Universidade Federal do Paraná

Lauri P. Laux Jr

De volta ao passado: Memória Virtual com segmentação para máquinas com memória RAM "infinita"

Curitiba PR
2017

Lauri P. Laux Jr

De volta ao passado: Memória Virtual com segmentação para máquinas com memória RAM "infinita"

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Roberto Hexsel.

Curitiba PR
2017

L391d Laux Jr, Lauri P.
De volta ao passado: memória virtual com segmentação para máquinas com memória RAM “infinita” / Lauri P. Laux Jr. – Curitiba, 2017.
88 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2017.

Orientador: Roberto Hexsel.

1. Memória virtual. 2. Segmentação. 3. Paginação sob-demanda. 4. Simulação. I. Universidade Federal do Paraná. II. Hexsel, Roberto. III. Título.

CDD: 006.4

MINISTÉRIO DA EDUCAÇÃO
SETOR CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA

ATA Nº35

ATA DE SESSÃO PÚBLICA DE DEFESA DE MESTRADO PARA A OBTENÇÃO DO GRAU DE MESTRE EM INFORMÁTICA

No dia cinco de Dezembro de dois mil e dezessete às 10:00 horas, na sala AUDITÓRIO, Departamento de Informática, foram instalados os trabalhos de arguição do mestrando **LAURI PAULO LAUX JUNIOR** para a Defesa Pública de sua Dissertação intitulada **De volta ao passado: Memória Virtual com segmentação para máquinas com memória RAM quase infinita**. A Banca Examinadora, designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná, foi constituída pelos seguintes Membros: ROBERTO ANDRÉ HEXSEL (UFPR), MARCO AURELIO WEHRMEISTER (UTFPR), LUIS CARLOS ERPEN DE BONA (UFPR), . Dando início à sessão, a presidência passou a palavra ao discente, para que o mesmo expusesse seu trabalho aos presentes. Em seguida, a presidência passou a palavra a cada um dos Examinadores, para suas respectivas arguições. O aluno respondeu a cada um dos arguidores. A presidência retomou a palavra para suas considerações finais. A Banca Examinadora, então, reuniu-se e, após a discussão de suas avaliações, decidiu-se pela aprovação do aluno. O mestrando foi convidado a ingressar novamente na sala, bem como os demais assistentes, após o que a presidência fez a leitura do Parecer da Banca Examinadora. A aprovação no rito de defesa deverá ser homologada pelo Colegiado do programa, mediante o atendimento de todas as indicações e correções solicitadas pela banca dentro dos prazos regimentais do programa. A outorga do título de mestre está condicionada ao atendimento de todos os requisitos e prazos determinados no regimento do Programa de Pós-Graduação. Nada mais havendo a tratar a presidência deu por encerrada a sessão, da qual eu, ROBERTO ANDRÉ HEXSEL, lavrei a presente ata, que vai assinada por mim e pelos membros da Comissão Examinadora.

Curitiba, 05 de Dezembro de 2017.

ROBERTO ANDRÉ HEXSEL
Presidente da Banca Examinadora (UFPR)

MARCO AURELIO WEHRMEISTER
Avaliador Externo (UTFPR)

LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UFPR)

MINISTÉRIO DA EDUCAÇÃO
SETOR CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **LAURI PAULO LAUX JUNIOR** intitulada: **De volta ao passado: Memória Virtual com segmentação para máquinas com memória RAM quase infinita**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 05 de Dezembro de 2017.

ROBERTO ANDRÉ HEXSEL
Presidente da Banca Examinadora (UFPR)

MARCO AURELIO WEHRMEISTER
Avaliador Externo (UTFPR)

LUIS CARLOS ERPEN DE BONA
Avaliador Interno (UFPR)

Rua Cel. Francisco H. dos Santos, 100 - Centro Politécnico da UFPR - Curitiba - Paraná - Brasil
CEP 81531-980 - Tel: (41) 3361-3101 - E-mail: ppginf@inf.ufpr.br

Para Paula, Pedro e meu Pai. Sem vocês tudo seria muito mais difícil.

Agradecimentos

"What should we do then? Make the best use of what is in our power, and treat the rest in accordance with its nature." - Epictetus, Discourses I, 1.17

Agradeço a minha esposa Paula Baena pela paciência com minhas noites e fins de semana em que fiquei, de forma produtiva ou não, "fazendo as coisas do mestrado". Meu filho por entender que o pai estava "fazendo tarefa de casa". E também ao meu pai, que sempre está disponível em todas as situações que eu preciso de ajuda, de alguém para conversar ou apenas de conselhos. E agradeço a minha mãe pelo brilho nos olhos quando fala que seu filho agora é mestre.

Gostaria de agradecer ao meu orientador Roberto Hexel pela paciência e parceria nestes anos de mestrado. Aprendi muito com nossas conversas, divergências e nosso encontros regados a café.

Resumo

Memória virtual foi concebida na década de 1960 para contornar as limitações de uma memória RAM escarça e cara. Nos próximos anos esperamos ter computadores com 2^{64} bytes de memória RAM instalada. A paginação sob demanda é ineficiente para sistema com uma grande quantidade de memória porque o espaço (tabela de páginas) e o tempo (*page walks*) são custosos. Propomos a utilização de segmentação de memória e criação de um *buffer* de segmentos (*segment buffer - SB*) para diminuir o numero de mapeamentos entre memória virtual e física. Para testar nossa proposta coletamos traços de execução de 6 aplicações reais e comparamos a quantidade de faltas entre TLBs e SBs de tamanho e complexidade similares aos encontrados em processadores comerciais. A quantidade de faltas nas SBs é de 2 a 4 ordens de magnitude menores que as TLBs. Discutimos as implicações no projeto de sistemas com segmentação e SBs.

Palavras-chave: Memória Virtual, Segmentação, Paginação sob-demanda, Simulação.

Abstract

Virtual Memory was devised in a time of scarce resources. In the coming decade we expect to see physical memory systems populated with 2^{64} bytes of RAM. Demand paging is inefficient for such large memories because the space (Page Tables) and time (Page Table walks) overheads become too high. We propose a segmented memory model with a segment buffer (SB) to reduce the number of virtual to physical address mappings. To test the proposal we collected execution traces from 6 applications and then measured the miss ratios for TLBs and Segment Buffers (SBs) of similar complexity to those found in current x86-64 processors. The miss rates for SBs are 2 to 4 orders of magnitude smaller than those for TLBs. We discuss some of the design implications of segmented systems and of SBs.

Keywords: Virtual Memory, Segmentation, Demand paging, Simulation.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

SB	*Buffer* de segmentos (Segment Buffer)
TLB	*Translate Look-aside Buffer*
TLB32	TLB com 32 posições, totalmente associativa
TLB64	TLB com 64 posições, totalmente associativa
TLB128	TLB com 128 posições, com 8 conjuntos associativos (*8 way set-associative*)
TLB256	TLB com 256 posições, totalmente associativa
TLB1024	TLB com 256 posições, totalmente associativa
SB32	SB com 32 posições, totalmente associativa
SB64	SB com 64 posições, totalmente associativa
SB128	SB com 128 posições, totalmente associativa
SB128Assoc	SB com 128 posições, com 8 conjuntos associativos
B	byte (2^0)
KiB	kibibyte (2^{10})
MiB	mebibyte (2^{20})
GiB	gibibyte (2^{30})
TiB	tebibyte (2^{40})
PiB	pebibyte (2^{50})
EiB	exbibyte (2^{60})
ZiB	zebibyte (2^{70})
YiB	yobibyte (2^{80})

Capítulo 1

Introdução

Processadores de 64 *bits* de uso geral foram introduzidos para o mercado de consumo por volta de 1995 com uma capacidade de endereçamento de memória e com barramentos com largura entre 2^{32} a 2^{36} *bytes*. Por volta de 2005 a quantidade de bits disponíveis para o endereçamento de memória atingiu 40 *bits*. Recentemente temos barramentos disponíveis em processadores modernos com 2^{50}. Se a tendência de crescimento de 1/2 a 1 *bit* de endereçamento por ano for mantida, teremos em meados da década de 2020 processadores com barramentos utilizando todos os 64 *bits* disponíveis para endereçamento [27]. Quando nos referimos a uma quantidade "infinita" de memória *RAM* estamos considerando um espaço de endereçamento de 2^{64} bits.

O gerenciamento de uma quantidade tão grande de memória utilizando paginação sob demanda é custoso e ineficiente. As razões para essa afirmação são: (1) com uma quantidade de memória *RAM* enorme, a estrutura de dados para manter a paginação (a tabela de páginas, PT) é também enorme; e (2) máquinas com uma grande quantidade de memória *RAM* (e programas capazes de utilizar essa memória) referenciam grandes volumes de dados o que causa tráfego adicional da *RAM* para a *TLB* (*Translate Lookaside Buffer*), dispositivo responsável pela tradução de endereços virtuais em endereços físicos.

O uso de paginação sob demanda em cargas de trabalho modernas está sendo questionado devido ao impacto na performance em sistemas com grande quantidade de memória *RAM*. Hornyack *et.alli.* [28] mostram que a quantidade de faltas na *TLB* implica em um perda significativa no desempenho de algumas aplicações devido ao número de ciclos que o processador gasta resolvendo essas faltas. O impacto na performance ultrapassa os 50% para algumas cargas de trabalho. Hornyack sugere que memória virtual utilizando segmentação pode ser um bom candidato para diminuir a quantidade de mapeamentos entre memória virtual e física.

O uso de segmentação para implementar memória virtual em sistemas com uma quantidade enorme de memória *RAM* tende à diminuir a quantidade de mapeamentos necessários entre memória física e virtual. O mecanismo de segmentação poderia ser implementado de forma similar ao proposto no *MULTICS* [7].

Praticamente nenhum dos processadores modernos suportam segmentação. A arquitetura *Intel x86* possui suporte nativo, mas a arquitetura sucessora, *Intel x86-64* definida há mais de 15 anos removeu o suporte nativo a segmentação. O mercado de processadores de uso geral não considera segmentação como uma solução adequada, mas a quantidade de memória *RAM* disponível hoje a um custo baixo é duas ou três ordens de grandeza maior que no incio dos anos 2000.

Trabalhos recentes consideram a possibilidade da utilização de memória *RAM* não volátil com densidade e desempenho e similares à tecnologia *DRAM* (*Dynamic Random Access Memory*).

Nesse cenário, o uso de segmentação pode trazer outros benefícios, como tratar arquivos como segmentos de maneira similar ao sistema de arquivos implementado no *MULTICS* [7].

Propomos a utilização de segmentação de memória em conjunto com uma *cache* totalmente associativa que mantenha próximo ao processador os segmentos mais utilizados. Para testar nossa proposta coletamos traços de execução de 6 programas reais executados em um processador Intel x86-64 e comparamos o desempenho da tradução de endereço utilizando uma *TLB* para paginação e uma SB para segmentação. A ferramenta que utilizamos para simulação do *SB* cria uma tabela de segmentos (*Segment Table, ST*) para o processo simulado e re-escreve os endereços lineares em endereços segmentados, compostos de um identificador de segmento (*SegmentID*) e um deslocamento (*displacement*) dentro do segmento. De posse do endereço linear e do endereço segmentado, calculamos as taxas de faltas na *TLB* e no *SB*.

Nossas medições mostram que a taxa de faltas para tradução de endereço utilizando um *SB* é 10^2 à 10^4 vezes menor que utilizando paginação sob demanda com uma *TLB*. Esse resultado, e os ganhos potenciais de gerenciar uma estrutura de dados menor e mais simples, indicam que para máquinas modernas com uma grande quantidade de memória *RAM*, o modelo de gerenciamento de memória utilizando segmentação pode ser uma alternativa viável para substituir a paginação sob demanda.

No Capitulo 2 é apresentado o conceito de memória virtual, paginação sob demanda e segmentação, e como o gerenciamento de memória virtual utilizando paginação sob demanda pode ser custosa para alguns tipos de aplicação. Abordamos e discutimos brevemente algumas das soluções propostas para diminuir o impacto na performance da tradução de endereço para paginação sobre demanda no Capítulo 3. No Capítulo 4 explicamos como foram coletados os traços de execução e como esses traços foram simulados de modo a comparar a quantidade de faltas na *TLB* e na *SB*. Apresentamos os resultados obtidos com o simulador no Capitulo 5 e apresentamos as conclusões do trabalho no Capitulo 6.

Capítulo 2

Fundamentação

2.1 Memória Virtual

Em meados da década de 1950, o tamanho dos programas começou a ficar maior que a memória física (memória *RAM*) disponível, o que levou ao desenvolvimento do conceito de memória virtual. O objetivo da memória virtual é permitir o compartilhamento seguro e eficiente da *RAM* entre vários programas, e remover as limitações de programação decorrentes de uma quantidade pequena de memória principal.

Os primeiros sistemas utilizavam segmentação para implementar memória virtual porque o conceito de segmentos é o mais intuitivo para o programador. Um programa é dividido logicamente em três segmentos: (1) código (*code* ou *text*); (b) dados (*data*); e (c) pilha (*stack*). Para executar um programa, partes da memória *RAM* deveriam ser alocados para esses segmentos [12].

Segmentação tem uma grande desvantagem: conforme o sistema é utilizado, e novos programas são carregados para a memória, os espaços contíguos para alocação de novos segmentos diminuem até o ponto em que os espaços entre os segmentos de memória são muito pequenos para que novos segmentos possam ser alocados. Se somarmos os espaços entre todos os segmentos há espaço livre suficiente para acomodar novos segmentos. A isso chamamos de "fragmentação externa" (*external fragmentation*). Existe espaço disponível na memória, embora não seja possível criar novos segmentos porque o espaço está fora dos segmentos alocados em memória, e esses espaços de memória contígua livres são muito pequenos para acomodar novos segmentos.

Um grupo da universidade de Manchester projetou o primeiro sistema com paginação sob demanda para gerenciamento de memória [30]. Nesse projeto, a memória seria dividida em blocos de tamanho fixo (páginas) e a capacidade da memória física foi aumentada utilizando "memória virtual" (*virtual memory*). Uma combinação inteligente entre *software* (o sistema operacional) e o uso de memória secundária permite oferecer ao usuário mais memória *RAM* que o computador realmente possui. Assim, do ponto de vista do programador, a percepção é que a memória *RAM* é infinita.

A paginação sob demanda produz "fragmentação interna" porque cada segmento de programa (segmento de código por exemplo) desperdiça em média metade de uma página [1]. Um arquivo mapeado em memória com tamanho de 3KiB precisa estar em uma página com 4KiB, desperdiçando 1KiB. Novas páginas somente são alocadas na memória *RAM* se a dinâmica da execução do programa mostrar que tal é necessário.

Com paginação, a utilização de memória *RAM* melhorou significativamente em detrimento ao aumento de trafego entre a memória *RAM* e a memória secundária, causada pela

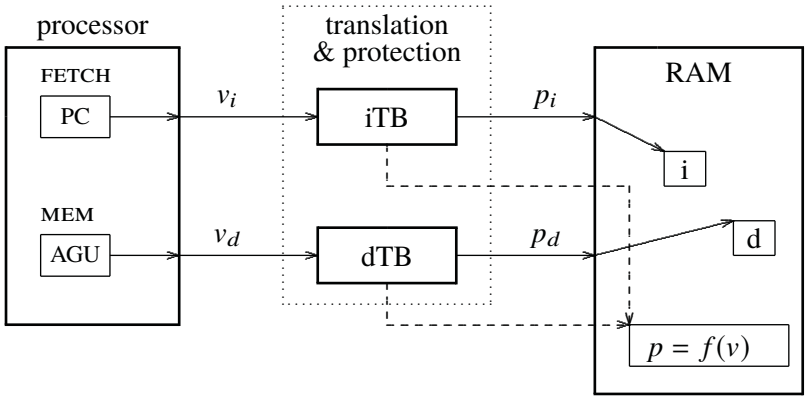

Figura 2.1: Modelo conceitual de memória virtual.

transferência frequente de páginas entre os níveis de memória. Quando um novo programa precisa de mais memória *RAM* do que o sistema tem disponível, o sistema operacional transfere páginas da memória primária para o disco, liberando espaço para que novos programas possam ser carregados do disco para a memória e executados. Esse modelo de gerenciamento de memória foi utilizado com sucesso por mais de cinquenta anos.

A Figura 2.1 mostra um modelo conceitual de um sistema moderno de gerenciamento de memória. Dois fluxos de endereços são gerados pelo processador, um para o *Program Counter* (*PC*) e outro da *address generation unit* (*AGU*). Essas duas unidades geram endereços v_i e v_d para referenciar instruções e dados. Para cada fluxo de endereços existe um "*translation buffer*" (*TB*) entre o processador e a memória principal. Os *TBs* são implementados com memória associativa rápida e guardam um pequeno sub-conjunto do mapa de memória do processo, representado pela função $f(\,)$ no diagrama. A memória principal é referenciada com os endereços físicos p_i e p_d.

Para proteger o espaço de memória, cada programa tem uma faixa de endereços de memória disponíveis para uso privado, que nenhum outro programa pode acessar. Essa faixa de endereços de memória é chamada de espaço de endereçamento (*address space*). A memória virtual implementa o mapeamento entre os endereços do espaço de endereçamento privado do programa (endereços virtuais) e os endereços reais (físicos) presentes na memória *RAM* [34].

Memória virtual permite o compartilhamento seguro e eficiente da *RAM* entre vários programas, e remove as limitações de programação decorrentes de uma quantidade pequena de memória instalada. Para um dado processo, o seu mapa de memória guarda todas as associações válidas entre endereços virtuais e físicos. O mapa provê duas funções importantes: (a) alocação de memória física removendo o acoplamento entre endereços físicos e virtuais de forma que o processo possa usar qualquer área de memória livre disponível; e (b) segurança para impedir que um processo referencie endereços alocados a outro processo. Para aumentar a performance, essas funções precisam ser rápidas e portanto devem ser implementadas próximas ao processador.

Quando um processo inicia, seu mapa de memória é parcialmente preenchido pelo carregador (*loader*), e um pequeno sub-conjunto de instruções e dados são carregados da memória secundária. Conforme a execução continua, um subconjunto diferente do espaço de endereçamento pode ser carregado, e o mapa de memória deve ser atualizado para refletir os novos endereços.

O domínio da função de mapeamento $f(\,)$ é todo o espaço de endereçamento, geralmente 2^{64} nos processadores modernos. A imagem de $f(\,)$ é algum sub-conjunto do espaço de endereçamento. A função principal da memória virtual é prover, do ponto de vista do usuário, a percepção é de uma memória *RAM* infinita. Temos que admitir que 2^{64} é um número muito grande.

2.2 Paginação

O modelo conceitual mostrado na Figura 2.1 é válido tanto para segmentação quanto para paginação sob demanda. Em paginação, os blocos de memória são chamados de páginas e o tamanho de página mais comum é 4KiB. Alguns sistemas possuem *super pages/huge pages*, com tamanhos que variam de 64KiB até 1GiB, dependendo do modelo do processador e do projeto do sistema operacional.

A função de mapeamento é chamada "tabela de páginas" (*Page Table*, PT), e o tamanho do seu domínio, para páginas com 2^n bits $2^{64}/2^n = |f()|$. Para páginas com 4K *bytes*, a tabela de páginas possui até 2^{52} elementos. Para páginas com tamanho de 1G *bytes* o domínio é de 2^{34} elementos. Mesmo com páginas maiores o número de elementos da tabela de página é significativo.

Cada elemento da tabela de páginas é codificado com 8 ou 16 *bytes* o que torna essas tabelas muito grandes. Projetos bem feitos utilizam modelos hierárquicos para reduzir o tamanho da tabela de páginas. Mesmo assim, para algumas aplicações como gerenciadores de banco de dados, a tabela de páginas é proporcionalmente grande. Pelo menos um pequeno sub-conjunto da tabela de páginas precisa estar sempre presente na memória *RAM*.

O endereço virtual é dividido em dois campos, um número de pagina virtual (*Virtual Page Number, VPN*) e um deslocamento (*displacement*) dentro da página. Quando o processador procura uma referência, o VPN é pesquisado na *translate look-aside buffer* (TLB). Se é encontrada uma referência ao VPN (um *hit* na TLB) a referência segue para memória *RAM* e é então concluída. Se o mapeamento do endereço não é encontrado na TLB (um *miss* na TLB), a tabela de páginas é indexada com o VPN e, se o mapeamento é válido, o mapeamento do endereço virtual (VPN) para a memória física (*physical page number*, PPN) é carregado na *TLB*, possivelmente substituindo um mapeamento existente. As ações de gerenciamento da *TLB* podem ser executadas por uma máquina de estados (como nos processadores Intel x86) ou por um sequencia de instruções (como nos processadores MIPS).

Se o mapeamento na tabela de páginas não é válido, é possível que a página não esteja presente na memória *RAM* e precise ser trazida da memória secundária (uma falta de página). Também é possível que a referência seja para uma região não mapeada no espaço de endereçamento (uma *segmentation fault*), ou a referência é considerada ilegal (uma *protection fault*). De qualquer modo o sistema operacional (OS) precisa executar uma séria de operações para se recuperar das falhas, ou terminar a execução do processo.

Cada processo possui uma *PT*. Em sistemas bem projetados, o gerenciamento da tabela de páginas é feito apenas em *kernel mode*, garantindo a proteção de memória e não expondo ao usuário nenhum detalhe da existência da *PT* ou mesmo da memória física.

Hoje, de dispositivos móveis até grande servidores com dezenas de núcleos, a quantidade de memória *RAM* é medida em *gigabytes*. Mesmo dispositivos móveis mais simples possuem uma quantidade de memória igual, ou mesmo superior, a servidores de dez anos atrás. Com a popularização da oferta de memória *RAM*, a um custo relativamente baixo, a estratégia de paginação não é mais necessária para dar ao usuário a ilusão de uma memória *RAM* infinta [28]. Atualmente, a quantidade de memória primária disponível para um computador é diversas ordens de grandeza maior do que quando a estratégia de memória virtual com paginação foi inventada.

Para aplicações de larga escala que utilizam grande quantidade de memória, a paginação pode impactar significativamente no desempenho. Programas com essas características passam boa parte do seu tempo gerenciando faltas na *TLB*. Hornyack *et.alli* [28] demonstram que o impacto pode variar de uma pequena porcentagem dos ciclos do processador até 58% para algumas cargas de trabalho. Isso é esperado pois faltas na *TLB* são custosas e o espaço na *TLB* é

limitado devido à complexa implementação em *hardware* de *caches* associativos. Quanto mais dados o programa utiliza, mais mapeamentos de páginas são necessários, e mais provável é a necessidade de utilizar uma página cujo mapeamento não está na TLB.

Os mecanismos utilizados para implementar memória virtual com paginação (páginas, tabela de páginas e *TLBs*) funcionaram bem durante mais de cinquenta anos. Entretanto esses mecanismos estão começando a mostrar sua idade devido a tendências recentes como: (a) aumento significativo da memória física disponível; (b) aparecimento de aplições para análise intensiva de grandes volumes de dados; e (c) a eminente possibilidade de termos memória *RAM* não volátil com desempenho e capacidade próximas de *DRAM* [28].

Além disso, os custos da implementação em hardware, tanto em área de silício quanto em custo energético, de mecanismos de paginação são elevados. Em alguns casos chegam até a 15% do custo energético e, aproximadamente, a mesma proporção em espaço ocupado para implementar esse recurso em *hardware* [28].

2.3 Segmentação

Programadores não veem a memória *RAM* como um arranjo linear de *bytes*, alguns bytes contendo instruções e outros contendo dados. Em vez disso programadores preferem considerar a memória como um conjunto de segmentos de tamanho variável, os quais não estão dispostos em nenhuma ordem ou sequência em particular. É natural considerar um programa como um conjunto de sub-rotinas, procedimentos, funções ou módulos, diversas estruturas de dados: tabelas, arranjos, pilhas e variáveis. Um nome é usado para referenciar a cada um desses módulos ou estruturas de dados.

O mecanismo de segmentação possibilita uma visão de memória em que o espaço de endereçamento lógico é uma coleção de segmentos. Cada segmento tem um nome e um tamanho, e os endereços de memória são compostos por duas partes: o identificador do segmento e a posição dentro do segmento [1].

Um processo precisa de pelo menos três segmentos: (a) código (*code* ou *text*) que contém o binário executável do programa; (b) dados (*data*) que contém as estruturas de dados que são utilizadas pelo programa; e (c) pilha (*stack*). Normalmente o tamanho do segmento de código é determinado em tempo de compilação e o tamanho dos segmentos de dados e pilha podem variar durante a execução. Cada biblioteca ligada dinamicamente ao processo contribui com segmentos de código e dados, e esses segmentos precisam ser adicionados ao espaço de endereçamento do processo.

Um endereço virtual segmentado é dividido em duas partes, o número do segmento (*Segment Number*, VSN) e um deslocamento dentro do segmento (*displacement*) que possui um tamanho arbitrário. Normalmente uma parte dos *bits* mais significativos do endereço são reservados para indexar a tabela de segmentos, ou o *buffer* de segmentos (*Segment Buffer*, SB). As ações necessárias para traduzir um endereço virtual para o seu endereço físico correspondente são similares aos da paginação.

É necessário mapear os endereços virtuais segmentados que são bidimensionais (compostos por identificador do segmento e deslocamento) para a memória física que é constituída como uma sequência unidimensional de *bytes*. Esse mapeamento é feito por intermédio da tabela de segmentos (*segment table*, ST).

Cada entrada na tabela de segmentos tem uma base e um tamanho. A base do segmento contém o endereço físico inicial a partir do qual o segmento está armazenado na memória, e o tamanho especifica o endereço físico limite (base + tamanho) deste segmento. O acesso a um elemento da tabela de segmentos contido em uma *cache* associativa pode ser feito rapidamente, e

a operação de adição com a base e a comparação com o limite podem ser feitas simultaneamente para economizar tempo [1].

Se o mapeamento do segmento é encontrado no *segment buffer* (um *hit* no SB), o endereço físico é enviado a memória *RAM*. Se ocorre uma falta (SB *miss*) é necessário procurar o número do segmento físico (*physical segment number*, PSN) na tabela de segmentos na memória *RAM*, e o mapeamento (VSN $\mapsto$ PSN) é carregado no SB, possivelmente substituindo outro mapeamento.

Se o mapeamento na tabela de segmentos for inválido existem três possibilidades: (a) pode ser um SB *miss* e o segmento que causou o *miss* precisa ser carregado da ST, alocada na memória *RAM*; (b) pode ser uma *protection fault* decorrente de uma referência a um endereço ilegal para o processo sendo executado; ou (c) uma *segmentation fault* decorrente de uma referência a um endereço fora do espaço de endereçamento.

A tecnologia disponível durante as décadas de 1960 e 1970 era insuficiente para implementar um sistema ambicioso como o *MULTICS*. Os projetistas do sistema sugeriram que o sistema de arquivos fizesse parte do mecanismo de memória virtual [7]. Mesmo que o sistema operacional considere arquivos abertos como segmentos, como acontece no sistema operacional *MULTICS*, a tabela de segmentos precisaria, a grosso modo, de algo em torno de 2000 elementos. Uma tabela de segmentos é muito menor que uma tabela de páginas. O número aproximado de segmentos de um processo é diversas ordens de grandeza menor que o número de páginas que o processo pode necessitar.

Uma das grandes desvantagens da segmentação é a fragmentação externa de memória. Conforme os processos são criados e mortos, a memória disponível para novos processos fica fragmentada em pedaços pequenos. Mesmo considerando um sistema com memória "infinita", se o sistema executar por um longo período de tempo, a memória ficará fragmentada.

Para minimizar os problemas decorrentes da segmentação, durante o ciclo de vida do sistema operacional, fragmentos de memória devem ser identificados e os segmentos organizados (compactados) de forma que um novo segmento obtenha espaço livre para ser criado. Como o mecanismo de segmentação usa um algoritmo de tradução dinâmica de endereços, a compactação de memória pode ser feita em qualquer momento. Se o escalonador de CPU precisa esperar que um processo termine, devido a um problema de alocação de memória, ele pode continuar a procurar na fila de processos prontos por outro processo que possa ser executado. Se o tamanho médio dos segmentos é pequeno, a segmentação externa será pequena. Como os segmentos são menores que o processo como um todo, é mais provável que sejam encontrados blocos disponíveis de memória para armazená-los [1].

A escolha do bloco de memória a ser utilizado pode ser realizado utilizando o algoritmo de melhor encaixe. Esse algoritmo escolhe o menor bloco que seja suficientemente grande para satisfazer a requisição. A tabela de segmentos deve ser ordenada de acordo com o tamanho dos blocos livres, de forma a evitar que toda a tabela de segmentos seja percorrida [1]. Essa estratégia fornece o bloco cujo espaço livre no bloco alocado a cada requisição seja o menor possível.

Hornyack et.alli demonstram que aplicações que consomem grandes quantidades de memória apresentam um comportamento de alocar itens de código ou de dados em regiões contíguas de memória, que são mapeadas em páginas virtuais contíguas. Esse padrão de uso cria o que eles chamam de *Virtual Memory Areas* (*VMAs*). Utilizar segmentação para representar esse arranjo contíguo de páginas em memória reduziria significativamente o esforço necessário para manter informações de proteção e mapeamento de páginas virtuais em páginas físicas.

Pelos motivos expostos, um mecanismo simples de segmentação de memória é suficiente para fornecer proteção e mapeamento de endereços virtuais em físicos. Nos referimos a segmentos de forma natural, pois são mais próximos de como os programas são organizados. Normalmente,

um programa é composto por um segmento de código, um segmento para dados e *heap* e um segmento para a pilha (*stack*). Os dois últimos podem crescer para acomodar o ciclo de vida do programa. Para criar um sistema operacional baseado em segmentação, é necessário manter uma tabela de segmentos pequena para cada processo em um *buffer* de mapeamento de segmentos (*SB*) que mantém próximo ao processador as informações de qual é o endereço inicial de cada segmento, qual seu limite, e quais as permissões deste segmento.

2.4 Memória *RAM* não volátil – *NVRAM*

A memória *RAM* não volátil (*NVRAM*) tem o potencial de influênciar o projeto de sistemas operacionais, componentes de sistema, programas e o desempenho e confiabilidade dos computadores em um futuro próximo [5]. Novos projetos de memória virtual precisam levar em conta que a *NVRAM* com desempenho, capacidade e latência similares aos da *DRAM* seja parte, total ou parcial, da memória principal das máquinas do futuro. Muitas das estratégias que utilizamos atualmente para gerenciar a memória primária de nossas máquinas, como a paginação sob demanda, não fazem mais sentido se a memória *RAM* for abundante e persistente.

Tecnologias emergentes de memória não volátil e com endereçamento em bytes (*byte addressable*) como *phase-change-memory* e *memristors* oferecem uma alternativa ao disco magnético, com performance da mesma ordem de magnitude da *DRAM* [36]. A densidade que se espera conseguir indica que a memória não volátil será abundante e barata. Assim a grande vantagem da paginação para gerenciar memória primária deixa de ser importante [5].

Várias perguntas relacionadas ao projeto de *software* para a tecnologia *NVRAM* estão em aberto. Podemos destacar as seguintes: (a) devemos considerar a *DRAM* um cache para a *NVRAM*? (b) devemos construir sistemas com *DRAM* (rápida) e *NVRAM* (lenta)? (c) como as aplicações devem ser projetadas para se beneficiarem da organização de memória e decidir qual o melhor local para guardar seus dados? (d) como as aplicações se comportam para diferentes tempos de acesso (*latency*) e vazão (*bandwidth)* das *NVRAM* [36]? e (e) supondo que a metade da memória principal seja composta por *NVRAM*, quais ganhos de eficiência, velocidade e simplicidade são viáveis através de mudanças no sistema operacional [9]?

2.4.1 Paginação e memória não volátil

Qual a utilidade da paginação se parte da memória principal for rápida, abundante e não volátil?

Supondo que a metade da memória principal seja composta por *NVRAM* [9], quais ganhos de eficiência, velocidade e simplicidade são viáveis através de mudanças no sistema operacional? Vamos supor também que a capacidade total de memória *RAM*, somando *DRAM* e *NVRAM*, seja próxima do *petabyte* num computador com barramento de 50 *bits*, a pergunta é: a paginação faz sentido em um cenário como este?

Provavelmente não. Para projetos considerados nesse cenário é necessário repensar uma grande parte do sistema operacional devido a introdução da memória não volátil. Sem discos mecânicos ou memória secundária lenta, a paginação é apenas um mecanismo que faz alocação de memória e provê segurança através da separação dos espaços de endereçamento. Com NVRAM não é necessário que páginas sejam persistidas em memória secundária. Seguindo esse raciocínio precisamos apenas de um mecanismo para alocação de memória e de proteção que podem ser implementados com um mecanismo mais simples tal como a segmentação de memória.

2.4.2 Endereçamento

Existem propostas para criar uma interface de persistência de dados que seja endereçável por byte em vez de endereçável por blocos [10][8][37]. Isso transformaria radicalmente a forma como informações persistentes são gerenciadas e guardadas [4], embora não aproveite todo o potencial que a *NVRAM* oferece.

Se a interface dos discos de estado sólido (*SSDs*) permitir transferências do tamanho de um *byte* ou de uma palavra (4 *bytes*) muitas aplicações serão beneficiadas pela diminuição de tráfego entre a memória e o disco. Benefícios ainda maiores serão alcançados com uma menor granularidade da unidade de armazenamento. Considere, por exemplo, a complexidade necessária para que sistemas de banco de dados mantenham a consistência dos dados armazenados em um bloco no disco. Se removermos o conceito de "bloco" do sistema e substituirmos por uma interface com endereçamento direto, como a memória *RAM*, *locking* em banco de dados pode se tornar coisa do passado. Como resultado teremos sistemas de banco de dados mais simples, rápidos e confiáveis.

A ideia de um dispositivo com uma interface baseada em blocos, ou *block device interface*, é uma ideia que percola diversas camadas do sistema operacional. Implementando uma *character interface* para armazenamento pode levar a simplificação de uma grande parte do sistema operacional [8].

2.4.3 Consistência e reinicialização

Suponha que a tecnologia de memória não volátil seja acessível em termos de custo e capacidade em um futuro próximo. Suponha também que a *NVRAM* seja rápida a abundante. Se temos a garantia da persistência dos dados, essas características levam a necessidade de um novo projeto de sistema de memória virtual.

