUÇÃO

A popularidade dos serviços de Internet, em especial os mecanismos de busca, comércio eletrônico e armazenamento em nuvem, tem provocado a mudança de foco do processamento para a comunicação e armazenamento da informação, aumentado a demanda de capacidade e desempenho por parte das tecnologias de armazenamento secundário, tanto na computação pessoal quanto nos *datacenters* [44]. A tecnologia mais popular de armazenamento secundário ainda é o disco magnético (HDD – *Hard Disk Drive*), que alia grande capacidade de armazenamento e baixo custo por MB. Contudo, o HDD tem tempos de acesso elevados, devido às suas características mecânicas (latência rotacional e de movimento da cabeça de leitura). Essas latências impactam sobretudo os acessos aleatórios, levando a um baixo desempenho.

A mudança do foco do processamento para a comunicação e armazenamento de informação objetiva a confiabilidade, escalabilidade e uma boa relação custo/desempenho. Embora um *crash* em uma aplicação seja uma situação ruim para um usuário, situação pior seria a perda de um dado importante. Nesse sentido, os dispositivos de armazenamento constituem itens de grande relevância em uma arquitetura computacional. Além da dependência do dado consistente, outro fator que acentua a importância desses dispositivos é a relação entre vazão e latência (tempo de resposta).

Os *storages* (dispositivos de armazenamento) possuem diferentes vertentes a serem analisadas na sua integração com um sistema computacional, uma vez que há vários tipos de tecnologias envolvidas na constituição dos dispositivos de armazenamento. Além de cada dispositivo de armazenamento isolado, é importante considerar a relação entre diferentes dispositivos operando em conjunto em um mesmo sistema, cenário este mais comum na maioria dos casos.

A Figura 1.1 apresenta uma visão hierárquica dos principais dispositivos de armaze-

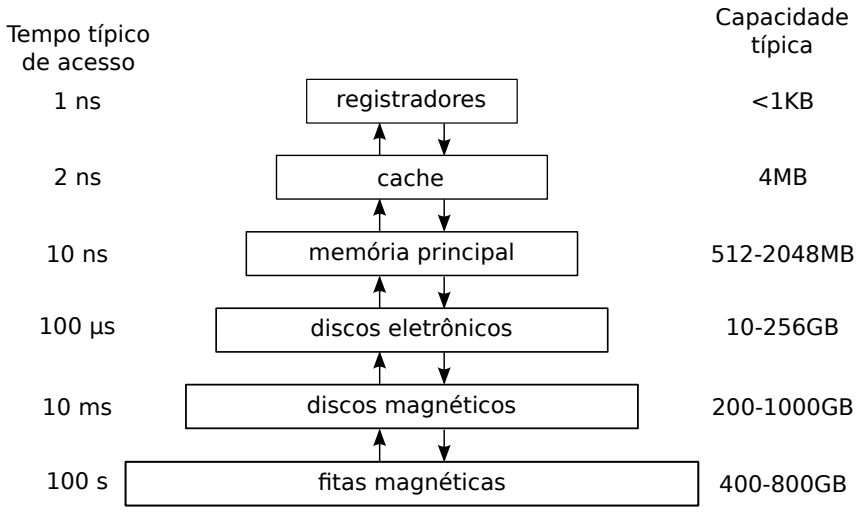


Figura 1.1: Hierarquia de dispositivos de armazenamento. [42, 44, 7]

namento de acordo com a velocidade e o custo. Ao descer na hierarquia o custo por *bit* em geral diminui (bom), ao passo que o tempo de acesso aumenta (ruim). Essa proporção mostra-se lógica, visto que, se o espaço de armazenamento fosse mais rápido e menos caro, não haveria a necessidade de memórias lentas e dispendiosas.

Os dispositivos de armazenamento podem ser voláteis ou não-voláteis. Os dispositivos voláteis são aqueles que perdem seus conteúdos quando os computadores são desligados, pois precisam de energia para se manter. Na ausência de baterias e geradores, os dados devem ser armazenados em dispositivos não-voláteis. Na Figura 1.1 os dispositivos acima dos discos eletrônicos são considerados voláteis, os abaixo, não-voláteis. Os discos eletrônicos podem ser voláteis ou não-voláteis, de acordo com a sua tecnologia.

Um sistema de armazenamento completo deve buscar o equilíbrio, levando em consideração diversos fatores, dentre eles, o uso de memória de alto custo/desempenho juntamente com memórias não-voláteis e de menor custo, por exemplo. No cenário onde ocorrem altos tempos de acesso ou baixa vazão, a abordagem mais comum é a utilização de caches, com o objetivo de "esconder" a latência e aumentar a vazão [42].

O surgimento dos SSDs como dispositivos de alto desempenho não-volátil (abaixo da memória RAM e acima dos HDDs), mostra-se como alternativa viável para uso em conjunto (caching) ou em substituição aos HDDs [7, 20]. Neste cenário, princípios como algoritmos de substituição de cache, rearranjos e parametrizações desses dispositivos pre-

cisam ser revisitados, uma vez que os SSDs não apresentam determinados problemas comuns aos HDDs, como o tempo de rotação e movimentação da cabeça de leitura [7].

No entanto, dispositivos usados para cache são, apesar de mais rápidos, mais caros e menores que o dispositivo de origem dos dados. Por isso, os dados do dispositivo original não cabem todos no cache, o que leva à necessidade de *algoritmos de substituição* para manter no cache os dados mais relevantes a cada momento. O núcleo Linux oferece dois subsistemas de gerenciamento de cache em SSD, o *DMCache* [47] e o *BCache* [36], que usam abordagens e algoritmos distintos.

Outra forma usual de aumentar o desempenho de sistemas de armazenamento secundário com HDDs consiste em combinar diversos discos em arranjos RAID (*Redundant Array of Independent Disks*) [37]. Por permitir acessos paralelos aos discos físicos, um sistema RAID configurado adequadamente oferece ganhos significativos no desempenho de acesso aos dados. É importante observar que sistemas RAID e caches SSD podem ser usados em conjunto.

Dessa forma, este trabalho estuda o comportamento dos subsistemas de cache em SSD oferecidos pelo núcleo Linux, considerando diversos arranjos de discos SSD e HDD, com várias cargas de trabalho. O objetivo principal é compreender o funcionamento desses subsistemas, mas também identificar problemas, propor melhorias e definir diretrizes para a configuração de arranjos SSD/HDD em ambientes computacionais de maior porte.

Dentre as limitações deste trabalho incluem-se: teste das implementações de caching de segundo nível com SSDs, BCache e DMCache, ambiente de testes sobre o sistema operacional Linux, arranjos RAID nível 0 e nível 1 em software, e análise de desempenho por meio de microbenchmaks e macrobenchmarks.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta as principais estruturas de armazenamento de dados do Linux, bem como os dispositivos de armazenamento secundário avaliados. No Capítulo 3 são discutidos algoritmos básicos de substituição de cache e os principais algoritmos modernos para esse fim, dentre eles o algoritmo Multi-Queue (MQ) [52], implementado pelo DMCache e alvo de testes deste trabalho. O Capítulo 4 apresenta as implementações de cache avaliadas, BCache

e DMCache. O Capítulo 5 apresenta a metodologia utilizada nos experimentos e trabalhos relacionados. O Capítulo 6 apresenta os resultados obtidos através de experimentos com microbenchmarks e macrobenchmarks. Os microbenchmarks são usados como linha de base para a interpretação dos macrobenchmarks, que simulam experimentos similares aos encontrados em ambientes reais de servidores. Por fim, o Capítulo 7 apresenta as considerações finais e conclusões do trabalho.

CAPÍTULO 2

ESTRUTURAS DE ARMAZENAMENTO SECUNDÁRIO

Neste Capítulo será apresentado o caminho padrão do armazenamento de dados no Linux, desde quando uma requisição de I/O é submetida por uma aplicação até o momento em que esta requisição é respondida pelo dispositivo de armazenamento. Deste modo, as Seções seguintes seguirão uma estrutura *bottom-up*, indo de hardware a software, nesta ordem. A Seção 2.1 detalha os principais dispositivos de armazenamento de dados. A Seção 2.2 apresenta o caminho padrão do I/O no *kernel*. A Subseção 2.4 apresenta os mecanismos de cache em memória principal do Linux.

2.1 Dispositivos de Armazenamento

Nesta Seção apresentaremos os dispositivos de armazenamento de dados estudados. Os Hard Disk Drives (HDD) serão apresentados na Seção 2.1.1. Os dispositivos de armazenamento em estado sólido (Solid State Disk — SSD) serão apresentados na Seção 2.1.2. Por fim, detalharemos os arranjos RAID na Seção 2.1.3, que consistem de agrupamentos lógicos compostos por HDDs e/ou SSDs.

2.1.1 Hard Disk Drive (HDD)

Os discos magnéticos ou HDDs representam uma das principais formas de armazenamento secundário, visto que os dados não são acessados diretamente pelo processador, diferente do que acontece na memória RAM. O disco é um dispositivo de característica não-volátil, útil para o armazenamento de grandes quantidades de dados. Quanto à sua estrutura, possui lâminas metálicas sólidas de constituição rígida semelhante aos CD-ROMs chamadas de *platters*, que possuem composição magnética e permitem que dados sejam gravados na sua superfície. Os primeiros discos rígidos possuíam uma grande quantidade de *platters*, atualmente o número máximo utilizado é de quatro *platters*, em alguns casos os

discos chegam a possuir apenas um *platter*. O diâmetro dos *platters* determina o tamanho do disco fisicamente, enquanto que a capacidade está associada à densidade (área utilizada para guardar uma unidade mínima de armazenamento). Atualmente os discos mais comuns são os de 3.5 polegadas [33].

A Figura 2.1 apresenta os principais componentes da estrutura interna de um disco rígido. O *platter* é dividido em estruturas concêntricas circulares chamadas de *tracks* (trilhas), as quais são contadas de forma incremental da parte externa para a interna do disco. As trilhas, por sua vez, são divididas em *sectors* (setores), que são a menor unidade física de armazenamento de um HDD. Cada setor é composto por *header* (cabeçalho), área de dados e *trailer*. *Header* e *trailer* armazenam informações que são utilizadas somente pela controladora do disco, como o número do setor e informações para correção de erros.

Sobre cada superfície do *platter* há uma *head* (cabeça de leitura) associada. As cabeças de leitura estão conectadas a um *arm* (braço de leitura) o qual é sustentado por um *arm assembly* (concentrador dos braços). Deste modo, as cabeças de leitura não têm autonomia umas sobre as outras. O conjunto de trilhas em que as cabeças de leitura se concentram em determinado instante chama-se *cylinder* (cilindro).

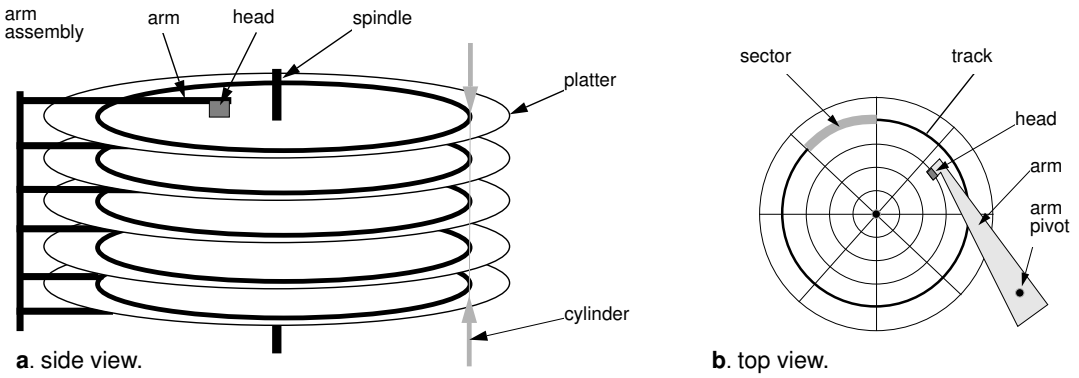


Figura 2.1: Visão interna (lateral e superior) de um disco rígido. [41]

O disco possui também um motor de rotação responsável por girar os *platters*. Os primeiros discos eram construídos com motores de 3.600 a 5.600 rpm (*rotations per minute*, ou rotações por minuto). Os discos mais modernos chegam a 7.500, 10.000 ou 15.000 rpm [33]. A rotação é um dos fatores determinantes na performance dos discos rígidos, uma vez que a cabeça de leitura esteja posicionada no setor sobre o qual se deseja fazer a

leitura.

Outros fatores importantes para a performance dos discos são a *taxa de transferência* e o *tempo de posicionamento*. A taxa de transferência está relacionada ao posicionamento dos dados no disco (trilhas mais internas ou mais externas aos *platters*), à velocidade de rotação e ao barramento de dados. Uma taxa de transferência comum entre os discos de 7.200 rpm é de 70 MB/s para acesso sequencial. A posição dos dados nas partes mais externas dos *platters* podem prover até 40% a mais de performance [33]. O segundo fator, o tempo de posicionamento, está relacionado a outros dois fatores, o tempo para posicionar o braço de leitura no cilindro desejado (*seek time*) e o tempo para rotacionar o disco de modo que o objeto desejado esteja sob a cabeça de leitura. A melhoria do *seek time* é um dos principais desafios de escalonamento de disco e, consequentemente, da melhoria no tempo de acesso aleatório.

2.1.2 Solid-State Disk (SSD)

Nos últimos anos um novo dispositivo de armazenamento passou a ser utilizado com mais frequência em computadores portáteis e *data centers*. Esse dispositivo possui características que trazem novas expectativas quanto a consumo de energia, tamanho, capacidade e performance. Trata-se dos *solid state disks* (SSDs), os quais já estão presentes há mais de 30 anos, mas até então eram de alto custo para utilização em larga escala [7].

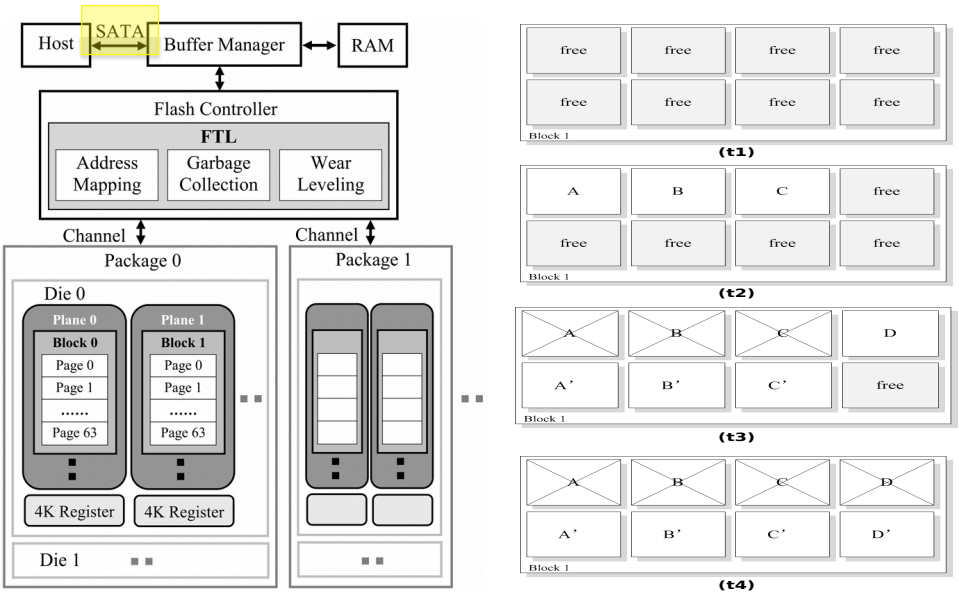
Ao contrário dos HDDs que utilizam peças mecânicas na sua estrutura, o que reduz o desempenho e aumenta o consumo de energia, os SSDs utilizam memória não-volátil (NAND flash) para armazenar os dados. A ausência de partes móveis deu aos SSDs o nome de "*solid-state*" e, consequentemente, tornou-os menos frágeis do que os discos magnéticos.

Os dispositivos de armazenamento utilizados em *smartphones* e câmeras digitais como SD (Secure Digital) e CF (CompactFlash) apresentam versões menos complexas dos SSDs. O fator determinante na preferência dos SSDs em vez desses dispositivos de armazenamento são os requisitos de performance das aplicações. Uma câmera digital normalmente possui menos demanda de acessos a dados do que um *laptop multicore*, por exemplo, que

exige melhores implementações de SSDs.

Tanto os HDDs quanto os SSDs representam as principais estruturas de armazenamento não-volátil e fazem parte da classe de dispositivos de armazenamento chamada de *block device*. Cada *block device* é constituído de três partes principais: mídia de armazenamento, controladora (para gerenciamento da mídia) e interface para acesso à mídia. A mídia de armazenamento é o fator chave por trás da performance e do custo dos SSDs.

A grande maioria dos SSDs em uso atualmente são construídos sobre a tecnologia NAND *flash memory*. As células de memória NAND armazenam elétrons em um capacitor por tempo indefinido sem o uso de energia. A escrita ou exclusão de dados nesses chips é feita ou adicionando elétrons (*programming*) ou removendo (*erasing*) a célula de memória com a utilização de pulsos de alta voltagem. A leitura é realizada de maneira simples através de um conversor analógico-digital com a injeção de sinal de *bias voltage* às células lidas. Diferentes tipos de memória utilizam limites distintos para determinar o valor de uma célula de memória [7].



(a) Arquitetura interna. (b) Processo de escrita.

Figura 2.2: Arquitetura e I/O de escrita em SSDs [4].

A Figura 2.2(a) apresenta a arquitetura padrão de um SSD. Conforme citado anteriormente, o processo de escritas difere-se substancialmente da leitura, sendo processos assíncronos para esses dispositivos, uma vez que para escrever (*programming*) dados no

SSD, em alguns casos, é necessário haver uma exclusão (*erasing*). Atentando-se para os blocos e páginas apresentados na Figura 2.2(b) podemos observar o processo de escrita. Em *t1* uma solicitação de escrita com as páginas A, B e C são gravadas em bloco vazio, em *t2* uma nova solicitação escreve a página D e atualiza ABC, para isso, as páginas ABC são invalidadas e reescritas (conforme *t3*). Em *t4* a página D é atualizada e o bloco chega ao seu limite de utilização.

As operações de remoção apagam os blocos para novas escritas de páginas. Quando um bloco chega a um limite determinado de páginas escritas (50%, por exemplo), passa a atuar o *garbage collector*, o qual libera novos blocos e reorganiza páginas de blocos sobrecarregados; isso acontece para que nas solicitações de escritas não haja a necessidade de limpeza, diminuindo a performance. Além disso, os SSDs sofrem de limites como a quantidade de escritas sobre o mesmo bloco, para isso processos de balanceamento de uso (*wear leveling*) atuam para evitar que haja blocos muito usados e outros pouco usados. Essas duas técnicas necessárias a esse dispositivo, no entanto, produzem uma quantidade de escritas superior ao uso comum das aplicações, causando *write amplification*, o que penaliza a escrita [20]. Na próxima Seção serão apresentados os mecanismos de discos redundantes (RAID).

2.1.3 Redundant Array of Inexpensive Disks (RAID)

Nos últimos anos, o desempenho das CPUs tem crescido de forma exponencial, chegando a duplicar a cada 18 meses, o que não acontece com os dispositivos de disco. De 1970 aos dias atuais, o tempo médio de movimentação da cabeça de leitura (*seek time*) dos discos melhorou de 50 a 100 ms para menos de 10 ms. Para outros ramos da tecnologia este avanço poderia ser bom, mas para a computação é um ganho muito pequeno. Assim, a disparidade entre CPUs e dispositivos de disco tem ficado cada vez mais acentuada [44].

