Cache comprimido adaptativo: projeto, estudo e implementação

Rodrigo Souza de Castro

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIAS

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Alair Pereira do Lago

– Durante o desenvolvimento deste trabalho, -
– o autor recebeu apoio financeiro da FAPESP e CNPq –

São Paulo
2003

Cache comprimido adaptativo: projeto, estudo e implementação

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Rodrigo Souza de Castro e aprovada pela comissão julgadora.

São Paulo, 06 de junho de 2003.

Banca examinadora:

- Prof. Dr. Alair Pereira do Lago (orientador) (IME-USP)
- Prof. Dr. Siang Wun Song (IME-USP)
- Prof. Dr. Rômulo Silva de Oliveira (UFSC)

Agradecimentos

Inicialmente, agradeço à Fundação de Amparo à Pesquisa do Estado de São Paulo – FAPESP, pela bolsa de estudos concedida que me permitiu trabalhar integralmente no opção que considerei como sendo a que mais me traria satisfação pessoal nesses últimos 2 anos.

Sou grato ao Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq, e ao Instituto de Matemática e Estatística (IME) da Universidade de São Paulo (USP) pelas duas bolsas de iniciação científica que me permitiram efetuar um primeiro estudo de vital importância acerca do assunto assim como um primeiro contato com a implementação que veio a ser recomeçada e concluída nesse trabalho de mestrado. Sem essas bolsas, possivelmente eu teria tomado rumos diferentes na minha vida.

Expresso também os meus agradecimentos ao Prof. Dr. Imre Simon, do Departamento de Ciência da Computação do IME-USP, que através do projeto 465091 do CNPq, ajudou-me gentilmente a complementar a reserva técnica fornecida pela FAPESP com a finalidade de uma compra de um computador para o desenvolvimento do projeto.

Ao meu orientador, o Prof. Dr. Alair Pereira do Lago, sou grato pela sua paciência e pela dedicação nesse projeto de mestrado, sempre querendo estar a par de cada problema enfretado, ajudando na maior parte das vezes com sugestões e recomendações que eu nem mesmo vislumbrava. Agradeço pela sua confiança em mim, por me tratar como um amigo e pelas diversas conversas nas quais ajudou-me a ver possibilidades de futuro. E por fim, agradeço o seu apoio desde a graduação. Se não fosse a sua orientação e as suas idéias de pesquisa, provavelmente eu não teria feito uma iniciação científica nem decidido seguir no mestrado.

À Dra. Dilma Menezes da Silva, na prática a minha co-orientadora, primeiramente pelas suas excelentes aulas enquanto professora no IME-USP. Agradeço ao seu interesse pelo meu desenvolvimento como pesquisador, assim como aos diversos comentários e sugestões perspicazes sobre esse projeto durante o meu mestrado e no seu estado embrionário na iniciação científica. Também sou lhe grato pela confiança e pela sua disponibilidade constante em me ajudar em qualquer problema, apesar da sua vida super corrida como pesquisadora.

Ao Prof. Dr. Fabio Kon pelo todo apoio e confiança que demonstrou ter por mim. Agradeço o seu esforço em me conseguir um espaço no IME-USP para eu trabalhar, em me ajudar com conselhos e revisões de texto, e por me delegar atividades como a coordenação

dos seminários em sistemas de computação. Aqui também agradeço enormemente a sua participação nas bancas do exame de qualificação e do exame de dissertação do mestrado.

A todos os meus amigos que me apoiaram e sempre confiaram que na minha competência, eu lhes sou muito grato. Ao Livio por estar sempre a par dos problemas, pelas suas inúmeras sugestões inteligentes de como melhorar o projeto, pelo tempo empregado me auxiliando no que possível. Ao Rafael pela sua amizade, e ao Breno e Betinho, por estarem sempre presentes e dispostos a me ajudar (e pelos jantares, é claro). E ao Caetano pela confiança e pela ajuda.

Aos meus pais Marcos e Marta pelo apoio incondicional durante todo o mestrado e pela confiança no caminho que tomei, mesmo que eles não soubessem com clareza se seria o melhor para mim. E ao meu irmão Diego, pela sua constante alegria para me animar mesmo em momentos de grandes pressões. À Alessandra pelo carinho e paciência que teve durante o momento que esteve junto a mim nesse projeto.

Ao Scott Kaplan pela sua disponibilidade e simpatia em responder às minhas dúvidas em relação ao seu trabalho. Ao Marcelo Tosatti, pela ajuda nas suas explicações e idéias no começo do projeto que auxiliaram o projeto a ter um desenvolvimento mais eficiente. Ao Paolo Ciarrocchi, pelo seu grande interesse e disponibilidade em efetuar exprimentos desde as versões mais primitivas da implementação. Ao Marc-Christian Petersen, pelo seu retorno sobre o desempenho do cache comprimido, integração do código ao seu conjunto de *patches* para o kernel do Linux, e sua pronta disponibilidade em testar as mais diversas versões.

Por último, aos que demonstraram não acreditar que esse projeto poderia ser concluído com sucesso, por qualquer motivo, racional ou não, devo também lhes agradecer. A sua existência, de alguma forma, ajudou-me a trabalhar mais intensamente.

A rational mind does not work under compulsion; it does not subordinate its grasp of reality to anyone's orders, directives, or controls; it does not sacrifice its knowledge, its view of the truth, to anyone's opinions, threats, wishes, plans, or "welfare". Such a mind may be hampered by others, it may be silenced, proscribed, imprisoned, or destroyed; it cannot be forced; a gun is not an argument. (An example and symbol of this attitude is Galileo). It is from the work and the inviolate integrity of such minds - from the intransigent innovators - that all of mankind's knowledge and achievements have come. It is to such minds that mankind owes its survival.

Ayn Rand, "Capitalism: The Unknown Ideal"

The great creators - the thinkers, the artists, the scientists, the inventors - stood alone against the men of their time. Every great new thought was opposed. Every great new invention was denounced. The first motor was considered foolish. The airplane was considered impossible... But the men of unborrowed vision went ahead. They fought, they suffered and they paid. But they won.

Ayn Rand, "For the New Intellectual"

Resumo

Nesse trabalho, reavaliamos o uso do cache comprimido adaptativo para melhorar o desempenho do sistema através da redução dos acessos aos dispositivos secundários de armazenamento. Nós propomos uma nova política de adaptabilidade que ajusta o tamanho do cache comprimido em tempo de execução, e avaliamos o sistema de cache comprimido com essa política através de uma implementação em um sistema operacional amplamente usado, o Linux. Também reprojetamos o cache comprimido de modo a prover melhoras de desempenho para todos os workloads testados e acreditamos que ele aborde os problemas encontrados em trabalhos e implementações anteriores. Entre essas modificações fundamentais, o nosso cache comprimido é o primeiro a também comprimir páginas do cache de arquivos e a desabilitar adaptativamente a compressão de páginas limpas quando necessário.

Testamos um sistema com o nosso cache comprimido adaptativo com muitos aplicativos e benchmarks, cada um com diferentes pressões de memória. Os resultados mostraram melhoras de desempenho (até 171,4%) em todos eles se há pressão de memória, overhead mínimo (até 0,39%) quando há muito baixa pressão de memória e praticamente nenhum overhead quando não há nenhuma pressão de memória. Acreditamos que esse trabalho mostre que esse projeto de cache comprimido adaptativo deva ser considerado como um efetivo mecanismo para melhoras no desempenho do sistema.

Abstract

In this work, we reevaluate the use of adaptive compressed caching to improve system performance through the reduction of accesses to the backing stores. We propose a new adaptability policy that adjusts the compressed cache size on-the-fly, and evaluate a compressed caching system with this policy through an implementation in a widely used operating system, Linux. We also redesign compressed caching in order to provide performance improvements for all the tested workloads and we believe it addresses the problems faced in previous works and implementations. Among these fundamental modifications, our compressed cache is the first one to also compress file cache pages and to adaptively disable compression of clean pages when necessary.

We tested a system with our adaptive compressed cache under many applications and benchmarks, each one with different memory pressures. The results showed performance improvements (up to 171.4%) in all of them if under memory pressure, minimal overhead (up to 0.39%) when there is very light memory pressure and almost no overhead under no memory pressure. We believe this work shows that this adaptive compressed cache design should be actually considered as an effective mechanism for improvement in system performance.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Cache comprimido é um método usado para melhorar o tempo médio de acesso às páginas de memória. Ele é considerado um novo nível na hierarquia de memória virtual onde uma parte da memória principal é alocada para o *cache comprimido* e é utilizada para armazenar páginas comprimidas por algoritmos de compressão de dados. Armazenar um número de páginas no formato comprimido aumenta o tamanho efetivo da memória e, para a maioria dos workloads, essa melhora reduz o número de acessos a dispositivos secundários de armazenamento, tipicamente discos rígidos lentos. Esse método faz uso da distância, cuja tendência é só crescer, entre o poder de processamento da CPU e o tempo de latência do disco. Esta latência é atualmente cerca de seis ordens de magnitude mais lenta que um acesso à memória principal. Essa distância é responsável por, entre outros, a subutilização da CPU quando as necessidades do sistema excedem a memória disponível. Um exemplo desse efeito é a compilação do kernel do Linux. Mesmo quando muitos processos do compilador são executados concorrentemente para compilar a árvore do código do kernel, o uso de CPU cai significativamente se a memória disponível não é suficiente para o armazenar o seu *working set*. E isso também é verdadeiro para sistemas atuais típicos com centenas de megabytes de memória que executam exigentes workloads, como servidores web, servidores de arquivos e sistemas de banco de dados. Nesses cenários, o cache comprimido pode fazer melhor uso do poder da CPU reduzindo acessos aos dispositivos de armazenamento e suavizando as quedas de desempenho quando a memória disponível não é suficiente. A preocupação com esses cenários em que a memória disponível não é suficiente é ainda objeto de estudo nos sistemas operacionais atuais ao se procurar melhorais para os seus sistemas de memória virtual, em particular em suas políticas de substituição de páginas.

Apesar da redução de acessos devido à compressão tender a melhorar o desempenho do sistema, a redução da memória não-comprimida (memória principal não alocada para uso do cache comprimido) tende a piorá-lo. Essa compromisso inerente ao cache comprimido nos leva à questão de quanta memória deve ser utilizada pelo cache comprimido em determinado momento do sistema. A quantidade de memória que atinge o melhor desempenho para o sistema depende do workload. Um cache comprimido que adapta o seu tamanho durante a execução do sistema de modo a obter um bom equilíbrio é chamado de *adaptativo* e um com tamanho fixo é chamado de *estático*.

O uso do cache comprimido para reduzir paginação em disco foi primeiramente proposto por

Wilson [78, 79], e Appel e Li [3]. Douglis implementou um cache comprimido adaptativo no sistema operacional Sprite, obtendo melhoras de desempenho para alguns experimentos e pioras para outros [18]. Ele também apontou que um cache comprimido estático só é adequado para um pequeno número de aplicativos.

Dados os resultados inconclusivos de Douglis, o problema foi revisto por muitos autores. Russinovich e Cogswell implementaram um cache comprimido estático no Windows 95 [62]. A análise deles foi baseada no benchmark Ziff-Davis Winstone, resultando em conclusões negativas: "O resultado que obtemos foi que, apesar de parecer possível na teoria obter um benefício de um cache comprimido do arquivo de swap, para parâmetros realistas é extremamente difícil.". Por outro lado, Kjelso et al [31, 32, 33] empiricamente avaliou a compressão da memória principal, concluindo que ela poderia aumentar o desempenho do sistema para aplicativos com uso intensivo de memória. Kaplan [26, 80] demonstrou através de simulações que um cache comprimido pode prover alta redução nos custos de paginação. Os seus experimentos confirmaram as afirmações de Douglis sobre as limitações de um sistema de cache comprimido estático. Ademais, ele propõe um esquema adaptativo que detecta durante a execução do sistema quanta memória o cache comprimido deve usar. Esse esquema garantiu melhorias nos custos de paginação para todos os seis programas que ele simulou usando "traces" de memória. Uma implementação de um cache comprimido estático no sistema operacional Linux foi feita por Cervera et al. [8]. Apesar das limitações inerentes de um cache comprimido estático, eles mostraram melhoria de desempenhos para a maior parte dos workloads testados.

Nesse trabalho, nós reavaliamos o cache comprimido adaptativo através de aplicativos reais e benchmarks numa implementação no sistema operacional Linux. Nós demonstramos que um cache comprimido adaptativo pode prover melhora significativa para a maior parte das aplicativos com um projeto que minimize os custos e o impacto no sistema de memória virtual assim como também utilize eficientemente a memória alocada. Nossa implementação usa uma nova política de adaptabilidade que intenta identificar durante a execução do sistema a quantidade de memória que o cache comprimido deve usar para chegar ao melhor termo comum. Essa política procura adicionar overhead de memória e CPU mínimos ao sistema.

Nós também mostraremos que o uso de cache comprimido altera o comportamento de diversas partes do sistema operacional e que os seus custos estão além de compressões e descompressões de páginas de memória. Em particular, mostramos que taxas de compressão ruins para páginas de memória são contornáveis e que não afetam o desempenho do cache comprimido tão gravemente quanto afirmado em estudos anteriores. Ademais, o cache comprimido adaptativo é revisto numa época em que a razão principal que motivou essa idéia, a diferença entre a poder de processamento da CPU e os tempos de acesso a disco, nunca foi tão grande.

Na próximo capítulo, descrevemos os conceitos de memória virtual, além de uma descrição do sistema de gerenciamento de memória do Linux com alguns trechos do código-fonte comentado. No Capítulo 3, apresentamos uma visão geral do cache comprimido juntamente com as principais decisões de projeto; e no Capítulo 4 descrevemos a política de adaptabilidade projetada e implementada. O Capítulo 5 trata dos mais detalhes da implementação importantes, enquanto o Capítulo 6 apresenta o conjunto de testes feitos, os seus resultados com a nossa implementação e análise dos resultados. O Capítulo 7 apresenta os trabalhos relacionados e faz uma análise comparativa com

as inovações desse trabalho. Por fim, o Capítulo 9 lista os possíveis trabalhos futuros relacionados a esse trabalho e o Capítulo 10 apresenta as conclusões finais.

Capítulo 2

Memória Virtual

Nessa seção apresentaremos uma visão geral do sistema dos conceitos de memória virtual assim descreveremos brevemente como o kernel do Linux 2.4.18 [28] implementa esses conceitos. O número da versão do kernel é mencionado visto que o Linux, no seu desenvolvimento constante, está sujeito a grandes mudanças mesmo numa versão estável, como é a série 2.4, e que podem alterar inclusive o seu gerenciamento de memória virtual (mais detalhes a respeito das séries do kernel do Linux podem ser encontrados em [29]). Além disso, a versão citada é a utilizada como base para os nossos experimentos exibidos nesse texto. Dessa forma, quando mencionarmos Linux daqui para a frente, estaremos nos referirmos especificamente à sua versão 2.4.18.

Antes de entrarmos em detalhes sobre o funcionamento da políticas e algoritmos de liberação de memória do Linux, faz-se necessário revermos brevemente alguns conceitos básicos de sistema operacional e de memória virtual presentes em sistemas atuais, como é o caso do Linux.

2.1 Conceitos

Nessa seção temos os conceitos básicos que fundamentam a concepção do cache comprimido. Os conceitos de sistema operacional, gerenciamento de memória, cache, swap e memória virtual são apresentados de modo sucinto a seguir.

2.1.1 Sistema Operacional

Relembraremos, em primeiro lugar, o que é um sistema operacional [67, 68, 64]. Um computador pode ser analisado em camadas: uma mais interna, o hardware, e outra mais externa, o software. O hardware consiste dos dispositivos físicos. Entre os diversos dispositivos essenciais de um computador, podemos destacar aqueles por onde as informações mais circulam: CPU, memória e disco (veja Fig. 2.1).

O software, por sua vez, consiste do conjunto de operações a serem executadas pelo hardware. Este conjunto é pré-programado e pode efetuar diversas tarefas, como armazenar e processar informações. Entre as categorias de software, a mais básica e que cuida do gerenciamento dos recursos

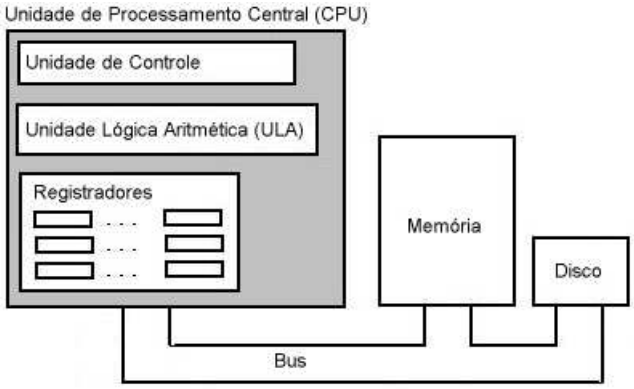

Figura 2.1: CPU, memória e disco.

do computador é conhecida como sistema operacional. O sistema operacional também exerce um papel de máquina virtual através da qual os aplicativos dos usuário têm acesso aos recursos do sistema.

Ao usuário final, o que interessa é a execução de seus aplicativos (ou também programas). Contudo, um sistema operacional que saiba administrar bem os recursos do sistema é normalmente um fator decisivo para o bom funcionamento destes aplicativos.

Muito poderia ser falado sobre as técnicas utilizadas pelos sistemas operacionais modernos para uma melhor utilização da CPU ou dos discos. Aqui, queremos apenas observar que, como tendência geral, para um melhor aproveitamento da CPU, um sistema operacional moderno costuma executar diversos processos concorrentemente. Direcionaremo-nos, a partir de agora, para o gerenciamento da memória.

2.1.2 Gerenciamento de Memória

Apesar de por um lado o preço por byte[1] da memória ter diminuído ao longo dos anos, por outro lado os programas têm sido cada vez mais vorazes em suas exigências por memória. Como regra geral, pouco tempo após uma expansão da memória física disponível num particular sistema de computação, já se faz notar sua escassez. Assim, não é de se estranhar que o gerenciador de memória dentro do sistema operacional seja um de seus subsistemas mais complexos e importantes. Cabe ao gerenciamento de memória a responsabilidade de determinar onde são colocados os processos em execução (alocação e liberação de memória), onde são colocadas as próprias estruturas de dados do sistema operacional, como proteger os dados de um processo dos outros processos concorrentes, implementar um sistema de memória virtual, lidar com o fato de que um processo precisa de uma quantidade variável de memória ao longo da sua execução e, de uma maneira mais geral, com o fato de que dispomos apenas de uma quantidade finita de memória disponível.

Uma das técnicas mais importantes que se cristalizaram dentro do sistema de memória ao longo

[1] Ao longo desse texto, Kb significa 1024 bytes, Mb significa 1024Kb e Gb significa 1024Mb.

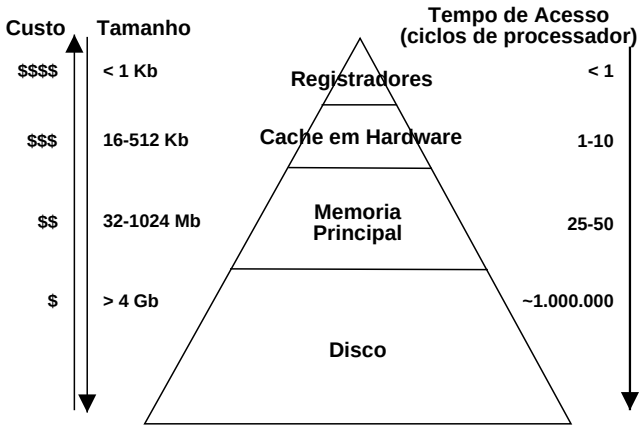

Figura 2.2: Hierarquia entre os dispositivos.

do tempo é a do sistema de *cache*. Antes de vê-lo, veremos alguns princípios que justificam sua arquitetura.

A primeira motivação para a arquitetura de um sistema de *cache* é o **princípio da localidade**. Ele consiste no fato de que as referências à memória feitas por um processo não são aleatórias. Dado um instante de execução, é muito mais provável que itens de memória que serão acessados dentro de um período razoável de tempo sejam vizinhos dos itens acessados recentemente (espacial) e que esses itens previamente acessados sejam acessados novamente (temporal).

A segunda motivação é a natural hierarquia entre os diversos dispositivos de armazenamento de dados [54, 9]. Esta hierarquia é composta principalmente dos seguintes níveis: registradores, memória principal e disco. Na Figura 2.2, podemos observar valores típicos de seus tempos de acesso e tamanho. Vale ainda observar que o custo por byte decresce dos registradores para os discos [13] (veja também outros gráficos [11, 12]).

2.1.3 Cache

A necessidade básica que levou ao *cache* é querer ter um acesso mais rápido a uma informação armazenada num nível lento e grande numa hierarquia de memória. O conceito fundamental num sistema de *cache* é que, dispondo de um nível intermediário mais rápido porém limitado em tamanho (devido a ser mais caro que o primeiro, por exemplo) e valendo o princípio da localidade, podemos manter nesse nível intermediário cópias das vizinhanças das informações ora acessadas no nível mais lento. Devido ao princípio da localidade, podemos trabalhar com estas cópias durante um bom tempo, sem ter que acessar o sistema mais lento. Isto leva à situação em que o programa enxerga o espaço de endereçamento do sistema grande, com os tempos de acesso do sistema rápido. Para que o sistema funcione, é necessário que o conjunto das vizinhanças acessadas caiba no sistema intermediário por um tempo satisfatório.

Exemplos

O conceito de sistema de *cache* é aplicado em diversos níveis pelo sistema operacional. Um primeiro exemplo da aplicação do conceito de sistema de *cache* dá-se devido às diferenças de tempo de acesso e tamanho entre os dois níveis superiores da hierarquia acima exposta (veja Fig 2.2). No decorrer dos anos, a diminuição do tempo de acesso das pastilhas de memória não acompanhou a evolução da velocidade de processamento das CPUs, gerando um problema pois grande parte do tempo de processamento é desperdiçada esperando-se por dados da memória principal. Para amenizar esse problema, diversos níveis de memória com taxas mais rápidas de acesso (e custos mais altos) foram adicionados entre memória e CPU. Essa memória de acesso rápido é conhecida como *memória cache* – ou muitas vezes simplesmente *cache*, e é usualmente memória estática. Ela pode ser introduzida em mais de um nível e pode estar dentro ou fora da CPU. Atualmente é o mais comum termos até três níveis e os seus nomes serem *caches* L1, L2 e L3, mas é importante observar que há máquinas com mais ou diferentes níveis.