Tome, por exemplo, a persistência de dados. Com *NVRAM*, um simples *flush* das caches do processador é suficiente para garantir que o estado de memória da aplicação seja persistente, e consistente, sem a necessidade de mecanismos complexos de *callback* e *locking* [4]. Se o processador salvar seu estado, e conteúdo da cache, antes de uma reinicialização, o custo para continuar a execução também é similar ao custo de uma troca de contexto.

2.4.4 Sistema de arquivos

Utilizar um sistema de arquivos para acessar *NVRAM* não é muito eficiente. Tradicionalmente, os sistemas de arquivo não são projetados para operar em dispositivos com uma latência pequena. O disco mecânico e *SSDs* tem latências grandes o suficiente para esconder os custos em *software* como, por exemplo, consistência de dados com *journalling*. A latência da *NVRAM* é pequena o bastante para que a utilização de interfaces de armazenamento tradicionais seja contraprodutivo [4]. Conforme a tecnologia for amadurecendo, as aplicações tais como sistemas de arquivo e caches de banco de dados utilizarão a memória não volátil diretamente como memória principal e não como dispositivo de armazenamento.

Supondo que a memória *RAM* seja "infinita", persistente e composta por apenas um nível, por que não transformar arquivos em segmentos que permanecem em memória? Quando um processo abre um arquivo, o sistema adiciona um novo descritor de segmento na tabela de segmentos do processo. Para fechar o arquivo o segmento correspondente é movido para uma tabela de segmentos de arquivos temporariamente não utilizados. Se um arquivo é apenas um segmento de memória, que pode ser adicionado ou removido do espaço de endereçamento de um processo, diversas premissas que guiam o projeto de um sistema de arquivos não são mais

necessárias [7]. Nossa discussão não se aprofunda nos detalhes técnicos de como implementar tal sistema de arquivos, embora os projetistas do *MULTICS* tenham implementado a maioria destes conceitos.

2.4.5 Alocação de memória

Em um sistema com memória principal composta por *NVRAM* e *DRAM*, o gerenciador de memória virtual pode decidir em qual tipo de memória os segmentos de uma aplicação serão colocados, a partir de suas necessidades. Informações que devem estar sempre persistentes são alocadas diretamente na memória não volátil. Informações que podem ser temporárias, ou não podem ser persistidas por motivos de segurança, são alocadas diretamente na parte da memória principal que é volátil.

2.5 Proposta: *Buffer* de segmentos (SB) totalmente associativo

Propomos a utilização de segmentação de memória em conjunto com uma *cache* totalmente associativa que mantenha próximo ao processador os segmentos mais utilizados. Chamamos esta *cache* de *buffer* de segmentos (*segment buffer*, SB). Também propomos que o modelo de segmentação seja transparente para o programador. O gerenciamento dos segmentos lógicos é feito pelo ligador (*linker*) e a tabela de segmentos é carregada e mantida pelo sistema operacional. Para otimizar a utilização da memória o compilador pode calcular o tamanho estimado de cada segmento, ou permitir que o programador informe o seu tamanho, ou informe que o segmento é dinâmico e seu tamanho varia dependendo da execução do programa.

A tradução de um endereço virtual em um endereço físico deve ser feita o mais rapidamente possível. Com paginação sob demanda, a tradução de um endereço virtual em um endereços físico é feita rapidamente, com uma consulta simples na TLB.

A Figura 2.2 mostra um diagrama de blocos de uma SB proposta para a arquitetura *MIPS32*. O diagrama mostra a SB composta por: (a) etiqueta do endereço virtual – *(virtual) tag*; (b) permissões para o segmento – *status*; (c) endereço base do segmento – *(physical) base*; e (d) endereço limite do segmento – *(physical) limit*. O endereço virtual (*virtual_addr*) e dividido em identificador do segmento (*VSN*) e deslocamento (*displ*). Se o *VSN* for encontrado na etiqueta de uma entrada da SB temos um acerto, se não for encontrado temos uma falta na SB.

Com segmentação são três as operações necessárias para traduzir um endereço: (a) a soma da base (PSBase) do segmento com seu deslocamento (displ) para obter o endereço físico (PA); (b) a comparação do endereço físico (PA) com o limite do segmento (PSLimit), para validar se o endereço está dentro dos limites do segmento; e (c) a comparação do endereço físico (PA) com a base do segmento.

A primeira soma está no caminho crítico e precisa ser realizada o mais rapidamente possível. Podemos imaginar uma aproximação na qual o tempo médio de acesso à memória poderia ser alongado em 1-2% sem que isso acarrete uma diminuição significativa no desempenho, quando comparamos com um sistema com paginação. Podemos afirmar isso porque o nossos resultados indicam que um SB sofre de 10 a 50 vezes menos faltas que uma TLB de mesma capacidade. Hornyack *et all.* realizaram medições na quantidade de ciclos gastos resolvendo faltas de páginas e concluiram que o impacto é de alguns pontos percentuais, mas que podem chegar até 58% do total de ciclos para algumas cargas de trabalho [28].

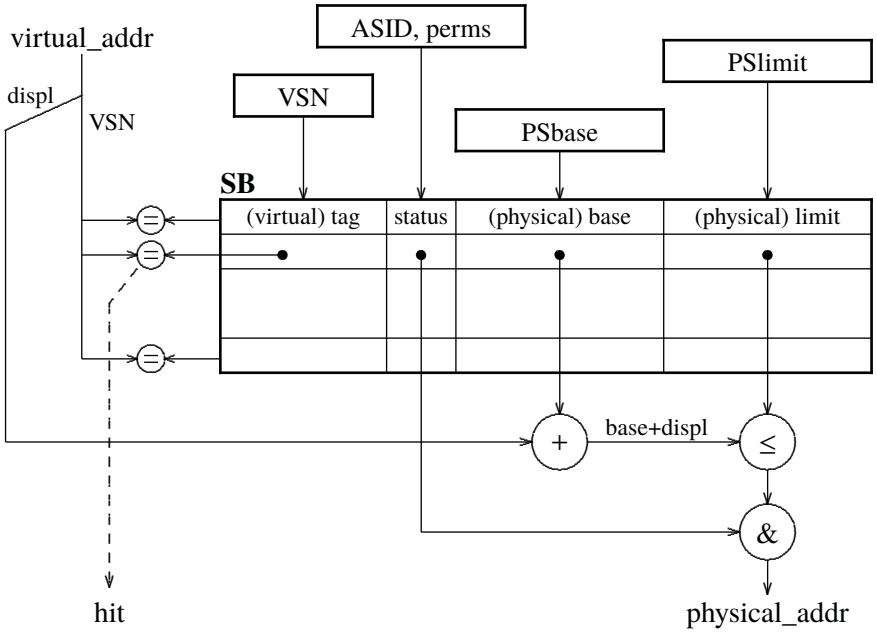

Figura 2.2: Diagrama de blocos de um *Segment Buffer*.

Com segmentação o trafego de dados entre a memória e o processador é reduzido, porque a quantidade de faltas é de 10 a 50 vezes menores que a paginação. A tabela de segmentos também é significativamente menor que a tabela de páginas e seu gerenciamento é mais rápido e simples. Esses fatores podem compensar a eventual demora que a soma no caminho crítico da tradução de endereços adiciona ao sistema com segmentação.

A segunda soma pode estar fora do caminho crítico porque, se houver uma exceção, ela somente precisa de ser sinalizada quando a instrução for finalizada (*commit*), e isso somente ocorre ao final do *pipeline*. O mesmo se aplica a uma exceção relacionada a permissões do segmento. Uma terceira soma precisa ser realizada para que o modelo de programação siga inalterado para segmentação. É necessário verificar se o PA é menor que a base do segmento. Isso pode ocorrer, por exemplo, se um erro ou um programa malicioso gerar um endereço virtual (VA) negativo, que então é adicionado à base do segmento e resulta em um PA menor que a base do segmento, criando uma situação de exceção. Essa comparação, que não é mostrada na Figura 2.2, também pode ser feita no final do *pipeline* e mantida fora do caminho crítico.

Fragmentação externa é um problema em sistemas com segmentação. Estamos supondo que os sistemas computacionais terão memória abundante e estarão dispostos a sacrificar espaço em troca de um tempo menor de execução. Mesmo assim, com o tempo, a tendência é a memória ficar fragmentada. Uma sugestão para minimizar a fragmentação é criar uma mecanismo para desfragmentar a memória, movendo e otimizando os segmentos em momentos em que o sistema esteja ocioso [32]. Esse mecanismo é semelhante a um controlador de acesso direto à memória.

Está fora do escopo deste trabalho solucionar os problemas relacionados ao somador no caminho crítico, fragmentação externa e a arquitetura completa de um processador com suporte a um *buffer* de segmentos. Os resultados apresentados apontam que é importante explorar as possibilidades de ganho de desempenho utilizando segmentação.

Iremos realizar simulações do *buffer* de segmentos proposto com diversas configurações de tamanho, e comparar o resultado com o desempenho da uma TLB de mesma capacidade.

Capítulo 3

Gerenciamento de memória virtual

3.1 Segmentação com paginação no *MULTICS*

O *MULTICS* (*Multiplexed Information and Computing Service*) é um sistema operacional desenvolvido para o *mainframe* GE-645 pelo MIT (*Massachusetts Institute of Technology*), GE (*General Electric*) e Bell Labs.

O sistema operacional *MULTICS* influenciou o projetos do *UNIX* desenvolvido por Ken Thompson e Dennis Ritchie com o sistema de arquivos hierárquico e o interpretador de comandos (*shell*). O *MULTICS* utilizou outros conceitos bastante avançados para a época como *dynamic linking*, suporte para diversos processadores, re-configuração *online* do *hardware* e gerenciamento de memória de nível único (*single level store*). O gerenciamento de memória virtual do *MULTICS* consiste em um número de segmento, de 18 *bits* (EL), e um deslocamento, de 16 *bits* (PS), criando um espaço de endereçamento de 34 *bits*.

Os projetistas do *MULTICS* encontraram dois problemas em potencial: (a) com segmentos de 64KiB de palavras, cada palavra contendo 36 *bits*, o tamanho médio dos segmentos no *MULTICS* iria causar fragmentação externa; e (b) tempo de pesquisa para alocação de um segmento, utilizando um algoritmo de melhor encaixe, também seria muito longo [1].

Os projetistas do sistema precisavam decidir como resolver o desperdício de espaço devido à fragmentação externa e o tempo excessivo gasto em encontrar o melhor encaixe para alocar um novo segmento. A solução adotada foi utilizar o mecanismo de paginação para cada segmento. A paginação elimina a fragmentação externa e resolve o problema do tempo gasto em alocação de memória. Cada página no *MULTICS* tem 1KiB palavras. A posição no segmento de 16 *bits* (PS) é formado pelo número da página com 6 *bits* (NPG) e o deslocamento na página de 10 *bits* (DSP).

A diferença entre o mecanismo de segmentação puro e o mecanismo de gerenciamento de memória virtual do *MULTICS* é que no *MULTICS* a entrada da tabela de segmentos não contém mais o endereço base do segmento e sim o endereço base para a tabela de páginas deste segmento. existe uma tabela de páginas separada para cada segmento, e cada tabela de páginas possui apenas as entradas necessárias àquele segmento [1].

O endereço lógico (EL) é um valor representado em 18 *bits*, identificando até 262.144 segmentos e demandando uma tabela de segmentos grande para o hardware da época. Para mitigar esse problema o *MULTICS* utiliza paginação na tabela de segmentos. O número do segmento (EL) de 18 *bits* é dividido em um número de página (ELNP) de 8 *bits* e um deslocamento na página (ELDSP) de 10 *bits*. Desta forma a tabela de segmentos é representada por uma tabela de páginas que tem até 2^8 entradas.

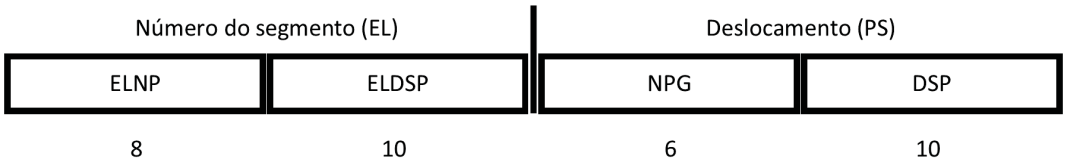

Figura 3.1: Endereço de memória do *MULTICS*.

Um endereço lógico no *MULTICS* pode ser visto na Figura 3.1, onde ELNP é um índice na tabela de páginas para a tabela de segmentos, e ELDSP é a posição na página da tabela de segmentos. Com ELNP e ELDSP encontramos a página que contém o segmento desejado. Para encontrar a palavra da memória é usado o NPG como índice na tabela de páginas do segmento desejado e DSP é a posição na página que contém a palavra da memória procurada. Os passos necessários para encontrar o endereço físico a partir de um endereço virtual no *MULTICS* estão representados na Figura 3.2.

O *MULTICS* utiliza uma pequena TLB para guardar os mapeamos mais recentes e evitar constantes consultas à memória *RAM*. São usados 16 registradores associativos contendo os endereços das 16 páginas usadas mais recentemente. O valor contido em cada registrador é composto por uma chave e um número de bloco. A chave é um valor de 24 *bits* formado por um número de segmento seguido por um número de página.

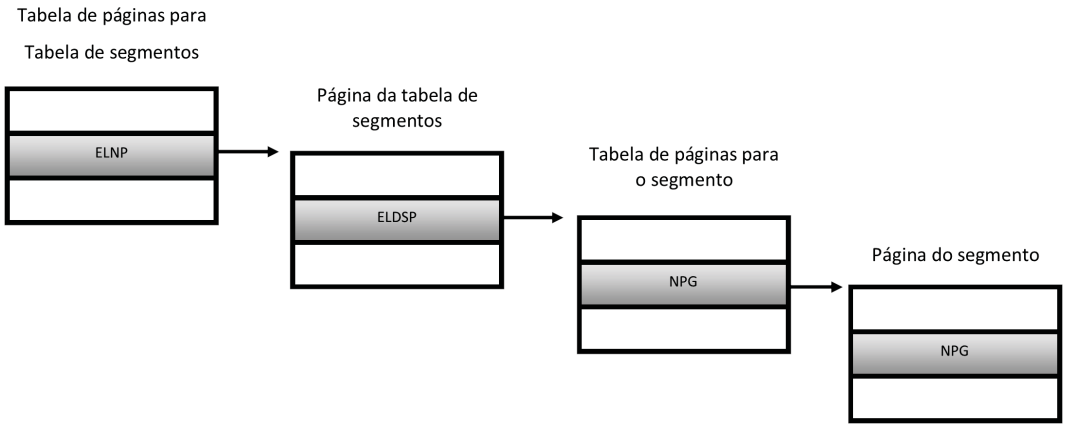

Figura 3.2: Tradução de endereços no *MULTICS*.

O projeto do *MULTICS* utiliza segmentação paginada para otimizar o uso da escassa, e cara, memória *RAM* dos *mainframes* ddas décadas de 1960 a 1980. A motivação para do uso da segmentação é permitir o compartilhamento de informações de uma forma mais simples e automática do que em sistemas não segmentados. A paginação foi utilizada para: (a) simplificar a alocação de memória e evitar a tarefa de "compactar" informação na memória RAM (*core* na terminologia do *MULTICS*) e em disco; (b) como somente as páginas referenciadas de um segmento precisam estar em memória, o segmento em si pode ser maior que a memória física disponível; (c) paginação sob demanda explora a localidade espacial dos programas para manter em memória as páginas de segmentos que são necessárias naquele momento, em vez de manter todos os segmentos em memória; e (d) paginação sob demanda permite que o um programa que foi projetado para executar em uma máquina específica possa executar, com desempenho reduzido, em outra máquina com menos recursos [7].

Utilizamos alguns conceitos do *MULTICS* como ponto de partida para a proposta de segmentação utilizando um *buffer* de segmentos. Em sistemas modernos com memória

RAM abundante a paginação pode trazer um impacto significativo na performance de algumas aplicações que fazem uso intensivo de memória *RAM* [28]. As necessidades e restrições que levaram o *MULTICS* a ser projetado com paginação sob demanda para otimizar o uso de espaço na memória *RAM* não existem em servidores modernos com memória *RAM* abundante.

3.2 Segmentação no *Intel X86* em *protected mode*

Os processadores da família *Intel x86*, iniciando no modelo *386* até o modelo *Pentium IV* possuem um modo de gerenciamento de memória chamado "modo protegido" (*protected mode*), que pode ser ativado pelo sistema operacional utilizando o *bit* 31 (o *bit* mais significativo) do registrador de controle CR0 [11]. Neste texto descrevemos somente o *protected mode*.

Em *protected mode* o gerenciamento de memória do *x86* e do *MULTICS* são similares, e ambos empregam segmentação com paginação. Uma das diferenças mais significativas entre o *MULTICS* e o *x86* é que neste é possível desabilitar completamente a paginação, fazendo o processador trabalhar com segmentação pura, alterando o *bit* 31 (o *bit* mais significativo) do registrador de controle CR0 [11].

O *MULTICS* possui até 256KiB segmentos, cada segmentos com até 64KiB palavras de 36 *bits*. O *x86* possui 16KiB segmentos, cada segmento com até um bilhão de palavras de 32 *bits*. Apesar do processador da *Intel* suportar menos segmentos, o tamanho máximo de segmento é muito maior [1].

O espaço de endereçamento lógico de um processo no *x86* é dividido em duas partições: (a) a tabela de descrições locais (TDL); e (b) a tabela de descrições globais (TDG). A TDL contém 8KiB segmentos de uso privativo do processo e a TDG contém 8KiB segmentos de uso compartilhado entre todos os processos. Cada entrada nas tabelas TDL e TDG tem 8 *bytes* e contém informações detalhadas sobre um determinado segmento, incluindo o endereço inicial e o seu tamanho. O número máximo de segmentos de um processo é 16KiB e cada segmento pode ter até 4GiB com páginas de 4KiB. A TDL contém segmentos do programa como código, dados e pilha e a GTD contém segmentos do sistema, bibliotecas compartilhadas e o próprio sistema operacional.

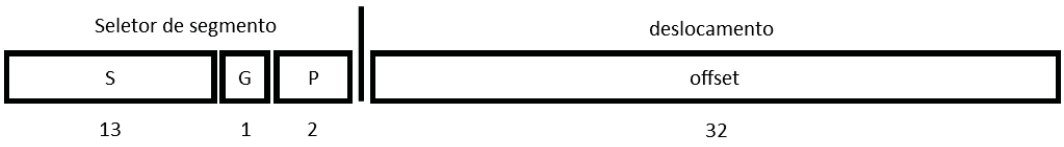

Figura 3.3: Endereço virtual no *x86* em *protected mode*.

Um endereço virtual no *x86* é composto por um par (seletor, deslocamento) como mostra a figura 3.3. O seletor é um número de 16 *bits* composto por três partes: (a) um seletor de segmento *S* de 16 *bits*; (b) um *bit G* que indica se o segmento está na TDL ou na TDG; e (c) dois *bits P* com informações relativas a proteção. O deslocamento é a posição referenciada (*offset*) dentro do segmento.

O seletor do segmento é utilizado para encontrar o descritor de segmento na TDL ou na TDG. O descritor de segmento é um número de 64 bits (8 *bytes*) que contém o endereço base do segmento, tamanho e outras informações. A arquitetura *x86* contém seis registradores dedicados à segmentação: (a) os registradores CS, DS, SS e ES apontam para o segmento de código do programa em execução; (b) o registrador DS aponta para o segmento de dados do programa em execução; (c) o registrador SS aponta para a pilha (*stack*) do programa em execução;

e (d) o registrador ES aponta para um segmento definido pelo programador. Para acessar algum segmento o programa carrega um seletor de segmento em um dos registradores de segmentos.

No momento em que um seletor de segmento é carregado para um registrador, o descritor correspondente é carregado da TDL ou da TDG para um registrador de micro-programação (*microprogram register*), o seletor e o descritor do segmento podem ser acessados rapidamente pelo processador. O formato do seletor foi concebido para que o descritor relacionado seja encontrado facilmente.

Com base no *bit* 2 do seletor a TDL ou a TDG é selecionada. O seletor é então copiado para um registrador interno, onde os 3 *bits* menos significativos são preenchidos com zeros, e o endereço da TDL ou da TDG é somado ao valor do seletor. O resultado é uma referência direta ao descritor de segmento desejado. Localizado o descritor do segmento, o endereço base do segmento é somado ao deslocamento para calcular o endereço linear, como mostra a Figura 3.4. Se a soma resultar em um número maior que o limite do segmento, uma exceção é gerada e o controle é passado ao sistema operacional.

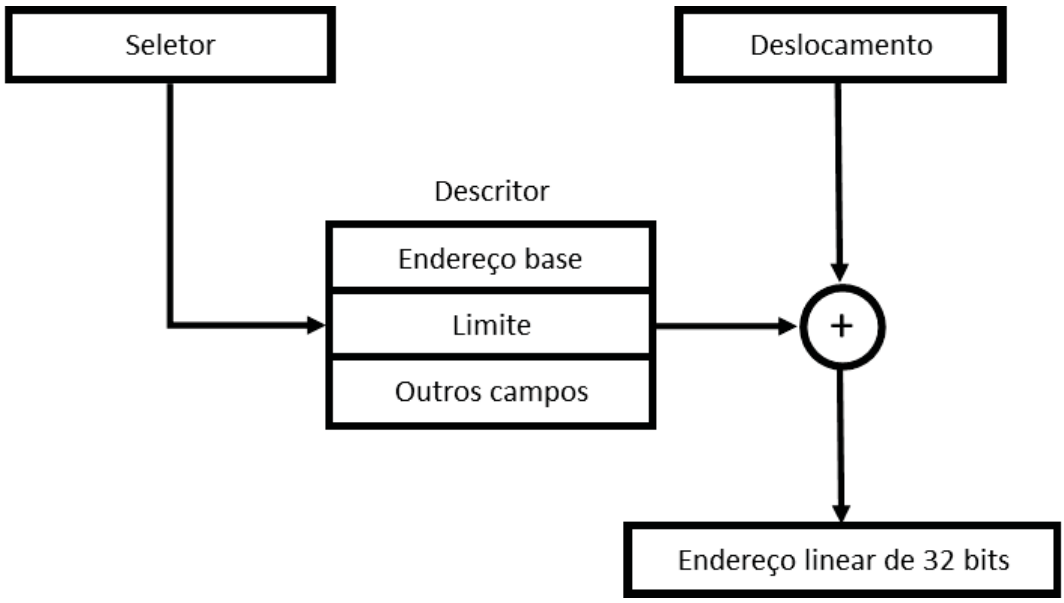

Figura 3.4: Conversão de um par (seletor, deslocamento) em um endereço linear no *Intel x86*.

Se a paginação estiver desabilitada o endereço linear é interpretado como endereço físico e enviado à unidade de memória para executar uma leitura ou escrita. Não existe nenhum mecanismo em hardware que impeça que um segmento invada o espaço de memória de outro segmento, ficando a cargo do sistema operacional prover esse controle.

Se a paginação estiver habilitada o endereço linear é considerado como um endereço virtual, e mapeado em um endereço físico utilizando uma tabela de páginas. Como o endereço virtual tem 32 *bits* e cada página tem 4KiB de tamanho, um segmento pode conter um milhão de páginas. Para reduzir o tamanho da tabela de páginas é utilizando um mapeamento em dois níveis. Cada entrada no diretório de páginas aponta para uma tabela de páginas contendo 1024 entradas de 32, *bits* onde cada entrada aponta para um *page frame* como mostra a Figura 3.5.

A Figura 3.5 também mostra o endereço linear divido em três partes: (a) índice do diretório de páginas (dir); (b) índice da tabela de páginas (page); e (c) posição (*offset*). Cada programa possui um diretório de páginas (*page directory*) com 1024 entradas de 32 *bits*. O diretório de páginas está em um endereço apontado pelo valor do registrador global CR3 e o campo *dir* é usado para indexar o diretório de páginas e encontrar o endereço da tabela de páginas. O campo *page* é usado para indexar a tabela de páginas e encontrar o endereço para o *page frame*

procurado. Finalmente, o *offset* é somado ao endereço do *page frame* para produzir o endereço físico [11].

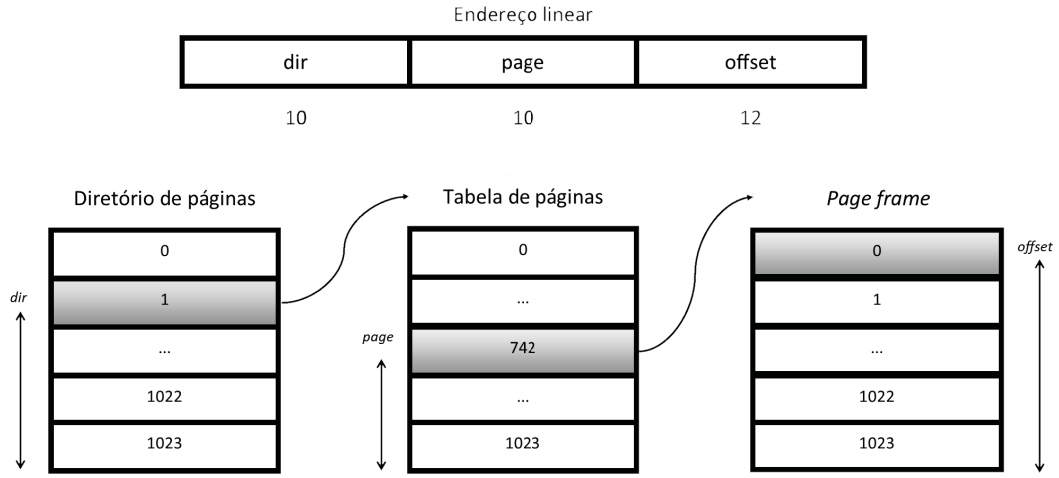

Figura 3.5: Mapeamento de endereço linear em endereço físico.

Cada tabela de páginas tem 1024 4KiB páginas. Uma única tabela de páginas ocupa 4MiB de memória. Um segmento menor que 4MiB terá um diretório de páginas com apenas uma entrada apontando para a única tabela de páginas necessária para o segmento. Para evitar referências constantes à memória para realizar a tradução de endereços, o x86 possui uma TLB que guarda os mapeamentos mais recentes (*dir*, *pages*) diretamente para o endereço físico do *page frame*.

Em alguns processadores x86 de 32 *bits* é possível endereçar mais que 4GiB de memória utilizando uma funcionalidade chamada *Physical Address Extensions* (PAE), que define uma hierarquia para a tabela de páginas com três níveis em vez dos dois níveis disponível em *protected mode*. A entradas da tabela de páginas são de 64 *bits* e possibilitam o endereçamento de mais de 4GiB de memória física. Embora o tamanho das tabelas de página não mude. O nível adicional introduzido com o PAE na tabela de página se chama *Page Directory Pointer Table* e contém 4 entradas com ponteiros para 4 diretórios de páginas. O *hardware* e o sistema operacional precisam suportar a funcionalidade PAE para endereçar mais de 4GiB de memória *RAM* [11].

A arquitetura *x86* não possui um *buffer* de segmentos e depende de um conjunto de registradores pré-definido para suportar paginação pura. A performance da tradução de endereços é prejudicada pelo número limitado de segmentos que podem estar referenciados e próximos do processador, para serem utilizados na tradução de endereços virtuais em endereços físicos. Outras limitações são o tamanho dos registradores (32 *bits*), o modo de endereçamento complexo para manter a compatibilidade com versões mais antigas da arquitetura e a limitação de endereçamento de apenas 4GiB de memória, supondo que as extensões de endereçamento presentes em alguns processadores x86 não estejam presentes.

Ganhos de desempenho utilizando segmentação para programas que utilizam muita memória *RAM* necessitam de um modo de endereçamento mais simples e um *buffer* de segmentos grande o suficiente para guardar próximo ao processador os segmentos mais utilizados, e garantir que a tradução de endereços ocorra sem a necessidade constante de acessar a memória *RAM* para buscar informações sobre os segmentos.

3.3 Paginação em processadores *Intel x86-64*

A arquitetura *x86* foi estendida para 64 *bits* pela *AMD* em meados do ano 2000. Essa mudança introduziu novos registradores de 64 *bits* e removeu o suporte à segmentação. A nova arquitetura, chamada inicialmente de AMD64 pela AMD e depois chamada de *x86-64* pela *Intel* utiliza um novo modo de funcionamento chamado *long mode*.

Em processadores x86-64 executando em *long mode* nativo, a tradução de endereços utiliza uma extensão da PAE, adicionando um quarto nível à tabela de páginas, além de aumentar o *Page Directory Pointer Table* de 4 para 512 entradas. Atualmente 48 *bits* do endereço virtual são traduzidos em um espaço de endereçamento virtual de até 256TiB.

3.4 Alcance da TLB e páginas maiores (*huge pages*)

O número de entradas da TLB, multiplicado pelo tamanho da página, resulta no alcance da TLB (*TLB reach*). O alcance da TLB é crítico para o desempenho de uma aplicação. Se o alcance da TLB não comporta a quantidade de dados referenciados por um processo ele pode passar uma parte significativa do tempo de execução resolvendo faltas na TLB. A carga de trabalho pode ser modificada para minimizar a quantidade de faltas e se adequar ao alcance da TLB, mas isso nem sempre é possível, prático ou fácil de implementar. Uma forma de aumentar o alcance da TLB é aumentar o tamanho da página. Processadores como o MIPS R10000 (Mips94), Ultrasparc II (Sparc97) e o PA8000 (PA-RISC) possuem essa funcionalidade implementando entradas na TLB com tamanho configurável, de modo que o alcance da TLB possa ser alterado para satisfazer as necessidades da carga de trabalho de um processo específico [25].

Páginas maiores que o tamanho padrão são chamadas de *super pages*, *large pages* ou *huge pages* dependendo do sistema operacional utilizado. A arquitetura *intel* x86-64 executando em *long mode* possui suporte para páginas com 4KiB, 2MiB e 1GiB. A plataforma ARM v7, popular em dispositivos móveis, possui suporte para páginas com tamanho 4KiB, 64KiB, 1MiB (chamada *section*) e 16MiB (chamada *supersection*) [22].

Para se beneficiar do suporte à páginas maiores, o sistema operacional deve prover suporte à páginas com tamanhos diferentes. Isso implica em diversos desafios relacionados a forma como o sistema operacional aloca memória, controla a proteção e compartilhamento de memória e impacta na paginação em disco [25]. Os sistemas operacionais recentes possuem suporte a páginas com tamanhos diferentes. O *Linux* possui suporte a *huge pages* desde a versão 2.6.38. Na plataforma *Windows* o suporte à *large pages* está presente na família de produtos *Windows Server* desde a versão 2003 (SP1 e superiores) e na família de sistema operacionais para *desktop* desde o Windows Vista. O sistema operacional *Solaris* versão 9 e o sistema *FreeBSD 7.2* também oferecem suporte à *huge pages* [22].

Ganhos em desempenho utilizando páginas maiores estão relacionados ao tipo da aplicação, sua carga de trabalho, localidade espacial e padrão de uso. Em algumas situações, a quantidade de faltas na TLB pode ser reduzida em até 98%. O ganho varia de aplicação para aplicação, e nem sempre utilizar o maior tamanho de página disponível é a melhor opção para reduzir o impacto de faltas na TLB [25]. Para extrair o máximo de desempenho de páginas de diferentes tamanhos a aplicação deve ser escrita de forma a explorar esse recurso, avisando o sistema operacional das suas necessidades de memória para que o melhor tamanho de página seja escolhido.

Aumentar o tamanho das páginas pode piorar a fragmentação interna. Muitas aplicações criam vários mapeamentos contiguos pequenos, o que limita os benefícios de páginas maiores. Um aumento de tamanho de página para 16KiB ou 32KiB é frequentemente razoável, mas

páginas maiores produzem muita fragmentação interna. O número de mapeamentos necessários entre a memória virtual e física também não é reduzido significativamente, mesmo que o sistema operacional seja capaz de alocar o maior tamanho de página possível para cada necessidade [28].

3.5 Espaço único de endereçamento (*single address space*)

Em sistemas com espaço único de endereçamento (*single address space*), todas as aplicações compartilham um espaço de endereçamento virtual único. Proteção é implementada utilizando *protection domains*, que definem quais páginas do espaço de endereçamento virtual global um processo pode referenciar. Sistemas *single address space* possuem duas grandes vantagens: (a) estimulam o compartilhamento de informações entre *protection domains*; e (b) como eles definem um mapeamento único entre endereços virtuais e físicos, a tradução de endereço pode ser removida do caminho crítico do processador [31].

Sistemas com espaço único de endereçamento são uma alternativa à segmentação utilizando um *buffer* de segmentos. Se a memória física for grande o suficiente, a tradução de endereços pode ser tornar desnecessária. Uma estrutura similar a um SB ou uma TLB deve existir, para implementar segurança e proteção entre processos.

3.6 Paginação com parte do espaço de endereçamento virtual mapeado para um segmento

Analisando o desempenho de servidores executando aplicações que utilizam muita memória *RAM*, como servidores de banco de dados, análise de grafos e aplicações que utilizam uma grande quantidade de dados em *caches* em memória, foi observado um impacto de até 10% dos ciclos de execução gerenciando faltas na TLB. Mesmo utilizando páginas maiores, há impacto no desempenho por conta de faltas na TLB. Basu *et.all* propõem mapear parte do espaço de endereçamento virtual linear do processo com um segmento direto (*direct segment*), e manter o restante do espaço de endereçamento utilizando paginação sob demanda [6].

Segmentos diretos utilizam pouco hardware adicional. Alguns registradores para armazenar o endereço base, o limite do segmento e o offset, o suficiente para mapear regiões contíguas de memória virtual em regiões contíguas de memória física. Segmentos diretos removem a penalidade de faltas na TLB para estruturas de dados importantes como *buffers* de banco de dados e tabelas de dados com pares (chave, valor). O espaço de endereçamento virtual do segmento direto pode ser convertido para paginação quando necessário.

