EBRES: Efficient Buffer Replacement with Exponential Smoothing

Gustavo Moraes[1], **Angelo Brayner**[1], **José de Aguiar Moraes Filho**[2]

[1]Universidade Federal do Ceará – UFC – CE – Brasil

[2]SERPRO – Fortaleza – CE – Brasil

{gustavomoraes94, jaguiarm}@gmail.com, brayner@dc.ufc.br

Abstract. *Database management systems (DBMS) implement a replacement algorithm to determine which data to swap between main and secondary memories. Such algorithms are constantly evolving. We may be starting a new era of these algorithms with extensive advances in machine learning. This article proposes the EBRES (Efficient Buffer Replacement with Exponential Smoothing) algorithm to embed an intelligent page replacement policy based on a low-overhead prediction model to predict future data accesses.*

Resumo. *Os Sistemas gerenciadores de banco de dados (SGBD) implementam um algoritmo de substituição para determinar quais dados trocam entre as memórias principal e secundária. Esses algoritmos estão em constante evolução. Podemos estar iniciando uma nova era desses algoritmos com avanços extensivos no aprendizado de máquina. Este artigo propõe o algoritmo EBRES (Efficient Buffer Replacement with Exponential Smoothing) para incorporar uma política de substituição de página inteligente baseada em um modelo de previsão de baixa sobrecarga para prever acessos futuros a dados.*

1. Introdução

Apesar dos avanços tecnológicos, a hierarquia de memória ainda é o principal modelo para minimizar os efeitos da diferença de tempo de acesso entre memória principal e memória secundária, impactando diretamente na arquitetura de um SGBD [Effelsberg and Härder 1984]. Assim, SGBDs implementam um gerenciador de *buffer* para prover uma hierarquia de memória autônoma à do sistema operacional. Mesmo com a evolução dos componentes de hardware como os SSDs *(solid-state drives)*, que apresentam um desempenho superior aos HDDs *(hard disk drives)*, ainda há uma grande diferença, em termos de largura de banda de IO, entre mídias de memória secundárias e a memória principal. Em [Harizopoulos et al. 2008], encontra-se evidência que o gerenciador de buffer apresenta um maior consumo de CPU (aproximadamente 34.6%) dentre diversos componentes de um SGBD. Logo, melhorias neste componente são fundamentais para otimizar desempenho dos SGBDs.

Um dos pontos a otimizar dentro dentro do gerenciador de *buffer* são as políticas de substituição de páginas, responsáveis por determinar quais páginas devem residir na memória principal (*hit*). Quando há requisição de acesso a uma página P, o gerenciador de buffer solicita bloco que contém P ao sistema de arquivos e mantém uma cópia de P na memória principal em uma matriz de *páginas* (*buffer pool*). Cada *página* é mapeada

por um descritor contendo os metadados de uso do bloco, como por exemplo o tipo da operação (leitura ou escrita). Quando não há memória suficiente para alocar uma nova página, o gerenciador de buffer consulta a política de substituição que escolhe uma página P (no *buffer pool*) como vítima (evicção) [Hellerstein et al. 2007], remove P da memória principal, e aloca a página requisitada (*miss*).

Algoritmos como LRU *(Least Recently used)* e LFU *(Least Frequently Used)* escolhem as vítimas considerando aspectos como a recência e frequência. SGBDs foram desenvolvidos sob a premissa que a memória secundária utilizaria HDDs como meio de armazenamento. Em HDD, o tempo para executar operações de leitura e escrita são iguais (mídia simétrica). Por sua vez, SSDs possuem um acesso assimétrico à mídia. Em SSDs o tempo de execução de operações de escrita são maiores que o tempo de operações de leitura [Park et al. 2011]. Para uma estratégia de substituição se beneficiar da assimetria da mídia de armazenamento, ela deve ter mecanismos para diferenciar as páginas entre as que sofreram operações de leitura ou de escrita. Reduzindo o número de páginas com escritas a serem escolhidas como vítimas, pode reduzir o tempo médio de acesso à memória secundária.

Esse artigo propõe um novo algoritmo de substituição de páginas em *buffer*, denominado EBRES *(Efficient Buffer Replacement with Exponential Smoothing)*. O EBRES apresenta as seguintes propriedades: *(i)* usa um modelo de aprendizagem de máquina complexidade constante $O(1)$; *(ii)* considera a assimetria da mídia de armazenamento secundária, e; *(iii)* busca manter um equilíbrio entre o número de escritas e a taxa de acerto (*hit ratio*).

As próximas Seções deste trabalho estão estruturadas da seguinte forma. A Seção 2 revisa brevemente alguns dos trabalhos relacionados. A Seção 3 descreve a abordagem proposta. Em seguida, a Seção 4 analisa os resultados experimentais. Por fim, a Seção 5 apresenta as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

LRU e MRU (Most recently used) são estratégias de substituição que podem ser implementadas usando uma lista. Quando uma página é acessada ou uma nova página é inserida, ela é movida para o topo da lista, indicando que foi a última página referenciada. Podemos escolher a página menos recente (LRU) ou a mais recente (MRU) como vítima. A estratégia LFU usa um contador de referências em cada página (incrementado a cada acesso) e escolhe a vítima com menor número de referências.

