Clara Daia Hilgenberg Darú

Comparação de desempenho de caches de segmentação e de paginação

Curitiba - PR

2018

Clara Daia Hilgenberg Darú

Comparação de desempenho de caches de segmentação e de paginação

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação.

Universidade Federal do Paraná

Setor de Ciências Exatas

Departamento de Informática

Orientador: Prof. Dr. Roberto André Hexsel

Curitiba - PR

2018

Agradecimentos

A Lauri Laux Jr., autor de [Lau17], que é a base deste trabalho, pela disponibilidade, pela colaboração e pelas sugestões.

A Roberto Hexsel, meu orientador, pela disponibilidade, pela paciência e por tudo que aprendi.

Resumo

Desde a década de 1970, a memória virtual é usada com páginas de 4Kbytes. A quantidade crescente de memória RAM disponível e do espaço de endereçamento cedido aos processos resulta em tabelas de páginas muito grandes e uma grande quantidade de faltas na TLB, o que tem impacto no desempenho dos programas. A segmentação pode ser um método melhor de virtualização da memória do que a paginação para os sistemas atuais. Os experimentos realizados mostram que a taxa de faltas em uma cache de segmentos é muitas vezes menor do que em uma TLB para as aplicações estudadas.

Palavras-chaves: memória virtual. segmentação. paginação.

1 Introdução

A quantidade restrita de memória primária (RAM) nos sistemas pode fazer necessário armazenar, temporariamente, parte dos dados de um processo em memória secundária (disco magnético). A tarefa de movimentar explicitamente estes dados entre os níveis de memória é complexa e tediosa. Para facilitar o desenvolvimento de aplicações, foi introduzida uma camada de abstração do espaço de endereçamento dos processos chamada de *memória virtual*.

Esta camada de abstração oferece ao programador um espaço de memória contíguo e linear, cujo tamanho é limitado apenas pelo tamanho dos apontadores, em linguagens como C. Para tanto, a memória é dividida em blocos associados a restrições de acesso de acordo com seu conteúdo, e que são movidos inteiros entre a memória primária e a memória secundária. Uma das funções do subsistema de memória virtual é realizar o transporte destes blocos entre a memória primária e a secundária, de forma transparente ao programador.

Os dois principais modelos de virtualização da memória se diferenciam pela política de divisão: a *paginação* utiliza blocos de tamanho fixo, e a *segmentação* utiliza blocos de tamanho variável alinhados com a divisão lógica do espaço de endereçamento.

A paginação é o modelo mais utilizado, e o tamanho mais comum de páginas é 4KBytes, definido para otimizar o *tradeoff* entre tempo de transporte entre memória principal e memória secundária e fragmentação interna [Den70]. Ao longo de quase meio século, este tamanho se manteve a despeito do crescimento no espaço de endereçamento virtual cedido aos processos. Atualmente temos tabelas de páginas gigantescas, e o espaço de endereçamento deve continuar aumentando.

Estruturas auxiliares da paginação, como tabelas hierárquicas e caches, devem buscar um equilíbrio entre o espaço ocupado na memória apenas para suportar a virtualização e o tempo extra de acesso aos dados. O processo de tradução de endereços é responsável por uma parcela não-neglicenciável do tempo de execução de um processo, podendo chegar a mais de 50% em aplicações com localidade particularmente ruim – um *benchmark* de análise de grafos, por exemplo [HCG+15].

Laux Júnior sugere, em [Lau17], que a segmentação é uma técnica de virtualização da memória mais apropriada para os sistemas atuais e os que estão por vir, com grande potencial de reduzir o impacto de desempenho causados pelas estruturas de dados empregadas para implementar paginação.

O capítulo 2 define brevemente a arquitetura geral de um computador. O capítulo 3

explica o funcionamento da paginação e da segmentação e faz comparações conceituais entre os principais elementos usados na sua implementação, principalmente no que diz respeito à quantidade de segmentos e a quantidade de páginas que um processo pode usar.

Os experimentos realizados em [Lau17] mostram uma quantidade muito menor de segmentos do que de páginas em determinadas aplicações, e consequentemente taxas de faltas muito menores em caches da tabela de segmentos do que em caches da tabela de páginas. Este trabalho tem como objetivo verificar que esta característica se mantém em outras aplicações, realizando experimentos semelhantes.

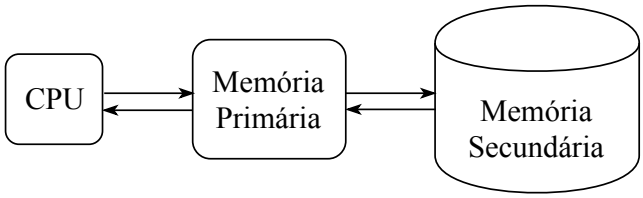
O capítulo 4 descreve a metodologia dos experimentos realizados e os resultados obtidos no conjunto de aplicações avaliadas. O capítulo 5 apresenta as conclusões e indica os próximos passos para uma necessária revisão da forma de implementar memória virtual.

2 Modelo de Computador

Os computadores considerados neste trabalho possuem arquitetura baseada no modelo de Von Neumann: são compostos por uma unidade de processamento (o processador ou a *Central Processing Unit*, CPU) executando instruções de um programa previamente carregado em uma unidade de *memória primária*. Esta memória é usualmente implementada com memória do tipo *Random Access Memory* (RAM), com tempo de acesso da ordem de 100ns. A unidade mínima dos dados armazenados na RAM é chamada de *palavra*, e o índice de uma palavra é chamado de *endereço físico*.

As máquinas em questão geralmente possuem *memória secundária* persistente implementada com um ou mais discos magnéticos, com tempo de acesso na faixa de 10ms, e/ou discos de estado sólido (*Solid-State Drives* ou SSDs) com tempo de acesso na faixa de 0.1ms.

Figura 1 – Computador



Um programa sendo executado no processador é chamado de *processo*. Os dados usáveis pelo processo – aqui não apenas aqueles que o programa consome ou cria, mas também código e pilha, por exemplo – podem não caber na memória RAM instalada, e neste caso usa-se a memória secundária para armazená-los.

A CPU não acessa a memória secundária diretamente, e dados que estejam nela devem ser trazidos para a memória primária antes do uso no processamento. Pode ser necessário mover outros dados para a memória secundária para livrar espaço na memória primária. O Sistema Operacional reserva uma parte da memória secundária para este fim, que é chamada de área de *swap* (troca), e as operações de troca de dados entre os níveis de memória são chamadas de operações de *swap*.

3 Memória virtual

Até a década de 1960, as aplicações realizavam explicitamente a movimentação de seus dados entre as memórias primária e secundária. Isto exigia dos programadores um amplo conhecimento da arquitetura do sistema para o qual escreviam software, ao mesmo tempo em que "amarrava" o código à mesma e tornava inviável manter vários processos ativos alternando o uso da CPU.

Para mitigar estes problemas, em 1962 foi concebida a virtualização da memória [KELS62]. A *memória virtual* é uma abstração que expõe ao programador um espaço de endereçamento unidimensional contínuo e que é independente do espaço de endereçamento físico. Este espaço é chamado *virtual*, e os programas são escritos considerando apenas o espaço virtual de endereços como a "memória" em que o programa executa.

Os endereços virtuais são mapeados para endereços físicos, e esta associação é invisível aos programadores. Isto só é possível com a adição de *hardware* ao processador e de módulos ao sistema operacional para executar três tarefas: (i) a tradução dos endereços virtuais acessíveis pelo programa para endereços físicos, (ii) a movimentação dos dados entre as memórias primária e secundária quando necessário, e (iii) a aplicação de permissões de execução, leitura e escrita na memória.

Estes mecanismos essenciais à virtualização da memória resultam na ocupação de parte da memória por dados não diretamente acessados pelo processo – um prejuízo chamado de "fragmentação" em [Den70], e aumentam o tempo de acesso à memória. No entanto, as vantagens da virtualização – permitir aos programadores focar no desenvolvimento de soluções e não na administração da memória, e possibilitar o escalonamento de processos ativos – superam os malefícios.