Basu *et.all* propõem a criação do espaço de memória virtual contíguo gerenciado com um segmento direto durante o inicio da execução da aplicação, chamando esse espaço de região primária (*primary region*) [6]. O gerenciamento da região primária é responsabilidade do sistema operacional, que deve avaliar se existe uma região contígua em memória para alocar o segmento. Se não houver uma região contígua em memória devido à fragmentação de memória, o espaço é mapeado com paginação. As aplicações podem avisar o sistema operacional que uma região primária é necessária, ou o próprio sistema operacional através da observação do padrão de uso da memória, define automaticamente a região primária.

O uso de segmentação pura para gerenciar memória virtual é mais intuitivo e pode ser abstraído para que, do ponto de vista do programador, não existam diferenças significativas para uma sistema com paginação sob demanda. Do ponto de vista da aplicação, utilizar um *buffer* de

segmentos para acelerar a tradução de endereços é mais simples e flexível que utilizar um único segmento diretamente mapeado por processo.

3.7 *NVRAM* e memória virtual

Avanços recentes tornaram a tecnologia de memória RAM não volátil (*NVRAM*) uma realidade. A perspectiva de uma memória primária persistente, rápida e abundante tem o potencial de mudar a forma como projetamos *hardware* e *software*. Muitas das estratégias que utilizamos atualmente para gerenciar a memória primária de nossas máquinas, como na paginação sob demanda, não fazem mais sentido se ela for abundante e persistente. Diante dessas inovações precisamos de uma nova forma de pensar o projeto de *hardware* e *software* para o futuro [5].

Por mais de cinquenta anos, as premissas para o projeto de sistemas operacionais foram: (a) memória principal rápida, volátil e limitada; e (b) memória secundária, com discos magnéticos, lentos e persistentes. Com a possibilidade de uso de *NVRAM*, essas premissas não são mais verdadeiras e diversos componentes do sistema operacional podem evoluir para uma implementação mais simples e eficiente.

A pesquisa de sistemas de arquivo específicos para *NVRAM* tem recebido bastante atenção. Existem sistemas de arquivo como *BPFS* [5] [4] que são endereçados em *bytes* e não em setores e blocos. O modo de endereçamento direto é mais eficiente que sistemas de arquivos tradicionais quando utilizados em conjunto com *NVRAM*, mas o projeto do *BPFS* ainda leva em conta dois níveis de memória.

Sistemas de arquivos foram criados sob a premissa de que a memória secundária é formada por discos magnéticos lentos, pouco confiáveis e endereçados em setores e blocos. Para otimizar os sistemas de arquivo utiliza-se estruturas de indexação para agilizar o acesso aos dados, e *journaling* para manter os dados íntegros. Sistemas de arquivos tradicionais não foram projetados para dispositivos com baixa latência.

Enquanto os discos e a memória *flash* possuem latência suficientemente alta para esconder o *overhead* de software, a baixa latência da *NVRAM*, torna esses custos contraprodutivos [4]. Oikawa [33] mostra que a performance da *NVRAM* é comparável a da *DRAM*, mas as estruturas de dados utilizadas no sistema de arquivo impactam negativamente o desempenho da *NVRAM*. Ele também demonstra que estruturas de dados eficientes em discos tradicionais nem sempre são eficientes em memória não volátil.

A *Intel* propõe um modelo de utilização de memória não volátil, chamada de *NVM*, utilizando a tecnologia *3D XPoint*. Com *NVM* os programas possuem uma nova camada disponível para armazenar seus dados além da memória *RAM* e armazenamento em disco. Essa camada adicional se conectada diretamente ao barramento de memória. A *NVM* oferece maior capacidade do que *DRAM* e desempenho significativamente superior ao armazenamento em disco. As aplicações podem acessar estruturas de dados persistentes residente na memória *NVM*, como ocorre com a memória tradicional, eliminando a necessidade da troca de blocos de dados entre memória e disco. Para obter este acesso direto a *NVM* com baixa latência, é utilizada uma interface de programação de aplicações (*Application Programing Interface, API*) que permite que os programas interajam com a memória persistente [37].

Arquivo mapeado em memória é um recurso que existe nos sistemas operacionais modernos há muito tempo. Para *NVM*, a *API* para mapear arquivos em memória disponíveis no sistema operacional é o cerne do modelo de programação para memória persistente, publicado pela *Storage Networking Industry Association* (*SNIA*) [3]. Nesse modelo, para utilizar a *NVM* a *Intel* disponibiliza para o programador uma *API* chamada *NVML* (*NVM Libraries*), utilizando as

funções presentes na *NVML* é possível mapear um arquivo para a memória não volátil, onde os dados podem ser acessados com endereçamento em *bytes* (*byte level access*).

Uma vez que um arquivo é mapeado para a *NVM*, o programa tem acesso direto ao espaço de endereçamento virtual em que o arquivo está. Mapeamentos diretos para a memória persistente é um recurso conhecido como acesso direto (*DAX*). O suporte para este recurso é o que diferencia um sistema de arquivos normal de um sistema de arquivos com memória persistente. O *DAX* é suportado hoje pelo Windows e Linux [37].

Supondo que a memória principal é composto em sua totalidade por *NVRAM* e infinita (2^{64}), a arquitetura do computador precisa ser revista. O conceito de arquivo em disco não é mais aplicável aos dados de um processo ou de um usuário específico. Em um cenário como esse a paginação sob demanda seria responsável apenas por implementar a segurança e proteção entre processos. O impacto em desempenho da tradução de endereços para uma memória imensa será proporcional.

A segmentação pura pode ser uma saída para diminuir o número de mapeamentos necessários enquanto provê segurança entre processos, e ainda para representar arquivos como segmentos de forma similar ao *MULTICS* [7].

Capítulo 4

Paginação *versus* segmentação

4.1 Simulação de memória virtual

Em um futuro breve teremos memória *RAM* com capacidade de endereçamento de 2^{64} *bits* nos atuais processadores de 64 *bits*, tornado o tamanho da *RAM* praticamente "infinito". A evolução das tecnologias relacionadas à memória *RAM* não volátil sugerem que a capacidade e desempenho da *NVRAM* serão semelhantes aos da *DRAM*, também em um futuro próximo [4] [5] [37]. Nessas condições o modelo de gerênciamento de memória com segmentação pode ser mais eficiente que o modelo de gerenciamento de memória com paginação [28] [6]. Nesse trabalho avaliamos o gerenciamento de memória com paginação e segmentação, utilizando traços de programas reais executados em um ambiente *Linux*, em uma máquina *Intel x86-64*.

Os traços são gerados utilizando o programa *Valgrind*, que realiza a instrumentação em tempo real do programa nativo, intercepta as instruções, e gera instruções simplificadas que são equivalentes ao programa original. As instruções geradas pelo *Valgrind* contêm apenas referências ao endereço da instrução sendo executada (*InstructionAddress*) e os endereços de dados que a instrução utiliza (*DataAddress)*.

Simulamos a *Translation Lookaside Buffer* (*TLB*) e um *buffer* de segmentos (*Segment Buffer, SB*). A simulador conta ao número de faltas na *TLB* e no *SB* para o mesmo conjunto de traços de execução. Faltas na *TLB* ou no *SB* ocorrem quando um endereço virtual não é encontrado no respectivo *buffer*. Faltas de páginas ou faltas de segmento fazem com que descritor de página, ou de segmento, seja carregado da memória *RAM*. A memória *DRAM* é duas ordens de magnitude mais lenta que acessos a *TLB* ou *SB*. Comparando a quantidade de faltas na *TLB* e no *SB* podemos avaliar a eficiência da tradução de endereços com os dois modelos de gerenciamento.

Avaliamos a quantidade de faltas utilizando páginas de tamanho 4KiB porque os testes foram realizados em um processador *Intel x86-64*. Três diferentes configurações de *TLB* foram utilizadas: (a) *TLB* com 32 posições, totalmente associativa (TLB32); (b) *TLB* com 64 posições, totalmente associativa (TLB64); e (c) *TLB* com 128 posições, totalmente associativo (TLB128). A politica de reposição no *buffer* é *Least Recently Used (LRU)* perfeito.

O modelo de gerenciamento de memória com paginação emprega espaço de endereçamento bidimensional (PV + deslocamento na página) com os *bits* do endereço apontando para uma página virtual. O modelo de gerenciamento de memória com segmentação também possui um espaço de endereçamento em duas dimensões: (a) um identificador de segmento (*SegmentID*); e (b) um deslocamento dentro do segmento (*displacement*).

Para obter os segmentos ativos do processo utilizamos o utilitário *pmap* para gerar o mapa de memória do processo durante a simulação. O mapa de memória mostra a faixa de endereços virtuais que as diversas partes do processo ocupam, as bibliotecas dinamicamente

alocadas, pilha, etc. Com o mapa de memória, transformamos o endereço de memória linear dos traços em um endereço composto por um identificador de segmento (*SegmentID*) e um deslocamento (*displacement*).

Com o endereço composto pela tupla (*SegmentID*, *displacement*) é possível simular uma memória segmentada e contar a quantidade de faltas no *buffer* de segmentos. As configurações utilizadas para o *SB* foram as mesmas utilizadas para a *TLB*: (a) *SB* com 32 posições, totalmente associativo (SB32); (b) *SB* com 64 posições (SB64), totalmente associativo e; (c) *SB* com 128 posições (SB128), totalmente associativo. A politica de reposição no *buffer* também é *Least Recently Used (LRU)* perfeito.

Nesta seção são mostrados com detalhes: (a) o processo de geração dos traços; (b) como a obtemos o mapa de memória dos processos, e construímos a tabela de segmentos; (c) a simulação de memória virtual com paginação; (d) como foram gerados os endereços para simulação de memória virtual com segmentação; (e) a simulação com segmentação; e (f) o funcionamento do simulador *jBluePill*.

4.2 Traços de execução

Traços (*Traces)* ou traçados são arquivos no formato texto que contêm a sequência de instruções executadas pelo processador para um determinado programa. Os traços podem conter a execução de todas as instruções da arquitetura (*ISA*), ou um sub-conjunto dela que seja relevante para os testes desejados.

Foram utilizados traços contendo uma simplificação do conjunto de instruções do *ISA Intel x86-64*, somente com as instruções que fazem referência à memória. Os traços gerados contém endereços de código (instrução executada) e endereços de leitura e escrita de dados. Esses dados contêm toda a informação necessária para avaliar o desempenho de uma *cache* ou *TLB*.

4.3 Valgrind

Para gerar os traços foi utilizado o programa *Valgrind*, versão 3.13, com o módulo *Lackey*. O *Valgrind* é um *framework* para construção de utilitários que realizam instrumentação de código em tempo real. A ferramenta possui módulos para detecção de *bugs* de acesso a memória (vazamento de memória por exemplo) e análise de problemas em aplicações que fazem uso de programação paralela utilizando *threads* (como *dealocks*). O *Valgrind* também possibilita medições de uso de memória, padrões de uso de instruções, complexidade, e análise de desempenho (*profiling*) do programa que está sendo analisado.

A instrumentação de código realizada pelo *Valgrind* consiste em adicionar instruções ao programa, em tempo de execução, com o objetivo de extrair informações para serem utilizadas em medições e detecção de problemas. Qualquer binário pode ser instrumentado com o *Valgrind* sem a necessidade de recompilar o código fonte ou incluir bibliotecas adicionais. Utilizamos essa funcionalidade em conjunto com o módulo *Lackey* para gerar as sequências de referências de memória dos programas que utilizamos.

Para realizar a instrumentação, o *Valgrind* traduz sequências de código binário nativo para uma representação intermediária chamada *VEX*, que é similar a um conjunto de instruções *RISC*. Em uma plataforma *CISC*, como a plataforma *Intel x86* e *Intel x86-64*, existem instruções complexas tais como a instrução "*movsw %edi, %esi*" que move uma palavra do local apontado pelo segundo parâmetro para o endereço de memória apontado pelo primeiro parâmetro. Quando

o *Valgrind* avalia a instrução *movsw* ele a traduz para a tripla: (a) instrução (*instr*); (b) *load %esi*; e (c) *store %edi* .

Os traços gerados pelo módulo *Lackey* refletem esse conjunto de instruções *VEX* e possuem quatro tipos de referência à memória: (a) referência ao endereço de memória de instruções (*instr*); (b) referencia a uma leitura de um endereço de memória de dados (*load*); (c) referência a uma escrita em um endereço de memória de escrita (*store*); (d) referência a um endereço de memória modificado – um *load* e um *store* realizados no mesmo endereço. A Tabela 4.1 mostra vários exemplos de tradução de instruções nativas *Intel x86-64* para *VEX*.

Tabela 4.1: Tradução de instruções x86 para Valgrind VEX.

Instrução	**Acesso a memória**	**Sequência de eventos**
add %eax, %ebx	No loads or stores	instr
movl (%eax), %ebx	loads (%eax)	instr, load
movl %eax, (%ebx)	stores (%ebx)	instr, store
incl (%ecx)	modifies (%ecx)	instr, modify
cmpsb	loads (%esi), loads(%edi)	instr, load, load
call*l (%edx)	loads (%edx), stores -4(%esp)	instr, load, store
pushl (%edx)	loads (%edx), stores -4(%esp)	instr, load, store
movsw	loads (%esi), stores (%edi)	instr, load, store

4.4 Geração de traços

O procedimento para gerar traços de execução de um programa envolve três programas e vários arquivos de configuração e dados. A Figura 4.1 mostra os três processos interagindo: (a) o processo instrumentado pelo *Valgrind* que gera o fluxo de instruções em um *socket*; (b) um script *Python* que acompanha a execução do programa instrumentado, monitorando seu mapa de memória e salvando essa informação em um arquivo; e (c) uma ferramenta de coleta de traços (*Trace Collector*) escrita em *Java*, que recebe o fluxo de instruções do programa instrumentado, mensagens de erro, *warnings* e meta-dados e gerencia o armazenamento dessas informações.

O procedimento de coleta de traços inicia com a execução do *Trace Collector*, que é o processo responsável por gerir todas as informações relacionadas à execução do programa instrumentado e por receber o seu fluxo de instruções. O *Trace Collector* recebe como parâmetros: (a) arquivo *XML* que armazena informações sobre a execução; (b) o nome do arquivo de traços; (c) a quantidade máxima de instruções que o arquivo de traços pode conter; (d) opção para compactar os arquivos de traços; e (e) a forma de leitura do fluxo de instruções: arquivo, entrada padrão ou *socket*.

A forma de leitura do fluxo de instruções pelo *Trace Collector* que apresentou melhores resultados foi utilizando *sockets*. Dentre as vantagens observadas destaca-se que a saída padrão do processo instrumentado não é alterada. Outra vantagem é que a programação necessária para consumir o fluxo de instruções pelo *Trace Collector* é mais eficiente pois pode-se utilizar classes prontas para utilização de *sockets* e *threads* da biblioteca padrão do *Java*. Todos os traços coletados para a simulação foram coletados utilizando a comunicação entre o *Valgrind* e o *Trace Collector* utilizando *sockets*.

O *Trace Collector* está configurado por padrão para ler 10 bilhões de instruções *VEX* e então parar a captura. O número de 10 bilhões de instruções foi escolhido porque: (a) os traços

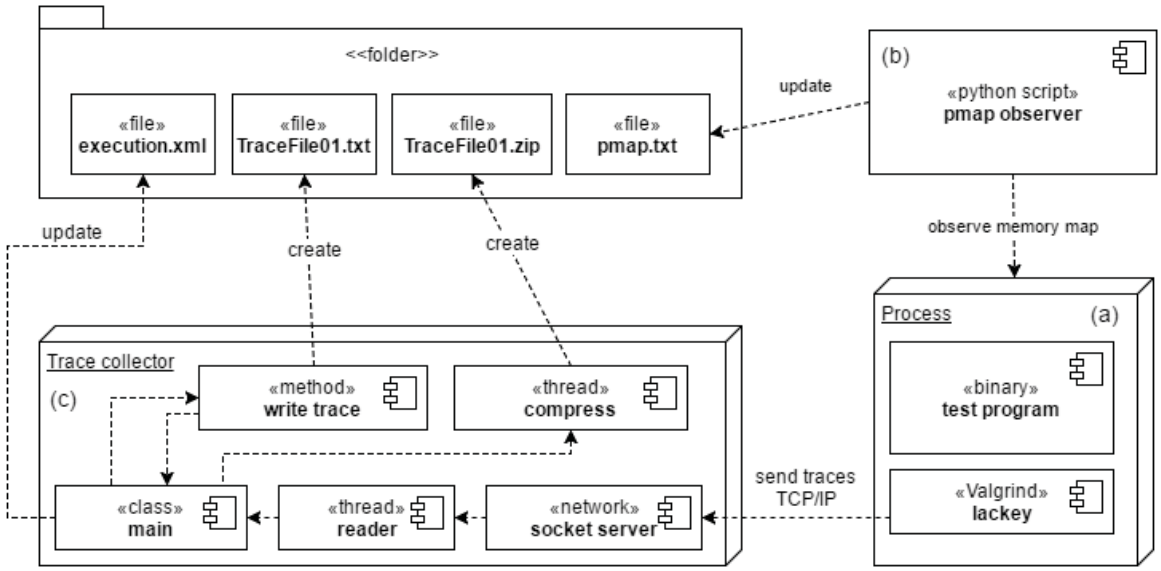

Figura 4.1: Sistema para gerar traços de execução.

de um programa são gerados em cerca de 10 horas, e o espaço em disco ocupado por 10 bilhões de instruções em média é de 14GiB em arquivos compactados, um número maior de instruções aumentaria ainda mais o espaço necessário para armazena-los e o tempo para coleta dos traços; (b) os traços são coletados durante a inicialização do programa e nosso testes mostram que esse número de instruções é suficiente para demonstrar as diferenças de desempenho entre o modelo de gerenciamento de memória e avaliar a efetividade das configurações de *TLB* e *SB* propostas; e (c) simulações realizadas com mais de 10 bilhões de instruções não apresentaram diferenças significativas nos resultados obtidos.

Após a inicialização, o *Trace Collector* cria o arquivo *execution.xml* que contém os seguintes elementos: (a) lista de arquivos de traços gerados; (b) informação se os arquivos de traços estão compactados; (c) informação se os arquivos texto originais foram apagados; (d) mensagens geradas pelo *Valgrind* recebidas junto com o fluxo de instruções (tais como mensagens contendo informações ou erros); (e) quantidade total de instruções geradas; (f) caminho absoluto para o diretório contendo os traços; e (g) objetos *Java* serializados (salvos em formato *XML*) que guardam o estado e a configuração do programa, que podem ser recuperados e utilizados para extrair informações de execução ou *debug*.

A classe principal do *Trace Collector* cria um objeto *ServerSocket*, configurado para escutar a porta 1500, e espera conexões vindas do *Valgrind*. Cada processo instrumentado pelo *Valgrind* cria uma conexão com o *Trace Collector* e cada conexão representa um fluxo independente de instruções. Dessa forma, se o processo instrumentado criar processos filhos, uma nova conexão à porta 1500 é feita pelo *Valgring* e o *Trace Collector* cria uma nova *thread* com um *socket* associado para ler esses fluxo de forma independente.

O mapa de memória do processo contém os endereços do binário do programa, sua área de dados e pilha, e todas as bibliotecas ligadas dinamicamente. Se o processo instrumentado cria processos filhos com binários diferentes, o mapa de memória do processo inicial é disjunto dos mapas dos novos fluxos de instruções, pois seus endereços fazem referência a outro mapa de memória. Por esse motivo configuramos o *Valgrind* para não gerar o fluxo de instruções dos programas filho. Ignorar os processos filhos não impacta na simulação de faltas na SB e TLB porque ambas recebem o mesmo fluxo de instruções.

Tabela 4.2: Comando para geração de traços do comando *UNIX ls*.

Linha de comando
`/opt/valgrind/bin/valgrind --log-socket=127.0.0.1 --trace-children=no -tool=lackey --trace-mem=yes ls`

A sequência de referências é gravada em um arquivo de texto. O tamanho máximo do arquivo é configurável e o limite é definido pelo número máximo de instruções *VEX* que o arquivo comporta. Quando esse limite é atingido, o arquivo é fechado e a classe principal do *Trace Collector* cria um novo arquivo de traços acrescentando um sufixo ao nome do arquivo, indicando a sua ordem de criação. O simulador cria arquivos sob demanda até que o fluxo de instruções termine. Para cada arquivo de traços criado, o seu nome é adicionado ao arquivo *execution.xml*.

Se o simulador estiver configurado para comprimir os traços, ele detecta quando um arquivo de traço no formato texto terminou, e cria uma *thread* para a sua compactação. Enquanto a ferramenta está gravando a sequência de instruções no arquivo de traços atual, o arquivo anterior é compactado. A *thread* de compactação cria um novo arquivo com sufixo *.zip* e remove o arquivo de texto original quando seu trabalho termina. Antes de finalizar, a *thread* de compactação atualiza o arquivo *execution.xml*, adicionando o arquivo de traços compactado.

Durante o desenvolvimento do trabalho descobrimos que os traços gerados tomam muito espaço em disco para guardar as 10 bilhões de instruções coletadas para simulação. Todas as coletas de traços para as simulações foram realizadas com o *Trace Collector* configurado para compactar os traços. Com isso conseguimos armazenar os traços com um tamanho bastante otimizado. Mesmo utilizando compactação, a média de tamanho de um diretório contendo uma captura de traços com 10 bilhões de instruções é de 14GiB.

A Tabela 4.2 mostra o comando para instrumentar o comando *ls* do sistema operacional *Linux*, que lista os arquivos contidos em um diretório. Para iniciar o envio de instruções para o *Trace Collector*, o *Valgrind* é executado especificando o módulo *Lackey*. O parametro *–trace-children=no* faz com que apenas as instruções do programa principal sejam inseridas no traço.

O parâmetro *–tool=lackey* diz ao *Valgrind* para executar o gerador de traços no programa instrumentado e o parametro *–trace-mem=yes* faz com que o *lackey* gere traços de todas as referências à memória. As instruções geradas pelo *Lackey* são direcionadas para um *socket* que aponta para o *loopback* com o parâmetro *–log-socket=127.0.0.1*. Assim, as instruções não são enviadas para a saída padrão e não perturbam a execução do programa. Todos os traços coletados utilizaram os mesmos parâmetros de configuração do *Valgrind*.

Para gerar os traços de alguns processos foi necessário descobrir a linha de comando que inicia o programa que estamos instrumentando e todos os parâmetros utilizados. Em muitos casos, como o do banco de dados *MySQL*, descobrimos os passos executados pelo *shell script* que controla a execução do *deamon* do processo. Por exemplo: quando iniciamos o *daemon* do banco de dados *MySQL*, os comandos abaixo são executados antes da chamada ao programa:

```
1 test -e /var/run/mysqld
2 install -m 755 -o mysql -g root -d /var/run/mysqld
3 su - mysql -s /bin/sh
```

Somente depois de executado o script mostrado acima é que o comando para executar o *Valgrind*, mostrado na Tabela 4.3, é invocado para gerar os traços de execução do *MySQL*. No

Tabela 4.3: Exemplo de comando para gerar traços do banco de dados *MySQL*.

Linha de comando
`/opt/valgrind/bin/valgrind --log-socket=127.0.0.1 --trace-children=no --tool=lackey --trace-mem=yes /usr/sbin/mysqld --basedir=/usr --datadir=/var/lib/mysql --plugin-dir=/usr/lib/mysql/plugin --log-error=/var/log/mysql/error.log --pid-file=/var/run/mysqld/mysqld.pid --socket=/var/run/mysqld/mysqld.sock --port=3306 --log-syslog=1 --log-syslog-facility=daemon --log-syslog-tag=`

Tabela 4.4: Exemplo de mapa de memória para o comando. *lsblk*.

Address	Size	Rights	Owner
0000.0000.0040.0000	72K	r-x--	lsblk
0000.0000.0061.1000	4K	r----	lsblk
0000.0000.0061.2000	4K	rw---	lsblk
0000.0000.0400.0000	152K	r-x--	ld-2.23.so
0000.0000.0402.6000	8K	rw---	[anon]
0000.0000.0402.8000	28K	r--s-	gconv-modules.cache
0000.0000.0404.b000	4K	rw---	[anon]
...	...	...	...
ffff.ffff.ff60.0000	4K	r-x--	[anon]

exemplo da Tabela 4.3, a chamada ao *MySQL* inicia com o comando */usr/sbin/mysqld* seguido dos parâmetros necessários para a execução.

A Figura 4.1 também mostra o script *pmap observer*. Sua função é monitorar o mapa de memória do processo *lackey* utilizando o comando *pmap*, e direcionar seu conteúdo a um arquivo de texto, que na Figura 4.1 é chamado de *pmap.txt*. O *script* monitora o processo a cada 10 milissegundos, até que o processo *lackey* termine. O *script* mantém um histórico de todas as mudanças do mapa de memória, adicionando um contador iniciado em 1 como parte do nome do arquivo, e gera um arquivo *CSV* com o histórico com: (a) o momento em que o mapa de memória mudou; (b) a quantidade de segmentos; (c) quantidade de bibliotecas (segmentos que contém ".so" no seu nome); e (d) quantidade de segmentos anônimos (com "anon" na descrição do segmento).

No sistema operacional *Linux* a chamada `mmap` cria um novo mapeamento no espaço de endereçamento virtual do processo que efetua a chamada. A chamada pode ser utilizada para criar um mapeamento para um arquivo em disco ou para criar um mapeamento anônimo. Mapeamentos anônimos são entradas no espaço de endereçamento virtual que não estão associados a um arquivo em disco e seu conteúdo é inicializado em zero [2].

A Tabela 4.4 mostra um exemplo do mapa de memória do comando *lsblk*, que lista os dispositivos de bloco presentes no sistema e contém as seguintes informações: (a) endereço inicial do segmento (Address); (b) tamanho do segmento (Size); (c) informações de proteção (Rights); (d) origem do segmento (Owner).

A instrumentação e os traços gerados pelo *Valgrind* possuem limitações. Algumas das limitações documentadas no código fonte do *lackey* são:

a) a ferramenta não gera traços do sistema operacional. Chamadas de sistema (*System Calls*) e *Signal Handling* são ignorados;

b) uma pequena parte do código do programa é substituída por código do *Valgrind* e não são gerados traços desse trecho de programa. Por exemplo, não são alterados trechos de código responsáveis por *scheduling operations* e *signal handling*;

c) o fluxo de instruções é alterado pelo *Valgrind* de forma sutil devido à dificuldade de simular algumas instruções nativas. Por exemplo, na arquitetura *x86* a simulação das instruções *bts, btc, btr* sempre acessa a memória;

d) a organização de memória do programa instrumentado difere significativamente do programa sem instrumentação, o que torna inviável a utilização de endereços absolutos para compará-los.

Apesar das diferenças entre o programa original e a sua versão instrumentada isso não afeta a simulação realizada para o SB e a *TLB*, pois é utilizado apenas o fluxo de endereços do programa instrumentado para executar a simulação. Isso garante que ambas os *caches* simulem as mesmas instruções na mesma ordem, o que é suficiente para comparar a quantidade de faltas na TLB com a quantidade de faltas na SB.

4.5 A tabela de segmentos para simulação

O mapa de memória da Tabela 4.4 é composto de segmentos, e cada segmento representa um espaço de endereçamento que pode conter: (a) dados; (b) código; (c) pilha; (e) bibliotecas compartilhadas com seus segmentos de dados e código; e (f) regiões que separam componentes (*guard regions*) utilizadas geralmente para que um eventual estouro de pilha não invada o próximo segmento.

O simulador de memória virtual utiliza endereços com segmentação a partir de traços reais de um programa que utiliza paginação. Isso é possível porque transformamos o mapa de memória gerado por *pmap* em uma tabela de segmentos, e a partir da tabela de segmentos, transformamos um endereço linear em um "endereço segmentado". A construção da tabela de segmentos é a primeira função executada pelo simulador, antes que qualquer arquivo de traços seja aberto e simulado.

O mapa de memória é utilizado como base para construir a tabela de segmentos (*Segment Table, ST*). Adicionamos as seguintes informações à *ST*: (a) identificador de segmento (*SegmentID)*; e (b) limite superior de segmento (*Top Address*).

A Tabela 4.5 mostra uma parte da tabela de segmentos do processo `lsblk`. O comando `lsblk` mostra detalhes do sistema de arquivos no sistema operacional *Linux*. A Tabela 4.5 contém os seguintes dados: (a) identificador de segmento (VSN); (b) endereço inicial do segmento (BaseAddr); (c) endereço final do segmento (TopAddr); (d) tamanho do segmento (Size); (e) informações de proteção (Rights); e (f) nome do segmento (Name).

Para calcular a limite superior (*TopAddr*) do segmento `0x0000` usamos seu endereço base (*BaseAddr*) `0x0040.0000` e somamos a ele o tamanho do segmento em *bytes* `A800` resultando no limite superior do segmento (*TopAddr*) `0x0040.A800`.

O identificador do segmento é um inteiro inicializado em zero e incrementado de um para cada linha da tabela de segmentos. O simulador percorre o mapa de memória, do inicio ao fim, observando se os 16 *bits* mais significativos do endereço linear são compostos apenas por zeros. Se sim, a coluna *SegmentID* recebe o incremento do *SegmentID*. Se os 16 *bits* mais significativos do endereço são diferentes de zero estes passam a ser o *SegmentID*.

Tabela 4.5: Sub-conjunto da tabela de segmentos para o *lsblk*.

VSN	BaseAddr	TopAddr	Size	Rights	Name
0x0000	0x0040.0000	0x0040.A800	43.008	r-x--	lsblk
0x0001	0x0061.1000	0x0061.2000	4.096	r----	lsblk
0x0002	0x0061.2000	0x0062.3000	4.096	rw---	lsblk

Na execução do programa instrumentado pelo *Valgrind* o mapa de memória do processo muda de forma dinâmica. A quantidade de segmentos aumenta quando uma nova biblioteca dinâmica é carregada. O *pmap observer* não monitora o processo do *Lackey* em tempo real, ele o faz em intervalos de 10 milissegundos. Existe a possibilidade de que, entre cada consulta ao mapa de memória pelo *pmap*, a quantidade de segmentos tenha mudado. Optamos por não monitorar o processo do *lackey* em tempo real para simplificar o processo de coleta de traços, e por ser um tempo suficiente para capturar mudanças significativas no mapa de segmentos do processo.

4.6 Ciclo de vida do mapa de memória dos processos

A quantidade de segmentos mapeados cresce muito rapidamente no inicio da execução, porque as bibliotecas compartilhadas são carregadas e segmentos de dados para *heap* e pilha são alocados. Outros recursos do sistema podem ser mapeados para segmentos anônimos específicos (como dispositivos por exemplo). E alguns segmentos são alocados para separar regiões de memória (*guard regions*).

A Figura 4.2 mostra a quantidade total de segmentos mapeados para o processo do navegador *Firefox* e quantos desses segmentos são bibliotecas compartilhadas. Cada vez que o *pmap observer* monitora o mapa de memória do processo utilizando o comando `pmap`, e detecta uma mudança no mapeamento, o *script* adiciona uma entrada em um arquivo *CSV*. O gráfico da Figura 4.2 mostra, no eixo "X", a ordem da leitura no arquivo *CSV* (tempo) e, no eixo "Y", a quantidade de segmentos. O gráfico termina quando o programa termina a execução de 10 bilhões de instruções.

O mapa de memória mostrado na Figura 4.2 é dinâmico. O sistema operacional aloca segmentos para o processo conforme a necessidade, removendo segmentos que não são mais utilizados. A listagem abaixo mostra um desses casos, quando mapeamento de um segmento de biblioteca compartilhada (`libresolv-2.23.so`) é alterado para um mapeamento de uma região anônima.

```
1 diff pmap-75.txt pmap-76.txt
2
3 234c234
4 < 000000000e7ce000 8K r-x-- libresolv-2.23.so
5 ---
6 > 000000000e7ce000 8K rw--- [ anon ]
```

Observamos poucas mudanças no mapa de memória envolvendo mais que um ou dois segmentos entre duas execuções do `pmap observer`. Existem casos em que o sistema operacional remove diversos mapeamentos no mesmo momento. A Figura 4.2 mostra um desses picos na quantidade de segmentos durante a execução da coleta de traços entre os tempos 1081 e 1225.

Na listagem abaixo temos um exemplo de um desses picos. Nesse exemplo são removidos 13 segmentos anônimos e é adicionado um novo segmento anônimo entre duas coletas com o `pmap observer`. Essa natureza dinâmica do mapa de memória de um processo traz dificuldades adicionais para simular a segmentação com acurácia. Mostra também que este é um potencial cenário onde o gerenciamento de memória utilizando segmentação pode ser usado para combinar vários segmentos pequenos em um único segmento grande.

```
diff pmap-1148.txt pmap-1149.txt

939,952c939
< 00000065f6551000 640K ----- [ anon ]
< 00000065f65f1000 64K r-x-- [ anon ]
< 00000065f6601000 64K ----- [ anon ]
< 00000065f6611000 64K r-x-- [ anon ]
< 00000065f6621000 320K ----- [ anon ]
< 00000065f6671000 64K r-x-- [ anon ]
< 00000065f6681000 64K ----- [ anon ]
< 00000065f6691000 64K r-x-- [ anon ]
< 00000065f66a1000 192K ----- [ anon ]
< 00000065f66d1000 64K rw--- [ anon ]
< 00000065f66e1000 256K ----- [ anon ]
< 00000065f6721000 60K r-x-- [ anon ]
< 00000065f6730000 4K r-x-- [ anon ]
< 00000065f6731000 1046592K ----- [ anon ]
---
> 00000065f6551000 1048512K ----- [ anon ]
```

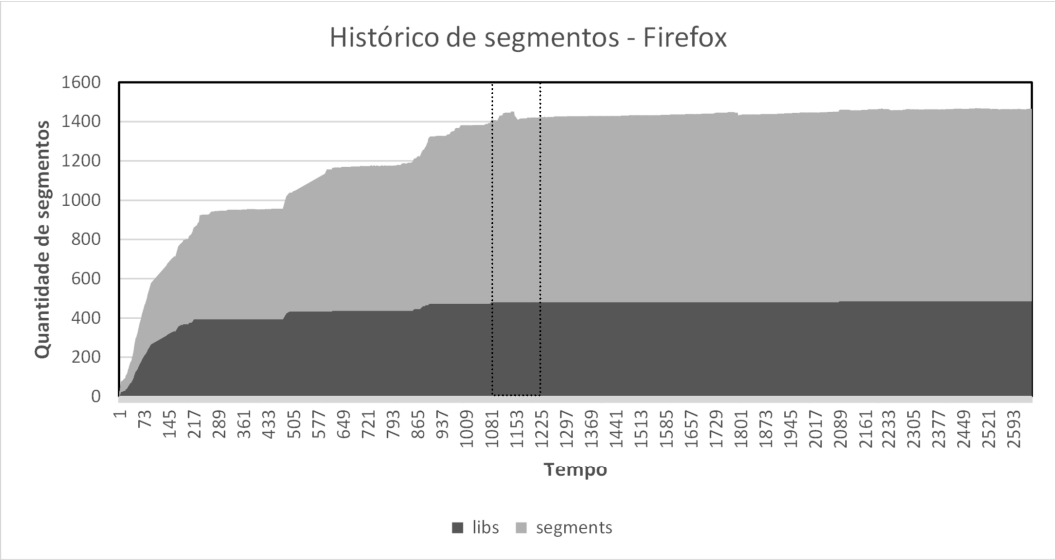

Figura 4.2: Segmentos criados pelo Firefox durante a execução.