Pelo fato do processamento paralelo tornar as CPUs mais rápidas, este fato levou muitos a acreditarem que o mesmo poderia ser feito para dispositivos de E/S a fim de agilizar o acesso a disco e diminuir a disparidade de desempenho entre os dispositivos de entrada e saída e os processadores, assim como de confiabilidade. Neste sentido [37] sugeriram seis

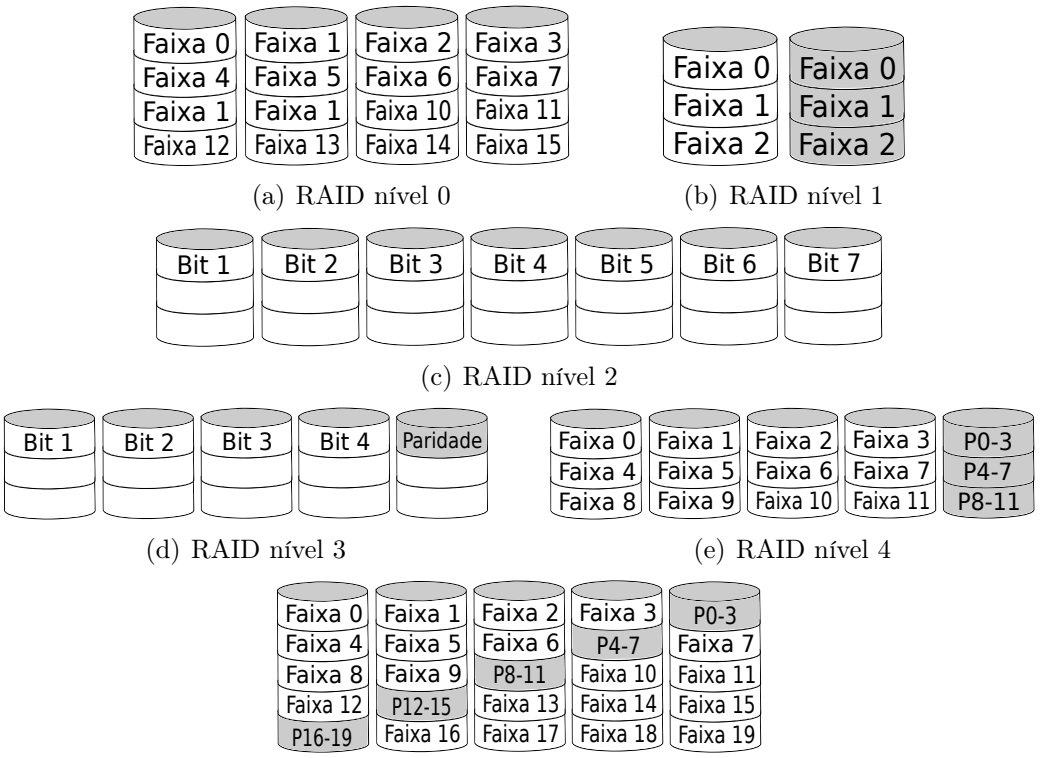
organizações para os discos e as definiram como *Redundant Array of Inexpensive Disks* (RAID - arranjo redundante de discos baratos), que a indústria adotou rapidamente e substituiu o *'I'* por 'Independentes', por questões mercadológicas [37, 44].

A ideia básica em que o RAID se baseia é juntar vários discos em uma única caixa e se apresentar ao sistema como um único disco, substituindo a placa controladora por uma placa RAID, visto que discos ATA e SCSI têm um bom desempenho, confiabilidade e baixo custo, podendo ainda permitir vários dispositivos em uma única controladora (15 para dispositivos mais robustos) [29].

O RAID nível 0 é mostrado na Figura 2.3(a) e descreve um modelo de RAID baseado em *stripping*, que consiste na divisão dos dados em faixas, cada uma contendo k setores. A faixa 0 consiste nos setores de 0 a $k-1$, a faixa 1 consiste nos setores de k a $2k-1$ e assim por diante. O processo de gravação é realizado através de uma alternância consecutiva entre as faixas, mais conhecido como *round-robin*. No exemplo da Figura 2.3(a), quando uma operação de leitura é recebida, o controlador RAID quebra esta operação em outras quatro, que são enviadas para as controladoras de disco e processadas em paralelo, aumentando o desempenho proporcionalmente ao número de discos que processam as requisições paralelas. Deste modo, o RAID0 trabalha melhor em requisições maiores. Uma desvantagem do RAID0 acontece na perda de dados, pois, como os dados estão divididos em várias faixas, a perda de um disco causa a perda total dos dados [44].

Outro tipo de RAID é o RAID nível 1, mostrado na Figura 2.3(b). Neste tipo de RAID todos os dados são espelhados em outros discos de forma que existam discos primários e discos de espelho. Durante o processo de escrita, cada faixa é escrita duas vezes; durante o processo de leitura, os dados podem ser lidos de qualquer uma das faixas. O desempenho é duas vezes superior ao de um único disco sem RAID. A tolerância a falhas é a grande vantagem dessa abordagem, uma vez que a perda de um dos discos não ocasionará a perda dos dados, visto que o disco de espelho atuará em substituição ao disco danificado. A restauração dos dados é feita apenas com a instalação de um novo disco, copiando para ele os dados do disco ativo (espelho).

Os níveis 2 e 3 de RAID não serão utilizados neste trabalho, contudo merecem um



(a) RAID nível 0 (b) RAID nível 1

(c) RAID nível 2

(d) RAID nível 3 (e) RAID nível 4

(f) RAID nível 5

Figura 2.3: RAID níveis 0 a 5. Os discos cópia de segurança e paridade estão sombreados [44].

breve detalhamento. O *RAID nível 2*, conforme mostrado na Figura 2.3(c), em vez de utilizar faixas, como no RAID nível 0 e 1, utiliza palavras (conjuntos de bits) ou, em alguns casos, *bytes*. Ele trabalha quebrando essas palavras na quantidade de discos disponíveis, sendo que parte desses bits são utilizados para correção de erros (*Error Correcting Code*). A grande desvantagem desse nível de RAID é que os discos devem estar sincronizados para que esses bits de correção possam funcionar corretamente. Essa abordagem só faz sentido caso se utilize uma quantidade substancial de discos (o que, mesmo com 32 discos, tem-se uma sobrecarga de 19%). O *RAID nível 3*, mostrado na Figura 2.3(d), opera de modo semelhante ao RAID nível 2, porém mais simplificado, pois guarda um único bit de paridade em um disco separado, chamado de disco de paridade. Nesse RAID os discos também devem estar perfeitamente sincronizados. O RAID nível 3 não faz correção de erros, mas essa afirmação só é válida para os erros aleatórios e desconhecidos. Para casos de erros em que um disco se perde, a posição danificada é conhecida e os bits necessários para reconstruir os dados no disco substituído podem ser facilmente calculados. A desvantagem

dos níveis 2 e 3 de RAID é que, mesmo que forneçam grandes quantidades de dados com certo nível de disponibilidade, o número de requisições que eles podem tratar por segundo não é melhor do que um único disco [44].

O RAID nível 4 novamente utiliza faixas, conforme mostrado na Figura 2.3(e), não necessitando que os discos estejam sincronizados. É similar ao RAID nível 0, mas exige que a paridade entre as faixas sejam escritas em um disco extra. Se as faixas têm k *bytes* de tamanho, é calculado um OU EXCLUSIVO entre as faixas, que resulta em uma faixa de paridade de k *bytes*, que é escrita em um disco separado. Esse nível de RAID é suficiente para a perda de um único disco, mas tem o funcionamento prejudicado em casos de pequenas atualizações, pois, para cada atualização, todos os discos devem ser lidos para que seja recalculada e reescrita a paridade.

Por o RAID nível 4 utilizar apenas um único disco para paridade, este pode estar sobrecarregado e comprometer toda a performance do arranjo. Esse gargalo não acontece no RAID nível 5, ilustrado na Figura 2.3(f), pois este distribui os bits de paridade por todos os discos do arranjo de modo uniforme e circular, não sobrecarregando um único disco. Entretanto, na quebra de um dos discos, a reconstrução dos dados torna-se mais dolorosa, visto que este processo torna-se mais complexo.

2.2 Acesso a Disco no Kernel do Linux

O *path*, ou caminho, das operações diretas aos dispositivos de I/O é o *raw I/O*. O *path* para as operações do Virtual File System (VFS) e de *file systems* é o *file system I/O*, que inclui *Direct I/O* (operações sem cache) [13]. Exemplificaremos o comportamento das requisições de *file system I/O* (Figura 2.4), para isso tomaremos uma solicitação de leitura enviada por uma aplicação de usuário a um dispositivo de bloco. Por padrão, todas as chamadas de sistema são direcionadas ao VFS (Figura 2.4a) para processamento inicial. Essas chamadas são conhecidas como POSIX padrão, a exemplo, *open*, *lseek*, *read*, *write*, entre outras. Essa interface é denominada camada superior do VFS e possibilita uma comunicação comum entre o processo do usuário e o sistema de arquivos. Após processamento, o VFS encaminha essas chamadas de sistema a uma interface inferior,

denominada *interface VFS*, que faz a comunicação com os diversos sistemas de arquivos existentes [44].

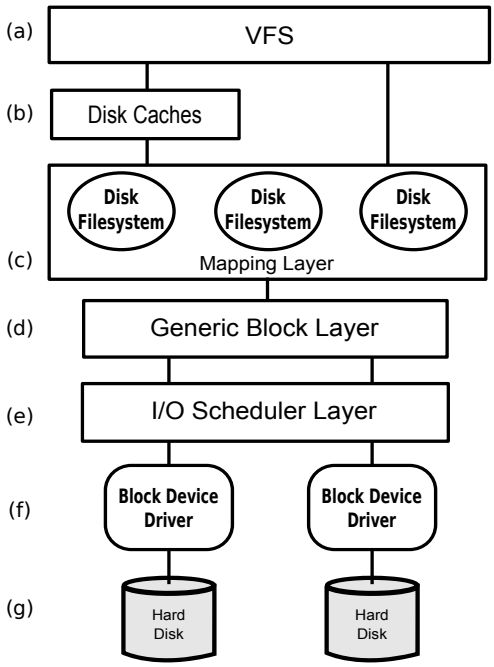


Figura 2.4: Pilha genérica de I/O no Linux [3].

O VFS mantém as últimas informações acessadas em uma cache na memória principal (Figura 2.4b) e é responsável por decidir se essas informações serão utilizadas a partir dessa cache ou do dispositivo de armazenamento diretamente. Há dois mecanismos de cache em memória principal (nomeada nível 1 neste trabalho): o cache de páginas (gerenciada pelo VFS) e o cache de *buffers* (gerenciada pela camada de mapeamento de bloco) [13], apresentados com mais detalhes na Seção 2.4.

A próxima camada, *Mapping Layer* (Figura 2.4c), é interna aos sistemas de arquivos e responsável pelo mapeamento entre os blocos (*i-node*, *offset* e *size*) do arquivo solicitado pela aplicação nível usuário — por meio do VFS — e o endereçamento desses blocos no disco. Esse mapeamento é feito pelo sistema de arquivos e depende de fatores como tabelas de indexação, gerenciamento de espaço livre, implementação de diretórios e método de alocação.

Após encontrar o endereço dos blocos requisitados, uma solicitação de I/O é gerada e encaminhada à próxima camada, a *Generic Block Layer* (Figura 2.4d). Cada operação deve englobar um ou mais blocos físicos adjacentes, caso contrário, várias operações de

I/O serão criadas. O *Generic Block Layer* abstrai detalhes dos dispositivos de bloco para as camadas superiores. Mapeamentos de dispositivos virtuais, volumes lógicos como LVM (*Logical Volume Manager*) [30] e RAID em software (*Redundant Array of Independent Disks*) [37], apresentado em detalhes na Seção 2.1.3, são gerenciados por esta camada.

As requisições da *Generic Block Layer* são encaminhadas a um escalonador de disco (Figura 2.4e), sendo que cada dispositivo de bloco possui seu próprio escalonador. O escalonador guarda uma fila ordenada por setor das requisições pendentes, as quais podem ser atendidas em ordem ou de acordo com alguma política de escalonamento pré-estabelecida. Com o intuito de minimizar a latência das requisições, a maioria dos escalonadores agrupam as requisições de blocos adjacentes, mesmo que solicitados em separado, de forma que sejam atendidas em uma única operação. As requisições já escalonadas são enviadas para uma fila conhecida como *dispatch queue* e, em seguida, enviadas para o *driver* do dispositivo (Figura 2.4f), responsável pela comunicação com a controladora do periférico (Figura 2.4g).

2.3 Escalonamento de E/S

No Linux, o escalonador de disco pode ser configurado na inicialização do sistema, pelo *driver* do dispositivo ou em tempo de execução pelo administrador (através do *sysfs*). Os quatro principais algoritmos de escalonamento que estão ou já estiveram presentes no kernel são descritos a seguir:

- **Noop:** encaminha as requisições na ordem em que chegam, sem nenhum tratamento especial, implementando portanto uma política FCFS (*First Come, First Served*).
- **Deadline:** cada requisição possui um limite de tempo de $500ms$ para leitura e $5s$ para escrita; as operações de leitura possuem maior urgência por geralmente serem síncronas. Além disso, o *deadline* possui quatro filas, duas ordenadas por setor para leitura e escrita e duas ordenadas por *deadline* para leitura e escrita.
- **Anticipatory:** é semelhante ao *Deadline*, diferindo-se por possuir tempos menores (leitura $125ms$ e escrita *250ms*) e pela forma como são atendidas as requisições nas

filas, podendo voltar o braço de leitura a requisições próximas, por exemplo. Este algoritmo foi removido do núcleo a partir da versão 2.6.33, após ser verificado que seria possível obter o mesmo efeito com o *tuning* de outros algoritmos dessa lista [24].

- ***Completely Fair Queueing* (CFQ):** padrão no kernel. Baseia-se nos algoritmos de enfileiramento justo (ou *fair queueing*), normalmente utilizados para tráfego de rede. Algoritmos de enfileiramento justo são usados quando diversas fontes de solicitação de dados precisam utilizar o mesmo canal de comunicação [34].

É importante notar que os algoritmos de escalonamento são de grande importância para dispositivos de armazenamento que possuem partes mecânicas (HDDs), uma vez que, neste caso, a ordenação inadequada poderá causar um baixo desempenho em relação a outro padrão de ordenação das requisições. No caso dos SSDs essa preocupação não é em princípio considerada, visto que a latência independe da posição física dos blocos [20]. Na próxima Seção serão apresentadas as estruturas de cache de primeiro nível (RAM) do Kernel Linux.

2.4 Estrutura de Caches no Linux

O Linux emprega várias técnicas para melhorar o desempenho do sistema, reduzindo o acesso a disco e direcionando este acesso para a memória principal tanto quanto possível. Uma das principais abordagens abertas para a "esconder" a latência de I/O é o mecanismo de *disk caches*. O *disk cache* diz respeito a um mecanismo de software *kernel-level* que permite ao sistema operacional manter em memória RAM dados normalmente armazenados no disco, de forma que futuros acessos a esses dados sejam realizados mais rapidamente e sem acesso a disco [3].

Esse mecanismo é composto por três componentes principais: *dentry cache*, *page cache* e *buffer cache*. O *dentry cache* trata-se de uma tabela em RAM utilizada pelo VFS (Virtual File System) que acelera a tradução de endereços de *i-nodes* (metadados de sistemas de arquivos que contêm as informações dos arquivos). Os outros dois componentes são os

principais responsáveis pelo cache de dados dos arquivos.

Buffer Cache

O *buffer cache* é uma área de memória que contém dados de blocos de disco. Cada bloco refere-se a áreas, geralmente contíguas, de *bytes* na superfície de um disco (ou qualquer outro dispositivo de bloco). O tamanho do bloco depende do sistema de arquivos utilizado, normalmente 4 KB, o que equivale a oito setores físicos (512 *bytes*).

O principal objetivo do *buffer cache* é fazer com que processos não sejam prejudicados pela alta espera relacionada à baixa performance dos discos magnéticos. Para isso, os dados são escritos em *buffer* e a escrita de dados no disco é atrasada, sendo realizadas atualizações em lote da memória para o disco em intervalos fixos (normalmente 30s).

Uma vez que todas as informações "sujas" devem ser enviadas para o disco, mais cedo ou mais tarde, é necessário identificá-las. Para isso, o *kernel* mantém duas informações sobre os blocos: um *timestamp* e um *dirty bit* (ou *bit* sujo). O *dirty bit* indica quais dados da memória estão diferentes do disco e, portanto, marcados para escrita. O *timestamp* é utilizado para saber a idade do dado, sendo de grande utilidade para as políticas de alocação e substituição do cache.

Além disso, as páginas de memória são alocadas sob demanda e liberadas tão logo não sejam mais utilizadas. As duas principais estruturas que constituem o *buffer cache* são as seguintes: (a) um conjunto de *buffer headers*, que descrevem os *buffers* e (b) uma tabela *hash*, que ajuda o *kernel* a encontrar o *buffer head* correspondente a um par *{device,block_number}* .

Page Cache

O *page cache* é um tipo de cache de disco que armazena *page frames* que contêm dados pertencentes a arquivos. Isso o diferencia do *buffer cache*, visto que os *page frames* no *page cache* não necessariamente correspondem a blocos fisicamente contíguos no disco.

O *page cache* possui uma estrutura muito mais simples do que o *buffer cache* e é utilizado basicamente para cache de dados acessados por operações de *page* I/O. Todos os

acessos a arquivos regulares feitos por operações como *read*(), *write*() e *mmap*() acontecem por meio de *page cache*. A unidade que armazena os dados nesse cache é a página, uma vez que operações de I/O transferem páginas de dados. Além disso, uma página não necessariamente contém blocos de disco adjacentes e, portanto, não são identificadas por chaves como {*device*,*block_number*}, neste caso, a página é identificada pelo *inode* do arquivo e o *offset* desejado.

Em sua estrutura, o *page cache* possui dois componentes principais: uma *page hash table* e uma *i-node queue*. O primeiro permite ao *kernel* encontrar de forma rápida o endereço do descritor de página associado ao *inode* e ao *offset* do arquivo solicitado. O segundo possui uma lista de descritores de páginas que correspondem às páginas de dados de arquivos individuais, identificados por *inodes*. A manipulação do *page cache* envolve basicamente adicionar ou remover entradas nessas estruturas de dados, bem como atualizar objetos de *inodes* que referenciem arquivos em cache.

Apesar da apresentação separada entre *buffer cache* e *page cache*, tal separação foi mantida somente até a versão 2.4 do *kernel*, sendo unificados nas versões posteriores. Um dos objetivos dessa junção é evitar redundância, onde blocos presentes no *buffer cache* podiam aparecer repetidos no *page cache*. Com a mudança, caso um bloco já esteja armazenado no *buffer cache* será apenas mapeado no *page cache* [12].

Comparativo entre Buffer e Page Cache

I/O Operation	Cache	System Call	Função do Kernel
Lê em um block device	Buffer	read()	block_read()
Escreve em um block device	Buffer	write()	block_write()
Escreve um diretório Ext2	Buffer	getdents()	ext2_bread()
Lê um arquivo regular Ext2	Buffer	read()	generic_file_read()
Escreve um arquivo regular Ext2	Page, Buffer	write()	ext2_file_write()
Acesso a arquivo mapeado em memória	Page	None	file_map_nopage()
Acesso a páginas swapped-out	Page	None	do_swap_page()

Tabela 2.1: Uso de *buffer cache* e *page cache*. [3, Adaptado]

A Tabela 2.1 apresenta algumas chamadas de sistema e funções de *kernel* em que esses *caches* atuam. Conforme a tabela, operações de *buffer* I/O fazem uso somente das funções

de *buffer cache*. Contudo, operações *page* I/O usam tanto *page* quanto *buffer cache* (em alguns casos). As duas últimas funções da tabela não apresentam uma *system call* específica, isso significa que tanto o acesso a esta função quanto o atraso de carregamento no disco de páginas que não estão na memória são transparentes ao programador.

2.5 Conclusão

Neste Capítulo apresentamos as principais estruturas que envolvem o acesso a dados no Linux bem como os principais dispositivos de armazenamento atuais estudados neste trabalho. É possível observar diversos fatores que envolvem o armazenamento de dados atualmente como, por exemplo, as deficiências dos discos magnéticos frente ao seu baixo custo e as altas performances dos SSDs, mesmo diante de desafios como *write-amplification*. Nesses cenários, é notável a necessidade de estudos que direcionem um melhor aproveitamento desses recursos.

CAPÍTULO 3

ALGORITMOS DE SUBSTITUIÇÃO DE CACHE

Neste Capítulo apresentamos os principais algoritmos de substituição de cache básicos e atuais. O objetivo fundamental desses algoritmos é obter a maior quantidade possível de *hits* (acertos) em cache com um tempo de processamento aceitável, com a finalidade de melhorar o desempenho. Dois algoritmos apresentados, LRU e Multi-Queue (MQ), são utilizados pelos mecanismos de cache estudados neste trabalho (BCache e DMCache), os demais algoritmos são apresentados apenas para fins de comparação.

A Seção 3.1 traz o algoritmo Ótimo, um algoritmo que não tem utilização prática mas que é adotado como linha base para avaliação de outros algoritmos factíveis. A Seção 3.2 apresenta o LRU (Least Recently Used), um algoritmo que tem a recência como única métrica de avaliação; o LRU é um dos algoritmos mais utilizados nos sistemas atuais e a base de implementação para outros algoritmos posteriores. As Seções 3.3 (LIRS) e 3.4 (ARC) apresentam dois algoritmos modernos que trazem melhorias ao LRU e ainda consideram a frequência dos dados como um segunda métrica de avaliação, ambos com baixa complexidade computacional. Por último, na Seção 3.5, apresentamos o algoritmo Multi-Queue (MQ), outro moderno algoritmo desenvolvido especificamente para caches de segundo nível de armazenamento hierárquico que busca garantir as características dos anteriores sendo ainda a base do DM-Cache (mecanismo de cache em SSDs do Linux), apresentado na Seção 4.2 e que será avaliado neste trabalho.