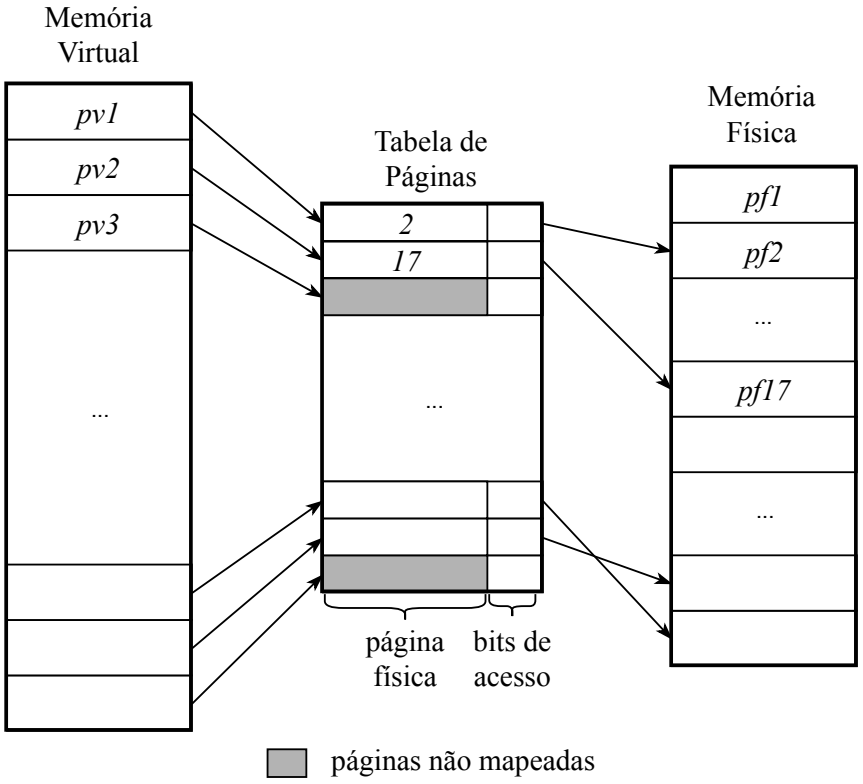
3.1 Paginação

O método mais utilizado de mapeamento entre a memória virtual e a memória física é a paginação, que divide o espaço de endereçamento em blocos de tamanho fixo, chamados de *páginas*.

Na paginação, um endereço virtual é dividido em duas partes: n bits identificam a página virtual e m bits indicam o deslocamento do dado dentro da página. O mapeamento entre páginas virtuais e páginas físicas é mantido em uma estrutura chamada *tabela de páginas* (TP), que contém, para cada página do processo, a identificação da página física em que a página virtual, ou lógica, está alocada e bits de permissão de acesso, como mostrado na figura 2.

As páginas não mapeadas não estão presentes na memória física. Podem ter sido temporariamente movidas para a memória secundária ou não serem válidas para o processo em questão.

Figura 2 – Mapeamento de páginas



A sequência e a contiguidade de páginas virtuais não é necessariamente mantida entre as páginas físicas para as quais estão mapeadas, porque a TP se comporta como um mapeamento totalmente associativo.

A tradução de um endereço virtual inicia consultando-se a tabela de páginas para verificar se o processo tem permissão de acesso ao endereço desejado. Se o acesso é permitido, verifica-se na tabela se a página virtual está alocada em uma página física e obtém-se o endereço físico a partir da identificação da página física e do deslocamento contido no endereço virtual. Então, completa-se o acesso.

Neste processo, podem ocorrer dois tipos de falhas: *protection faults*, relacionadas a permissões, e *page faults*, relacionadas à presença da página requisitada na memória primária.

Se houver, por exemplo, uma tentativa de escrita em uma página para a qual o processo só tem permissão de leitura – uma página de código – ocorre uma *protection fault* e o processo é terminado pelo sistema operacional. O mesmo acontece se uma instrução tentar qualquer tipo de acesso em uma página inválida, que não foi alocada para o processo.

Se a página virtual for válida mas não estiver mapeada para uma página física, ocorre uma *page fault* e o processo é interrompido até que a página seja trazida da memória secundária para a memória primária.

Quando a página é trazida da memória secundária, se não houver uma página física vazia para instalar a página virtual requisitada, pode-se sacrificar uma página presente na memória. Como o tamanho das páginas é sempre o mesmo, a substituição é relativamente simples, bastando copiar a página vítima para a memória secundária e em seguida a nova página para o espaço recém-liberado.

A tabela de páginas precisa estar na memória física. Como todo acesso à memória do processo exige tradução de endereço, isto significaria que para cada referência seriam necessários dois acessos à RAM: o primeiro para se obter o endereço físico da palavra referenciada, e o segundo para completar a referência. Uma tradução assim dobraria o tempo de uma referência à memória, e para diminuir o impacto no desempenho usa-se uma cache dedicada para a TP, conhecida como *Translation Look-aside Buffer*.

Além disso, o tamanho típico de uma página é 4KBytes. Isto significa que para endereços de 48 bits, uma tabela de páginas com o mapeamento completo do espaço de endereçamento teria $2^{48}/2^{12} = 2^{36}$ elementos. Não é factível armazenar uma estrutura tão grande na memória primária. Por isso, modificações ao modelo original foram introduzidas, tais como tabelas de páginas hierárquicas e páginas que são de duas a seis ordens de grandeza maiores do que 4Kbytes e são chamadas de *huge pages* (HPs).

3.1.1 *Translation Look-aside Buffer*

Como mencionado, se a tabela de páginas estiver inteira alocada na memória primária, cada tradução de endereço exige um acesso adicional à TP, que é mantida na própria memória primária para obter o endereço físico, e este segundo acesso causa uma redução substancial no desempenho dos programas.

Felizmente, na maior parte dos programas, um subconjunto relativamente pequeno dos dados é acessado em um período relativamente curto de tempo, e esta propriedade é chamada de *localidade de referência*. Pode-se tirar proveito da localidade para evitar o segundo acesso à memória primária, se os elementos mais recentemente usados da tabela de páginas forem mantidos em uma cache especializada chamada de *Translation Look-aside Buffer* (TLB).

A tradução de endereços inicia com uma consulta à TLB. Se o mapeamento da página virtual estiver presente na TLB, então o número da página física, obtido da TLB, é concatenado com o deslocamento, e o endereço físico resultante é enviado para a memória RAM. Durante o acesso à TLB são verificadas as permissões de acesso à página referenciada. Se o mapeamento não estiver na TLB, então ocorre uma *falta na TLB*, e o

mapeamento deve ser buscado da tabela de páginas na memória primária.

3.1.2 Tabela de páginas hierárquica

Em geral, os programas utilizam apenas uma parcela pequena de todo o espaço de endereçamento virtual disponível, que é de 2^{48} bytes nas máquinas disponíveis atualmente [Int18]. Para estes programas, a maior parte da tabela é de páginas inválidas e não deve ser acessada. Se a TP completa for armazenada em memória, os elementos inválidos são um desperdício de espaço.

Para reduzir este desperdício, pode-se usar uma tabela de páginas *hierárquica* ou *multinível*. Estas TPs são estruturadas com uma ou mais tabelas intermediárias indexadas por pedaços do número da página virtual. Cada tabela intermediária aponta para outra tabela contendo as informações da faixa de páginas virtuais indexada. A tabela "folha" contém o número da página física. Esta estrutura permite registrar nas tabelas intermediárias se toda uma faixa de páginas virtuais não está alocada, o que reduz a quantidade de elementos total nas tabelas "folha". Mais detalhes em [Hex18].

A hierarquia reduz o espaço da memória desperdiçado pela tabela de páginas, mas torna faltas na TLB significativamente mais custosas, uma vez que a obtenção do número de uma página física exige caminhar por todas as tabelas intermediárias, podendo este processo envolver até cinco acessos à memória primária.