Quando o programa instrumentado muda o seu mapa de memória antes de uma execução do *pmap observer*, existe a possibilidade da simulação conter instruções que fazem referência a endereços de um segmento que não constam no último mapa de memória salvo do processo. Segmentos que foram removidos do mapa de memória durante a execução do processo fazem com que instruções e referências à memória presente nos traços não estejam contidos em nenhuma segmento, pois o segmento foi removido. A referência se torna "órfã" pois seu endereço de memória não está contido em nenhum segmento do mapa de memória.

Tabela 4.6: Segment table (fragment).

VSN	BaseAddr	TopAddr	Size	Rights	Name
0000	0x0108000	0x012F000	159744	r-x-	firefox
0001	0x032E000	0x0330000	8192	rw--	firefox
0008	0x0330000	0x4000000	63766528	---	[simulated 0]
0002	0x4000000	0x4026000	155648	r-x-	ld-2.23.so
0003	0x4026000	0x4028000	819	rw--	[anon]

Nos traços gerados foi observado uma incidência baixa desse problema e, quando ocorre, o número de instruções com referência a endereços superiores ao tamanho do segmento é na ordem de 10 milhões de instruções. Em uma execução na ordem de bilhões de instruções, representa uma distorção de aproximadamente $0,001\%$ nos resultados obtidos.

Para evitar referências a endereços de memória "órfãos" de um segmento optamos por simplificar a simulação. Sempre que o simulador encontra uma instrução fazendo referência a um endereço de memória que não está contido em nenhum segmento presente na última execução do `pmap observer`, o simulador cria um novo segmento "simulado" para acomodar o endereço na tabela de segmentos. O segmento "simulado" tem como endereço inicial o endereço do topo do segmento anterior ao endereço referenciado. E o topo do segmento "simulado" será o endereço base do próximo segmento. Dessa forma a simulação da *TLB* e da *SB* utilizam todos as instruções do traço,e referência os endereços que estão contidos nos segmentos "simulados" produzem faltas e acertos que podem ser mensurados.

Nos traços utilizados, as simulações criam entre 0 e 5 segmentos "simulados" durante a execução. Isso indica que o mapa de memória final do processo possui poucas referências aos segmentos que foram removidos ou alterados durante a execução do processo. A Tabela 4.6 mostra um exemplo de segmento "simulado" de número `0008`, adicionado a tabela de segmentos do processo do *Firefox*.

4.7 Simulação de troca de contexto

O coletor de traços *Trace Collector* pode ignorar uma quantidade configurável de instruções do *Valgrind* antes de começar a gravar os traços em arquivos. Esse mecanismo foi utilizado para simular uma troca de contexto. Nesse cenário o *Trace Collector* ignora os primeiros 10 bilhões de instruções e somente então coleta 2 bilhões de instruções em arquivos. Os dois bilhões de instruções já possuem um mapa de memória com uma quantidade significativa de bibliotecas e segmentos mapeados, e dessa forma o padrão de faltas na *TLB* e na *SB* muda significativamente em comparação a simulação do inicio do processo.

4.8 De endereços lineares para endereços segmentados

A segmentação de memória é a divisão da memória utilizada pelo programa em segmentos lógicos. Em um sistema operacional que utiliza segmentação de memória, um endereço de memória é composto por um identificador de segmento e um deslocamento (*offset*) dentro do segmento. O identificador do segmento é o indice na tabela de segmentos que mantém informações de todos os segmentos presentes em memória. Um dos campos da tabela de

segmentos é o endereço físico inicial do segmento. A soma do endereço físico inicial apontado pelo *SegmentID* mais o *offset* resulta no endereço fisico de memória [20].

Segmentos são utilizados em arquivos objeto (*object files*) de programas compilados para representar divisões naturais do programa, como pilha, dados e código. Geralmente um arquivo objeto contém 3 segmentos, com todo o necessário para a execução do programa. Esse tipo de segmentação encontrada em um programa compilado é mais intuitiva para o programador que escreve o compilador, ligador e carregador do que paginação.

A Figura 4.3 mostra como um endereço presente no traço de execução de um programa é transformando em um segmentado para fins de simulação. A tabela de segmentos é construída a partir do mapa de memória do processo em execução, extraída pelo comando *pmap*. Com as informação da tabela é possível obter o "endereço segmentado" que corresponde a um "endereço linear".

Durante a execução do simulador, para cada endereço (*AddrPag*) lido no arquivo de traços, é feita uma busca na tabela de segmentos para encontrar a qual segmento ele pertence. A busca é feita comparando o "endereço linear" ao endereço inicial (*BaseAddr*) e final (*TopAddr*) da tabela de segmentos com o endereço *AddrPag*.

$$BaseAddr \leqslant AddrPag \leqslant TopAddr$$

Se o endereço *AddrPag* está na faixa de endereços entre o inicial e o final de algum segmento, a busca é considerada bem sucedida. O identificador de segmento é separado para gerar o endereço segmentado.

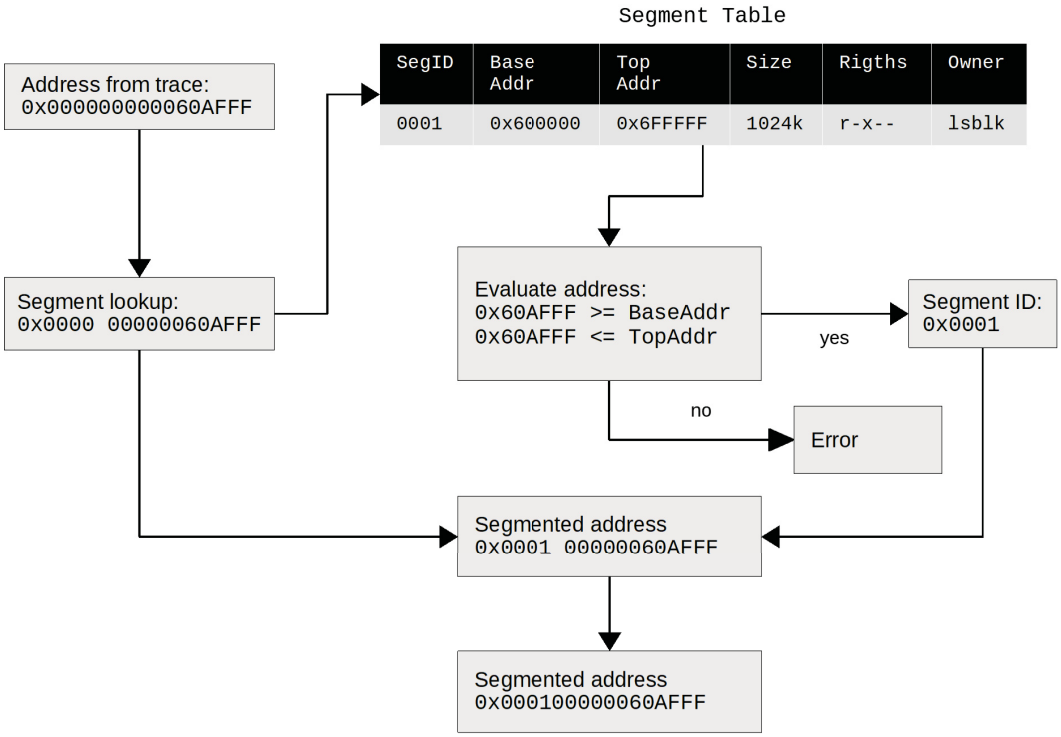

Figura 4.3: Passos para transformar endereços lineares em endereços segmentados.

Encontrado o identificador de segmento, os 16 *bits* mais significativos do *AddrPag* são substituídos pela reapresentação em 16 *bits* do identificador do segmento, como mostrado na Figura 4.3. Nesse exemplo, temos o *AddrPag* com valor *0x60AFF* entre o valor de *BaseAddr* e *TopAddr* na tabela de segmentos, e com *SegID* igual a *0x0001*, resultando em um endereço

de memória segmentado *0x**0001**00000060AFFF*, sendo os 16 *bits* mais significativos como *SegmentID* e o restante como *offset*.

Todas as aplicações avaliadas possuem um segmento com os 16 *bits* mais significativos diferentes de zero. Esse segmento sempre é o último segmento reportado no mapa de memória pelo *pmap*, e nossas simulações não mostram nenhuma referência a este segmento em nenhum dos traços coletados.

Definimos a tabela de segmentos e um algoritmo para transformar os "endereços lineares" em "endereços segmentados" a partir de uma traço de execução real, possibilitando a simulação de um *buffer* de segmentos. Com isso, temos recursos suficientes para realizar a simulação da *TLB* e do *SB* e comparar a quantidade de faltas em cada um dos *buffers* para traços de execuções reais.

4.9 O simulador

Efetuamos simulações de um conjunto de traços de execução contendo referências a memória, e avaliamos o desempenho da tradução de endereços virtuais para físicos utilizando como métrica principal a quantidade de faltas que os *Segment Buffer* e *TLB* sofrem.

A arquitetura dos *SBs* e *TLBs* é simples e inspirada na plataforma *MIPS*. A comparação entre *TLB* e *Segment Buffer* é realizada com o mesmo tamanho e organização, com uma complexidade similiar de implementação em *hardware*.

O simulador de traços mostrado na Figura 4.4 é um programa escrito em *Java*, composto de: (a) uma rotina principal que recebe parâmetros de execução e o arquivo *execution.xml* com as informações da captura dos traços de execução do programa por simular; (b) classes que fazem a leitura dos traços e preparam os dados para serem consumidos pela simulação do *SB* e *TLB*; (c) uma *thread* que executa a simulação de *TLB*; (d) uma *thread* que executa a simulação do *SB*; (e) um conjunto de classes que faz a consolidação dos resultados; e (f) um conjunto de métodos que escrevem os resultados consolidados em um arquivo texto.

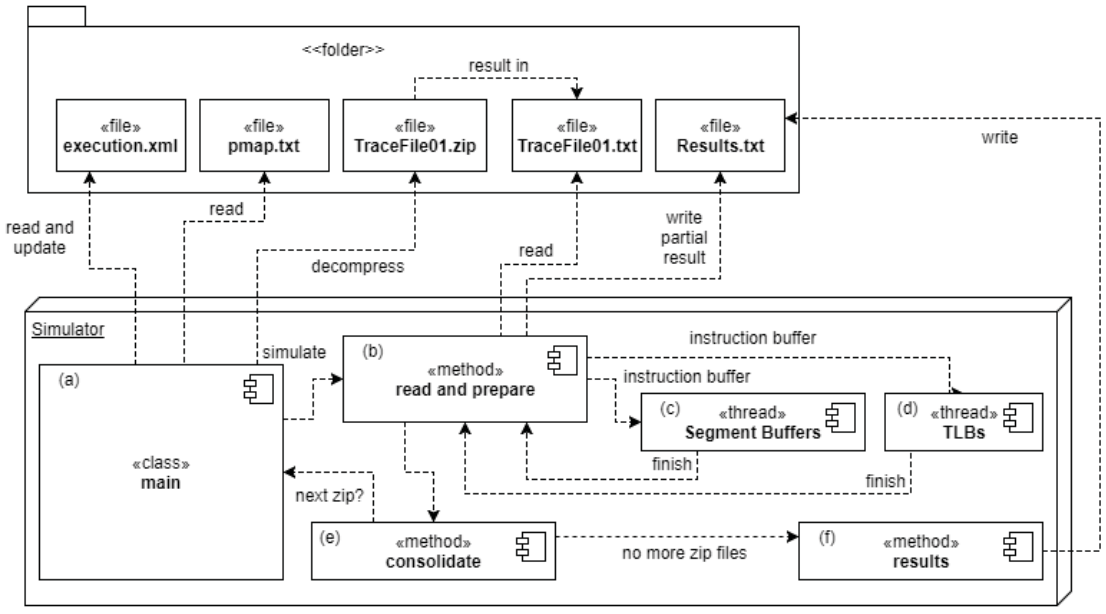

Figura 4.4: Simulador de *TLB* e *Segment Buffer*.

A simulação da *TLB* utiliza os endereços diretamente do arquivo de traços e nenhuma alteração nas referências à memória é necessária para realizar a simulação, pois os traços foram coletados de um programa executando na plataforma *Intel x86-64* com paginação. Definimos para simulação o tamanho das páginas é 4KiB e uma *TLB* não hierárquica. Para o *Segment Buffer*, a simulação transforma os endereços de memória do traço em endereços segmentados utilizando a tabela de segmentos do processo original.

Para iniciar a execução do simulador é necessário especificar o arquivo contendo as informações de coleta de traços *execution.xml*, o arquivo texto com o mapa de memória (*pmap.txt*) e o arquivo em que os resultados devem ser escritos (*result.txt*), como mostra a Figura 4.4. A primeira tarefa do simulador, após validar os parâmetros, é construir a tabela de segmentos utilizando o arquivo com o mapa de memória. Outra tarefa é a criação de uma lista de objetos representando as diferentes configurações de *TLBs* (*tlbList*) e uma lista de objetos representando as configurações de SBs (*sbList*) por simular.

O simulador representado na Figura 4.4 é implementado pela classe *Trace CollectorSimulator*. A execução da classe *Trace CollectorSimulator* itera a lista de arquivos de traços contida no arquivo *execution.xml*, descompactando o arquivo de traço atual em uma pasta temporária. Se o arquivo de traço está no formato texto, ele é usado diretamente. Utilizar compactação dos traços é uma funcionalidade configurável do *Trace Collector* e o simulador está preparado para utilizar os dois formatos.

A rotina do simulador "*read and prepare*" mostrada na Figura 4.4, é realizada pela classe *TraceCollectorSimulator* que lê e armazena a sequência de instruções em uma lista de instruções de tamanho fixo (um *buffer* de instruções). Quando não há mais espaço disponível na lista, o simulador cria duas *threads* para consumir as instruções: (a) uma *thread* responsável por simular a lista de *TLBs* (*TlbWorkerThread*); e (b) uma *thread* responsável por simular o a lista de *SegmentBuffers* (*SBThread*).

As *threads TlbWorkerThread* e *SegmentBufferWorkerThread* executam a simulação consumindo a lista de instruções enquanto a *thread* principal do programa (*MainThread*) itera a lista de instruções e atualiza a tabela de segmentos com a quantidade de referências a memória. Quando a *thread* principal termina de contar as referências à memória ela fica aguardando o termino das execuções das *threads* de simulação.

Quando as execuções das *threads* finalizam, a lista de instruções é descartada e outra lista é criada. O simulador retoma a leitura do arquivo atual, continuando a simulação quando a lista de instruções estiver completa novamente. O simulador continua o processo de popular a lisa de instruções, contar referências e simular a *TLB* e a *SB* até que não existam mais instruções para serem lidas do arquivo. Quando isso ocorre, quaisquer instruções que estejam na lista de instruções são simuladas e o simulador lê o próximo arquivo de traços contido no arquivo *execution.xml*.

A cada 200.000 instruções, o simulador mostra na saída padrão o resultado parcial da simulação das *TLBs* e *SegmentBuffers*, com a quantidade de consultas e faltas. Ao final de cada arquivo de traço processado, o arquivo texto *results.txt* com o resultado consolidado da simulação é criado com um sufixo com o mesmo número identificador do arquivo de traços que acabou de ser lido.

4.10 Simulação de *caches*

A arquitetura do simulador de caches é mostrada no diagrama de classes da Figura 4.5. A classe *Cache* define as funcionalidades comuns presentes tanto na simulação da *TLB* quanto na simulação do *SB*. As funcionalidades comuns são os atributos tamanho (*size*) de linhas da

cache e sua associatividade (*associativity*), e estas definem a organização da *Cache*. Discutimos adiante como esses atributos se relacionam com as outras classes do diagrama, e como diferentes configurações de *TLB* e de *SB* podem ser simuladas.

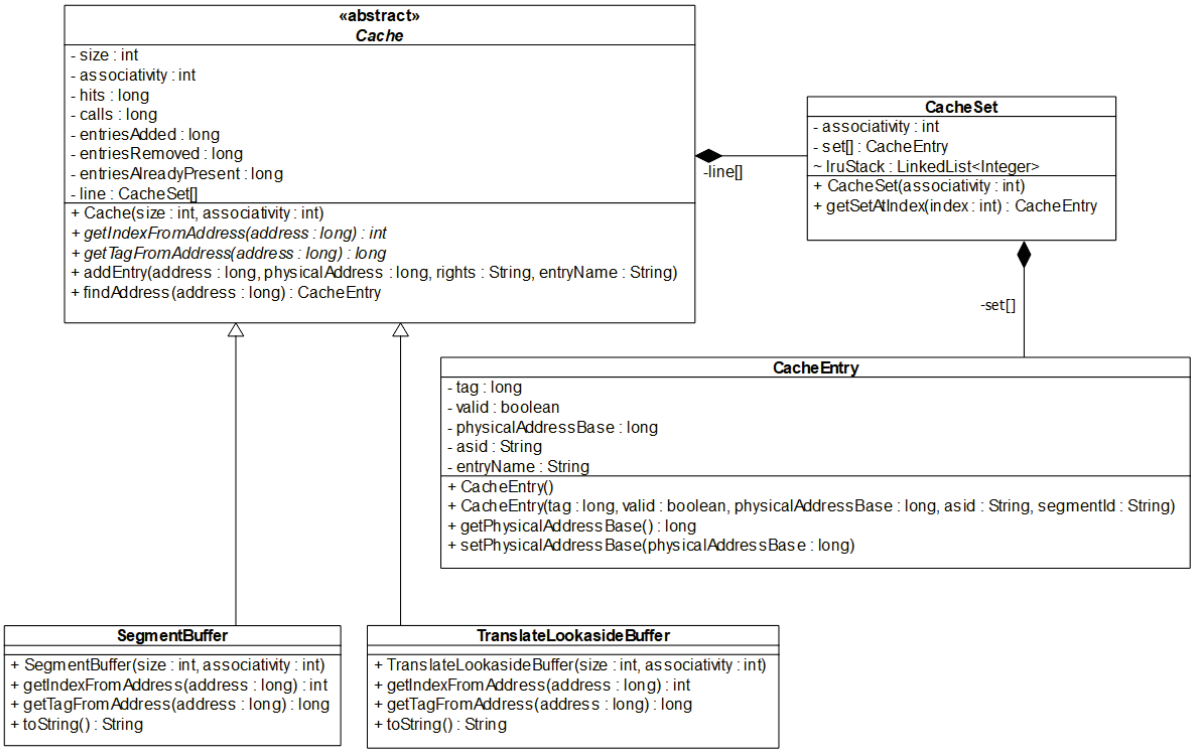

Figura 4.5: Implementação da *TLB* e do *Segment Buffer*.

A classe *Cache* é uma classe abstrata. Classes abstratas em *Java* definem: (a) um modelo (*template*) para uma funcionalidade; e (b) uma implementação incompleta, a parte genérica da funcionalidade, que é compartilhada por todas as classes concretas que extendem a classe abstrata. Classes concretas devem fornecer uma implementação para toda funcionalidade definida no modelo [15]. No simulador, *Cache* é uma classe abstrata. Ela define o *template* para dois métodos:

1. *getIndexFromAddress(long address)*: retorna um inteiro utilizado para indexar a matriz de *CacheSet* a partir de um endereço de memória passado como parâmetro; e

2. *getTagFromAddress(long address)*: retorna a etiqueta, um identificador do endereço, a partir do endereço passado como parâmetro.

Na classe *Cache*, o índice aponta para o conjunto, ou "linha", onde está a entrada que estamos procurando. O conjunto é representado pela matriz de objetos *CacheSet* mostrada na Figura 4.5. Cada *CacheSet* possui uma matriz de *CacheEntry* que representa um elemento com os atributos necessários para realizar a simulação. A classe *CacheSet* possui também o atributo *lruStack* que é uma lista encadeada de inteiros responsável pelo controle de qual *CacheEntry* do *CacheSet* é o *Least Recently Used* (LRU). Adiante mostramos em detalhes como é realizado o cálculo do *LRU*.

A classe *CacheEntry* possui dois atributos importantes para a simulação: (a) a etiqueta (*tag)* que guarda parte do endereço de memória e é utilizada para validar se o endereço contido

na *CacheEntry* é realmente o endereço procurado; e (b) o atributo *valid* que indica se o endereço em *CacheEntry* é válido e pode ser usado.

Com o índice do endereço e sua etiqueta (*tag*) é possível descobrir se o endereço está presente na *Cache*. O índice indexa a matriz de instâncias de *CacheSet*. Descoberto qual instância de *CacheSet* o simulador itera sobre a matriz de *CacheEntry* contida no *CacheSet*, comparando a etiqueta do endereço procurado com a etiqueta presente na instância de *CacheEntry*. Se a etiqueta for encontrada em alguma instância de *CacheEntry*, o simulador incrementa o contador de acertos na respectiva *Cache*.

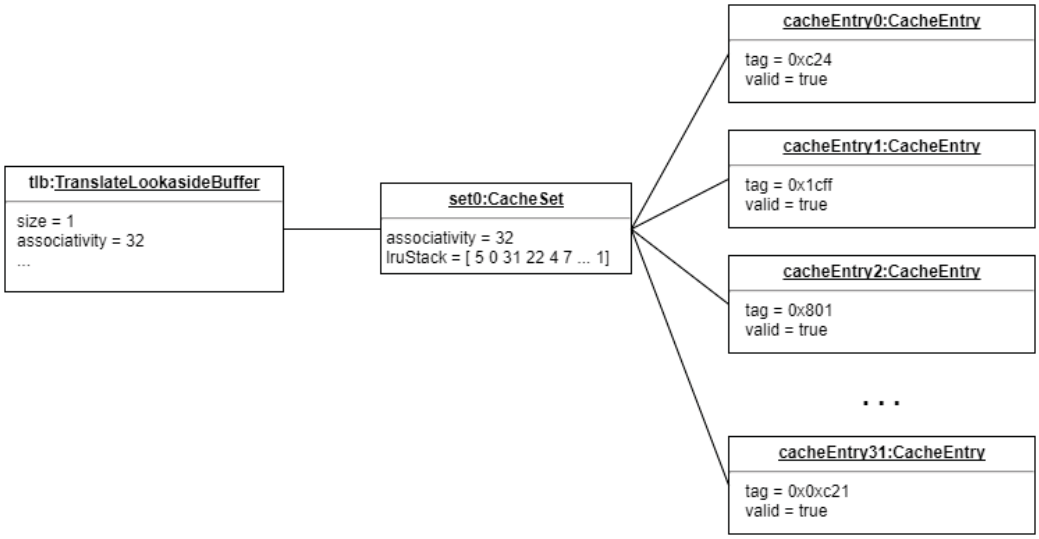

Figura 4.6: Exemplo de *TLB* totalmente associativa com 32 posições.

A Figura 4.6 mostra uma *TLB* totalmente associativa com 32 posições. O exemplo tem uma instância da classe *Cache* que contém apenas uma instância da classe *CacheSet*, pois a *TLB* é totalmente associativa, e a instância de *CacheSet* tem uma matriz com 32 instâncias da classe *CacheEntry*.

Uma *TLB* com 128 posições configurada com 16 linhas e cada linha com um conjuntos associativo de 8 elementos é mostrada na Figura 4.7. O exemplo mostra uma instância da classe *TLB* que contém 16 instâncias da classe *CacheSet*, e cada instância de *CacheSet* tem uma matriz com 8 instâncias da classe *CacheEntry*.

A implementação da simulação do *SB* é similar, pois a classe *SegmentBuffer* também estende a classe pai *Cache* e implementa os métodos concretos necessários. Portanto os diagramas para a *SB* são estruturalmente iguais, e a única mudança é a utilização de uma instância da classe *SegmentBuffer* no lugar da classe *TranslateLookAsideBuffer*.

4.11 Substituição do *Least Recently Used* (LRU)

O simulador utiliza o atributo *lruStack* da classe *CacheSet* para manter o controle do *Least Recently Used* (LRU). A *lruStack* é uma lista encadeada de inteiros em que cada posição contém um índice da matriz de *CacheEntry*. O primeiro elemento do *lruStack* é o elemento que foi utilizado mais recentemente. O último elemento da lista encadeada é o elemento que está há mais tempos sem ser utilizado.

Quando o simulador procura um endereço na *Cache* e o encontra, o *lruStack* é atualizado. O índice da *CacheEntry* é removida da sua posição atual da *lruStack* e inserido no início da lista

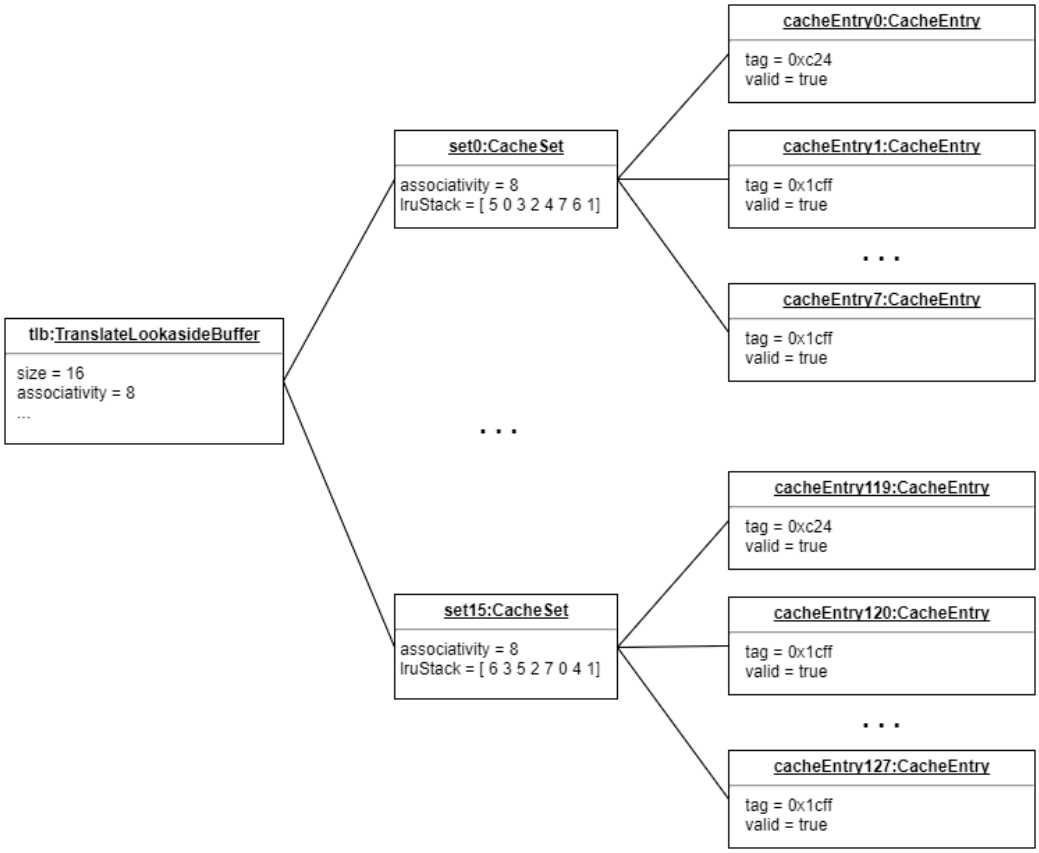

Figura 4.7: Exemplo de *TLB* com 128 posições, e 16 linhas *8 way set associative*.

```
Configuration ...........: TLB with 128 entries, configured with
                           16 lines and 8 ways set-associative (LRU).
Total calls: 14231850245, Total hits: 14218717189
Entries created: 13133056, Entries already present: 0
Entries removed: 13132928
Final cache contents:
Line: 0  (LRU indexes =[ 4 3 2 0 7 6 5 1])
>> Set  0 - CacheEntry [tag=0x42d, valid=true, entryName='[ anon ] rwx--']
>> Set  1 - CacheEntry [tag=0x43d, valid=true, entryName='[ anon ] rwx--']
>> Set  2 - CacheEntry [tag=0x45, valid=true, entryName='tb_cmips r-x--']
>> Set  3 - CacheEntry [tag=0x433, valid=true, entryName='[ anon ] rwx--']
>> Set  4 - CacheEntry [tag=0x428, valid=true, entryName='[ anon ] rwx--']
>> Set  5 - CacheEntry [tag=0x429, valid=true, entryName='[ anon ] rwx--']
>> Set  6 - CacheEntry [tag=0x4a, valid=true, entryName='tb_cmips r-x--']
>> Set  7 - CacheEntry [tag=0x42a, valid=true, entryName='[ anon ] rwx--']
Line: 1  (LRU indexes =[ 5 3 7 6 4 0 1 2])
>> Set  0 - CacheEntry [tag=0x43a, valid=true, entryName='[ anon ] rwx--']
>> Set  1 - CacheEntry [tag=0x45, valid=true, entryName='tb_cmips r-x--']
>> Set  2 - CacheEntry [tag=0x4d, valid=true, entryName='tb_cmips r-x--']
>> Set  3 - CacheEntry [tag=0x435, valid=true, entryName='[ anon ] rwx--']
>> Set  4 - CacheEntry [tag=0x42d, valid=true, entryName='[ anon ] rwx--']
>> Set  5 - CacheEntry [tag=0x51, valid=true, entryName='tb_cmips r-x--']
>> Set  6 - CacheEntry [tag=0x42, valid=true, entryName='tb_cmips r-x--']
>> Set  7 - CacheEntry [tag=0x434, valid=true, entryName='[ anon ] rwx--']
```

Figura 4.8: Exemplo de cache com cálculo do *Least Recently Used* (LRU).

encadeada. Dessa forma, a última posição da lista encadeada *lruStack* aponta sempre para o elemento mais antigo de *CacheEntry*, como mostrado na Figura 4.8.

Caso nenhuma etiqueta presente na matriz de *CacheEntry* seja a procurada, o simulador incrementa o contador de faltas da *Cache* e adiciona uma nova entrada para o endereço de memória utilizando o método *addEntry()*, mostrado na Figura 4.5. Um novo elemento na *Cache* resulta em uma nova instância de *CacheEntry* dentro de um *CacheSet*. O *lruStack* é atualizado com o índice da nova instância *CacheEntry*, sendo colocado como primeiro elemento da lista.

Se não existe espaço disponível no *CacheSet* para adicionar um novo elemento, o *lruStack* é consultado para descobrir qual é a última *CacheEntry* da lista. Esta então é removida para que uma nova instância de *CacheEntry* possa ser adicionada.

4.12 Implementação da *TLB*

A simulação da *TLB* é realizada pela classe *TranslateLookAsideBuffer* que implementa os métodos concretos *getIndexFromAddress* e *getTagFromAddress* para suportar as configurações simuladas: (a) TLB32; (b) TLB64; e (c) TLB128.

```
1  //
2  // TLB methods.
3  //
4  public static final long MASK = 0xfL;
5
6  @Override
7  public int getIndexFromAddress(long address) {
8        int index = 0;
9        if (getSize() > 1) {
10             long indexAndTag = address >>> 12;
11             index = ((int) (indexAndTag & MASK)) % getSize();
12       } else {
13             index = 0;
14       }
15       return index;
16 }
17
18 @Override
19 public long getTagFromAddress(long address) {
20       return address >>> 16;
21 }
```

O trecho de código acima mostra a implementação do método *getIndexFromAddress*. O método recebe como parâmetro um endereço de memória e retorna um inteiro, que é o índice da instância de *CacheSet* mapeado para o endereço. O objetivo da *TLB* é implementar uma *Cache* para páginas de memória com tamanho de 4KiB, então os 12 primeiros *bits* do endereço são descartados, pois fazem referência a endereços que estão contidos dentro de uma mesma página.

Os 4 primeiros *bits* dos 52 *bits* restantes são divididos do número de instâncias *CacheSet* e o resto da divisão resulta no índice que aponta para a linha da *cache* que contém a entrada desejada. Se a *TLB* for completamente associativa o método *getIndexFromAddress()* sempre retorna zero. A etiqueta é calculada utilizando os 16 *bits* mais significativos do endereço de memória passado como parâmetro como etiqueta (*tag*).

O calculo do resto para normalização do índice é efetuado para que o método *getIndexFromAddress* seja versátil e retorne índices para outras configurações de *Cache* com até 16 instâncias de *CacheSet*. Durante o desenvolvimento do simulador testamos *TLBs* com 64 posições *4-way set associative* (16 instâncias) e com 32 posições *4-way set associative* (8 instâncias) e a implementação de *getIndexFromAddress* não precisou ser alterada para executar a simulação.

O cálculo da etiqueta de um endereço de memória é realizado pelo método *getTagFromAddress* mostrado no código listado acima. O método descarta os 12 *bits* referentes ao tamanho da página e mais 4 *bits* utilizados para indexação da *cache*. Os 48 *bits* restantes são a etiqueta do endereço.

4.13 Implementação do *SB*

Simulamos um endereço de memória com segmentação a partir de um traço de execução real de um programa executado em uma *CPU Intel x86-64*, que utiliza endereços de memória lineares. O endereço segmentado utiliza os 48 *bits* menos significativos do endereço original e um identificador de segmento (*SegId*) gerado sequencialmente, na criação do segmento, nos 16 *bits* mais significativos.