ARC foi projetado para dar suporte às mudanças de padrões da carga de trabalho. O algoritmo usa duas listas LRU para equilibrar a frequência e recência e mais duas listas LRU como históricos *(*ghosts) usadas para suportar o mecanismo de adaptação alterando dinamicamente os tamanhos das listas com a escolha das vítimas [Megiddo and Modha 2003]. Propostas para mídias assimétricas como H-ARC [Fan et al. 2014] e SCMBP-SCCW [Tavares 2015] usam os benefícios do mecanismo de adaptação do ARC para também se adaptar às mudanças na carga de trabalho. LIRS foi baseado no princípio *Inter-Reference Recency* (IRR). Considerando uma página P, o valor IRR(P) pode ser calculado como o número de acessos distintos de outras páginas entre dois acessos consecutivos de P. O LIRS classifica as páginas em dois status, LIR *(Low IRR)* e HIR *(High IRR)*. Ele elimina as páginas HIR do buffer primeiro, já que tendem a

não serem re-acessadas segundo o algoritmo [Jiang and Zhang 2002].

CFLRU foi a primeira estratégia de substituição projetada para mídia assimétrica. O CFLRU tenta retirar primeiro as páginas de leitura *(clean)*, mitigando o custo das operações de escrita *(dirty)* [Park et al. 2006]. Abordagens como CFDC [Ou et al. 2009] e CCF-LRU [Li et al. 2009], aprimoram o algoritmo CFLRU, porém apresentam um problema de *cache starvation*, no qual uma região de páginas (no caso, leitura) fica tão pequena devido aos mecanismos do algoritmo que é incapaz de crescer novamente. Além disso, O CFDC usa um mecanismo de clusterização de páginas de escrita, que pode impactar na complexidade do algoritmo. LLRU, outra abordagem para mídia assimétrica, divide o buffer em quatro listas LRUs, que combinam os aspectos de recência, frequência, leitura e escrita. Na seleção da vítima o algoritmo escolhe a página LRU com o menor custo de substituição entre as quatro listas. LLRU apresenta características para intensificar o *hit ratio* e diminuir o número de escritas [He et al. 2017]. Porém, apresenta o problema de *cache starvation*, corrigido pela sua variante AM-LRU [Wu et al. 2019].

3. EBRES: Um Gerenciador de *Buffer* Inteligente

Técnicas de aprendizagem de máquina que executam múltiplos algoritmos simultaneamente [Vietri et al. 2018] [Rodriguez et al. 2021] ou redes neurais [Choi and Park 2022] apresentam complexidade computacional significativa. No contexto de gerenciamento de buffer, é fundamental que as políticas de substituição tenham uma complexidade computacional baixa.

EBRES é uma política inteligente de substituição de páginas em *buffer*. A ideia é fazer uso do modelo de predição de suavização exponencial com uma baixa sobrecarga no sistema para tentar identificar páginas importantes no buffer, melhorando o *hit ratio*. A suavização exponencial foi usada em [Levandoski et al. 2013], no contexto de bancos de dados em memória principal para classificar dados (frios e quentes) predizendo o seu número de acessos (frequência). No caso específico do EBRES, a ideia é utilizar o modelo para predizer quando será o próximo acesso de uma página P e, com isso, basear a decisão de evicção. EBRES também lida com mídia assimétrica, buscando reduzir o número de escritas.

3.1. A suavização exponencial no contexto de gerenciamento de buffer

As requisições geradas aos gerenciadores de buffer podem ser representadas por uma série temporal. Em diversos casos, existe uma relação entre páginas requisitadas e o tempo de requisição. Modelos de predição de séries temporais se baseiam em dados históricos, para prever algum evento num tempo futuro. A suavização exponencial é um modelo de predição de séries temporais que utiliza uma equação de médias móveis simples, ponderadas exponencialmente. O modelo é capaz de ser treinado apenas com as observações da própria série de dados [Veríssimo et al. 2013].

A Figura 1 apresenta um exemplo de conjunto de requisições de páginas feita a um gerenciador de buffer. Cada página pode produzir uma série temporal e indica uma estimativa de quando essa página será acessada no futuro próximo. Selecionando apenas a página de Id 4, que aparece nos tempos 1, 4, 5, 8 e 100, podemos calcular as distâncias *(D)* entre as requisições subsequentes da página 4. O modelo de suavização exponencial lida com o conjunto dos valores de distância (3, 1, 3 e 92) de maneira ponderada para

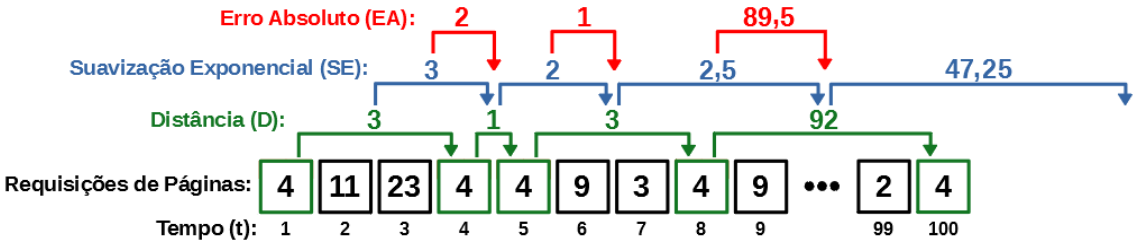

Figura 1. Exemplo de suavização exponencial em requisições de páginas

cada distância. Considera-se que as distâncias mais recentes recebam um peso maior, que declina exponencialmente até as distâncias mais antigas. Logo, as requisições muito antigas da página perdem uma parte da sua influência na predição. Há diversos métodos de suavização exponencial: simples, dupla (método de Holt), e tripla (método de Holt-Winters). Este trabalho foca no método simples usando a Fórmula $SE_t = SE_{t-1} + \alpha(D_t - SE_{t-1})$ [Gardner Jr 2006] adaptada para esse contexto, onde, SE_t é o valor esperado da predição, D_t é a distância observada da série temporal no período *t*, SE_{t-1} é o valor da predição calculada anteriormente, e α (fator de suavização, varia entre 0 e 1) é um parâmetro de ajuste para calibrar a suavização.

