Análise de desempenho e propostas de melhoria para as Transparent Huge Pages do kernel Linux

Pedro Demarchi Gomes[1], **Islene Calciolari Garcia**[1]

[1] Instituto de Computação
Universidade Estadual de Campinas (UNICAMP) – Campinas, SP – Brasil

Abstract. *Most modern operating systems use huge pages to reduce address translation overhead in applications that use large amounts of memory. This paper explores the use of huge pages in Linux, showing their behavior in different situations so that one can have a better understanding of the tradeoffs involved. This makes it possible to identify the cause of several problems that occur with the use of huge pages, such as memory bloat and increased latency on page faults. Based on this information, this paper presents the LRBHPS (Linux reservation-based huge pages system), a new system based on Linux to correct/mitigate these problems.*

Resumo. *Grande parte dos sistemas operacionais modernos usam huge pages para reduzir o overhead de tradução de endereços em aplicações que utilizam grandes quantidades de memória. Esse trabalho explora o uso de huge pages no Linux, mostrando seu comportamento em diferentes situações para que se possa ter um melhor entendimento dos tradeoffs envolvidos. Com isso é possível apontar a causa de diversos problemas que ocorrem com o uso de huge pages, como memory bloat e aumento de latência nas faltas de página. A partir dessas informações, esse trabalho apresenta o LRBHPS (Linux reservation-based huge pages system), um novo sistema baseado no Linux para corrigir/mitigar esses problemas.*

1. Introdução

A paginação é uma técnica de gerenciamento de memória utilizada desde a década de 1960, com o objetivo de implementar o conceito de memória virtual. Ela foi elaborada em uma época em que o principal problema a ser atacado era a escassez de memória RAM. Na paginação, o espaço de endereçamento é dividido em páginas de tamanho fixo, usualmente 4K, e as páginas são mapeadas para a memória física de acordo com a execução dos programas.

Uma tabela é usada para mapear as páginas virtuais para páginas físicas, chamada de tabela de páginas. Normalmente, o endereço físico gerado pela tradução é utilizado para acessar a memória principal, com o conteúdo correspondente sendo retornado à CPU. Um 'acerto' (*hit*) na tabela de páginas refere-se ao caso em que a tradução está presente na tabela.

Com o objetivo de acelerar o processo de tradução de endereços, os processadores se valem de uma *Translation Lookaside Buffer* (TLB). A TLB é uma memória cache que armazena as traduções dos endereços virtuais acessados mais recentemente. Ela possui uma alta velocidade de acesso, mas baixa capacidade de armazenamento, e serve

para manter dados que foram utilizados recentemente na esperança de que voltem a ser utilizados em um futuro próximo. Para traduzir um endereço virtual, inicialmente busca-se este endereço na TLB. Se encontrado, retorna-se a tradução diretamente para a CPU (um fenômeno conhecido como TLB *hit*). Caso contrário, um TLB *miss* acontece, e o endereço é procurado na tabela de páginas, sendo, em seguida, inserido na TLB.

O tamanho da memória física dos computadores vem crescendo cada vez mais com o passar dos anos, bem como o uso de memória das aplicações. Isso causa um maior *overhead* na tradução de endereços, pois acarreta um maior número de TLB *misses*. [Hornyack et al. 2013] mostra que o tempo gasto para tratar TLB *misses* pode chegar a 58% do tempo de execução da aplicação. Para mitigar esse problema, os sistemas operacionais e processadores passaram a utilizar páginas maiores, chamadas de *huge pages*, visando cobrir mais endereços de memória por entradas na TLB e, com isso, reduzir o número de TLB *misses* das aplicações. Diferentes processadores aceitam diferentes tamanhos de *huge pages*, porém foram implementados nos sistemas operacionais Linux e FreeBSD apenas *huge pages* de 2 MB de tamanho.

Os benefícios do uso de *huge pages* são óbvios, o problema que recai sobre os sistemas operacionais é como gerenciá-las eficientemente de forma transparente às aplicações, evitando aumento da fragmentação interna, o alto uso de CPU para criar uma região livre contígua de memória, e o aumento da latência nas faltas de página.

Experimentos realizados no Linux em [Zhu 2022] mostram que a utilização das *huge pages* é majoritariamente bimodal, ou seja, separando as *huge pages* pelo número de páginas base utilizadas, a grande parte irá se concentrar nos extremos, com poucas páginas utilizadas (0-50) ou quase todas (460-512). O grande número de *huge pages* com pouca utilização demonstra o problema da fragmentação interna, que nesse caso chegou a desperdiçar 2.7 GB de memória.