O simulador utiliza a classe *SegmentBuffer* para simular a *SB* com endereços de memória segmentados. A classe *SegmentBuffer* estende a classe *Cache* e implementa os métodos *getIndexFromAddress* e *getTagFromAddress* como mostra o trecho de código abaixo:

```
1  //
2  // SB methods.
3  //
4  public static final long MASK = 0xfL;
5
6  @Override
7  public int getIndexFromAddress(long address) {
8        int index = 0;
9        if (getSize() > 1) {
10             long segId = address >>> 48;
11             index = ((int) (segId & MASK)) % getSize();
12       } else {
13             index = 0;
14       }
15       return index;
16 }
17
18 @Override
19 public long getTagFromAddress (long address) {
20       return (address >>> 48);
21 }
```

O método *getIndexFromAddress()* utiliza o identificador do segmento para calcular o índice da *Cache*. Se o *SegmentBuffer* não for completamente associativo (*size* >1) os primeiros 48 *bits* do endereço recebido como parâmetro são descartados. Os 4 primeiros *bits* dos 16 *bits* restantes são normalizados pelo resto da divisão pelo número de instâncias *CacheSet*, resultando no índice. Se o *SegmentBuffer* for completamente associativo, o método *getIndexFromAddress()* sempre retorna zero. A etiqueta é calculada utilizando os 16 *bits* mais significativos do endereço de memória que também é o identificador do segmento (SegmentID).

Como *SegmentBuffer* estende *Cache* da mesma forma que a classe *TranslateLookAsideBuffer* e implementa os métodos *getIndexFromAddress* e *getTagFromAddress*, por herança (orientação a objetos), o funcionamento das duas classes é igual. A implementação especializada dos métodos concretos faz a classe *SegmentBuffer* interpretar endereços segmentados e a classe *TranslateLookAsideBuffer* entender endereços lineares. O cálculo de faltas, a criação de novas entradas na *Cache*, a procura de um endereço na *Cache*, funcionam da mesma forma nas duas classes porque estas funcionalidades (métodos) estão implementados na classe abstrata *Cache*.

Tabela 4.7: Exemplo do arquivo de resultados *simulation-results*.

Configuration	TLB		SB	
	Hits	Miss	Hits	Miss
32 entries, fully associative	8.669.830.499	66.969.423	8.736.784.740	15.182
64 entries, fully associative	8.722.540.802	14.259.120	8.736.799.550	372

Tabela 4.8: Exemplo do arquivo de resultados *memtable.csv*.

SegID	Base Address	Size	Rights	Owner	Instr.	Mem.
0022	0x52D6000	1.830.912	r-x–	libc-2.23.so	1.580.841	28.108
0023	0x5495000	2.097.152	—–	libc-2.23.so	0	0
0024	0x5695000	16.384	r—-	libc-2.23.so	0	17.680
0025	0x5699000	8.192	rw—	libc-2.23.so	0	42.611
0026	0x569B000	16.384	rw—	[anon]	0	21.802
0027	0x569F000	16.384	r-x–	libuuid.so.1.3.0	48	689
0028	0x56A3000	2.093.056	−−—	libuuid.so.1.3.0	0	0
0029	0x58A2000	4.096	r—-	libuuid.so.1.3.0	0	277
002A	0x58A3000	4.096	rw—	libuuid.so.1.3.0	0	49

4.14 Saída da simulação

O simulador produz um arquivo texto chamado "*simulation-results*" que contém os resultados da simulação. O arquivo de resultados é gravado toda vez que o simulador termina de simular todas as instruções contidas em um arquivo de traço. Quando o arquivo de traço termina, o simulador adiciona no final do nome do arquivo de resultados a quantidade de instruções simuladas até o momento. Por exemplo: o arquivo "*simulation-results.txt-2549999999.txt*" contém o resultado da simulação de 2.549.999.999 instruções. Ao final da execução da simulação, quando todos os arquivos de traços foram consumidos, o simulador cria o arquivo "*simulation-results.txt*" com o resultado final da simulação.

O arquivo de resultado contém a quantidade de referências, acertos e faltas em cada *TLB* ou *SB* simulada. Um exemplo de resultado é mostrado na Tabela 4.7 que contém as seguintes informações: (a) a configuração da *cache* simulada (*Configuration*; (b) os resultados da TLB divididos em quantidade de acertos (*Hits*) e faltas (*Miss*); e (c) os resultados da SB divididos em quantidade de acertos (*Hits*) e faltas (*Miss*).

A Tabela 4.7 mostra um comparativo entre os resultados da simulação da execução de 8.736.799.922 instruções do script "*pip3 install −−upgrade*", executado em um interpretador *Python3*, para diferentes configurações de *SB* e *TLB*. O script *pip* é um gerenciador de pacotes para *Python* e seu funcionamento e resultados são mostrados no capitulo 5.

O simulador gera outro arquivo de resultados, com informações da tabela de segmentos utilizada pela simulação da *SB*, e chamado "*memtable.csv*". O arquivo "*memtable.csv*" é um arquivo texto com valores separados por vírgulas (*CSV - Comma-separated values*) e contém a tabela de segmentos utilizada pelo simulador, e informações sobre a quantidade de referências a endereços de memória, de dados e instruções, de cada segmento. O arquivo *CSV* também é gerado toda vez que o simulador executa a simulação de todas as instruções de um arquivo de traço, da mesma forma que o arquivo de resultados "*simulation-results*". Dessa forma é possível acompanhar as mudanças na tabela de segmentos conforme progride a simulação.

Tabela 4.9: Exemplo de arquivo de estatísticas para o processo *lsblk*.

buffer	calls	hits	miss	added	present	removed	rogue
TLB with 32 entries	2.586.688	2.586.336	352	352	0	320	0
TLB with 64 entries	2.586.688	2.586.619	69	69	0	5	0
TLB with 128 entries	2.586.688	2.585.340	1.348	1.348	0	1.220	0
SB with 32 entries	2.586.688	2.586.674	14	14	0	0	0
SB with 64 entries	2.586.688	2.586.674	14	14	0	0	0
SB with 128 entries	2.586.688	2.586.674	14	14	0	0	0

Cada linha do arquivo *CSV* contém um segmento com os valores referentes a ele separados por virgulas, representando as colunas. Um exemplo do arquivo *CSV* de resultados pode ser visto na Tabela 4.8. Nele podemos ver as colunas: (a) *SegId*, com o identificador do segmento; (b) *Base Address*, contendo o endereço inicial do segmento; (c) *Size* representando o tamanho do segmento; (d) *Rights* que mostra permissões de escrita, leitura e execução do segmento; (e) *Owner* com uma descrição do segmento; (f) *Instr.* com a quantidade de referência a instruções simuladas; e (g) *Mem* que contém a quantidade de referências à memória (*load*, *store* e *modify*). O exemplo mostrado na Tabela 4.8 reflete algumas linhas da tabela de segmentos da execução do simulador *cMIPS* após a execução de 2^{10} instruções.

Também é criado um arquivo de estatísticas no formato *CSV* contendo o histórico dos resultados da simulação. O nome do arquivo de histórico tem o prefixo "stats-" concatenado ao nome do arquivo de resultado, por exemplo: "*stats-simulation-results.txt-2549999999.txt*". O arquivo de histórico contém: (a) a configuração do *buffer* utilizado; (b) a quantidade de referências a memória; (c) total de acertos; (d) total de faltas; (e) quantidade de entradas já presentes *buffer*; (f) quantidade de entradas criadas no *buffer*; (g) quantidade de entradas removidas; e (h) quantidade de referências a memória sem segmento associado. A Tabela 4.9 mostra um exemplo de arquivo de estatísticas para o processo *lsblk*.

O arquivo de estatísticas mantém os valores de quantidade de entradas criadas, quantidade de entradas removidas e a quantidade de referências à memória sem um segmento associado com o objetivo de validar o funcionamento da*TLB* e da *SB*. Quantidade de entradas já presentes no *buffer* (*present*) conta quantas vezes o simulador tentou criar um segmento que já existe no *buffer* e deve ser sempre zero. A quantidade de entradas adicionadas (*added*) menos a quantidade de entradas removidas (*removed*) deve ser igual ou menor ao tamanho do *buffer*. A quantidade de referências a memória sem um segmento associado (*rogue*) deve ser zero porque o simulador cria segmentos simulados sempre que uma referência órfã é encontrada.

Capítulo 5

Validação

5.1 Programas utilizados

Os traços coletados são de seis aplicações reais: (a) MySQL; (b) Firefox; (c) LibreOffice; (d) QEMU; (e) Tomcat; e (f) Python. Foram simulados dois cenários: (a) no primeiro coletamos 10 bilhões de instruções durante desde o inicio da execução da aplicação, a este cenário damos o nome de simulação inicial; e (b) o segundo cenário simula uma troca de contexto, o *Trace Collector* ignora os primeiros 10 bilhões de instruções produzidas pelo *Valgrind*, então inicia a simulação com os *caches* inicialmente vazios até atingir 2 bilhões de instruções. A este segundo cenário damos o nome de simulação de troca de contexto. A simulação de troca de contexto utiliza traços com 2 bilhões de instruções porque as simulações não mostram diferenças significativas nos resultados com traços maiores.

Para os dois cenários executamos a simulação para 5 configurações de TLB e 3 configurações de SB. Utilizamos o termo *cache* para se referir a TLBs e SBs no texto porque tanto as TLBs quanto as SBs utilizam a mesma implementação de *cache* descrita no Capitulo 4, e diferem apenas na forma como os endereços paginados e endereços segmentados são armazenados na entradas da *cache*.

As configurações de TLB simuladas são: (a) totalmente associativa com 32 posições (TLB32); (b) 64 posições totalmente associativa (TLB64); (c) 128 posições totalmente associativa (TLB128); (d) 256 posições totalmente associativa (TLB256); (e) 1024 posições totalmente associativa (TLB1024). Todas as implementações de TLB utilizam um algoritmo de LRU perfeito para substituir entradas na *cache*.

As configurações TLB32, TLB64 e TLB128 são utilizadas para comparar o desempenho com *buffers* de segmentos (SB) de organização e tamanho similares. As TLBs TLB256 e TLB1024 são configurações de TLB propositalmente grandes e complexas, e representam um cenário "perfeito" no qual a complexidade e tamanho para implementação em hardware são ignoradas. Essas duas configurações representam o caso "limite" de desempenho da TLB.

As configurações de SB simuladas são: (a) 32 posições totalmente associativa (SB32); (b) 64 posições totalmente associativa (SB64); (c) 128 posições totalmente associativa (SB128). Todas as implementações de SB utilizam um algoritmos de substituição de entradas que calcula LRU perfeita para substituir elementos na *cache*.

Para cada programa simulado são apresentados os seguintes resultados: (a) um gráfico mostrando a quantidades de segmentos carregada pelo programa ao longo do tempo da simulação inicial; (b) um histograma com o tamanho dos segmentos; (c) um gráfico com a quantidade de faltas em cada um dos *caches* testados para o cenário de simulação inicial; e (d) um gráfico com a quantidade de faltas para cada um dos *caches* para o cenário simulação de troca de contexto.

Os histogramas de tamanho de segmentos do processo foram criados arredondando o tamanho dos segmentos do programa para a potência de dois mais próxima. Os segmentos "simulados" citados no Capitulo 4 estão presentes nos histogramas, esses segmentos possuem um tamanho aproximado e podem ser maiores que os segmentos originais. Portanto, a soma do tamanho dos segmentos no histograma não resulta na quantidade total de memória alocada para o programa. Os histogramas mostrados neste capítulo tem como objetivo caracterizar o tamanho dos segmentos e mostrar quais os agrupamentos mais comuns e como esses agrupamentos estão distribuídos. A medição de desempenho que está sendo avaliada para cada programa é a quantidade de faltas em cada *cache*, e não tem relação com a caracterização dos segmentos.

5.1.1 Firefox

O *Firefox* é um navegador de software livre multiplataforma desenvolvido pela *Mozilla Foundation* e atualmente é o segundo navegador para *desktop* mais utilizado no mundo em 2017, com cerca de 13% do total de acessos a *World Wide Web*. O Firefox é navegador padrão na grande maioria das distribuições *Linux* [16].

Os traços foram gerados utilizado o *Firefox* para abrir a página da *Intel* no serviço de *streaming* de vídeos *Youtube*. A página foi escolhida porque proporciona uma carga de trabalho maior ao navegador que uma página estática simples. O comando para instrumentar o *Firefox* é:

```
valgrind --log-socket=127.0.0.1 --trace-children=no
--tool=lackey --trace-mem=yes
/usr/lib/firefox/firefox http://www.youtube.com/intel
```

O *Firefox* carrega a maioria das suas bibliotecas compartilhadas no inicio da execução, como mostra a Figura 5.1. Conforme o tempo de execução passa o número de segmentos alocados para bibliotecas estabiliza. O número de segmentos totais aumenta conforme mais memória é necessária para atender as demandas do programa.

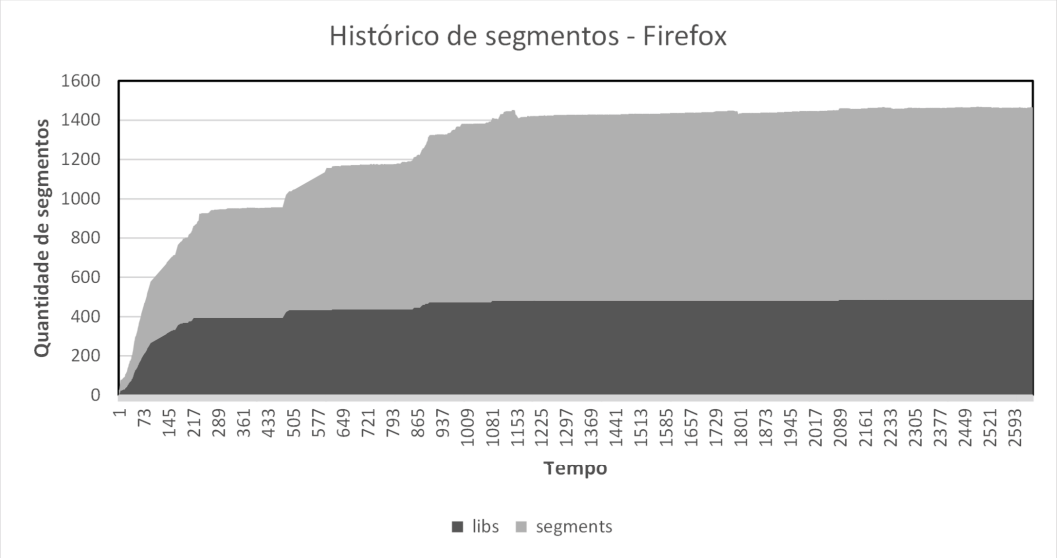

Figura 5.1: Histórico de segmentos da execução do Firefox.

O histograma de segmentos para o *Firefox* mostrado na Figura 5.2 tem 299 segmentos de 4KiB refletindo a otimização que o compilador, sistema operacional e carregador fazer para paginação. Sempre que o processo precisa de memória adicional uma nova página de 4KiB é alocada. O segundo tamanho de página mais numeroso é o de 2MiB com 147 segmentos. O

histograma também mostra segmentos grandes: (a) 44 segmentos de 8MiB; (b) 18 segmentos de 16MiB; (c) 7 segmentos de 32MiB; e (d) 4 segmentos de 64MiB.

Em um sistema baseado em segmentação pura, a tendência é o processo utilizar um menor número de segmentos, com espaços de endereçamento maiores para diminuir a quantidade de mapeamentos necessária entre memória virtual e memória física. A quantidade de segmentos de 4KiB utilizada pelo processo para mapear segmentos de dados pode ser otimizada em um sistema segmentado com um segmento de dados para cada biblioteca carregada.

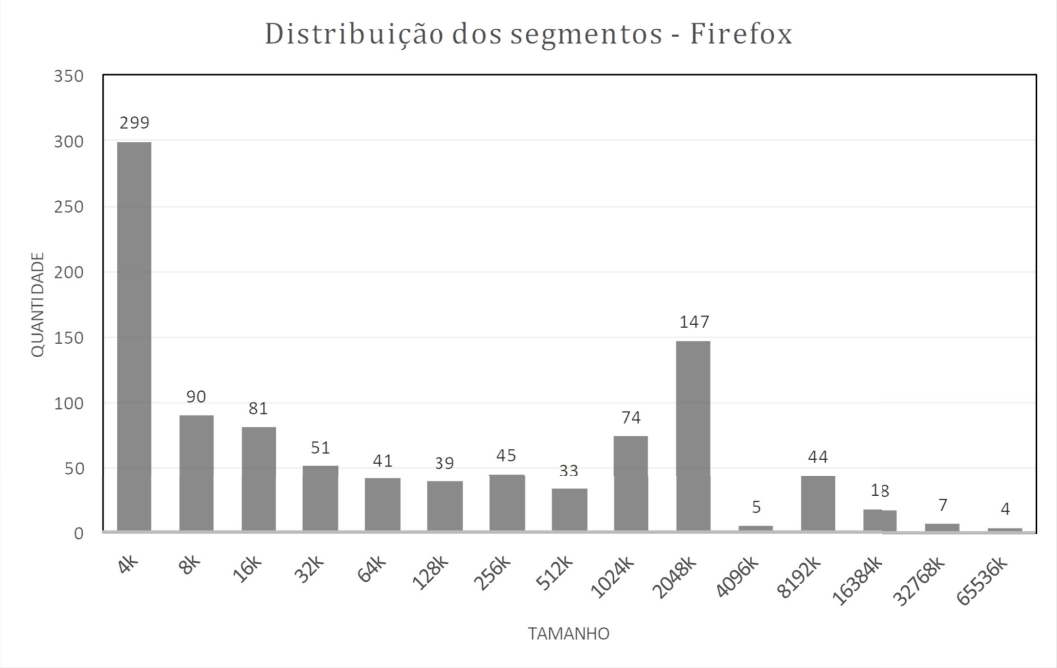

Figura 5.2: Histograma de segmentos da execução do *Firefox*.

Os resultados para a simulação inicial do *Firefox* mostrados na Figura 5.3 refletem a vantagem da segmentação, mesmo para programas que utilizam uma grande quantidade de segmentos. Nesta simulação o *Firefox* carregou 979 segmentos sendo 485 desses segmentos de bibliotecas. Os resultados mostram: (a) a SB32 é 14 vezes mais eficiente que a TLB32; (b) a SB64 é 7,6 vezes mais eficiente que TLB64; (c) a SB128 é 4,5 vezes mais eficiente que a TLB128; (d) A TLB256 é 2 vezes mais eficiente que a SB32 e apresenta praticamente o mesmo desempenho que a SB64; e (e) a TLB1024, supera todas as configurações de SB em uma ordem de grandeza, com 10^4 faltas contra 10^6 de todas as SBs testadas.

Para o *Firefox*, a SB64 tem um desempenho similar ao desempenho da TLB256. O desempenho de todas as configurações de SBs foi na mesma ordem de grandeza da TLB256, com 10^6 faltas, mostrando que mesmo a menor configuração de SB, a SB32, consegue prover um desempenho similar a uma TLB com 8 vezes mais elementos e maior complexidade em hardware.

A simulação de troca de contexto para o *Firefox*, mostrada na Figura 5.4, tem resultados de desempenho melhores para as SBs que o cenário da simulação inicial. No momento da coleta dos traços a aplicação já carregou parte das bibliotecas compartilhadas, o que faz com que o número de segmentos com endereços de memória referenciados nos traços seja menor.

Os resultados para o cenário de simulação de troca de contexto mostrados na Figura 5.4 são: (a) a SB32 é 29 vezes mais eficiente que a TLB32; (b) a SB64 é 82 vezes mais eficiente que TLB64; (c) a SB128 é 7.832 vezes mais eficiente que a TLB128; (d) A TLB256 tem 2,5 vezes mais faltas que a SB32; (e) a A SB64 é 25 vezes mais eficiente que a TLB256; (f) a A SB128 é

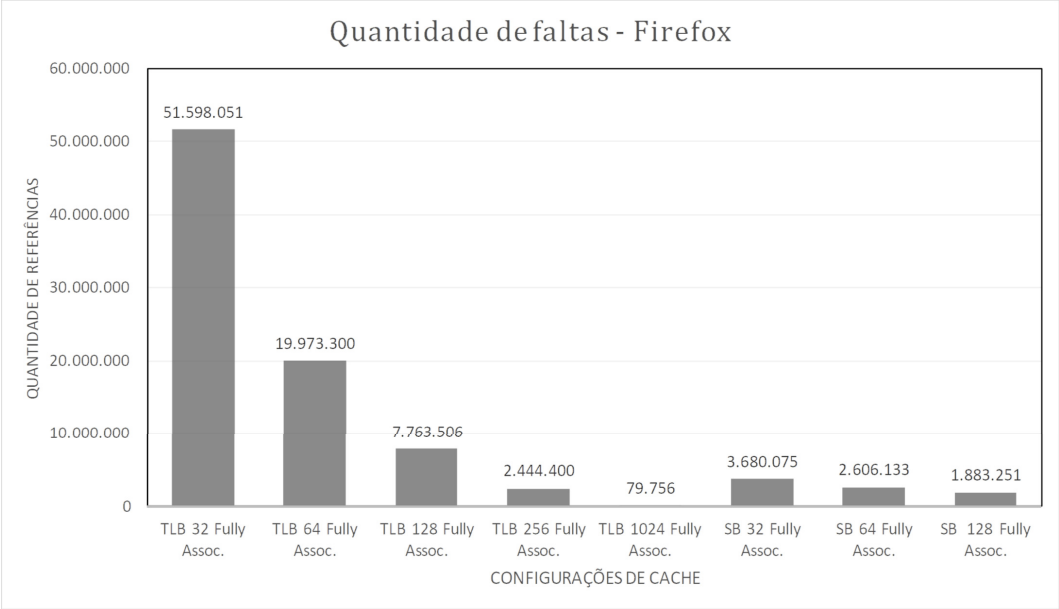

Figura 5.3: Resultados da simulação inicial para o *Firefox*.

4.450 vezes mais eficiente que a TLB256; e (f) a TLB1024 com seu tamanho e complexidade proibitivos tem 646 faltas a mais que o SB128.

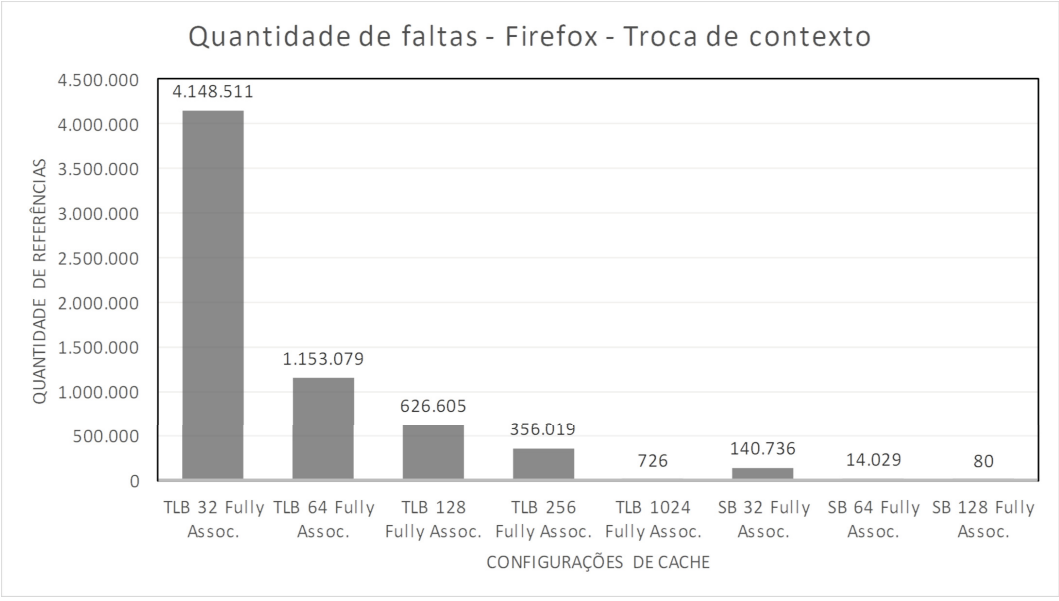

Figura 5.4: Resultados da simulação de troca de contexto para o Firefox.

A troca de contexto entre programas promovida pelo sistema operacional representa quase a totalidade do tempo de execução do *Firefox*. Uma vez iniciado o usuário utiliza o navegador durante um longo período de tempo. Neste cenário a segmentação possui um desempenho excelente. A TLB1024 possui 8 vezes o número de elementos da SB128, entretanto a SB128 tem menos faltas que a TLB1024. Implementações similares à SB128 (um cache com 128 posições totalmente associativa) são encontradas em processadores de uso geral disponíveis atualmente no mercado.

5.1.2 Libreoffice

Mantido pela *The Document Foundation*, o *Libreoffice* é um conjunto de aplicativos de software livre composto por editor de texto, planilha eletrônica, editor de apresentações, gerenciador de banco de dados, editor de formulas matemáticas e um editor de diagramas. Disponível na maioria das distribuições *Linux* e também nos sistemas operacionais da *Microsoft*, o *Libreoffice* é o conjunto de aplicativos para escritório baseado em software livre mais utilizado no mundo [19].

O *Libreoffice* foi escolhido devido a grande quantidade de segmentos de programa que utiliza durante a sua execução. Foi escolhido também para representar a classe de aplicativos de escritório. Para a simulação foram gerados traços do editor de textos (*Writer*) abrindo um arquivo no formato *Microsoft Word* com 314KiB composto por 13.999 palavras. O comando para gerar os traços está listado abaixo:

```
1 valgrind --log-socket=127.0.0.1 --trace-children=no
2 --tool=lackey --trace-mem=yes
3 /usr/lib/libreoffice/program/soffice.bin --writer test.doc
```

O histórico de segmentos para o *Libreoffice* na Figura 5.5 mostra um pico de alocação de memória no inicio da execução. Apos o pico inicial a quantidade de segmentos aumenta gradativamente durante a execução do programa, mas a quantidade de segmentos de bibliotecas não acompanha o crescimento do total de segmentos com a mesma proporção até ao final da coleta dos traços. O histórico de segmentos do *Libreoffice* é diferente do histórico de segmentos do *Firefox* mostrado na Figura 5.1, o *Firefox* carrega praticamente todos segmentos do programa antes da metade da execução enquanto o *Libreoffice* faz um carregamento progressivo das bibliotecas necessárias.

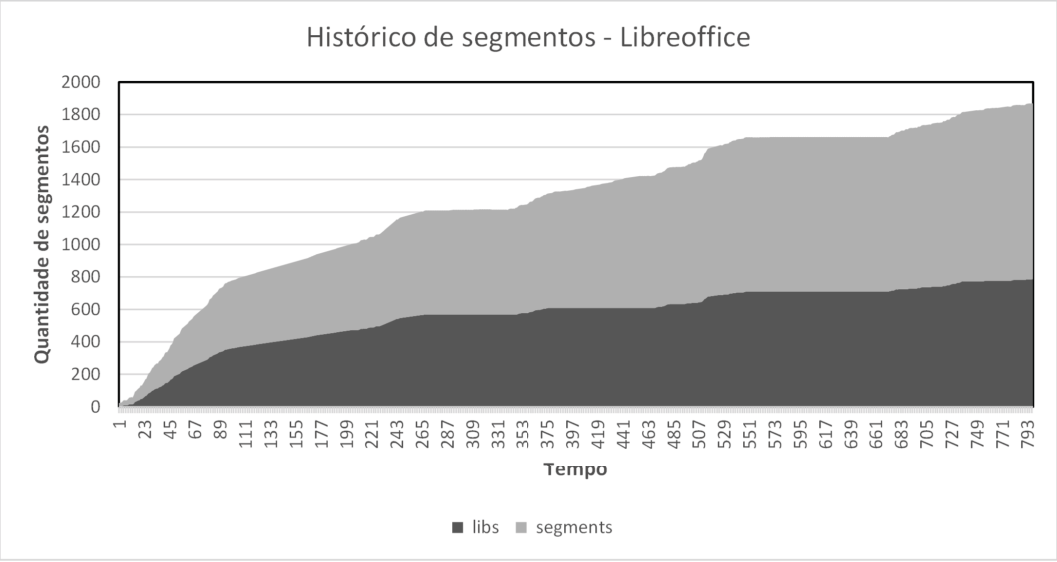

Figura 5.5: Histórico de segmentos da execução do LibreOffice.

A distribuição do tamanho dos segmentos para o processo do *Libreoffice* é mostrado no histograma da Figura 5.6. Os segmentos de 4KiB são os mais numerosos com 361 segmentos alocados. O segundo tamanho de segmento mais numeroso é o com tamanho de 2MiB com 225 segmentos, seguido por 89 segmentos de 8KiB e 68 segmentos com 256KiB. Temos 45 segmentos grandes, com tamanho igual ou maior que 8MiB. Nesta simulação o *Libreoffice* carregou 1.085 segmentos sendo 785 desses segmentos de bibliotecas.

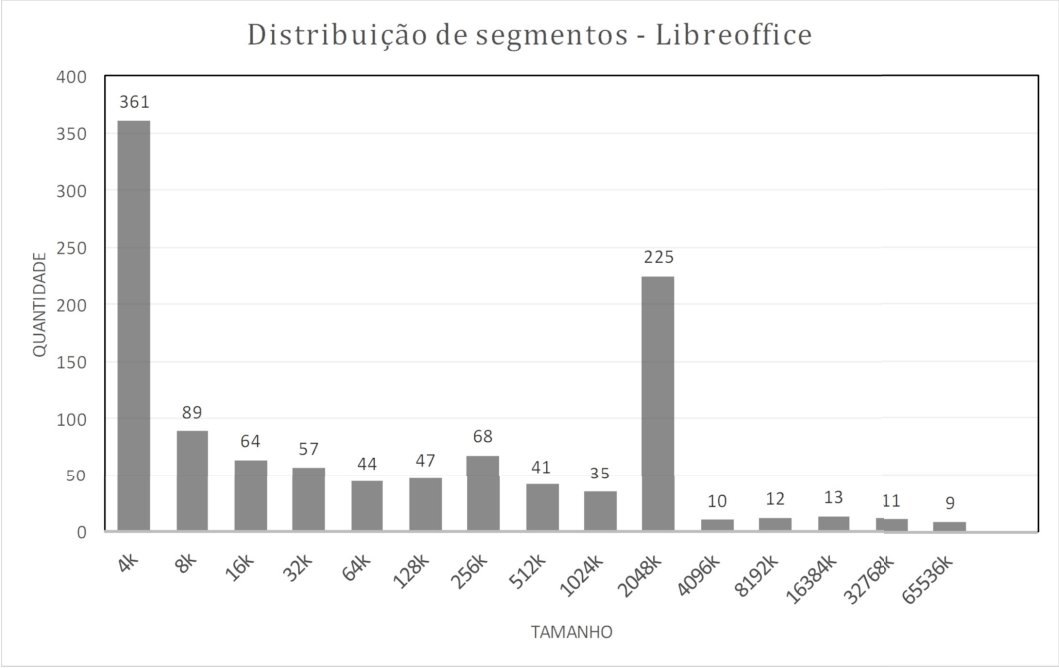

Figura 5.6: Histograma de segmentos da execução do LibreOffice.

A Figura 5.7 mostra os resultados da simulação inicial para o *Libreoffice*. A TLB32 tem 4 vezes mais faltas que a SB de tamanho e configuração iguais. A TLB64 tem uma quantidade de faltas 2 vezes maior que o SB de mesmo tamanho e configuração. Enquanto que a TL128 tem uma quantidade de faltas similar a quantidade de faltas da SB com a mesma configuração.

Os resultados das SBs mostram: (a) a SB32 é 4 vezes mais eficiente que a TLB32; (b) a SB64 é 2 vezes mais eficiente que TLB64; (c) a SB128 tem praticamente o mesmo desempenho que a TLB128; (d) A TLB256 é 19 vezes mais eficiente que a SB32, 13 vezes mais eficiente que a SB64 e 10 vezes mais eficiente que a SB128; e (e) a TLB1024 possui cerca de 10^5 faltas, e supera em desempenho todas as configurações de SB que possuem em média 10^6 faltas.

Os resultados para o *Libreoffice* mostram vantagem para o uso de segmentação, mas com margens menores que os resultados do Firefox. Estes resultados são o reflexo da grande quantidade de segmentos (1.085) presentes na execução do *Libreoffice*. Com uma quantidade maior de segmentos o desempenho das SBs diminuiu. Os traços foram coletados em um sistema otimizado para paginação. Em um sistema com segmentação pura, o compilador, ligador e carregador seriam otimizados para diminuir a quantidade de segmentos necessária para programas grandes como o *Libreoffice*, consequentemente diminuindo a quantidade de faltas.

Os resultados para a simulação de troca de contexto do *Libreoffice* mostram as SBs com um desempenho muito superior ao desempenho das TLBs. Neste cenário a aplicação já carregou boa parte das suas bibliotecas e não está alocando memória com a mesma frequência do cenário da simulação inicial. A Figura 5.8 mostra os seguintes resultados: (a) a SB32 é 12 vezes mais eficiente que a TLB32; (b) a SB64 é 25 vezes mais eficiente que TLB64; (c) a SB128 é 183 vezes mais eficiente que a TLB128; (d) A TLB256 é 2 vezes mais eficiente que a SB32; (e) a A SB64 é 3 vezes mais eficiente que a TLB256; (f) a A SB128 é 16 vezes mais eficiente que a TLB256; e (g) como observado com o *Firefox* a TLB1024 tem praticamente o mesmo desempenho da SB128, com a diferença entre as duas sendo de apenas 4.195 faltas.

O desempenho da segmentação no cenário simulação de troca de contexto tem a SB128 como destaque. Com apenas 7.319 faltas essa *cache* se iguala ao limite teórico de desempenho de uma TLB, representado pela TLB1024.