3.1 Algoritmo Ótimo

O algoritmo ótimo, também conhecido como OPT ou MIN, tem a taxa de erros mais baixa dentre todos os algoritmos de substituição. O seu funcionamento básico substitui a página que não será utilizada pelo maior período de tempo. Contudo, não é possível saber antecipadamente qual página levará mais tempo para ser utilizada, o que torna a

implementação deste algoritmo possível apenas em modo *off-line*, ou seja, deve haver um conhecimento prévio da sequência de referências [42].

Apesar de não ser implementável sem tal conhecimento, o algoritmo ótimo é útil como parâmetro de comparação com outras implementações de algoritmos. Por exemplo, apesar de uma implementação não ser ótima, pode estar 12,03% ou 10% pior que o algoritmo OPT.

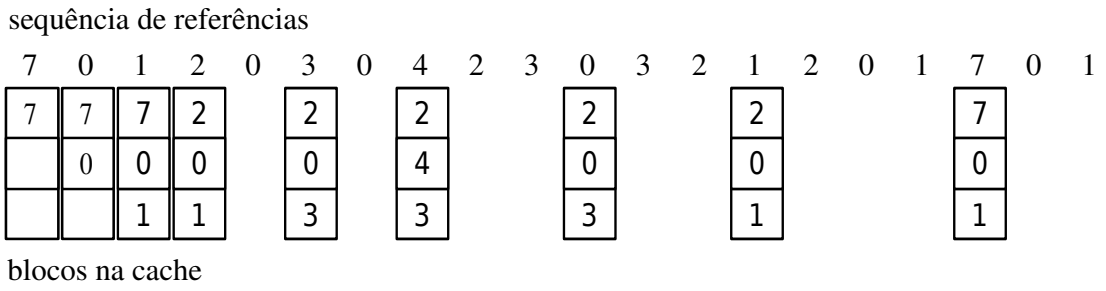


Figura 3.1: Funcionamento do Algoritmo Ótimo (OPT) [42].

A Figura 3.1 apresenta um exemplo de uso do algoritmo ótimo de substituição. A linha superior apresenta as páginas requisitadas na cache sequencialmente. As primeiras solicitações *{7,0,1}* são preenchidas compulsoriamente (cache compulsória), por conta de fazerem o primeiro preenchimento do cache que encontra-se vazio. Na próxima solicitação (a *2*), o OPT substitui *7*, uma vez que *7* será solicitado por último dentre os blocos pertencentes à cache naquele momento (que são *{7,0,1}*).

3.2 LRU

Visto que o algoritmo ótimo não é factível para uso geral, aproximações são sugeridas, dentre elas o algoritmo *Least Recently Used* (LRU). Enquanto o algoritmo OPT se baseia no futuro para saber qual página deverá ser substituída, o LRU analisa o passado recente das referências como uma aproximação para o futuro próximo. Deste modo, a premissa básica do LRU é que páginas utilizadas recentemente têm grande possibilidade de serem reutilizadas em breve [44].

A Figura 3.2 apresenta a funcionamento do algoritmo LRU. Para o mesmo conjunto de referências aplicado ao algoritmo OPT, o LRU apresenta 12 faltas (também chamadas

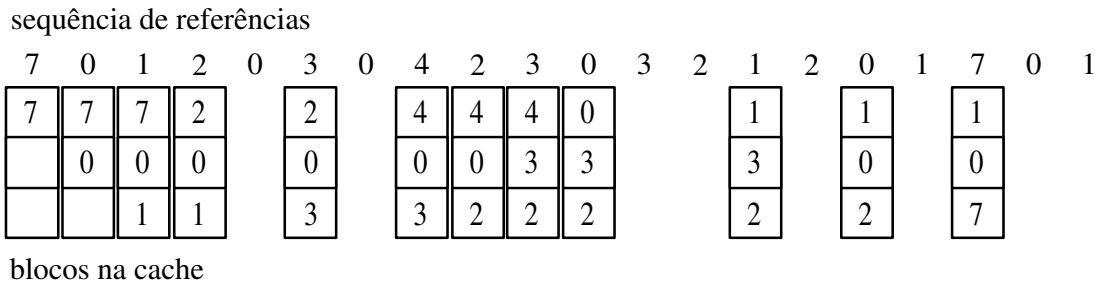


Figura 3.2: Funcionamento do Algoritmo LRU [42].

de *cache miss* no caso de *caches*). As três primeiras faltas são as mesmas que ocorrem no algoritmo OPT, essas faltas são chamadas de faltas compulsórias, as quais ocorrem por a cache estar vazia, devendo ser preenchida inicialmente. A primeira referência que exige substituição é a referência inicial à página 4, neste caso, o LRU irá verificar que, dos quadros que estão na memória, a página 2 foi a menos recentemente utilizada e deve ser descartada. Assim, o LRU substituirá a página 2 sem saber que ela será utilizada na próxima referência. Apesar deste problema, a substituição do LRU com 12 faltas é melhor que algoritmos mais simples como o *First-In-First-Out* (FIFO) com 15 faltas. O algoritmo FIFO segue uma fila simples, onde o primeiro item a entrar será o primeiro a sair e assim sucessivamente.

Apesar do LRU ser teoricamente realizável tem um custo alto. Para implementá-lo, por exemplo, é necessário manter uma lista encadeada de todas as páginas com a página mais recentemente usada (MRU) em uma das pontas da lista e a página menos recentemente usada (LRU) na outra ponta. Ocorre que, para cada acesso a dados, a página correspondente deste acesso na lista deverá ser buscada, removida e colocada na ponta da lista, o que gera um *overhead* alto [44].

Outra opção é armazenar um *timestamp* (carimbo do tempo) da última utilização de cada página para, deste modo, decidir oportunamente e com base neste dado qual página será substituída, descartando aquela que está há mais tempo sem ser utilizada [42]. Ambas as abordagens possuem um alto custo, a primeira em cada acesso às páginas, a segunda a cada substituição de páginas. Apesar disso, o LRU possui complexidade constante (ou baixa complexidade) e boa relação de *hit I/O* diante do *overhead* causado. Adicionalmente, o LRU é um dos algoritmos mais utilizados para casos gerais, contudo atenta-se

apenas à recência, o que traz problemas a vários *workloads*, conforme apresentado nas próximas seções.

Apesar do algoritmo LRU ser amplamente utilizado para aplicações de uso geral por conta da sua simplicidade, alguns comportamentos anômalos podem ser encontrados em *workloads* típicos em que taxas de *hit I/O* crescem muito pouco ou desproporcionalmente ao aumento da cache. Essas observações demonstram a não aptidão de adaptação do LRU para diferentes padrões de acesso com fraca localidade: *file scanning*, acessos regulares a mais blocos do que o tamanho da cache e acesso a blocos com frequências variadas [18]. No primeiro caso, o LRU pode evitar blocos importantes trocando-os por blocos que não serão mais usados [35]. No segundo caso, arquivos maiores que a cache com leituras em *looping* podem fazer com que blocos da cache necessários à leitura estejam sempre fora da cache, causando um péssimo desempenho [43]. No último caso, blocos com maior probabilidade de acesso, como índices em um banco de dados, podem ser evitados para dar lugar a blocos com menor probabilidade, como os dados das linhas e colunas do banco [18].

Apesar das limitações apresentadas, o LRU possui vantagens na baixa complexidade e adaptabilidade, uma vez que faz uso de apenas um parâmetro, a recência. Outros algoritmos *Least Frequently Used*-like (LFU-*like*), para solucionar este problema, buscam trabalhar com mais informações históricas e espaciais dos blocos, o que aumenta a complexidade desses algoritmos, impossibilitando de serem adotados para uso geral.

3.3 LIRS

O algoritmo *Low Inter-reference Recency Set* (LIRS), proposto por [18], é um algoritmo de cache de primeiro nível (em memória principal) construído não com objetivo de solucionar todos os problemas do LRU, mas de reduzir os problemas citados anteriormente, mantendo a baixa complexidade do LRU. Para alcançar isso, o LIRS faz uso da *Inter-Reference Recency* (IRR - ou recência entre referências) como histórico de referência de cada bloco, onde o IRR de um bloco b corresponde ao número de outros blocos acessados sem repetições entre as duas últimas referências a b (Tabela 3.1). O algoritmo assume que

Blocks / Virtual time	1	2	3	4	5	6	7	8	9	10	Recency	IRR
E									x		0	inf
D		x					x				2	3
C				x							4	inf
B			x		x						3	1
A	x					x		x			1	1

Tabela 3.1: Exemplo de como um bloco é escolhido para substituição no LIRS. X representa um acesso a um bloco no *virtual time* apresentado no cabeçalho da coluna. Os valores de recência e IRR apresentados são considerando o *virtual time* 10. O conjuntos LIR e HIR são, respectivamente, $\{A, B\}$ e $\{C,D,E\}$ [18].

se o IRR de b é alto, muito provavelmente o próximo IRR de b será alto também, deste modo, blocos com alto IRR são selecionados para substituição. É interessante notar que mesmo blocos com baixa recência (especicamente, o número de blocos acessados entre o tempo atual e o último acesso a b) podem ser substituídos por conta do alto IRR.

O LIRS também possui os *High Inter-Reference Recency* (HIR – ou recência entre referência alta), conforme o nome sugere, representa os blocos mais propícios a substituição. Na implementação do LIRS, [18] gerencia a cache de modo que todos os blocos do conjunto LIR estejam dentro da cache. Blocos HIRs que estão em uma pequena porção da cache são chamados HIR persistentes, os demais blocos são os blocos HIR não persistentes.

Por fim, o LIRS apresenta melhor desempenho sobre algoritmos anteriores baseados em LRU e minimiza as limitações apresentadas sem a necessidade de *tuning* ou adaptação de parâmetros sensíveis. Além disso, LIRS demostra que a partir de certo tempo o IRR torna-se um parâmetro confiável para diversos *workloads*.

3.4 ARC

Diante das limitações apresentadas na Seção 3.3 tanto para o algoritmo LRU quanto para os algoritmos de frequência LFU-*like* e de propostas como o LIRS não serem completamente adaptáveis a diversos *workloads*, Megiddo et al. (2003) [31] propuseram um novo algoritmo, o Adaptive Replacement Cache (ARC), com o objetivo de ser uma solução às alterações contextuais dos *workloads* sem a necessidade de *tuning* ou adaptações manuais sobre o algoritmo. Para isso, o ARC divide os mecanismos de cache em duas listas, uma

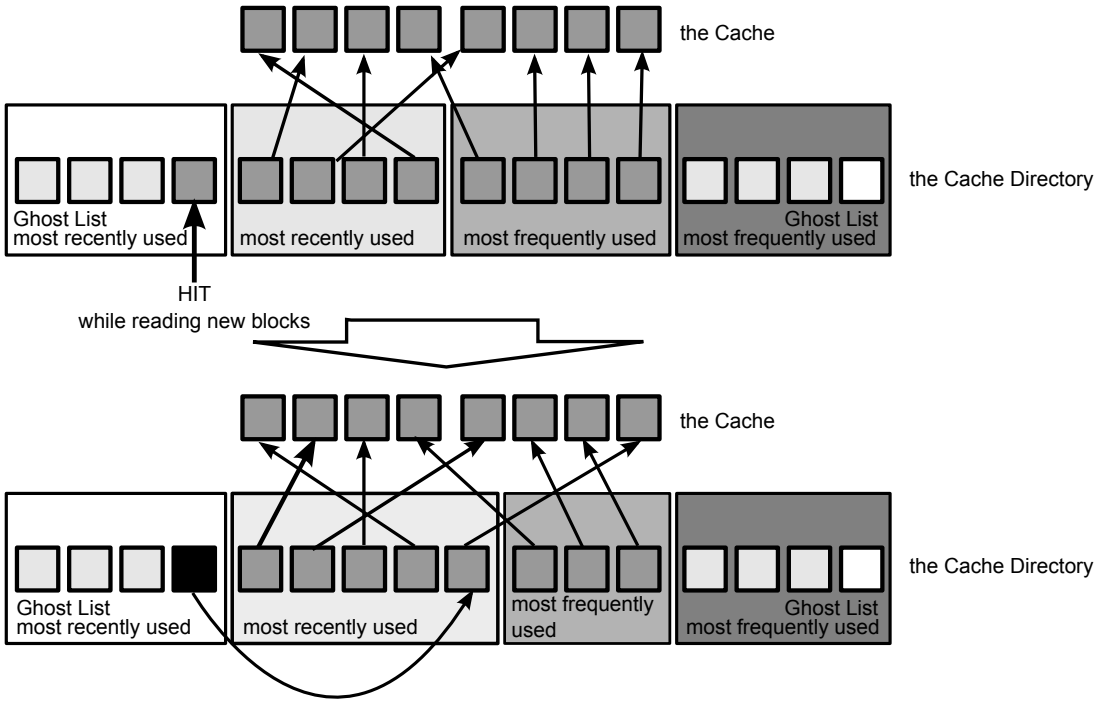


Figura 3.3: Funcionamento do algoritmo de substituição ARC [6].

para tratar os blocos *Most Frequently Used* (MFU, ou mais frequentemente usados) e outra para os *Most Recently Used* (MRU, ou mais recentemente usados). A Figura 3.3 apresenta a arquitetura básica do algoritmo.

Além disso, há duas listas adicionais na Figura 3.3, são as *Ghost lists*, uma para cada fila principal, MRU e MFU, respectivamente. Deste modo, mesmo que um bloco seja descartado da cache, este é enviado para as *Ghost list* e as suas informações históricas são mantidas. A ideia de guardar o histórico dos blocos descartados é que as suas informações são úteis para a tomada de decisões na ocasião do retorno do bloco à cache. Quando esses blocos retornam à cache ocorre um *phantom hit* (ou hit fantasma), e a decisão de colocar o bloco na MFU ou MRU é tomada com melhor precisão. Além disso, a lista MFU/MRU para a qual um novo *cache hit* é enviado é incrementada em um item na cache e a lista restante decrementada em um. Isso diferencia o ARC dos algoritmos anteriores e garante a sua adaptabilidade aos diversos tipos de *workloads* e possíveis mudanças ocasionais, aumentando ou diminuindo as suas listas de acordo com que os blocos são descartados ou novamente acessados.

Para reduzir *overhead* o ARC não realiza contagem de frequência nas filas MFU.

Ambas as filas MRU e MFU são construídas com LRU. As filas de recência (MRU) contêm blocos que foram acessados apenas uma vez desde que saíram das filas MRU/MFU. Similarmente, as filas de frequência (MFU) guardam todos os blocos que foram acessados mais de uma vez desde que saíram de MRU/MFU [31].

Por fim, o ARC apresenta-se como um moderno algoritmo de substituição para caches em geral que supera propostas anteriores em desempenho e/ou adaptabilidade. Quanto à complexidade, o ARC é constante, tendo desempenho próximo ao LRU [31]. O ARC realiza *auto-tuning*, e se adapta rapidamente aos diversos *workloads*, mostrando-se uma boa escolha para *workloads* com padrões não estáticos.

3.5 Multi-Queue (MQ)

O Multi-Queue foi desenvolvido essencialmente para caches de segundo nível (cache em SSD, por exemplo) de característica local — uma vez que toma a decisão de que página será substituída sem ter informações do primeiro nível de cache (cache em RAM, por exemplo). Uma vantagem dessa abordagem é que não são necessárias modificações na estrutura do primeiro nível de cache, como dos bancos de dados ou dos sistemas de arquivos, por exemplo [52].

A ideia principal deste algoritmo é manter blocos com diferentes frequências de acessos por diferentes períodos de tempo em uma cache de segundo nível, como SSDs ou similares. Para alcançar isso, o MQ mantém múltiplas filas LRU (de Q_0 a Q_m, sendo m configurável), conforme apresentado na Figura 3.4. Blocos armazenados em Q_j possuem mais tempo em cache que Q_i ($Q_j > Q_i$). Além disso, os blocos que são descartados da cache têm as suas informações de frequência e identificadores armazenados em um *history buffer*, cuja função é garantir que, mesmo que um bloco saia da cache, as suas informações históricas e portanto a sua importância para o sistema que o opera sejam mantidas para acessos futuros. O *history buffer* opera em FIFO, o que remove qualquer *overhead*.

A escolha da fila (Q_k) em que um novo bloco (b) será colocado é feita com base na função $k = \log_2 f$, onde f é a frequência de acesso ao bloco b, assim, este bloco será removido do *history buffer* e armazenado na sua fila correspondente. Por exemplo, o

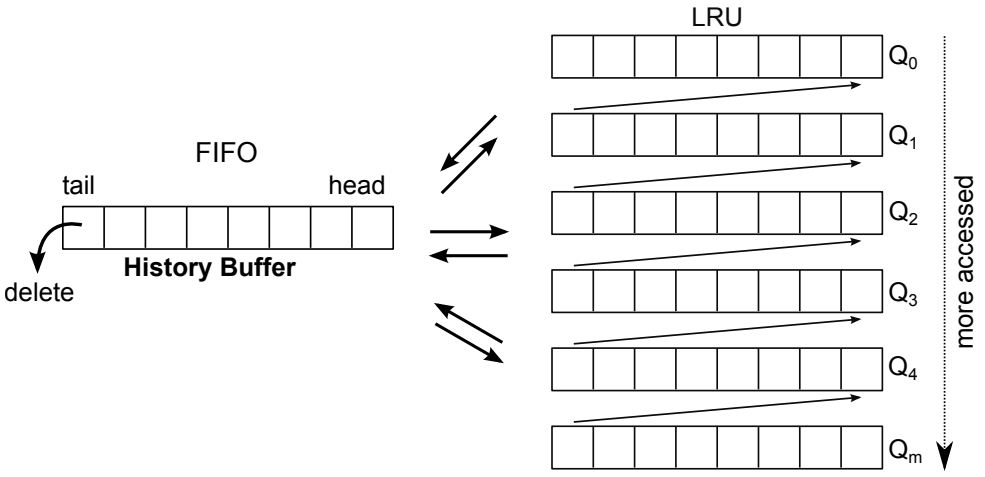


Figura 3.4: Funcionamento do algoritmo de substituição Multi-Queue.

oitavo acesso a um bloco que já se encontra na cache irá promover este bloco de Q_2 para Q_3 de acordo com a função. A escolha da vítima para substituição é feita a começar no bloco LRU de Q_0, caso esteja vazio, segue-se para de Q_1 e assim por diante. Após identificado o bloco vítima, este segue para o começo da fila do *history buffer*. Caso o bloco que está entrando na cache esteja no *history buffer*, será removido e inserido na fila Q_k correspondente e a sua frequência incrementada em 1; caso não esteja na cache, será inserido na fila Q_k com o valor de f iniciado em 1.

A demoção dos blocos nas filas é baseada em um valor de envelhecimento do bloco ($expireTime$). Este parâmetro é inicializado no momento que o bloco entra na cache sendo $expireTime = currentTime + lifeTime$. Todos esses valores são tempos lógicos com base na quantidade de acessos aos blocos. O tempo em que cada bloco pode permanecer em cache é o $lifeTime$, o qual é o único desses valores configurável. A cada acesso ao cache, $currentTime$ é incrementado e todos os LRUs das filas são verificados e, caso algum desses blocos tenha o $expireTime$ vencido, será movido para uma fila inferior.

Além disso, o algoritmo Multi-Queue tem a complexidade $O(1)$, considerando que todas as filas são LRU e que m possui valores geralmente baixos (menor que 10). Também possui simples implementação, sendo ainda mais rápido do que outros algoritmos como LRU-K, que têm complexidade $O \log_2 n$ [52].

3.6 Conclusão

Por fim, este Capítulo apresentou os principais algoritmos de substituição de cache atuais. O algoritmo OPT serve como linha de base de comparação de *hit cache* entre outros algoritmos. O LRU é um dos algoritmos mais utilizados em diversas implementações, pela sua baixa complexidade e eficiência. LIRS, ARC e MQ são algoritmos de substituição modernos que consideram não só recência, mas também frequência, sendo que apenas ARC e MQ são adaptativos. Desses algoritmos, apenas o LRU e MQ são utilizados para substituição de cache de segundo nível, os quais serão testados neste trabalho.

CAPÍTULO 4

IMPLEMENTAÇÕES DE CACHE NO LINUX

Neste Capítulo apresentaremos as principais características das duas implementações de cache de segundo nível no Kernel Linux. A Seção 4.1 apresenta o BCache, um algoritmo que opera diretamente na *Generic Block Layer* e faz uso do algoritmo LRU para substituição de blocos. A Seção 4.2 detalha o DMCache que, diferentemente do anterior, usa o algoritmo MQ para substituição e opera no *device-mapper* do Linux. A Seção 4.3 compara as duas implementações e a Seção 4.4 apresenta as conclusões deste capítulo.