Para gerenciar os dados entre a CPU, *caches* e memória principal, existe um circuito especial chamado de controlador de *cache*. E para que os seus dados sejam consistentes, são empregadas políticas de gerenciamento de *cache* que consistem nas políticas de descarte, de leitura e escrita (na memória principal).

Outro exemplo de sistema de *cache* é resultante do acesso às informações armazenadas nos sistemas de arquivos. Face à lentidão dos discos comparada ao tempo de acesso à memória principal, os sistemas operacionais mais modernos costumam reservar parte da memória principal para um sistema de *cache* de dados dos arquivos de interesse dos processos em execução.

2.1.4 Swap e Memória Virtual

Numa situação normal de trabalho, com vários processos executando, carregar um novo processo, ou mesmo lidar com uma necessidade maior de memória por parte deles, pode ser impossível caso estejamos limitados apenas à memória principal. Dessa forma, surgiu uma técnica que é mover processos ou partes deles da memória principal para o disco e vice-versa, o que permite a maior utilização da CPU, seja pelo maior número de processos e/ou processos maiores. Esse processo é conhecido como *swapping*, o disco ou partição utilizada para essa tarefa é tida como *swap* e o espaço no swap é manejado pelo gerenciador de memória. No entanto, swap surgiu quando ainda se armazenavam processos inteiros na memória. Um pouco depois, veio à tona outro problema: gerenciar grandes processos que precisam de mais memória que a disponível para eles. A solução consagrada foi a da memória virtual (VM), normalmente associada a técnicas como paginação. Cada programa tem um endereçamento virtual, que é traduzido para endereço físico pela unidade de gerenciamento de memória (ou MMU, memory management unit). Esse endereçamento é dividido em unidades chamadas páginas e a sua contra-partida na memória física é a página física. Os seus tamanhos tipicamente variam de 512 bytes a 8 kilobytes (sempre potências de 2), e a relação do endereçamento virtual com o físico é feita através de tabelas de páginas. Quando um processo acessa um endereço virtual que não possui uma relação com um endereço físico, a CPU volta a executar o sistema operacional, que deve recuperar-se dessa falha. Essa situação é conhecida como falha de página. Nesse caso, o sistema operacional deve estabelecer o vínculo entre o endereçamento

virtual e real, alocando uma página na memória, caso ela não esteja alocada, e lendo os seus dados de um dispositivo de armazenamento, se for um endereço armazenado em algum dispositivo de armazenamento (como sistema de arquivo ou swap).

No caso de necessidade de espaço na memória principal, far-se-á uma escolha entre as páginas na memória seguindo algum algoritmo pré-determinado, que são conhecidos como algoritmos de substituição de páginas, e as páginas escolhidas serão liberadas, possivelmente sendo armazenadas em um dispositivo secundário de armazenamento, se necessário. Sob um certo ponto de vista, a memória principal passa a ser um *cache* da memória virtual. As páginas não presentes neste "*cache*" são guardadas no swap. O algoritmo de substituição de páginas mais comum nos sistemas operacionais é o LRU (Least Recently Used, ou Menos Recentemente Usado). Com o LRU, as páginas menos recentemente usadas são escolhidas para serem liberadas pelo sistema operacional quando há necessidade de espaço na memória principal.

2.2 Linux

Nessa seção, falaremos brevemente do Linux e da sua história, dando em seqüência conceitos de memória virtual implementados no Linux, e também descrevendo os principais caches, estruturas de dados e heurísticas de tratamento da memória virtual.

2.2.1 Visão Geral

O Linux é um kernel de sistema operacional livre compatível com o UNIX. Ele está sob a licença GNU General Public License, ou conhecida por GPL [22], e o código-fonte do Linux é livremente distribuído. Pelo fato de o seu código-fonte estar licenciado sob a GPL, qualquer pessoa tem acesso a ele, podendo estudá-lo e alterá-lo, desde que ele continue sendo livre (mais detalhes na licença GPL). Além disso, o Linux é o sistema operacional livre mais difundido no mundo atualmente, possuindo uma grande quantidade de desenvolvedores, trabalhando nele por hobby ou por estarem em empresas de alguma forma relacionadas ao Linux. Conhecer melhor a arquitetura do Linux permite que se esteja em contato com novidades e com tecnologias de alto nível. O sistema adquiriu graus de sofisticação surpreendentes e há muito tempo deixou de ser um projeto de estudante – que foi a maneira como começou, veja abaixo – a ponto de operar, por exemplo, 30% de todos os servidores Web mundiais [24].

O Linux não é oficialmente compatível com a especificação IEEE POSIX (Portable Operating Systems based on Unix), que é um dos padrões para padronização dos Unices, mas preenche a maioria dos seus requisitos. Os padrões atuais especificam apenas uma interface de programação de aplicativo, ou API (application programming interface), por isso até sistemas operacionais não compatíveis com o Unix podem ser compatíveis com padrões como POSIX, como é o caso do Windows NT [82]. Ademais, pelo fato de o Linux implementar a maior parte das especificações dos ramos System V e BSD Unix [7], portar software de qualquer um deles (por exemplo, do FreeBSD [19] ou do Sun Solaris [65]) não encontra grandes obstáculos.

Originalmente, o Linux foi desenvolvido para a arquitetura Intel i386, mas atualmente funciona, ou possui algum projeto proposto a fazê-lo funcionar, numa grande gama de arquiteturas [37, 52],

incluindo sistemas com multi-processadores ou arquiteturas NUMA[39]. Veja alguns exemplos na Tabela 2.1.

Intel IA-64	Hewlett-Packard/Apollo Domain
AMD x86-64	Apple Macintosh
PowerPC	Placas VME baseadas no 680x0
IBM PPC64	HP 9000/300
Power Macintosh	Sun 3/XXX
Amiga PowerUP	Máquinas 68000 sem MMU
Arquitetura IBM CHRP	Intel XScale, StrongARM e outros
Microchannel RS/6000	MIPS
Nubus Power Mac	Silicon Graphics hardware.
PReP, PowerPC Reference	Qube - Cobalt
Embedded PowerPC	Compaq (ex-Digital) DECStation
IBM S/390	Compaq/DEC VAX
Alpha	HP PA-RISC
UltraSPARC 32 e 64 bits	ETRAX Asix
Série 68000	Sistemas sem MMU
Amiga	SONY Playstation 2
Atari ST and TT, Medusa, Falcon	Intel 8086/80286

Tabela 2.1: Exemplos de processadores com os quais o Linux funciona, além do i386, processador para o qual foi originalmente criado.

O código-fonte do kernel do Linux é muito complexo, pois a eficiência tem preferência sobre a clareza, além de conter conceitos avançados de sistemas operacionais. A versão 2.4.18 possui mais de 4,1 milhões de linhas de código, sendo o código na linguagem C (maioria) e em linguagem de máquina. A maior parte relacionada ao gerenciamento de memória está localizada no diretório `mm/`, sigla para *memory management*. Os arquivos desse diretório possuem aproximadamente 15 mil linhas de código-fonte no total.

Comum em projetos de software livre, novos desenvolvedores ou pesquisadores, para adquirir um conhecimento detalhado de como o kernel funciona, devem estudar o código-fonte, linha por linha, o que requer um grande investimento de tempo. Em particular, isso ocorre quando a implementação difere muito de documentos – como artigos ou livros – os descrevendo. Além disso, apesar de nos últimos anos terem sido lançados livros a respeito do kernel [6, 61], não há uma rica documentação sobre tópicos específicos do kernel como o sistema de memória virtual.

O código-fonte do Linux, como citado acima, é separado em duas partes, independente e dependente de arquitetura, como o MachOS [41]. A parte dependente de arquitetura está localizada nos diretórios `arch/` e `include/asm-*`, contendo os arquivos de código e *headers*, respectivamente. Os arquivos independentes de arquitetura são armazenados em diretórios de acordo com a área com a qual o arquivo está relacionado. No diretório `drivers/` temos os arquivos relacionados a drivers de dispositivos. Em `fs/`, todos os arquivos de sistemas de arquivos, e em `net/` arquivos relacionados

a parte de redes. Abaixo, na Tabela 2.2, temos a listagem do primeiro nível dos diretórios do código-fonte do Linux, juntamente com os segundos níveis dos diretórios `include/` (que contém os *headers*) e `arch/` (com os arquivos com o código). À esquerda, temos a soma de todos os arquivos e subdiretórios de um determinado diretório.

tam	diretório	tam	diretório	tam	diretório
145Mb	linux-2.4.18/	5,8Mb	Documentation/		asm-sparc/
27Mb	arch/	74Mb	drivers/		asm-sparc64/
	alpha/	11Mb	fs/		asm-sparc64/
	arm/	20Mb	include/		asm-sparc64/
	cris/		asm-alpha/		math-emu/
	i386/		asm-arm/		net/
	ia64/		asm-cris/		pcmcia/
	m68k/		asm-generic/		scsi/
	mips/		asm-i386/		video/
	mips64/		asm-ia64/	28Kb	init/
	parisc/		asm-m68k/	92Kb	ipc/
	ppc/		asm-mips/	392Kb	kernel/
	s390/		asm-parisc/	124Kb	lib/
	s390x/		asm-ppc/	416Kb	mm/
	sh/		asm-s390/	6,5Mb	net/
	sparc/		asm-s390x/	472Kb	scripts/
	sparc64/		asm-sh/		

Tabela 2.2: Primeiro nível dos diretórios do código-fonte do Linux, juntamente com a soma dos tamanhos dos arquivos dentro de cada um deles. Também há o segundo nível de diretórios para os diretórios `include/` e `arch/`.

2.2.2 Histórico

Na época em que o Linux começou a ser desenvolvido, o DOS, vendido pela Microsoft [46], era a única opção para os usuários de computadores pessoas, conhecidos como PCs. Apesar de existirem soluções da Apple Computers, elas eram muito caras.

Uma opção ao mundo Unix, que também tinha um alto custo, além de não ser focada nos PCs, era o Minix, criado por Andrew S. Tanenbaum, professor da Vrije Universiteit, na Holanda. O Minix foi criado em janeiro de 1987 e foi desenvolvido para ser um sistema destinado ao ensino. O seu código-fonte era aberto, ele rodava em processadores Intel 8086, mas ainda era um sistema operacional de estudantes e não poderoso o suficiente para uso comercial. Além do Minix, existia o desenvolvimento do kernel do GNU Hurd, mas devido a algumas razões, entre elas a própria estrutura escolhida do kernel (microkernel), o seu desenvolvimento ainda demoraria alguns anos para se ter uma versão que pudesse ser utilizada.

Tendo em vista esse cenário em que não havia expectativas de um sistema operacional robusto para PCs a curto prazo, Linus Benedict Torvalds, um estudante de 21 anos que cursava o segundo ano de ciência da computação na Universidade de Helsinki, na Finlândia, iniciou o desenvolvimento do Linux em 1991 (o sistema operacional ainda não tinha nenhum nome divulgado na época). Em 25 de agosto de 1991, Linus divulgou que estava fazendo esse novo sistema operacional através de uma mensagem de correio eletrônico para o *newsgroup* do Minix, que é reproduzida integralmente abaixo. Nessa mensagem, ele perguntava por recursos que as pessoas gostariam de ter no Minix, pois isso poderia guiar o seu desenvolvimento.

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI>
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki


Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones.  This has been brewing
since april, and is starting to get ready.  I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want.  Any suggestions
are welcome, but I won't promise I'll implement them :-)

               Linus (torvalds@kruuna.helsinki.fi)

PS.  Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that's all I have :-(.
```

A primeira versão do Linux, numerada de 0.01 e com 8400 linhas de código C e assembly, foi lançada no meio de setembro de 1991 [36], mas não foi anunciada pois o estado inicial do código era precário. A seguir, a versão 0.02 foi lançada no dia 5 de outubro do mesmo ano. Na divulgação dessa versão é que o nome Linux surgiu pela primeira vez (veja o nome do diretório do servidor de FTP onde o código estava localizado na mensagem do anúncio da versão 0.02 abaixo).

No entanto, o primeiro nome oficial do sistema operacional era Freax, mas um funcionário da Universidade de Helsinki é que, insatisfeito com o nome, usou Linux no seu lugar para disponibilizar o código no servidor de FTP da universidade. O nome original pode ainda ser achado no arquivo `kernel/Makefile` da versão 0.11 [30].

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: Free minix-like kernel sources for 386-AT
Message-ID: <1991Oct5.054106.4647@klaava.Helsinki.FI>
Date: 5 Oct 91 05:41:06 GMT
Organization: University of Helsinki

Do you pine for the nice days of minix-1.1, when men were men and wrote
their own device drivers? Are you without a nice project and just dying
to cut your teeth on a OS you can try to modify for your needs? Are you
finding it frustrating when everything works on minix? No more all-
nighters to get a nifty program working? Then this post might be just
for you :-)

As I mentioned a month(?) ago, I'm working on a free version of a
minix-lookalike for AT-386 computers.  It has finally reached the stage
where it's even usable (though may not be depending on what you want),
and I am willing to put out the sources for wider distribution.  It is
just version 0.02 (+1 (very small) patch already), but I've successfully
run bash/gcc/gnu-make/gnu-sed/compress etc under it.

Sources for this pet project of mine can be found at nic.funet.fi
(128.214.6.100) in the directory /pub/OS/Linux.  The directory also
contains some README-file and a couple of binaries to work under linux
(bash, update and gcc, what more can you ask for :-).  Full kernel
source is provided, as no minix code has been used.  Library sources are
only partially free, so that cannot be distributed currently.  The
system is able to compile "as-is" and has been known to work.  Heh.
Sources to the binaries (bash and gcc) can be found at the same place in
/pub/gnu.

ALERT! WARNING! NOTE! These sources still need minix-386 to be compiled
(and gcc-1.40, possibly 1.37.1, haven't tested), and you need minix to
set it up if you want to run it, so it is not yet a standalone system
for those of you without minix. I'm working on it. You also need to be
something of a hacker to set it up (?), so for those hoping for an
alternative to minix-386, please ignore me. It is currently meant for
hackers interested in operating systems and 386's with access to minix.
```

```
The system needs an AT-compatible harddisk (IDE is fine) and EGA/VGA. If
you are still interested, please ftp the README/RELNOTES, and/or mail me
for additional info.

I can (well, almost) hear you asking yourselves "why?".  Hurd will be
out in a year (or two, or next month, who knows), and I've already got
minix.  This is a program for hackers by a hacker.  I've enjouyed doing
it, and somebody might enjoy looking at it and even modifying it for
their own needs.  It is still small enough to understand, use and
modify, and I'm looking forward to any comments you might have.

I'm also interested in hearing from anybody who has written any of the
utilities/library functions for minix. If your efforts are freely
distributable (under copyright or even public domain), I'd like to hear
from you, so I can add them to the system. I'm using Earl Chews estdio
right now (thanks for a nice and working system Earl), and similar works
will be very wellcome. Your (C)'s will of course be left intact. Drop me
a line if you are willing to let me use your code.

               Linus

PS. to PHIL NELSON! I'm unable to get through to you, and keep getting
"forward error - strawberry unknown domain" or something.
```

A próxima versão foi a 0.03, depois de algumas semanas, em que havia um bom funcionamento do sistema. Em novembro de 1991, pulou-se diretamente para a versão 0.10, uma vez que o sistema começou a funcionar muito bem. Nessa época, tinha-se somente suporte para discos rígidos AT e não havia login (a inicialização rodava automaticamente o shell `bash`). Logo em seguida, a versão 0.11 foi lançada com suporte a teclados multi-linguais, drivers de disquete, suporte para VGA, EGA, CGA, MDA, Hercules, ferramentas do sistema de arquivos (mkfs/fsck/fdisk), entre outros. Na versão 0.12, a licença inicial do código foi substituída pela GPL, que era menos restritiva. Dessa versão, o Linux foi para a versão 0.95. A versão 0.96 tornou-se a primeira versão capaz de rodar o sistema de janelas X Window [87, 86]. Depois de mais de dois anos de desenvolvimento, a versão 1.0 foi lançada no dia 16 de abril de 1994, possuindo cerca de 170 mil linhas de código.

A partir da versão 1.0, o desenvolvimento do kernel do Linux foi dividido. Desde então, uma versão do Linux é formada por três números, separados por pontos. As versões com o segundo número par, como 1.0.3, 1.2.10 e 2.0.30, denotam kernels estáveis prontos para uso, ao passo que números pares ímpares, como 1.1.9 e 1.3.32, denotam kernels de desenvolvedores que são apenas para desenvolvimento e testes. Durante as versões de desenvolvimento, assim que os recursos desejados são incorporados, o código usualmente passa por uma fase de congelamento (*code freeze*), sendo então renomeado como o primeiro kernel de uma série estável.

Ao longo do tempo, muitos recursos foram adicionados ao Linux. Módulos carregáveis (*loadable modules*) foram introduzidos em um kernel 0.99. No kernel 1.2, lançado no dia 6 de março de 1995, o Linux foi estendido para processadores Alpha, Sparc e MIPS, deixando de ter tão forte dependência com os processadores x86 como no seu início (situação na qual o próprio Linus dizia que portar o kernel era impossível). A capacidade básica de lidar com sistemas de memória compartilhada com multi-processadores foi incluído na versão 2.0 do kernel. A sua versão estável atual, 2.4, com mais de 4 milhões de linhas de código, funciona em cada importante plataforma atual, além de possuir drivers para os mais variados tipos de hardware existentes, incluindo USB e Firewire.

2.2.3 Memória Paginada

No Linux, toda a memória é dividida em páginas, cujos tamanhos são de acordo com a arquitetura de hardware. No entanto, apenas alguns tipos de páginas do sistema são liberadas quando há necessidade de espaço na memória principal, ou seja, apenas alguns tipos são paginados. Eles são descritos a seguir:

1. páginas utilizadas para mapear arquivos de um meio secundário de armazenamento (por exemplo, um disco rígido) através da chamada de sistema *mmap()*;

2. páginas não vinculadas a um meio secundário de armazenamento, conhecidas como *páginas anônimas*, sendo um exemplo uma pilha de um processo;

3. páginas de uma região de memória compartilhada utilizada para comunicação entre processos ou *IPC* (InterProcess Communication);

4. páginas originalmente pertencentes a um mapeamento de memória privado que foram modificadas, perdendo o seu vínculo com o arquivo que estava mapeando.

Outros tipos de páginas, que não as mapeadas pelos processos ou que contenham dados armazenados em um meio secundário de armazenamento, não são paginadas. Entre elas, temos as páginas utilizadas pelo segmento de código do kernel ou pelas suas estruturas de dados.

2.2.4 Page Cache

O page cache, ou cache de páginas, é composto por todas as páginas da memória que contenham dados vinculados a um dispositivo secundário de armazenamento. As páginas desse cache são indexadas por uma estrutura chamada de `address space`, que define o inode ou dispositivo de bloco ao qual a página está vinculada, assim como um índice ou deslocamento dentro dele. O `address space` pode definir também os métodos para lidar com as páginas pertencentes a ele, como a função para leitura de página, de escrita, preparação de escrita e assim por diante.

Swap Cache

O swap cache armazena páginas cujo dispositivo secundário de armazenamento é a área de swap (arquivo ou partição). Uma página sem dispositivo secundário de armazenamento, como uma página anônima, uma página modificada de mapeamento privado ou de um região de memória compartilhada IPC, é adicionada ao **swap cache** primeiramente para poder ser escrita no swap. Em determinadas condições, como o caso em que o swap está cheio, essa página pode ser removida do swap cache.

Uma página do swap cache é na verdade somente uma página do page cache cujo `address space` está definido como `swapper_space`, por esse motivo este cache é, na prática, apenas uma parte do page cache.

2.2.5 Cache de Estruturas de Dados

O Linux provê uma infra-estrutura para alocação de estrutura de dados do seu kernel através do alocador *slab* [5, 72]. Dentro do kernel, é possível alocar certas estruturas criando slab caches para elas e assim utilizar-se dessa infra-estrutura. Ela automaticamente faz o caching dos objetos que são freqüentemente alocados e liberadas, aumentando o desempenho pelo fato de se evitar novas alocações de objetos complexos. E também permite que todos os slab caches liberem os seus objetos quando há pressão de memória no sistema, além de um esquema que permita melhor uso do cache de hardware, assim como do bus.

Estruturas como inodes, dentry e buffers são alocadas através de slab caches. No entanto, apenas as estruturas com os meta-dados são alocadas utilizando o slab cache, os dados em si são armazenados em páginas que fazem parte do page cache.

2.2.6 Buffers

No Linux, páginas podem ser lidas ou escritas em um dispositivo de bloco usando estruturas auxiliares chamadas de buffers, que armazenam dados de blocos em páginas do cache de página. Buffers são usados para diminuir a espera dos processos por dados que serão lidos ou escritos em discos, além de aumentar o desempenho do disco. Por isso, ao invés de efetuar todas as escritas imediatamente depois que elas foram submetidas, o kernel armazena os dados para serem escritos no page cache, utilizando buffers conjuntamente, esperando para ver as escritas podem ser agrupadas para minimizar o movimento de cabeça de disco. Além disso, os dados também são escritos aos poucos em intervalos regulares para minimizar o impacto no desempenho do sistema. Para alcançar esses objetivos é utilizada a infra-estrutura de buffers.

2.2.7 Listas

As páginas de memória do page cache são passíveis de serem paginadas[2]. Elas são agrupados em duas listas, **ativa** e **inativa**, de modo a separar as páginas candidatas a ser despejadas daquelas

[2]Por paginação, entendemos a técnica de aumentar o espaço de memória disponível movendo partes pouco usadas dos processos para dispositivos secundários de armazenamento. A unidade utilizada é uma página,

que se quer manter na memória. A razão para a criação dessas listas foi, segundo [73], "as más interações entre o envelhecimento das páginas e o page flushing, resultavam em páginas limpas que foram referenciadas recentemente eram liberadas antes de páginas sujas antigas". O envelhecimento de páginas é feito no Linux através de um bit de referência, mas em versões anteriores da série 2.4, usava-se um contador de idade.