EBRES aplica a suavização exponencial simples como suporte à escolha de vítima (ver Seção 3.4). Considere a página 4 nos tempos 1, 4, 5, 8, e 100, com $\alpha = 0,5$. No tempo 1, não é possível calcular o SE_1, pois não temos outra requisição para calcular a distância da página 4. No tempo 4, como não foi calculado nenhuma predição anterior (SE_{t-1}) vamos assumir para esse contexto aplicado que o valor da predição será igual a distância ($SE_4 = 3$). No tempo 5, temos a distância $D_5 = 1$ e a predição anterior (SE_4) assumimos que é 3. Logo, podemos aplicar o modelo $SE_5 = 3 + \alpha(1 - 3) = 2$. No tempo 8, temos a distância $D_8 = 3$ e a predição anterior $SE_5 = 2$. Logo, aplicando o modelo $SE_8 = 2 + \alpha(3 - 2) = 2,5$. No tempo 100, temos a distância $D_{100} = 92$ e a predição anterior $SE_8 = 2,5$. Logo $SE_{100} = 2,5 + \alpha(92 - 2,5) = 47,25$. Podemos avaliar o erro das predições usando o método do erro absoluto *(EA)*, subtraindo o último valor predito com a distância atual $EA = |SE_{t-1} - D_t|$. Como o valor de *SE* no tempo 100 é grande, isto significa que a página não foi muito acessada ultimamente. Para o contexto de gerenciamento de buffer as páginas menos recentes com predições e erros maiores podem ser vistas como candidatas a vítimas primeiramente.

3.2. Fluxo de execução

EBRES utiliza quatro listas (Figura 2) e duas listas histórico *(ghost)* que armazenam apenas metadados. O histórico *IN* com as informações de entrada das últimas páginas no buffer. O histórico *OUT* com registros de saída de páginas vítimas. As quatro listas principais possuem tamanhos dinâmicos ajustados em tempo de execução. As duas listas histórico usam os parâmetros de ajuste *IN_SIZE* e *OUT_SIZE* com tamanhos fixos. EBRES usa quatro parâmetros de ajuste de tamanho mínimo para cada lista principal, desta forma evitando o problema do *cache starvation*. EBRES introduz o conceito de frequência de leitura e escrita ($read\_frequency$ e $write\_frequency$), que é um mecanismo que controla dinamicamente o tamanho das listas, usando a coleta de estatísticas com o histórico *IN* para calcular as duas frequências (ver Seção 3.3). Cada uma das qua-

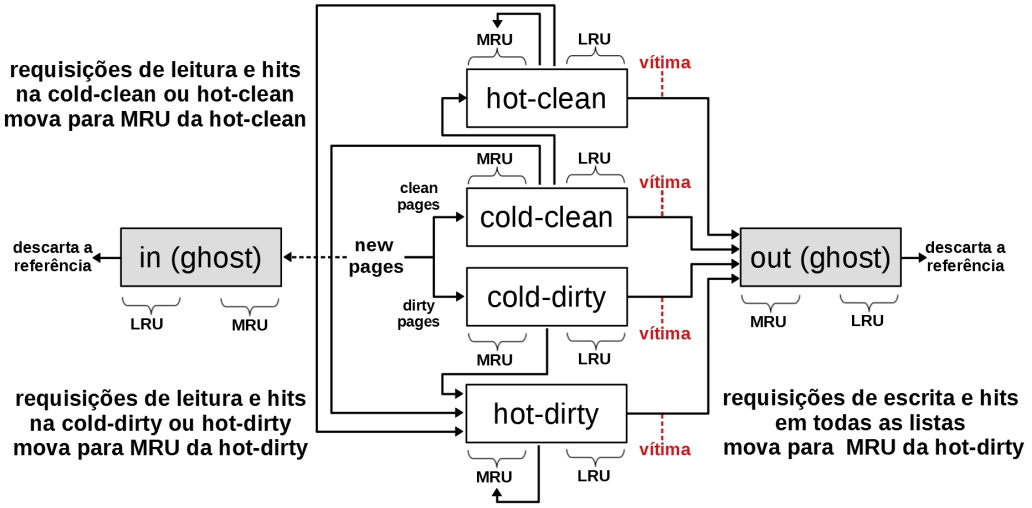

Figura 2. EBRES fluxo de execução

tro listas terá uma variável que determina o seu tamanho desejável (ver Algoritmo 2). Quanto mais a frequência (leitura ou escrita) se aproxima de zero, menor será o tamanho desejável das listas correspondentes. Chegando a zero, isso não significa que teremos um problema de *cache starvation* porque a partir do momento que a carga de trabalho muda, as frequências também mudam, fazendo com que as listas vazias cresçam novamente.

A Figura 2 descreve o fluxo de execução do algoritmo EBRES. A estratégia usa um contador global *(GC)* que é incrementado a cada requisição ao buffer. Para cada requisição, é inserido um registro no histórico *IN* com as informações do tipo da operação (leitura ou escrita) e se ocorreu um *hit* ou *miss*. As novas páginas são inseridas nas listas *cold-clean* ou *cold-dirty* dependendo do tipo da operação. Em cada página é armazenado cinco metadados extras e eles são inicializados da seguinte forma: *references* recebe o valor 1, *last* recebe o valor atual do *GC*, *list* recebe um enumerador indicando em qual das listas *cold* a nova página reside, *SE* e *error* recebem o valor 0. Quando ocorre *hit* em alguma das quatro listas principais temos três casos. Caso 1, requisições de leitura e *hits* na *cold-clean* ou *hot-clean* as páginas são movidas para a MRU da *hot-clean*. Caso 2, requisições de leitura e *hits* na *cold-dirty* ou *hot-dirty* as páginas são movidas para a MRU da *hot-dirty*. Caso 3, requisições de escrita e *hits* em qualquer uma das quatro listas principais as páginas são movidas para a MRU da *hot-dirty*.