O uso de *huge pages* de maneira transparente às aplicações foi implementado na versão 2.6.38 do kernel Linux em 2011 [Corbet 2011], onde foram chamadas de *Transparent Huge Pages* (THP), e apesar dos esforços dos desenvolvedores do kernel para tornar seu uso eficiente, diversas aplicações ainda recomendam que as *huge pages* sejam desabilitadas para obter um melhor desempenho [Hadoop sd] [postgreSQL sd] [mongoDB sd].

Esse trabalho descreve o funcionamento das *huge pages* nos sistemas operacionais Linux e FreeBSD, e também os propostos em Ingens [Kwon et al. 2016], HawkEye [Panwar et al. 2019] e Quicksilver [Zhu et al. 2020]. São comparadas as características e resultados apresentados por cada sistema, identificando suas vantagens e desvantagens em diferentes situações. Com base nessas observações foi implementado o LRBHPS, um novo sistema baseado no Linux, que busca gerenciar o uso de *huge pages* para superar as desvantagens identificadas nos sistemas anteriores.

O LRBHPS cria uma *huge page* apenas quando ela tem uma alta probabilidade de ser utilizada, evitando o *overhead* de criação de uma *huge page* que não trará desempenho para a aplicação. Para isso, a região de memória de uma *huge page* é reservada na primeira falta de páginas, porém a *huge page* só é mapeada após um número t de páginas serem utilizadas, o que evita a criação de *huge pages* sem real necessidade de uso. As reservas que não forem promovidas após um período de tempo são destruídas, liberando memória sem causar impacto no desempenho da aplicação.

Foram testadas duas versões do LRBHPS com valores $t = 64$ e $t = 256$. O desempenho das versões foi comparado com o Linux utilizando *workloads* sintéticos e reais, onde pode-se observar uma melhora de desempenho significativa em certas aplicações, principalmente em ambientes com a memória extremamente fragmentada.

O restante deste documento está estruturado da seguinte forma. A Seção 2 apresenta a forma que as THPs são tratadas em diferentes sistemas. A Seção 3 apresenta o sistema LRBHPS implementado nesse trabalho. A Seção 4 apresenta as avaliações de desempenho e seus resultados para o sistema LRBHPS em comparação com o Linux. Por fim, a Seção 5 apresenta a conclusão do trabalho.

2. Trabalhos relacionados

Essa seção apresenta o modo como é feito o gerenciamento de THP no Linux, FreeBSD, Ingens [Kwon et al. 2016], HawkEye [Panwar et al. 2019] e Quicksilver [Zhu et al. 2020].

2.1. Linux

O Linux implementa as THPs desde a versão 2.6.38 do kernel [Corbet 2011]. Nele, as THPs são criadas apenas para regiões anônimas de memória. Uma *huge page* é alocada, criada e mapeada na primeira falta de página em uma região alinhada em 2 MB. Caso não exista uma região de memória contínua livre para a *huge page*, o comportamento padrão é compactar a memória para criá-la. Isso pode causar um aumento na latência das faltas de página em sistemas com memória muito fragmentada, e nem sempre resultar na criação de uma *huge page*, pois a compactação pode falhar em criar uma região de 2 MB de memória livre. Para impedir que a compactação prejudique o desempenho, sua falha faz com que as próximas 2^n tentativas de criar uma *huge page* não executem a compactação. O valor de n inicia em 0, e é incrementado a cada falha da compactação em criar uma *huge page*. O limite máximo de n é 6, e seu valor é zerado quando a compactação obtém sucesso em alocar uma *huge page*.

Além disso, o Linux implementa o daemon khugepaged, que escaneia de maneira assíncrona as tabelas de página dos processos, em busca de regiões anônimas alinhadas em 2 MB que contenham pelo menos uma página marcada como *dirty*, para serem juntadas para se criar uma *huge page*. Ao identificar uma região como essa, as faltas de página para essa região são bloqueadas, é alocada uma *huge page* para onde o conteúdo das páginas marcadas como *dirty* será copiado, e o restante da *huge page* é zerado. Por fim, a *huge page* é mapeada. Essa forma de alocar *huge pages* é mais lenta que a descrita anteriormente, pois bloqueia o acesso a uma região de, pelo menos, 2 MB de memória, além de causar TLB *shootdowns*. Devido a isso a configuração padrão do khugepaged determina que apenas 4096 páginas sejam escaneadas a cada 10 segundos. O khugepaged tem como objetivo alocar *huge pages* que não puderam ser alocadas nas faltas de página, seja pela falha da compactação em conseguir memória livre no momento, ou pelo tamanho da região de memória, que inicialmente não tinha tamanho o suficiente para uma *huge page*, mas foi crescendo conforme a execução da aplicação.