O código do Linux tem um recurso que atribui baixa prioridade para dados que tendem a ser usados apenas uma vez e não novamente (pelo menos por um longo tempo). Páginas contendo tais dados devem ser identificadas e liberadas o mais rápido possível e usualmente essas páginas são geradas por I/O (entrada e saída, ou E/S) de arquivos, por leituras ou escritas, e residem no page cache. A abordagem do Linux é colocar uma página do page cache inicialmente na lista inativa (ao invés da lista ativa, como fora feito inicialmente), sendo que ela é marcada como não sendo referenciada. Quando ela é acessada pela primeira vez, é marcada como referenciada, mas somente no segundo acesso é que ela é movida para a lista ativa. O efeito disso é que todas as páginas do cache de páginas lidas ou escritas uma vez são colocadas para liberação imediata, mas introduz um atraso antes que que a liberação realmente ocorra. Se a página for acessada em um tempo relativamente curto, ela será resgatada da liberação.

Nas listas de páginas, o Linux utiliza uma variante do algoritmo de aproximação LRU *second-chance* (ou também conhecido como *clock*, sendo que os dois só diferem na implementação).

Dessa maneira, a lista **inativa** em geral agrupa as páginas de memória que foram acessadas menos recentemente que as presentes na lista **ativa**. Para a liberação desses tipos de páginas (que são os tipos passíveis de paginação), o sistema de memória virtual verifica páginas da lista **inativa** que podem ser liberadas (será explicado abaixo), eventualmente escrevendo-as no seu meio de armazenamento se necessário.

O trabalho do sistema de memória virtual é verificar essas listas e executar a movimentação de páginas entre elas. Para decidir quais páginas devem ser liberadas, as páginas nas listas são envelhecidas de acordo com referências que ocorreram desde a última vez que ela foi observada. Uma página que não tenha sido referenciada na lista ativa desde a última observação é movida para a lista inativa (e marcada como referenciada para voltar à lista ativa com somente um acesso). Acessos a páginas da lista inativa as levam de volta à lista ativa. Toda essa verificação e mudança das páginas de uma lista para outra só é feita quando há necessidade de liberação de memória.

Páginas nessas listas, tanto da ativa como da inativa, podem estar sendo usadas, seja por alguma atividade do kernel, buffers ou estar mapeadas para processos (o caso mais comum). Para os casos em que estão mapeadas por processos, é necessário desmapear essas entradas de tabelas de páginas para poderem ser liberadas. Para o caso específico de páginas anônimas, além do desmapeamento, as entradas da tabela de páginas também devem ser remapeadas para outro endereço (um endereço de swap) de modo que essas páginas possam ser encontradas ao serem acessadas novamente. Para outros casos em que não estão mapeadas por processos, as páginas não podem estar sendo usadas por outras partes do kernel do Linux. Para serem liberadas (e também para verificar se o desmapeamento ocorreu), o contador da página é verificado antes da liberação. Além disso, as páginas só podem ter os seus conteúdos liberados se elas não estiverem sujas. Quando esse caso ocorre, o conteúdo da página é primeiramente escrito num meio secundário de armazenamento,

que tem o seu tamanho variável de acordo com a arquitetura de hardware.

como um disco contendo os dados do swap ou de um sistema de arquivos, antes de ser liberado.

2.2.8 Liberação de Memória (*pageout*)

Além das páginas das listas citadas acima, o processo de liberação de páginas envolve páginas utilizadas para outros fins. As páginas que também são verificadas no processo de liberação são os *slab* caches, em que as estruturas em cache não utilizadas são liberadas (a não ser que o *slab* cache explicitamente tinha sido definido que não pode ser reduzido) assim como os dados e estruturas de dados dos caches dos sistemas de arquivos (*inodes*, *dentry* e *quota*).

A seguir é descrita, através das prioridades do Linux para cada tipo de memória, uma visão geral da liberação de memória do Linux, além da ação tomada caso consiga liberar a memória desejada.

Slab caches Tenta-se liberar primeiro a memória necessária de *slab* caches inutilizados, não recorrendo a nenhum outro tipo de memória se a memória liberada por esses caches for suficiente.

Listas Caso não se consiga memória suficiente com os *slab* caches, efetua-se uma intensa tentativa de liberação de memória com as páginas de memória presentes nas listas LRU. A grande maioria das páginas do sistema estarão nessas listas. Caso se consiga liberar memória suficiente, não se verificam outros tipos de memória.

Caches de Sistemas de Arquivos As últimas opções, caso não se liberem páginas suficientes nos dois primeiros casos, são as tentativas de redução dos caches de sistemas de arquivos. Independente se um dos caches liberou memória suficiente, os demais são verificados.

2.2.9 Nós e Zonas

O kernel do Linux provê suporte a arquiteturas NUMA (**N**on-**U**niform **M**emory **A**ccess, ou acesso não-uniforme à memória) [39], nas quais o tempo para acessar determinadas regiões da memória é maior que em outras. Isso é devido ao fato de que algumas regiões da memória estão em localidades diferentes de outras regiões. Em particular, o sistema de VM executa as tarefas necessárias por nó da arquitetura NUMA, se ela existir. Esse código é genérico e caso não exista uma arquitetura NUMA, assume-se que o sistema vigente é uma arquitetura NUMA de apenas um nó.

Existe uma divisão dos endereços lineares, ou os chamados endereços virtuais, entre o kernel e os aplicativos. O padrão do Linux, para arquiteturas de 32 bits, é dividir os 4 Gb de endereçamento linear em 3 Gb para os aplicativos do espaço do usuário e 1 Gb para o kernel, o que limita o tamanho de cada aplicativo em 3 Gb. Por esse motivo, apenas 1 Gb é permanentemente endereçado pelo kernel. Quando o sistema possui mais que 1 Gb de memória, utilizam-se técnicas que mapeiam temporariamente nessa região que é permanentemente mapeada pelo kernel.

A memória de cada nó numa arquitetura NUMA é dividida em zonas. A primeira é constituída dos primeiros 16 Mb do sistema e existe pelo fato de que em algumas arquiteturas só é possível fazer DMA com endereços abaixo de 16 Mb. Por esse motivo, o seu nome é zona DMA. A segunda

zona começa a partir dos 16 Mb e vai até 896 Mb, constituída da memória que é permanentemente mapeada pelo kernel (pela divisão descrita acima). Essa zona não vai até os 1024 Mb (exatamente a divisão do 1 Gb) pelo fato que 128 Mb são reservados para remapear páginas não-contíguas em um endereçamento linear contíguo. A partir dos 896 Mb, a zona é chamada de highmem (high memory). A memória foi dividida em zonas de modo a permitir que cada zona tivesse o seu gerenciamento diferenciado, dado que o uso de cada uma dela é feito diferentemente e com prioridades diferentes.

Nome da Zona	**Faixa de Memória Física**
DMA	0-16 Mb
Normal	16-896 Mb
High Memory	> 896 Mb

Tabela 2.3: Zonas de Memória no Linux.

A partir do momento em que há pouca memória disponível em alguma zona do sistema, a kernel thread `kswapd` é acordada para liberar páginas de memória. Se ela não consegue liberar memória a tempo, isso será feito pela própria chamada de funções de um processo quando estiver no espaço do kernel e precisar alocar memória, por exemplo durante uma falha de páginas. Para decidir quando o `kswapd` ou o próprio processo devem começar a liberar memória, existem parâmetros para cada zona de memória chamados de marcas d'água.

As marcas d'água auxiliam na detecção de quanta pressão de memória existe e elas determinam um limite conhecido como mínimo, baixo e alto. O número de páginas de cada uma das marcas d'água é calculado em função do número de páginas em cada zona, possuindo um limite mínimo de 20 páginas de memória (80 Kb na arquitetura x86) e um limite máximo, que é de 255 páginas (1020 Kb na arquitetura x86).

Abaixo apresentamos uma descrição mais detalhada de cada marca d'água:

- `pages_min`: Menor marca d'água, ela é utilizada em situações de emergência. Nesse caso, existem poucas páginas livres no sistema e qualquer chance de tentativa de liberação de memória deve ser tentada. Por isso, quando ela é atingida, quaisquer alocações de memória no espaço do kernel que não sejam emergenciais terminam por tentar liberar memória através do `kswapd`, mesmo que essas alocações venham a fornecer memória a funções do kernel para processos ativos na memória.

- `pages_low`: Segunda menor marca d'água, é duas vezes o valor do `pages_min`. O `kswapd` é acordado por quaisquer alocações de memória dentro do kernel quando o número de páginas livres está abaixo desse limite. A diferença em relação do `pages_min` é que aqui apenas acordamos o `kswapd`, que irá tentar liberar memória assincronamente.

- `pages_high`: Maior marca d'água, é calculada como sendo três vezes o valor da marca `pages_min`. Essa marca é verificada de tempos em tempos pelo `kswapd` e uma vez que foi atingida, ele começa a liberar memória da zona correspondente até que um número de páginas maior que essa marca seja atingido.

2.3 Código Comentado

Nessa seção, apresentamos algumas importantes funções do sistema de memória virtual. O código-fonte delas, na linguagem de programação C, é apresentado na totalidade abaixo e o seu funcionamento é comentado. Elas correspondem a uma versão adaptada e traduzida do documento preparado pelo autor para o projeto *Linux Kernel Documentation Project* [38].

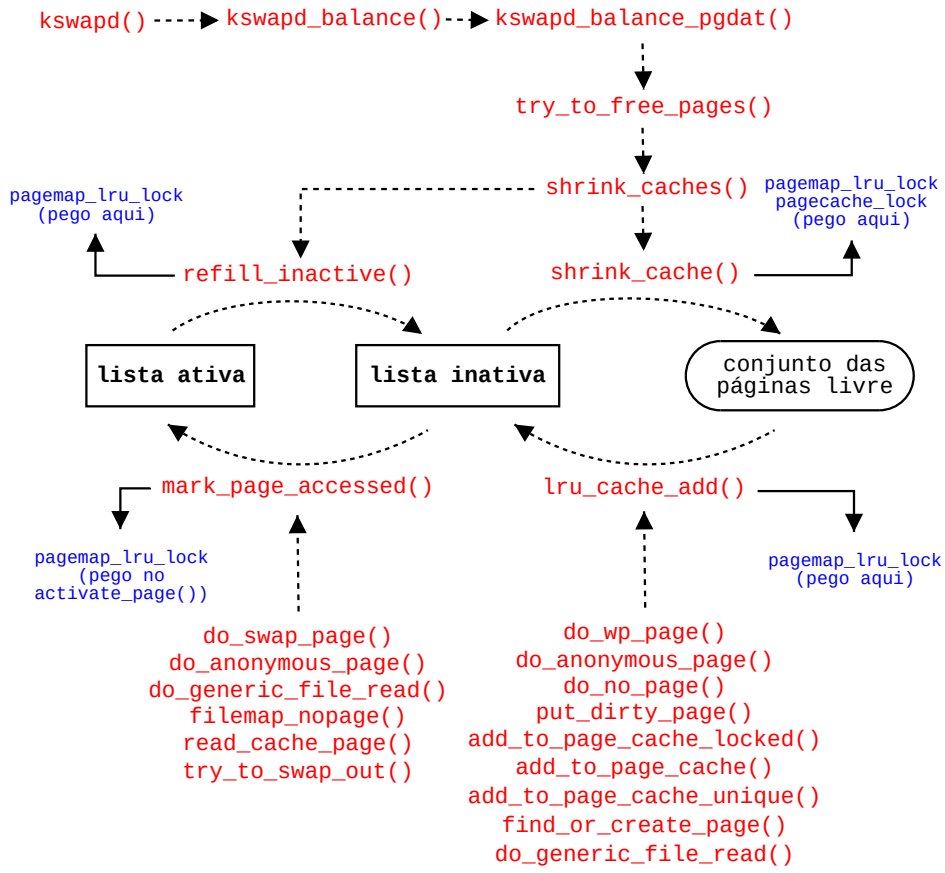

Figura 2.3: Nomes das funções e dos principais *spin locks* que fazem a manipulação de páginas de memória entre as listas ativa e inativa no sistema de memória virtual do Linux.

Na Figura 2.3, temos a listagem das funções que executam a movimentação das páginas entre as listas para a versão 2.4.18 do kernel do Linux. As funções têm os seus nomes terminados como "()", enquanto os nomes terminados por "lock" são os principais *spin locks* de sincronização dessas funções. As setas tracejadas indicam a hierarquia de chamada das funções e as pontilhadas a movimentação das páginas de memória. As funções responsáveis por movimentação são encontradas acima ou embaixo das setas pontilhadas.

As funções apresentadas a seguir são:

– `try_to_free_pages()`

- `shrink_caches()`
- `shrink_cache()`
- `refill_inactive()`
- `try_to_swap_out()`

Antes do seu código-fonte, elas são precedidas por uma breve descrição da sua funcionalidade, dos parâmetros que elas recebem e do tipo e significado do retorno delas. O objetivo aqui é exibir o código da função e explicar detalhes do seu funcionamento, por regiões do código-fonte. A questão de concorrência do código é brevemente vista, e não inclui a concorrência que ocorre em sistemas com mais de um processador. Logo, os *spin locks* não são abordados nos comentários do código.

2.3.1 Função `try_to_free_pages()`

Protótipo:

```
int try_to_free_pages(zone_t *classzone,
                      unsigned int gfp_mask,
                      unsigned int order)
```

Essa função, bastante simples, efetua as tentativas de liberação de páginas de memória de uma determinada zona de memória (o parâmetro `classzone` define qual zona). Essa tentativa de liberação é feita ao chamar a função `shrink_cache()`, que terá a sua prioridade aumentada à medida em que as tentativas forem falhando. A liberação de memória por essa função deve seguir a máscara GFP (get free page), que define qual é o tipo de operação que pode ser feita de acordo com as funções que efetuaram a chamada. Essa máscara é passada pelo parâmetro `gfp_mask`.

No caso em que não for possível liberar o número pré-definido de páginas, estamos na situação de falta de memória. Como conseqüência, algum processo ativo na memória poderá ser finalizado pelo sistema para liberar a sua memória. Isso é feito através da chamada da função `out_of_memory()`.

O seu valor de retorno é um inteiro. Um valor um significa que essa função foi bem-sucedida em liberar o número. O parâmetro `ordem` não é utilizado pela função.

```
int priority = DEF_PRIORITY;
int nr_pages = SWAP_CLUSTER_MAX;

gfp_mask = pf_gfp_mask(gfp_mask);
```

Se a tarefa atual ou alguma das funções que compõem a pilha de chamadas não puder bloquear para a execução de operações de I/O, a macro `pf_gfp_mask()` certifica que a variável `gfp_mask` sinalizará isso.

```
do {
        nr_pages = shrink_caches(classzone, priority, gfp_mask,
                                 nr_pages);
        if (nr_pages <= 0)
                return 1;
} while (--priority);
```

Começando com a prioridade mais baixa (ou seja, o valor mais alto na variável `priority`), tenta liberar o número definido de páginas. Se não for possível liberar esse número, aumenta-se a prioridade (ao decrementar a variável `priority`) e tenta novamente. A prioridade está relacionada ao número de páginas que serão averiguadas na tentativa de liberação.

```
/*
 * Hmm.. Cache shrink failed - time to kill something?
 * Mhwahahhaha! This is the part I really like. Giggle.
 */
out_of_memory();
return 0;
```

Se não foi possível liberar o número suficiente de páginas, mesmo com a prioridade mais alta, verifica se é o momento de finalizar algum aplicativo chamando a função `out_of_memory()`.

2.3.2 Função `shrink_caches()`

Protótipo:

```
int shrink_caches(zone_t * classzone,
                  int priority,
                  unsigned int gfp_mask,
                  int nr_pages)
```

Com um papel muito importante no processo de liberação de páginas, essa função define a prioridade para páginas armazenando cada tipo de dados (do page cache e caches caches slab, de dentries e quota), tentando liberar as páginas na ordem previamente definida.

Dada uma zona de memória (DMA, normal ou highmem), passada pelo parâmetro `classzone`, essa função tenta liberar o número requisitado de páginas (parâmetro `nr_pages`), segundo as permissões da pilha de chamadas do processo de liberação de páginas (`gfp_mask`) e uma prioridade (`priority`) que é usada para saber o quanto se deve tentar liberar páginas contendo um determinado tipo de dado.

O valor devolvido é um inteiro. Um valor zero significa que o número requisito de páginas foram liberadas. Um valor não-zero é o número de páginas que faltou para atingir o número inicialmente requisitado de páginas.

```
int chunk_size = nr_pages;
```

O número requisitado de páginas a serem liberadas é armazenado na variável `chunk_size`, uma vez que ele pode ser mudado e o valor original será necessário abaixo.

```
unsigned long ratio;

nr_pages -= kmem_cache_reap(gfp_mask);
```

Essa é a primeira tentativa para desalocar os objetos "cache" do caches slab que não estão sendo usados. Apenas os caches slab que não apresentam restrições quanto a serem diminuídos sofrem esse processo.

```
if (nr_pages <= 0)
        return 0;
```

Se foi possível liberar o número requisitado de páginas, simplesmente sai da função, devolvendo zero.

```
nr_pages = chunk_size;
```

Diminuir os caches slab pode não liberar o número original de páginas, então tentamos liberar o número original de páginas de outros tipos de páginas (page cache). Restaurar o número original de páginas ao invés de utilizar o número faltante de páginas é utilizado uma vez que o `shrink_cache()` (que será chamado) pode escrever páginas de memória em algum dispositivo secundário de armazenamento, e caso isso ocorra, é interessante que aproveite a oportunidade para escrever um conjunto delas.

```
/* try to keep the active list 2/3 of the size of the cache */
ratio = (unsigned long) nr_pages *
        nr_active_pages / ((nr_inactive_pages + 1) * 2);
refill_inactive(ratio);
```

O primeiro passo para liberar páginas do page cache é voltar a abastecer a lista inativa de páginas, uma vez que somente páginas dessa lista são liberadas. De modo a manter a lista ativa não vazia, por sua vez, é computado quantas páginas, no máximo, devem ser movidas para a lista inativa (variável `ratio`). Observação: o valor um é adicionado ao número de páginas inativas (`nr_inactive_pages + 1`) para tratar o caso onde `nr_inactive_pages` é zero.

```
nr_pages = shrink_cache(nr_pages, classzone, gfp_mask, priority);
```

Independente de quanto a lista inativa foi reabastecida, a função `shrink_cache()` é chamada para diminuir o cache de páginas.

```
if (nr_pages <= 0)
        return 0;
```

Sai da função e devolve zero se o número original requisitado de páginas foi liberado do page cache,

```
shrink_dcache_memory(priority, gfp_mask);
shrink_icache_memory(priority, gfp_mask);
#ifdef CONFIG_QUOTA
shrink_dqcache_memory(DEF_PRIORITY, gfp_mask);
#endif

return nr_pages;
```

Como uma última tentativa, tenta-se diminuir o cache de dentries, inode e também o de quota, se essa estiver habilitada. Mesmo se esses caches tiverem sido reduzidos, retorna o número de páginas que faltaram, ao reduzir o page cache, para atingir o número de páginas original. Essa última tentativa é feita para liberar alguma memória e evitar muitas alocações falhas, mas ela não necessariamente evitará que o sistema chame a função de falha `out_of_memory()`.

2.3.3 Função `shrink_cache()`

Protótipo:

```
int shrink_cache(int nr_pages,
                 zone_t * classzone,
                 unsigned int gfp_mask,
                 int priority)
```

Essa função encolhe o page cache, verificando a lista inativa e tentando liberar páginas dela. Pode ser necessário limpar páginas sujas escrevendo-as, o que será feito quando as permissões de liberação de páginas (parâmetro `gfp_mask`) permitirem.

O valor devolvido é um inteiro. Se zero, significa que a função pode liberar o número de páginas requisitado previamente (parâmetro `nr_pages`). Se diferente de zero, o valor indica quantas páginas faltaram para liberar de modo a atingir o valor requisitado de páginas.

```
struct list_head * entry;
int max_scan = nr_inactive_pages / priority;
int max_mapped = min((nr_pages < < (10 - priority)),
                     max_scan / 10);
```

Aqui é calculado quantas páginas (variável `max_scan`), no máximo, serão verificadas por essa função se ela não puder liberar o número requisitado antes. Esse valor a ser computado é baseado no número de páginas inativas (páginas na lista inativa) e na prioridade (parâmetro `priority`). Nesse caso, quanto menor a prioridade, maior o número de páginas que podem ser verificadas.