Sempre que ocorre um *hit* em uma página o valor do campo *references* é incrementado e o enumerado do campo *list* é atualizado para a lista atual em que a página reside. Em seguida atualizamos os valores dos campos *SE*, *error* e *last* usando o Algoritmo 1, onde α se refere ao fator de suavização. Quando ocorre um *hit* no histórico *OUT*, primeiramente é preciso escolher uma vítima (ver Seção 3.4). Posteriormente, a estratégia recupera os metadados salvos dessa página *ghost* e carrega os dados da página na memória, assim como ARC e outras abordagens semelhantes. Além disso, também é executado o Algoritmo 1 para essa página.

3.3. Coletor de estatísticas

O histórico *IN* mantém informações de todas as últimas *IN_SIZE* requisições feitas ao buffer. A estrutura funciona como um fila, caso o tamanho máximo *IN_SIZE* do histórico

Algoritmo 1: SE(p)

Entrada: Uma requisição para calcular o *SE* de uma página p

1 $distance \leftarrow GC - p.last$;
2 **se** $p.SE$ ***is 0*** **então**
3 $\quad p.SE \leftarrow distance$;
4 **senão**
5 $\quad p.error \leftarrow \mid distance - p.SE \mid$;
6 **fim**
7 $p.SE \leftarrow p.SE + \alpha * (distance - p.SE)$;
8 $p.last \leftarrow GS$;

seja atingido, a entidade mais antiga (LRU) é descartada. Cada entidade do histórico mantém duas *flags*: *is_miss* e *is_dirty*, que informam se o histórico dessa requisição tenham sofrido um *hit* ou *miss* e se foi uma operação de leitura ou escrita.

Para monitorar as últimas *IN_SIZE* requisições, basta percorrer as entidades do histórico e realizar uma contagem das ocorrências, em seguida podemos extrair quatro estimativas que variam de 0 a 1. As estimativas $read\_frequency$ e $write\_frequency$ informam a porcentagem de requisições de leitura e escrita, respectivamente. As estimativas $read\_miss\_frequency$ e $write\_miss\_frequency$ informam a porcentagem de requisições de leitura e escrita, respectivamente que tiveram um *miss*. O percorrimento de *IN_SIZE* entidades a cada chamada torna o procedimento inviável. Para mitigar esse problema, EBRES executa o procedimento de análise assincronamente ao gerenciador de buffer, mantendo uma complexidade constante durante o fluxo de execução da estratégia. O procedimento é executado em intervalos de tempo ou de requisições.

3.4. Escolha da vítima

Algoritmos como o LLRU e AM-LRU usam parâmetros de ajuste para determinar o custo das operações de leitura e escrita. É possível escolher esses custos usando as informações da velocidade de acesso da própria mídia assimétrica ou de maneira mais subjetiva sugerindo ao algoritmo que o custo de escrita é maior que o de leitura. EBRES faz uso deste último e utiliza os parâmetros $write\_cost$ e $read\_cost$, ambos variam de 0 a 1.

A Equação 1 mostra o cálculo do custo de uma página *p*. O $page\_score$ é calculado combinando a predição da suavização exponencial e o seu erro decompostos pelo o número de referências, $page\_score \leftarrow (p.SE + p.error) \div p.references$. Logo, quanto maior o número de referências da página, menor será o custo. Porém, quando a página para de receber acessos, seu $p.SE$ e $p.ERROR$ aumentam e consequentemente aumentam o custo, indicando que essa página deixou de ser frequente.

$$cost(p) = \begin{cases} page\_score * write\_frequency + (1 - write\_cost), & \text{Se p é dirty.} \\ page\_score * read\_frequency + (1 - read\_cost), & \text{Se p é clean.} \end{cases} \quad (1)$$

O Algoritmo 2 descreve o processo de escolha da vítima. Primeiramente (linhas 1 e 2), selecionamos as quatro páginas LRU candidatas a vítima de cada uma das listas principais e guardamos suas referências no *array* $lru\_pages[\,]$. O procedimento $calc\_desirable\_sizes()$ (linha 3) calcula o tamanho desejável das quatro listas

Algoritmo 2: get_victim()

Entrada: Uma requisição para escolher uma vítima
Saída: Retorna a página vítima (*victim*)

```
1  P_CC ← cold-clean.LRU;
2  P_CD ← cold-dirty.LRU;
3  P_HC ← hot-clean.LRU;
4  P_HD ← hot-dirty.LRU;
5  lru_pages[ ] ← {P_CC, P_CD, P_HC, P_HD} ;
6  calc_desirable_sizes()/* calcula os tamanhos desejáveis      */
7  save_from_eviction()/* não substitui as páginas LRUs de
       tamanho ≤ desejável                                      */
8  se read_miss_frequency > 0 então
9      se cold-clean.size > BUFFER_SIZE * read_miss_frequency * read_cost
        então
10         retorna P_CC
11     fim
12 fim
13 se write_miss_frequency > 0 então
14     se cold-dirty.size > BUFFER_SIZE * write_miss_frequency *
        write_cost então
15         retorna P_CD
16     fim
17 fim
18 victim ← null
19 para i = 0; i < 4; i++ faça
20     se lru_pages[i] ≠ null então
21         se victim is null ou cost(lru_pages[i]) > cost(victim) então
22             victim ← lru_pages[i]
23         fim
24     fim
25 fim
26 retorna victim
```

usando os parâmetros de ajuste dos tamanhos mínimos e as estatísticas de frequência das operações obtidas pelo histórico *IN*. Por exemplo, no caso da lista *cold-clean*: $desirable\_cold\_clean$ = *MIN_COLD_CLEAN* * $read\_frequency$. Em seguida, o procedimento $save\_from\_eviction()$ (linha 4) evita que as páginas das listas com seu tamanho menor ou igual ao tamanho desejável sejam substituídas, basta atribuir nulo na posição do *array* das LRU correspondente. Por exemplo, no caso da *cold-clean*: Se *cold-clean.size* $\leq desirable\_cold\_clean$, então $lru\_pages[0] = null$.