4.1 BCache

BCache é uma implementação *block layer cache* para dispositivos de armazenamento secundário e permite que dispositivos mais rápidos, como SSDs, possam atuar como cache para um ou mais dispositivos lentos, como os discos rotacionais. Dessa forma, propõe-se à criação de um dispositivo virtual que abstraia do usuário final o mecanismo de cache entre os dispositivos de armazenamento secundário. Quanto aos sistemas de arquivos, BCache é chamado de *filesystem agnostic* (agnóstico ao sistema de arquivos), ou seja, visto que atua na camada de bloco do *kernel*, tudo acontece de forma transparente e independente do sistema de arquivos adotado [36].

A alocação e divisão dos blocos de dados no BCache é feita através de *buckets* (ou baldes). O *bucket* portanto é a unidade de alocação do BCache, normalmente alinhada com a unidade mínima de *erasing* do SSD subjacente. A estrutura de metadados do BCache é um híbrido entre árvore *btree* e estrutura de *log* (*log-structured*) [40]. Cada entrada do índice da árvore aponta para um *bucket* que contém a estrutura de *log* ordenada internamente com chaves que apontam para os blocos em disco.

O uso de *log structured* e *btree* proporciona ao BCache a eficiência das árvores balanceadas em conjunto com uma estrutura compacta. Além disso, por conta da estrutura de

log, o BCache é *copy-on-write* (cópia na escrita), o que beneficia os SSDs diante de dados sobrescritos, fazendo atualização ou remoção conjunta dos *buckets* e evitando em demasia escritas ao SSD.

Quanto à escrita, o BCache possui dois modos de operação que determinam como ocorre a sincronia entre os SSDs e os HDDs; estão disponíveis os modos *write-back* (padrão), *write-through*. O modo *write-back* escreve os blocos no *cache* e os descarrega no dispositivo subjacente quando os blocos estão "sujos" (*dirty*). O modo *write-through* escreve em ambos os dispositivos, confirmando cada escrita apenas quando completada no dispositivo subjacente. Além disso, o BCache pode trabalhar com *journal*, que confirma as escritas imediatamente escrevendo os metadados em um *buffer* circular antes de enviar à *btree*, enviando-as em *background* assim que possível. Isso alivia a pressão de inserções sobre a *btree* em situações de muitas escrita aleatória. O journal do BCache não tem propósito de consistência dos dados, apenas desempenho.

Por padrão, o BCache não faz cache de I/Os sequenciais, somente de leituras e escritas aleatórias, uma vez que os discos rígidos apresentam boa vazão em operações sequenciais e baixa vazão em operações aleatórias, devido a fatores como o *seek time* [39]. Neste ponto tanto o BCache quanto o DMCache diferem-se das abordagens tradicionais de algoritmos de caches do Capítulo 3, uma vez que, naqueles casos, o algoritmo substitui o bloco vítima assim que possível, independente de fatores externos como o *workload*.

A passagem direta de operações sequenciais é alcançada basicamente por limites ajustáveis de tamanhos de escritas e leituras contíguas. Além disso, o BCache mantém uma média das operações de I/O sequencial das tarefas e evita cache as operações dessas tarefas a partir de um limiar estabelecido, deste modo, *backup* e operações sequenciais em grandes arquivos não afetarão o cache. Por fim, o algoritmo de substituição dos blocos adotado pelo BCache é o LRU, semelhante ao exposto na Seção 3.2.

4.2 DMCache

Assim como o BCache, o DMCache [21] também visa criar volumes híbridos com SSDs e HDDs. O DMCache se diferencia do BCache em: (a) ele demanda ao menos três

dispositivos (ou partições), para *dados*, *caching* e *metadados*; (b) sua implementação usa a estrutura de mapeamento *device-mapper* [38], tornando sua implementação mais simples e menos intrusiva no núcleo; (c) as políticas de cache são implementados separadamente, como módulos configuráveis; (d) utiliza nativamente uma adaptação do algoritmo MQ em vez do LRU para substituição de blocos.

Em sua organização, o DMCache toma proveito da estrutura pré-existente do *device-mapper*, dessa forma não se preocupa com o mapeamento de diferentes dispositivos na *generic block layer*. No *device-mapper* a unidade de alocação são os *extends*, que podem ou não ter o mesmo tamanho de um bloco. Os metadados do DMCache contém informações de mapeamento entre o cache (SSD) e dispositivo de origem (HDD). A localização dos metadados no acesso aos dados, assim como no BCache, é implementada sobre árvores *btree*, no entanto não possui estrutura de *log* (*log-structured*) internamente aos nós *btree*. No caso do DMCache, cada nó de metadados contém uma unidade de mapeamento.

Na escrita, o DMCache possui três modos de operação: *write-back*, *write-through* e *write-around*. Os dois primeiros seguem os mesmos padrões já apresentados na Seção 4.1. O modo *write-around* usa apenas o dispositivo subjacente para escrita; é útil em casos de inconsistência nos dados e inserção de novos dispositivos de cache.

As políticas de cache definem como os dados migram entre os dispositivos SSD e HDD; estão disponíveis as políticas *cleaner* e *multiqueue* (padrão). A política *cleaner* apenas escreve todos os blocos sujos no dispositivo subjacente e cancela as escritas futuras no cache, o que é útil em casos de alterações nos dispositivos, como remoção ou troca de SSDs [47].

A política *multiqueue* implementa uma adaptação aproximada do algoritmo MQ (Seção 3.5), proposto por [52], implementando três conjuntos de 16 filas, sendo duas para o cache e uma para pré-cache (*history buffer*). Apresenta como principal diferença o pré-cache, que no *paper* usa FIFO e no DMCache usa outro conjunto de filas. O algoritmo MQ considera que, uma vez que continua existindo um primeiro nível de cache (RAM), o padrão de acesso aos dados no segundo nível de cache é diferente do primeiro nível, visto que são dados normalmente demovidos removidos do primeiro nível, dessa forma, mantém

diferentes filas com diferentes frequências de acesso aos dados [52].

Por fim, no DMCache todas as decisões adicionais são tomadas através das políticas. Parâmetros como limites para detecção de I/O sequencial, por exemplo, são implementados como complementos da política *multiqueue*. Essa política também é responsável por ajustes de parâmetros de contagem de acesso a blocos na leitura e escrita antes da promoção ao cache. No DMCache, por padrão, a leitura é mais priorizada do que a escrita, pois considera as limitações de escrita dos SSDs, contudo essas políticas são ajustáveis em tempo de execução.

4.3 Comparativo

A Tabela 4.1 apresenta uma comparação entre as implementações do BCache e do DM-Cache quanto aos seus principais aspectos, que inclui desde o algoritmo de substituição até os modos de escrita adotados por esses algoritmos.

	BCache	**DMCache**
Algoritmo de substituição	LRU	MultiQueue
Dispositivos envolvidos	2 (SSD=cache+metadados, HDD=dados)	3 (SSD=cache, SSD=metadados, HDD=dados)
Estrutura de metadados	Híbrida: BTree, Log-Structured	BTree (aproveita implementação do DM)
Localização	Generic Block Layer	Device Mapper
Modos de escrita	write-back, write-throuth	write-back, write-throuth, pass-throuth
Bypassing de I/O sequencial	sim	sim
Políticas modulares	não	sim
Journaling	sim	não

Tabela 4.1: Comparativo entre BCache e DMCache.

4.4 Conclusão

Neste Capítulo apresentamos as implementações de cache de segundo nível (SSD) para o Linux. O BCache é uma implementação que utiliza *btree* para a sua estrutura de metadados e LRU para substituição dos blocos, mostrando-se uma implementação simples

para uso geral. O DMCache utiliza a estrutura do *device-mapper* na sua organização e MQ para a substituição dos blocos em cache, o que sugere um algoritmo mais complexo para uso em *workloads* mais específicos. Ambos os algoritmos serão testados neste trabalho, buscando-se identificar a melhor aplicação de cada implementação para as cargas de trabalho utilizadas nos testes.

CAPÍTULO 5

ANÁLISE COMPARATIVA DE ALGORITMOS DE CACHING DE SEGUNDO NÍVEL

Diante do que foi apresentado nos capítulos anteriores, é possível observar uma variedade de implementações, parametrizações, algoritmos, arranjos e dispositivos envolvidos no armazenamento de dados, além de diversas cargas de trabalho sobre as quais todos essas configurações podem sofrer influência. Isso suscita diversas questões, em especial sobre qual combinação de itens melhor se relaciona com cada *workloads* e os porquês.

Levando em consideração que o Linux é um dos sistemas operacionais mais utilizados no mundo, especialmente em servidores, dentre diversas implementações de cache com SSDs, o BCache e o DMCache foram as escolhidas pela comunidade que administra o sistema operacional para integrarem o kernel do sistema, seja pela estabilidade ou pelas diferenças de ambas, o que dá aos administradores de sistemas duas interessantes possibilidades de escolha. Assim, essas recentes implementações possuem objetivos comuns com abordagens distintas, principalmente no algoritmo de substituição de blocos e em sua arquitetura dentro do núcleo do sistema operacional, uma na camada genérica de blocos e outra junto ao *device-mapper*.

Este trabalho realiza uma análise experimental desses fatores no cenário de armazenamento de dados, restringindo-se em especial ao sistema operacional Linux, aos arranjos híbridos BCache e DMCache, e aos antigos arranjos RAID. O Linux foi adotado por ser um dos sistemas operacionais mais utilizados atualmente para servidores. BCache e DMCache são as duas atuais implementações para caches de segundo nível (SSD). RAID foi incluído nos testes por ser uma tecnologia consolidada, a qual pode justificar ou não o uso de caches de segundo nível para determinadas cargas de trabalho. Durante o trabalho, discutiremos melhores práticas para o uso de SSDs e HDDs isoladamente, hierarquicamente, ou com uso de arranjos RAID, através de experimentações específicas sobre diversos *workloads*

conhecidos.

5.1 Objetivos

Este trabalho tem por objetivos:

- Analisar o desempenho das implementações de cache de segundo nível BCache e DMCache para o Kernel Linux;
- Verificar as reais vantagens e desvantagens do uso de discos híbridos (cache SSD) frente a outras abordagens, como o uso de discos redundantes e espelhados (Seção 2.1.3);
- Apontar melhores caminhos na escolha dos arranjos de discos e tecnologias;
- Identificar falhas e/ou possíveis otimizações para os algoritmos e/ou parametrizações inerentes às duas abordagens.

5.2 Metodologia

Os testes serão realizados por meio de *microbenchmarks* e *macrobenchmarks*. Os *microbenchmarks* serão utilizados de modo a traçar uma linha de base comportamental dos algoritmos em cargas de trabalho extremamente controladas, como leitura e escrita e padrões de acesso sequencial e aleatório. Assim, os *microbenchmarks* darão maior clareza à análise dos *macrobenchmarks*. A ferramenta utilizada para os testes de *microbenchmarks* será o *fio* [11], que possibilita a emissão de diversos padrões de acesso de I/O com diferentes parâmetros de forma customizada e flexível. Na Seção 6.2 apresentaremos mais detalhes a respeito das cargas de teste executadas nos diferentes cenários.

Para analisar o comportamento dos arranjos e dispositivos em situações próximas às encontradas em ambientes reais, uma sequência de *macrobenchmarks* será executada. Neste caso, utilizaremos a ferramenta de testes *Filebench* [10], por ser amplamente utilizada e conter diversos *workloads* pré-definidos, simulando o padrões reais de acesso em

servidores com diferentes finalidades. Os *workloads* que serão utilizados nos experimentos foram Fileserver, Varmail e OLTP. Mais detalhes sobre os macrobenchmarks serão apresentados na Seção 6.3.

Além de diversas combinações de parâmetros nas ferramentas de testes, serão testados frente aos mecanismos de cache, arranjos RAID, especialmente o nível 0 (*striping*) e 10 (espelhamento e *striping*), com o intuito de verificar o real ganho das modernas abordagens de cache SSD frente a uma tecnologia bastante conhecida e consolidada entre os administradores de sistema. Um validação mais específica desses experimentos, por exemplo, é verificar com quantos discos rígidos em RAID0 ou RAID10 podemos alcançar o mesmo desempenho e confiabilidade de arranjos similares com o uso de mecanismos de cache SSD (BCache e DMCache). Nesse sentido, a Seção 6.4 apresenta um estudo de caso com análise econômica para SSDs e HDDs em arranjos similares e compara valores monetários médios atuais e performance desses dispositivos.

Os resultados obtidos serão analisados por meio de observações gráficas dos resultados dos experimentos sobre diferentes padrões de configuração seguindo padrões de análise de benchmarks similares aos apresentados em [46] e [48]. Todos os casos de testes seguirão uma quantidade de execuções considerável de modo que possam gerar intervalos de confiança aceitáveis. Casos imperceptíveis nos gráficos serão analisados numericamente através de tabelas e numeração específica no topo dos gráficos e explicados junto ao gráfico. Casos mais particulares serão analisados em detalhes nos *logs* e/ou através de outras informações extraídas do sistema operacional e/ou da ferramenta de coleta e benchmarks utilizada. Mais detalhes sobre cada experimento serão apresentados nas seções 6.2 e 6.3.

Após analisados os resultados, este trabalho será capaz de propor melhores práticas para uso das unidades de armazenamento modernas (SSDs) e tradicionais (HDDs) isoladamente e/ou em RAID, identificar casos do mau uso das tecnologias analisadas e apontar melhorias nas implementações de cache em SSD.

5.3 Trabalhos Relacionados

Apesar da existência de diversas implementações e algoritmos de cache de primeiro nível, para o segundo nível de cache apenas duas implementações foram adotadas pelo Kernel Linux, juntamente com os seus algoritmos de substituição principais. Além disso, geralmente implementações recentes desses algoritmos não são construídas para sistemas reais de armazenamento, levando-se em consideração as suas particularidades, conforme mostramos nos capítulos anteriores. Normalmente são realizados testes em simuladores, sendo medido apenas os *hits I/O* (métrica nem sempre válida para avaliação de desempenho), já que o desempenho depende muito do tamanho das requisições de I/O e dos padrões de acessos diante das características dos dispositivos de armazenamento. A seguir, são discutidos os trabalhos relacionados de maior relevância.

Em [52], conforme mostrado na Seção 3.5, é apresentado um novo algoritmo de cache específico para segundo nível, o Multi-Queue (MQ), que independe do uso de SSDs para este fim, o que significa que pode trazer vantagens quando aplicado a servidores Web, ambientes virtualizados, serviços de armazenamento em rede, como NFS, CIFS ou Samba, todos serviços que fazem uso de alguma forma de cache de segundo nível. O MQ tem baixa complexidade, $O(1)$, e mantém uma organização constante da cache, de forma a reter blocos mais importantes na cache por mais tempo. Como limitação, os resultados apresentados no trabalho utilizam *traces* proprietários e tamanhos de cache muito reduzidos, chegando a no máximo 1GB para caches de nível um e 2GB para cache de nível dois, o que difere substancialmente da proposta apresentada, onde os tamanhos das caches são linearmente maiores e próximos dos servidores atuais, conforme se aproxima do dispositivo de origem do armazenamento.

No *Adaptive Replacement Cache* (ARC) [31], semelhante ao caso anterior, os autores se limitam a testes em cache de primeiro nível e os resultados são apresentados somente com referências a *hits I/O* para *workloads* basicamente sintéticos, não refletindo o uso em ambientes reais, como em situações de *stress* ou a análise de parâmetros como uso de CPU, vazão, latência, IOPS e memória. O tamanho do cache em todos os testes também não acompanha as expansões dos servidores atuais, não se provando a escalabilidade dos

testes e o comportamento frente às questões de armazenamento do presente. O ARC é implementado para o sistema operacional Oracle Solaris [32], tanto para caches de primeiro nível (em memória principal) quanto para caches de segundo nível (com SSDs), contudo não será analisado e comparado neste trabalho por não haver implementações abertas para Linux e, ao se analisar tal comportamento frente ao SO, essa comparação seria parcial por conta de características peculiares a cada sistema operacional.

Em [5], os autores apresentam uma técnica de cache secundário com SSDs entre disco e memória principal que monitora padrões de acesso a disco e identifica blocos mais acessados de acordo com a região do disco em que se encontram, chamando essa região de *região quente*. Com o passar do tempo essas regiões populam o cache e retiram os dados pouco acessados. Foram realizados experimentos sobre um protótipo com *traces* de I/O do banco de dados DB2, apresentando substanciais ganhos de desempenho. Contudo, [5] é de uso específico, se aplicando estritamente a bancos de dados, ao passo que as abordagens estudadas neste trabalho são de uso geral.

Em [19] são estudados os impactos dos tamanhos de caches, gerenciamento de metadados e fluxos de dados sobre diversas configurações hierárquicas de armazenamento, incluindo memória RAM, SSDs e discos magnéticos. O autor propõe um algoritmo para otimizar a ordenação de dados armazenados hierarquicamente. Adicionalmente, os esforços desse trabalho são para otimizações relacionadas a bancos de dados que estão sobre armazenamento hierárquico.

Em [14] é realizada uma comparação entre as implementações de cache em SSD, BCache e FlashCache, apresentando análises limitadas a microbenchmarks. Não ficam claras questões como aquecimento de cache e caracterização dos workloads. Além disso, o *bypassing* de I/O sequencial foi desabilitado, dessa forma o BCache e o FlashCache não funcionam como subsistemas híbridos como seriam naturalmente. Por fim, em [22] o desempenho do DMCache é avaliado comparando ambientes físicos e virtualizados. No entanto o *workingset* é limitado e faltam informações sobre os workloads apresentados.

CAPÍTULO 6

RESULTADOS EXPERIMENTAIS

Neste Capítulo apresentaremos os resultados obtidos de forma experimental para diversos padrões de configuração escolhidos, de forma a obter uma linha geral sobre cargas de trabalho comuns ou similares às analisadas. Na Seção 6.1 apresentamos a configuração do ambiente de hardware e software utilizado nos testes. Na Seção 6.2 apresentamos os microbenchmarks, que servirão como linha de base para a interpretação dos macrobenchmarks, apresentado na Seção 6.3. Por fim, a Seção 6.4 traz uma análise econômica de custo/benefício dos diversos arranjos, diante dos resultados apresentados.

6.1 Ambiente de Testes

Durante todos os experimentos utilizamos um computador com processador Intel(R) Xeon(R) CPU E5620 2.40GHz e memória RAM de 12GB. Os discos magnéticos utilizados nos testes foram Hitachi HUA723030ALA640 2TB, 7.200 RPM, 64MB de cache interna, interface SATA 6Gb/s, taxa de transferência sustentada de 157 MB/s e *random seek time* de $8.2ms$ [50]. Como unidade de cache, utilizamos SSDs Intel (R) 320 Series SSDSA2BW160G3 160GB, interface SATA 3.0, 3 Gb/s, 39.000 IOPS de *random read* e 21.000 IOPS de *random write* para alcançe de 8 GB e 600 IOPS para alcançe de 100% do dispositivo [15]. Todos os discos são conectados a uma controladora MegaRAID(R) SAS 8708ELP, com 3Gb/s e cache de escrita interna DDR II SDRAM de 128MB. Nos microbenchmarks todas as operações foram realizadas com a *flag* O_DIRECT – o que evita o uso de caches do sistema operacional. Para todos os casos de testes, inclusive *macrobenchmarks*, a memória foi limitada a 20% do tamanho do cache SSD através da diretiva de inicialização *mem* do kernel do Linux. As versões de software utilizadas (geralmente coletadas pela *flag* *–version*) foram: Kernel Linux 3.17.3 (*commit* 7623e24), Systemtap 2.6/0.159 [16], GCC 4.9.2, IOstat/SysStat 10.0.5, Filebench 1.4.9.1, Fio 2.0.8 e Debian

GNU/Linux Wheezy 7.6.

6.2 Microbenchmarks

Nesta Seção analisaremos resultados referentes aos *microbenchmarks*. A intenção dos *microbenchmarks* é identificar padrões de comportamentos dos dispositivos HDD, SSD, RAID e cache que possam servir de base para a análise dos *macrobenchmarks*. A Seção 6.2.1 detalha o ambiente lógico de testes, parametrizações necessárias e explica como foram coletadas as métricas para esses workloads. Como testes sobre cache pressupõem que os dados podem sofrer bastante variação até alcançar estabilização que possibilite a coleta, na Seção 6.2.2 analisamos o aquecimento do cache que ocorre antes da coleta dos resultados. Na Seção 6.2.3 analisamos os resultados finais dos microbenchmarks.