3.1.3 *Huge pages*

O tamanho da página é inversamente proporcional ao tamanho da tabela de páginas. Portanto, outra forma de reduzir o espaço ocupado pela tabela de páginas, e que também potencialmente diminui a quantidade de faltas na TLB, é usar páginas maiores.

Arquiteturas mais recentes de processadores permitem o uso de páginas maiores do que o padrão de 4KBytes, com tamanhos que variam de 1MB até 1GB. Entre 2006 e 2007 foi adicionado ao *kernel* do Linux suporte a *huge pages* [LF15]. Estas páginas são alocadas na inicialização do sistema operacional, e não estão sujeitas a *swapping*. Aplicações podem fazer uso delas por meio de *system calls* de alocação de memória específicas ou configurações predefinidas de permissões.

Em 2011 foi introduzido o modo de *Transparent Huge Pages* (THP), que permite ao *kernel* utilizar páginas grandes em qualquer aplicação quando o *kernel* julgar apropriado [Cor11].

3.1.3.1 Fragmentação interna

Nos primórdios da paginação, em máquinas com pouca memória física disponível, levava-se em consideração o desperdício de memória causado pela última página ocupada

por um segmento lógico. Se o espaço de endereçamento "útil" S não for um múltiplo do tamanho de página p, a última página terá $p - (S \bmod p)$ bytes desperdiçados, e este desperdício é chamado de *fragmentação interna*.

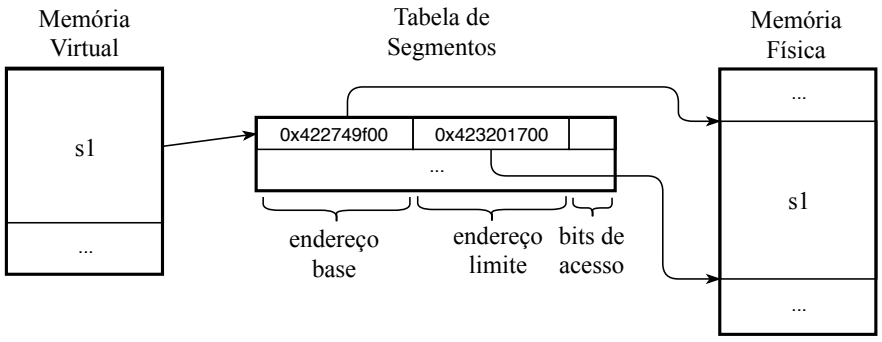
Com a quantidade de memória disponível atualmente e usando páginas de 4KBytes, a fragmentação interna é praticamente irrelevante. Páginas gigantes, no entanto, possibilitam desperdiçar uma quantidade considerável de memória, se por exemplo dividirmos o espaço de endereçamento de uma aplicação que usa algo como 10^7 bytes de memória em páginas de 10^9 bytes (1GByte). Portanto, orienta-se o uso de páginas de tamanho na ordem de megabytes para aplicações que usam gigabytes de memória, e páginas de tamanho na ordem de gigabytes em aplicações que chegam a terabytes de memória [Red18a].

3.2 Segmentação

A *segmentação* é um modelo de virtualização da memória no qual a memória virtual é dividida em blocos de tamanho variável, que são os segmentos lógicos do programa – código, dados, *heap*, pilha, bibliotecas e arquivos abertos. Por conta do tamanho variável, não existem segmentos físicos predefinidos, e o espaço na memória física é alocado de acordo com a demanda dinâmica imposta pelos processos em execução [DD68, BCD72].

Com segmentação, uma quantidade fixa dos bits do endereço virtual é usada para identificar o segmento virtual, e o restante define o deslocamento dentro do segmento. Um segmento físico instalado em memória pode iniciar em qualquer endereço, de acordo com as condições de utilização da memória no momento da alocação. A estrutura que mapeia segmentos virtuais em segmentos físicos é a *tabela de segmentos* (TS). A TS contém, para cada segmento virtual instalado na memória primária, o endereço físico inicial – a *base* do segmento – e o endereço final do segmento – o seu limite – além dos bits de permissão de acesso, como mostrado na figura 3.

Figura 3 – Mapeamento de segmentos



A tradução de um endereço virtual inicia, assim como na paginação, verificando-se na tabela de segmentos se o tipo de acesso é permitido para aquele segmento. Autorizado

o acesso, o deslocamento é somado ao endereço base do segmento para obter o endereço físico.

Se o acesso for proibido, ocorre uma *protection fault* e o processo é terminado.

Se o segmento em questão não estiver alocado na memória física, ocorre uma *segmentation fault*, e o programa é interrompido até que o segmento tenha sido buscado da memória secundária.

Uma vez que o número de bits do deslocamento pode indexar um espaço maior do que o tamanho do segmento, há uma verficação adicional de proteção depois da soma do deslocamento: um deslocamento maior do que o tamanho limite do segmento também gera uma *protection fault* e causa o término do processo.

Como a tabela de segmentos deve residir em memória primária, é necessária uma referência à TS para então completar a referência à memória. Pode-se evitar o acesso adicional à memória usando uma cache especializada, chamada de *Segment Buffer* (SB). No início da tradução, consulta-se a SB e se o segmento virtual não estiver presente, deve ser buscado na memória principal.

3.2.1 Fragmentação Externa

Com paginação simples, a divisão da memória é uniforme - todos os blocos têm o mesmo tamanho. Quando uma página é removida da memória, o espaço livre é exatamente o tamanho necessário para instalar uma nova página. Portanto, um mecanismo de alocação/carregamento pode usar a primeira página física livre que encontrar e, caso não haja páginas livres, a única decisão relevante para a substituição é escolher uma página para ser sacrificada.

Os diferentes tamanhos possíveis de segmentos, por outro lado, dificultam tanto a alocação de novos segmentos quando a substituição na memória primária.

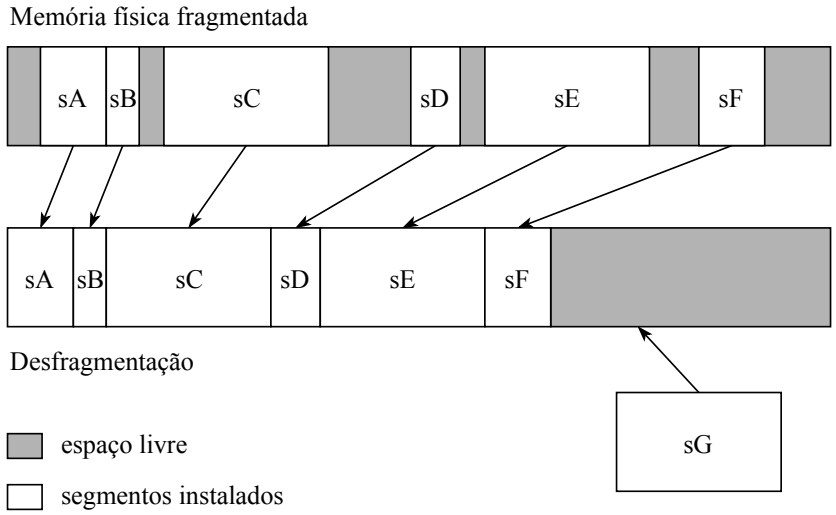
O mecanismo de alocação é mais complexo na segmentação, pois deve buscar um trecho de memória livre que tenha um tamanho maior ou igual ao do segmento requisitado. A substituição/realocação pode ficar particularmente complexa se nenhum segmento instalado ou trecho livre tiverem o tamanho necessário para o novo segmento.

À medida em que processos são criados e terminados, é possível e provável que segmentos não contíguos sejam removidos da memória, deixando-a com "buracos". Se forem muitos os "buracos" pequenos, ocorre um desperdício de espaço chamado de *fragmentação externa* [HCG+15, SGG12].

A figura 4 mostra um exemplo de alocação de memória nesta condição. Pode-se observar que para instalar o novo segmento sG, nenhum dos trechos livres é grande suficiente, mas se reposicionarmos todos os segmentos alocados de modo a concentrar o

espaço livre, há mais espaço do que o necessário.