O número máximo de páginas mapeadas que pode ser achado durante o processo de verificação também é computado aqui (variável `max_mapped`). Ele será o valor máximo entre o `nr_pages` vezes um valor dependente da prioridade e de um décimo do valor de `max_scan`. Ambos os valores (`max_scan` e `max_mapped`) são exemplos dos conhecidos valores mágicos.

```
spin_lock(&pagemap_lru_lock);
while (--max_scan >= 0 &&
       (entry = inactive_list.prev) != &inactive_list) {
```

Esse código é bastante claro. O número de páginas verificadas é o menor valor entre o número de páginas `max_scan` e a lista inativa toda. Duas outras condições de retorno serão encontradas abaixo. Se o número máximo de páginas mapeadas é atingido ou o número requisitado de páginas liberado, essa função retornará.

```
        struct page * page;

        if (unlikely(current->need_resched)) {
                spin_unlock(&pagemap_lru_lock);
                __set_current_state(TASK_RUNNING);
                schedule();
                spin_lock(&pagemap_lru_lock);
                continue;
        }
```

Aumenta a justiça entre os processos reescalonando o processo corrente se ele estiver consumindo os recursos da CPU por um longo tempo.

```
        page = list_entry(entry, struct page, lru);

        BUG_ON(!PageLRU(page));
        BUG_ON(PageActive(page));

        list_del(entry);
        list_add(entry, &inactive_list);
```

Obtém a página do ponteiro `struct list_head`, e a move para a fim da lista inativa.

```
        /*
         * Zero page counts can happen because we unlink the pages
         * _after_ decrementing the usage count..
         */
        if (unlikely(!page_count(page)))
                continue;
```

Uma vez que a página pode ser removida das listas ativa ou inativa depois de ter o seu contador decrementado, uma condição de disputa (*race condition*) pode ocorrer: essa página é acessada aqui depois de ter o seu contador zerado, mas antes de ser removido das listas. A condição acima cuida desse caso.

```
if (!memclass(page_zone(page), classzone))
        continue;
```

Verifica apenas páginas que são da zona que está sob pressão.

```
/* Racy check to avoid trylocking when not worthwhile */
if (!page->buffers && (page_count(page) != 1 ||
                                        !page->mapping))
        goto page_mapped;
```

Antes de tentar travar a página, primeiro verifica-se se ela está mapeada ou é anônima e não possui buffers a serem liberados. Se alguma das condições for verdadeira, contabilize-a como uma página mapeada (veja abaixo). No caso de possuir buffers, mesmo se mapeado para processos, siga em frente tentando liberá-los (o que pode implicar em sincronizá-los com algum dispositivo secundário de armazenamento).

```
/*
 * The page is locked. IO in progress?
 * Move it to the back of the list.
 */
if (unlikely(TryLockPage(page))) {
        if (PageLaunder(page) &&
                        (gfp_mask & __GFP_FS)) {
                page_cache_get(page);
                spin_unlock(&pagemap_lru_lock);
                wait_on_page(page);
                page_cache_release(page);
                spin_lock(&pagemap_lru_lock);
        }
        continue;
}
```

Tenta travar a página sem liberar a CPU. Se estiver travada e o bit `PageLaunder` estiver ligado, espera até que ela fique destravada. O bit `PageLaunder` será somente ligado para uma página que foi submetida a uma operação de escrita de modo a ser limpa nessa função. A espera pela página acontecerá somente se as permissões do `gfp_mask` permitirem.

Uma referência a essa página é feita (função `page_cache_get()`) antes de esperar por essa página para garantir que ela não será liberada nesse meio-tempo.

```
if (PageDirty(page) && is_page_cache_freeable(page) &&
                                        page->mapping) {
```

Páginas sujas não referenciadas que estão no page cache são candidatas a serem escritas no dispositivo secundário de armazenamento. Mesmo se uma entrada de tabela de páginas acessar essa página, ela será somente remapeada depois que a operação de I/O estiver completa.

```
                /*
                 * It is not critical here to write it only if
                 * the page is unmapped beause any direct writer
                 * like O_DIRECT would set the PG_dirty bitflag
                 * on the physical page after having successfully
                 * pinned it and after the I/O to the page is
                 * finished, so the direct writes to the page
                 * cannot get lost.
                 */
                int (*writepage)(struct page *);

                writepage = page->mapping->a_ops->writepage;
                if ((gfp_mask & __GFP_FS) && writepage) {
                        ClearPageDirty(page);
                        SetPageLaunder(page);
                        page_cache_get(page);
                        spin_unlock(&pagemap_lru_lock);

                        writepage(page);
                        page_cache_release(page);

                        spin_lock(&pagemap_lru_lock);
                        continue;
                }
        }
```

Apenas páginas do page cache que têm a função `writepage()` definida no seu `address space` podem ser limpas. Além disso, as permissões expressas pelo parâmetro `gfp_mask` também é verificada para saber se é permitido que esse processo de liberação de páginas faça operações com o código do sistema de arquivos. Quando ambas são verdadeiras, o bit `PageLaunder` é ligado, o seu bit `Dirty` é desligado, e a função `writepage()` é chamada. Aqui uma referência a essa página é pega para evitar que essa página seja eventualmente liberada no meio-tempo.

```
        /*
         * If the page has buffers, try to free the buffer
         * mappings associated with this page. If we succeed
         * we try to free the page as well.
         */
        if (page->buffers) {
                spin_unlock(&pagemap_lru_lock);

                /* avoid to free a locked page */
                page_cache_get(page);
```

```
if (try_to_release_page(page, gfp_mask)) {
```

Sempre que temos uma página com estruturas auxiliares de I/O buffers, não importa se não é referenciada, tentamos liberá-los chamando a função `try_to_release_page()` que eventualmente chamará `try_to_free_buffers()`. A última função liberará os buffers se eles estiverem limpos, caso contrário os sincronizará com a cópia no dispositivo secundário de armazenamento (se as permissões do `gfp_mask` permitirem).

```
if (!page->mapping) {
        /*
         * We must not allow an anon page
         * with no buffers to be visible
         * on the LRU, so we unlock the
         * page after taking the lru lock
         */
        spin_lock(&pagemap_lru_lock);
        UnlockPage(page);
        __lru_cache_del(page);

        /* effectively free the page here */
        page_cache_release(page);

        if (--nr_pages)
                continue;
        break;
```

A página teve os seus buffers liberados. Uma página anônima, ou seja, sem um `address space`, com buffers precisa ter sido desmapeadas de todos os processos que a mapearam. Também tem que ter sido removida do page cache uma vez que o seu mapeamento teve que invalidar/truncar as suas páginas. Nesse caso, simplesmente remove a página da lista inativa e a libera.

```
} else {
         /*
          * The page is still in pagecache
          * so undo the stuff before the
          * try_to_release_page since we've
          * not finished and we can now
          * try the next step.
          */
          page_cache_release(page);

          spin_lock(&pagemap_lru_lock);
}
```

Os buffers das páginas foram liberados, então sigamos para o próximo passo uma vez que a página ainda está no page cache. Ela precisa ser removida do page cache se estiver completamente desmapeada do processo. Caso contrário, desiste de liberá-la nessa iteração dado que está ainda referenciada e não pode ser liberada nesse momento.

```
                } else {
                        /* failed to drop the buffers
                         * so stop here */
                        UnlockPage(page);
                        page_cache_release(page);

                        spin_lock(&pagemap_lru_lock);
                        continue;
                }
        }
```

Os buffers não puderam ser liberadas, então desiste dessa página nessa iteração. É o momento de tentar outra página da lista das inativas.

```
        spin_lock(&pagecache_lock);

        /*
         * this is the non-racy check for busy page.
         */
        if (!page->mapping || !is_page_cache_freeable(page)) {
                spin_unlock(&pagecache_lock);
                UnlockPage(page);
page_mapped:
                if (--max_mapped >= 0)
                        continue;
```

Para páginas anônimas sem buffers, existe uma verificação de condição de disputa pois elas foram provavelmente removidas do page cache no meio-tempo. Para páginas do page cache que tiveram os seus buffers liberadas e estão ainda mapeadas pelos processos, contabilize-as à variável `max_mapped`.

```
                /*
                 * Alert! We've found too many mapped pages on the
                 * inactive list, so we start swapping out now!
                 */
                spin_unlock(&pagemap_lru_lock);
                swap_out(priority, gfp_mask, classzone);
                return nr_pages;
        }
```

Quando um número `max_mapped` de páginas foram verificadas como sendo referenciadas em alguma parte do código ou mapeadas para processos, começa-se a desmapear as páginas ainda mapeadas. Essa é a razão pela qual a função `swap_out()` é chamada aqui. Depois que é chamada, a função corrente retorna visto que atingir o número `max_mapped` de páginas mapeadas é uma das condições para parar o processo de verificação.

```
        /*
         * It is critical to check PageDirty _after_ we made sure
         * the page is freeable* so not in use by anybody.
         */
        if (PageDirty(page)) {
                spin_unlock(&pagecache_lock);
                UnlockPage(page);
                continue;
        }
```

Aqui verificamos se a página está suja. A razão para essa segunda verificação está em que ela pode ter sido suja depois de ter sido desmapeada por qualquer processo, como na função `__free_pte()`, do arquivo memory.c.

```
        /* point of no return */
        if (likely(!PageSwapCache(page))) {
                __remove_inode_page(page);
                spin_unlock(&pagecache_lock);
        } else {
                swp_entry_t swap;
                swap.val = page->index;
                __delete_from_swap_cache(page);
                spin_unlock(&pagecache_lock);
                swap_free(swap);
        }

        __lru_cache_del(page);
        UnlockPage(page);

        /* effectively free the page here */
        page_cache_release(page);

        if (--nr_pages)
                continue;
        break;
}
spin_unlock(&pagemap_lru_lock);
```

```
return nr_pages;
```

Essa é a parte do código onde uma página não é mapeada por nenhum processo, não está suja e não possui buffers, então pode ser removida do page cache, da lista inativa e finalmente liberada.

2.3.4 Função `refill_inactive()`

Protótipo:

```
void refill_inactive(int nr_pages)
```

Essa função tenta mover um determinado número de páginas (parâmetro `nr_pages`) da lista ativa para a lista inativa. Também atualiza a idade de cada página verificada. A idade é representada pelo bit `Referenced`.

```
struct list_head * entry;

spin_lock(&pagemap_lru_lock);
entry = active_list.prev;
while (nr_pages && entry != &active_list) {
```

Essa função é composta de apenas um laço principal, que pára quando todas as páginas na lista ativa foram verificadas ou o número requisitado de páginas foram movidas para a lista inativa.

```
    struct page * page;

    page = list_entry(entry, struct page, lru);
    entry = entry->prev;
    if (PageTestandClearReferenced(page)) {
        list_del(&page->lru);
        list_add(&page->lru, &active_list);
        continue;
    }
```

Páginas com o bit `Referenced` ligado foram provavelmente acessadas recentemente, então esse bit é desligado e essas páginas são movimentadas para o fim da lista ativa, uma vez que é provável que elas sejam acessadas recentemente.

```
    nr_pages--;

    del_page_from_active_list(page);
    add_page_to_inactive_list(page);
    SetPageReferenced(page);
}
spin_unlock(&pagemap_lru_lock);
```

Páginas que não possuem o bit `Referenced` ligado são consideradas antigas, então podem ser movidas para a lista inativa. Essas páginas são marcadas como sendo `Referenced` de modo que se elas forem acessadas na lista inativa, elas serão movidas de volta à lista ativa depois do primeiro acesso.

2.3.5 Função `try_to_swap_out()`

Protótipo:

```
int try_to_swap_out(struct mm_struct * mm,
                    struct vm_area_struct* vma,
                    unsigned long address,
                    pte_t * page_table,
                    struct page *page,
                    zone_t * classzone)
```

O papel da função `try_to_swap_out()` é tentar desmapear páginas das tabelas de páginas dos processos que as mapeiam. Essa é a primeira etapa de um processo de liberação de páginas, uma vez que páginas podem ser liberadas se todos os processos que as mapeavam não as mapearem mais e não houver referências do kernel. Desmapear significa que, dada uma entrada de tabela de página (`pte`), ou ela será zerada (para páginas de arquivos mapeados) ou remapeada para um endereço de swap (páginas anônimas). Em ambos os casos, o bit `Present` da nova entrada de tabela de página estará desligado. Por esse motivo, o processo ao qual essa página pertence não será capaz de acessar seus dados diretamente, causando uma falha de página num acesso futuro.

Essa função devolve um valor inteiro. Esse valor será zero se nenhuma página que possa ser liberada (i.e., uma página não mapeada mais por nenhum processo nem referenciada por qualquer código) tiver sido desmapeada. Isso acontecerá mesmo no caso em que uma página foi desmapeada de um processo, mas ainda é mapeada por outros processos. O valor de retorno será um se uma página foi liberada do seu último processo (nenhum processo está mapeando-a nem algum trecho do código referenciando-a no momento em que essa função termina).

```
pte_t pte;
swp_entry_t entry;

/* Don't look at this pte if it's been accessed recently. */
if ((vma->vm_flags & VM_LOCKED)
    || ptep_test_and_clear_young(page_table)) {
        mark_page_accessed(page);
        return 0;
}
```

Essa é parte do processo de envelhecimento do VM. Aqui, baseado no bit `Young` da entrada da tabela de páginas, `try_to_swap_out()` define essa página como acessada (bit `Accessed`). Se essa

página já é definida como acessada (i.e., a segunda vez em que é acessada) e ela ainda está na lista inativa, `mark_page_accessed()` moverá essa página para a lista ativa. A página previamente marcada como acessada terá o bit `Accessed` desligado.

A página também será marcada como acessada se essa `vmarea` estiver travada pela chamada de sistema `mlock`.

```
/* Don't bother unmapping pages that are active */
if (PageActive(page))
        return 0;
```

Página ativas são supostamente acessadas freqüentemente. Por esse motivo, não é vantajoso desmapeá-las uma vez que é provável que elas sejam remapeadas em breve.

```
/* Don't bother replenishing zones not under pressure.. */
if (!memclass(page_zone(page), classzone))
        return 0;
```

Também não é razoável liberar páginas de outras zonas que não a que estamos focando no momento, mesmo que elas venham a ter pouca memória disponível.

```
if (TryLockPage(page))
        return 0;
```

Tentamos aqui travar a página sem liberar a CPU. A razão é que o processo de desmapeamento não é dependente de uma página específica e não vale a pena dormir para tentar desmapear uma página qualquer.

```
/* From this point on, the odds are that we're going to
 * nuke this pte, so read and clear the pte.  This hook
 * is needed on CPUs which update the accessed and dirty
 * bits in hardware.
 */
flush_cache_page(vma, address);
pte = ptep_get_and_clear(page_table);
flush_tlb_page(vma, address);
```

Os dados da entrada da tabela de páginas são lidos na variável `pte`. Também a entrada da tabela de páginas é limpa para evitar tê-la modificada no meio-tempo (por exemplo, nos casos onde CPUs que atualizam bits como `Accessed` e `Dirty` em hardware, como explicado no comentário do código).

```
if (pte_dirty(pte))
        set_page_dirty(page);
```

```
/*
 * Is the page already in the swap cache? If so, then
 * we can just drop our reference to it without doing
 * any IO - it's already up-to-date on disk.
 */
if (PageSwapCache(page)) {
        entry.val = page->index;
        swap_duplicate(entry);
set_swap_pte:
        set_pte(page_table, swp_entry_to_pte(entry));
```

No caso em que essa página foi adicionada ao swap cache (que é parte do page cache), há apenas a necessidade de incrementar o contador de swap (através da função `swap_duplicate()`) e ajustar a entrada da tabela de páginas para o endereço de swap para o qual a página do swap cache já está definida. O endereço de swap é definido no campo `index` da `struct page`.

```
drop_pte:
        mm->rss--;
```

O processo que possui essa página desmapeada de suas tabela tem o número de páginas residentes, ou RSS, decrementado.

```
        UnlockPage(page);
        {
                int freeable = page_count(page) - !!page->buffers <= 2;
                page_cache_release(page);
                return freeable;
        }
}
```

Se não há mais usuários dessa página (ou seja, referências, incluindo processos a mapeando), o valor de retorno será um, visto que essa página pode ser liberada. Caso contrário, o valor de retorno será zero.

```
/*
 * Is it a clean page? Then it must be recoverable
 * by just paging it in again, and we can just drop
 * it..  or if it's dirty but has backing store,
 * just mark the page dirty and drop it.
 *
 * However, this won't actually free any real
 * memory, as the page will just be in the page cache
 * somewhere, and as such we should just continue
```

```
 * our scan.
 *
 * Basically, this just makes it possible for us to do
 * some real work in the future in "refill_inactive()".
 */
if (page->mapping)
        goto drop_pte;
if (!PageDirty(page))
        goto drop_pte;

/*
 * Anonymous buffercache pages can be left behind by
 * concurrent truncate and pagefault.
 */
if (page->buffers)
        goto preserve;
```

Páginas anônimas com buffers foram mapeadas para um `address space`, mas não são mais devido às operações concorrentes de truncamento e de falha de página.

```
/*
 * This is a dirty, swappable page.  First of all,
 * get a suitable swap entry for it, and make sure
 * we have the swap cache set up to associate the
 * page with that swap entry.
 */
for (;;) {
        entry = get_swap_page();
        if (!entry.val)
                break;
        /* Add it to the swap cache and mark it dirty
         * (adding to the page cache will clear the dirty
         * and uptodate bits, so we need to do it again)
         */
        if (add_to_swap_cache(page, entry) == 0) {
                SetPageUptodate(page);
                set_page_dirty(page);
                goto set_swap_pte;
        }
```

Essa é uma página anônima e suja, então vamos adquirir um endereço de swap para atribuir-lhe de modo a remapear a sua entrada de tabela de páginas para esse novo endereço. Assim que o

endereço de swap foi adquirido, essa página será adicionada ao swap cache, que irá, por sua vez, adicionar ao page cache e à lista inativa.

Dado que essa página não possui um dispositivo secundário de armazenamento, essa página precisa ser marcada como suja de modo a não ser liberada antes de ser armazenada no swap.

```
        /* Raced with "speculative" read_swap_cache_async */
        swap_free(entry);
}
```

Quando uma falha de página para um endereço de swap é tratada, algumas páginas são lidas antecipadamente se a página da entrada que ocasionou a falha não estiver presente no swap cache. Nessa situação, uma página poderá ter sido adicionada ao swap cache pelo código de leitura antecipada (*read-ahead*) com o endereço de swap acima, mas antes desse caminho do código ter adicionado a página ao cache. Então é necessário diminuir o contador desse endereço de swap e adquirir um novo.

```
/* No swap space left */
preserve:
set_pte(page_table, pte);
UnlockPage(page);
return 0;
```

Não havia endereços de swap disponíveis, então não há espaço livre no swap, seja por não haver swap ou o(s) swap(s) está(rem) completamente usado(s). Por isso, essa função é incapaz de desmapear essa página anônima. Logo, define a entrada da tabela de páginas de volta ao valor original e devolve zero, uma vez que nenhuma página que pudesse ser liberada foi desmapeada depois dessa tentativa.

Capítulo 3

Cache Comprimido

Nesse capítulo apresentamos o conceito do cache comprimido e como esse conceito foi implementado no Linux. Iniciamos através de uma visão geral do cache onde o seu funcionamento básico e as estruturas básicas da implementação são explicadas. Fazemos então algumas considerações de overhead, e a seguir apresentamos as mais importantes decisões de projetos que nortearam a nossa implementação. Mostramos também a razão pela escolha do Linux na implementação e uma breve apresentação dos algoritmos de compressão para os quais demos suporte na implementação e que utilizamos nos nossos experimentos.

3.1 Introdução

O tempo médio de acesso às páginas é calculado em função do tempo de acesso às páginas nos diversos níveis da hierarquia de memória, que inclui memória cache, a memória principal, e a área de swap em disco. Uma das principais causas da degradação do tempo de acesso está exatamente localizada nos caros tempos de acesso ao disco, já que o mesmo é cerca de 5 a 6 ordens de magnitude mais lento que a memória principal. Em muitos sistemas, melhora-se o desempenho do sistema de memória virtual introduzindo-se um disco especial de acesso rápido para exercer o papel de swap. Outra maneira, nem sempre fácil, é diminuir o número de acessos; e um dos modos de se fazer isso é a inserção de um novo nível de cache entre a memória principal e o disco.

O *cache comprimido* é uma das técnicas propostas para se diminuir o número de acessos a dispositivos secundários de armazenamento, como discos rígidos. A técnica consiste em dividir a memória principal em duas partes (Fig. 3.1): a *memória não-comprimida* e o *cache comprimido*. Na primeira parte, armazenam-se as páginas em seu estado natural, e essas páginas são acessadas pela CPU, possivelmente através de intervenções do gerenciador de memória; na segunda, armazenam-se as páginas comprimidas por algoritmos especiais de compressão. A principal conseqüência da manutenção de páginas no estado comprimido na memória é que, pelo fato de elas usualmente ocuparem menos espaço para serem armazenadas, é possível armazenar um número maior de páginas. Isso resulta em um aumento do tamanho efetivo da memória – pois armazenam-se mais páginas em memória – e dessa maneira, diminui-se o acesso a dispositivos de armazenamento, como discos

rígidos. Contudo, há custos envolvidos. Seja o custo direto do overhead gerado pela compressão e descompressão das páginas de memória, seja o custo indireto da diminuição da memória disponível para certas atividades do sistema operacional.

Uma maneira pesquisada de se tentar diminuir o custo de compressão e descompressão foi através do uso de algoritmos de compressão especialmente desenvolvidos para a compressão de páginas de memória [80, 26]. Estes algoritmos são rápidos, necessitam de pouca memória auxiliar e levam em conta as particularidades (como padrões) de dados do segmento de dados dos processos que estejam na memória, procurando evitar altos overheads. No entanto, eles não comprimem tanto quanto outros algoritmos que não são voltados a dados de memória [51]. Ademais, estudos anteriores [18, 31, 8, 32, 26, 1] mostraram que os dados de memória, independente se a definição inclui todos os dados de memória ou apenas dados dos segmentos de dados, possuem uma compressibilidade que pode ser explorada.

Ao longo dos últimos anos, os tempos de acesso a disco têm diminuído muito lentamente se compararmos com a diminuição dos tempos de execução das instruções dos processadores. Enquanto a taxa média de aumento da velocidade das CPUs é de cerca 60% por ano, a taxa de latência (acesso) dos discos decresce a menos de 10% anuais (veja [26, 13]), o que resulta, numa análise de ciclos de processamento, em uma diferença de 5 a 6 ordens de magnitude. Observa-se que, uma vez que a diferença entre o desempenho da CPU e do disco é grande – e a perspectiva é que ela aumente cada vez mais, o uso do disco para armazenar dados de programas que estão na memória (swap) tornar-se-á relativamente cada vez mais caro em termos de tempo de acesso (ciclos de CPU). Assim, o overhead introduzido por um sistema como o cache comprimido tornar-se-á cada vez mais desprezível.

Mostraremos aqui que resultados insatisfatórios e não conclusivos de implementações anteriores devem-se parcialmente a problemas que abordamos nesse trabalho. Essa análise vem através de uma implementação no Linux 2.4.18, que nos permitiu avaliar diversos aspectos da influência do cache comprimido no restante do sistema, ao contrário de uma simulação, que permite o desenvolvimento em um ambiente controlado, mas falta a visão geral das conseqüências possíveis no ambiente no qual o projeto está inserido.