6.2.1 Ambiente dos Microbenchmarks

Para a execução dos *microbenchmarks*, utilizamos a ferramenta *fio*, um artefato completo para testes de *stress* e verificação de hardware, projetado especialmente para dispositivos de armazenamento [11]. O *fio* possui suporte a uma série de *engines* de I/O (*sync, mmap, libaio, posixaio, SG v3, splice, null, network, syslet, guasi, solarisaio*), taxas de I/O, *jobs* configuráveis (*forked* ou *threaded*), entre outras parametrizações bastante flexíveis. Durante os experimentos, o *fio* foi configurado da seguinte forma:

- *ioengine: sync* – ideal para requisições síncronas aos dispositivos de I/O. Garante a ordem das requisições.
- *direct: true* – utilizado para que operações *open* sejam realizadas com a *flag* O_DIRECT, o que evita o uso de caches tanto quanto possível. Permite uma verificação de desempenho mais próxima dos dispositivos de armazenamento com pouca intervenção do sistema operacional.
- *size:* $2 \times ssd\_cache\_size$ – parametrizado em duas vezes o tamanho do cache SSD. Equivale à quantidade de dados lidos ou gravados e também ao espaço de dados

tocados no dispositivo físico. Este é um tamanho mínimo necessário para que os testes sobre o cache tenham validade e não haja 100% de *hits* [17].

- *loops: 10* – cada rodada é executada dez vezes sobre o mesmo conjunto de dados. Neste caso, o *size* é varrido *loops* vezes durante cada execução completa do fio. Isso ajuda a aumentar as repetições de acesso ao mesmo conjunto de dados, gerando repetições que permitam a atuação do cache.

- *randomrepeat: true* – permite que haja repetições de acesso ao mesmo bloco dentro de *workloads* aleatórios. Se esta opção estivesse desabilitada o *fio* executaria todo o *workload* sem nenhuma repetição, o que não serve para testes de cache.

- *bs: 64k, 512k, 4m* – corresponde ao tamanho do bloco utilizado nas operações de I/O. Este parâmetro é equivalente ao tamanho das requisições de I/O.

Neste texto adotamos o termo *dispositivo de origem* para o dispositivo subjacente de onde os dados se originam (os HDDs); para o cache SSD é usado o termo *dispositivo de cache* e para o arranjo que envolve o dispositivo de cache e o dispositivo de origem é usado o termo *arranjo de cache*.

Os resultados estão divididos entre operações de leitura e escrita com padrão de acesso aleatório (os testes foram executados apenas para *workloads* aleatórios, uma vez que o caching de acesso sequencial é evitado pelo BCache e DMCache). Em todos os casos de teste, o arranjo de cache teve seu tamanho ajustado em 30 GB, correspondente a 20% do dispositivo de origem, com tamanho ajustado em 150 GB. A quantidade de memória RAM foi ajustada com 20% do tamanho do cache, ou seja, 6 GB, o que facilita a aferição dos resultados com a mínima interferência dos caches em memória. A proporção de 20% foi escolhida por ser uma relação de uso comum em benchmarks recentes para as implementações de cache testadas [23, 14]. O tamanho dos dispositivos físicos foi ajustado por particionamento simples. Nos casos de arranjos com RAID, cada membro do arranjo RAID recebeu uma redução proporcional.

O ajuste do tamanho dos dispositivos foi feito através de particionamento comum, através da tabela de partições do sistema operacional. Nos casos de arranjos que envol-

vessem RAID, cada partição membro do arranjo RAID recebia redução proporcional. Por exemplo, para a criação de um RAID0 de quatro SSDs que seriam utilizados para cache, criamos uma partição de 7,5 GB em cada SSD e apresentamos ao RAID, totalizando 30 GB. O objetivo dessas divisões é a manutenção dos mesmos tamanhos finais para dispositivo de cache, de origem, do arranjo de cache e manter equilibrada a proporção de espaço pertencente ao arranjo RAID em cada dispositivo, uma vez que o propósito dos testes é medir prioritariamente desempenho.

Arranjo de cache	**# dispositivos**		**Nível RAID**		**Mecanismo de cache**
	SSDs	**HDDs**	**SSDs**	**HDDs**	
1h	-	1	-	-	-
1s	1	-	-	-	-
4h0	-	4	-	raid0	-
4s0	4	-	raid0	-	-
4h10	-	4	-	raid10	-
dmcache-1s-1h	1	1	-	-	dmcache
bcache-1s-1h	1	1	-	-	bcache
dmcache-1s-4h0	1	4	-	raid0	dmcache
bcache-1s-4h0	1	4	-	raid0	bcache
dmcache-1s-4h10	1	4	-	raid10	dmcache
bcache-1s-4h10	1	4	-	raid10	bcache
dmcache-4s0–4h0	4	4	raid0	raid0	dmcache
bcache-4s0-4h0	4	4	raid0	raid0	bcache
dmcache-4s0-4h10	4	4	raid0	raid10	dmcache
bcache-4s0-4h10	4	4	raid0	raid10	bcache

Tabela 6.1: Composição dos arranjos utilizados nos testes.

Para os testes foram selecionados 15 arranjos distintos envolvendo HDDs e SSDs, RAID0 e RAID10, DMCache e BCache, escolhidos por serem os mais representativos de situações distintas. A Tabela 6.1 apresenta a composição dos arranjos escolhidos. O nome dos arranjos reflete sua estrutura: por exemplo, o nome "*4h0*" indica um arranjo com 4 HDDs em configuração RAID0, sem mecanismo de cache. Já o nome "*bcache-4s0-4h10*" indica um arranjo de 4 SSDs em RAID0 e 4 HDDs em RAID10, usando o mecanismo *BCache*.

HDD e SSD puros foram testados com o intuito de servirem de linha de base para análise dos demais resultados. Dentre os níveis de RAID existentes, RAID0 e RAID10 foram adotados por serem bastante utilizados em ambientes reais e por representarem

bom desempenho e redundância. Como SSDs são usados apenas como cache dos HDDs, não experimentamos SSDs em RAID10 nos arranjos de cache.

Os dados dos experimentos foram coletados através da sumarização apresentada pelo *Fio* e por ferramentas auxiliares, como *iostat* [26] (informações dos dispositivos de I/O do sistema), *SysFS* (dados do BCache) e *dmsetup* [25] (informações do DMCache). Casos que se fizeram necessários, como na análise do aquecimento do cache (Seção 6.2.2), utilizamos o envio de mensagens ao *dmsetup* [25], que retorna informações de *hits*, *demotions*, *promotions*, entre outros ao sobre o DMCache, e o sysfs (*/sys*) que permite a coleta de informações do BCache.

Em todos os testes executamos 10 rodadas internas do *fio* com repetições de acesso (*randrepeat = true*) a blocos sobre o mesmo conjunto de dados. O conjunto de dados de cada execução excedia duas vezes o tamanho dos caches SSD, para que o cache pudesse oferecer algum efeito ao desempenho [17]. Uma rodada de aquecimento de 1.200s foi executada antes de cada execução, sendo descartada nos resultados finais. Mais detalhes sobre o aquecimento são apresentados na Seção 6.2.2.

Cada teste foi executado de 5 a 15 vezes (mais as 10 repetições internas sobre o mesmo conjunto de dados), parando numa rodada intermediária caso os resultados já obtidos apresentassem um coeficiente de variação inferior a 5% ($c_v = s/\bar{x}$). Na próxima Seção (6.2.2) apresentamos como foram realizadas as rodadas de aquecimento e, após isso, seguimos para a análise dos resultados finais dos microbenchmarks na Seção 6.2.3.

6.2.2 Rodadas de Aquecimento

Antes de extrairmos os resultados, realizamos rodadas de aquecimento com o objetivo de observar o que acontece nos arranjos de cache antes e depois da estabilização de performance e determinar com melhor precisão quando e como a estabilização acontece. Isso permitirá uma coleta mais justa e precisa dos resultados, já que essa rodada será removida dos testes comparativos finais e não ocorre na maior parte do tempo em ambientes reais. Para avaliação desses gráficos, além dos resultados sumarizados do *fio*, utilizamos métricas da ferramenta *iostat* [26], que coleta, além de outras informações, medições de

CPU, trocas de contexto, latência, IOPS e vazão para cada dispositivo de forma ativa durante todas as cargas de trabalho. A Figura 6.1 apresenta algumas nomenclaturas de fluxos de I/O que utilizaremos durante a análise dos resultados.

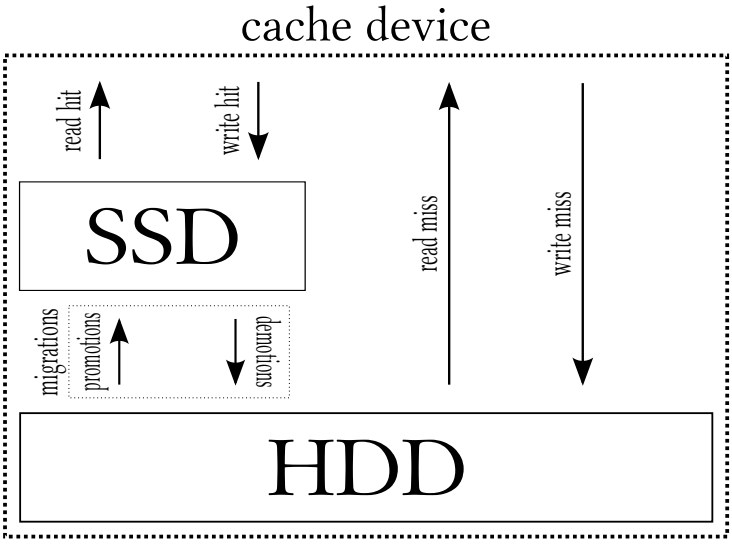
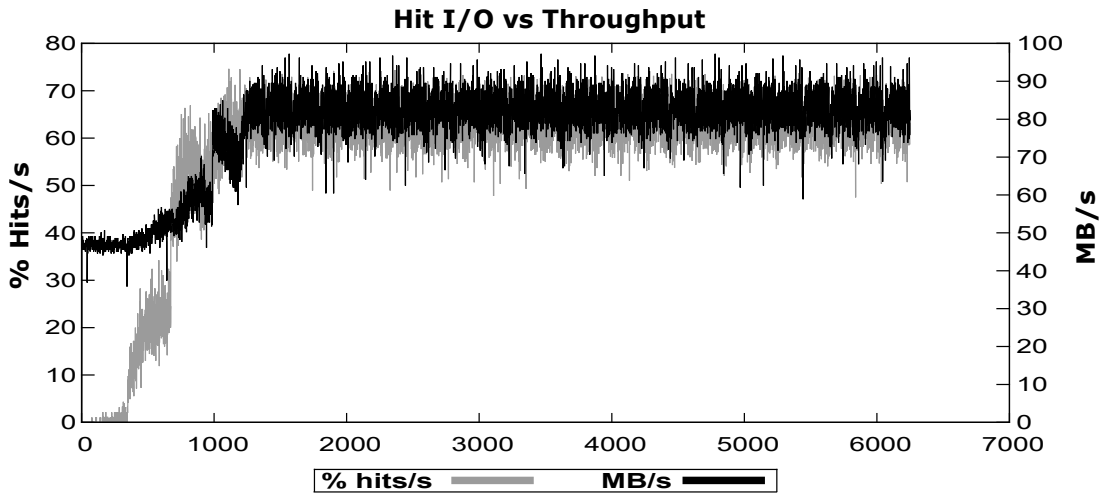


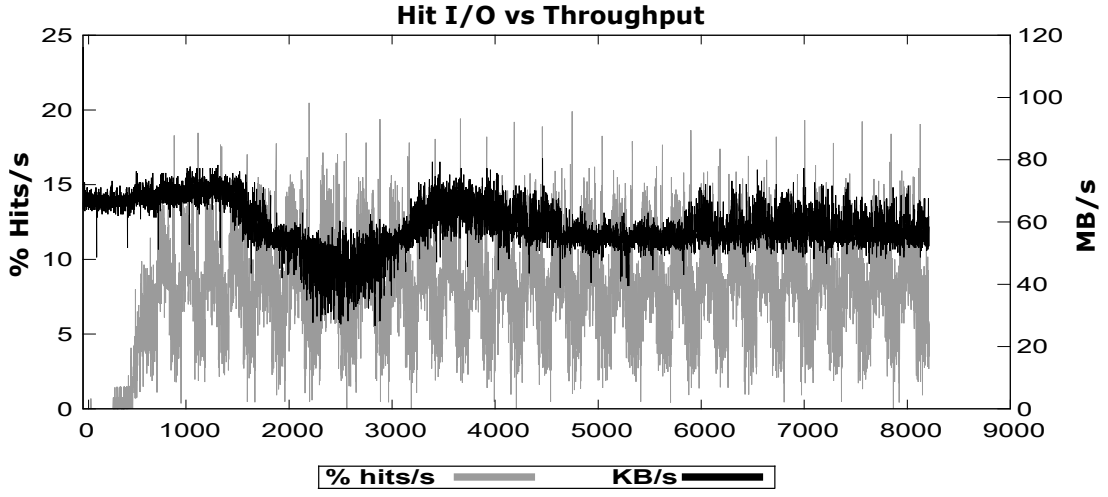
Figura 6.1: Fluxos de operações de I/O realizadas nos dispositivos de cache DMCache e BCache.

A Figura 6.2 apresenta os resultados de leitura e escrita aleatória para o algoritmo DMCache. Para a leitura aleatória é possível observar um crescimento linear na taxa de *hits* (acertos na cache) que vai do começo do *workload* até aproximadamente 1200s, após isso, tanto a taxa de *hits* quanto a vazão são estabilizadas para este *workload*. Dentre os resultados apresentados, este é o *workload* que apresenta melhor comportamento após a estabilização. A taxa de *hits* tem completa relação com a vazão, o que pode ser percebido de forma clara na Figura 6.2(a).

O início deste *workload*, ou seja, a fase de aquecimento, é caracterizada por leituras ao HDD, numa vazão média de 50 MB/s, distribuídas entre entrega (*fio – seek*, *read*) e migração (*promotions* e *demotions*). Junto a essa leitura ocorre paralela escrita ao cache SSD (*promotions*), numa vazão média 10 MB/s, soma-se a isso também escritas de 1 MB/s à partição de metadados do SSD. Ainda no aquecimento, a vazão leitura do HDD diminui e aumenta a vazão de leitura do SSD e, com isso sobe a vazão global, alcançando 90 MB/s na estabilização, sendo uma soma de (30 MB/s SSD e 60 MB/s HDD). A partir daí os resultados permanecem estáveis até o final da execução.



(a) Leitura Aleatória - DMCache



(b) Escrita Aleatória - DMCache

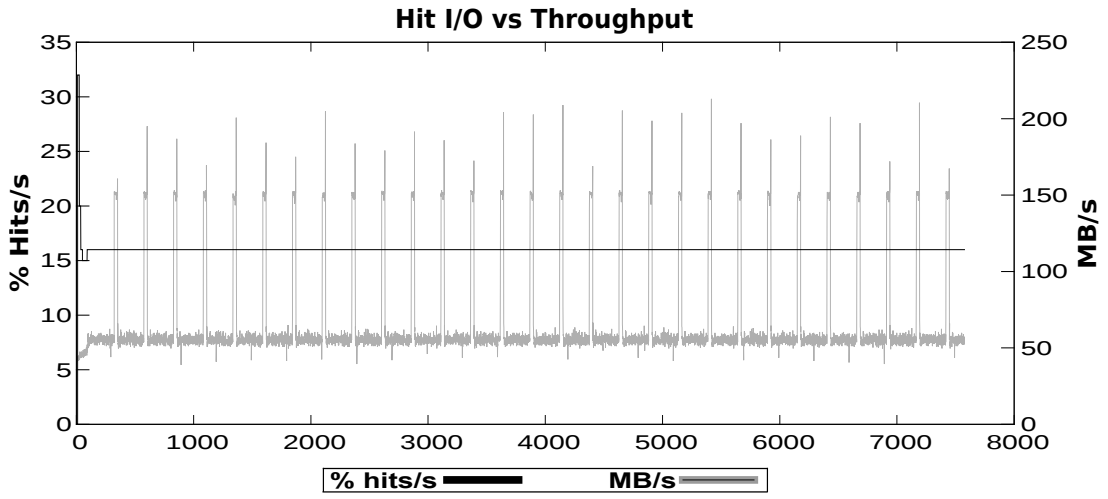
Figura 6.2: Leituras e escritas aleatórias para o algoritmo DMCache. As barras verticais indicam a taxa de acertos em caches (esquerda) e a vazão (direita). A barra horizontal indica o tempo em segundos.

A Figura 6.2(b) apresenta o desempenho do DMCache sobre acesso de escrita aleatória. Assim como na leitura, a escrita tem um aquecimento aproximado de 1200s. Entretanto, o desempenho é 50% inferior em relação à vazão de leitura após estabilização. A taxa de migração (*demotion*) do SSD para o HDD neste *workload* foi de 2 MB/s após estabilidade. O vazão do SSD teve baixa variação durante todo o workload, $\approx$ 4 MB/s, e a vazão do HDD variou mesmo após aquecimento do cache. O resultado final consistiu na soma da vazão do SSD com a do HDD, alcançando uma soma de 55 MB/s. O resultado deste experimento foi acompanhado principalmente pelo desempenho do HDD, com pouca influência do SSD.

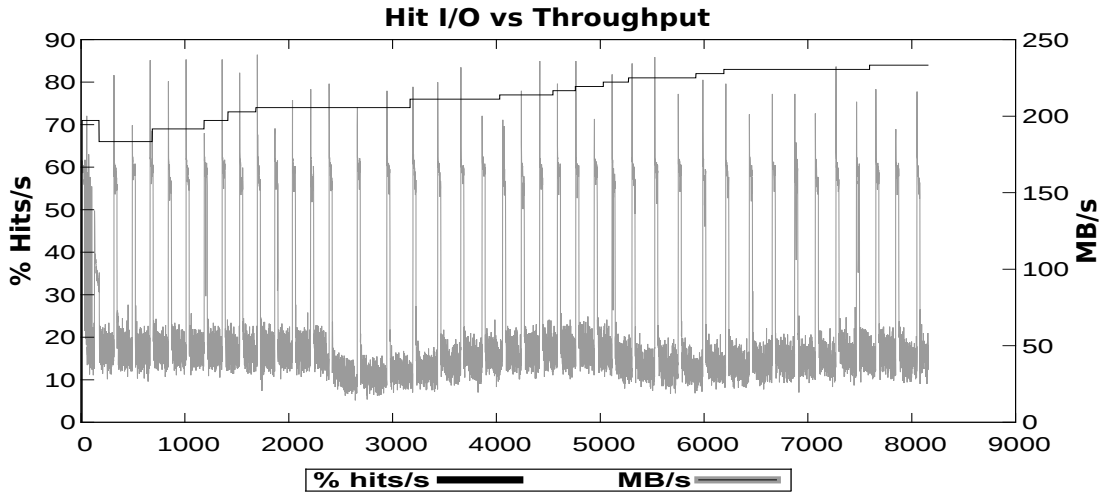
A razão para a baixa taxa de *hits* (10%) deve-se ao *tuning* do parâmetro *write_promote-_threshold* do DMCache, que tem a função de contar os acessos/alterações aos blocos antes de promovê-los. Neste caso de teste, esse parâmetro seguiu a configuração padrão (8 acessos), o que faz com que dados pouco acessados quase não subam para o cache. A razão desta configuração do DMCache é para não penalizar os SSDs na escrita, já que este dispositivo tem desempenho inferior à leitura por problemas de *write-amplification* e limite físico de escritas por página. Outro fator que influencia neste resultado é que informações históricas são zeradas quando blocos são demovidos do cache e, portanto, não são melhor aproveitadas posteriormente.

Os resultados de leitura e escrita aleatória para o BCache são apresentados na Figura 6.3. Para a leitura o BCache apresentou um rápido aquecimento com *promotions* até os primeiros 100s, não promovendo nem demovendo mais nenhum bloco até o final do teste. O desempenho final, assim como nos casos do DMCache foi realizado pela soma da vazão do SSD (27,7 MB/s) e HDD (37,8 MB/s). A medição de *hits* do BCache não apresentou confiança aos resultados, uma vez que não representa proporcionalidade quanto à vazão ou IOPS. Além disso, de acordo com o código fonte do BCache, a coleta de informação dos *hits* ocorre 22 vezes a cada 5 minutos (a cada 13,63s), o que é um intervalo grande e assíncrono em relação à coleta da vazão (a cada 1s). Mesmo com a alteração da contagem de *hits* para a cada 1s essa métrica não apresentou resultados com a devida correspondência.