Figura 4 – Fragmentação externa



A fragmentação externa é um grande obstáculo a implantação de segmentação e está fora do escopo deste trabalho.

3.2.2 Segmentação *versus* Paginação

À medida em que a quantidade de memória RAM disponível aumenta e, ao mesmo tempo, o espaço de endereçamento cedido aos processos, características vantajosas da paginação se tornam menos relevantes e os prejuízos ficam mais evidentes.

[HCG+15] sugere que a segmentação deve ser reconsiderada como técnica de virtualização, uma possibilidade analisada extensamente em [Lau17]. Nas seções que seguem são apresentadas comparações entre a segmentação e os dispositivos auxiliares da paginação, que suportam esta proposta.

3.2.2.1 Tabela de Segmentos *versus* Tabela de Páginas Hierárquica

A quantidade de segmentos lógicos dos programas costuma ser pequena quando comparada à quantidade de páginas: além das regiões de código, pilha e dados, arquivos abertos e bibliotecas têm seus próprios segmentos. Podemos estimar que a tabela de segmentos não terá mais de 2000 itens, considerando um processo com 100 bibliotecas com 2 segmentos cada (código e dados), um para código, um para dados, um ou mais para *heap*, um para pilha e mais uma porção de arquivos abertos.

Se esta estimativa for válida, significa que não é necessário estruturar a TS com vários níveis, como a tabela de páginas hierárquica, para economizar memória.

Muitos programas podem ter a TS completa armazenada em uma SB de poucos elementos. Mesmo que haja muitas faltas, se a TS não for hierárquica o custo de uma falta na SB é apenas um acesso à memória primária, enquanto uma falta na TLB pode resultar em até 5 acessos a uma tabela de páginas hierárquica.

3.2.2.2 Tabela de Segmentos *versus Huge Pages*

A própria existência de *huge pages* denuncia que há uma defasagem entre a implementação de paginação como é e as características dos sistemas atuais. A motivação para o uso de páginas maiores é melhorar o desempenho dos programas reduzindo o tamanho da tabela de páginas e as faltas na TLB, semelhante à motivação para o uso da segmentação. Algumas arquiteturas suportam tamanhos variados de páginas gigantes, o que permite dividir melhor a memória de acordo com as características das aplicações – a segmentação nada mais é que a flexibilização máxima desta divisão.

Como mencionado na seção, o uso de páginas gigantes é recomendado para aplicações cujo uso total de memória é cerca de 2^{10} vezes maior que o tamanho da página. Na melhor das hipóteses, o espaço de endereçamento pode ser todo dividido em páginas gigantes, o que resultaria em tabelas de páginas com $|\text{TP}| \approx 2^{10}$ elementos, o que não é significativamente menor do que a estimativa de 2 mil elementos para uma tabela de segmentos. Na prática, porém, uma vez que a paginação também tem a função de aplicar proteção às regiões de memória, nunca haverá segmentos lógicos diferentes mapeados para uma mesma página, de modo que a tabela de segmentos é sempre menor ou, na melhor das hipóteses do mesmo tamanho que a tabela de páginas [Wik18a].

Além disso, *huge pages* podem ocasionar desperdício significativo de memória se o tamanho de página escolhido for inadequado, por conta da fragmentação interna. Este tipo de fragmentação ocorre com qualquer tamanho de página, mas só se torna um problema com páginas grandes com relação ao tamanho total da memória. Não existe fragmentação interna em segmentos, uma vez que o tamanho do segmento tem granularidade de palavra(s).

3.2.2.3 *Swapping*

Os tamanhos variados dos segmentos interferem nas operações de *swapping*: a fragmentação que ocorre na memória pode ocorrer de forma semelhante na área de *swap* no disco, e o tempo de cópia de um segmento pode ser várias vezes maior do que o tempo de cópia de uma página. A paginação é mais conveniente para *swap* do que a segmentação.

Entretanto, operações de *swap* são menos frequentemente necessárias à medida em que aumenta a quantidade de RAM disponível. Hornyack [HCG+15] mostra, para aplicações de usuários comuns, que a maior parte da memória virtual reside a maior parte

do tempo na memória primária. Isto significa que as vantagens da segmentação podem compensar potenciais perdas de desempenho em *swapping*.

4 Experimentos

Uma métrica possível do impacto de desempenho causado pela virtualização da memória é a taxa de faltas na TLB: a razão entre a quantidade de faltas e a quantidade de acertos na TLB. [HCG+15] mostra que uma porcentagem considerável do tempo de execução de um processo é gasto na resolução deste tipo de falta que, como mencionado em na seção 3.1.2, pode exigir mais de um acesso à memória.

A quantidade de segmentos estimada para um processo e a quantidade de páginas calculada para endereços de 48 bits permite supor que a taxa de faltas na SB em um sistema que emprega segmentação será muito menor do que a taxa de faltas na TLB em um sistema com paginação, para o mesmo processo.

As aplicações testadas em [Lau17] demonstram, de fato, taxas de faltas em Segment Buffers da ordem de 10^3 a 10^4 vezes menores do que as taxa de faltas em TLBs.

Os experimentos descritos no que segue caracterizam o comportamento das aplicações Redis, SQLite, PyCharm, VLC e Blender no que diz respeito à taxa de faltas em SBs e TLBs. Estas aplicações são descritas adiante.

4.1 Metodologia

Para comparar a taxa de faltas em uma TLB e em uma SB, na falta de um processador com suporte a segmentação [Wik18c], podemos recorrer a simulações do comportamento de ambos sob uma determinada sequência de acessos à memória. As sequências de acessos utilizadas nas simulações são de processos reais de aplicações com uso significativo de memória relativo à quantidade total de memória disponível em computadores pessoais comuns.

4.1.1 Coleta de traços

Para obter as sequências de acessos à memória, utilizamos *Valgrind*, que é uma plataforma de *debugging* e *profiling* de programas, mais o *Lackey*, uma ferramenta que reporta os endereços de memória acessados por um processo.

A saída do *Valgrind* é direcionada para uma aplicação Java desenvolvida por Lauri, chamada de *TraceCollector*, por meio de um *socket*. O *TraceCollector* registra os traços de acesso à memória em arquivos de 5mi instruções e os compacta para reduzir o espaço total ocupado – o conjunto de arquivos de traços compactados ocupa entre 15 e 25GBytes.

4.1.2 Tabela de segmentos

Durante a coleta de traços é usada uma ferramenta escrita em *Python*, o *PmapObserver*. Esta ferramenta monitora, a cada segundo, a saída do programa *pmap*, que mostra o mapa de memória do processo como registrado pelo *kernel* do Linux. Toda vez que há uma alteração no mapa, o *PmapObserver* o registra em um novo arquivo.

Os mapas de memória contém identificação, endereço e bits de permissão para os segmentos definidos para o programa no momento da amostragem. Estes mapas são úteis para observações qualitativas e quantitativas dos segmentos alocados, e permitem gerar a Tabela de Segmentos usada na simulação.

4.1.3 Simulação

Os traços de acesso à memória gerados pelo *TraceCollector* são usados por outra aplicação Java, o *TraceSimulator*. Esta aplicação lê o mapa de memória gerado pelo *pmap* e cria a partir dele uma Tabela de Segmentos. Com a tabela, é possível simular o comportamento de um *Segment Buffer*.

As simulações são feitas com TLBs e SBs de 32, 64 e 128 elementos. As caches são totalmente associativas – qualquer linha da tabela pode ser instalada em qualquer linha da cache – e com política de substituição *Least Recently Used* (LRU) perfeita – sacrifica-se sempre a entrada menos recentemente usada para instalar uma nova.

A simulação do *Translation Look-aside Buffer* é feita dividindo o endereço em 52 bits para o número da página virtual e 12 bits para o deslocamento, considerando-se o tamanho de página típico de 4Kbytes.

Ao final do processamento de cada registro de 5mi instruções, o estado das caches (um *snapshot*) é registrado em um arquivo, e a quantidade total de acessos e faltas em cada cache é computada.