O Linux busca retardar ao máximo quebrar uma *huge page*. Desmapear parte de uma *huge page* não faz com que sua memória seja liberada imediatamente. Essas *huge pages* são inseridas em uma fila, para que quando a memória estiver sob pressão sejam

quebradas. Isso evita o trabalho desnecessário de quebrar *huge pages* durante a syscall exit, e também quando não há uma pressão sobre o uso de memória.

2.2. Ingens

Ingens [Kwon et al. 2016] utiliza uma abordagem de monitoramento do uso das páginas e frequência de acesso. Dessa forma, decisões como quebrar, promover e migrar páginas são tomadas de maneira mais inteligente, diminuindo a latência, fragmentação de memória, e distribuindo de forma mais justa as THPs entre os processos.

Em Ingens a promoção de páginas ocorre apenas de forma assíncrona, diferindo do Linux onde elas ocorrem de forma síncrona e assíncrona (khugepaged), com o objetivo de evitar uma alta latência na falta de página. Além disso, a promoção ocorre apenas após 90% das páginas de uma região de 2 MB serem mapeadas. A compactação de memória, utilizada pelo Ingens para manter livre porções contíguas de memória, evita migrar páginas muito acessadas para diminuir seu impacto no desempenho da aplicação. Ela é executada sempre que um limite de fragmentação é atingido, compactando, no máximo, 100 MB de memória a cada 5 segundos. Quando uma *huge page* é parcialmente desmapeada, o Ingens busca adiar a quebra dessa *huge page*, o fazendo apenas quando o número de páginas mapeadas fica abaixo de um limite.

Para distribuir as *huge pages* de forma justa entre os processos, Ingens calcula um valor de prioridade para cada processo, utilizando o número de *huge pages*, o número de páginas inativas (páginas com baixa frequência de acesso), e um número de prioridade estático definido pelo usuário para cada processo.

2.3. HawkEye

HawkEye [Panwar et al. 2019] é um sistema semelhante ao Ingens. As *huge pages* são criadas apenas de maneira assíncrona, porém o *threshold* para sua criação é de apenas uma página, assim como o Linux. Para promover as páginas de maneira mais eficiente, privilegiando as que resultarão em maior aumento de desempenho, a cada 30 segundos as regiões candidatas a serem promovidas são amostradas, limpando os *access bits* da tabela de página e testando o número de bits setados após 1 segundo, para saber o número de páginas acessadas naquele intervalo de tempo. Com isso é calculada a Média Móvel Exponencial (MME) de páginas acessadas da região, que será usada para separar as regiões em 10 baldes, que contêm regiões com uma MME de páginas acessadas semelhante.

Em HawkEye, as páginas são zeradas de maneira assíncrona, diminuindo a latência nas faltas de página, dado que as páginas já estão zeradas nesse momento. Para recuperar memória, HawkEye escaneia as *huge pages* para identificar o número de páginas base zeradas que elas contêm. Caso esse número esteja acima de um limiar, essa *huge page* é quebrada e as páginas zeradas são compartilhadas, reduzindo o uso de memória.

2.4. FreeBSD

Diferentemente do Linux, o FreeBSD cria *huge pages* para todos os tipos de memória, tanto para regiões anônimas quanto para regiões que estão mapeando arquivos ou executáveis. Aqui é utilizada uma abordagem baseada em reservas para a criação de uma *huge page* [Talluri and Hill 1994] [Navarro et al. 2003]. Ao invés de alocar, preparar e mapear uma *huge page* na primeira falta de página, como ocorre no Linux, a memória

é apenas alocada, e as páginas base são preparadas conforme são acessadas. No caso de regiões anônimas, essa preparação é feita apenas para a página base acessada, ou seja, 4 KB são preparados a cada falta de página. Já no caso de regiões que estão mapeando arquivos, 64 KB são preparados a cada falta de página, para diminuir o *overhead* de IO. Apenas quando todas as páginas forem preparadas, e suas permissões, *dirty* e *access bits* forem iguais, a *huge page* é mapeada. No caso do mapeamento de arquivos, uma *huge page* é alocada apenas se a região de memória for maior que 2 MB. Para regiões anônimas de memória é sempre tentado alocar uma *huge page*, independente do tamanho da região, pois espera-se que a região cresça no futuro, e que então se possa mapeá-la com uma *huge page*. Isso substitui uma das principais funções do daemon khugepaged no Linux [Zhu et al. 2020].

Diferentemente do Linux, no FreeBSD não é realizada a compactação direta, durante a falta de página, para se alocar uma *huge page*. Caso a memória esteja fragmentada, e não haja uma região livre de 2 MB, a reserva não é criada.

Mapear uma *huge page* somente quando todas suas páginas base forem preparadas faz com que não seja desperdiçada memória mapeando *huge pages* para regiões que não serão acessadas. Porém, não é a melhor estratégia do ponto de vista de desempenho, pois retarda o mapeamento das *huge pages*, e em regiões com grande frequência de acesso, mas apenas algumas poucas páginas não preparadas, o desempenho do mapeamento de uma *huge page* é sacrificado em troca da economia de uma pequena quantia de memória.