Assim como o DMCache, o BCache mostrou um comportamento bem diferente da



(a) Leitura Aleatória - BCache



(b) Escrita Aleatória - BCache

Figura 6.3: Leituras e escritas aleatórias para o algoritmo BCache. As barras verticais indicam a taxa de acertos em caches (esquerda) e a vazão (direita). A barra horizontal indica o tempo em segundos.

leitura para os testes de escrita aleatória, porém com suas particularidades. A maior parte da execução se deu no HDD, o que fez com que a onda da vazão da Figura 6.3(b) fosse semelhante à apresentada pelo DMCache (Figura 6.2(b)). No entanto, o BCache realizou uma alta taxa de promoções (diluída entre escritas reais e de migração – SSD-*write* – 34,7 MB/s) e demoções (*read*-SSD – 25,8 MB/s) entre HDD e SSD. O que se ressalta neste *workload* é que o desempenho líquido do arranjo de *cache* foi, em geral, inferior ao desempenho total do HDD (real e em *background*), à exceção de picos de *hits* no SSD. Essa "anomalia" ocorre por conta da grande quantidade de demoções do SSD para HDD, o que aumenta a vazão bruta do disco rígido mas não o desempenho do arranjo de cache.

Após a análise do aquecimento, identificamos que o pior caso de estabilização ocorria após 1.200s. Dessa forma, adotamos esse tempo de aquecimento como padrão para início da coleta dos dados nos microbenchmarks.

6.2.3 Análise dos Microbenchmarks

Nesta Seção analisaremos os resultados dos *microbenchmarks* apresentados nas Figuras 6.4 e 6.5. Essas figuras apresentam, respectivamente, os resultados de leitura e escrita aleatória para os casos de teste já citados na Tabela 6.1. Inicialmente analisaremos o comportamento dos HDDs e SSDs isolados, com a intenção de medir a menor unidade que compõe o desempenho dos arranjos. Depois analisaremos o desempenho dos arranjos RAID e posteriormente o impacto desses dispositivos nos arranjos de cache, BCache e DMCache.

Leitura Aleatória

A Figura 6.4 traz os resultados de vazão para leitura aleatória, com todos os tipos de arranjos testados. Barras muito elevadas foram truncadas, com seu valor numérico indicado no topo. O intervalo de confiança (usando a distribuição *t-student*) é indicado no topo de cada barra, na mesma escala. Cada caso de teste mostra três barras, com os os tamanhos das requisições de dados (64KB, 512KB e 4MB).

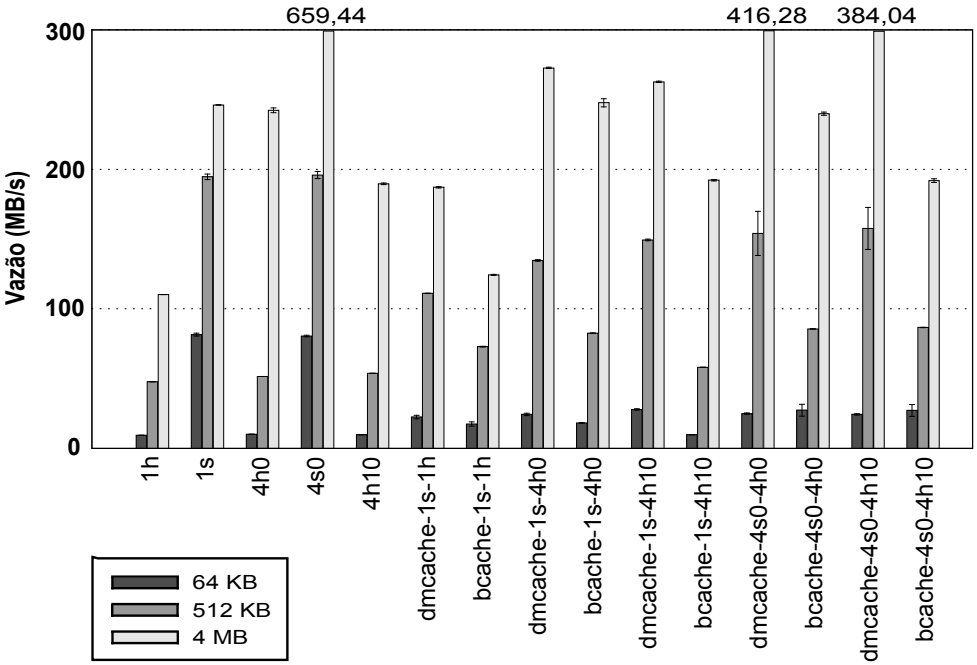


Figura 6.4: Microbenchmark de Leitura Aleatória.

Para leitura aleatória, analisando-se os cinco primeiros casos de testes que envolvem HDDs, SSDs, RAID0 e RAID10, para tamanhos de bloco de 64KB e 512KB, observa-se o mesmo desempenho entre os arranjos *1h*, *4h0* e *4h10*; o mesmo ocorre entre os arranjos *1s* e *4s0*. Isso ocorre porque o tamanho padrão do *chunk* de dados do RAID em software do Linux é de 512KB, ou seja, requisições menores ou iguais a 512KB não são paralelizadas. Como todas as requisições deste caso de testes são aleatórias, o *workload* não se beneficia de *merges* de requisições no escalonador de disco nem de mecanismos de *read-ahead* do sistema de arquivos. O mesmo comportamento se observa nos dispositivos de cache usando o DMCache, para blocos menores ou iguais ao *chunk* do RAID, por exemplo. Numa análise similar, o DMCache apresentou uma variação média na vazão de 7,9% nos casos com blocos de 64KB, 13,5% com blocos de 512KB, mas chegou a 30,9% com blocos de 4MB (nos quais o RAID têm influência).

Apesar disso, diversos casos de leitura aleatória com arranjos de cache obtiveram desempenho superior aos arranjos de HDDs em RAID. Por exemplo, *dmcache-1s-1h* teve desempenho 2,2× superior ao *4h0* e foi 2,3× melhor que *4h10* para 64KB e 512KB. Isso se deve ao melhor desempenho do SSD em relação ao HDD, que contribui para que o arranjos de cache obtenham melhor desempenho que arranjos RAID, especialmente com

blocos menores que o *chunk* do RAID. Isso evidencia que usar caches SSD em *workloads* de leitura aleatória melhora o desempenho frente a RAID para casos onde o tamanho médio de requisições seja menor que o *chunk*. Isso porque o paralelismo do RAID não atua nesses casos, mas o ganho do dispositivo SSD tem impacto no desempenho.

Observado esse melhor comportamento dos arranjos de cache, deve-se observar também a vantagem do SSD sobre o HDD. Para os casos de teste, o SSD teve 8,6× melhor desempenho que o HDD para blocos de 64KB, 4,1× para 512KB e 2,2× para 4MB. Em testes separados com blocos de 4KB, o SSD chegou a ≈ 50× melhor desempenho que o HDD para leituras aleatórias; conforme o padrão de acesso torna-se mais sequencial (com blocos maiores, por exemplo), a diferença tende a diminuir.

Pode-se também observar que o DMCache foi superior ao BCache em quase todos os casos de configuração igual ou muito similar. Entre *bcache-1s-1h* e *dmcache-1s-1h*, o DMCache teve um desempenho 28,6% superior para requisições de 64KB, 52,4% para 512KB e 50,4% para requisições de 4MB. Além disso, o DMCache teve um desempenho médio 23,9% acima do BCache em todos os *workloads* com cache. Esse desempenho se deve a uma melhor taxa de acertos do algoritmo de substituição do DMCache, fazendo com que o SSD seja mais utilizado durante os *workloads* e assim melhorando o desempenho global.

Escrita Aleatória

A Figura 6.5 apresenta os resultados de vazão para escrita aleatória. Ao analisar discos isolados (*1h* e *1s*), observa-se que o HDD mantém a mesma vazão da leitura aleatória, devido ao dispositivo ter os mesmos tempo de acesso nessas operações. Já o SSD mostrou diferença entre leituras e escritas, o que remete à assincronia entre essas operações nestes dispositivos.

Ao analisar HDD (*1h*) e SSD (*1s*) isoladamente, podemos observar que o disco rotacional mantém a mesma vazão apresentada na leitura aleatória, que se deve à característica do dispositivo de possuir o mesmo tempo de acesso a essas operações nos dois casos. Já o SSD apresentou diferenças da leitura em relação à escrita, o que remete à assincronia

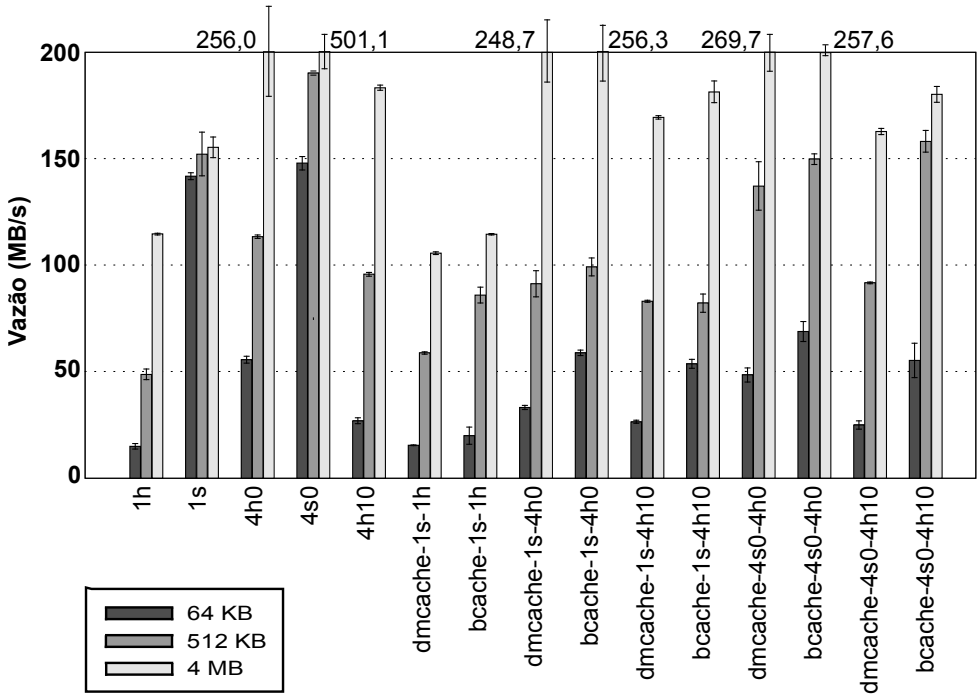


Figura 6.5: *Escrita Aleatória.*

entre essas operações no SSD. Além disso, o SSD quase não apresentou diferenças na escrita quando do aumento do tamanho do bloco, diferente do que ocorreu na leitura. Isso ocorreu porque todas as operações foram realizadas com a flag *sync* habilitada, o que faz com que uma leitura só possa ocorrer depois que a anterior é completada. Neste caso, tanto HDDs quanto SSDs sofrem a espera pelo dado e o tamanho de bloco importa. Para a escrita, a grande maioria dos SSDs trabalham com memórias caches internas, que confirmam a escrita rapidamente e aceleram a vazão, diminuindo a influência do tamanho de bloco. Isso pôde ser percebido através das métricas de latência das requisições do *iostat* e do *fio* para os SSDs.

Outra diferença entre escritas e leituras aleatórias é a vazão de arranjos RAID para escritas com bloco menor que o *chunk* (64KB e 512KB). Nestes casos, a escrita teve melhor vazão em RAID0 e RAID10, por exemplo, *4h0* foi 3,7× melhor que *1h* para blocos de 64KB. As solicitações de escritas são entregues ao dispositivo RAID, que as paraleliza quando possível, através de filas e *merges*, gerando ganhos. O mesmo não pode ocorrer na leitura síncrona, mesmo com o uso de RAID, deixando o desempenho refém do *seek time* nos HDDs e da latência normal da operação nos SSDs. Para HDDs em RAID10 (*4h10*),

o desempenho seguiu o mesmo padrão apresentado anteriormente, com uma leve redução do desempenho de escrita aleatória em relação ao RAID0, por conta das confirmações de escrita do nível 1 de RAID presente no RAID10. Como esperado, o RAID10 (*4h10*), por exemplo, obteve metade (26 MB/s) do desempenho do RAID0 (*4h0*, 54 MB/s) e foi 2× melhor que um HDD (14,6 MB/s), para blocos de 64KB.

Em relação aos arranjos de cache, nos testes de escrita aleatória ocorre um comportamento inverso à leitura aleatória: o BCache é superior ao DMCache na maioria dos testes. Esse comportamento ocorre porque o DMCache prioriza leituras em relação a escritas, por conta das limitações dos SDDs, como *write-amplification* e limite de escrita por bloco. Além disso, os SSDs possuem desempenho médio de escrita aleatória bem inferior ao desempenho de leitura. Para validação, submetemos a teste requisições de 4KB (tamanho padrão de bloco nos sistemas de arquivos Ext4 [51]), o que aumenta o *seek time* e diminui o desempenho dos HDDs, e verificamos que o SSD foi 88× superior ao HDD na leitura e apenas 37× na escrita.

O DMCache regula a prioridade de escritas e leituras com os parâmetros *write_promote_adjustement* (default = 8) e *read_promote_adjustement* (default = 4). Antes de serem trazidos para o dispositivo de cache, os blocos precisam ter ao menos 8 acessos em escrita ou 4 em leitura. Os efeitos desses parâmetros são percebidos nos resultados apresentados. Apesar do resultado inferior do DMCache na escrita, em situações reais a junção da escrita e leitura aliada à importância dos dados mantidos em cache determina um melhor desempenho para o DMCache. Os demais casos de testes entre arranjos de cache seguem os mesmos padrões de desempenho obtidos para a leitura aleatória, seguindo o desempenho de *dmcache-1s-1h* e *bcache-1s-1h* com as diferenças de desempenho dos arranjos RAID envolvidos, já discutidas anteriormente.

6.3 Macrobenchmarks

Nesta Seção apresentaremos os macrobenchmaks, *workloads* sintéticos que simulam situações próximas a ambientes reais de produção, como servidor de arquivos (Fileserver), servidor de e-mail (Varmail) e bancos de dados relacionais (OLTP). Antes de analisarmos

esses experimentos, realizaremos uma análise das características desses *workloads* quando à sua aleatoriedade no acesso aos dados, ao tamanho das requisições e à quantidade de escritas frente às leituras, após isso traçaremos uma linha de base sobre a qual possamos analisar os resultados encontrados. Na análise dos experimentos, coletaremos principalmente informações de vazão desses *workloads* para os casos de testes já apresentados na Tabela 6.1.

A ferramenta escolhida para os experimentos dos macrobenchmarks foi o Filebench [10], por ser amplamente utilizada, conter diversos workloads pré-definidos e pelo seu ganho recente de popularidade (é citado em três artigos no FAST 2010 e em quatro no OSDI 2010) [45]. O Filebench simula padrões reais de acesso a servidores com diferentes finalidades e modela seus workloads através de uma linguagem de modelagem de *workloads* flexível (Workload Modeling Language - WML). Os *workloads* definidos para esses experimentos são descritos a seguir:

- **Fileserver:** simulando o funcionamento de um servidor de arquivos. O *workload* consiste em uma sequência de operações de criação, exclusão, escrita, leitura e operações sobre meta-dados em um conjunto de 500.000 arquivos, com tamanho médio de arquivo de 128 KB, 50 *threads* e *append* médio de 16 KB. As operações realizadas pelo *workload*, nesta ordem, foram *create*, *write*, *append*, *read*, *delete* e *stat*. As escritas e leituras de cada arquivo foram realizadas de forma integral. Este *workload* caracteriza-se a princípio como prioritariamente sequencial e as operações foram 1:2 entre leitura e escrita, nesta ordem.

- **Varmail:** Simula a atividade de I/O de um servidor de e-mails que armazena cada mensagem como um arquivo individual. O *workload* consiste em uma sequência de operações do tipo *create-append-sync*, *read-append-sync*, leitura e exclusão de arquivos em um mesmo diretório. A configuração desse *workload* consistia em 4.000.000 de arquivos com tamanho médio de 16 KB, *append* médio de 16 KB e 16 threads. As requisições possuíam tamanho menor do que fileserver, o que torna este *workload* menos sequencial do que o anterior. As operações de escritas e leituras obedecem proporção semelhante, 1:1.

- **OLTP:** Emula o padrão de acesso de um banco de dados transacional. Este *workload* testa o desempenho de múltiplas operações aleatórias de leitura e escrita sensíveis a latência. O padrão de acesso é baseado no banco de dados Oracle 9i. A configuração desse *workload* consiste de 10 arquivos com tamanho de 6 GB cada, com cache do *workload* habilitada e cache de escritas do sistema operacional desabilitada (O_DIRECT), 200 *threads* e 1 MB por *thread*. Diferente dos *workloads* anteriores, este possuía dois processos (com seus respectivos *filesets*), um processo de escrita no banco (*dbwr*), com operações de I/O de 2 KB em iterações de 100 repetições, e outro para escrita dos logs (*lgwr*), com operações de I/O de 256 KB sem repetições.

Cada experimento foi executado durante uma hora onde, ao final, extraímos informações como IOPS de leitura e escrita, vazão, utilização de CPU e latência. Ao iniciar cada *workload* o Filebench cria o conjunto de arquivos sobre os quais os testes serão executados. Essa inicialização é desconsiderada na contabilização final do resultado. Ao final da inicialização dos dados, alteramos o Filebench para que executasse *scripts* de coleta (iostat e systemtap) de informações extra ao Filebench como *seek*, estatísticas de cada dispositivo de I/O, *hit* em memória e tamanho médio das requisições.

O sistema de arquivos adotado para os dispositivos sobre os quais os workloads foram executados foi o Ext4, pela simplicidade, por ser o sistema de arquivos padrão do sistema operacional em testes e ser amplamente utilizado em ambientes reais de servidores. A quantidade de arquivos e tamanho médio dos arquivos foi alterada com a intenção de manter o *workingset* com o dobro do arranjo de cache SSD, seguindo o mesmo padrão de configuração adotado nos microbenchmarks. A quantidade de *threads* foi reduzida nos casos necessários de modo a não atingir o limite de uso de processador (CPU *bound*) e comprometer a medição do I/O. Além disso, assim como nos microbenchmarks, todas as informações coletadas foram gravadas em um dispositivo de I/O separado, visando não comprometer os resultados.

Os resultados coletados com o Filebench indicam a resposta a operações de I/O no nível do serviço solicitante dos dados; na pilha de I/O, a camada do VFS. As informações coletadas do *iostat* referem-se às operações que atingiram o dispositivo de I/O, após pas-

sarem por todas as camadas da pilha. Em todas as análises apresentadas esses resultados serão observados. A próxima Seção analisa a caracterização dos *workloads* executados.

6.3.1 Caracterização dos Workloads

Nesta Seção apresentaremos os padrões de acesso de cada *workload* testado para os macro-benchmarks, sendo eles Fileserver, Varmail e Oltp. Para a caracterização foram coletadas informações da execução de cada *workload* apenas sobre um HDD, pois identificamos que o padrão se repete independente do dispositivo de I/O sobre o qual a execução é direcionada. Utilizamos o *systemtap* com o *probe ioblock.request*, que intercepta informações da *generic block layer*, coletando os dados apresentados na caracterização. Dessa forma, as informações apresentadas são de solicitações dirigidas diretamente a cada dispositivo de I/O do ponto de vista desta camada. Além disso, implementamos diversas ferramentas para a extração e sumarização dos dados gerados pelo *systemtap*.

Dentre as métricas extraídas, apresentaremos aquelas que consideramos mais relevantes para os workloads apresentados. Utilizaremos (1) a taxa de leituras e escritas para cada *workload* (2) histogramas de *seek* (ou *jump*), entre as requisições de I/O e (3) histogramas com os tamanhos das requisições de I/O. As taxas de leitura e escrita (Figura 6.6) dizem respeito ao número de requisições para cada caso bem como ao montante em bytes dessas requisições. Os *seeks* (Figuras 6.7(a), 6.7(c) e 6.7(e)) representam a distância no espaço do disco que o dispositivo se deslocou para acessar os dados entre uma requisição e a sua subsequente; este valor foi coletado em setores (512 bytes), no entanto, apresentamos os resultados em KB, MB e GB para facilitar a visualização. O tamanho das requisições de I/O (Figuras 6.7(b), 6.7(d) e 6.7(f)) é mostrado para que possamos identificar *merges*, reordenações e sequencialidade/aleatoriedade dos *workloads*.