Na Seção 3.2, apresentamos brevemente a razão pela qual o Linux foi escolhido para ser o sistema no qual implementamos o cache comprimido. Na Seção 3.3, daremos uma visão geral da implementação, e faremos considerações de overhead que a implementação introduz na Seção 3.4. Finalmente, na Seção 3.5, descrevemos importantes decisões de projeto.

3.2 Por que Linux?

O sistema operacional Linux foi escolhido por ser um software de fonte livre (ver licença GPL [22]), o que nos permite ter acesso a seu código-fonte, estudá-lo e alterá-lo. Além disso, ele possui tecnologia de software de alto nível, além de contar com desenvolvedores muito experientes e com grande conhecimento para ajuda e esclarecimento. Entre os sistemas operacionais livres, apesar de possuir pouca documentação, provavelmente é o que possui a maior quantidade disponível, que tende cada vez a aumentar com a sua popularização. Por fim, é o sistema mais difundido no mundo, o que nos permite uma maior visibilidade do projeto, com uma base significa-

tiva de usuários, o que permite maiores testes e também opiniões e sugestões de modo a melhorar a implementação.

3.3 Visão Geral

Numa arquitetura de cache comprimido, a memória principal é dividida em memória não-comprimida e o cache comprimido. As páginas de memória são comprimidas e armazenadas no cache comprimido quando o sistema de memória virtual despeja (*evicts*) algumas páginas da memória para conseguir espaço na memória para novas alocações. Essas páginas podem ser provenientes do segmento de dados de um processo ou de uma página que está em algum dos caches do sistema, por exemplo.

O algoritmo de compressão utilizado pelo cache comprimido deve comprimir os dados sem perda pois precisamos ser capazes de recuperar corretamente esses dados. Além disso, para ser viável a sua utilização, deve conseguir atingir uma boa taxa de compressão em um tempo razoável. Em casos extremos, altas taxas de compressão são obtidas a partir de páginas que só possuam zeros. Por outro lado, há casos em que os dados da páginas não são passíveis de compressão, podendo até mesmo a ter um tamanho aumentado após a compressão. Quando esse caso acontece, a página é armazenada no seu estado não-comprimido. Por simplicidade, ao longo desse trabalho, qualquer página armazenada no cache comprimido é conhecida como *página comprimida*, independente da forma (comprimida ou não) na qual foi armazenada. Uma página na memória não-comprimida é conhecida como *página não-comprimida*. Qualquer atributo que uma página não-comprimida tinha no momento que foi despejada é herdado pela página comprimida. Como exemplo, uma página comprimida suja é uma página que estava suja quando foi comprimida.

Estudos e implementações anteriores projetaram caches comprimidos que armazenavam apenas páginas que poderiam ser armazenadas no swap. Ao contrário deles, na nossa implementação todas as páginas que podem ser armazenadas em qualquer dispositivo secundário de armazenamento (incluindo, por exemplo, páginas do cache de arquivos) são candidatas a serem comprimidas e armazenadas no cache comprimido. A razão por trás dessa decisão será discutida na Seção 3.5.1.

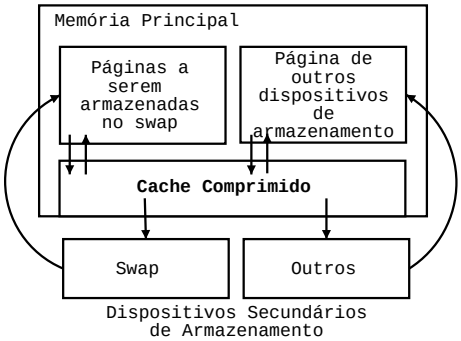

Figura 3.1: Hierarquia de memória com o cache comprimido

Em nossa implementação, no momento em que não há espaço disponível no cache comprimido

para inserir uma nova página, o cache comprimido tem duas opções:

1. Alocar mais memória para o seu uso. Nessa opção, o cache comprimido aumenta o seu tamanho em uma célula (veja definição abaixo), disponibilizando essa memória para armazenar novas páginas comprimidas.

2. Liberar algumas páginas comprimidas. Essa opção é tomada quando decide-se não alocar mais memória para o cache comprimido e não é possível reagrupar as páginas comprimidas (chamada de compactação, veja abaixo) para disponibilizar espaços fragmentados.

A decisão sobre qual dessas ações acima deve ser tomada depende do comportamento recente do sistema, e será discutida com detalhes na Seção 4.3. Discutiremos esta última ação agora, ao passo que a ação de alocar mais memória é descrita mais tarde nessa seção.

No cache comprimido, quando uma página comprimida precisa ser liberada, a página mais antiga[1] é escolhida. Entretanto, antes de serem liberadas, páginas comprimidas sujas precisam primeiramente ser escritas no dispositivo secundário de armazenamento. Antes de serem submetidas à operação de escrita, essas páginas podem ter os dados descomprimidos, dependendo dos dados que armazenam. Na nossa abordagem, apenas páginas comprimidas com dados que podem ser armazenados no swap são escritas na forma comprimida. Páginas comprimidas a serem armazenadas em outros dispositivos secundários de armazenamento são primeiro descomprimidas, dado que um sistema de arquivos, por exemplo, supõe que os seus dados sejam escritos na sua forma natural.

Cada página armazenada comprimida no swap é armazenada num único bloco do dispositivo de swap (que é do tamanho da página). Esse procedimento é comumente conhecido como *null-padding* apesar de não haver, na prática, o trabalho de zerar o restante do bloco como é usual no procedimento. Armazenar essas páginas no dispositivo de swap na forma comprimida adia a descompressão para a operação de "swapin" e evita descompressão de dados nunca reaproveitados pelo sistema[2]. Detalhes sobre essa decisão de projeto são apresentados na Seção 3.5.5.

Páginas comprimidas requisitadas por qualquer operação do kernel de modo a serem imediatamente usadas são ditas requisitadas (*reclaimed*). Essa definição inclui uma página requisitada por uma falha de página e uma página armazenando dados de um bloco de um dispositivo que esteja em cache na memória. Páginas requisitadas são removidas do cache comprimido, descomprimidas, e os seus dados são armazenados em páginas de memória alocadas. Se uma página comprimida tiver sido armazenada num dispositivo de armazenamento e não estiver presente no cache comprimido (nem na memória não-comprimida), ela é lida do dispositivo de armazenamento – e descomprimida se o dispositivo é o swap. Nesse caso, ela não é adicionada ao cache comprimido quando lida do dispositivo de armazenamento, apenas ao page cache, que fica na memória não-comprimida. Veja a Figura 3.1 para uma hierarquia completa.

Um cache comprimido adaptativo precisa alocar memória para si de forma que o seu tamanho varie durante a execução do sistema. Por essa razão, o gerenciamento do espaço de memória alocado

[1] A ordem na qual as páginas comprimidas são liberadas segue a ordem na qual elas foram armazenadas no cache comprimido.

[2] Páginas que são lidas antecipadamente ("read-ahead") são apenas decomprimidas se algum processo falha nelas, i.e., elas são mapeadas de volta por uma tabela de páginas de um processo.

para o cache comprimido é feito através de um esquema de paginação, pois a infra-estrutura nos sistemas operacionais para o seu uso já está disponível e é robusta. No cache comprimido, a menor quantidade de memória que pode ser alocada ou desalocada é conhecida como *célula* e uma célula é formada por um número constante de *páginas de memória contíguas*, sendo usada para armazenar uma ou mais páginas comprimidas. É importante notar que duas células alocadas consecutivamente não necessariamente têm endereços contíguos.

Em uma célula, o *espaço livre final* é a região contígua no final dela que não armazena nenhuma página comprimida. Quando uma página é comprimida no cache comprimido, procura-se a célula com o menor espaço livre final onde a página comprimida caiba e a armazena no início do espaço livre final. No momento em que uma página é liberada do cache comprimido, seja porque o cache comprimido está cheio, seja porque a página foi requisitada por qualquer operação do kernel, ela é simplesmente removida da célula onde estava armazenada. Para evitar overhead desnecessário, nenhuma movimentação de páginas comprimidas dentro das células é feita quando páginas são adicionadas ou removidas do cache comprimido. Isto leva à fragmentação da área disponível ao armazenamento das páginas comprimidas dentro de uma mesma célula. O *espaço livre* de uma célula consiste da soma de espaço em todas as regiões na célula que não são usadas para armazenar alguma página comprimida. Veja uma ilustração destes conceitos no exemplo da Figura 3.2.

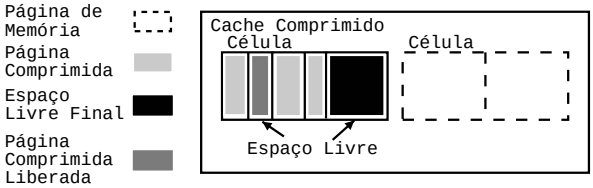

Figura 3.2: Uma célula no cache comprimido

Dependendo da utilização do cache comprimido, pode ser que não se encontre nenhuma célula cujo espaço livre final é grande o suficiente para armazenar uma nova página comprimida, mas pode existir uma página cujo espaço livre seja suficiente. Nesse caso, uma célula com o menor espaço livre onde a página comprimida pode ser armazenada é selecionada. Então, essa célula é *compactada*, i.e., todas as páginas comprimidas são movimentadas de modo que não exista espaço livre entre elas, tornando todo o espaço livre disponível como espaço livre final. Antes de aumentar o cache comprimido ou liberar qualquer página comprimida, nossa implementação tenta primeiro uma compactação (veja Figura 3.3).

3.4 Considerações de Overhead

O tempo gasto pelos algoritmos de compressão é a nossa primeira preocupação em relação ao overhead. Além do tempo para comprimir ou descomprimir uma única página, que será discutido na Seção 6.4.4, o número total de compressões e descompressões de páginas também deve ser levado em conta numa visão global dos custos e benefícios. Essa quantidade depende de quanta memória os processos usam, os seus padrões de acesso, e o uso de memória do cache comprimido em determinado

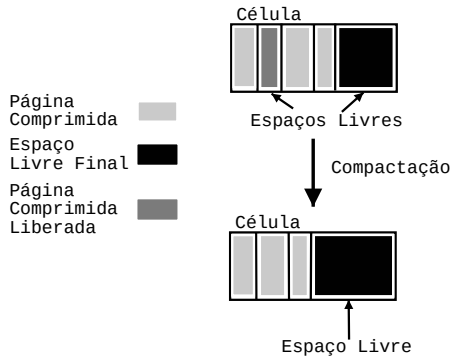

Figura 3.3: A compactação em uma célula do cache comprimido.

momento. Podemos ter casos onde o tempo total gasto comprimindo e descomprimindo páginas pode ser demasiado mesmo que haja redução dos acessos ao dispositivo de armazenamento em decorrência do maior número de páginas presentes na memória. O tempo gasto com essas operações é particularmente grande quando há um alto número de páginas comprimidas e descomprimidas. Esses são os casos onde contando os custos de compressão e descompressão de páginas são maiores que o benefício provido pelo cache comprimido.

Também podemos detectar algum overhead quando, na maior parte do tempo, existem processos prontos para rodar na CPU, mesmo que muitas operações de I/O sejam feitas concorrentemente à execução dos processos. Nesse caso, se todos as compressões e descompressões utilizarem mais que o tempo ocioso da CPU, esse uso penaliza os processos em execução, tornando as suas execuções mais demoradas.

Outras considerações importantes envolvem o efeito do cache comprimido no sistema, como a diminuição da memória disponível que pode ser diretamente mapeada por tabelas de páginas de processos. Isso ocorre pelo fato de que o cache comprimido aloca uma quantidade de memória para armazenar páginas comprimidas. Por essa diminuição, o número de falhas de páginas tende a crescer. Alocações de páginas também tendem a crescer por duas razões: *(i)* mais páginas são necessárias para atender o maior número de falhas; *(ii)* menos blocos provavelmente possuirão seus dados mantidos em cache na memória, logo mais alocações são necessárias para prover páginas para os blocos que serão armazenados e removidos do cache com maior freqüência (veja Seção 3.5.1) do que se houvesse uma quantidade maior de memória para armazenar os dados dos blocos por mais tempo. Como conseqüência, o overhead introduzido pelo cache comprimido também é composto dos custos de tratar falhas de páginas e liberações adicionais de páginas. Esses custos, notadamente os das funções de liberação de páginas, podem ser substanciais.

Outro efeito importante do cache comprimido é o *overhead dos metadados* que ele introduz. Cada célula alocada para o cache comprimido precisa de metadados[3] sobre a(s) página(s) comprimidas que ela armazena. Além disso, cada página comprimida tem metadados sobre dados da

[3]Em comparação com um cache comprimido de mesmo tamanho e células compostas de uma página de memória, apenas metade das estruturas de dados para esses metadados é necessária quando células com duas páginas de memória contíguas são usadas.

página original que foi comprimida, além de dados auxiliares para o gerenciamento do cache comprimido, como a sua localização na célula ou posição em tabelas auxiliares de busca. Dependendo do número de células, i.e., o tamanho do cache comprimido, e de quantas páginas são armazenadas nele, o espaço de memória usado por essas estruturas de dados pode ser bastante significativo, por isso um sistema de cache comprimido deve também levar em conta os custos de metadados que a sua implementação requer. Sabendo que os custos de metadados podem ser significativos, algoritmos e estratégias mais simples podem ser mais efetivos, particularmente se considerarmos que o sistema será usado sob pressão de memória.

No Linux, páginas podem ser lidas ou escritas em um dispositivo de bloco usando estruturas auxiliares chamadas de *buffers*, que armazenam dados de blocos em páginas do page cache (page e buffer cache são explicados nas Seções 2.2.4 e 2.2.6). Ao contrário das operações de I/O que não utilizam buffers – nesse caso marcam as próprias páginas que armazenam os dados como limpas ou sujas, – quando as estruturas auxiliares de buffers são usadas, os buffers em si são marcados como limpos ou sujos ao invés das páginas de memória que contêm os seus dados. No processo de liberação de páginas (como visto na Seção 2.2.8), as páginas contendo dados de buffers sujos precisam ser escritas nos dispositivos de armazenamento antes de se tornarem candidatas à liberação. Ademais, dado que o número de alocações de páginas é maior quando o cache comprimido é utilizado (como descrito acima), muito mais buffers sujos podem ter que escrever os seus dados para permitir que as páginas os armazenando sejam eventualmente liberadas pelo sistema. Por isso, o cache comprimido, numa tentativa de reduzir leituras dos dispositivos de armazenamento, pode aumentar o número de operações de escritas.

Inicialmente, adicionamos suporte para o armazenamento de páginas contendo dados de buffers sujos, mas ele foi removido por algumas razões:

1. Não atingia praticamente nenhum ganho de desempenho. Em alguns casos, havia certamente um ganho de desempenho pelo fato de que há uma diminuição das operações de escrita, mas na maior parte, o impacto desse menor número de escrita de dados de buffer não resultava em melhora de desempenho.

2. Essas páginas não podiam ser comprimidas devido ao código de tratamento de buffers. Isso ocorre pois o código de buffers no Linux acessa essas páginas diretamente para as suas operações de escritas regulares, ou quando um determinado bloco é requisitado por algum aplicativo ou pelo próprio kernel.

3. O suporte implicava mudanças indesejáveis à estrutura do cache comprimido. Nesse caso, as mudanças podem ser explicitadas por uma grande quantidade de casos especiais.

3.5 Decisões de Projeto

Nessa seção, nós damos mais detalhes sobre algumas importantes decisões de projeto na nossa implementação. Entre elas, apresentamos o suporte inédito ao cache de arquivos, a ordenação das páginas dentro do cache comprimido, as células com páginas contíguas de memória e outras decisões que foram fundamentais para os resultados que alcançamos e que são apresentados nesse trabalho.

3.5.1 Page Cache

Todos os estudos anteriores propuseram ou implementaram caches comprimidos apenas para páginas que pudessem ser armazenadas no swap. Quando um cache comprimido que armazena somente esses tipos de páginas é utilizado, o espaço alocado para o cache comprimido, além de toda a sua infra-estrutura de gerenciamento, como fazem parte os metadados, são utilizados apenas para um tipo de dado. Logo, a memória é efetivamente aumentada para esses dados que podem, além de utilizarem a memória não-comprimida, também utilizar a parte da memória alocada para o cache comprimido, mas tem um impacto negativo em páginas armazenando outros tipos de páginas.

Com o uso da memória alocada para o cache comprimido por apenas um tipo de dados, esses dados podem usufruir da compressão para a diminuição dos acessos ao swap. Entretanto, todos os demais caches do sistema tornam-se menores uma vez que eles têm menos memória disponível para competir. Em particular, no Linux, os caches de sistema são o page cache, e caches de estruturas de dados internas do kernel, conhecido genericamente como *slab caches*. Exemplos de slab caches para estruturas de dados internas ao kernel são o cache de buffer, inode, dentry e quota.

O cache comprimido tem uma forte tendência a influenciar o page cache, por ele ser comumente muito maior que os outros caches. Isso ocorre pelo fato que o page cache armazena um grande número de páginas de memória, que tem vários kilobytes de tamanho (no caso do i386, 4 Kb), ao passo que os demais caches armazenam estruturas de dados da ordem de bytes de tamanho.

Páginas armazenando dados de blocos de todos os dispositivos de armazenamento (como dados de buffer, de arquivos normais, metadados de sistemas de arquivos e mesmo páginas com dados do swap) são armazenadas no page cache. Assim como outros caches de sistema, o page cache possui uma grande tendência de ser menor em um sistema com um cache comprimido que armazena apenas páginas armazenáveis no swap. Como uma conseqüência dessa possível redução de tamanho, blocos (usualmente de arquivos regulares) terão menos páginas com os seus dados espelhados na memória (em cache), o que conseqüentemente deve aumentar o total de I/O, pois processos que achariam os dados de blocos requisitados na memória não os acharão mais, tendo que submeter leituras ao disco. E aqui é importante notar que usualmente leituras a mais se refletem em um pior desempenho dos processos ativos.

Nessa pesquisa sobre o cache comprimido, motivados por esse impacto negativo em outros caches, verificamos que ele não deve levar em conta apenas a sua eficácia para um determinado tipo de dado, como é o caso dos caches comprimidos para dados que podem ser armazenados no swap, mas também o quanto o seu uso dos recursos do sistema pode degradar o desempenho geral. Em nossos experimentos na Seção 6.4, mostraremos testes que evidenciam esse impacto negativo no desempenho de alguns aplicativos.

Em nossa abordagem, escolhemos também armazenar no cache comprimido páginas cujos dados não são armazenáveis em swap. Esses dados são apenas os que podem ser armazenados em algum dispositivo secundário de armazenamento. Isto exclui páginas usadas para dados do kernel, como tabela de páginas de processos, pois implicaria a criação de uma infra-estrutura para efetuar a paginação com essas páginas. Isto envolveria grandes modificações no kernel, além da complexidade teórica, contrário à nossa política de mínima intrusão no Linux.

Como o page cache no Linux contém todas as páginas que podem ser armazenadas em algum dispositivo secundário de armazenamento, provemos total suporte ao cache comprimido para

armazenar páginas provenientes desse cache.

3.5.2 Ordenação das Páginas

Em nossa implementação do cache comprimido, temos a preocupação de manter as páginas comprimidas na ordem em que elas foram liberadas da memória não-comprimida pelo sistema de memória virtual e armazenadas no cache comprimido. Como nós verificamos em experimentos no Linux, que utiliza um política de substituição de páginas baseada na página menos recentemente usada (less recently used, ou LRU), a não manutenção da ordem na qual as páginas comprimidas foram armazenadas no cache comprimido raramente aumenta o desempenho do sistema e normalmente o degrada severamente.

Discutiremos a seguir uma situação que a ordenação das páginas do cache comprimido pode ser alterada. Essa situação é ocasionada pela leitura antecipada de dados, operação que é comumente implementada nos sistemas operacionais numa tentativa para aumentar o desempenho dos aplicativos que efetuam leituras em dispositivos secundários de armazenamento com discos rígidos.

O Linux, como a maioria dos sistemas operacionais, quando lê os dados de um determinado bloco do dispositivo de armazenamento, também lê blocos adjacentes antecipadamente. A razão pela qual outros blocos adjacentes são lidos conjuntamente é que leitura desses possui um custo substancialmente menor que a leitura do primeiro bloco. Isso ocorre em dispositivos como disco rígido, que apesar da sua alta taxa de transferência, possuem tempo de acesso e de procura muito grandes. Ler blocos antecipadamente é conhecido como *read-ahead* e os blocos lidos à frente são armazenados em páginas do page cache na memória não-comprimida.

Quando ocorre uma operação de read-ahead em um sistema sem cache comprimido, a leitura desses dados antecipadamente pode forçar a liberação de algumas páginas da memória não-comprimida de modo a armazenar esses dados. Mesmo que a leitura ocorra em um sistema com memória livre suficiente, essas páginas serão consideradas mais recentemente usadas que um determinado número de páginas na memória, número esse dependente dos acessos que se teve às páginas anteriormente presentes na memória. Contudo, não é necessariamente verdade que essas páginas foram mais recentemente usadas, apesar de terem sido lidas. Nesse caso, elas devem ser consideradas menos recentemente usadas que qualquer página na memória, exceto páginas que foram lidas antecipadamente e se encontram nas mesmas condições de recentidade.

Assim que o cache comprimido é introduzido no sistema, a situação descrita acima, além de ocorrer com os dados lidos antecipadamente de blocos dos dispositivos secundários de armazenamento, também tende a ocorrer com dados lidos do cache comprimido. Abaixo temos os cenários em que a situação de alteração da ordenação das páginas ocorre envolvendo dados do cache comprimido:

- **Leitura de dispositivo secundário de armazenamento.**

  Nesse caso, a leitura é submetida ao dispositivo e a leitura de blocos adjacentes também é requisitada. No caso em que os dados requisitados já estão presentes em uma página na memória não-comprimida, não é submetida a leitura dos seus dados ao dispositivo. Além disso, essa página não é modificada, nem a sua idade alterada.

Entretanto, se os dados do bloco requisitado estão no cache comprimido, é seguida a operação padrão para dados não presentes na memória não-comprimida, que consiste em tornar esses dados disponíveis nessa memória. Dessa forma, as dados armazenados no cache comprimido são descomprimidos para a memória não-comprimida.