Desmapear parte de uma *huge page* faz com que seu mapeamento seja quebrado imediatamente, o que também ocorre caso seja preciso modificar o *dirty bit* de uma de suas páginas base. Nesse caso, o mapeamento é quebrado e o *dirty bit* daquela página é modificado. A região poderá voltar a ser mapeada com uma *huge page* assim que todas as páginas sejam marcadas como *dirty*. O FreeBSD evita quebrar *huge pages* para evitar a fragmentação. Porém, quando necessário, as reservas podem ser quebradas e as páginas não mapeadas utilizadas pelas aplicações.

2.5. Quicksilver

O Quicksilver [Zhu et al. 2020] é baseado no FreeBSD, e mantém sua principal característica: a alocação de memória baseada em reservas, que desacopla a alocação física da preparação das *huge pages*. No FreeBSD, a *huge page* é mapeada somente após a última página de memória da região reservada ser preparada. Já no Quicksilver a *huge page* é mapeada após um limiar t de páginas dentro da região serem preparadas. Isso acelera a velocidade com que as *huge pages* são criadas.

Existem duas versões do Quicksilver, com a preparação das *huge pages* de forma síncrona e assíncrona. Na síncrona a *huge page* é mapeada no tratamento da falta de página. Ao se atingir o limiar t de páginas preparadas em uma reserva, o restante das páginas da reserva é zerado e a *huge page* é mapeada. Ao invés de zerar página por página, como ocorre nos demais sistemas apresentados, é utilizado o *bulk zeroing*, que consiste em executar instruções *assembly* para zerar um pedaço de memória maior que uma página. Em [Zhu et al. 2020] foi mostrado que utilizar o *bulk zeroing* diminui o tempo para se zerar uma *huge page*. Na versão assíncrona as reservas que atingirem o limiar t de páginas preparadas são zeradas periodicamente, página por página, começando pelas reservas mais ativas. Quando todas as páginas de uma reserva forem zeradas, uma

falta de página na região fará com que seja criado o mapeamento da *huge page*.

As condições para mapear uma *huge page* também são relaxadas. No FreeBSD, os *dirty* e *access bits* das páginas de uma reserva precisam ser iguais para se criar uma *huge page*. Já no Quicksilver, para regiões anônimas de memória os dois bits são ignorados, e para regiões de memória que mapeiam arquivos apenas o *access bit* é ignorado. Isso faz com que mais *huge pages* sejam mapeadas.

Para que as reservas de 2 MB sejam alocadas, é preciso que se tenha 2 MB de memória contígua livre. Para isso o Quicksilver escaneia as reservas parcialmente populadas a cada segundo, em busca de reservas inativas. Uma reserva é considerada inativa se sua população (número de páginas mapeadas) não foi alterada por mais de 5 segundos. Quando encontradas, as reservas inativas são quebradas, e suas páginas que estão mapeadas são migradas, criando assim uma região livre de 2 MB. A migração é limitada a usar no máximo 1 GB/s de banda de memória, para evitar competir com as aplicações.

3. LRBHPS design

Essa seção apresenta os detalhes dos mecanismos empregados no LRBHPS[1] para o gerenciamento da memória. O LRBHPS é baseado no kernel Linux versão 6.1.38, e implementa um sistema de reservas e promoção semelhante ao Quicksilver. O sistema de reservas foi implementado a partir do *patch* [Yznaga 2018]. As reservas são criadas na primeira falta de página em qualquer região anônima alinhada em 2 MB. Assim como no Linux, o LRBHPS cria *huge pages* apenas em regiões anônimas.

As reservas são criadas mesmo que ultrapassem o tamanho da *Virtual Memory Area* (VMA), pois caso a VMA aumente e cubra a reserva ela poderá ser mapeada como uma *huge page*. Para implementar o sistema de reservas, cada *Memory Descriptor* ganha uma tabela *hash* onde são inseridas suas reservas, tendo como chave o endereço onde se inicia a reserva. Quando ocorre uma falta de página essa tabela *hash* é consultada com o endereço faltante. Caso não exista nenhuma reserva naquele endereço ela é criada, alocando 2 MB de memória para a reserva e a inserindo na tabela *hash*. Porém, apenas uma página de memória da reserva é preparada por falta de página, até que um número t de páginas sejam preparadas, para que então se possa mapear a região com uma *huge page*. Assim como no Quicksilver aqui também é utilizado o *bulk zeroing* para preparar o restante da *huge page*.