Taxas de I/O: Essa métrica variou de forma substancial entre os *workloads*. Apesar de Fileserver ter 1/2 operações de leitura/escrita, a relação apresentada para os dispositivos de I/O é de 23/76 em requisições e 42/57 em bytes. Atribuímos o comportamento da quantidade de bytes aos *appends* de escrita que, apesar de dobrar a quantidade de requisições de escrita, são responsáveis por tamanhos menores de requisições se comparados a

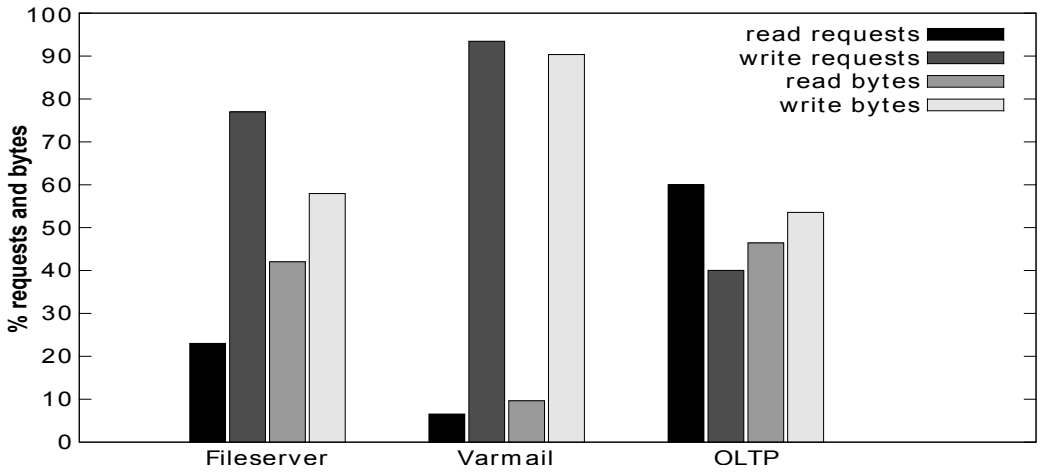


Figura 6.6: Taxas de leituras e escritas em quantidade requisições e tamanho total em bytes.

escritas e leituras de arquivos completos, que são 1/1 neste *workload*. A quantidade de requisições de escrita aumenta consideravelmente também por conta das operações sobre metadados (*create* e *delete*) ocorrerem mais na escrita e serem traduzidas para as camadas inferiores da pilha de I/O. Outro fator que influencia este resultado é a taxa de acerto em cache para leitura/escrita (93,8%/7,6%), o que faz com que muitas solicitações não cheguem a atingir as camadas inferiores na leitura.

No Varmail, apesar das leituras e escritas serem 1/1, os resultados da caracterização mostram 7/93 (r/w) para a quantidade de requisições e 1/9 para bytes acessados. Isso se deve ao fato de a maioria das requisições de leituras acertarem o cache (99,28% de *read cache hit*), sendo repassadas para as camadas inferiores da pilha de I/O apenas parte das requisições fazendo, dessa forma, com que a maioria das operações *read* não seja contabilizada. Para a escrita todas as requisições de I/O ocorreram diretamente no disco, sem a interferência do *pagecache*, o que justifica a diferença entre as solicitações do Filebench e o que atingiu o disco. Neste ponto, vale ressaltar que a memória RAM foi reduzida nos macrobenchmarks a 20% do disco SSD e, dependendo do *workload*, as operações devem ocorrer com a interferência do cache, conforme ocorre em ambientes reais.

Assim como no Varmail, a taxa de leitura/escrita é a mesma no OLTP (1/1). No entanto, este *workload* apresentou um resultado bastante equilibrado entre o que foi soli-

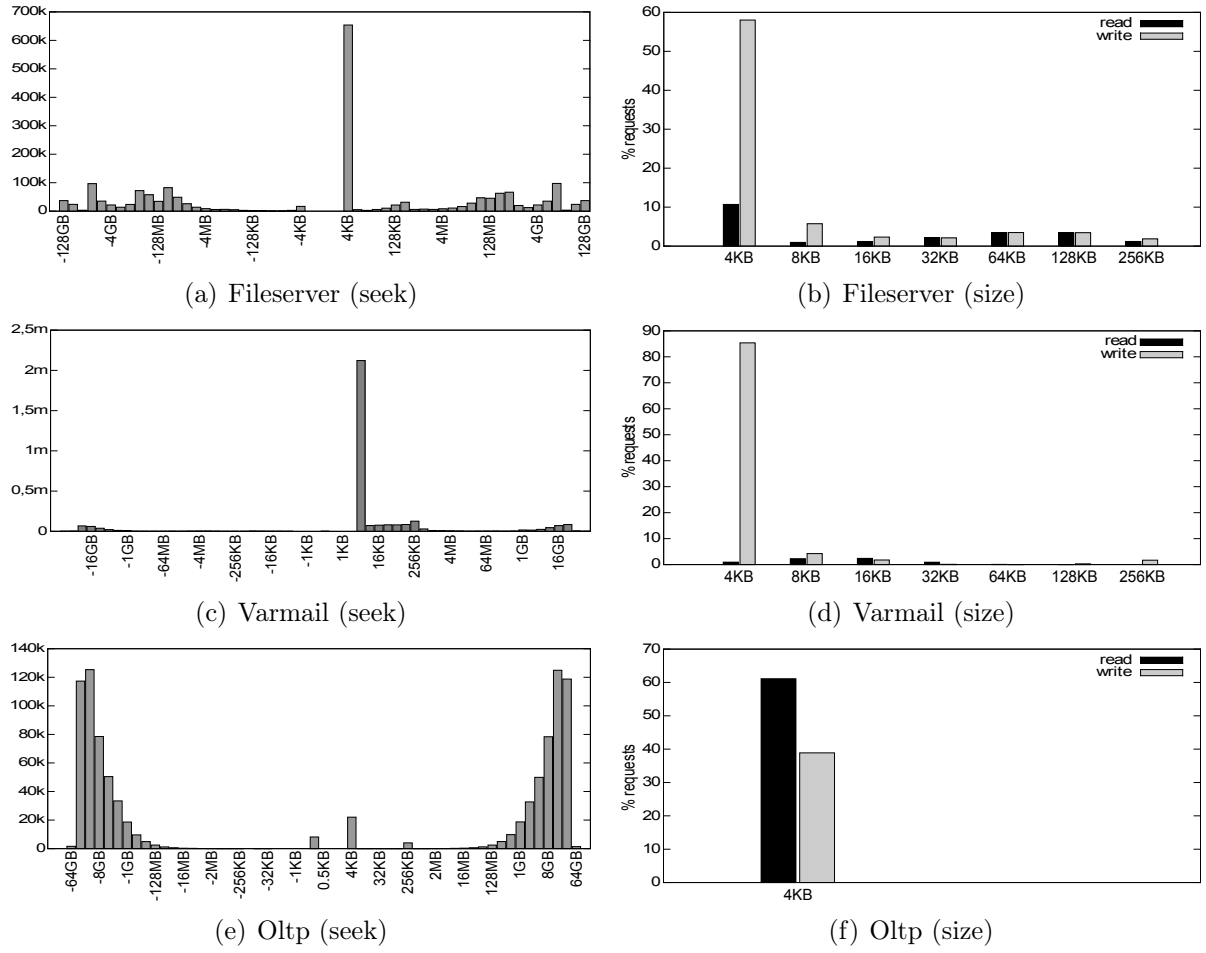
citado pelo Filebench e o que foi recebido na *generic block layer*. Assim como nos casos anteriores, o cache foi decisivo para o resultado, neste caso sem muita interferência, pois a maioria das operações no OLTP foi executada com a *flag* O_DIRECT, o que evita caches. As taxas de acerto em cache foram 0,5%/0,3% (R/W). Este foi o único *workload* em que a quantidade de requisições se inverteu com o valor em bytes das requisições. Atribuímos isso ao tamanho médio das requisições de leitura serem menores do que as de escrita, pois, apesar de haver processos 1/1 de leitura e escrita menores (2 KB), havia o *logwriter* (*lgrw*) com requisições maiores, de 256 KB ocorrendo durante todo o *workload* e desequilibrando estes valores.

Seek (deslocamento): Os gráficos de *seek* mostram que os *workloads* apresentados possuem um alto nível de sequencialidade. Com exceção do OLTP, tanto o Fileserver quanto o Varmail possuem grande parte dos *seeks* em 4 KB. Na coleta desse dado, foram extraídas as informações do setor para o qual a solicitação de I/O foi dirigida, portanto, não aparecem barras com valores zerados para deslocamento, visto que não descontamos na solicitação subsequente o tamanho da última solicitação, o que faz com que os valores de deslocamento sejam absolutos. No entanto, ao notarmos que 4 KB é o tamanho do bloco configurado no sistema de arquivos (Ext4), verificamos alta taxa de sequencialidade nesses *workload*. Partindo dessa premissa, observamos 33,6% de sequencialidade para Fileserver, 58,6% para Varmail e 2,1% para OLTP, sendo este último, portanto, o *workload* mais randômico dos experimentos. Outros 62,1% das requisições do Fileserver foram de *seeks* menores ou iguais ao tamanho do *workingset*, até 64 GB, esses são *seeks* entre diferentes arquivos do mesmo *workingset*. O restante das requisições (4,3%), que fogem a esse escopo, ocorreu sobre metadados e se espalham no restante do disco.

No Varmail, outros 30,6% das requisições se espalharam sobre a extensão do *workingset*. A soma desses seeks totalizam 89,2% do Varmail. O restante das requisições são atualizações de metadados sobre operações de sincronização (*fsync*) que ocorrem no início do disco. Este *workload* mostrou-se o mais sequencial dos três, bem como o que obteve a maior taxa de acerto sem memória.

O OLTP apresentou-se como o *workload* mais aleatório e com o menor acerto em

cache. A maior parte dos *seeks* (69,6%) ocorriam no intervalo entre 4 e 32 GB (dentro dos limites do *workingset*), o que caracteriza o *workload* de longos *seeks*, quando comparado à aleatoriedade dos demais. Apenas 13,5% do OLTP ocorria entre valores maiores do que o *workingset*, ocasionado por operações de metadados e sincronização no início do dispositivo.



(a) Fileserver (seek)

(b) Fileserver (size)

(c) Varmail (seek)

(d) Varmail (size)

(e) Oltp (seek)

(f) Oltp (size)

Figura 6.7: *Seek* (deslocamento) e *size* (tamanho) das operações de I/O nos *workloads*.

I/O Size: Todas as operações de I/O em todos os *workloads* ocorreram com tamanhos de no mínimo 4 KB e no máximo 256 KB, com raras exceções de operações em metadados menores do que 4 KB. O tamanho de requisição de I/O que mais apareceu nos resultados foi 4 KB, sendo 68,69% para Fileserver, 85,93% para Varmail e 98,08% para OLTP. Isso se deve ao tamanho mínimo do bloco do sistema de arquivos. Nos casos de Fileserver e Varmail, há correspondência entre grande parte das requisições de 4 KB e os *seeks* do mesmo tamanho já citados.

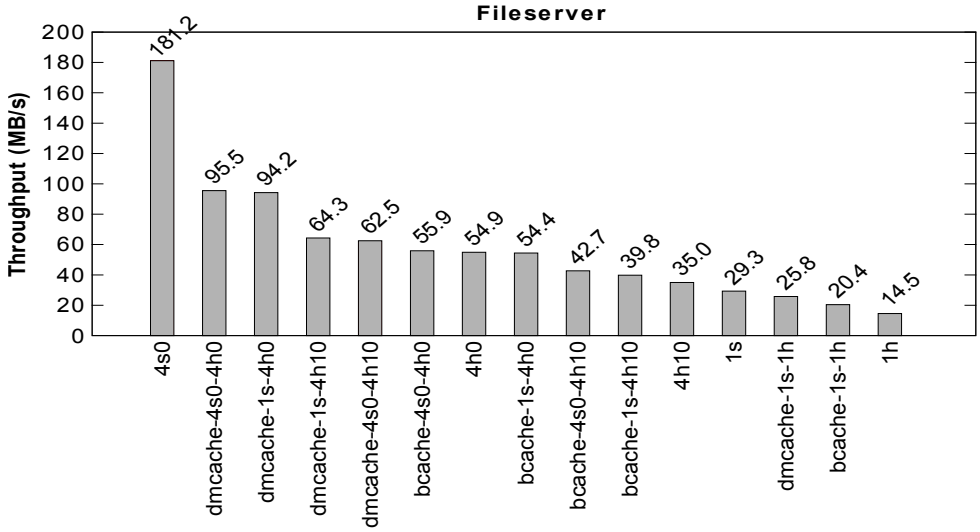
Todavia, para o OLTP a quantidade de *seeks* de 4 KB não passa de 3%, ao passo que o tamanho da maioria das requisições de I/O é de 4 KB. Atribuímos isso ao tamanho médio das leituras e escritas do OLTP (2 KB), que é arredondada pelo sistema de arquivos ao seu tamanho de bloco padrão. Essas observações, mais uma vez, reforçam a aleatoriedade do OLTP. Os únicos tamanhos de requisição que apareceram para o OLTP além de 4 KB foram 128 KB e 256 KB (somando 0,51%, cortados da Figura por o valor ser muito baixo). Isso ocorre devido ao tamanho médio da requisição de escrita no *log* pelo processo *logwriter*. A taxa de *merges* do escalonador de disco foi desprezível para o OLTP e os *merges* não ocorreram por conta da alta aleatoriedade do *workload*.

O Fileserver também apresentou requisições de 16 KB (3,47%), 64 KB (6,94%) e 128 KB (6,86%). Isso ocorre devido ao tamanho médio dos arquivos (128 KB) e à quantidade de *merges* no escalonador de disco realizadas (237 *merges*/s) para este *workload*. O tamanho médio da requisição deste *workload* foi 64 KB. Assim como no Fileserver, o tamanho médio dos arquivos influenciou o tamanho das requisições do Varmail. Grande parte das requisições aleatórias aconteceu para os tamanhos 8 KB (6,46%) e 16 KB (4,14%), justificados pelo tamanho médio dos arquivos e pelo tamanho médio dos *appends* (16 KB). Além disso, o Varmail alcançou a mais alta taxa de *merges* no escalonador de disco, 571 *merges*/s, e tamanhos médios de requisições de 32 KB, justificados pelos *merges* e a consequente sequencialidade alcançada.

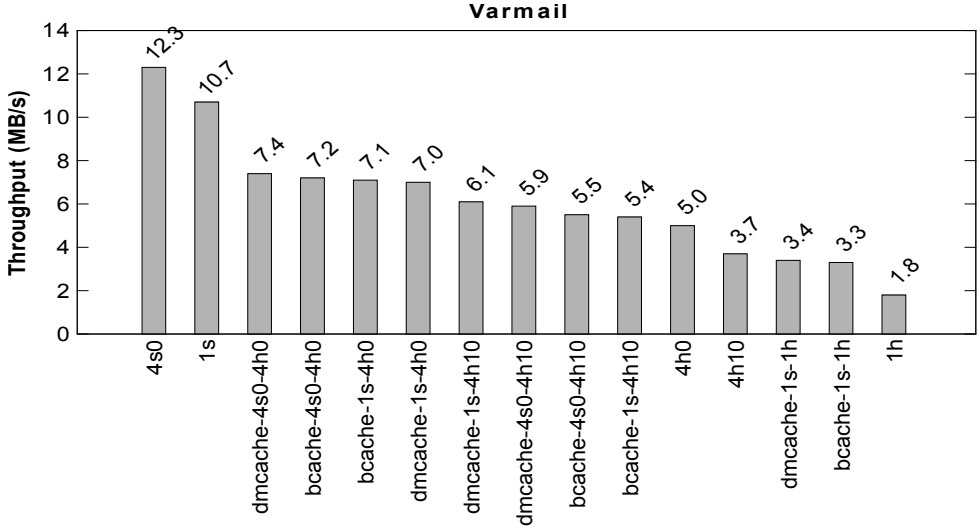
6.3.2 Resultados dos Macrobenchmarks

Nesta Seção apresentaremos os resultados obtidos para os macrobenchmarks Fileserver, Varmail e OLTP. Os resultados tratarão a respeito principalmente métrica vazão, sendo nos casos necessários apresentadas métricas como IOPS, latência, uso de CPU, *cache hit* e outras julgadas necessárias. Trataremos de cada *workload* separadamente, comparando cada arranjo RAID ou Cache com os seus equivalentes, por exemplo, *bcache-1s-1h* com *dmcache-1s-1h*. Ao final, separamos a Seção 6.4 onde analisaremos custos de dispositivos atuais sobre situações de uso com base nos resultados deste trabalho.

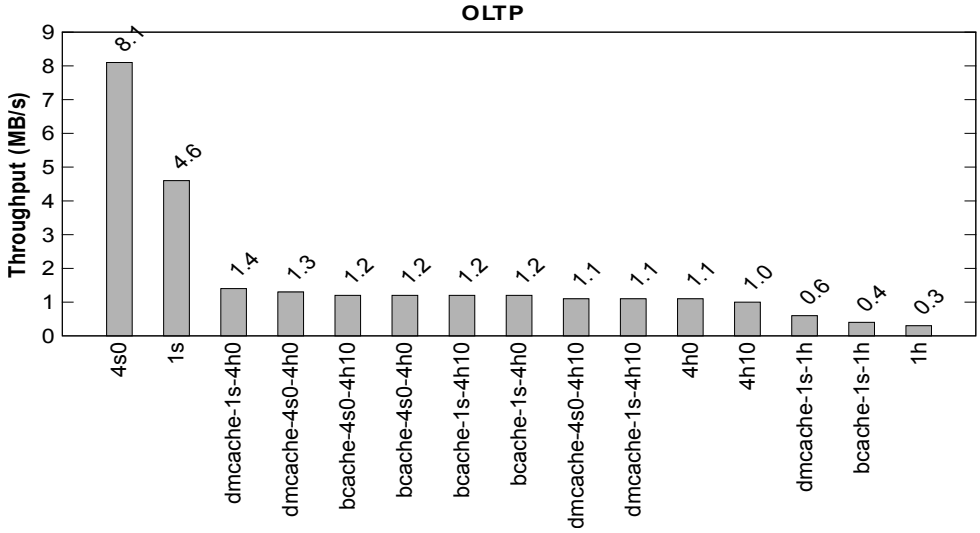
A Figura 6.8 apresenta a vazão alcançada para cada *workload* separadamente. O



(a) Vazão do *workload* Fileserver.



(b) Vazão do *workload* Varmail.



(c) Vazão do *workload* OLTP.

Figura 6.8: Vazão alcançada para cada *workload* dos microbenchmarks em MB/s.

eixo vertical indica a vazão em MB/s e o eixo horizontal os dispositivos testados. Cada figura refere-se a um *workload* (Fileserver: Figura 6.8(a), Varmail: Figura 6.8(b) e OLTP: Figura 6.8(c)). Os resultados encontram-se ordenados de acordo com o desempenho dos dispositivos dentro de cada figura a fim de facilitar a comparação e visualização.

Em todos os *workloads* é possível perceber que o melhor e o pior caso entre os dispositivos se repetem. Para o melhor caso se destacou quatro SSDs em RAID0 (*4s0*) e para o pior caso um HDD (1h). O resultado é esperado de acordo com os resultados individuais e em RAID dos dispositivos apresentados nos microbenchmarks, notável também pelas características tecnológicas dos dispositivos. Um SSD isolado tem desempenho superior a um HDD isolado, sendo 8,6× na leitura e 9,5× na escrita aleatória de 64 KB. Com o paralelismo do RAID0 o SSD alcança destacadamente melhor desempenho esperado para todos os *workloads*. No entanto, este tem o maior custo por GB dentre todos os arranjos, o que faz com que os demais arranjos tornem-se ainda atraentes para determinados ambientes.

Ao se acrescentar arranjos de cache com um SSDs (*bcache-1s-1h* e *dmcache-1s-1h*), em todos os *workloads* percebe-se ganho de desempenho se esses arranjos são comparados com um HDD (*1h*) isolado. Este ganho aparece mais acentuado apenas para determinados *workloads*. Fileserver, por exemplo, tem ganho de 41% para *bcache-1s-1h* e 78% para *dmcache-1s-1h* em relação a *1h*. Varmail apresenta ganho de 83% para *bcache-1s-1h* e 89% para *dmcache-1s-1h*, tendo ainda estes arranjos desempenhos bastante próximos de *4h10*. Isso acontece porque o *workload* Varmail é caracterizado prioritariamente por escritas aleatórias de requisições menores, o que faz com que o RAID10 obtenha pouca vantagem na escrita e seja indiferente na leitura comparado ao RAID0, tendo aproximadamente o dobro do desempenho de um HDD (por conta do nível 0 no RAID10). Para o *workload* OLTP, o mesmo ganho foi de 33% para *bcache-1s-1h* e 100% para *dmcache-1s-1h*. Em todos esses casos ao se acrescentar caches em SSD os workloads apresentaram consideráveis ganhos de desempenho.

Por outro lado, num *workload* aleatório com arquivos menores, qualquer composição com RAID nos dispositivos de origem e SSDs obteve melhor desempenho se comparado

a um similar que use RAID de apenas HDDs. Isso é perceptível nos *workloads* Varmail e OLTP, caracterizados pela alta aleatoriedade e de tamanho das requisições menores, o que traz pouca vantagem ao uso de RAID. Nestes workloads os arranjos de cache superaram as duas composições RAID (*4h0* e *4h10*). Isso pode ser confirmado pelo desempenho do *4h0* no Fileserver, que pôde superar arranjos de cache com RAID por conta das requisições maiores e *merges* de requisições menores.