– **Requisição de página comprimida.**

Por padrão, requisições de páginas que não estão disponíveis na memória não-comprimida executam a operação de *read-ahead*. Dessa maneira, sempre que uma página comprimida é lida para a memória não-comprimida, páginas no próprio cache comprimido contendo dados de blocos adjacentes são descomprimidas. Além dessas, também são submetidas leituras a disco para os dados que não estão armazenados por páginas na memória não-comprimida nem no cache comprimido.

Em ambos os casos, ocorre a descompressão de páginas comprimidas que armazenam dados de blocos adjacentes ao que está sendo lido de um dispositivo (ou descomprimido do cache comprimido, se for o caso). Por conseqüência, essas páginas seriam consideradas mais recentemente usadas que todas as páginas do cache comprimido e de um número de páginas da memória não-comprimida. A razão para essa consideração é que as páginas realmente são liberadas da memória somente quando são liberadas do cache comprimido. Dessa forma, além de serem consideradas mais recentemente acessadas, elas também estariam mais distantes da liberação. Como conseqüência, é possível que essa mudança force a liberação de páginas que não estão em conformidade com o algoritmo de substituição de páginas.

No segundo caso, é importante notar que, além da mudança na ordenação das páginas, ocorre uma quantidade maior de operações de I/O submetidas quando é feito o read-ahead em virtude de uma página lida do cache comprimido. Isso é devido ao fato que será feita a leitura de blocos adjacentes ao bloco cujos dados estão armazenados na página comprimida.

Pelas razões explicitadas acima, tentamos evitar essas situações na nossa implementação. Para o primeiro caso, páginas comprimidas contendo dados que fizerem parte do conjunto a ser lido antecipadamente, por causa de uma leitura a um dispositivo de armazenamento, não são descomprimidas do cache comprimido, da mesma maneira que as páginas na memória não-comprimida não são alteradas quando fazem parte do read-ahead. Isso evita os efeitos citados acima, além de não possuir desvantagem em descomprimir essas páginas em outros momentos, se necessário.

No segundo caso, quando uma página é lida do cache comprimido, uma operação de leitura antecipada não deve ser efetuada, pois não existe vantagem em descomprimir antecipadamente páginas comprimidas. Além disso, não são feitas desnecessárias leituras de dispositivos secundários de armazenamento para servir essas leituras antecipadas.

Ademais, páginas comprimidas lidas do swap (que são armazenadas na forma comprimida, como descrito na Seção 3.3) são somente descomprimidas quando explicitamente requisitadas para uso pelo sistema de memória virtual.

Ao contrário desses casos que acabamos de tratar de páginas lidas somente devido à operação de read-ahead, uma página comprimida que é descomprimida e lida para uso imediato preserva a ordenação de páginas LRU, uma vez que será mais recentemente usada que qualquer página no cache comprimido.

Nós também consideramos essencial preservar a ordem na qual as páginas foram comprimidas de modo a sermos capazes de verificar a eficácia do cache comprimido. Caso contrário, os resultados poderiam ser influenciados por esse fator extra.

3.5.3 Células com Páginas de Memória Contíguas

Nós dizemos que a *taxa de compressão* de uma página comprimida é a taxa do tamanho da página comprimida sobre o seu tamanho original (vezes 100%). Empregamos *alta compressibilidade*, *baixa compressibilidade*, ou *quasi incompressibilidade* se a taxa de compressão média é menor que 50%, entre 50% e 70%, ou acima de 70%, respectivamente.

Se não temos alta compressibilidade, células do cache comprimido compostas de apenas uma página de memória podem armazenar apenas uma página comprimida, em média. Para minimizar esse problema de baixas taxas de compressibilidade, nós propomos a adoção de células compostas de páginas de memória contíguas. Com células maiores, é mais provável termos ganhos de espaço de memória mesmo que a maior parte das páginas não comprima muito bem. Por exemplo, mesmo que as páginas comprimam para 65% em média, ainda teremos ganhos de espaço ao utilizarmos células compostas de ao menos duas páginas de memória contíguas. De fato, nesse caso, é possível armazenar três páginas comprimidas em uma célula.

Entretanto, nós devemos observar que a decisão de quantas páginas de memória contíguas devem ser alocadas por célula deve levar em conta os seguintes aspectos:

- Quanto maior o número de páginas contíguas, maior é a probabilidade de falha ao alocá-las.

  As páginas só podem ser alocadas contiguamente no kernel em potências de dois. Logo, podemos alocar uma (2^0), duas (2^1), quatro (2^2), e assim por diante. Quanto maior o número de páginas contíguas alocadas no kernel, é mais difícil conseguir alocar esse número. Isso deve-se à fragmentação que ocorre na memória do sistema onde, depois de diversas alocações e liberações de memória, fica cada vez mais difícil encontrar memória contígua livre no sistema, em particular para maiores quantidade de páginas.

  Em conseqüência dessa menor probabilidade de se encontrar maiores quantidades de memória disponível, ao usarmos células com grandes quantidades de páginas, será difícil alocá-las, principalmente para o redimensionamento adaptativo do cache. É preciso encontrar uma quantidade que não se encontre dificuldades ao alocá-la, e ao mesmo tempo que seja suficiente para minimizar o problema da baixa compressibilidade de determinadas páginas da memória.

- Quanto maior a célula, maior é a probabilidade de fragmentação nela e conseqüentemente o maior custo de compactação das suas páginas comprimidas.

  Em geral, células de tamanho maior armazenam mais páginas do que células de tamanho menor, o que gera um maior tráfego de inserções e remoções. Por esse motivo, e pelo fato de que não há movimentação das demais páginas comprimidas na célula voltada a diminuir a fragmentação quando uma nova página é inserida ou removida, a fragmentação dentro da célula tende a ser maior. Nesse caso, o custo de compactação dessas páginas comprimidas, que é efetuada quando não é encontrado espaço livre final em outra célula, pode se tornar

proibitivo. O número de páginas por célula também deve ser avaliado em qual será o custo de sua manutenção interna.

– Menos metadados para células maiores.

Como um bom efeito colateral, dado que parte dos nossos metadados é usada para armazenar dados sobre as células, o uso de células maiores reduz em igual proporção essas estruturas de dados.

Experimentalmente, nós concluímos que dois é o número de páginas contíguas a ser alocado para cada célula que atinge os melhores resultados na nossa implementação. Na Seção 6.4, nós vemos alguns testes onde essa decisão de projeto é fundamental.

3.5.4 Suspensão da Compressão de Páginas Limpas

Para alguns workloads, nenhum esquema de cache comprimido pode melhorar o desempenho do sistema. Isso ocorre quando nenhuma página ou muito poucas páginas são lidas do cache comprimido entre todas as páginas que foram comprimidas. Nesse caso, é imediato observar que uma grande quantidade de páginas foram comprimidas e liberadas, sem benefício real ao sistema. Comprimir essas páginas adicionou os custos inerentes como compressão, descompressão, gerenciamento e metadados.

Teoricamente, esse cenário pode acontecer com páginas em qualquer estado: limpas e sujas. No sistema, a distribuição de páginas limpas e sujas acontece de acordo com o dispositivo de armazenamento em que elas podem ser armazenadas. Páginas sujas são normalmente armazenáveis no swap, enquanto as páginas que são armazenadas em outros dispositivos secundários de armazenamento são usualmente limpas, o que se deve às próprias características de implementação do Linux. Primeiramente, quanto às páginas do swap, o Linux as marca como sujas quando elas são mapeadas por um processo que tenha permissão de escrita, o que é o caso comum. No tocante às páginas que são armazenadas em outros dispositivos de armazenamento, elas em geral fazem uso de buffers, logo elas não são marcadas como sujas em si (e sim os buffers).

Apesar de teoricamente ocorrer com páginas em qualquer estado de sujeira, nos nossos experimentos pudemos observar esse cenário – em que nenhum esquema de cache comprimido pode melhorar o desempenho do sistema – apenas com páginas limpas. Nós não encontramos um aplicativo realista cujas páginas sujas tivessem esse problema. Ademais, para páginas limpas esse problema é claramente mais evidente uma vez que essas páginas são liberadas do cache comprimido sem necessitar que nenhuma operação de I/O seja executada. Assim, os custos de compressão são destacados.

Na nossa implementação, adotamos uma heurística para tentar detectar quando páginas limpas estão sendo comprimidas sem benefício para o sistema. Essa heurística tenta detectar os cenários onde uma grande quantidade de páginas limpas são comprimidas, não requisitadas de volta enquanto estão no cache comprimido, e liberadas, sem ter provido benefício para o sistema através da compressão e dos seus custos inerentes.

A nossa heurística se baseia na verificação da relação entre quantas páginas comprimidas limpas são requisitadas por qualquer operação do kernel e quantas páginas comprimidas limpas são

liberadas do cache comprimido sem serem requisitadas. Assim que o cache comprimido detectar que muito mais[4] páginas foram liberadas do que requisitadas, ele desabilita a compressão de novas páginas despejadas da memória não-comprimida. A partir desse momento, páginas limpas despejadas da memória não-comprimida são liberadas sem serem primeiramente armazenadas no cache comprimido. Assim que nós suspendemos a compressão de páginas limpas liberadas da memória não-comprimida, o cache comprimido registra quais foram as últimas páginas limpas liberadas sem terem sido armazenadas no cache comprimido. Dados suficientes que identificam essas últimas páginas liberadas são armazenados e todas as páginas lidas dos dispositivos de armazenamento são verificadas para ver se fazem parte dessa lista. Quando o sistema detecta que muitas[5] páginas lidas do disco foram recentemente despejadas da memória sem serem comprimidas, nós reabilitamos a compressão de páginas limpas.

Um efeito importante da suspensão da compressão de páginas limpas é o impacto na ordenação LRU das páginas. Se nós liberamos páginas limpas da memória não-comprimida sem comprimi-las no cache comprimido, a ordenação de páginas LRU é alterada. Isso decorre do fato de que algumas das páginas liberadas pelo sistema de memória virtual serão armazenadas no cache comprimido e outras não, dessa forma algumas têm um caminho de liberação mais longo do que outras. Apesar disso, uma vez que poucas das páginas limpas eram requisitadas pelo sistema enquanto estivessem no cache comprimido, a maior parte delas seria liberada do cache comprimido de qualquer forma. Por isso, é esperado que liberá-las anteriormente não tenha grande impacto no desempenho do sistema. O overhead de metadados e processamento introduzidos por essa heurística são insignificantes.

Os parâmetros usados nas detecções de quando suspendemos a compressão de páginas limpas e de quando reabilitamos essas compressões foram decididos experimentalmente. Na Seção 6.5 nós veremos alguns testes onde essa decisão de projeto é muito importante para minimizar substancialmente o overhead.

3.5.5 Compressão do Swap

Outros trabalhos anteriores de cache comprimido [18, 8] implementaram, juntamente com a compressão de páginas na memória, o armazenamento comprimido dos dados no swap, aumentando assim o tamanho efetivo do swap. Esse armazenamento dos dados no formato comprimido no swap, além de aumentar o seu tamanho efetivo, também pode reduzir o número de operações de I/O pois podemos, principalmente, escrever diversas páginas comprimidas em apenas um bloco, reduzindo as operações de escrita.

Em nosso cache comprimido, também implementamos a compressão de swap como uma opção de configuração. No Linux (mesmo quando há o cache comprimido sem compressão de swap),

[4]Por muito mais, queremos dizer que devemos ter de 5 a 40 liberações a mais que requisições. Essa faixa varia de acordo com a porcentagem de páginas comprimidas limpas em relação ao total. Se todas as páginas comprimidas forem limpas, esse valor é 40, ao passo que se tivermos nenhuma ou poucas páginas comprimidas, esse valor é 5.

[5]Nesse caso, muitas páginas significa termos 10 vezes mais hits que o tamanho da tabela de hash que armazena as páginas comprimidas limpas liberadas recentemente. Essa tabela tem o tamanho de 1/7 do tamanho máximo do cache comprimido (em função do número de páginas de memória).

os endereços de swap são atribuídos às páginas bem antes de elas serem efetivamente escritas no swap. Quando há a compressão do swap, os endereços reais em que as páginas comprimidas serão escritas só podem ser atribuídos quando a página realmente vai ser escrita. Para isso, há um nível de indireção do endereçamento de swap. Os endereços atribuídos pelo Linux no desmapeamento das páginas são considerados virtuais, e quando as páginas serão escritas é que essa indireção é completada com o endereço real. E enquanto a página com um dado armazenável no swap está na memória, ela é endereçada pelo endereço virtual. Dessa maneira, somente quando há um acesso ao dispositivo secundário de armazenamento é que o endereço de swap virtual precisa ser traduzido para o endereço real antes de ser feito o acesso. Deve ser notado que esse nível de indireção varia em função do tamanho do swap e do tamanho máximo de aumento do tamanho do swap que é permitido.

Efetuamos alguns experimentos preliminares com esse recurso ativado de modo a verificar o seu efeito no desempenho. Ao contrário do esperado, em alguns aplicativos o desempenho foi diminuído em relação ao cache comprimido sem esse recurso, enquanto em outros o seu desempenho foi aumentado. Primeiro, deve ser notado que as operações de escrita em geral exercem um impacto menor no desempenho, logo as vantagens da redução dela em geral são menores que as da leitura. Segundo, o impacto do custo de memória dos metadados de indireção do endereçamento de swap sobre aplicativos que estejam sobre grande pressão de memória pode ser maior que o benefício trazido pelo menor número de escritas.

Por não ter sido claro o benefício da compressão de swap, notadamente em relação a situações em que há uma grande pressão de memória, a nossa decisão de projeto em relação a esse recurso foi a de não o utilizar por padrão. Mas ele permanece como uma opção de configuração ao usuário.

3.5.6 Cache Comprimido de Tamanho Variável

Em nossos experimentos, analisamos muitos casos de caches comprimidos estáticos, chegando a conclusões sobre eles e sobre o significado de um cache comprimido adaptativo. Uma vez que essa é uma questão chave no nosso trabalho, discutiremos, no próximo capítulo, as decisões de projeto relacionadas a ela.

3.6 Algoritmos de Compressão

Na nossa implementação, provemos suporte a três algoritmos de compressão para a avaliação do cache comprimido. Um deles é o LZO, que é implementação do Lempel-Ziv, enquanto os demais são algoritmos da família WK criada por Scott Kaplan e Paul Wilson. Esses algoritmos são brevemente descritos nessa seção.

3.6.1 LZO

LZO significa Lempel-Ziv-Oberhumer e é um algoritmo e implementação [40] do Lempel-Ziv [88, 89] por Markus Oberhumer.

A sua implementação é voltada à velocidade, e a velocidade de descompressão foi favorecida em relação a velocidade de compressão. Essa diferença de velocidades nem sempre é interessante para o uso do algoritmo para a compressão de dados de memória, uma vez que o número de descompressões em relação ao de compressões é muito maior que o caso comum do uso de um compressor.

Para a compressão dos dados, o LZO necessita de 64 Kb de memória para a compressão (para o seu dicionário, como é comum com compressores Lempel-Ziv). O seu código-fonte é distribuído sob a licença GPL [22], o que permite a sua utilização em projetos de software livre. Na nossa implementação, nós utilizamos o miniLZO, que é uma versão mais leve da biblioteca LZO.

Em relação aos algoritmos WKdm e WK4x4 a serem vistos a seguir, o LZO alcança melhores taxas de compressão, especialmente quando comprimimos informações outras que não dos segmentos de dados, com o custo maior para os tempos de compressão e descompressão.

3.6.2 WKdm

O WKdm [26, 80] é um algoritmo criado por Scott Kaplan e Paul Wilson para dados dos processos de memória, ou mais precisamente, o conteúdo do segmento de dados dos aplicativos. Não é de se esperar que o algoritmo comprima bem outros tipos de conteúdo. Na sua tese de doutorado, Kaplan estuda diversos algoritmos de compressão e desenvolve algoritmos voltados aos dados de memória, que possuem desempenho competitivo e taxas de compressão melhores que outros algoritmos para muitos dos programas experimentados por ele.

O WKdm requer um pequeno dicionário (16 bytes) em comparação com algoritmos de compressão comuns do estilo LZ que precisam de dicionários muito maiores (64 Kb). Ele mantém um dicionário das 16 palavras recentemente encontradas, gerenciando esse dicionário como um cache mapeado direto. Em relação ao WK4x4 (veja abaixo), o WKdm comprime com uma taxa de compressão próxima ao dele, sendo mais rápido devido à sua estrutura de dicionário.

A velocidade de compressão e de descompressão do WKdm são próximas, logo o WKdm comprime uma página não muito mais devagar do que a descomprime. Esse tipo de algoritmo de compressão é conhecido como simétrico.

A compressão no WKdm ocorre por blocos de dados do tamanho de uma palavra de memória de 32 bits, por vez. Essa palavra é lida, verificada se casa com uma entrada do dicionário e um código de dois bits é gravado na saída. Se for uma palavra vazia, o que é verificado antes, grava-se o padrão de quatro bits. A descompressão ocorre lendo a entrada comprimida e verificando os códigos de dois bits, e tomando a ação apropriada de acordo com o código.

Outros algoritmos de compressão como o X-Match [31] ou o algoritmo de compressão proposto por Rizzo [59] também são voltados para a compressão de dados de memória.

3.6.3 WK4x4

O WK4x4 é outro algoritmo da família de algoritmos criado por Scott Kaplan e Paul Wilson. Esse algoritmo compartilha todas as características citadas do WKdm acima, exceto a maneira como o dicionário é gerenciado. No WK4x4, o dicionário é gerenciado como um cache associativo por uma matriz 4x4, utilizando a política LRU para substituição para cada conjunto.

Esse algoritmo atinge a maior taxa de compressão entre os algoritmos da família WK, mas não é tão rápido quanto o WKdm (veja acima).

Capítulo 4

Adaptabilidade

Nesse capítulo apresentamos a nossa heurística de adaptabilidade para o redimensionamento do cache comprimido. Inicialmente temos uma análise dos caches comprimidos estáticos na Seção 4.1, possuindo a seguir uma apresentação sobre como é feito o redimensionamento do cache na Seção 4.2. O detalhamento da nossa heurística de adaptabilidade, precedido por uma visão geral dela, é apresentado na Seção 4.3.

4.1 Tamanho Estático

Observamos em nossos experimentos que, dado um aplicativo específico, tamanhos diferentes para caches comprimidos estáticos atingem relações de custo/benefício diferentes. Vale observar que mesmo com a melhor relação de custo/benefício, o benefício pode ser menor que que o custo introduzido pelo uso do cache comprimido. Assim, a melhor configuração entre os diversos tamanhos de cache comprimido não implica necessariamente que haja um ganho real de desempenho em comparação com um sistema sem cache comprimido.

Caches comprimidos estáticos menores do que o que tem a melhor relação de custo/benefício, em geral, provêm ganhos menores[1] que um do tamanho ótimo, ao reduzir acessos aos dispositivos de armazenamento. Ademais, eles introduzem o overhead inerente ao cache comprimido (veja mais detalhes na Seção 3.4), e o sistema tem um benefício menor do uso do cache comprimido. Por outro lado, caches comprimidos estáticos maiores que aquele com a melhor relação de custo/benefício provêm mais ganhos reduzindo acessos aos dispositivos de armazenamento . Mas também introduzem mais overhead ao sistema pela redução da memória não-comprimida (veja Seção 3.4). Esse overhead o impede de melhorar o desempenho proporcionalmente aos ganhos que ele é capaz de prover reduzindo acessos aos dispositivos de armazenamento. Veja a Figura 4.1 para uma ilustração desse comportamento. O caso adaptativo da figura será explicado na Seção 4.3.

Também é importante observar que o custo do cache comprimido é destacado quando a memória alocada para ele não é usada completamente. Isso quer dizer que o cache comprimido possui

[1] Apesar de ser possível ter um benefício similar em casos que a memória alocada não é totalmente usada.

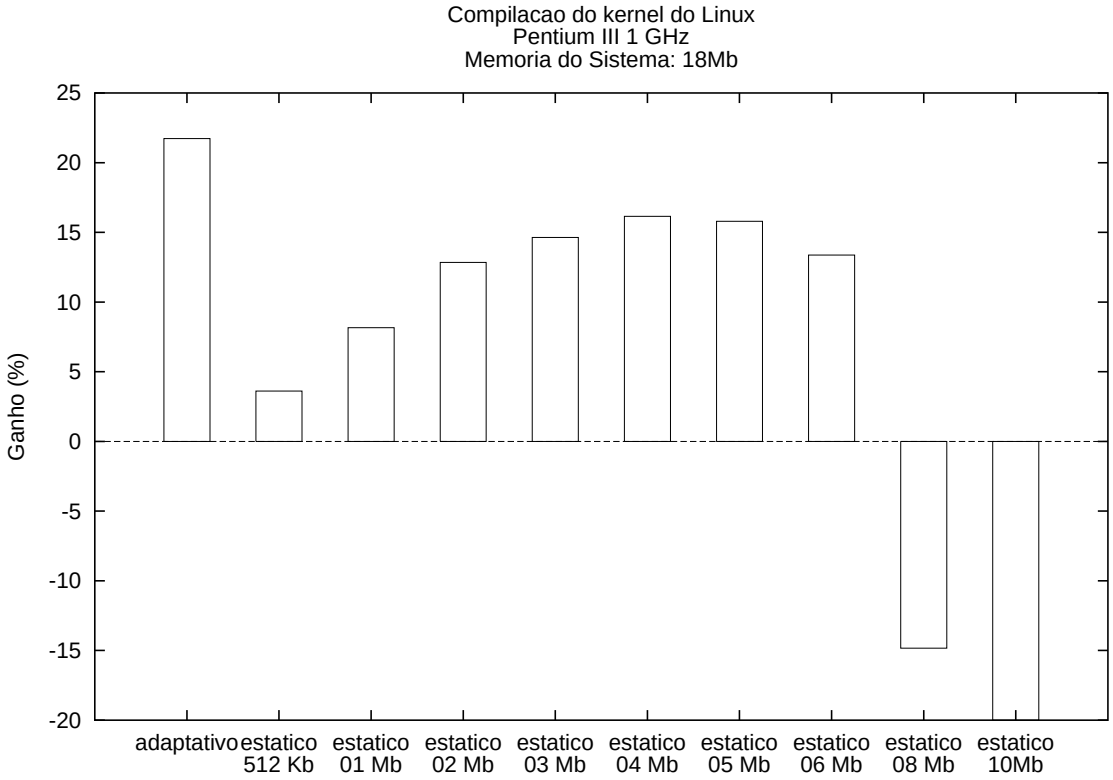

Figura 4.1: Comparação de diversos caches comprimidos com um kernel sem cache comprimido, exibindo ganhos relativos do tempo total de compilação do kernel do Linux (`j1`). Esses dados relativos foram obtidos para o cache comprimido adaptativo e para caches comprimidos estáticos com tamanhos variando de 512Kb a 10Mb.

memória alocada para o armazenamento de páginas comprimidas, mas esse espaço não é efetivamente utilizado pelo cache comprimido. Dessa maneira, ele introduz overhead ao sistema reduzindo a quantidade de memória disponível para a memória não-comprimida, mas provê parcialmente os benefícios que poderiam ser alcançados com um cache comprimido desse tamanho. Essa situação é muito comum com caches comprimidos estáticos por não se adaptarem à utilização de memória. Em geral essa situação é decorrência de o sistema utilizar uma quantidade maior de memória que a disponível na memória não-comprimida, mas incapaz, com a compressão, de utilizar toda a memória alocada para o cache comprimido. Veja a Figura 4.2 para uma ilustração da baixa utilização da memória alocada para o cache comprimido.