Nas linhas 5 e 10, temos dois casos especiais, com a intenção de proteger as páginas que residem as listas *hot* de acessos sequenciais. EBRES usa as frequências de erro *(miss)*, o custo das operações e a capacidade do buffer $(BUFFER\_SIZE)$ para identificar esse padrão de acesso, removendo imediatamente as páginas LRU de uma das listas *cold*. Na linha 16 o algoritmo escolhe uma das quatro candidatas a vítimas guardadas no *array*. Para cada candidata diferente de nulo, iremos calcular o custo da página usando

a Equação 1. Finalmente o algoritmo escolhe a página com o maior custo como vítima. Páginas escolhidas como vítimas são despejadas do buffer e inseridas no histórico *OUT*. Caso o tamanho do histórico *OUT* chegue a *OUT_SIZE*, a sua entidade mais antiga (LRU) é descartada da lista.

4. Avaliação Experimental

Nesta seção, apresentamos os experimentos que foram realizados e os resultados obtidos.

4.1. Configuração do Ambiente de Experimentação

Para avaliar o EBRES utilizamos três *traces* reais: TPC-C [Association 2007b], TPC-E and [Association 2007c] e File Server [Association 2007a]. A Tabela 1 apresenta as principais características de cada *trace*. O número de requisições representa a quantidade total de operações (leitura ou escrita) que acessaram páginas do banco de dados. Do número total requisições, separou-se a quantidade de acesso para leitura, a quantidade para escrita. As páginas acessadas informam a quantidade de dados acessados do tamanho total do banco de dados do *trace*. Os dois *traces* ZIPF [Tavares 2015] são sintéticos, gerados com 20% e 80% de operações de escrita. Para garantir uma experimentação correta, foi definido que as páginas dos bancos de dados têm tamanho de 4KB.

Nome	Requisições	Leitura	Escrita	Páginas Acessadas	Tamanho do BD em páginas
File Server	1.175.607	790.070	385.537	487.495 (1,9 GB)	2.778.458 (11,3 GB)
TPC-C	3.000.000	2.811.727	188.273	919.367 (3,7 GB)	6.929.239 (28,3 GB)
TPC-E	3.000.000	2.864.708	135.292	1.482.576 (6,0 GB)	5.264.225 (21,5 GB)
ZIPF20%W	1.500.000	1.200.000	300.000	285.281 (1,1 GB)	999.993 (4,0 GB)
ZIPF80%W	1.500.000	300.000	1.200.000	285.281 (1,1 GB)	999.993 (4,0 GB)

Tabela 1. Descrição dos *traces* usados

Foi implementado um gerenciador de arquivos e gerenciador de buffer [1] em linguagem C, capaz de realizar chamadas diretas ao sistema operacional e emular o processamento das transações usando os *traces*. O gerenciador de buffer encapsula diferentes políticas de substituição de páginas, onde em cada experimento apenas uma é ativada. Para comparação, usamos os algoritmos LRU, ARC e LIRS, e CFDC e AM-LRU. Estes têm foco em considerar a assimetria da mídia de armazenamento, com relação a operações de leitura e de escrita. Os experimentos foram executados em uma máquina de 64 GB de RAM e 480 GB de SSD Kingston SA400S37480G com Ubuntu Linux 18.04 LTS Kernel 4.15.0-74-generic, Intel i7-9700k 3,60 GHz.

4.2. Experimentos

Para todos os algoritmos, foram executados os *traces*, variando o tamanho do buffer (em páginas) em 0,1%, 1%, 2%, 5% e 10% da quantidade das páginas acessadas (eixo x, nos gráficos). Computamos o tempo de execução em segundos, o número de *hits* e escritas (eixo y, nos gráficos). Como já mencionado, o gerenciador de arquivos usou o tamanho do bloco de dados de 4KB. Como parâmetro de ajuste, no CFDC foi usado 50% do tamanho do buffer na *working region*. Para o LIRS, 1% do tamanho do buffer para páginas do tipo

[1] O código dos gerenciadores implementados pode ser acessados em github.com/ggustavo/SNK-DB

HIR. Para o AM-LRU, o *min_len* foi usado 10% do tamanho do buffer. Para o EBRES, o *IN_SIZE* foi configurado como 10% do tamanho do buffer, *OUT_SIZE* igual ao tamanho do buffer, os tamanhos mínimos das quatro listas principais como 10% do tamanho do buffer, e os custos $read\_cost$ e $write\_cost$, 0,2 e 0,8. Por fim, o α como 0,5.