Ao comparar os arranjos de cache entre si, é interessante notar que para o *workload* Fileserver o DMCache é dominante em relação ao BCache, sendo bastante superior em arranjos equivalentes, por exemplo, *dmcache-1s-1h* é 26% superior a *bcache-1s-1h* e *dmcache-4s0-4h0* é 71% superior a *bcache-4s0-4h0*. Isso se deve à alta taxa de hits do DMCache, chegando a 58,8% para o cache de segundo nível e 74,5% para cache em RAM, contra 28,2% no segundo nível e 60% no primeiro nível para o BCache. A baixa taxa de acertos em RAM do BCache deve-se ao maior consumo de memória deste algoritmo nas suas estruturas de dados em determinados *workloads*. No segundo nível o DMCache obtém vantagens no seu algoritmo de cache frente às características deste *workload*, que apesar de ser randômico (o que faz os caches SSD operarem), tem bastante repetição de acesso a blocos e com isso boa localidade.

Para Varmail e OLTP a comparação anterior apresenta-se bastante diferente. BCache e DMCache mostram-se similares para dispositivos equivalentes. Por exemplo, *dmcache-4s0-4h0* é apenas 3% superior ao *bcache-4s0-4h0*, no *workload* Varmail. Neste caso, o BCache é beneficiado pelo *workload*, por ser prioritariamente de escrita, o que diminui consideravelmente o desempenho do DMCache. Ainda assim, o DMCache apresenta melhor desempenho global, 5% em média, situação prevista nos microbenchmarks.

Para OLTP os caches apresentam desempenho pouco significativo em todos os casos de teste. Este *workload* apresentou 100% de *cache miss* em RAM, devido às operações diretas (*flag* O_DIRECT) e acerto médio em cache SSD de 23,5% (bastante inferior aos demais *workloads*). No entanto, apesar de pouco expressivo, o desempenho do melhor caso de arranjo cache (*dmcache-1s-4h0*) ainda foi superior aos arranjos RAID de HDDs (entre 30% e 40%). Além disso, o uso de arranjos RAID nos dispositivos de cache não elevou

o desempenho, em alguns casos chegou a gerar *overhead* no RAID (*dmcache-4s0-4h0*), neste caso o tamanho médio das requisições foi determinante.

Por fim, os *workloads* Varmail e OLTP também apresentaram destaque ao desempenho do SSD isolado em relação aos caches. Neste caso, repete-se o comportamento anterior, o RAID toma pouco proveito do tamanho das requisições, beneficiando-se em parte apenas do desempenho dos *merges*, notável no desempenho entre *4s0* e *1s* do Varmail (15% de ganho). No entanto para estes *workloads* a aleatoriedade traz impacto significativo e, aliada ao tamanho médio das requisições, geram uma alta taxa de IOPS, métrica favorável a SSDs isolados, que nesses casos praticam desempenho superior a arranjos de cache com baixa taxa de acertos. A próxima Seção continua a avaliar os resultados dos macrobenchmarks sobre a perspectiva do custo/benefício.

6.4 Estudo Econômico de Caso de Uso

Nesta Seção analisaremos o impacto econômico dos casos de teste nos arranjos apresentados, comparando preços atuais de HDDs e SSDs frente aos resultados de desempenho discutidos anteriormente. Essa análise de custo, ainda que sumária, esclarece os benefícios do uso de cache atualmente. Adotaremos a vazão como métrica representativa do desempenho, visto que os arranjos serão comparados dentro de um mesmo *workload*, dessa forma, não há interferência de métricas como latência e IOPS, que estarão representadas na vazão, neste caso.

A Tabela 6.2 apresenta os valores coletados de quatro SSDs [2] e quatro HDDs [1] atuais disponíveis no mercado e dos dispositivos utilizados nos casos de teste deste trabalho. Todos os dispositivos escolhidos são para interface SATA III. A escolha dos discos rígidos foi limitada a 7.200 RPM e 1 TB de capacidade. O valor médio por GB dos SSDs é U$ 0,35 e dos HDDs U$ 0,054, sendo os SSDs $\approx 6{,}5\times$ mais dispendiosos do que os HDDs nos dispositivos atuais, para os dispositivos utilizados neste trabalho essa comparação sobe para $\approx 18{,}8\times$. Conforme se pode observar, esse distanciamento tende a cair por conta da redução do custo de tecnologia nos dispositivos mais atuais, em especial dos SSDs.

No entanto, ao tratar de desempenho, dependendo do *workload*, os SSDs podem apre-

Modelo	**Valor**	**Tamanho**	**U$/GB**
SSDs atuais			
Samsung 850 EVO	U$ 97,99	250 GB	U$ 0,39
Kingston Digital SSDNow	U$ 76,99	240 GB	U$ 0,32
SanDisk Ultra II	U$ 89,99	240 GB	U$ 0,37
PNY CS1111	U$ 79,99	240 GB	U$ 0,33
HDDs atuais			
HGST Travelstar 7K1000	U$ 54,99	1024 GB	U$ 0,054
Western Digital Blue	U$ 52,99	1024 GB	U$ 0,051
Toshiba DT01ACA100	U$ 52,95	1024 GB	U$ 0,051
Samsung Spinpoint	U$ 64,00	1024 GB	U$ 0,062
HDDs e SSDs utilizados nos testes			
Intel SSDSA2BW160G3H	U$ 102,93	160 GB	U$ 0,64
Hitachi HUA723020ALA641	U$ 69,95	2048 GB	U$ 0,034

Tabela 6.2: Custos de HDDs e SSDs atuais e utilizados nos testes [1, 2].

sentar valores que podem justificar o seu uso diante do custo. O *workload* OLTP é um excelente caso de atuação dos SSDs em relação aos HDDs. Um SSD (*1s0*) isolado apresenta desempenho 15,3× superior a um HDD (*1h0*), o que compensa o custo por GB para os dispositivos atuais (6,5×), sendo este o melhor dos casos. Para Varmail, o mesmo desempenho é 5,94× superior e apenas 2× para Fileserver, não compensando o uso nestes *workloads*, caso seja considerado unicamente o custo como parâmetro de decisão.

Ao analisar os dispositivos de caches em relação aos arranjos RAID de HDDs, o custo também apresenta um impacto significativo. Para calcular o custo efetivo dos arranjos de cache unificamos os valores dos dispositivos pela unidade mínima de custo por GB (U$/$GB$). Para os arranjos de cache que obedeciam a proporção de 20% dos HDDs, consideramos apenas os GB do cache no custo, por exemplo, *dmcache-1s0-4h0* teve custo do arranjo (c_a) calculado por $c_a = [(c_s \times 1) \times 0{,}2] + (c_h \times 4)$, sendo c_s = *custo médio por GB dos SSDs* e c_h = *custo médio por GB dos HDDs*. Para calcular o custo do desempenho (c_d) dividimos o custo do arranjo (c_a) pelo desempenho do arranjo (d_a) em MB/s, sendo $c_d = (c_a/d_a)$. A variável c_d passa a expressar de forma mais clara o custo/benefício dos arranjos testados.

A Tabela 6.3 apresenta os valores calculados para a variável c_d, considerando como desempenho do arranjo d_a os valores de vazão apresentados nos resultados dos macrobenchmarks apresentados na Figura 6.8. O custo do arranjo (c_a) foi calculado pelo valor

Device/Workload	**Fileserver**	**Varmail**	**OLTP**
1h	3,76	**30,28**	181,67
4h10	6,23	58,92	218,00
4h0	3,97	43,60	198,18
1s	*11,95*	32,71	**76,09**
4s0	7,73	*113,82*	172,84
bcache-1s-1h	6,10	37,73	311,25
bcache-1s-4h10	7,24	53,33	415,00
bcache-1s-4h0	5,29	40,56	415,00
bcache-4h0-4h10	11,66	90,55	415,00
bcache-4h0-4h0	8,91	69,17	415,00
dmcache-1s-1h	4,83	36,62	207,50
dmcache-1s-4h10	4,48	47,21	*452,73*
dmcache-1s-4h0	**3,06**	41,14	355,71
dmcache-4h0-4h10	7,97	84,41	*452,73*
dmcache-4h0-4h0	5,21	67,30	383,08

Tabela 6.3: Comparativo de custo/desempenho (c_d) entre os arranjos e *workloads* testados.

médio dos *SSDs atuais* da Tabela 6.2. Todos os valores de c_d foram multiplicados por 10^3 para facilitar a visualização dos números. O melhor caso de cada *workload* encontra-se destacado em negrito e o pior caso em itálico na tabela.

Para o *workload* Fileserver o menor c_d foi no arranjo *dmcache-1s0-4h0*, tendo este arranjo custo 23% mais baixo do que *4h0* e desempenho 72% superior. Para o Varmail o arranjo de cache menos custoso foi o *bcache-1h0-4h0*, com c_d 7% mais baixo do que *4h0* e desempenho 42% superior. No OLTP o melhor caso de cache quanto ao custo foi o *dmcache-1h0-4h0*, no entanto, mesmo este arranjo foi 79% mais custoso do que *4h0* e desempenho somente 27% superior.

Tanto para Varmail quanto para OLTP os casos de melhor custo de desempenho foram no uso de apenas um SSD (*1s*); seguido por RAID de SSDs (*4s*). Isso ocorreu por questões de aleatoriedade, quantidade de escritas (Varmail) e tamanho médio das requisições com impacto no RAID. Nos casos com essas características, não recomendamos o uso de arranjos de cache, mas o uso tanto quanto possível de SSDs para áreas mais nobres de dados que, caso utilizados em RAID, com o ajuste adequado do tamanho do *chunk*, pode oferecer melhor desempenho do que outros arranjos a um baixo custo.

Casos como o Fileserver, com alta taxa de acerto nos cache (em RAM e nos SSDs)

e tamanhos de requisições maiores, recomendamos o uso de cache tanto quanto possível, visto que este caso, apesar do ganho no custo de desempenho ser pouco superior, a melhoria de performance é bastante considerável. O tamanho do *chunk* do RAID permanece importante também neste caso em situações onde seja necessário agregar mais SSDs no mesmo arranjo cache.

CAPÍTULO 7

CONCLUSÃO

Este trabalho compara duas implementações recentes de cache em SSD para Linux: BCache e DMCache. Além disso, analisa o impacto desses dispositivos diante de abordagens tradicionais como RAID. As análises foram realizadas a partir de microbenchmarks e macrobenchmarks. Os microbenchmarks traçaram uma linha base do desempenho dos dispositivos isolados para leitura e escrita aleatória. Os macrobenchmarks apresentaram o desempenho desses dispositivos em situações similares a ambientes reais de servidores de arquivos, de e-mail e de banco de dados.

Os testes revelaram um grande impacto do tamanho das requisições frente ao tamanho de *chunk* do RAID e mostraram como os SSDs amenizam essa situação para casos de escrita aleatória. Nos microbenchmarks, para os arranjos de cache de segundo nível, observam-se vantagens do DMCache sobre o BCache na leitura aleatória e resultado inverso na escrita aleatória, devido a parametrizações *default* do DMCache, por considerar as limitações de escrita dos SSDs. Mesmo assim, o DMCache apresentou um desempenho global superior ao BCache, por conta da leitura. Em alguns casos, ambas as implementações de cache mostraram-se como uma melhor alternativa em relação ao RAID de HDDs, como na leitura aleatória de blocos menores que o tamanho de *chunks*.

Para os macrobenchmarks os SSDs apresentaram-se bastante superiores aos HDDs quando apresentados de forma isolada, em especial para Varmail e OLTP. O ganho de desempenho dos caches de um SSD em relação a um HDD apresentou desempenhos de 41%, 89% e 100% para Fileserver, Varmail e OLTP, respectivamente. O tamanho das requisições mostrou-se de grande impacto ao acrescentar RAID aos arranjos de cache, levando atenção ao tamanho dos *chunks* neste caso. Entre os arranjos de cache, o DMCache mostrou-se até 71% superior ao BCache para arranjos similares no Fileserver, e bastante equivalente no Varmail e OLTP. Os arranjos de cache trouxeram pouco impacto

positivo ao desempenho nestes dois *workloads*. Na análise de custo dos dispositivos, arranjos de cache ainda mostram-se como uma boa alternativa, principalmente em casos como o Fileserver em que a taxa de *hits* é determinante. Casos como Varmail e OLTP o uso apenas de SSDs apresentou um excelente custo/benefício, não sendo recomendado o uso de caches em SSD nesses casos.

Além dos resultados apresentados, este trabalho contribuiu para a identificação de um *bug* no kernel Linux [9] e sugeriu mudanças para a ferramenta de benchmarks TPCC-Uva [28]. O problema do kernel resultava em *crash* na montagem de volumes envolvendo RAID1 e o DMCache a partir do *commit 8c081b5*. Como o erro impedia a continuação dos testes, estes só puderam continuar a partir da última versão estável antes do *bug*. Já a ferramenta TPCC-Uva apresentava lentidão exagerada na criação das bases de dados para testes TPC-C [49]. Com este problema, a geração de bases de dados para tamanhos acima de 2GB poderia levar meses. Realizamos a correção da ferramenta e submetemos aos autores, que apreciaram a mudança, ocorrida oito anos após a última atualização oficial da ferramenta [27]. O TPCC-Uva não foi utilizado nos resultados finais deste trabalho, pois foi substituído pelo OLTP do Filebench. Outra contribuição relevante foi a submissão de um artigo intitulado "Avaliação de Caches em Dispositivos de Armazenamento Secundário com SSDs" ao SBESC 2015 - *V Brazilian Symposium on Computing Systems Engineering*, que está em processo de análise.

Por fim, possíveis expansões deste trabalho poderão se dar através de: (1) testes com *traces* extraídos de sistemas reais; (2) análise do comportamento dos arranjos de cache para diferentes proporções de caches; (3) análise de *workloads* mais próximos das situações de uso atuais, como ferramentas de MapReduce [8], sistemas de arquivos distribuídos, ambientes virtualizados e de computação em núvem.

BIBLIOGRAFIA

[1] HDDs - SATA III 6Gbits. http://goo.gl/oIWaCg, 2015.

[2] SSDs - SATA III 6Gbits. http://goo.gl/Bq3JML, 2015.

[3] Daniel Bovet e Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[4] Angelo Brayner e Mario A Nascimento. Solid-State Disks: How Do They Change the DBMS Game? *CSBC 2013 - SBBD*, 2013.

[5] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, e Christian A. Lang. SSD Bufferpool Extensions for Database Systems. *Proc. VLDB Endow.*, 3(1-2):1435–1446, setembro de 2010.

[6] Heoh Chin-Fah. ARC reactor also caches? http://storagegaga.com/arc-reactor-also-caches/, 2012.

[7] Michael Cornwell. Anatomy of a Solid-state Drive. *Commun. ACM*, 55(12):59–63, dezembro de 2012.

[8] Jeffrey Dean e Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, janeiro de 2008.

[9] Leonardo Antônio dos Santos. Re: [dm-devel] dmcache RAID1 bug? https://www.redhat.com/archives/dm-devel/2014-November/msg00076.html, 2014. [Online; accessed 02-Jan-2014].

[10] Filebench. http://filebench.sourceforge.net/, 2015.

[11] FIO. http://freecode.com/projects/fio, 2014.

[12] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. Linux Kernel Documentation, 2003.

[13] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Always learning. Prentice Hall, 2013.

[14] Christopher Hollowell, Richard Hogue, Jason Smith, William Strecker-Kellogg, Antonio Wong, e Alexandr Zaytsev. The effect of flashcache and bcache on i/o performance. *Journal of Physics: Conference Series*, 513(6):062023, 2014.

[15] Intel SSD 320 Series (160GB, 2.5in SATA 3Gb/s, 25nm, MLC). http://ark.intel.com/pt-br/products/56565/Intel-SSD-320-Series-160GB-2_5in-SATA-3Gbs-25nm-MLC, 2015.

[16] Bart Jacob, Paul Larson, Breno Henrique Leitao, e Saulo Augusto M Martins da Silva. SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems. http://www.redbooks.ibm.com/abstracts/redp4469.html, 2015. [Online; accessed 25-Jul-2015].

[17] Jens Axboe. Measuring IOPS. http://www.spinics.net/lists/fio/msg00830.html, 2015.

[18] Song Jiang e Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[19] I. Koltsidas, S. Viglas, e P. Buneman. *Flashing Up the Storage Hierarchy*. University of Edinburgh, 2010.

[20] Ioannis Koltsidas e Stratis D. Viglas. Data Management over Flash Memory. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMODÍ1, páginas 1209–1212, New York, NY, USA, 2011. ACM.

[21] Petros Koutoupis. Advanced Hard Drive Caching Techniques. *Linux J.*, 2013(233), setembro de 2013.

[22] Jaemyoun Lee e Kyungtae Kang. Assessment of dm-cache running on virtual linux. *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*, páginas 275–276, Dec de 2014.

[23] Performance Comparison among EnhanceIO, bcache and dm-cache. http://lkml.iu.edu/hypermail/linux/kernel/1306.1/01246.html, 2013. [Online; accessed 30-Jul-2015].

[24] Linux 2.6.33. http://kernelnewbies.org/Linux_2_6_33, 2015.

[25] dmsetup. http://linux.die.net/man/8/dmsetup, 2015.

[26] iostat. http://linux.die.net/man/1/iostat, 2015.

[27] Diego R. Llanos. Change-log: TPCC-UVa Version 1.2.4. http://www.infor.uva.es/~diego/tpcc-uva.php, 2014. [Online; accessed 02-Jan-2014].

[28] Diego R. Llanos. TPCC-UVa: A free, open-source implementation of the TPC-C Benchmark. http://www.infor.uva.es/~diego/tpcc-uva.html, 2014.

[29] Michael T. LoBue, Harry Mason, Thomas Hammond-Doel, Eric Anderson, Mike Alexenko, e Tom Clark. Surveying Today's Most Popular Storage Interfaces. *Computer*, 35(12):48–55, dezembro de 2002.

[30] LVM. LVM2 Resource Page. http://sourceware.org/lvm2/, Acessado em 10 Julho, 2014.

[31] Nimrod Megiddo e Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, páginas 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[32] Sun Microsystems. The Solaris ZFS filesystem, 2008.

[33] C. Morimoto. *Hardware, o Guia Definitivo*. OGDH Press e Sul Editores, 2nd edition, 2007.

[34] J. Nagle. On Packet Switches with Infinite Storage. *Communications, IEEE Transactions on*, 35(4):435–438, Apr de 1987.

[35] Elizabeth J. O'Neil, Patrick E. O'Neil, e Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, páginas 297–306, New York, NY, USA, 1993. ACM.

[36] Kent Overstreet. Linux BCache. http://evilpiepirate.org/git/linux-bcache.git/tree/Documentation/bcache.txt, 2003.

[37] David A. Patterson, Garth A. Gibson, e Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *SIGMOD Conference*, páginas 109–116, 1988.

[38] Device-mapper Resource Page. https://www.sourceware.org/dm/, 2014.

[39] Pedro Rocha e Leonardo Santos. Uma análise experimental de desempenho dos protocolos de armazenamento aoe e iscsi. *CSBC 2012 - SEMISH*, jul de 2012.

[40] Mendel Rosenblum e John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, fevereiro de 1992.

[41] Chris Ruemmler e John Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27(3):17–28, março de 1994.

[42] Abraham Silberschatz, Peter Baer Galvin, e Greg Gagne. *Fundamentos de Sistemas Operacionais, Sexta Edição*. LTC Editora, 2004.

[43] Yannis Smaragdakis, Scott Kaplan, e Paul Wilson. EELRU: Simple and Effective Adaptive Page Replacement. *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, páginas 122–133, New York, NY, USA, 1999. ACM.

[44] Andrew S. Tanenbaum. *Sistemas Operacionais Modernos, Terceira Edição*. Pearson Education, São Paulo, SP, Brasil, 2010.

[45] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, e Margo Seltzer. Benchmarking file system benchmarking: It *is* rocket science. *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, páginas 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[46] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, e Erez Zadok. Virtual machine workloads: The case for new benchmarks for NAS. *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February de 2013. USENIX Association.

[47] Joe Thornber. DM-Cache. https://www.kernel.org/doc/Documentation/device-mapper/cache.txt, 2014.

[48] Avishay Traeger, Erez Zadok, Nikolai Joukov, e Charles P. Wright. A Nine Year Study of File System and Storage Benchmarking. *Trans. Storage*, 4(2):5:1–5:56, maio de 2008.

[49] TPC-C is an on-line transaction processing benchmark. http://www.tpc.org/tpcc/, 2014.

[50] Datasheet - Ultrastar (TM) 7K3000. http://www.hgst.com/tech/techlib.nsf/techdocs/EC6D440C3F64DBCC8825782300026498/$file/US7K3000_ds.pdf, 2015.

[51] Darrick J. Wong. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout, 2015.

[52] Yuanyuan Zhou, Zhifeng Chen, e Kai Li. Second-Level Buffer Cache Management. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):2004, 2004.