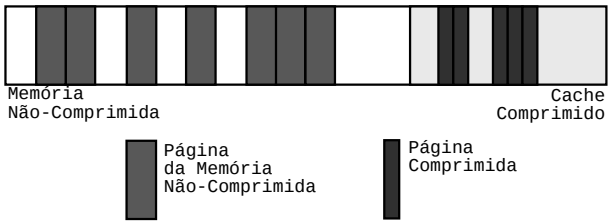

Figura 4.2: Cache comprimido com pouca utilização do espaço alocado.

Apesar das limitações mencionadas acima, caches comprimidos estáticos ainda podem melhorar o desempenho dos aplicativos. Entretanto, mesmo que um cache comprimido estático com um certo tamanho seja capaz de melhorar o desempenho para um determinado aplicativo, dado que os programas têm necessidades de memória diferentes durante a sua execução, isso não significa que esse cache comprimido estático pode prover melhora de desempenho para outros aplicativos. Ademais, um cache comprimido estático com um tamanho específico que melhore o desempenho de um aplicativo pode até degradar severamente o desempenho de outros. Essas conclusões estão de acordo com as análises já relatadas e/ou previstas por trabalhos anteriores [18, 62, 26].

Uma detecção confiável, no tempo de execução, de quanto memória deve ser mantida comprimida no sistema é o maior desafio para o cache comprimido tornar-se uma solução de propósito geral para a redução de acessos aos dispositivos secundários de armazenamento.

Juntamente com a detecção, a manutenção de um baixo overhead é vital para a viabilidade do redimensionamento. Ao se observar um alto overhead nesse processo, o seu benefício pode não compensar o custo que ele incorre.

4.2 Redimensionamento do cache

A base da nossa abordagem de adaptabilidade consiste em um cache comprimido de tamanho justo, sem alocação de memória supérflua. Isso melhora a eficiência no uso da memória, uma vez que memória alocada e não utilizada não melhora o desempenho, possivelmente até o piora, como dito anteriormente.

A quantidade de memória alocada para o cache comprimido será aumentada se, no momento em que uma nova página comprimida estiver por ser inserida no cache comprimido, ele satisfizer as

seguintes condições:

1. Não há espaço livre em nenhuma célula que seja grande o suficiente para armazenar a página comprimida;

2. Não há espaço livre (total, possivelmente fragmentado) em nenhuma célula que seja grande o suficiente para armazenar a página comprimida;

3. A política de adaptabilidade correntemente permite o crescimento do cache comprimido.

   Detalhes das situações em que a política de adaptabilidade barra o crescimento serão vistos à frente.

4. A alocação de páginas contíguas de memórias em número suficiente para a formação de uma nova célula não falha.

Por outro lado, a quantidade de memória alocada para o cache comprimido também pode ser diminuída de acordo com determinadas circunstâncias. A seguir, apresentamos os dois motivos pelos quais uma célula usada pelo cache comprimido pode ser liberada para o sistema:

1. A última de suas páginas comprimidas foi liberada.

   No momento em que uma célula contiver apenas uma página comprimida, e essa página for liberada, as páginas de memória alocadas que a compõem são liberadas para uso do sistema. Essa diretriz é condizente com a base da nossa política de adaptabilidade para a manutenção de um cache comprimido de tamanho justo.

2. A política de adaptabilidade tenta compactar o cache comprimido.

   Essa compactação, diferente da compactação de páginas comprimidas dentro de uma célula, acontece quando a política de adaptabilidade decide diminuir o tamanho do cache comprimido. A operação de compactação do cache comprimido consiste em selecionar uma célula e movimentar todas as páginas comprimidas dessa célula para novas células, permitindo que essa célula seja liberada para o sistema.

   Detalhes de quando há uma tentativa de compactação do cache comprimido são encontrados mais à frente nesse capítulo.

É importante notar que, uma vez que liberamos uma célula para o sistema, não é certo que consigamos alocá-la novamente quando necessário, especialmente células compostas de mais de uma página. De fato, alocações e liberações de páginas têm o custo inerente das funções que executam essas tarefas, mas elas são insignificantes. Isso é verdadeiro porque nós nos certificamos que alocações e liberações de páginas desse método são executadas sem que o sistema de memória virtual seja ativado para tentar forçar as liberações, evitando assim operações de alto custo. Em outras palavras, somente alocamos novas páginas se as páginas de memória de que nós necessitamos estão disponíveis no instante da alocação.

4.3 Heurística de Adaptabilidade

Nós projetamos e implementamos uma política de adaptabilidade para o cache comprimido que tenta detectar quando ele deve mudar o seu uso de memória de modo a prover mais benefícios e/ou diminuir os seus custos.

Nessa seção descrevemos a nossa heurística para implementar o cache comprimido adaptativo. Primeiramente, efetuamos uma descrição geral do seu funcionamento, seguido por um detalhamento maior de como a heurística é implementada.

4.3.1 Descrição Geral

Em termos de como o cache comprimido adapta o seu tamanho ao comportamento do sistema, nós partimos da hipótese que o cache comprimido é útil para o sistema. A nossa análise online toma atitudes em relação ao redimensionamento do cache comprimido depois que foi detectada alguma tendência do sistema. Essa análise é baseada nas requisições de páginas comprimidas pelo sistema.

Na inicialização do sistema, o cache comprimido começa com um tamanho mínimo e está livre para crescer, ou seja, aumentar a memória alocada para o seu uso (de acordo com os requisitos explicitados na Seção 4.2) de modo a armazenar as páginas comprimidas. Esse crescimento acontece enquanto não há requisições de páginas comprimidas ou a análise das requisições indicar que o cache comprimido não forneceu benefícios suficientes para superar os seus custos, como será visto com mais detalhes na próxima seção.

Quando as requisições indicarem, através da idade da página comprimida, que o cache comprimido não provê benefícios suficientes para os custos introduzidos, o cache comprimido primeiramente sofre um processo de travamento do seu tamanho, ou seja, ele não pode mais alocar memória para crescer de tamanho mesmo que os demais requisitos para o crescimento do cache, citados acima, sejam verdadeiros. Caso a análise continue indicando benefícios insuficientes para os custos, o cache comprimido começa a sofrer diminuição do seu tamanho. Essa diminuição começa pela tentativa de realocação de páginas comprimidas de uma célula nas demais do cache comprimido. Caso seja infrutífera essa tentativa, páginas comprimidas são forçadamente liberadas do cache comprimido.

4.3.2 Detalhamento

As páginas no cache comprimido são divididas em duas listas: *custo* e *lucro* (veja Figura 4.3). A lista custo armazena as páginas comprimidas que estariam na memória se o cache comprimido não fosse utilizado no sistema. A lista lucro armazena as páginas comprimidas que estão ainda na memória somente devido à utilização da compressão no cache comprimido. Essa divisão nas listas é baseada nos seguintes critérios:

– Ordenação das páginas comprimidas.

  Para a decisão de quais páginas são armazenadas na lista custo ou na lista lucro, utilizamos a ordem na qual elas foram inseridas no cache comprimido. As páginas que foram mais

recentemente inseridas no cache comprimido serão armazenadas na lista custo, ao passo que as que foram inseridas há mais tempo estarão na lista lucro.

– Taxa efetiva de aumento da memória com a utilização do cache comprimido.

  As duas listas, custo e lucro, contêm as páginas mais recentemente e mais antigamente armazenadas no cache comprimido respectivamente. No entanto, a ordenação das páginas comprimidas não é suficiente para a definição da fronteira entre essas duas listas.

  Essa fronteira é definida pela taxa efetiva de aumento da memória com a utilização do cache comprimido, que é expressa, em um dado momento, pela razão entre o número de páginas comprimidas armazenadas no cache comprimido naquele momento pelo número de páginas de memória alocadas para as células.

  Se a taxa efetiva de compressão for um, por exemplo, estaremos armazenando uma página comprimida por página de memória alocada. Dessa forma, todas as páginas comprimidas estarão na lista custo, uma vez que essas páginas estariam presentes na memória mesmo que não houvesse cache comprimido. Neste caso, não haveria "lucro" (aumento efetivo do tamanho da memória) devido ao uso do cache comprimido.

  Caso possuamos uma taxa efetiva de compressão equivalente a dois, por exemplo essa taxa significará que estaremos armazenando em média duas páginas comprimidas por página de memória alocada. Assim, a lista custo armazenará a metade das páginas comprimidas que foram mais recentemente inserida no cache comprimido e a lista lucro a outra metade. Essa divisão exibe de maneira clara a função da lista lucro, que armazena as páginas que nesse caso só estão ainda presentes na memória devido à compressão.

Como apenas páginas da lista lucro são liberadas pelo cache comprimido (por conter as páginas mais antigamente inseridas), a atualização das listas de páginas comprimidas é feita movimentando páginas da lista custo para a lista lucro. A verificação sobre a correta divisão das listas em determinado momento e a possível movimentação de páginas comprimidas de uma lista para outra ocorre quando páginas comprimidas são liberadas do cache comprimido, quando uma nova página comprimida é inserida no cache comprimido e quando há a tentativa de compactação do cache comprimido (realocação de páginas comprimidas em outras células de modo a liberar uma célula).

Conforme dito acima, a tentativa de detecção de quando o cache comprimido deve mudar o seu tamanho acontece quando páginas comprimidas são requisitadas pelo sistema ao cache comprimido. A análise baseia-se na informação de qual lista a página comprimida pertencia quando foi requisitada e no histórico recente contendo as listas às quais as páginas comprimidas requisitadas pertenciam.

A nossa política considera páginas comprimidas que são requisitadas da lista custo como sendo um sinal que o cache comprimido não está sendo vantajoso para o sistema. Essa atitude deve-se ao fato de estarmos acessando uma página que, sem o cache comprimido, seria acessada sem o overhead imposto por ter sido comprimida e armazenada no cache comprimido. Com o cache comprimido, possuímos um overhead inerente ao acesso a estas páginas.

Páginas comprimidas da lista lucro que são requisitadas pelo sistema indicam, segundo a nossa abordagem, que o cache comprimido está sendo benéfico, pois essas páginas continuam sendo armazenadas na memória devido à compressão. O mais importante é que, pelo fato de serem acessadas,

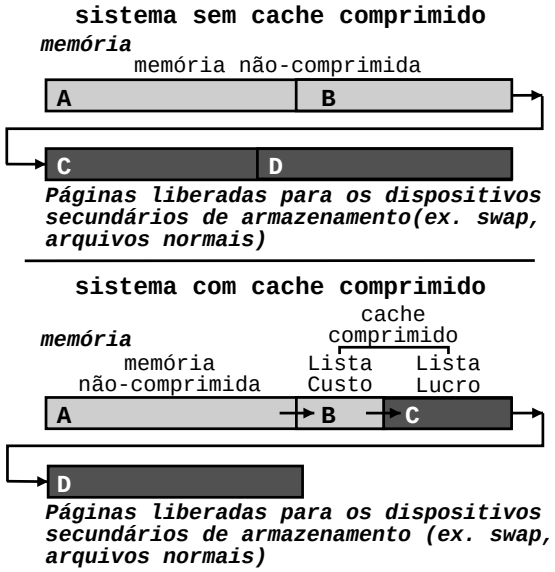

Figura 4.3: Listas no cache comprimido.

a compressão dessas páginas para armazená-las no cache comprimido está resultando em uma contribuição real ao sistema. O fato de apenas comprimir e armazenar mais páginas não contribui de maneira efetiva ao sistema caso essas páginas não sejam acessadas e conseqüentemente os acessos ao disco para a leitura delas não sejam economizados.

Caso a página requisitada seja da lista custo, para tomar alguma atitude em relação ao redimensionamento do cache comprimido, a nossa política leva em conta o histórico das últimas requisições. Quando uma primeira requisição a uma página da lista custo ocorre, esse acesso é armazenado, mas ainda não é tomada nenhuma atitude em relação ao crescimento do cache comprimido. Na segunda requisição consecutiva de páginas comprimidas da lista custo, a atitude tomada é travar o crescimento do cache comprimido. Assim, o cache comprimido não poderá alocar mais células para o seu uso pelo fato de termos sinais que o cache comprimido não está sendo benéfico ao sistema. Se houver uma terceira requisição consecutiva, há uma tentativa de compactação do cache comprimido, que tenta realocar páginas comprimidas de uma célula nas demais. Na impossibilidade de que isso ocorra, uma página comprimida é liberada. Assim que uma dessas ações é tomada, a barreira de crescimento é removida e o histórico de requisições é zerado.

Quando uma página da lista lucro é requisitada, qualquer barreira de crescimento existente é removida e o histórico de requisições reiniciado. Um sumário da heurística pode ser encontrado na Figura 4.4.

Essa política de adaptabilidade foi ajustada a partir dos nossos experimentos, cujos resultados e análise serão apresentados com detalhes na Seção 6.4. Dos resultados obtidos através dos experimentos, nós verificamos que um kernel com cache comprimido utilizando a nossa política de adaptabilidade consegue bons resultados em relação a um kernel sem o uso de cache comprimido.

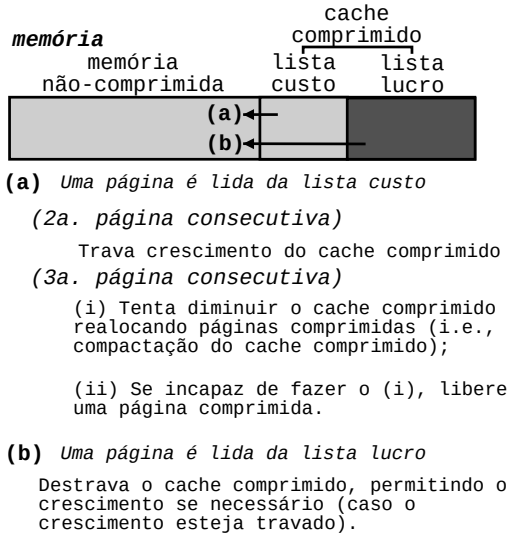

Figura 4.4: Sumário da heurística de adaptabilidade.

Com relação a kernels que utilizam caches comprimidos de tamanho estático, fizemos a comparação com os casos em que algum cache comprimido estático atinge uma melhora de desempenho em relação a um kernel sem o cache comprimido. Nesse caso, a nossa política de adaptabilidade normalmente acaba selecionando tamanhos durante a execução que atinge ganhos de desempenho muito próximos aos obtidos pelo melhor tamanho estático. Além desses resultados positivos, em alguns cenários, aplicativos podem se beneficiar da política de adaptabilidade (principalmente evitando memória alocada supérflua), atingindo resultados melhores do que qualquer cache comprimido estático (obviamente quando esse oferece melhora de desempenho).

Veja Figuras 4.1 e 4.5 para resultados de uma comparação entre um cache comprimido estático e adaptativo em relação a um kernel sem cache comprimido

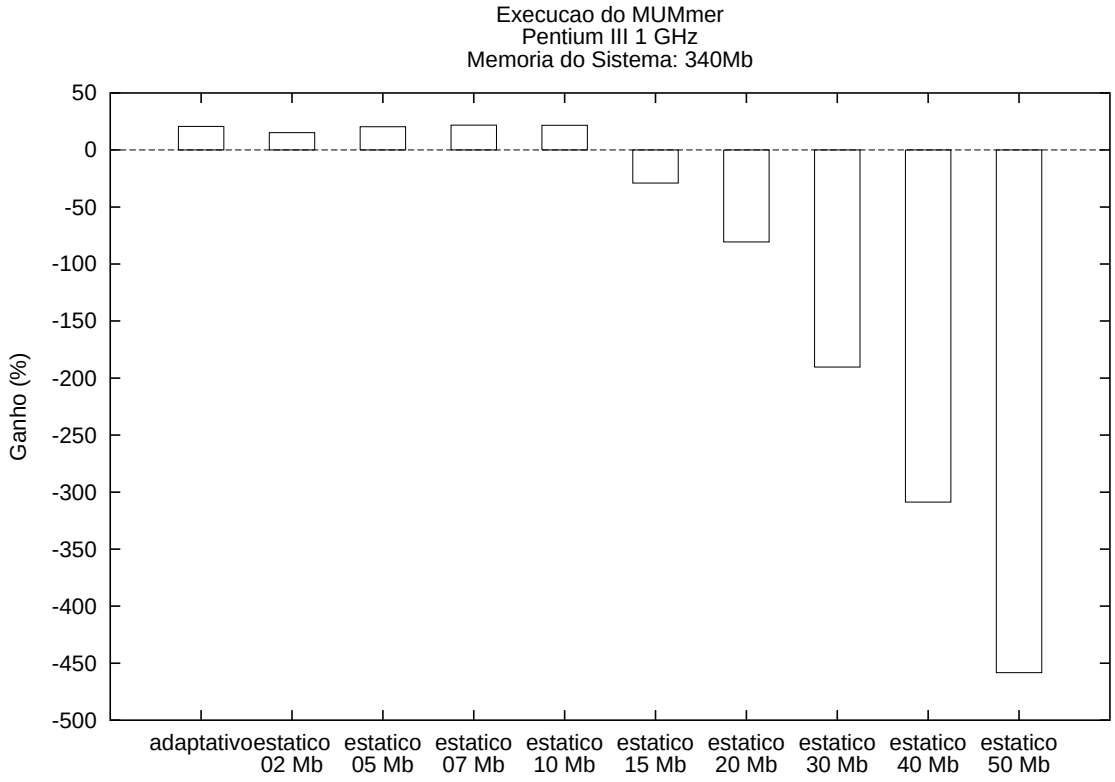

Figura 4.5: Comparação de diversos caches comprimidos com um kernel sem cache comprimido, exibindo ganhos relativos do tempo total de execução do MUMmer. Esses dados relativos foram obtidos para o cache comprimido adaptativo e para caches comprimidos estáticos com tamanhos variando de 2 a 50Mb.

Capítulo 5

Detalhes de Implementação

Detalhes específicos da implementação são o objetivo desse capítulo. Aqui apresentamos como lidamos com algumas limitações do sistema operacional onde implementamos o cache comprimido, assim como a maneira que algumas operações e estruturas foram implementadas objetivando o melhor desempenho possível na execução dessas tarefas. No final, fazemos uma apresentação da estrutura das alterações da implementação dentro da árvore do código-fonte do kernel do Linux.

5.1 Endereços Virtuais de Swap

No Linux, apenas páginas não mapeadas por qualquer tabela de página de processos e não referenciadas por qualquer parte do código do kernel (exceto o próprio código de liberação) é que podem ser liberadas. Essas páginas, antes de serem liberadas, precisam ter uma cópia atualizada em algum dispositivo de armazenamento, incluindo o swap, para o caso da página não estar vinculada com algum arquivo ou metadado de sistema de arquivos. No caso de páginas que são armazenáveis no swap, elas usualmente estão mapeadas para alguma tabela de página, então o desmapeamento dessa página da tabela é fundamental para a sua liberação (e eventual escrita no swap).

Em nossa implementação do cache comprimido, são armazenadas no cache comprimido apenas páginas sujas que são liberadas pelo sistema de liberação de memória[1] ou páginas limpas que ainda não foram comprimidas. Por essa razão, apenas páginas não mapeadas por tabelas de páginas nem referenciadas por alguma parte do código do kernel são livres para serem candidatas ao armazenamento no cache comprimido.

O primeiro passo para a liberação de páginas mapeadas em tabelas de páginas é o desmapeamento dessas páginas de suas respectivas tabelas. Isso é feito percorrendo os processos na memória e as suas entradas das tabelas de página. Entretanto, o processo de desmapeamento só se completa, para o caso de páginas não vinculadas a um sistema de arquivos ou dispositivo de bloco, atribuindo a elas e às entradas de tabelas de páginas as quais estavam mapeadas um novo endereço. Esse endereço não é um endereço real de memória, mas sim um endereço de swap. A partir dele é que

[1] Páginas sujas na verdade são comprimidas no momento em que elas são limpas pelo sistema de VM. Depois disso, elas são de fato liberadas mas já possuem uma cópia no cache comprimido.

essas páginas poderão ser recuperadas, enquanto estiverem na memória ou escritas no dispositivo de swap.

Enquanto o kernel possuir endereços de swap suficientes, as páginas serão desmapeadas das tabelas de páginas e remapeadas para esses endereços. E assim, dependendo da necessidade, o sistema de memória virtual as escreverá e/ou as liberará, e assim elas serão armazenadas no cache comprimido.