Espaço em branco proposital.

4.2 Resultados

Esta seção mostra os resultados obtidos com a coleta de traços e a simulação de acessos com segmentação e paginação para as aplicações avaliadas, principalmente no que diz respeito à taxa de faltas nas caches das respectivas tabelas de mapeamento.

4.2.1 Redis

Redis é um sistema gerenciador de banco de dados escrito em C, com estrutura do tipo chave-valor e cujos elementos são armazenados em memória primária. É usado principalmente como cache de *RESTful APIs* [1], para evitar acessos mais lentos a bancos de dados relacionais.

O projeto inclui uma ferramenta de *benchmarking* própria - um cliente configurável que executa uma série de operações e informa o tempo médio de execução para cada tipo de operação. Os traços foram coletados na execução deste cliente, configurado para executar apenas operações de escrita e leitura (*sets* e *gets*), com um espaço de chaves de 10^5 elementos utilizados aleatoriamente.

A primeira característica notável do mapa de memória ao final da coleta de traços é que contém apenas 64 segmentos. Isto significa que a tabela inteira de segmentos caberia em uma cache de 64 elementos.

Mesmo uma SB de menos de 64 elementos é o suficiente para reduzir as faltas apenas às compulsórias: a tabela 1 mostra uma quantidade de faltas constante em todos os tamanhos de SB simulados, tão pequena que é mais claramente representada pelo valor absoluto do que relativo à quantidade total de acessos à memória. Para os tamanhos de TLB simulados, a taxa de faltas fica entre 0.03% e 0.33%.

Tabela 1 – Quantidade de acessos e faltas por tamanho e tipo de cache - Redis

Tamanho da cache	Total de acessos	Faltas na TLB	Faltas na SB
32	5.02×10^8	1.67×10^7	36
64	5.02×10^8	2.83×10^6	36
128	5.02×10^8	1.55×10^6	36

Apesar de ser um resultado promissor se nos restringirmos à taxa de faltas, vale observar que a documentação do Redis instrui o usuário a remover *Transparent Huge Pages* das configurações do sistema para uso em produção. Isto porque o mecanismo de *backup* em disco do banco de dados é um processo separado que compartilha as páginas de dados com o processo principal.

[1] Modelo de interface comum de *web services*, utiliza requisições HTTP para realizar transações REST (REpresentational State Transfer). Ver [Wik18b].

Se ocorrem acessos ao banco durante o *backup*, cada página compartilhada acessada sofrerá *copy-on-write*, o que aumenta a latência de operações e o uso de memória pelo processo de *backup*. Páginas gigantes podem resultar em *copy-on-write* de "quase toda a memória do processo" [Red18b]. É provável que algo semelhante (e possivelmente pior, uma vez que segmentos podem ser muito maiores do que HPs) ocorra com segmentação.

No caso de uso do Redis, a adição de grandes quantidades de RAM não volátil (NVRAM) aos computadores prevista em [Lau17] pode tornar os *backups* desnecessários. Para outros programas com *forks* e regiões de memória compartilhadas, em um computador sem suporte à paginação, seria necessário projetar métodos alternativos ao *copy-on-write* para melhorar o desempenho de segmentos compartilhados entre processos. Na pior das hipóteses, aplicações em que isso fosse um grande problema poderiam dividir os dados em segmentos menores, emulando paginação.

4.2.2 SQLite

SQLite é uma biblioteca escrita em C que implementa um sistema gerenciador de banco de dados relacional, e pode ser configurada para armazenar os dados em memória primária, de maneira similar ao Redis. A versão utilizada foi a 3.42.0-1.

Os traços foram coletados durante a execução de um programa para *benchmarking* que realiza 10^7 inserções de dados e subsequentes consultas ordenadas e não ordenadas.

O mapa de memória mostra uma quantidade pequena de segmentos (67 no total), o que prenuncia as taxas baixas de faltas em SBs obtidas na simulação, que são mostradas na tabela 2.

Tabela 2 – Quantidade de acessos e faltas por tamanho e tipo de cache - SQLite

Tamanho da cache	Total de acessos	Faltas na TLB	Faltas na SB
32	1.46×10^{10}	1.83×10^{8}	64
64	1.46×10^{10}	7.48×10^{7}	46
128	1.46×10^{10}	9.58×10^{6}	46

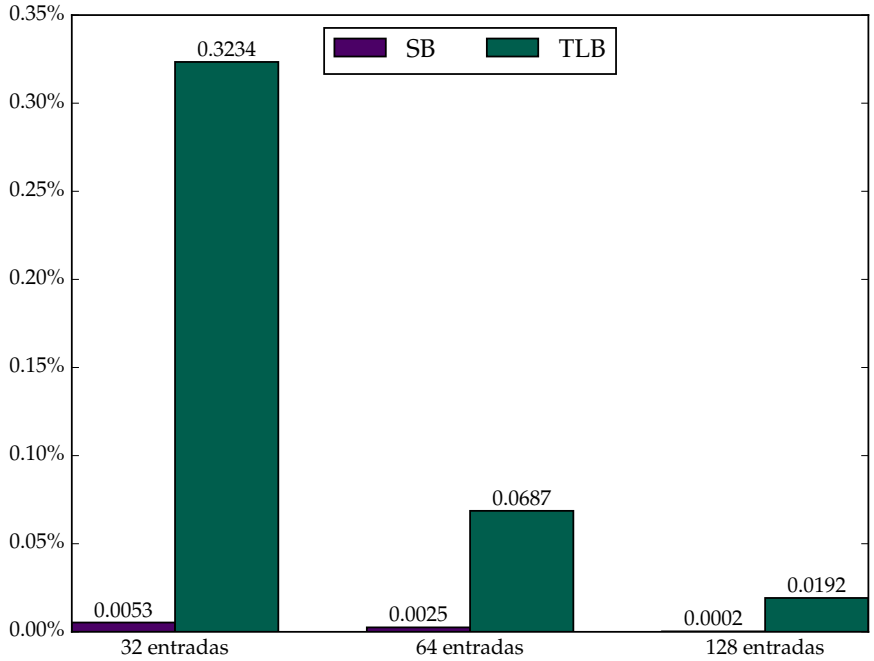
4.2.3 PyCharm

PyCharm é um *Integrated Development Environment* (IDE) para aplicações Python, escrita em Java. Os traços foram coletados no carregamento do programa com três arquivos de um projeto.

A figura 5 mostra as taxas de faltas para SBs e TLBs de 32, 64 e 128 elementos, respectivamente. Pode-se observar que a taxa de faltas em SBs é sempre menor do que a taxa de faltas em TLBs do mesmo tamanho – 61 vezes menor entre as caches de 32 elementos, 27 vezes menor entre as de 64 elementos, e 94 vezes menor entre as caches de

128 elementos. Mesmo a maior TLB, de 128 elementos, tem 3.6 vezes mais faltas do que a menor SB, de 32 elementos.