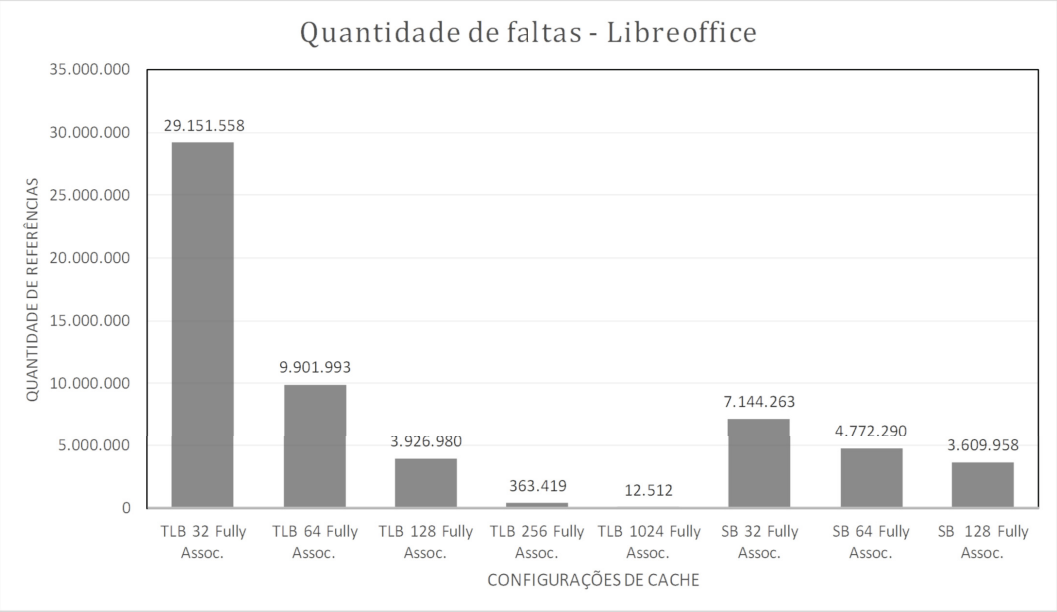

Figura 5.7: Resultados da simulação inicial para o LibreOffice.

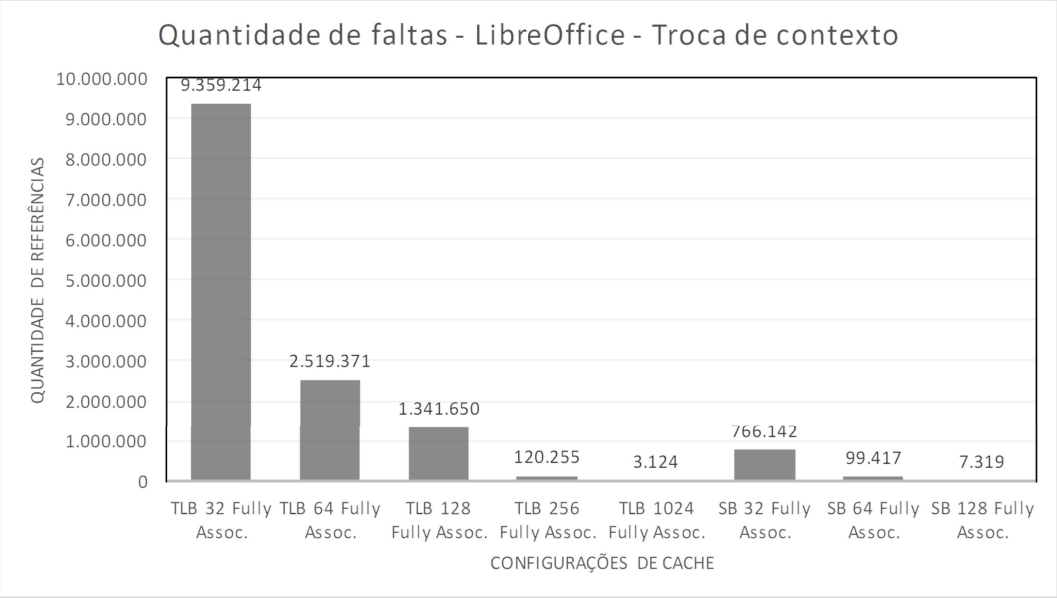

Figura 5.8: Resultados da simulação de troca de contexto para o Libreoffice.

5.1.3 MySQL utilizando Sysbench

O *MySQL* é um sistema de gerenciamento de banco de dados relacional (*RDBMS - Relational Database Management System*) *open source* mantido pela *Oracle Corporation*. O banco de dados *MySQL* é um dos principais componentes da arquitetura de desenvolvimento de aplicativos para internet baseado em software livre conhecido com o acrônimo LAMP (*Linux, Apache, MySQL, Pearl/PHP/Python*) [21]. O *MySQL* foi escolhido para representar a classe de aplicativos de banco de dados na simulação de segmentação.

Para realizar o teste no banco de dados *MySQL* utilizamos o utilitário *SysBench* disponível na maioria das distribuições *Linux*. O utilitário *SysBench* é uma ferramenta de medição de desempenho (*benchmark*) multi-plataforma e *multi-threaded*, utilizada para avaliar

parâmetros importantes para um sistema executando um banco de dados com uma alta carga de processamento [35].

Para executar transações no banco de dados *MySQL* configuramos o utilitário *SysBench* para realizar operações aleatórias em uma tabela com 1.000.000 de linhas utilizando 8 *threads* simultâneas durante 480 segundos. A tabela utilizada no banco de dados MySQL foi previamente criada utilizando o próprio utilitário *SysBench*. Foi utilizada a versão 5.7 do *MySQL* e a versão 0.4.12 do *SysBench*, disponíveis no repositório de pacotes do Ubuntu Linux 16.04 e instalados com o comando *apt-get install*. A linha de comando utilizada para iniciar o banco de dados MySQL pode ser vista abaixo:

```
valgrind --log-socket=127.0.0.1 --trace-children=no
--tool=lackey --trace-mem=yes
/usr/sbin/mysqld --basedir=/usr --datadir=/var/lib/mysql
--plugin-dir=/usr/lib/mysql/plugin
--log-error=/var/log/mysql/error.log
--pid-file=/var/run/mysqld/mysqld.pid
--socket=/var/run/mysqld/mysqld.sock --port=3306
--log-syslog=1 --log-syslog-facility=daemon
--log-syslog-tag
```

O histórico de segmentos mostrado na Figura 5.9 indica que o *MySQL* carrega poucos segmentos para bibliotecas compartilhadas. Uma vez carregadas todas as bibliotecas necessárias, o programa aloca memória de forma contínua durante todo o restante da execução do cenário da simulação inicial.

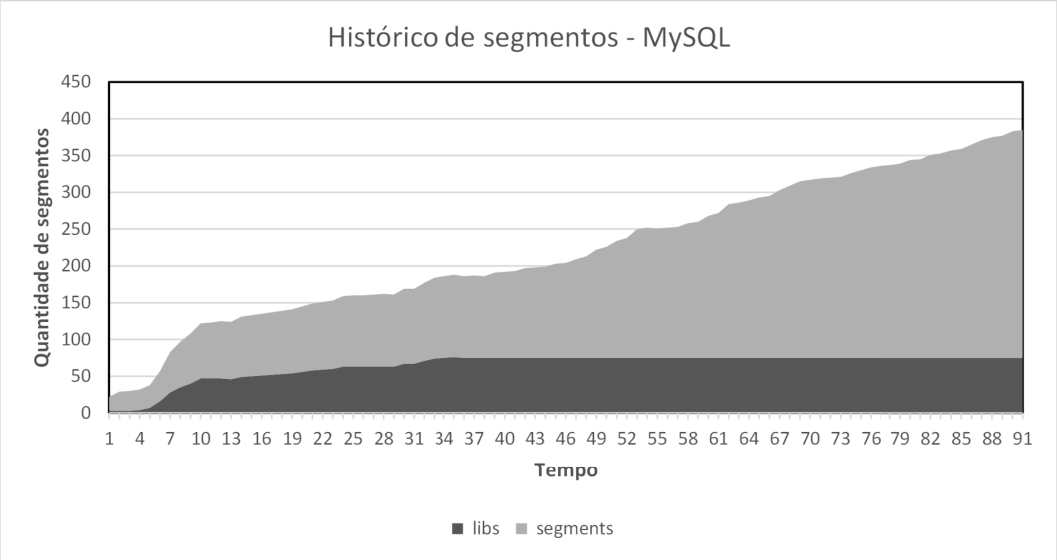

Figura 5.9: Histórico de segmentos da execução do MySQL.

O histograma com os tamanhos de segmento para o *MySQL* é mostrado na Figura 5.10. Os segmentos de 4KiB são os mais numerosos com 75 segmentos alocados. O segundo tamanho de segmento mais numeroso é o com tamanho de 1MiB com 38 segmentos, seguido por 34 segmentos de 16KiB e 31 segmentos com 256KiB. São 56 os segmentos grandes, com tamanho igual ou maior que 8MiB.

Os resultados para a simulação inicial do *MySQL* na Figura 5.11 mostram que a pequena quantidade de segmentos do *MySQL*, em comparação com o *Firefox* e *Libreoffice*, faz com que as SBs tenham um ótimo desempenho, muito acima do desempenho das TLBs de mesmo tamanho e organização. Nesta simulação o *MySQL* carrega 310 segmentos sendo 75 desses segmentos de

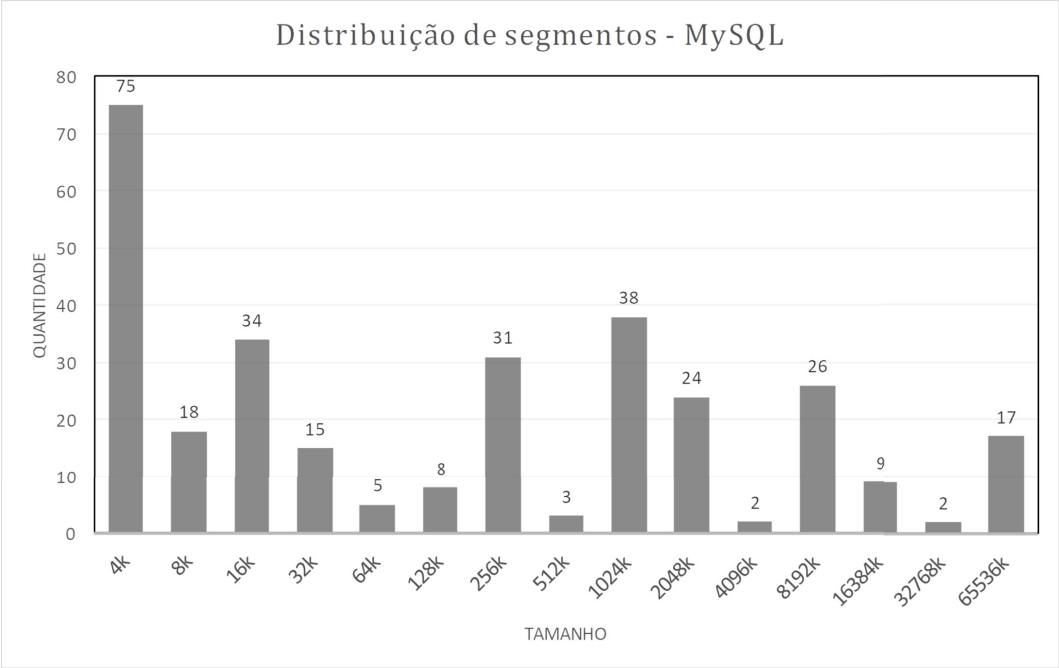

Figura 5.10: Histograma de segmentos da execução do MySQL.

bibliotecas. Os resultados mostram: (a) a SB32 é 215 vezes mais eficiente que a TLB32; (b) a SB64 é 244 vezes mais eficiente que TLB64; (c) a SB128 sofre apenas 151 faltas o que mostra que ela armazena quase a totalidade de segmentos ativos do programa; (d) A SB32 é 2,4 vezes mais eficiente que a TLB256; (e) a SB64 é 13 vezes mais eficiente que a TLB256; e (f) a SB128 mantem quase a totalidade de segmentos com referências à memória na *cache*, com 151 faltas atingindo um desempenho melhor que a TLB1024, com 9.466 faltas.

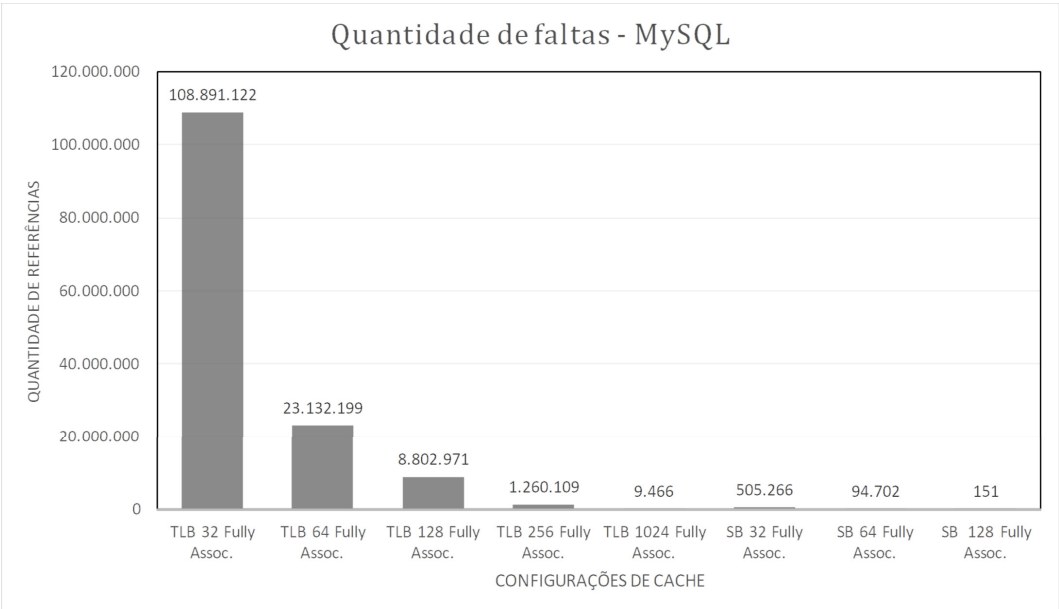

Figura 5.11: Resultados da simulação inicial para o MySQL.

Para o gráfico da simulação de troca de contexto do *MySQL*, mostrado na Figura 5.4 temos os seguintes resultados: (a) a SB32 é 257 vezes mais eficiente que a TLB32; (b) a SB64 é 341 vezes mais eficiente que TLB64; (c) a SB128 é 23.543 vezes mais eficiente que a TLB128;

(d) A SB32 é 3,5 vezes mais eficiente que a TLB256; (e) a A SB64 é 22 vezes mais eficiente que a TLB256; (f) a SB128 é a *cache* com melhor desempenho, porque armazena todos os segmentos ativos do processo dentro da *cache*, as 77 faltas são resultantes da primeira referência ao segmento, quando a *cache* está com suas entradas vazias; e (g) a TLB1024 mantém o número de faltas, na ordem de 10^3, como no cenário da simulação inicial.

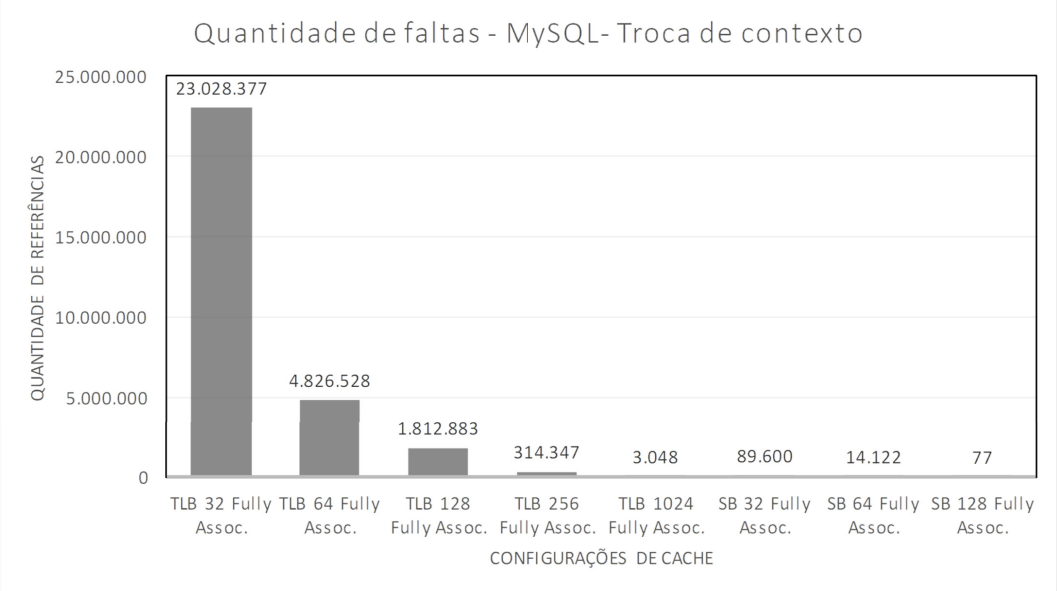

Figura 5.12: Resultados da simulação de troca de contexto para o MySQL.

O *MySQL* é um programa que utiliza poucos segmentos, quando comparado com o *Firefox* e o *Libreoffice*, mas aloca uma grande quantidade de memória para executar o *benchmark*. O gerenciamento de memória com segmentação traz um ganho de desempenho significativo para aplicações com as características do *MySQL*.

5.1.4 Python 3.5.2

Avaliamos ambiente de execução (*runtime) Python* utilizando um programa para gerenciamento e instalação de extensões para *Python* chamado *pip*. O *Python* é uma linguagem de programação de propósito geral, de alto nível, interpretada, imperativa, funcional, orientada a objetos de tipagem dinâmica e forte. Disponível em diversos sistemas operacionais, seu uso abrange software científico, serviços para *internet*, *scripts* de automação e programas *desktop* [23].

O comando *pip* é um gerenciador de pacotes escrito em *Python* e utilizado para instalar pacotes do *Python Package Index - PyPI*. Utilizamos o comando para instalar diversos pacotes úteis para o desenvolvimento de aplicativos em *Python*. O ambiente em que o e comando foi executado não possuía nenhum dos pacotes pré-instalados. O comando utilizando para gerar os traços pode ser visto abaixo:

```
1 valgrind --log-socket=127.0.0.1 --trace-children=no
2 --tool=lackey --trace-mem=yes
3 pip3 install --upgrade Prospector PyLint Pep8 Flake8
4 pydocstyle autopep8 jsonschema requests pip
```

O *Python* foi escolhido para representar uma linguagem interpretada e também porque a linguagem é bastante utilizada no meio acadêmico para pesquisas científicas.

O histórico de segmentos mostrado na Figura 5.13 indica que a quantidade de segmentos de bibliotecas acompanha o aumento no número total de segmentos durante a execução do cenário simulação inicial. O gráfico mostra que o *Python* carrega as bibliotecas compartilhadas sob demanda e aloca segmentos adicionais para suprir as necessidades de memória do programa.

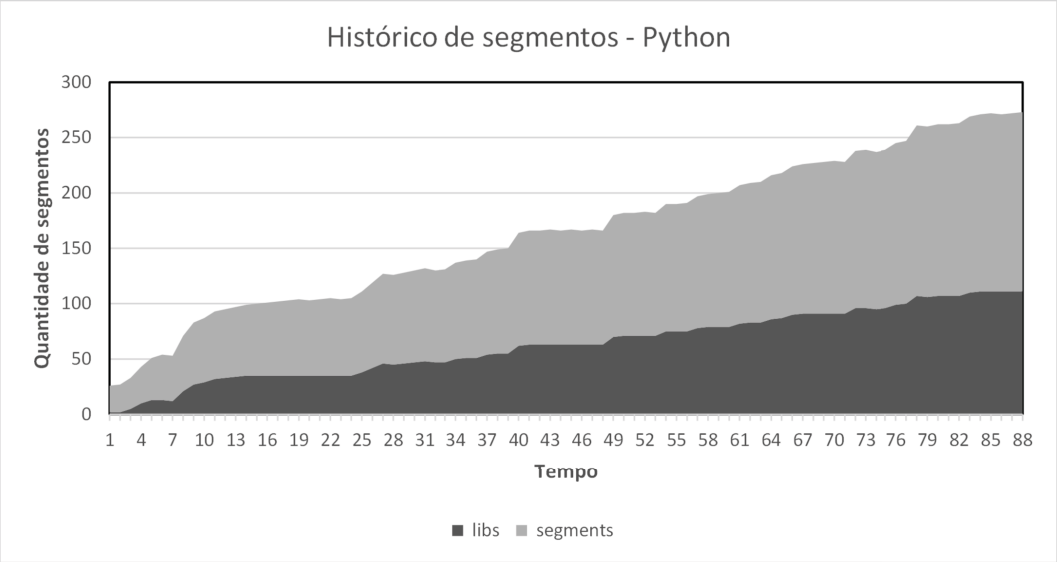

Figura 5.13: Histórico de segmentos da execução do Python.

O histograma com os tamanhos de segmento para o *Python* é mostrado na Figura 5.14. Os segmentos de 4KiB são os mais numerosos com 50 segmentos alocados. O segundo tamanho de segmento mais numeroso é o com tamanho de 2MiB com 34 segmentos, seguido por 12 segmentos de 8KiB e 11 segmentos com 16KiB. São 8 segmentos grandes, com tamanho igual ou maior que 8MiB. Pelo histograma nota-se claramente que a utilização de memória do processo é menor que os programas *Firefox*, *Libreoffice* e *MySQL*.

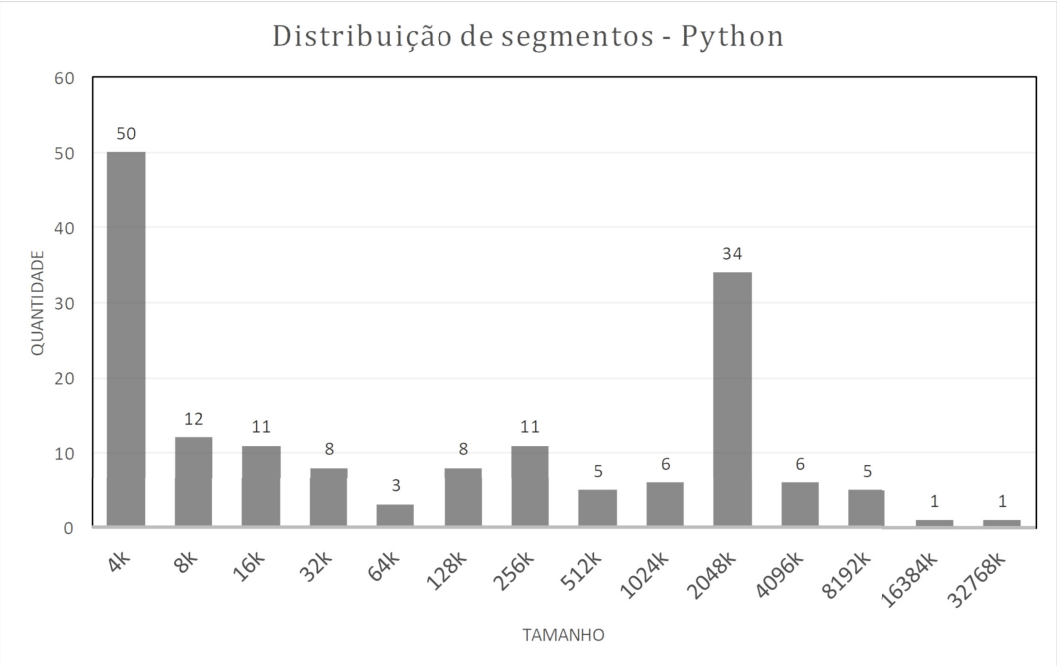

Figura 5.14: Histograma de segmentos da execução do Python.

Os resultados para a simulação inicial do *Python* mostrados na Figura 5.15, refletem a diferença de desempenho entre paginação e segmentação para programas que utilizam poucos segmentos. O *Python* carrega 162 segmentos sendo 111 desses segmentos de bibliotecas. Os resultados mostram: (a) a SB32 é 8.366 vezes mais eficiente que a TLB32; (b) a SB64 e a SB128 possuem menos que 500 faltas (10^2) enquanto a TLB64 tem 27.863.360 faltas e a TLB128 tem 4.571.311 faltas. A diferença de desempenho mostra o potencial de ganho para um sistema otimizado para utilizar segmentação; (c) A TLB256 e a TLB1024 são as únicas configurações de TLB que possuem a mesma ordem de grandeza de faltas. Mesmo a TLB1024 sofre 631 faltas a mais que a SB128.

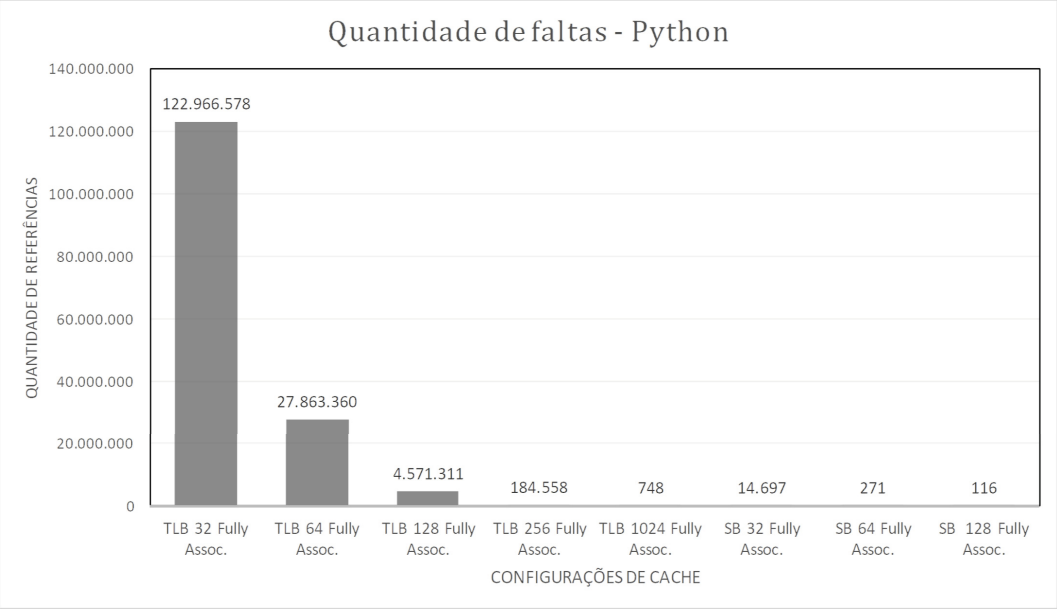

Figura 5.15: Resultados da simulação inicial para o Python.

Os resultados para o cenário de troca de contexto contidos na Figura 5.16 mostram os seguintes resultados: (a) A SB32 tem apenas 1.612 faltas contra mais de 26 milhões de faltas da TLB32; (b) A SB64 e a SB128 tem o mesmo número de faltas, porque as duas armazenaram todas os 61 segmentos ativos do *Python*, para o cenário da simulação, o que torna a comparação com a TLB64 e TLB128 irrelevante; (c) a TLB256 sofre 77.912 faltas o que é 92 vezes mais eficiente que a TLB64, mas longe a eficiência da SB32 com apenas 1.612 faltas; e (d) a TLB1024 é apenas 2,5 vezes mais eficiente que a SB32, o que parece ser uma diferença considerável, mas em termos absolutos representa uma diferença de 991 faltas.

5.1.5 QEMU 2.8.1.1 executando FreeDOS 1.2

O aplicativo de software livre *QEMU* é uma plataforma para virtualização de *hardware*. O *QEMU* faz emulação de diversas *CPUs* utilizando *binary translation* e fornece um conjunto de modelos de dispositivos de modo a executar varios sistemas operacionais de forma nativa, sem a necessidade de alterações. Disponível para diversos sitemas operacionais o *QEMU* emula varias plataformas: *x86*, *PowerPC*, *ARM*, *SPARC*, *MicroBlaze*, *MIPS*, *OpenRISC* [24]. O *QEMU* foi escolhido para representar a classe de aplicativos voltados para virtualização e emulação de sistemas.

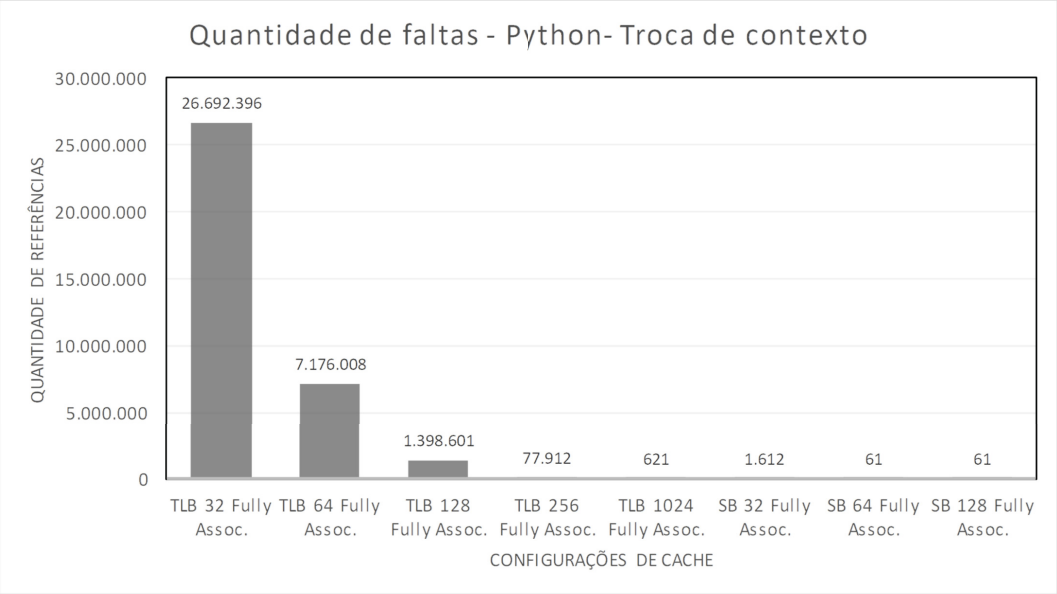

Figura 5.16: Resultados da simulação de troca de contexto para o Python.

O sistema operacional *FreeDOS* foi desenvolvido com o objetivo de ser uma alterativa baseada em *software* livre para o sistema operacional *DOS*, voltado para a execução de programas legados e suportar sistemas embarcados [17].

Configuramos o *QEMU* para emular uma máquina *Intel x86* com 32 *Mbytes* de memória *RAM*, uma imagem de um disco rígido virtual (*hdd.img)* e um drive de *CD-ROM*. Com a máquina virtual do *QEMU* criada, foi instalado o *FreeDOS* 1.2 no disco rígido virtual.

Os traços foram gerados a partir da execução do *QEMU* com um console texto. Os traços foram coletados durante a inicialização (*boot*) do *FreeDOS*, apresentação do *prompt* "c:" no terminal e execução de um programa de teste de memória pelo *FreeDOS* . O comando utilizado para que o *Valgrind* gerar os traços pode ser visto abaixo:

```
1 valgrind --log-socket=127.0.0.1 --trace-children=no
2 --tool=lackey --trace-mem=yes
3 /opt/qemu-intel/bin/qemu-system-i386 -sdl -m 32 -hda hdd.img -boot c
```

O histograma do QEMU, mostrado na Figura 5.17 possui três fases distintas: (a) uma carga inicial de segmentos de bibliotecas; (b) a criação de segmentos de biblioteca continua mas com um ritmo mais lento; e (c) na parte final a quantidade de segmentos de bibliotecas se mantem constante, mas a alocação total de segmentos aumenta.

O histograma com os tamanhos de segmento para o *QEMU* é mostrado na Figura 5.18. Os segmentos de 4KiB são os mais numerosos com 327 segmentos alocados. O segundo tamanho de segmento mais numeroso é o com tamanho de 2MiB com 170 segmentos, seguido por 84 segmentos de 1MiB e 50 segmentos com 8KiB. São 20 os segmentos grandes, com tamanho igual ou maior que 8MiB.

4

Os resultados para a simulação inicial do *QEMU* mostrados na Figura 5.19 indicam um empate entre as SB, todas com 10^6 faltas. O *QEMU* carrega 845 segmentos sendo 601 desses segmentos de bibliotecas. Os resultados mostram: (a) a SB32 é 18 vezes mais eficiente que a TLB32; (b) a SB64 é 2,6 vezes mais eficiente que TLB64; (c) a TLB128 é 2,2 vezes mais eficiente que a SB128; e (d) A TLB256 e a TLB1024 possuem um desempenho muito bom e dentro da mesma ordem de grandeza (10^3). A diferença entre elas é de 3.731 faltas.

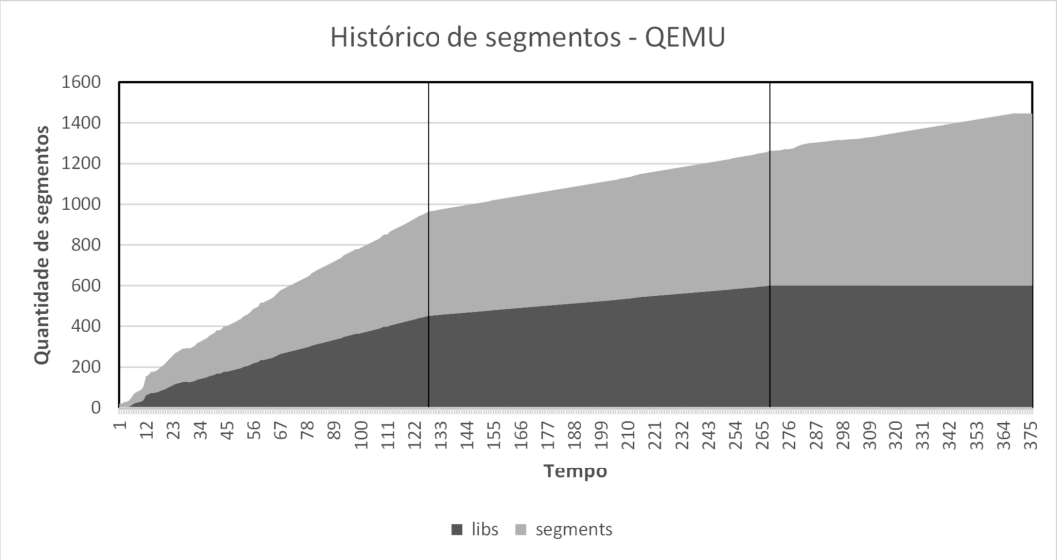

Figura 5.17: Histórico de segmentos da execução do QEMU.