Diversos fatores podem influenciar o tempo de execução dos experimentos (ver Figuras 5, 8, 11, 14, 17): fatores externos, como o sistema operacional e procedimentos internos do SSD podem contribuir com variações de tempo, e internos, como operações de escritas e de leitura (*miss*), fazem com que o tempo de execução aumente, pois o sistema realiza acessos diretos a mídia de armazenamento. Outro fator é a complexidade dos algoritmos, por exemplo o CFDC em vários cenários apresenta um tempo de execução elevado em comparação aos outros algoritmos. Acessos randômicos, podem prejudicar seu mecanismo de *clusters*, fazendo com que o CFDC precise manter a ordem das prioridades de vários *clusters* de apenas uma página. No LIRS, a operação de *stack pruning* e no AM-LRU, o mecanismo de segunda chance para páginas frequentes, quando executados sucessivas vezes, podem impactar diretamente o tempo de execução. Já as estratégias ARC, LRU, e EBRES como possuem uma complexidade constante, seu principal impacto no tempo de execução são as operações internas de leitura e escrita e os fatores externos.

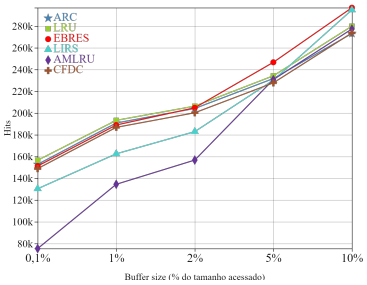

Figura 3. File Server - *hits*

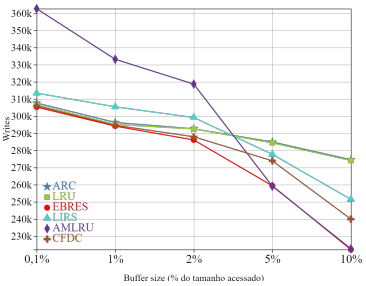

Figura 4. File Server - escritas

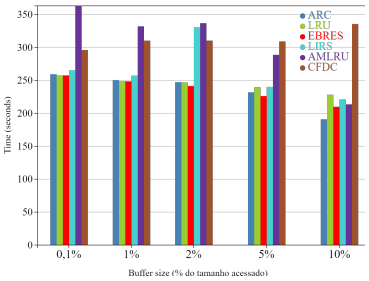

Figura 5. File Server - tempo

Para o File Server (Figuras 3, 4 e 5), temos uma carga de trabalho real de 67,21% de leituras e 32,79% de escritas. LIRS e AM-LRU possuem políticas mais sensíveis para páginas recentes e é possível que isso possa impactar os *hits* e escritas para tamanhos de buffer menores entre 0,1% e 2%, já que essas páginas podem precisar de um tempo de vida maior no buffer para serem acessadas novamente e consideradas frequentes. Para os demais algoritmos, nos tamanhos menores 0,1% e 2%, apresentam um número de *hits* equivalentes. EBRES, para todos os tamanhos, obteve um desempenho equilibrado de *hits* e número baixo de escritas, todavia no tamanho de 10% obteve o melhor desempenho nestes dois quesitos. No tamanho de 0,1% fica perceptível que o AM-LRU apresentou um elevado tempo de execução, principalmente por conta do seu baixo número de *hits* e número elevado de escritas. O mesmo ocorre com o CFDC, porém é provável que a causa seja a degradação do mecanismo de clusterização.

Para o TPC-C (Figuras 6, 7 e 8), temos uma carga de trabalho majoritariamente de leitura (93,72%). No número de *hits*, EBRES obteve um desempenho parcialmente inferior ao LIRS em todos os tamanhos, em especial nos tamanhos de buffer entre 0,1% e 1%, entretanto, a diferença média ficou em torno de 7%. Por outro lado, EBRES obteve o menor número de escritas em todos os tamanhos, exceto nos tamanhos de 0,1% e 10%, em que o CFDC atinge um melhor resultado, variando em apenas 0,2% no tamanho 0,1% e

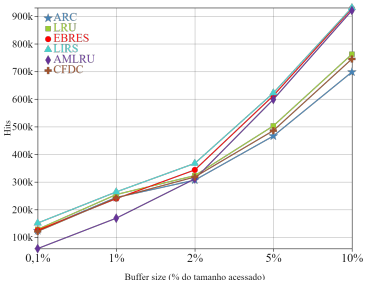

Figura 6. TPC-C - *hits*

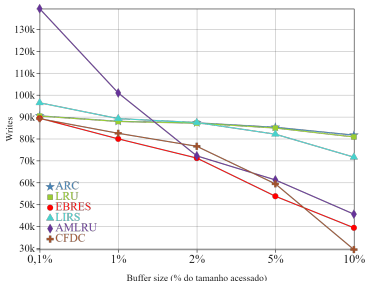

Figura 7. TPC-C - escritas

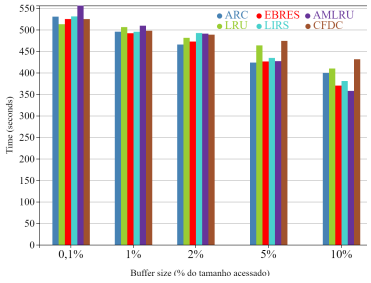

Figura 8. TPC-C - tempo

33,93% no tamanho de 10%. Para os algoritmos com complexidade constante ARC, LRU e EBRES, o tempo de execução diminui a medida que o tamanho do buffer aumenta.