Caso não haja 2 MB de memória contínua livres para serem alocadas é realizada a compactação da memória. Assim como o Linux, o LRBHPS utiliza a compactação direta, porém busca migrar primeiramente as reservas não utilizadas por mais de 5 segundos, o que diminui o número de páginas escaneadas para se fazer uma compactação.

O LRBHPS mantém o comportamento do Linux de adiar ao máximo a quebra de uma *huge page*, o fazendo apenas quando a memória está sob pressão, ou quando o número de páginas mapeadas de uma *huge page* for menor que t. Quando uma *huge page* é quebrada sua reserva é mantida, e ela pode ser promovida novamente no futuro, ou migrada para alocação de outra reserva.

[1] Disponível em https://github.com/pedrodemargomes/LRBHPS/

3.1. Criação de reservas

Como descrito anteriormente, as reservas são criadas na primeira falta de página em qualquer região anônima alinhada em 2 MB, mesmo que ultrapasse o tamanho do VMA. Essa política resolve um dos problemas para qual o daemon khugepaged foi desenhado, fazendo-o de forma mais eficiente.

No Linux, caso a *huge page* ultrapasse o VMA, a única forma de ela ser mapeada como uma *huge page* posteriormente, caso o VMA seja expandido, é através do khugepaged escaneando esse endereço e a criando. Como o khugepaged normalmente é configurado para escanear poucas regiões de memória por segundo, para evitar um alto consumo de CPU, essa promoção pode demorar ou até não acontecer durante a execução da aplicação.

A Figura 1 compara a quantidade de memória alocada em *huge pages* na execução do *benchmark* Canneal nos sistemas Linux e LRBHPS $t = 64$. Esse *benchmark* tem um grande uso de memória, e expande sua *heap* durante a execução. Dessa forma, a alocação especulativa das reservas faz com que, ao ser expandida, a *heap* possa ser mapeada com *huge pages*. Isso não ocorre no Linux, que espera o khugepaged escanear o endereço para mapeá-lo como *huge page*, o que não ocorre nesse caso devido a sua velocidade. Essa diferença de *huge pages* mapeadas na *heap* causa um ganho de desempenho para o LRBHPS, que será apresentado na Seção 4.

3.2. Bulk zeroing

Como mostrado em [Yang et al. 2011], zerar páginas de memória para inicializá-las tem um custo de até 12.7%, com uma média de 2.7 a 4.5% em uma máquina virtual utilizando arquitetura IA32.

Para zerar uma *huge page*, o kernel Linux zera uma página (4 KB) por vez, o que não aproveita todo o potencial das CPUs modernas, que podem zerar 2 MB de memória muito mais rápido utilizando *bulk zeroing*, chamando instruções *assembly* para zerar 2 MB diretamente. No processador AMD Ryzen 5 3600X, utilizado nesse trabalho, a instrução clzero faz o *bulk zeroing* de maneira non-temporal, sem poluir a cache.

A Tabela 1 compara o tempo real e do sistema para alocar e escrever em 2 GB de memória. O *bulk zeroing*, utilizando a instrução clzero se mostrou mais eficiente, diminuindo o tempo execução e de sistema.

Tabela 1. Tempos de execução usando bulk zeroing e 4 KB zeroing

	Tempo total de execução	Tempo de execução de sistema
4 KB zeroing	5.2508s	0.2184s
Bulk zeroing	4.9796s	0.1896s

4. Avaliação

O LRBHPS foi avaliado em duas versões, com $t = 64$ e $t = 256$, utilizando *benchmarks* com *workloads* variados. Os testes foram executados em um computador com processador AMD Ryzen 5 3600X e 16 GB de memória RAM DDR4. O *swapping* foi desativado em todos os testes. Os resultados apresentados são a média de 3 execuções. Para avaliar o

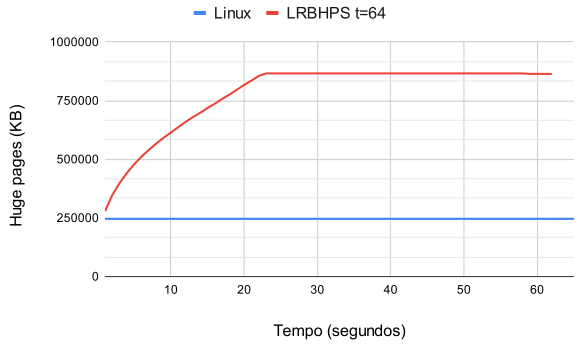

Figura 1. Huge pages na execução do benchmark canneal sem fragmentação

desempenho do sistema de THP, os *benchmarks* foram executados tanto com a memória fragmentada quanto não fragmentada. Dessa forma, é possível observar o comportamento do sistema de THP em uma situação adversa (memória fragmentada).