Figura 5 – Taxa de faltas por tamanho e tipo de cache - PyCharm

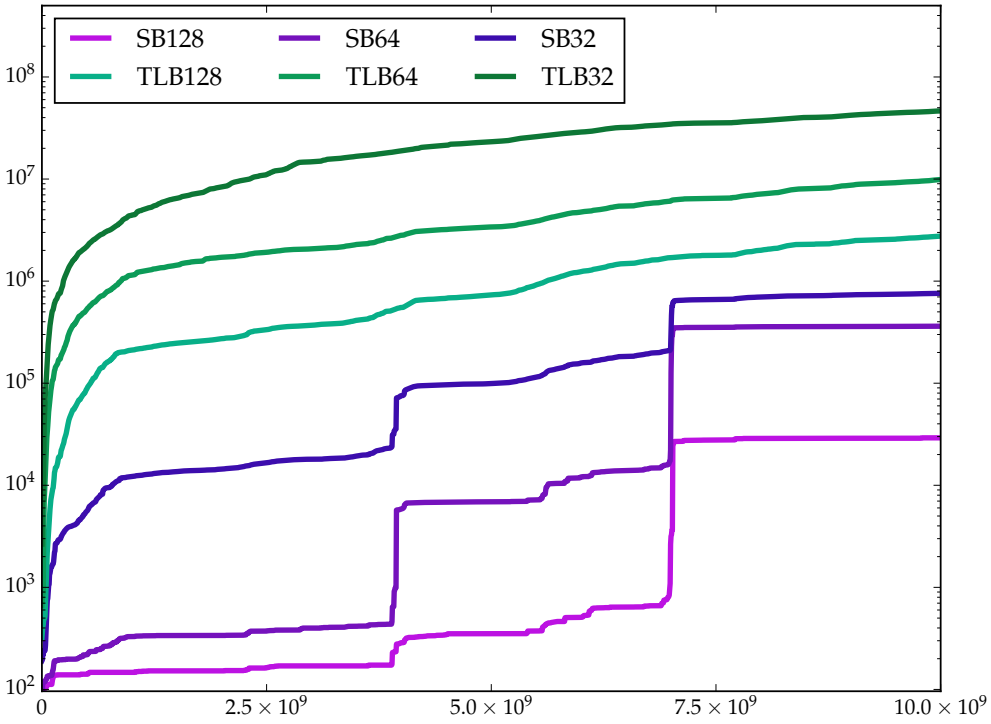


A figura 6 mostra a quantidade de faltas acumulada (no eixo vertical) para a quantidade de instruções processadas (no eixo horizontal). O eixo vertical é mostrado em escala logarítmica pois a quantidade de faltas nas TLBs é muito maior do que nas SBs. A quantidade de faltas cresce linearmente nas TLBs e tem picos bem destacados nas SBs. As bibliotecas mostradas nas SBs nos *snapshots* seguintes aos picos de faltas e que não estavam presentes nos snapshots anteriores– *libpixmap.so*, *libfreetype.so.6.16.1*, *libpng16.so.16.34.0*, *libgtk-x11-2.0.so.0.2400.32* e outras – sugerem que estes picos são relacionados ao carregamento da interface gráfica.

As linhas das TLBs não mostram os mesmos picos, o que pode significar que nenhum dos tamanhos simulados é suficiente para tirar proveito da localidade para esta aplicação: se as linhas nas TLBs estão sempre sendo substituídas, a mudança no conjunto de páginas úteis causada pela inicialização da interface gráfica fica "disfarçada"em meio a uma quantidade já muito alta de faltas.

Espaço em branco proposital.

Figura 6 – Quantidade de faltas acumulada por instruções processadas - PyCharm



4.2.4 VLC

VLC é um programa reprodutor de mídia. Os traços utilizados foram coletados durante a exibição de um fragmento de filme disponível na internet em formato AVC com resolução 1920×960 e cadência de 23.97 quadros por segundo.

A figura 7 mostra as taxas de faltas para SBs e TLBs de 32, 64 e 128 elementos, respectivamente. A taxa de faltas é 52 vezes maior para uma TLB de 32 elementos comparada uma SB de 32 elementos. Para as caches maiores, a taxa de faltas fica mais próxima, 43 vezes maior para uma TLB de 64 elementos comparada a uma SB do mesmo tamanho, e 17 vezes maior para uma TLB de 128 elementos comparada a uma SB do mesmo tamanho.

A figura 8 mostra a quantidade de faltas acumuladas em cada cache à medida em que as instruções foram processadas. As quantidade de faltas nas SBs de todos os tamanhos estabiliza no início da execução.

A taxa de crescimento da quantidade de faltas é linear para as TLBs, sendo suave para a TLB de 128 elementos e acentuada para TLBs de 64 e 32 elementos é mais acentuado, o que indica que estes dois últimos tamanhos de cache não são o suficiente para tirar proveito da localidade de referência para esta aplicação. O crescimento linear provavelmente se deve à natureza da exibição de vídeos, em que os dados são carregados em um *buffer*, interpretados e descartados em seguida para dar lugar a novos.

Figura 7 – Taxa de faltas por tamanho e tipo de cache - VLC

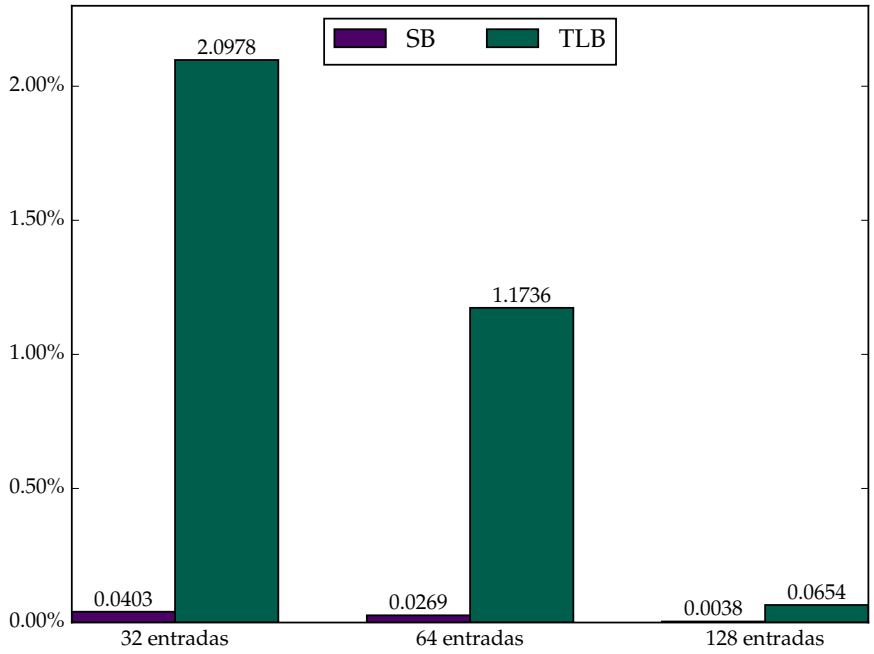
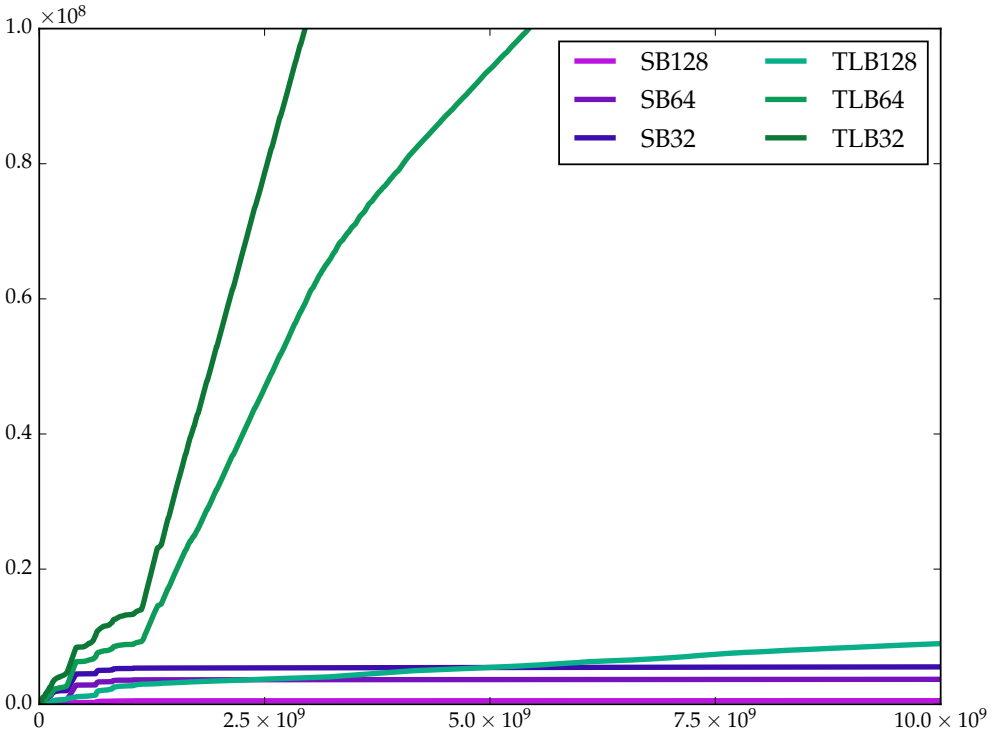