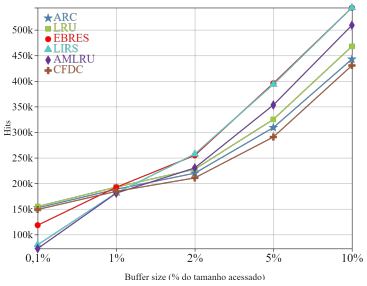

Figura 9. TPC-E - *hits*

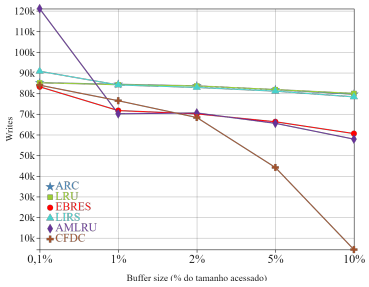

Figura 10. TPC-E - escritas

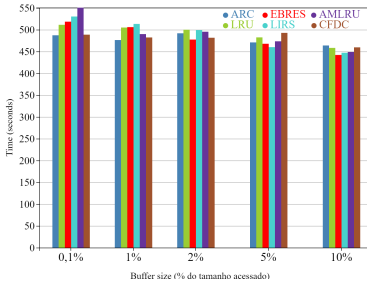

Figura 11. TPC-E - tempo

Para o TPC-E (Figuras 9, 10 e 11), EBRES apresenta um desempenho mediano comparado com os demais algoritmos no tamanho de 0,1%, no número de *hits*. Entre os tamanhos de 1% e 10% EBRES mantém um desempenho semelhante ao LIRS e AMLRU, em termos de *hits* e escritas, respectivamente. Embora o CFDC tenha obtido o menor número de escritas, para os tamanhos de buffer de 2% ou maior, com uma diferença significativa dos demais algoritmos no tamanho de 10%, ele teve o menor número de *hits*, com uma leve diferença do ARC. O problema do *cache starvation* pode ter impactado para diminuir o número de escritas. Como a carga de trabalho é majoritariamente de leitura (95,49%), a região de leitura da *priority region* é incapaz de crescer quando a região de escrita cresce e toma seu espaço, o CFDC foi capaz de manter aproximadamente 50% do seu espaço do buffer apenas para páginas de escrita, reduzindo significativamente o número de escritas, isso intensifica ainda mais, caso essas páginas de escrita sejam frequentes e possam residir na *working region*.

Para o ZIPF 20%W (Figuras 12, 13 e 14), para todos os tamanhos de buffer, AMLRU apresentou em média 11,95% menos escritas que o EBRES e uma variação média de 1,69% nos *hits* em relação ao EBRES. Seu tempo de execução também foi decrescente a medida que o tamanho do buffer aumenta. Parte das páginas classificadas como frequentes pelo LIRS também são páginas de escritas, mesmo a estratégia não considerando o tipo de operação, ela tende a ter bons resultados. Um dos impactos negativos do CFDC, durante os testes é devido a sua prioridade elevada em manter várias páginas de escrita no buffer, mesmo que hipoteticamente não sejam frequentes.

Os dois workloads sintéticos ZIPF de acessos randômicos apresentam a mesma ordem de requisições variando apenas o tipo da operação. Logo, nota-se que temos um

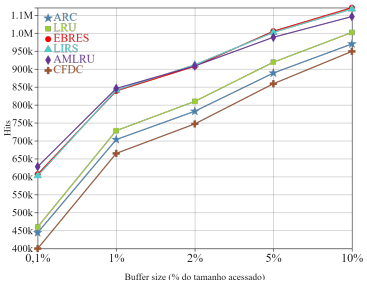

Figura 12. ZIPF 20%W - *hits*

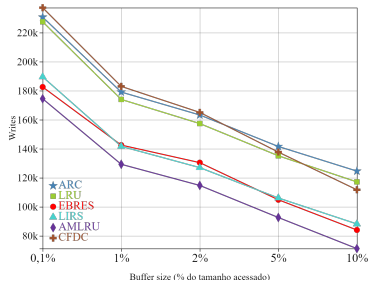

Figura 13. ZIPF 20%W - escritas

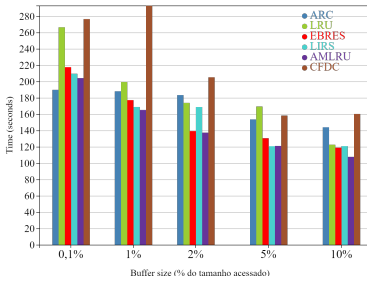

Figura 14. ZIPF 20%W - tempo

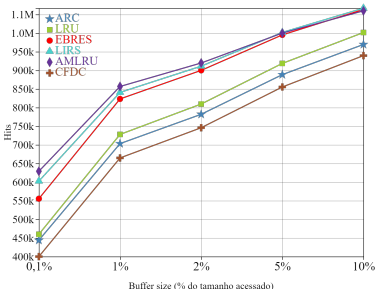

Figura 15. ZIPF 80%W - *hits*

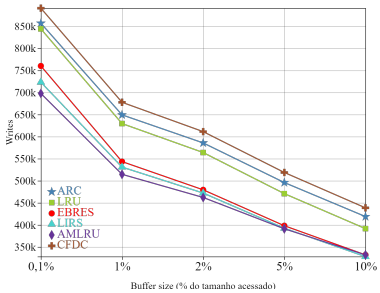

Figura 16. ZIPF 80%W - escritas

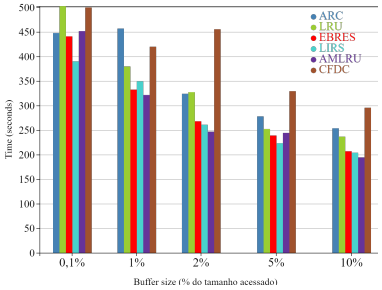

Figura 17. ZIPF 80%W - tempo

comportamento bem similar em relação ao visual dos gráficos, porém diferem nos resultados, por exemplo o número de escritas. Para o *trace* ZIPF 80%W (Figuras 15 e 16), os algoritmos LRU, ARC e CFDC apresentam um menor rendimento do que os algoritmos AM-LRU, LIRS e EBRES. Para esta carga com maioria de operações de escrita, temos uma influência maior dessas operações no tempo de execução. Algoritmos que realizaram mais escritas, tendem apresentar um tempo de execução um pouco maior. AM-LRU, LIRS e EBRES a partir do tamanho buffer de 2% apresentam resultados semelhantes.