A Seção 4.1 descreve os *benchmarks* utilizados. A Seção 4.2 apresenta o método de fragmentação utilizado nos testes. Por fim, a Seção 4.3 apresenta os resultados obtidos.

4.1. Benchmarks

GUPS realiza 2^{32} acessos seriais aleatórios em 2^{30} números inteiros de 64 bits [Joseph sd]. O ANN faz consultas aleatórias dos vizinhos mais próximos em uma tabela *hash* pré-processada de 2 GB [Bernhardsson]. XSBench executa o *Monte Carlo neutron transport algorithm* [Tramm et al. 2014]. Graphchi-PR [Kyrola et al. 2012] executa 3 iterações do algoritmo de pagerank nos dados pré-processados do *dataset* Twitter-2010 [Kwak et al. 2010]. BlockSVM [Chang and Lin 2011] treina um modelo de classificação no *dataset* kdd2010-bridge [Stamper et al. 2010]. Freqmine, Canneal, Bodytrack, Facesim, Ferret e Swaptions são benchmarks do PARSEC benchmark suite [Bienia et al. 2008]. Freqmine identifica padrões em bancos de dados transacionais. Canneal utiliza *cache-aware simulated annealing* (SA) para minimizar o custo de roteamento de um design de chip. Bodytrack faz o *tracking* de uma pose 3D de um corpo humano através de uma sequência de imagens de múltiplas câmeras. Facesim computa uma animação realista dos movimentos de uma face a partir de uma sequência de ativações de seus músculos. Ferret faz uma busca por similaridade em um conjunto de dados de imagens. Swaptions é uma aplicação do mercado financeiro que usa o *framework Heath-Jarrow-Morton* (HJM) para precificar um portfólio de *swaption*.

Os *benchmarks* Cold e Warm são *workloads* do Redis, e foram executados utilizando 8 *threads* e 16 *pipelines*. O *benchmark* Cold insere em uma instância Redis vazia 1 milhão de objetos de 4Kb. Já o benchmark Warm faz operações *get*/*set* nessa instância já populada com um ratio de 5:5 usando objetos de 4Kb. Com os *benchmarks* Cold e Warm são obtidos o número de operações por segundo, a latência média e as latências nos percentis 90 e 99.

Para medir o *memory bloat* causado pelos diferentes sistemas foram utilizados 4 *workloads* do Redis: Del-50, Del-70, Range-XL e Range-S. O Del-50 e Del-70 inserem 1.4 milhões de objetos de 8 KB e deletam, respectivamente, 50% e 70% desses objetos de maneira aleatória. O Range-XL insere 20 mil objetos de tamanhos de 256B a 1 MB, já o

Range-S insere 5 milhões de objetos de tamanhos de 256 B a 16 KB. Os *workloads* dos *benchmarks* do Redis foram baseados em [Zhu et al. 2020], adaptados para as dimensões de memória da máquina de testes utilizada nesse trabalho.

4.2. Fragmentação

Para fragmentar a memória do sistema foi utilizado o mesmo método usado em [Zhu et al. 2020]. Essa aplicação é executada no modo usuário, e primeiramente causa uma pressão na memória para posteriormente começar a fragmentação. Para fragmentar o sistema é alocada uma memória do tamanho de uma *huge page*, que ao ser escrita resultará na tentativa de se alocar uma *huge page*. No caso dessa alocação ter sucesso, parte dessa memória é desmapeada, gerando uma desalocação parcial dessa *huge page*.

4.3. Resultados

A Tabela 5 apresenta os *speedups* obtidos pelo LRBHPS em relação ao Linux. O Linux foi configurado para ser agressivo, buscando criar *huge pages* em toda alocação de memória anônima, inclusive executando uma compactação direta se for necessário.

Com a memória não fragmentada grande parte dos *benchmarks* obtiveram uma melhora de desempenho, sendo o Canneal o que apresentou maior *speedup*, devido a alocação especulativa de reservas, como explicado na Seção 3.1.

Com a memória do sistema fragmentada nenhum *benchmark* obteve uma piora de desempenho. Os maiores *speedups* foram no GUPS e Graphchi-PR. Os dois *benchmarks* utilizam muitas *huge pages*, porém devido a fragmentação o Linux não consegue alocá-las, e recorre a alocação de páginas base o que diminui seu desempenho.

As Tabelas 2 e 3 mostram, respectivamente, o número de operações por segundo, latência média e os percentis 90 e 99 da latência em milissegundos dos *benchmarks* Cold e Warm. Com a memória fragmentada há uma grande redução nas latências médias e, especialmente, dos percentis 90 e 99 do LRBHPS, tanto no *benchmark* Cold quanto no Warm. Porém, sem a fragmentação ocorre um pequeno aumento na latência de cauda do LRBHPS no *benchmark* Cold.