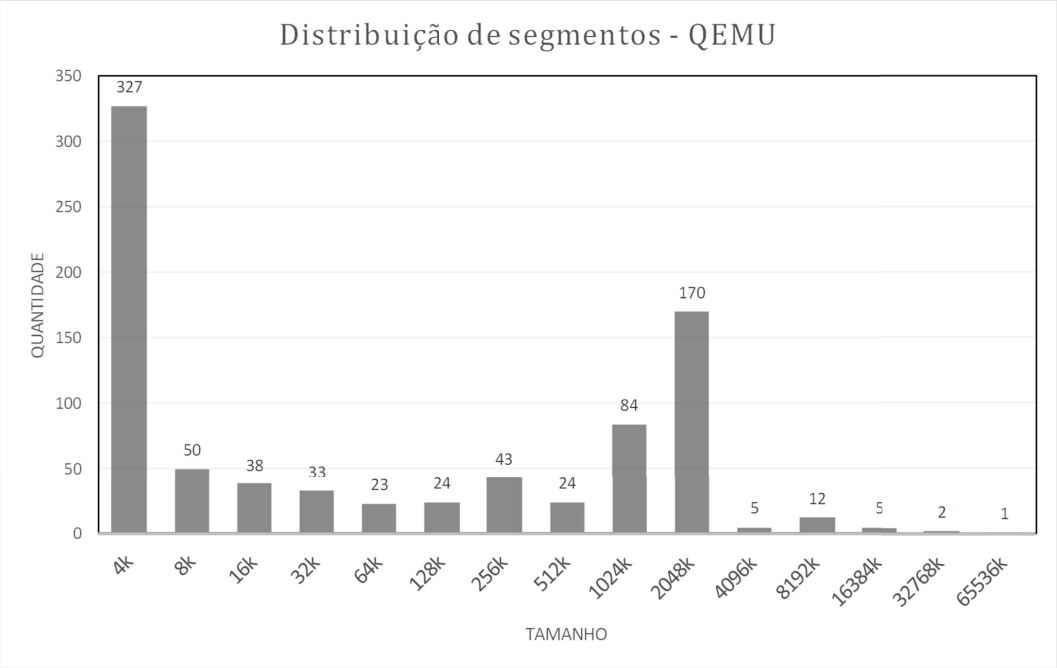

Figura 5.18: Histograma de segmentos da execução do QEMU.

Os resultados para o cenário de simulação de troca de contexto contidos na Figura 5.20 são: (a) a SB32 é 17 vezes mais eficiente que a TLB32; (b) a SB64 e a SB128 sofrem apenas 46 faltas, o que demonstra que armazenam todos os segmentos com referências à memória do traço; e (c) A TLB256 e a TLB1024 sofrem 167 faltas, o que indica que a execução dos traços utiliza apenas 167 páginas, e as faltas foram devido a instalação das páginas na *cache*.

5.1.6 Tomcat 8.0.43

A linguagem *Java* é utilizada de forma abrangente na *Internet* para prover serviços e aplicações *online*. Avaliamos a execução do *Servlet Container* e servidor *web Tomcat* utilizando a implementação *Java* em software livre *OpenJDK 1.8.0*.

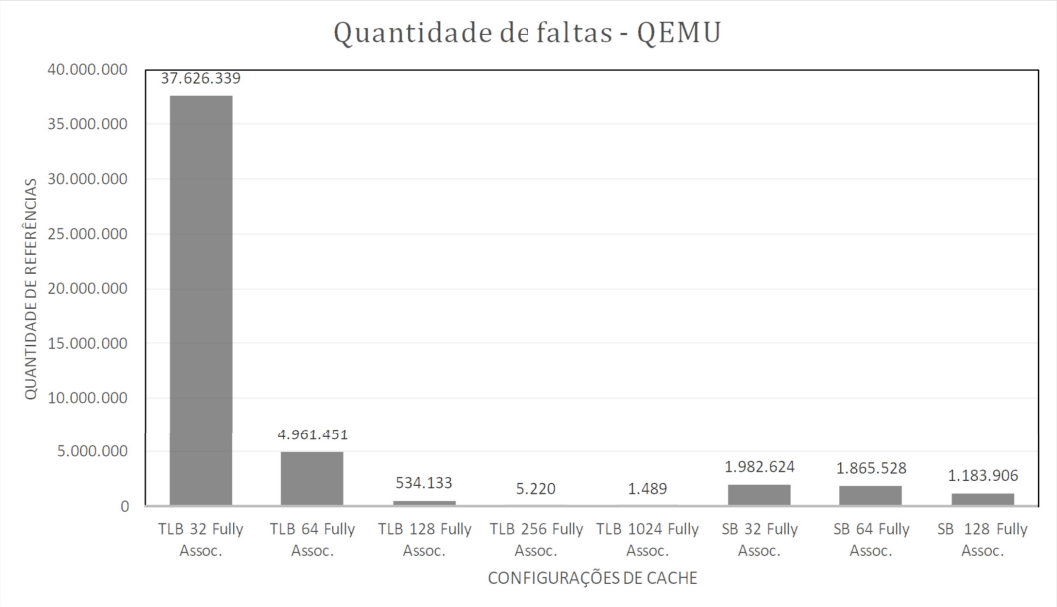

Figura 5.19: Resultados da simulação inicial para o QEMU.

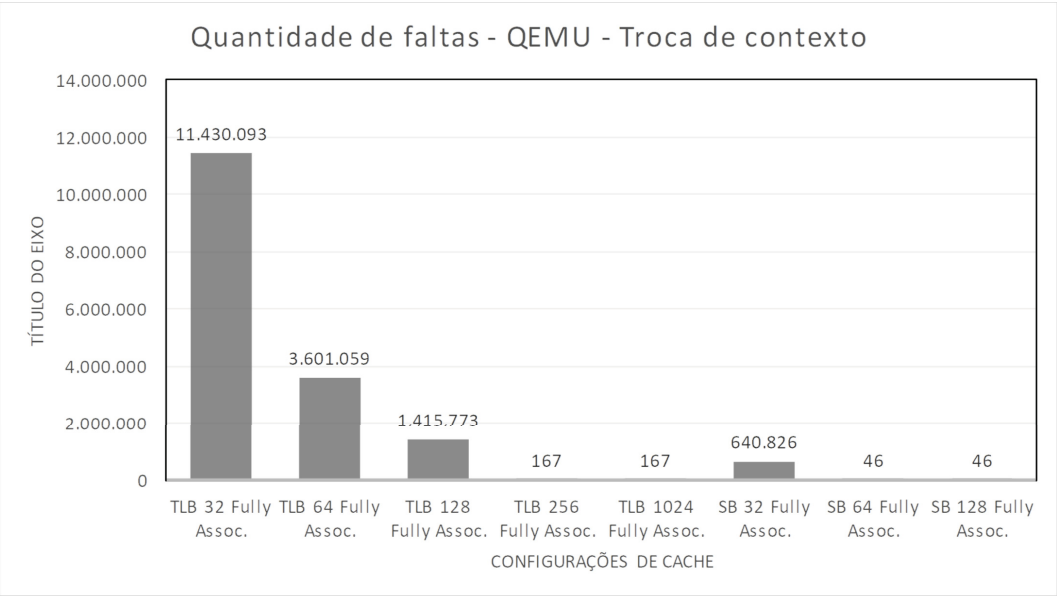

Figura 5.20: Resultados da simulação de troca de contexto para o QEMU.

Desenvolvida na década de 1990, a linguagem *Java* é interpretada, de uso geral, fortemente tipada e orientada a objetos. Um programa escrito em *Java* é compilado para um *bytecode* que é executado em uma máquina virtual Java (JVM). A JVM implementa compilação dinâmica de *bytecode* em código nativo da plataforma, utilizando um compilador *just in time* (*JIT compiler*). Com o *JIT*, o desempenho dos programas executados aumenta significativamente em comparação a uma JVM sem a funcionalidade habilitada [13].

Mantido e desenvolvido pela *Apache Foundation* o servidor *web* e *servlet container* Tomcat é utilizado para hospedar páginas *HTML*, *servlets Java* e executar páginas *web* dinâmicas escritas em *JSP (Java Server Pages)*. Criado inicialmente como uma implementação de referência para a especificação de *servlets*, o *Tomcat* é utilizado hoje para desenvolvimento e hospedagem de diversos serviços, páginas e aplicativos de internet baseados em *Java*[14].

Os traços da execução do *Apache Tomcat* foram gerados com o comando abaixo. Um navegador web foi utilizado para executar as páginas do contexto */examples/jsp/* do *Tomcat* que testam as funcionalidades de páginas *JSP*. Os exemplos foram executados em ordem (de cima para baixo) clicando no link *execute*. Quando a página de exemplo é exibida voltamos a página anterior com o botão voltar no navegador e o próximo exemplo é executado.

```
valgrind --log-socket=127.0.0.1 --trace-children=no
--tool=lackey --trace-mem=yes
/usr/bin/java
-Djava.util.logging.config.file=
/home/lauri/eclipse/apache-tomcat-8.0.43/conf/logging.properties
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
-Djdk.tls.ephemeralDHKeySize=2048
-Djava.protocol.handler.pkgs=org.apache.catalina.webresources
-Djava.endorsed.dirs=/home/lauri/eclipse/apache-tomcat-8.0.43/endorsed
-classpath /home/lauri/eclipse/apache-tomcat-8.0.43/bin/bootstrap.jar:
/home/lauri/eclipse/apache-tomcat-8.0.43/bin/tomcat-juli.jar
-Dcatalina.base=/home/lauri/eclipse/apache-tomcat-8.0.43
-Dcatalina.home=/home/lauri/eclipse/apache-tomcat-8.0.43
-Djava.io.tmpdir=/home/lauri/eclipse/apache-tomcat-8.0.43/temp
org.apache.catalina.startup.Bootstrap start
```

Os traços gerados são da máquina virtual *Java* (*JVM*) executando o *Tomcat*. Escolhemos o *Tomcat* porque ele é servidor de aplicações e serviços para a internet. Servidores de aplicações em *Java* tendem a utilizar grandes quantidades de memória e recursos computacionais.

O histórico de segmentos para o *Tomcat* mostrado na Figura 5.21, evidencia a grande quantidade de segmentos carregados pelo programa, e consequentemente a quantidade de memória alocada. Durante o início da execução os principais segmentos de bibliotecas compartilhadas são carregados e se mantém estáveis por cerca de 74 segmentos. Em seguida o histograma apresenta um pico com 90 segmentos, mantendo essa quantidade de segmentos de bibliotecas compartilhadas ativos por praticamente todo o restante da execução do programa. Em contrapartida, o total de segmentos cresce rapidamente no inicio da execução até atingir cerca de 430 segmentos, e mantém esse número total de segmentos por aproximadamente um terço da execução. Na parte final da execução, o total de segmentos sobe novamente atingindo um pico de 516 segmentos.

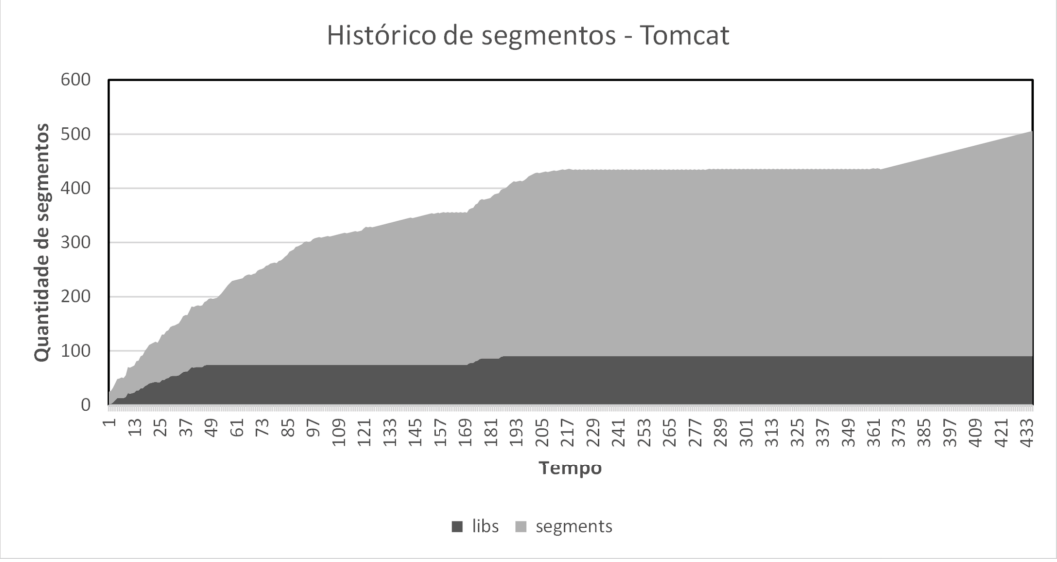

Figura 5.21: Histórico de segmentos da execução do Tomcat.

O histograma do *Tomcat* difere de todos os outros programas. Há um maior espaçamento na quantidade de segmentos em cada tamanho analisado. O tamanho mais numeroso é o de 16KiB com 66 segmentos, seguido pelo de 4KiB com 65 segmentos e o de 8KiB com 54 segmentos. Os três primeiro segmentos mais populares representam 44% do total de segmentos do programa. Segmentos maiores que 8MiB somam 47. O histograma mostra 21 de segmentos de 64MiB e 16 segmentos de 16MiB, segmentos grandes e que ocupam uma quantidade considerável de memória.

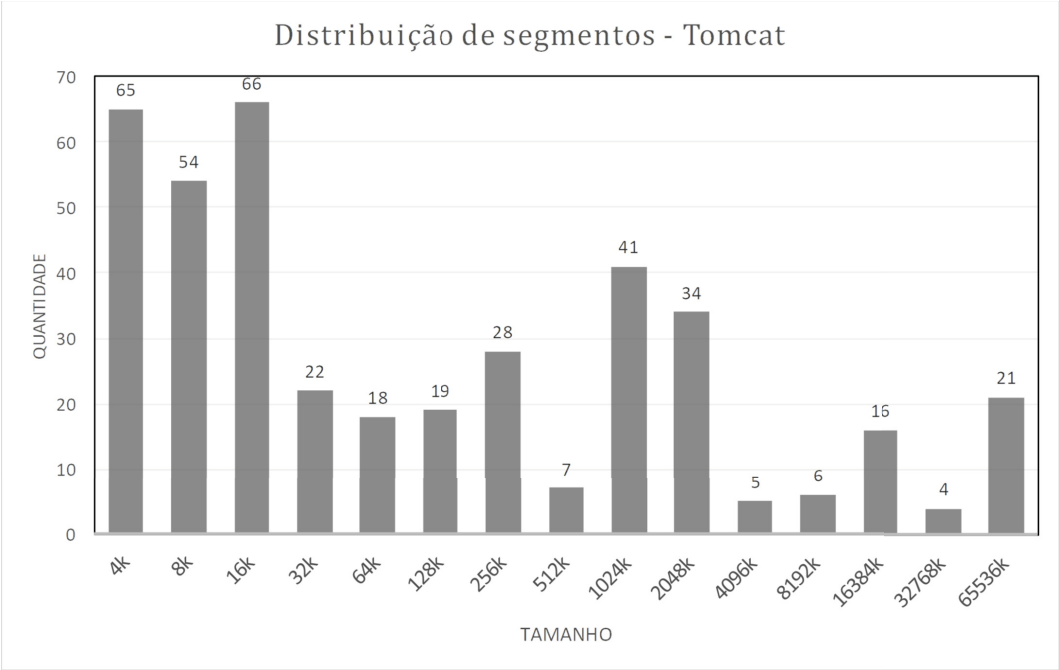

Figura 5.22: Histograma de segmentos da execução do Tomcat.

Os resultados para a simulação inicial do *Tomcat*, mostrados na figura 5.23 refletem a vantagem da segmentação para programas como o *Tomcat*. Nesta simulação, o *Tomcat* carrega 416 segmentos sendo 90 desses segmentos de bibliotecas. Os resultados mostram: (a) a SB32 é 225 vezes mais eficiente que a TLB32; (b) a SB64 é 2.027 vezes mais eficiente que TLB64; (c) a SB128 é 6.948 vezes mais eficiente que a TLB128; (d) A SB32 é 4 vezes mais eficiente que a TLB256; e (e) a TLB1024, sofre 2.319 faltas a mais que a SB64.

Os resultados para o cenário de simulação de troca de contexto mostrados na Figura 5.24 são: (a) a SB32 é 162 vezes mais eficiente que a TLB32; (b) a SB64 é 2.070 vezes mais eficiente que TLB64; (c) a SB128 é 4.630 vezes mais eficiente que a TLB128; (d) A SB32 é 4,8 vezes mais eficiente que a TLB256; (e) a A SB64 sofre 903 faltas, ou 2,5 vezes menos faltas que a TLB1024; e (f) a SB128 sofre apenas 122 faltas, o que indica que houve apenas uma falta a mais que o tamanho total da cache; e (f) a TLB1024 tem desempenho pior do que as SB64 e SB128.

5.2 Resultados consolidados

Os resultados individuais mostram uma quantidade de faltas significativamente menor para as SBs em comparação com as TLBs. A Tabela 5.1 mostra uma linha para cada programa executado, as colunas contém as seguintes informações: (a) a quantidade total de segmentos (*segs*); (b) quantos desses segmentos pertencem à bibliotecas compartilhadas (*libs*); (c) quantos segmentos possuem pelo menos uma referência ao seu espaço de endereçamento (*used*); e (d) a

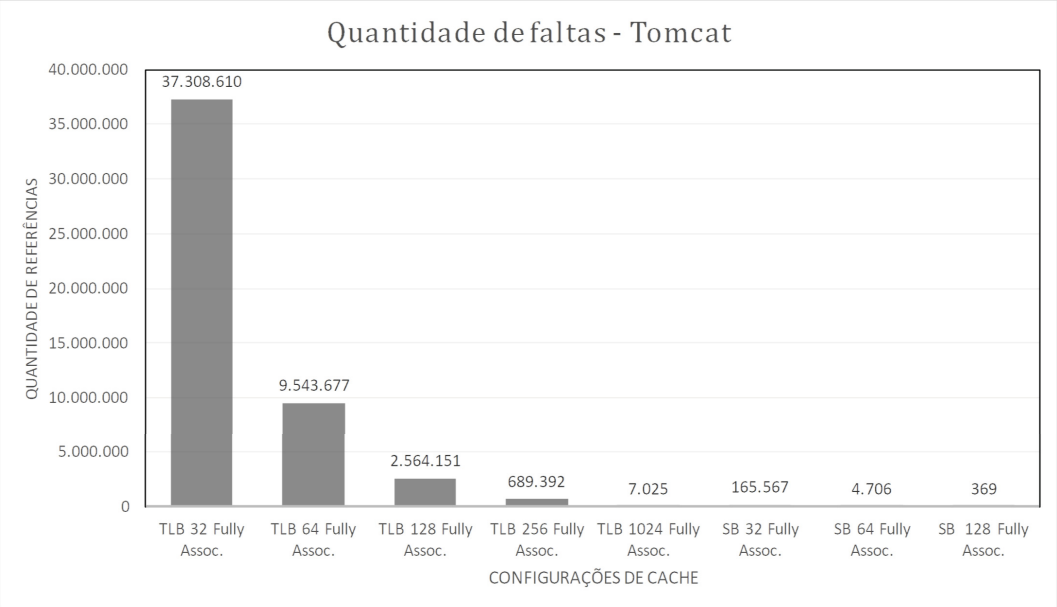

Figura 5.23: Resultados da simulação inicial para o Tomcat.

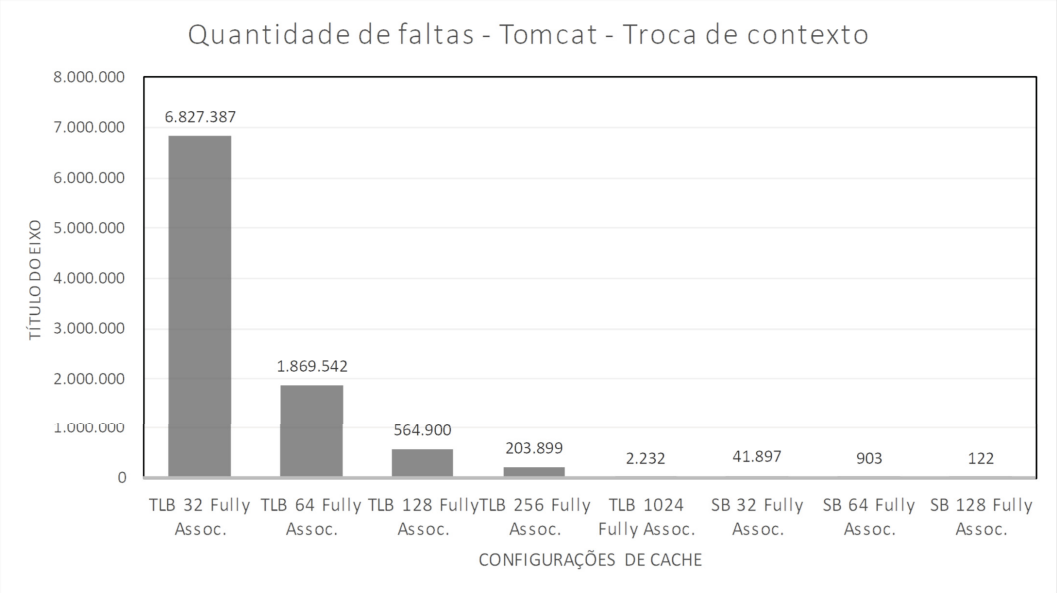

Figura 5.24: Resultados da simulação de troca de contexto para o Tomcat.

porcentagem de segmentos com referências à memória com relação ao total de segmentos (*used ratio*). Damos o nome de *segmento ativo* para o segmento que possui ao menos uma referência ao seu espaço de endereçamento presente nos traços.

O *LibreOffice* é o programa com o maior número de segmentos, 1.085 no total. O *LibreOffice* também possui a maior porcentagem de segmentos ativos, com 76% do segmentos carregados com pelo menos uma referência no seu espaço de endereçamento. O *Firefox* é o segundo processo em quantidade de segmentos totais com 979. Mas, diferente do *LibreOffice*, os traços do *Firefox* referenciam apenas 67% dos segmentos do processo. O *QEMU* é o terceiro processo com maior quantidade de segmentos totais, com 845 segmentos e 67% de segmentos ativos. Os três processos com maior número de segmentos (*LibreOffice*, *Firefox* e *QEMU*) possuem pelo menos o dobro de segmentos totais que os outros processos simulados e possuem

em média 676 segmentos ativos. Chamamos o sub-conjunto de programas composto por *Firefox*, *QEMU* e *LibreOffice* de *pbig*.

O quarto programa que com maior número de segmentos mostrado na Figura 5.1 é o *Tomcat* com 416 segmentos totais e 65% desses segmentos são segmentos ativos. O *MySQL* possui um total de 310 segmentos com 48% de segmentos ativos. O programa com o menor número de segmentos é o *Python* com 162 segmentos e 71% desses segmentos são segmentos ativos. O *MySQL*, *Python* e *Tomcat* possuem 179 segmentos ativos em média. Chamamos o sub-conjunto de programas composto por *Tomcat*, *Python* e *MySQL* de *psmall*.

Tabela 5.1: Utilização dos segmentos.

application	segs	libs	used	used ratio
LibreOffice	1.085	785	830	76%
Firefox	979	485	633	64%
QEMU	845	601	567	67%
Tomcat	416	90	272	65%
MySQL	310	75	151	48%
Python	162	111	116	71%

Nas seções abaixo são mostrados os resultados consolidados de todos os testes para cada configuração de *cache* utilizada. Cada linha da tabela é um programa simulado. As três primeiras colunas contém os resultados para a TLB: (a) total de acertos (*Hit*); (b) total de faltas (*Miss*); e (c) porcentagem de faltas *Miss %*). As três últimas colunas mostram os resultados da SB com: (a) total de acertos (*Hit*); (b) total de faltas (*Miss*); e (c) porcentagem de faltas (*Miss %*).

5.2.1 Cache com 32 posições, completamente associativo

A Tabela 5.2 mostra que a porcentagem média de faltas na *TLB* é de 0,4523% e a porcentagem média de faltas na SB é de 0,0157% na simulação inicial. A diferença absoluta da média de faltas entre a TLB e a SB é de 62.341.628 faltas. O desempenho da SB32 para os programas *MySQL* com apenas 0,0033% de faltas, o *Pyhton* com apenas 0,0001% de faltas e o *Tomcat* com 0,0011% de faltas são os programas que mais contribuem para que a quantidade média de faltas da SB seja baixa.

Os programas *psmall* possuem uma quantidade média de segmentos ativos pequena, e isso faz com que a segmentação seja eficiente mesmo com uma *cache* de tamanho reduzido como a SB32. O desempenho da TLB também é interessante, podemos observar que nenhuma das simulações atinge 1% de faltas. Isso se deve, possivelmente, pelo uso do LRU perfeito para substituição de entradas na *cache*.

Os resultados consolidados para a simulação de troca de contexto mostrados na Tabela 5.3 mostram uma quantidade média de faltas de 0,0097% para a SB32 e 0,4728% para a TLB32. A diferença do número de faltas entre a TLB32 e a SB32 na simulação de troca de contexto é de 13.300.861 faltas. Da mesma grandeza do número de faltas na simulação inicial.

A média de faltas na TLB32 aumenta em 0,0257 pontos percentuais em relação à simulação inicial, enquanto a média de faltas para da SB32 diminuiu em 0,006 pontos percentuais em relação a simulação inicial. Os programas *psmall* sofre poucas faltas, devido a quantidade menor de segmentos ativos presentes no processo, o que é potencializado com a simulação de troca de contexto, na qual o processo já carregou e inicializou suas bibliotecas compartilhadas, e a necessidade de alocar novos segmentos é pequena.

Tabela 5.2: Comparativo de desempenho para *cache* completamente associativo de 32 posições.

Program	TLB 32, fully assoc			SB 32, fully assoc.		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	14.597.379.155	51.598.051	0,3535%	14.645.297.131	3.680.075	0,0251%
Libre	13.487.897.048	29.151.558	0,2161%	13.509.904.343	7.144.263	0,0529%
MySQL	15.163.039.318	108.891.122	0,7181%	15.271.425.174	505.266	0,0033%
Python	14.375.835.702	122.966.578	0,8554%	14.498.787.583	14.697	0,0001%
QEMU	13.580.033.508	37.626.339	0,2771%	13.615.677.223	1.982.624	0,0146%
Tomcat	14.474.783.134	37.308.610	0,2577%	14.511.926.177	165.567	0,0011%
Average	14.279.827.978	64.590.376	0,4523%	14.342.169.605	2.248.749	0,0157%

Tabela 5.3: Troca de contexto - *Cache* com 32 posições, completamente associativo.

Program	TLB 32 fully assoc.			SB 32 fully assoc.		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	2.993.087.444	4.148.511	0,1386%	2.997.095.219	140.736	0,0047%
Libre	2.695.753.124	9.359.214	0,3472%	2.704.346.196	766.142	0,0283%
MySQL	3.010.113.983	23.028.377	0,7650%	3.033.052.760	89.600	0,0030%
Python	2.842.063.070	26.692.396	0,9392%	2.868.753.854	1.612	0,0001%
QEMU	2.772.041.878	11.430.093	0,4123%	2.782.831.145	640.826	0,0230%
Tomcat	2.919.926.819	6.827.387	0,2338%	2.926.712.309	41.897	0,0014%
Average	2.872.164.386	13.580.996	0,4728%	2.885.465.247	280.136	0,0097%

5.2.2 Cache com 64 posições, totalmente associativo

Os resultados da simulação inicial encontrados na Tabela 5.4 mostram que a porcentagem média de faltas na TLB64 é de 0,1109% e a porcentagem média de faltas na SB64 é 0,0109%, uma diferença de 0,0101 pontos percentuais entre as duas *caches*. Em números absolutos a TLB64 tem 14.338.725 faltas a mais que a SB64, uma diferença entre a TLB e SB na ordem de grandeza de $10^7 x$ faltas.

O desempenho para os programas *psmall* é eficiente na SB64, que é maior e pode acomodar mais segmentos. O *Python* sofre apenas 271 faltas, mostrando que a SB64 mantém grande parte dos segmentos da simulação inicial disponíveis na *cache* durante a inicialização do programa, quando as bibliotecas compartilhadas são carregadas e inicializadas. Entre os programas *pbig*, o *Firefox* é o que possui o maior percentual de faltas na SB64 com 0,0178%. A TLB64 para o *Firefox* tem um percentual de faltas de 0,1365%, uma diferença de 0,1187 pontos percentuais entre as duas caches que corresponde a 17.367.167 faltas a mais na TLB64.

Os resultados da simulação de troca de contexto para a TLB64 e SB64 estão na Tabela 5.5. Os resultados consolidados mostram uma quantidade média de faltas de 0,1223% para a TLB32 e 0,0007% para a SB32. A diferença do número de faltas entre a TLB32 e a SB32 na simulação de troca de contexto é de 3.502.835 faltas. A média de faltas na TLB64 aumenta de 0,1109% na simulação inicial para 0,1223% na simulação de troca de contexto. A TLB32 também apresenta uma média de faltas maior na simulação de troca de contexto do que na simulação inicial.

A SB64 tem uma média de faltas menor na simulação de troca de contexto comparada com a simulação inicial, de 0,0101 pontos percentuais. Os programas *psmall* possuem uma quantidade muito pequena de faltas. O que chama a atenção na Tabela 5.5 é o desempenho dos

Tabela 5.4: Comparativo de desempenho de para *cache* completamente associativo de 64 posições.

Program	TLB 64 fully assoc.			SB 64 fully assoc.		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	14.629.003.906	19.973.300	0,1365%	14.646.371.073	2.606.133	0,0178%
Libre	13.507.146.613	9.901.993	0,0733%	13.512.276.316	4.772.290	0,0353%
MySQL	15.248.798.241	23.132.199	0,1517%	15.271.835.738	94.702	0,0006%
Python	14.470.938.920	27.863.360	0,1925%	14.498.802.009	271	0,0000%
QEMU	13.612.698.396	4.961.451	0,0364%	13.615.794.319	1.865.528	0,0137%
Tomcat	14.502.548.067	9.543.677	0,0658%	14.512.087.038	4.706	0,0000%
Average	14.328.522.357	15.895.997	0,1109%	14.342.861.082	1.557.272	0,0109%

Tabela 5.5: Troca de contexto - *Cache* com 64 posições, completamente associativo.

Program	TLB 64 fully assoc.			SB 64 fully assoc.		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	2.996.082.876	1.153.079	0,0385%	2.997.221.926	14.029	0,0005%
Libre	2.702.592.967	2.519.371	0,0932%	2.705.012.921	99.417	0,0037%
MySQL	3.028.315.832	4.826.528	0,1594%	3.033.128.238	14.122	0,0005%
Python	2.861.579.458	7.176.008	0,2508%	2.868.755.405	61	0,0000%
QEMU	2.779.870.912	3.601.059	0,1295%	2.783.471.925	46	0,0000%
Tomcat	2.924.884.664	1.869.542	0,0639%	2.926.753.303	903	0,0000%
Average	2.882.221.118	3.524.265	0,1223%	2.885.723.953	21.430	0,0007%

programas *pbig* com: (a) *QEMU* com apenas 46 faltas (0%); (b) *Firefox* com apenas 14.029 faltas (0,0005%); e (c) *LibreOffice* com 99.417 faltas (0,0037%).

5.2.3 Cache com 128 posições, totalmente associativo

A Tabela 5.6 mostra os números para a TLB128, com 0,0327% de faltas e a SB128 com 0,0078%, uma diferença de desempenho em favor da SB128 de 0,0249 pontos percentuais, que correspondem a 626.525 faltas a mais na TLB128. Os programas *psmall* utilizam todo o espaço disponível na SB128 para armazenar seus segmentos, resultando em uma taxa de faltas de 0% em todos os três programas (*MySQL* 151 faltas, *Python* 116 faltas e *Tomcat* 369 faltas).

Os resultados para a simulação inicial da TLB128 também mostram um cenário de melhoria de desempenho comparado com às *caches* SB32 e SB64. Com mais espaço disponível, a porcentagem média de faltas da TLB128 (0,0327%) é menor que o da TLB64 (0,1109%). A SB64 é eficiente em manter as páginas mais utilizadas presentes na *cache* mas tem a metade do espaço para evitar colisões. O resultado médio da SB128 (0,0078%) é melhor que o resultado da SB64 (0,0109%) e indica que a SB128 também consegue distribuir os segmentos no espaço maior de *cache* de forma eficiente.

Os resultados para a simulação de troca de contexto, na Tabela 5.7 mostram todos os resultados para a SB128 com desempenho ótimo. O programa que mais sofre faltas na SB128 é o *LibreOffice* com 0,0003% de faltas, ou 7.319 faltas no total. Em média a SB128 tem 0,0% de faltas.

A TLB128 na simulação de troca de contexto tem um número de faltas médio de 0,0414%, muito acima que a SB128. A diferença entre as duas é grande, com a TLB128 com 10^6

Tabela 5.6: Comparativo de desempenho para *cache* completamente associativo de 128 posições.

Program	TLB 128, fully assoc.			SB 128, fully assoc.		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	14.641.213.700	7.763.506	0,0530%	14.647.093.955	1.883.251	0,0129%
Libre	13.513.121.626	3.926.980	0,0291%	13.513.438.648	3.609.958	0,0267%
MySQL	15.263.127.469	8.802.971	0,0577%	15.271.930.289	151	0,0000%
Python	14.494.230.969	4.571.311	0,0315%	14.498.802.164	116	0,0000%
QEMU	13.617.125.714	534.133	0,0039%	13.616.475.941	1.183.906	0,0087%
Tomcat	14.502.548.067	2.564.151	0,0177%	14.512.091.375	369	0,0000%
Average	14.338.561.258	4.693.842	0,0327%	14.343.305.395	1.112.959	0,0078%

Tabela 5.7: Troca de contexto - *Cache* com 128 posições, totalmente associativo.