Figura 8 – Quantidade de faltas acumulada por instruções processadas - VLC



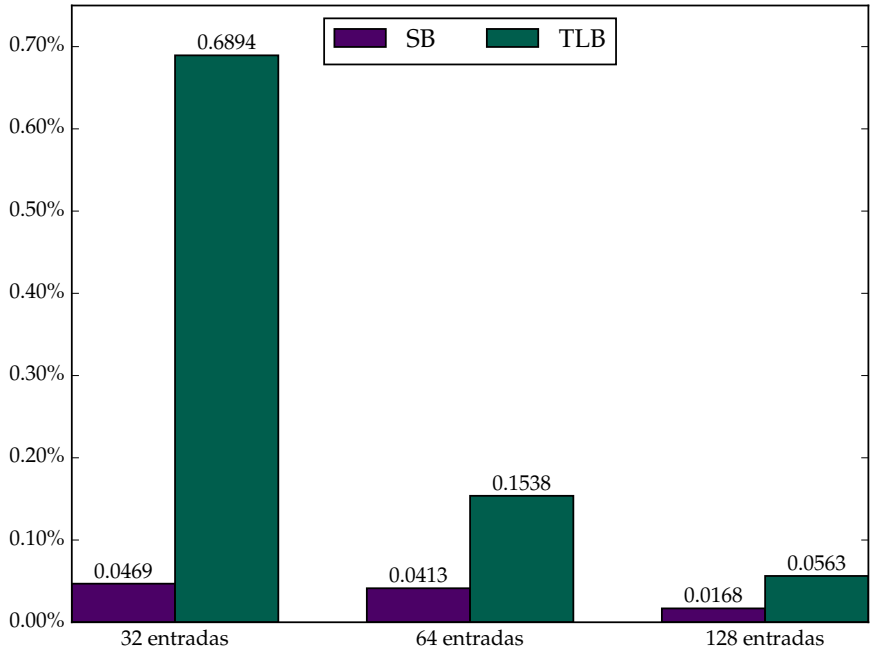
4.2.5 Blender

Blender é uma ferramenta de criação de modelos e animações em 3D.

Os traços foram coletados durante a renderização da animação de um objeto relativamente complexo – a cabeça de macaco, que é uma das formas padrão do programa, com efeito de "pegando fogo" com chamas e fumaça.

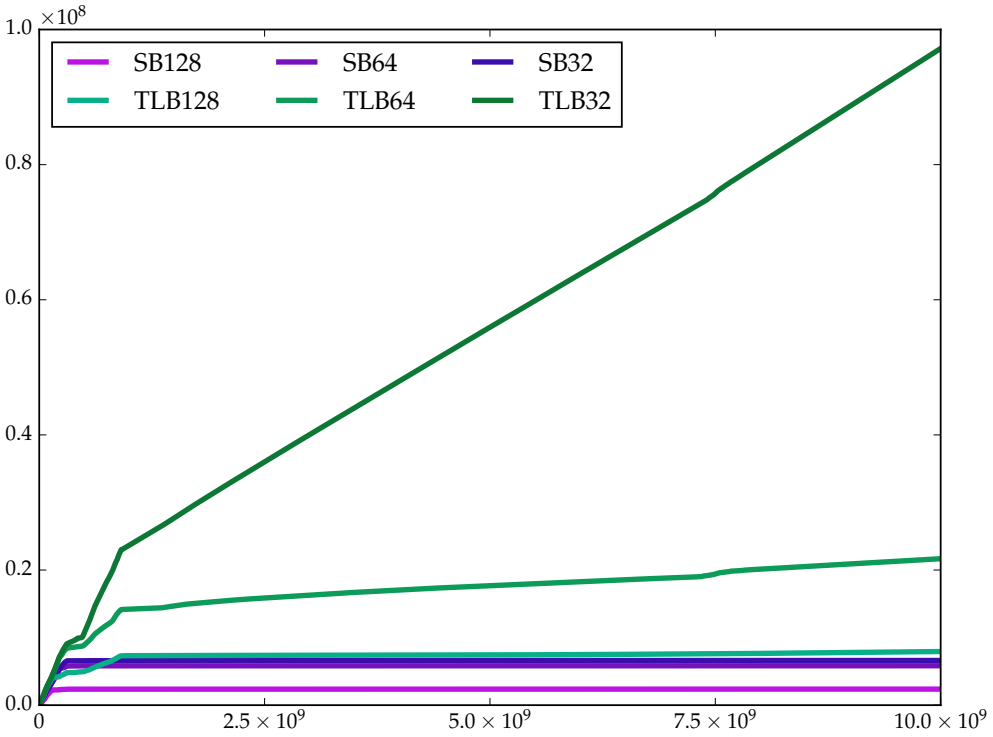
A figura 9 mostra a taxa de faltas para SBs e TLBs de 32, 64 e 128 elementos, respectivamente. Por ter uma quantidade maior de segmentos e envolver várias bibliotecas na renderização, a taxa de faltas é 3 vezes maior na TLB de 128 elementos do que na SB de 128 elementos.

Figura 9 – Taxa de faltas por tamanho e tipo de cache - Blender



A figura 10 mostra a quantidade de faltas em cada *cache*, acumuladas no processamento das instruções, em escala linear. A quantidade de faltas praticamente estabiliza em torno de 2.5×10^8 instruções para todos os tamanhos de SB e para a TLB de 128 elementos. No início do programa, no entanto, a TLB de 128 elementos tem um desempenho semelhante à SB de 32 elementos.

Figura 10 – Quantidade de faltas acumulada por instruções processadas - Blender



4.3 Observações gerais

A tabela 3 mostra a quantidade de páginas e segmentos mapeados durante o período de observação dos programas.

Tabela 3 – Quantidade de segmentos e páginas válidas por programa

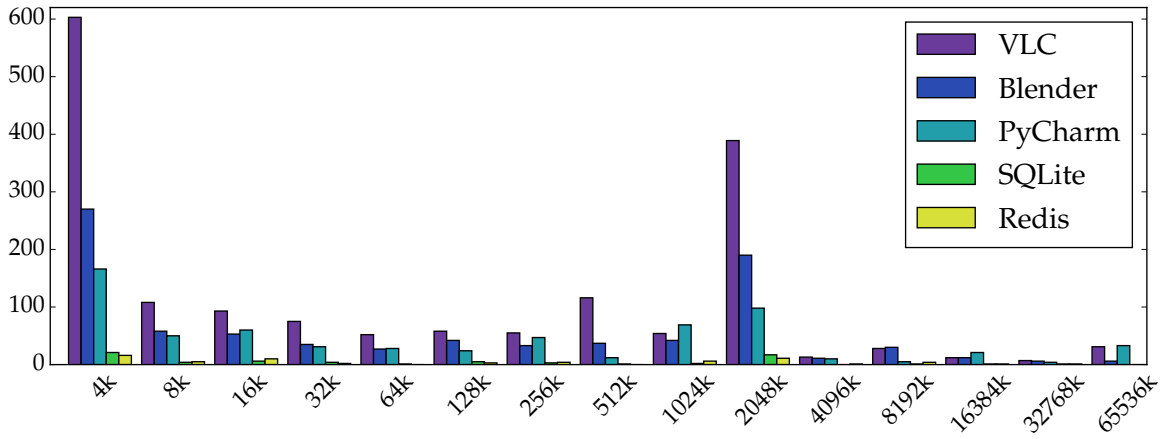
Programa	**Nº de páginas**	**Nº de segmentos**
Redis	22756	63
SQLite	17528	66
PyCharm	1197791	662
Blender	367209	852
VLC	854474	1694

Como esperado, a quantidade de segmentos é várias ordens de grandeza menor do que a quantidade de páginas, o que ajuda a explicar a diferença na taxa de faltas nas caches. Todas as aplicações analisadas têm menos segmentos do que a estimativa inicial máxima de 2 mil.