5. Conclusão

EBRES é uma proposta de estratégia de substituição de páginas que relaciona as requisições feitas ao buffer como uma série temporal individual para cada página. Ele provê uma complexidade constante, para isso, usamos um método simples de aprendizagem de máquina chamado suavização exponencial aplicado para cada página no buffer. EBRES coleta estatísticas como a frequência de leitura e escrita para identificar padrões de acesso e adaptar suas estruturas de dados. Durante os experimentos, EBRES se destaca dos demais algoritmos por manter um equilíbrio nos diferentes *traces*. Os algoritmos de complexidade não constante AM-LRU, LIRS e CFDC apresentam comportamentos bem mais variantes no quesitos de *hits*, escritas, e tempo de execução. Já algoritmos constantes LRU e ARC, embora possam ser competitivos no número de *hits* apresentam dificuldades com o número de escritas. Como trabalhos futuros, temos um desafio de projetar mecanismos que ajustem dinamicamente os parâmetros do EBRES durante a sua execução, como o α e os tamanhos mínimos das quatro listas principais.

Referências

[Association 2007a] Association, S. N. I. (2007a). Microsoft Storage File Server Traces. `http://iotta.snia.org/traces/158`. [Online; accessed 04-October-2014].

[Association 2007b] Association, S. N. I. (2007b). TPC-C SNIA Traces. `http://iotta.snia.org/traces/131`. [Online; accessed 10-October-2014].

[Association 2007c] Association, S. N. I. (2007c). TPC-E SNIA Traces. `http://iotta.snia.org/traces/133`. [Online; accessed 10-October-2014].

[Choi and Park 2022] Choi, H. and Park, S. (2022). Learning future reference patterns for efficient cache replacement decisions. *IEEE Access*, 10:25922–25934.

[Effelsberg and Härder 1984] Effelsberg, W. and Härder, T. (1984). Principles of database buffer management. *ACM Trans. Database Syst.*

[Fan et al. 2014] Fan, Z., Du, D. H. C., and Voigt, D. (2014). H-arc: A non-volatile memory based cache policy for solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11.

[Gardner Jr 2006] Gardner Jr, E. S. (2006). Exponential smoothing: The state of the art—part ii. *International journal of forecasting*, 22(4):637–666.

[Harizopoulos et al. 2008] Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M. (2008). Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD*, SIGMOD '08.

[He et al. 2017] He, J., Jia, G., Han, G., Wang, H., and Yang, X. (2017). Locality-aware replacement algorithm in flash memory to optimize cloud computing for smart factory of industry 4.0. *IEEE Access*, 5:16252–16262.

[Hellerstein et al. 2007] Hellerstein, J. M., Stonebraker, M., and Hamilton, J. (2007). Architecture of a database system. *Found. Trends databases*, 1(2):141–259.

[Jiang and Zhang 2002] Jiang, S. and Zhang, X. (2002). Lirs: An efficient low interreference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, page 31–42, New York, NY, USA. Association for Computing Machinery.

[Levandoski et al. 2013] Levandoski, J. J., Larson, P.-Å., and Stoica, R. (2013). Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 26–37. IEEE.

[Li et al. 2009] Li, Z., Jin, P., Su, X., Cui, K., and Yue, L. (2009). Ccf-lru: a new buffer replacement algorithm for flash memory. *IEEE Transactions on Consumer Electronics*, 55(3):1351–1359.

[Megiddo and Modha 2003] Megiddo, N. and Modha, D. S. (2003). Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*.

[Ou et al. 2009] Ou, Y., Härder, T., and Jin, P. (2009). Cfdc: A flash-aware replacement policy for database buffer management. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, page 15–20, New York, NY, USA. Association for Computing Machinery.

[Park et al. 2011] Park, S., Kim, Y., Urgaonkar, B., Lee, J., and Seo, E. (2011). A comprehensive study of energy efficiency and performance of flash-based ssd. volume 57, pages 354–365.

[Park et al. 2006] Park, S.-y., Jung, D., Kang, J.-u., Kim, J.-s., and Lee, J. (2006). Cflru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, page 234–241, New York, NY, USA. Association for Computing Machinery.

[Rodriguez et al. 2021] Rodriguez, L. V., Yusuf, F., Lyons, S., Paz, E., Rangaswami, R., Liu, J., Zhao, M., and Narasimhan, G. (2021). Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. USENIX Association.

[Tavares 2015] Tavares, J. A. (2015). *Database Buffer Management Strategies for Asymmetric Media*. Master dissertation, Universidade de Fortaleza - Unifor.

[Veríssimo et al. 2013] Veríssimo, A. J., da Cunha Alves, C., Henning, E., do Amaral, C. E., and da Cruz, A. C. (2013). Métodos estatísticos de suavização exponencial holt-winters para previsão de demanda em uma empresa do setor metal mecânico. *Revista Gestão Industrial*, 8(4).

[Vietri et al. 2018] Vietri, G., Rodriguez, L. V., Martinez, W. A., Lyons, S., Liu, J., Rangaswami, R., Zhao, M., and Narasimhan, G. (2018). Driving cache replacement with ml-based lecar. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'18, page 3, USA. USENIX Association.

[Wu et al. 2019] Wu, X., Cai, D., and Guan, S. (2019). A multiple LRU list buffer management algorithm. *IOP Conference Series: Materials Science and Engineering*, 569:052002.