Tabela 2. Redis Cold *benchmark*

COLD FRAG 0	Ops/sec	Avg. Latency	p90 Latency	p99 Latency
Linux	295,053.57	4.34	4.65	6.13
LRBHPS t=64	296,661.43	4.36	4.83	6.36
LRBHPS t=256	284,650.40	4.40	5.04	6.71

COLD FRAG 100	Ops/sec	Avg. Latency	p90 Latency	p99 Latency
Linux	261.756,22	4,89	6,60	9,04
LRBHPS t=64	296.128,89	4,37	5,01	6,66
LRBHPS t=256	291.104,06	4,40	5,31	6,90

Em [Zhu et al. 2020] é apontado o daemon khugepaged como causador de *memory bloat* no Linux, ao recriar *huge pages* parcialmente desalocadas nos *benchmarks* do Redis. Esse comportamento do khugepaged pode ser evitado alterando o valor do parâmetro max_ptes_none, que especifica o número de páginas ainda não mapeadas que podem ser alocadas para se criar uma *huge page*. Ao se utilizar o valor padrão 511, o daemon khugepaged passa a recriar todas as *huge pages* parcialmente desalocadas pelo Redis, consumindo mais memória.

Tabela 3. Redis Warm *benchmark*

WARM FRAG 0	Ops/sec	Avg. Latency	p90 Latency	p99 Latency
Linux	342,003.06	3.74	5.19	5.82
LRBHPS t=64	342,430.49	3.74	5.20	5.80
LRBHPS t=256	340,930.04	3.75	5.22	5.81

WARM FRAG 100	Ops/sec	Avg. Latency	p90 Latency	p99 Latency
Linux	323.029,75	3,96	5,32	6,10
LRBHPS t=64	338.055,89	3,78	5,11	5,74
LRBHPS t=256	339.150,21	3,77	5,11	5,73

A Tabela 4 mostra que o LRBHPS conseguiu reduzir o *memory bloat* em todos *workloads* testados. Nos *benchmarks* Del-50 e Del-70, o Linux inicia criando as *huge pages* conforme os objetos vão sendo inseridos, porém ao deletá-los as *huge pages* não são desalocadas, e sim inseridas em uma fila para serem desalocadas posteriormente, quando a memória estiver sob pressão. Isso faz com que as alocações futuras sejam mais demoradas, e muitas vezes falhem ao alocar uma *huge page*. No LRBHPS, ao se desalocar mais de t páginas de uma *huge page* ela é quebrada e transformada em uma reserva, que poderá ser facilmente quebrada para liberar memória. Os LRBHPS $t = 64$ e $t = 256$ criam praticamente o mesmo número de reservas+*huge pages* nos *benchmarks*, porém em Del-50 e Del-70 o número de *huge pages* quebradas e transformadas em reservas é maior em LRBHPS $t = 256$, o que diminui o uso de memória. Em Range-S, o número de *huge pages* criadas em LRBHPS $t = 256$ é um pouco menor, o que explica a pequena diminuição do uso de memória. Já em Range-XL são praticamente os mesmos.

Tabela 4. Uso de memória nos 4 *workloads* do Redis apresentados

Workload	Linux-4 KB	Linux	LRBHPS t=64	LRBHPS t=256
Del-50	10.75G	14.22G	13.98G	13.04G
Del-70	7.54G	14.19G	13.85G	10.21G
Range-XL	11.19G	12.12G	12.01G	12.01G
Range-S	9.42G	9.70G	9.56G	9.52G

A Tabela 6 mostra alguns dados chave para entender o comportamento da compactação do Linux e do LRBHPS durante a execução do *benchmark* Graphchi-PR com a memória do sistema fragmentada. O número de alocações de *huge pages* e mapeamentos do LRBHPS consegue ser maior que o do Linux, escaneando menos páginas (somando o *migrate* e *free scanner*) e obtendo mais sucessos na compactação direta.

5. Conclusão

O uso de THPs é extremamente importante para melhorar o desempenho de aplicações que demandam uma grande quantidade de memória. Nesse artigo foram mostradas as estratégias utilizadas para o gerenciamento das THPs nos sistemas Linux, FreeBSD, Ingens, HawkEye e Quicksilver. A partir delas foi implementado o LRBHPS, um sistema de gerenciamento de THPs baseado em reservas. Utilizando *benchmarks* com workloads sintéticos e reais, e comparando seus resultados com o Linux, foi constatado uma melhora de desempenho em quase todos os testes, inclusive nos testes de consumo de memória, mostrando que o LRBHPS consegue produzir uma melhora de desempenho causando menos *memory bloat*.