A figura 11 mostra um histograma de tamanhos de segmentos. A distribuição foi feita da mesma forma que em [Lau17], categorizando os segmentos pela potência de 2 maior mais próxima do seu tamanho.

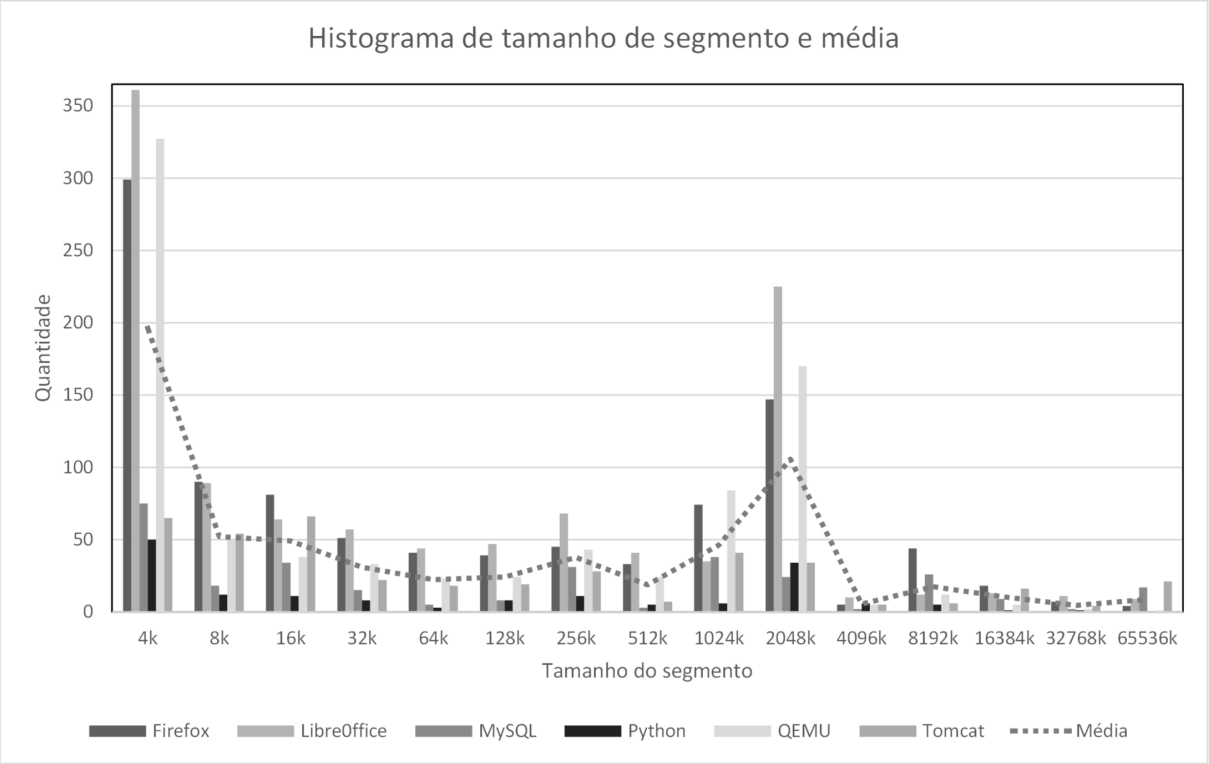
Pode-se observar um padrão na distribuição, sendo 4KBytes e 2048KBytes os tamanhos mais comuns. É provável que a alocação de dados seja planejada para otimizar o uso de páginas de 4KBytes, por isso a predominância de segmentos de até 4KBytes.

Figura 11 – Distribuição de segmentos por tamanhos aproximados



Estes resultados coincidem com os de Lauri [Lau17]. O histograma de segmentos por tamanho resultante das coletas de traços do seu trabalho está reproduzido na figura 12, e mostra uma frequência alta de segmentos de 4KBytes e de 2048KBytes.

Figura 12 – Histograma de tamanho de segmento e média



5 Conclusões e Trabalhos Futuros

A quantidade de segmentos das aplicações avaliadas, e consequentemente o tamanho da tabela de segmentos correspondente, é entre duas e três ordens de grandeza menor do que a quantidade de páginas válidas para todas as aplicações avaliadas. A maior tabela de segmentos chega a 1694 elementos, na execução do VLC, mas a quantidade de elementos válidos na tabela de páginas correspondente é de 854474, aproximadamente 504 vezes mais.

Mesmo em programas que usam muitas bibliotecas, como o Blender, a taxa de faltas é sempre muito menor em uma cache de segmentos do que em uma cache de tradução de páginas. Outras aplicações, como PyCharm, chegam a taxas de faltas até duas ordens de grandeza menores.

Estes resultados seguem a mesma linha dos obtidos em [Lau17] e corroboram a hipótese de que a segmentação deve ser reconsiderada como técnica de virtualização.

A questão de piora de desempenho com *copy-on-write* de blocos muito grandes levantada pelo caso das *huge pages* compartilhadas no Redis mostra, por outro lado, que voltar a usar exclusivamente segmentação para virtualizar a memória exige revisar todos os dispositivos auxiliares da memória virtual desenvolvidos e otimizados durante as décadas de vigência da paginação.

Além disso, segue necessário pesquisar e testar soluções para o problema da fragmentação externa.

Referências

[BCD72] A Bensoussan, C T Clingen, and R C Daley. The Multics virtual memory: Concepts and design. *Comm of the ACM*, 15(5):308–318, May 1972.

[Cor11] Jonathan Corbet. Transparent huge pages in [Linux Kernel] 2.6.38. https://lwn.net/Articles/423584/, 2011. [Online; acessado em 17/06/2018].

[DD68] Robert C Daley and Jack B Dennis. Virtual memory, processes, and sharing in Multics. *Comm of the ACM*, 11(5):306–312, May 1968.

[Den70] Peter J Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[HCG+15] P Hornyack, L Ceze, S Gribble, D Ports, and H M Levy. A study of virtual memory usage and implications for large memory. In *Proc Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2015.

[Hex18] Roberto A Hexsel. Software básico. Notas de aula, UFPR, Depto de Informática, 2018.

[Int18] Intel. Intel xeon processor scalable family. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-scalable-datasheet-vol-1.pdf, 2018. [Online; acessado em 17/06/2018].

[KELS62] T Kilburn, D B G Edwards, M J Lanigan, and F H Sumner. One-level storage system. In *IRE Trans on Electronic Computers*, EC-11, pages 223–235, 1962.

[Lau17] Lauri Paulo Laux Jr. De volta ao passado: Memória virtual com segmentação para máquinas com memória RAM quase infinita. Dissertação de mestrado, Universidade Federal do Paraná, 2017.

[LF15] Adam Litke and Steve Fox. libhugetlbfs README. https://github.com/libhugetlbfs/libhugetlbfs/blob/2f51e349b3c16d4c11a81836f1e81c4b6870000f/README, 2015. [Online; acessado em 17/06/2018].

[Red18a] RedHat. Huge pages and transparent huge pages. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge, 2018. [Online; acessado em 17/06/2018].

[Red18b] RedisLabs. Latency induced by transparent huge pages. https://redis.io/topics/latency#latency-induced-by-transparent-huge-pages, 2018. [Online; acessado em 17/06/2018].

[SGG12] A Silberschatz, P B Galvin, and G Gagne. *Operating Systems Concepts*. John Wiley, 9th edition, 2012. ISBN 978-1118063330.

[Wik18a] Wikipedia. Pigeonhole principle. https://en.wikipedia.org/wiki/Pigeonhole_principle, 2018. [Online; acessado em 17/06/2018].

[Wik18b] Wikipedia. Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer, 2018. [Online; acessado em 17/06/2018].

[Wik18c] Wikipedia. X86 memory segmentation. https://en.wikipedia.org/wiki/X86_memory_segmentation, 2018. [Online; acessado em 17/06/2018].