Program	TLB 128, fully assoc.			SB 128, fully assoc.		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	2.996.609.350	626.605	0,0209%	2.997.235.875	80	0,0000%
Libre	2.703.770.688	1.341.650	0,0496%	2.705.105.019	7.319	0,0003%
MySQL	3.031.329.477	1.812.883	0,0598%	3.033.142.283	77	0,0000%
Python	2.867.356.865	1.398.601	0,0488%	2.868.755.405	61	0,0000%
QEMU	2.782.056.198	1.415.773	0,0509%	2.783.471.925	46	0,0000%
Tomcat	2.926.189.306	564.900	0,0193%	2.926.754.084	122	0,0000%
Average	2.884.551.981	1.193.402	0,0414%	2.885.744.099	1.284	0,0000%

faltas em média enquanto a SB128 tem apenas 10^3. A comparação entre as duas *caches* mostra claramente que a segmentação, na da simulação de troca de contexto, tem um desempenho muito superior ao oferecido pela paginação.

5.3 Limite de desempenho da TLB

A análise dos resultados de todas as *caches* simuladas mostra um ganho de desempenho significativo para a segmentação, com uma menor quantidade de faltas na *cache* para cada configuração de SB e TLB simulada. Para comparar o desempenho das SBs em relação a um TLB grande o suficiente para representar o possível limite em *hardware*, simulamos duas outras configurações de TLB: (a) a TLB256, totalmente associativa com 256 posições; e (b) a TLB1024, totalmente associativa com 1024 posições. Com os resultados da TLB256 e da TLB1024 é possível comparar com as SBs simuladas e estimar qual o tamanho que uma TLB precisa ter para ultrapassar o desempenho das SB.

A Tabela 5.12 mostra os resultados de todas as TLBs e SBs para a simulação inicial, incluindo a TLB256 e TLB1024. A tabela é composta por: (a) a *cache* simulada (*Cache*); (b) a média de acertos de todos os programas simulados (*Average Hit*); (c) a média de faltas de todos os programas simulados (*Average miss*); e (d) a porcentagem de faltas na *cache* (*% miss*). A média foi calculada levando em conta todos os programas simulados. A Tabela 5.12 está ordenada pelo número de faltas na *cache* em ordem crescente.

Os resultados na Tabela 5.12 mostram a TLB1024 com o menor número de faltas na simulação inicial com 18.499 (10^4) faltas, seguida da TLB256 com 824.516 faltas (10^5). A SB128 fica em terceiro lugar com 1.112.959 faltas (10^6).

Tabela 5.8: Média de desempenho de todas as *caches* ordenada pela quantidade de faltas, para a simulação inicial.

Cache	Average hit	Average miss	% miss
TLB 1024, fully assoc.	14.344.399.855	18.499	0,0001%
TLB 256, fully assoc.	14.343.593.838	824.516	0,0057%
SB 128, fully assoc.	14.343.305.395	1.112.959	0,0078%
SB 64, fully assoc.	14.342.861.082	1.557.272	0,0109%
SB 32, fully assoc.	14.342.169.605	2.248.749	0,0157%
TLB 128, fully assoc.	14.338.561.258	4.693.842	0,0327%
TLB 64, fully assoc.	14.328.522.357	15.895.997	0,1109%
TLB 32, fully assoc.	14.279.827.978	64.590.376	0,4523%

Tabela 5.9: Média de desempenho de todas as *caches* ordenada pela quantidade de faltas, simulação de troca de contexto.

Cache	Average hit	Average miss	% miss
SB 128, fully assoc.	2.885.744.099	1.284	0,0000%
TLB 1024, fully assoc.	2.885.730.848	14.535	0,0005%
SB 64, fully assoc.	2.885.723.953	21.430	0,0007%
TLB 256, fully assoc.	2.885.566.616	178.767	0,0062%
SB 32, fully assoc.	2.885.465.247	280.136	0,0097%
TLB 128, fully assoc.	2.884.551.981	1.193.402	0,0414%
TLB 64, fully assoc.	2.882.221.118	3.524.265	0,1223%
TLB 32, fully assoc.	2.872.164.386	13.580.996	0,4728%

Todas as SB ficam à frente da TLB128 com 10^6 faltas em média, enquanto as TLBs menores (TLB32, TLB64) apresentam em média 10^7 faltas. Os resultados mostram que a SB128 tem 288.442 faltas a mais que a TLB256, sendo menor e com uma implementação em hardware mais simples. Podemos inferir que, para os programas que simulamos, se construirmos em *hardware* uma SB com a mesma complexidade de uma TLB, a SB sempre será mais eficiente e sofrerá menos faltas.

A Tabela 5.9 mostra os resultados de todas as TLBs e SBs para a simulação de troca de contexto, incluindo a TLB256 e TLB1024. A tabela é composta por: (a) a *cache* simulada (*Cache*); (b) a média de acertos de todos os programas simulados (*Average Hit*); (c) a média de faltas de todos os programas simulados (*Average miss*); e (d) a porcentagem de faltas na *cache* (*% miss*). A média foi calculada levando em conta todos os programas simulados. A Tabela 5.9 está ordenada pelo número de faltas na *cache* em ordem crescente.

Na simulação de troca de contexto, a SB128 possui o melhor desempenho entre todas as caches com apenas 1.284 faltas (10^3). A segunda *cache* mais eficiente é a TLB1024 com 14.535 faltas (10^4), e a terceira *cache* mais eficiente é a SB64 com 21.430 faltas (10^4). Os resultados alcançados pela SB128 na simulação de troca de contexto mostram que SB128 ultrapassa o desempenho da maior TLB simulada.

5.4 Considerações sobre a quantidade de segmentos nos programas testados

O histograma da Figura 5.25 mostra a quantidade de segmentos para cada classificação de tamanho. Para facilitar a visualização, o tamanho do segmento foi arredondado para a próxima potência de dois e os segmentos maiores que 64MiB foram excluídos. O histograma mostra uma barra com a quantidade de segmentos para cada programa simulado, e uma linha representando a quantidade média de segmentos para cada tamanho. Observando a linha da quantidade média de segmentos, vemos que uma concentração grande de segmentos de: (a) 4KiB com 200 segmentos; (b) 8KiB com 50 segmentos; e (c) 2MiB com 100 segmentos.

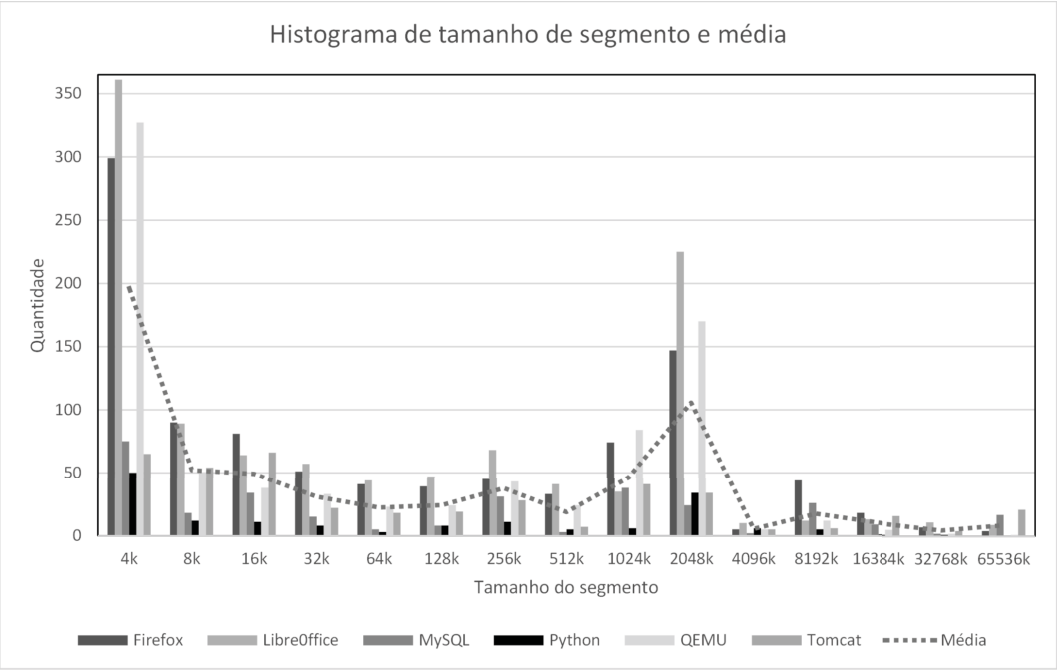

Figura 5.25: Histograma de segmentos de todas as simulações com valor médio.

Os resultados para segmentação foram calculados utilizando traços de execução de programas com paginação. Eles não representam traços de uma programa real que utiliza segmentação para gerenciamento de memória, e sim uma aproximação consistente para simular e descobrir o desempenho na tradução de endereços virtuais para endereços físicos que um sistema com segmentação possui utilizando traços de programas reais.

Em um sistema puramente segmentado, a quantidade de segmentos dos programas seria, provavelmente, menor. Muitos dos segmentos de 4KiB podem ser agrupados em segmentos maiores. O mesmo se aplica para os segmentos de 8KiB e 2MiB. Com compilador, ligador e sistema operacional otimizados para trabalhar com segmentação, o número total de segmentos de cada processo seria menor. Com menos segmentos na tabela de segmentos a quantidade de faltas seria, também, menor.

Em um sistema com suporte a *huge pages* (ou *super pages*) os segmentos de 2MiB seriam mapeados diretamente para páginas de 2MiB, aumentando o desempenho da TLB. Os outros segmentos entre 8KiB e 1024MiB utilizam páginas de de 4KiB. Mesmo com suporte a *huge pages* a quantidade de páginas seria diversas ordens de grandeza maior que a quantidade de segmentos.

Tabela 5.10: Comparativo de desempenho para *cache* completamente associativo de 128 posições *8 way set-associative*.

Program	TLB 128, 8 way.			SB 128, 8 way		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	14.610.877.634	38.099.572	0,2608%	14.647.272.217	1.704.989	0,0116%
Libre	13.485.255.669	31.792.937	0,2358%	13.513.372.460	3.676.146	0,0272%
MySQL	15.195.050.370	76.880.070	0,5060%	15.271.930.279	161	0,0000%
Python	14.394.816.274	103.986.006	0,7224%	14.498.802.163	117	0,0000%
QEMU	13.606.720.678	10.939.169	0,0804%	13.616.276.870	1.382.977	0,0102%
Tomcat	14.502.548.067	32.705.705	0,2255%	14.512.091.259	485	0,0000%
Average	14.299.211.449	49.067.243	0,3431%	14.343.290.875	1.127.479	0,0079%

A definição e projeto de um sistema completo, composto por compilador, ligador e sistema operacional com suporte a segmentação e a um *segment buffer* está fora do escopo do trabalho. Mas as suas implicações precisam ser consideradas quando analisamos os resultados das simulações.

5.5 Considerações da simulação de *caches* com conjuntos associativos

Simulamos uma configuração de *cache* com 128 posições utilizando 8 conjuntos associativos para a TLB e para a SB. Chamamos a TLB com 128 posições com 8 conjuntos associativos de TLB128Assoc. Chamamos a SB com 128 posições e 8 conjuntos associativos de SB128Assoc.

A Tabela 5.10 mostra os números para a TLB128Assoc, com 0,3431% de faltas e a SB128Assoc com 0,0079%, uma diferença de desempenho em favor da SB128Assoc de 0,3353 pontos percentuais, que correspondem a 47.939.764 de faltas a mais na TLB128Assoc. Os programas *psmall* utilizam todo o espaço disponível na TLB128Assoc para armazenar seus segmentos, resultando em uma taxa de faltas de 0% em todos os três programas (*MySQL* 161 faltas, *Python* 117 faltas e *Tomcat* 485 faltas).

Os resultados para a simulação inicial da TLB128Assoc também mostram um cenário distinto das *caches* totalmente associativas SB32 e SB64. Mesmo com mais espaço disponível, a porcentagem média de faltas da TLB128Assoc (0,3431%) é maior que o da TLB64 (0,1109%). A SB64, que é completamente associativa, é mais eficiente em manter as páginas mais utilizadas presentes na *cache* que a TLB128Assoc, que tem 128 posições divididas em 8 conjuntos associativos (*8 way set-associative*). O resultado médio da SB128Assoc (0,0079%) é melhor que o resultado da SB64 (0,0109%) e indica que a SB128Assoc consegue distribui os segmentos nos 8 conjuntos associativos de forma eficiente.

Os resultados para a simulação de troca de contexto, na Tabela 5.11 mostram todos os resultados para a SB128Assoc com desempenho ótimo. O programa que mais sofre faltas na SB128Assoc é o *LibreOffice* com 0,0003% de faltas, ou 8.360 faltas no total. Em média a SB128Assoc tem 0,0001% de faltas.

A TLB128Assoc na simulação de troca de contexto tem um número de faltas médio de 0,3885%, muito acima que a SB128Assoc. A diferença entre as duas é grande, com a TLB128Assoc com 10^7 faltas em média enquanto a SB128Assoc tem apenas 10^3. A comparação

Tabela 5.11: Troca de contexto - Cache com 128 posições, *8-way set associative*.

Program	TLB 128, 8 way.			SB 128, 8 way		
	Hit	Miss	Miss %	Hit	Miss	Miss %
Firefox	2.994.398.777	2.837.178	0,0947%	2.997.234.719	1.236	0,0000%
LibreOffice	2.694.780.615	10.331.723	0,3834%	2.705.103.978	8.360	0,0003%
MySQL	3.017.123.618	16.018.742	0,5309%	3.033.142.283	77	0,0000%
Python	2.847.070.972	21.684.494	0,7616%	2.868.755.405	61	0,0000%
QEMU	2.773.541.704	9.930.267	0,3580%	2.783.471.925	46	0,0000%
Tomcat	2.920.546.658	6.207.548	0,2125%	2.926.754.077	129	0,0000%
Average	2.874.577.057	11.168.325	0,3885%	2.885.743.731	1.652	0,0001%

entre as duas *caches* mostra claramente que a segmentação, na da simulação de troca de contexto, tem um desempenho muito superior ao oferecido pela paginação.

As observações dos resultados individuais de cada programa indicam que a SB128Assoc fornece os melhores resultados considerando tamanho e complexidade em hardware, versus o desempenho entregue. A implementação da SB128Assoc (*8-way*) é mais simples que a SB64 (*64-way*) e os resultados simulados são melhores.

5.6 Limite de desempenho da TLB

A análise dos resultados de todas as *caches* simuladas mostra um ganho de desempenho significativo para a segmentação, com uma menor quantidade de faltas na *cache* para cada configuração de SB e TLB simulada. Para comparar o desempenho das SBs em relação a um TLB grande o suficiente para representar o possível limite em *hardware*, simulamos duas outras configurações de TLB: (a) a TLB256, totalmente associativa com 256 posições; e (b) a TLB1024, totalmente associativa com 1024 posições. Com os resultados da TLB256 e da TLB1024 é possível comparar com as SBs simuladas e estimar qual o tamanho que uma TLB precisa ter para ultrapassar o desempenho das SB.

A Tabela 5.12 mostra os resultados de todas as TLBs e SBs para a simulação inicial, incluindo a TLB256 e TLB1024. A tabela é composta por: (a) a *cache* simulada (*Cache*); (b) a média de acertos de todos os programas simulados (*Average Hit*); (c) a média de faltas de todos os programas simulados (*Average miss*); e (d) a porcentagem de faltas na *cache* (*% miss*). A média foi calculada levando em conta todos os programas simulados. A Tabela 5.12 está ordenada pelo número de faltas na *cache* em ordem crescente.

Os resultados na Tabela 5.12 mostram a TLB1024 com o menor número de faltas na simulação inicial com 18.499 (10^4) faltas, seguida da TLB256 com 824.516 faltas (10^5). A SB128 fica em terceiro lugar com 1.127.479 faltas (10^6).

Todas as SB ficam à frente da TLB64 com 10^6 faltas em média, enquanto as TLBs menores (TLB32, TLB64 e TLB128) apresentam em média 10^7 faltas. Os resultados mostram que a SB128 com 128 posições e 8 conjuntos associativos tem 302.963 faltas a mais que a TLB256, sendo menor e com uma implementação em hardware mais simples. Podemos inferir que, para os programas que simulamos, se construirmos uma SB com a mesma complexidade de uma TLB, a SB sempre será mais eficiente e sofrerá menos faltas.

A Tabela 5.9 mostra os resultados de todas as TLBs e SBs para a simulação de troca de contexto, incluindo a TLB256 e TLB1024. A tabela é composta por: (a) a *cache* simulada (*Cache*); (b) a média de acertos de todos os programas simulados (*Average Hit*); (c) a média de

Tabela 5.12: Média de desempenho de todas as *caches* ordenada pela quantidade de faltas, para a simulação inicial.

Cache	Average hit	Average miss	% miss
TLB 1024, fully assoc.	14.344.399.855	18.499	0,0001%
TLB 256, fully assoc.	14.343.593.838	824.516	0,0057%
SB 128, 8-way set assoc.	14.343.290.875	1.127.479	0,0079%
SB 64, fully assoc.	14.342.861.082	1.557.272	0,0109%
SB 32, fully assoc.	14.342.169.605	2.248.749	0,0157%
TLB 64, fully assoc.	14.328.522.357	15.895.997	0,1109%
TLB 128, 8-way set assoc.	14.299.211.449	49.067.243	0,3431%
TLB 32, fully assoc.	14.279.827.978	64.590.376	0,4523%

Tabela 5.13: Média de desempenho de todas as *caches* ordenada pela quantidade de faltas, para a simulação inicial. Incluindo *cache* com conjunto associativo.

Cache	Average hit	Average miss	% miss
TLB 1024, fully assoc.	14.344.399.855	18.499	0,0001%
TLB 256, fully assoc.	14.343.593.838	824.516	0,0057%
SB 128, fully assoc.	14.343.305.395	1.112.959	0,0078%
SB 128, 8-way.	14.343.290.875	1.127.479	0,0079%
SB 64, fully assoc.	14.342.861.082	1.557.272	0,0109%
SB 32, fully assoc.	14.342.169.605	2.248.749	0,0157%
TLB 128, fully assoc.	14.338.561.258	4.693.842	0,0327%
TLB 64, fully assoc.	14.328.522.357	15.895.997	0,1109%
TLB 32, fully assoc.	14.279.827.978	64.590.376	0,4523%

faltas de todos os programas simulados (*Average miss*); e (d) a porcentagem de faltas na *cache* (*% miss*). A média foi calculada levando em conta todos os programas simulados. A Tabela 5.9 está ordenada pelo número de faltas na *cache* em ordem crescente.

Na simulação de troca de contexto, a SB128 possui o melhor desempenho entre todas as caches com apenas 1.652 faltas (10^3). A segunda *cache* mais eficiente é a TLB1024 com 14.535 faltas (10^4), e a terceira *cache* mais eficiente é a SB64 com 21.430 faltas (10^4). Os resultados alcançados pela SB128 na simulação de troca de contexto mostram que SB128 ultrapassa o desempenho da maior TLB simulada.

A SB128Assoc tem um desempenho semelhante a SB128 completamente associativa no cenário inicial, com apenas 14.520 faltas a mais para a SB128Assoc, como pode ser visto na Tabela 5.13. No cenário de troca de contexto o SB128Assoc tem um desempenho na mesma ordem de grandeza da SB128 completamente associativa, com apenas 368 faltas a mais para a SB128Assoc, como pode ser visto na Tabela 5.14.

5.7 Considerações finais

Os resultados mostram que o uso de segmentação para o gerenciamento de memória traz uma diminuição potencial no número de faltas para tradução de endereços virtuais em endereços físicos, na ordem 10^2 até 10^4 faltas. Mostram também que uma SB é sempre mais eficiente que uma TLB de mesmo tamanho e organização.

Tabela 5.14: Média de desempenho de todas as caches ordenada pela quantidade de faltas, simulação de troca de contexto. Incluindo *cache* com conjunto associativo.

Cache	**Average hit**	**Average miss**	**% miss**
SB 128, fully assoc.	2.885.744.099	1.284	0,0000%
SB 128, 8-way.	2.885.743.731	1.652	0,0001%
TLB 1024, fully assoc.	2.885.730.848	14.535	0,0005%
SB 64, fully assoc.	2.885.723.953	21.430	0,0007%
TLB 256, fully assoc.	2.885.566.616	178.767	0,0062%
SB 32, fully assoc.	2.885.465.247	280.136	0,0097%
TLB 128, fully assoc.	2.884.551.981	1.193.402	0,0414%
TLB 64, fully assoc.	2.882.221.118	3.524.265	0,1223%
TLB 32, fully assoc.	2.872.164.386	13.580.996	0,4728%

Com base nos resultados da simulação é possível supor como algumas categorias de aplicações se beneficiariam do ganho de desempenho da SB, que apresenta de 10 a 50 vezes menos faltas que uma TLB de mesma capacidade. Listamos abaixo algumas aplicações e conceitos que podem se beneficiar desse ganho de desempenho.

Um sistema de gerenciamento de banco de dados (SGBDs) utilizando segmentação pode guardar em alguns segmentos informações de *cache* para acelerar pesquisas, e para implementar *journaling* que garante a integridade relacional. Um segmento pode ser utilizado para guardar um sub-conjunto de uma tabela ou até mesmo todo seu conteúdo. A quantidade de segmentos necessários para o SGBD seria dependente do número de funcionalidades que o mesmo necessita manter em memória.

Bancos de dados em memória (*In Memory Databases*, IMDB) são gerenciadores de banco de dados que utilizam a memória *RAM* para armazenar os seus dados, diferentemente dos SGBDs que persistem os seus dados em disco. Os IMDBs são mais rápidos que os SGBDs, os algoritmos para otimização são mais simples e o fato dos dados estarem na memória principal elimina o tempo de procura (*seek time*), resultando em um desempenho maior, e mais previsível, do que utilizando discos [18].

As simulações sugerem que a utilização de segmentação para implementar um IMDB trará ganhos significativos em desempenho. Com uma quantidade de segmentos muito menor que a quantidade de páginas necessárias para mapear o seu espaço de endereçamento, a quantidade de faltas na SB será menor que na TLB. Com menos faltas na tradução de endereço o IMDB tem mais ciclos de processador disponíveis para tarefas relacionadas as necessidades do IMDB.

Os resultados mostram que aplicações *desktop* complexas e que necessitam de muita memória, como o navegador *Firefox*, tem ganhos de desempenho quando empregamos segmentação. Os ganhos podem ser ainda mais expressivos se o compilador, ligador e sistema operacional forem otimizados para utilizar segmentação de uma forma eficiente. O número total de segmentos de um programa otimizado para segmentação será menor, e a quantidade de faltas na SB também.

A simulação de troca de contexto mostra que depois da inicialização do programa, a sua execução utiliza menos segmentos, e o número de faltas na SB cai em relação ao inicio da execução do programa. A simulação de troca de contexto reflete o cenário de uso mais comum, no qual uma aplicação *desktop*, científica ou *server side* é executada durante um longo período de tempo após a sua inicialização.

Os resultados sugerem que o projeto da SB é significativamente mais simples, e menor, que o projeto de uma TLB para atingir o mesmo desempenho. Com isso o projetista da CPU pode modelar uma SB que tenha um bom desempenho e adicionar mais recursos para adicionar

mecanismos que tem o potêncial de diminuir ainda mais a quantidade de faltas. Um exemplo é projetar um pequeno cache associativo que pode ser colocado entre a SB e a memória principal e guardar os últimos segmentos que foram removidos da SB devido a falta de espaço ou devido à conflitos. Essa cache associativa pode ser utilizada para guardar segmentos úteis, trazidos ativamente da tabela de segmentos da memória *RAM* utilizando um mecanismo de *prefetch*, aliado a um algoritmo que escolha segmentos que podem ser usados em breve pela *CPU*. Essa *cache* associativa proposta pode ser uma *victim cache* [29] [26].

Capítulo 6

Conclusão

A memória não volátil já é uma realidade e pode ser encontrada em produtos de consumo. A *Intel* propõe um modelo de utilização de memória não volátil, chamada de *NVM* utilizando a tecnologia *3D XPoint*. O modelo adiciona uma camada com NVM conectada diretamente ao controlador de memória da *CPU* entre a memória principal e o disco. Para utilizar a *NVM* o programador tem um conjunto de funções e estruturas de dados (API) chamada *NVML* (*NVM Libraries*) que atua em conjunto com as chamadas do sistema operacional, para mapear arquivos em memória e persistir esses arquivos na *NVM*, onde os dados podem ser acessados em *bytes* (*byte level access*) em vez de serem acessados como um dispositivo baseado em blocos (*block device*).

A implementação de um sistema operacional moderno em um hardware com memória *RAM* "infinita" e não volátil, é potencialmente mais simples que as implementações atualmente utilizadas. Muitos dos artifícios utilizados para esconder a latência no acesso à disco, ou para acesso específico a *NVRAM*, simplesmente não fazem mais sentido pois todos os dados necessários estão na memória principal e são persistentes. Nesse cenário a paginação sob demanda não é mais necessária [32].

Na próxima década poderemos ter a produção dos primeiros sistemas equipados com 2^{64} *bytes* de memória física, e talvez uma boa parte desta memória seja composta por *NVRAM*. Nos acreditamos que devemos considerar com bastante seriedade um novo desenho para o nosso conhecido e confiável sistema gerecimento de memória com paginação sob demanda.

Para suportar essa recomendação propomos a utilização de gerenciamento de memória com segmentação e um *buffer* de segmentos (SB). Realizamos simulações utilizando traços de execução de programas reais, de *caches* com tamanho, complexidade e organização similares para paginação sob demanda (TLBs) e gerenciamento de memória com segmentação (SBs).

A simulação foi realizada a partir de traços de execução de programas instrumentados com o programa *Valgrind* utilizando o módulo *Lackey* para gerar um fluxo referência à memória – instruções e dados – que são capturados pelo utilitário *Trace Collector* e armazenados em arquivos compactados. Durante a execução do programa também é capturado o mapa de memória do processo com o comando *pmap*.

Utilizamos na simulação os programas: (a) *Firefox* – navegador *web*; (b) *LibreOffice* – conjunto de aplicativos de escritório de *software* livre; (c) *MySQL* – sistema gerenciador de banco de dados de software livre; (d) *Python* – linguagem e ambiente de execução interpretado baseado em software livre; (e) *QEMU* – emulador de processador e ambiente de virtualização de computadores *open source*; e (f) *Tomcat* – um servidor *web* e *servelet container open source* escrito em *Java*.

Para simular a paginação o simulador lê os arquivos de traços e utiliza os endereços lineares presentes no arquivo. As TLBs simuladas são: (a) TLB32; (b) TLB64; (c) TLB128; (d) TLB256; e (e) TLB1024; A TLB256 e TLB1024 representam o caso "limite" de desempenho da TLB.

Para segmentação o simulador utiliza o arquivo contendo o mapa de memória produzido pelo *pmap* e cria uma tabela de segmentos. Para todo endereço linear lido do arquivo de traços o simulador procura em qual segmento da tabela de segmentos o endereço linear pertence. Encontrado o segmento o simulador cria um endereço segmentado que pode ser simulado em uma SB. Se o endereço não possuir um segmento associado o simulador cria um segmento "simulado" para que a contabilização de faltas leve em conta todos os endereços presente no traço. As SBs simuladas são: (a) SB32; (b) SB64; e (c) SB128;

Descobrimos que as SBs sofrem de 10 a 50 vezes menos faltas que TLBs de mesma capacidade, o que não é uma surpresa, porque o número de segmentos de um programa é muito menor que a quantidade de páginas de uma aplicação. O gerenciamento de uma tabela de segmentos pequena é menos custoso que gerenciar uma tabela de páginas muito grande, também resultando em melhora no desempenho.

Diversas classes de aplicações podem se beneficiar da quantidade reduzida de faltas na tradução de endereços virtuais em endereços físicos que a segmentação oferece. Os resultados da simulação sugerem que sistemas de gerenciamento de banco de dados e bancos de dados em memória podem apresentar ganhos significativos em desempenho. Mostram também que aplicações *desktop* como o *Firefox* podem ter um desempenho melhor com segmentação.

Um sistema operacional com *RAM* "infinita" e não volátil precisa de um novo projeto para gerenciamento de memória virtual, pois vários mecanismos complexos para esconder a latência empregados nos sistemas operacionais atuais, ou acessar diretamente a *NVRAM*, não são mais necessários em um cenário com *RAM* "infinita" e não volátil.

Os resultados sugerem que é necessário um novo desenho para o nosso conhecido e confiável sistema gerecimento de memória com paginação sob demanda. Utilizar gerenciamento de memória com segmentação, *NVRAM* e um SB implica em novos desafios, e são necessários trabalhos futuros abordando o tema para: (a) avaliar a fragmentação externa produzida por um sistema puramente segmentado com "memória infinita", e sugerir medidas para mitigar os seus efeitos; (b) modelar um SB em um processador existente em *VHDL*, ou modificar um software de emulação de *CPU* como o *QMEU*, para prover um ambiente em que programas utilizando o SB com segmentação possam ser avaliados; (c) criar, ou modificar, um sistema operacional para implementar gerenciamento de memória com segmentação e com um SB; e (d) explorar as possibilidades da memória *RAM* não volátil para projetar sistemas que possam utilizar segmentação para gerenciamento de memória e persistência de dados.

Referências Bibliográficas

[1] Peter Baer Gavin Abraham Silberschatz. *Sistemas Operacionais - Conceitos*. Prentice Hall, 5th edition, 2000. ISBN 8587918028.

[2] Michael Kerrisk Andries Brouwer. Linux man pages online - mmap, Sep 2017.

[3] Storage Networking Industry Association. Persistent memory, Nov 2017. `https://www.snia.org/PM?utm_source=ISTV&utm_medium=Video&utm_campaign=ISTV_2017`.

[4] Anirudh Badam. How persistent memory will change software systems. *computer*, 46(8):45–51, August 2013.

[5] K Bailey, L Ceze, S D Gribble, and H M Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proc USENIX Conf on Hot Topics in Operating Systems*, pages 2–2, 2011.

[6] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 237–248. ACM, 2013.

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

[8] M Bjørling, P Bonnet, L Bouganim, and N Dayan. The necessary death of the block device interface. In *Proc 6th Biennial Conf on Innovative Data Systems Research (CIDR13)*, 2013.

[9] A M Caulfield, A De, J Coburn, T I Mollow, R K Gupta, and S Swanson. Moneta: a high-performance storage array architecture for next-generation, non-volatile memories. In *Proc 43rd IEEE/ACM Int Symp on Microarchitecture (MICRO'10)*, pages 385–395, 2010.

[10] J Condit, E B Nightingale, C Frost, E Ipek, B Lee, D Burger, and D Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc ACM 22nd Symp Operating Systems Principles (SIGOPS)*, pages 133–146, 2009.

[11] Intel Corporation. *Intel ® 64 and IA-32 Architectures Software Developer's Manual Volume 3A : System Programming Guide , Part 1*, volume 3. Intel Corporation, 2010.

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

[13] Wikimedia Foundation. Apache tomcat, Aug 2017. `https://en.wikipedia.org/wiki/Java_(programming_language)`.

[14] Wikimedia Foundation. Apache tomcat, Aug 2017. https://en.wikipedia.org/wiki/Apache_Tomcat.

[15] Wikimedia Foundation. Class - abstract and concrete, Jul 2017. https://en.wikipedia.org/wiki/Class_(computer_programming).

[16] Wikimedia Foundation. Firefox, Aug 2017. https://en.wikipedia.org/wiki/Firefox.

[17] Wikimedia Foundation. Freedos, Aug 2017. https://en.wikipedia.org/wiki/FreeDOS.

[18] Wikimedia Foundation. In memory database, Nov 2017. https://en.wikipedia.org/wiki/In-memory_database.

[19] Wikimedia Foundation. Libreoffice, Aug 2017. https://en.wikipedia.org/wiki/LibreOffice.

[20] Wikimedia Foundation. Memory segmentation, May 2017. https://en.wikipedia.org/wiki/Memory_segmentation.

[21] Wikimedia Foundation. Mysql, Aug 2017. https://en.wikipedia.org/wiki/MySQL.

[22] Wikimedia Foundation. Page - computer memory, Oct 2017. https://en.wikipedia.org/wiki/Page_(computer_memory).

[23] Wikimedia Foundation. Python, Aug 2017. https://en.wikipedia.org/wiki/Python_(programming_language).

[24] Wikimedia Foundation. Qemu, Aug 2017. https://en.wikipedia.org/wiki/QEMU.

[25] Narayanan Ganapathy and Curt Schimmel. General purpose operating system support for multiple page sizes. In *In Proceedings of the USENIX Conference. USENIX*, pages 91–104, 1998.

[26] Giancarlo C Heck and Roberto A Hexsel. The performance of Pollution Control Victim Cache for embedded systems. pages 46–51, 2008.

[27] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1st edition, 1990. ISBN 1558600698.

[28] Peter Hornyack, Luis Ceze, Steve Gribble, Dan Ports, and Hank Levy. A Study of Virtual Memory Usage and Implications for Large Memory. *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013.

[29] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA'90: 17th Intl Symp on Computer Arch*, pages 364–373. ACM Press, 1990.

[30] T Kilburn, DBG Edwards, M J Lanigan, and F H Sumner. Readings in computer architecture. chapter One-level Storage System, pages 405–417. Morgan Kaufmann Publishers Inc., 2000.

[31] Eggers S Koldinger E, Chase J. Architecture support for single address space operating systems. *SIGPLAN Not*, 27(9):175–186, 1992.

[32] Lauri P Laux Jr and Roberto A Hexsel. Back to the past: Segmentation with infinite and non-volatile memory. In *WSCAD-SSC'16: XVII Workshop em Sistemas Computacionais de Alto Desempenho*, pages 278–289, 2016.

[33] Shuichi Oikawa. Non-volatile main memory management methods based on a file system. *SpringerPlus*, 3(1):494, 2014.

[34] David A Patterson and John L Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2009. ISBN 9780123744937.

[35] Ubuntu Manpage Repository. sysbench - a modular, cross-platform and multi-threaded benchmark tool., Aug 2017. `http://manpages.ubuntu.com/manpages/xenial/man1/sysbench.1.html`.

[36] Dipanjan Sengupta, Qi Wang, Haris Volos, Ludmila Cherkasova, Jun Li, Guilherme Magalhaes, and Karsten Schwan. A framework for emulating non-volatile memory systemswith different performance characteristics. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 317–320. ACM, 2015.

[37] Usharano U. Introduction to programming with persistent memory from intel, Nov 2017. `https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel`.