Tabela 5. *Speedups* obtidos pelo Linux-4 KB, LRBHPS $t = 64$ e LRBHPS $t = 256$ em um sistema com e sem fragmentação

SPEEDUP FRAG 0	Linux-4 KB	LRBHPS t=64	LRBHPS t=256
GUPS	0.28	0,99	0,98
freqmine	1.00	1,00	1,00
canneal	0.95	1,08	1,07
ann	1.02	1,00	1,00
Graphchi-PR	0.86	0,99	0,98
xsbench	0.88	1,00	1,00
blocksvm	0.82	1,00	0,99
bodytrack	1.00	1,00	1,00
facesim	1.00	1,00	1,00
ferret	1.00	1,00	1,00
swaptions	1.00	1,00	1,00

SPEEDUP FRAG 100	Linux-4 KB	LRBHPS t=64	LRBHPS t=256
GUPS	0.85	2,59	2,52
freqmine	0.99	1,02	1,02
canneal	0.97	1,08	1,08
ann	1.01	1,02	1,01
Graphchi-PR	1.07	1,31	1,29
xsbench	0.94	1,03	1,03
blocksvm	0.86	1,03	1,01
bodytrack	0.93	1,00	1,00
facesim	0.99	1,00	1,00
ferret	0.99	1,00	1,00
swaptions	1.00	1,00	1,00

Tabela 6. Dados de criação de *huge pages* e compactação dos sistemas Linux e LRBHPS na execução do Graphchi-PR com a memória fragmentada

Graphchi-PR FRAG 100	huge page alloc	mapped huge pages	huge page allocation failed	migrate scanned	free scanned	compaction fail	compaction success	sum migrate and free scanned
Linux	30.731,67	30.731,67	9.322,67	2.762.193,33	566.667,67	37.320,67	1.886,67	3.328.861,00
LRBHPS t=256	39.794,67	39.719,67	0,00	139.182,33	874.733,33	1,33	2.975,67	1.013.915,67

Referências

Bernhardsson, E. Annoy. `https://github.com/spotify/annoy`. Acesso em: 31/03/2025.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA. Association for Computing Machinery.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Corbet, J. (2011). Transparent huge pages in 2.6.38. *LWN.net*. `https://lwn.net/Articles/423584/` Acesso em: 31/03/2025.

Hadoop (s.d.). Hadoop performance tuning guide. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf`. Acesso em: 31/03/2025.

Hornyack, P., Ceze, L., Gribble, S., Ports, D., and Levy, H. (2013). A study of virtual memory usage and implications for large memory. In *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*.

Joseph, I. E. (s.d.). Gups (giga-updates per second) benchmark. http://www.dgate.org/~brg/files/dis/gups/. Acesso em: 31/03/2025.

Kwak, H., Lee, C., Park, H., and Moon, S. (2010). What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA. Association for Computing Machinery.

Kwon, Y., Yu, H., Peter, S., Rossbach, C. J., and Witchel, E. (2016). Coordinated and efficient huge page management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA. USENIX Association.

Kyrola, A., Blelloch, G., and Guestrin, C. (2012). GraphChi: Large-Scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA. USENIX Association.

mongoDB (s.d.). Disable Transparent Huge Pages (THP). https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/. Acesso em: 31/03/2025.

Navarro, J., Iyer, S., Druschel, P., and Cox, A. (2003). Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104.

Panwar, A., Bansal, S., and Gopinath, K. (2019). Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 347–360, New York, NY, USA. Association for Computing Machinery.

postgreSQL (s.d.). Resource consumption. https://www.postgresql.org/docs/current/runtime-config-resource.html. Acesso em: 31/03/2025.

Stamper, J., Niculescu-Mizil, A., Ritter, S., Gordon, G., and Koedinger, K. (2010). Challenge data set from KDD Cup 2010 Educational Data Mining Challenge. http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp. Acesso em: 31/03/2025.

Talluri, M. and Hill, M. D. (1994). Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 171–182, New York, NY, USA. Association for Computing Machinery.

Tramm, J. R., Siegel, A. R., Islam, T., and Schulz, M. (2014). XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

Yang, X., Blackburn, S. M., Frampton, D., Sartor, J. B., and McKinley, K. S. (2011). Why nothing matters: the impact of zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 307–324, New York, NY, USA. Association for Computing Machinery.

Yznaga, A. (2018). Implement THP reservations for anonymous memory. https://marc.info/?l=linux-mm&m=154174619201183&w=4. Linux kernel mailing list (LKML) Acesso em: 31/03/2025.

Zhu, A. (2022). THP Shrinker. https://lore.kernel.org/lkml/cover.1661461643.git.alexlzhu@fb.com. Linux kernel mailing list (LKML) Acesso em: 31/03/2025.

Zhu, W., Cox, A. L., and Rixner, S. (2020). A comprehensive analysis of superpage management mechanisms and policies. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 829–842. USENIX Association.