Em situações de intenso uso do swap os endereços de swap podem se esgotar. Esse cenário a ser descrito pode acontecer também se não existir swap no sistema. A partir desse momento, entradas da tabela de páginas que já foram desmapeadas uma vez e remapeadas para um endereço de swap (se houver swap) mantêm o seu endereço de swap e podem continuar a ser desmapeadas. No Linux, é impossível desmapear as páginas que não possuam algum endereço de swap. Dessa forma, essas páginas não poderão ser escritas nem liberadas pelo sistema de memória virtual, e por sua vez, não poderão ser comprimidas e armazenadas no cache comprimido.

O fato de não comprimir tem duas conseqüências problemáticas para o cache comprimido:

- Toda página que não esteja vinculada a um dispositivo secundário de armazenamento não pode ser comprimida e armazenada no cache comprimido.

  Esse é o caso em que não há swap disponível no sistema. Dessa forma, as páginas não vinculadas a um dispositivo secundário de armazenamento não são desmapeadas das tabelas de páginas e remapeadas para um endereço de swap. Como elas continuam mapeadas, não podem ser escritas e/ou liberadas, logo não são comprimidas e armazenadas no cache comprimido. Apesar disso, o cache comprimido, nesse caso, pode ainda armazenar páginas comprimidas vinculadas a um dispositivo de armazenamento, como arquivos, metadados de arquivos. O armazenamento dessas páginas é discutido na Seção 3.5.1.

- Caso haja esgotamento dos swaps disponíveis, o uso do cache comprimido diminui o tamanho efetivo da memória.

  Esse caso é intrincado e requer algum cuidado na sua explicação, por isso vamos ignorar a possível compressão de outras páginas além daquelas armazenáveis no swap. Também não será levada em conta diminuições do cache comprimido devido a qualquer política de adaptabilidade.

  Supondo que haja swap, os seus endereços são associados a certas páginas que são desmapeadas e remapeadas quando há necessidade de liberação de memória no sistema. Logo, a partir do momento que esses endereços são associados às páginas, um bloco no swap é reservado para o eventual caso em que essa página é escrita para o disco (veja Figura 5.1).

  Os endereços de swap continuam a ser atribuídos às páginas, e essas páginas, supondo que a pressão de memória continue, são comprimidas e armazenadas no cache comprimido. Isso continua a acontecer enquanto os endereços de memória não se esgotam. Dessa forma, chegamos a uma situação como exemplificada na Figura 5.2, em que algumas páginas poderão ter sido escritas em swap, e as demais estarão no cache comprimido (e eventualmente na memória não-comprimida, se não couberem todas no cache comprimido). Nesse caso, ao se esgotarem os endereços de swap, as páginas do cache comprimido não serão escritas no swap

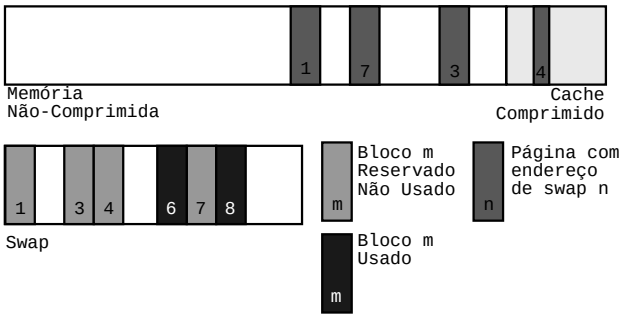

Figura 5.1: Situação da memória depois que endereços de swap foram atribuídos a páginas para serem desmapeadas. Quando isso acontece, as páginas podem ainda estar na memória (memória não-comprimida ou cache comprimido) mas os blocos no swap já estão reservados para futuro uso.

pelo fato de que não há mais páginas a serem comprimidas (por não haver endereços de swap a serem atribuídos), logo elas não são forçadas a saírem do cache comprimido. Dessa forma, essas páginas comprimidas utilizam o espaço de memória que estiverem ocupando, além de manter reservado o espaço no swap.

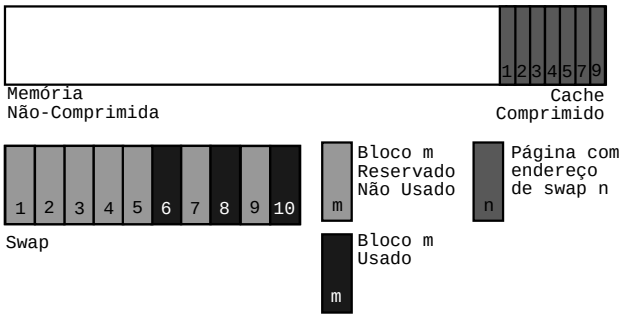

Figura 5.2: Situação da memória depois que endereços de swap foram atribuídos a páginas de modo a serem desmapeadas e se esgotaram. Quando isso acontece, as páginas continuam na memória (memória não-comprimida ou mais provavelmente no cache comprimido) e os blocos no swap estão reservados para futuro uso, utilizando assim os dois recursos.

Devido a essa limitação que é imposta pela necessidade de endereços de swap para o desmapeamento e escrita/liberação das páginas, a quantidade de memória máxima que pode ser alocada pelo sistema é menor quando essa situação ocorre em relação a um sistema sem cache comprimido. Abaixo, na Figura 5.3, verificamos a quantidade máxima que poderia ser alocada em um sistema sem cache comprimido, e também a quantidade de memória que pode ser alocada quando uma situação de esgotamento dos endereços de swap ocorre. No caso com cache comprimido, pelo fato de determinadas páginas continuarem na memória (e terem os seus blocos no swap reservados), a memória ocupada por essas páginas deixa de ser útil para o aumento efetivo da memória do sistema. Ademais, nesse caso, essa memória deixa de ser

ocupada por um dado que não teria espaço reservado no swap, o que na verdade diminui o espaço total de alocação.

Para lidar tanto com o problema de sistemas sem swap, em que o cache comprimido não é utilizado, como o do esgotamento do endereços de swap em sistemas com swap, nós implementamos uma solução conhecida como *endereços virtuais de swap*. No momento em que não há endereços de swap disponíveis, seja pelo esgotamento ou por não ter swap, páginas não vinculadas a nenhum dispositivo secundário de armazenamento, ao serem desmapeadas, recebem um endereço de swap virtual.

Utilizando esse endereçamento, é possível comprimir e armazenar no cache comprimido páginas não vinculadas a um dispositivo secundário de armazenamento mesmo que não haja swap (Figura 5.4. Por conseqüência, essas páginas ainda não são escritas em algum dispositivo secundário de armazenamento por se tratar de um endereço virtual de swap, mas temos a a possibilidade de um aumento efetivo do tamanho da memória pois utiliza-se compressão de memória.

Quanto ao segundo problema citado acima, os endereços virtuais de swap possibilitam que páginas armazenáveis no swap continuem sendo desmapeadas, forçando aquelas que possuam um bloco de swap reservado a saírem do cache comprimido, efetivamente aumentando o tamanho da memória. Os endereços virtuais continuarão a ser atribuídos às páginas desmapeadas enquanto houver espaço livre no cache comprimido (páginas no cache comprimido com endereços reais de swap são contadas como espaço livre pois podem ser escritas no swap) e enquanto não houver endereços reais de swap disponíveis. Veja Figura 5.5 para uma ilustração desse cenário.

Os endereços virtuais são formados de maneira análoga aos endereços reais utilizados pelo Linux. A diferença se encontra no dispositivo de swap que é utilizado, que no caso do endereçamento virtual é um dispositivo com numeração acima do limite de dispositivos de swap no sistema. Por se tratar de um endereço virtual, as funções de tratamento de endereços reais não se aplicam, e o suporte a esse endereçamento está presente na nossa implementação. Esse esquema introduz overhead de metadados proporcionais ao espaço extra de endereçamento de swap virtual.

5.2 Buscas

A buscas são uma parte fundamental da implementação do cache comprimido no quesito de desempenho, visto que elas são executadas um grande número de vezes. Um mau projeto das estruturas de dados das buscas e/ou das suas funções pode ser um impedimento para um bom desempenho do cache comprimido.

No cache comprimido, as buscas mais importantes são as seguintes:

– Verificação e localização de páginas comprimidas para servir requisições do kernel ou de programas de usuário.

  Em momentos de recuperação dos dados por parte do kernel, seja para alguma operação própria ou para um programa do espaço do usuário, é necessário verificar se uma página se encontra comprimida antes de submeter uma operação de leitura ao dispositivo secundário de armazenamento. Para essa verificação, é utilizada o par `mapping/index`, como explicado

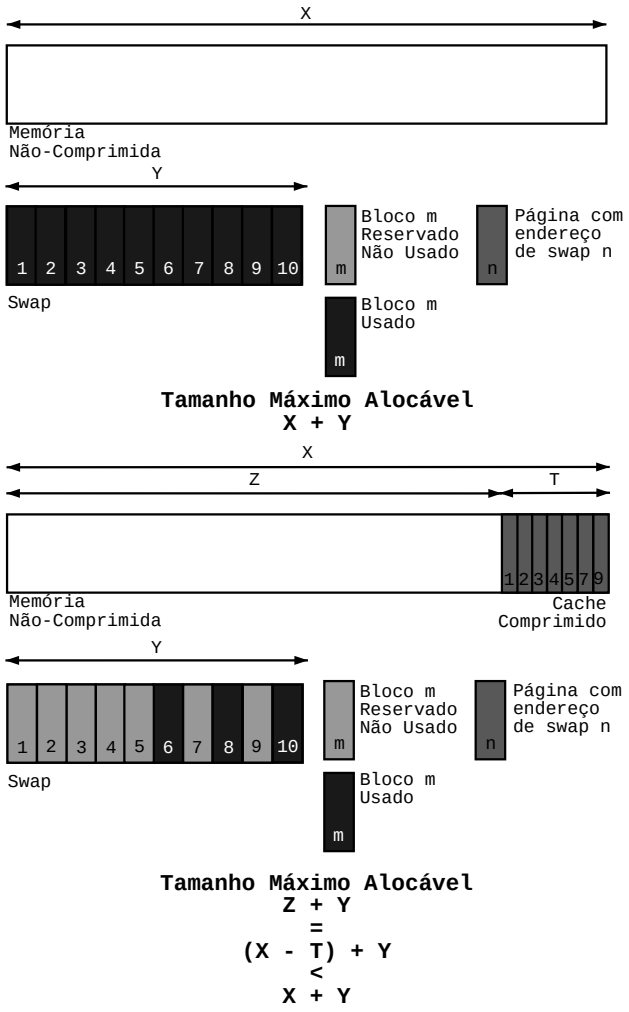

Figura 5.3: Quantidade máxima de memória que pode ser alocada em um sistema sem cache comprimido (figura de cima) e com o cache comprimido (figura de baixo). Com o cache comprimido sem endereços virtuais de swap, pelo fato de que suas páginas reservam blocos de swap ao adquirir um endereço de swap, o tamanho máximo alocável é menor, pois as páginas com endereços de swap continuam na memória ocupando espaço.

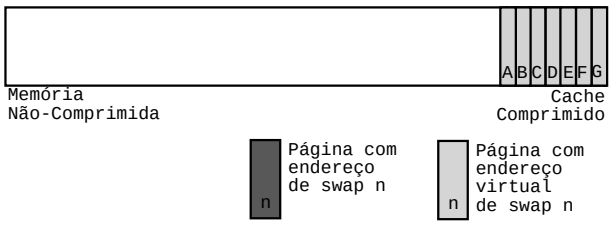

Figura 5.4: Sistema sem swap com a utilização de endereços virtuais de swap para poder comprimir armazenar páginas no cache comprimido.

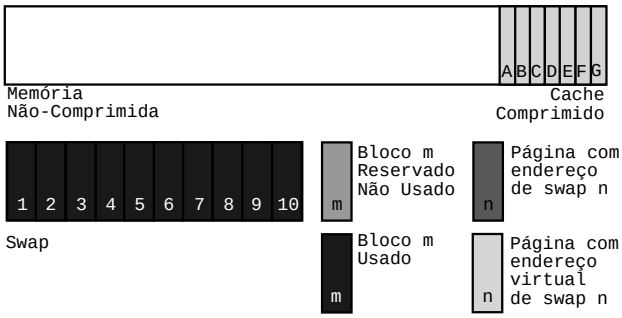

Figura 5.5: Sistema com swap utilizando o endereçamento virtual para poder efetivamente utilizar o cache comprimido e aumentar a memória do sistema.

à frente, para identificar a página. Essa procura é necessária para outras atividades como liberação de uma página comprimida.

– Procura de células com espaço suficiente para a inserção de uma nova página comprimida.

  Quando uma página é comprimida, é necessária a localização de uma célula que possa armazená-la prontamente, mesmo que isso implique em uma compactação das páginas que nela estão armazenadas. Para tal localização, precisamos manter registro do espaço livre, assim como do espaço livre final das células.

As duas buscas citadas acima contam com tabelas de hash como estruturas de dados, pelos baixos custos de inserção e remoção de entradas. A primeira busca utiliza uma tabela de hash aos moldes da que é utilizada pelo cache de páginas para a localização de páginas, fazendo uso da mesma função de hash. O tamanho dessa tabela é em função do número de células que existem no cache comprimido e caso o cache comprimido seja redimensionado, essa tabela de hash também o é.

No caso da segunda busca, são utilizadas duas tabelas de hash, que são similares. Nesse caso, apenas as chaves diferem, pois uma utiliza o espaço livre como chave, ao passo que outra utiliza o espaço livre final. O tamanho delas é fixo e é dependente do número de páginas alocadas por célula.

5.3 Ajuste das Marcas D'água

O sistema de gerenciamento de memória precisa ser capaz de alocar páginas de memória para serviços importantes do kernel mesmo sob pressão de memória. No Linux 2.4.18, marcas d'água existem para assegurar que exista memória suficiente para esse serviços emergenciais e também para controlar o gerenciamento da kernel thread responsável pela liberação de memória.

Conforme explicado na Seção 2.2.9 a respeito das marcas d'água, quando o número de páginas livres está abaixo de qualquer uma delas, o sistema de memória inicia o processo de liberação de páginas até que o número de páginas livres esteja acima de qualquer uma dessas marcas d'água.

As marcas d'água são calculadas inicialmente durante a inicialização do sistema e são baseadas no total de memória física. No caso de não haver adequação das marcas d'água ao tamanho da memória não-comprimida, tempo substancial do processamento pode ser despendido em funções do kernel para a manutenção de uma quantidade de memória livre na parte de memória não-comprimida maior do que seria necessário para o seu tamanho real em determinado momento. Essa é a razão pela qual, em um sistema com cache comprimido adaptativo, visto que o tamanho da memória não-comprimida muda ao longo da execução do sistema, as marcas d'água para controle das páginas livres precisam ser ajustadas dinamicamente de acordo com o tamanho da memória não-comprimida.

Em nossa implementação, a cada vez que o cache comprimido é aumentado ou diminuído, as marcas d'água são verificadas e, caso estejam incoerentes com o novo tamanho do cache comprimido, elas são redimensionadas. A única modificação no código original que foi necessária para permitir que esse ajuste nas marcas d'água fosse feito acabou sendo a declaração de três vetores de três valores inteiros cada, que deixaram de ser dados a serem descartados depois da inicialização do kernel do Linux

5.4 Estruturas de Dados

Aqui apresentamos em detalhes as principais estruturas de dados da implementação, que são as estruturas das células, das páginas comprimidas que são armazenadas nas células e das páginas temporárias utilizadas quando a nossa implementação executa escritas de páginas.

5.4.1 Células

O nosso cache comprimido é, como dito acima, formado por um número de células, que podem ser compostas de uma ou mais páginas de memória contíguas. Cada célula é descrita por uma estrutura de dados, cujo código-fonte é exibido na Figura 5.6. Essa estrutura armazena os seguintes dados:

- `page`: Endereço da primeira página de memória que faz parte da célula. Se houver apenas uma, é o endereço dessa página. Na atual implementação, todas as células têm um número de páginas contíguas igual, logo não há necessidade de armazenar esse tipo de informação na estrutura.

```
struct comp_cache_page {
        struct page * page;

        /* fields for compression structure */
        unsigned short free_offset;

        /* free space that can used right away */
        short free_space;

        /* total free space = free_space + fragmented space, ie the
         * sum of all freed fragments waiting to be merged */
        short total_free_space;

        struct list_head fragments;

        /* free space hash table */
        struct comp_cache_page * next_hash_fs;
        struct comp_cache_page ** pprev_hash_fs;

        /* total free space hash table */
        struct comp_cache_page * next_hash_tfs;
        struct comp_cache_page ** pprev_hash_tfs;
};
```

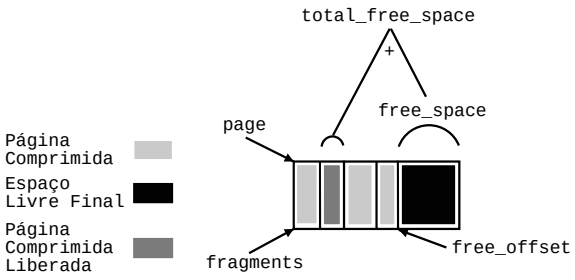

Figura 5.6: Declaração da estrutura de dados de uma célula no cache comprimido, em linguagem C (arquivo `linux/include/linux/comp_cache.h`). O espaço livre final é considerado como `free_space`, ao passo que a soma de todos os espaços livres é o `total_free_space`. Páginas comprimidas são conhecidas ao longo do código como fragmentos. Embaixo da declaração, alguns dados da estrutura ilustrados com uma figura de uma célula.

- `free_offset`: O começo do espaço livre final. Utilizado para saber onde armazenar uma nova página comprimida nessa célula.
- `free_space`: O espaço livre final, que pode ser utilizado imediatamente, sem a necessidade de se efetuar a compactação das páginas comprimidas.
- `total_free_space`: Aqui contabiliza-se o espaço livre total na página, somando o espaço livre com os eventuais espaços livres fragmentados.
- `fragments`: Uma lista circular duplamente ligada das páginas comprimidas que estão armazenadas nessa célula.
- `next_hash_fs/pprev_hash_fs`: Usados para a tabela de hash do espaço livre final, que é utilizada para procura de células com determinada quantidade de espaço livre final no momento da inserção de uma nova página comprimida. A chave para essa tabela de hash é o espaço livre final.
- `next_hash_tfs/pprev_hash_tfs`: Esses campos são usados para a tabela de hash do espaço livre total da célula. Essa tabela de hash é utilizada para procura de células com determinada quantidade de espaço livre total no momento da inserção de uma nova página comprimida. Nesse caso, a página precisa sofrer a compactação antes da inserção da nova página. O espaço livre (total) é a chave para essa tabela de hash.

5.4.2 Páginas Comprimidas

Além de uma estrutura descrevendo a célula, também é necessário armazenar dados sobre as páginas comprimidas que estão no cache comprimido. Na Figura 5.7, temos o código-fonte dessa estrutura, e abaixo descrevemos quais dados ela armazena:

- `list`: Utilizado para a lista das páginas comprimidas da célula da qual essa página comprimida faz parte.
- `lru_queue`: Utilizado para a lista de todas as páginas comprimidas do cache comprimido. Essa lista é usada para a ordenação da entrada das páginas no cache.
- `mapping_list`: Utilizado para a inclusão dessa estrutura na lista das páginas comprimidas de um determinado `address space`.
- `count`: Armazena o número de referências para essa página comprimida durante a execução do código. Se `0` (zero), essa estrutura de página comprimida está livre. Se maior que zero, está sendo usada para armazenar uma página comprimida e possivelmente está sendo acessada por alguma parte do código.
- `mapping`: Endereço da estrutura `address space` dos dados comprimidos. Essa estrutura define como um determinado tipo de páginas deve ser tratado pelo kernel, definindo operações, a qual inode ou dispositivo de bloco que as páginas estão vinculadas, como essas páginas

```
struct comp_cache_fragment {
        /* list of fragments in a comp page*/
        struct list_head list;

        /* list for lru queue ordering */
        struct list_head lru_queue;

        /* list of comp pages in the mapping */
        struct list_head mapping_list;

        /* usage count */
        atomic_t count;

        unsigned long index;
        struct address_space * mapping;

        struct swp_buffer * swp_buffer;

        /* offset in the compressed cache we are stored in */
        unsigned short offset;

        unsigned short compressed_size;
        unsigned long flags;

        struct comp_cache_page * comp_page;

        struct comp_cache_fragment * next_hash;
        struct comp_cache_fragment ** pprev_hash;
};
```

Figura 5.7: Declaração da estrutura de dados de uma página comprimida no cache comprimido, em linguagem C (arquivo `linux/include/linux/comp_cache.h`). No código-fonte da implementação, uma página comprimida é conhecida por fragmento, daí a razão do nome `comp_cache_fragment` para a estrutura.

devem ser alocadas, quais páginas estão sujas, quais estão travadas, entre outras informações. O cache de páginas é composto por um conjunto de `address space`'s. No caso de páginas de swap, um `address space` é empregado, definindo as funções para tratar essas páginas. Nesse caso em particular, como não há um sistema de arquivo do swap, não se utiliza o espaço na estrutura para designar o inode ou dispositivo de bloco. O `mapping` é utilizado com o `index` para identificar a página.

- `index`: Define o índice ou deslocamento dentro do `mapping` acima. Essas duas informações são suficientes para identificar os dados que uma página no cache de páginas armazena. Esses informações são mantidas pelo cache comprimido para todas as páginas comprimidas.

- `swp_buffer`: Endereço da página de memória utilizada como memória temporária durante o processo de escrita (aqui conhecida, por razões históricas do desenvolvimento como buffer do swap). Necessário por questões de concorrência.

- `offset`: Indica qual é a posição, dentro da célula, a partir da qual os seus dados estão localizados. Dessa forma, a página comprimida ocupa da posição `offset`, contada a partir do início da célula, até a posição `offset + compressed_size`.

- `compressed_size`: Tamanho dos dados comprimidos armazenados na célula.

- `flags`: Usadas para sinalizar determinada característica dessa página comprimida. Um exemplo é o caso em que os dados comprimidos vieram de uma página suja, o que a forçará a ser escrita caso essa página comprimida precise ser liberada.

- `comp_page`: Endereço da estrutura que descreve a célula que contém essa página comprimida.

- `next_hash/pprev_hash`: Utilizado para inclusão dessa página comprimida na tabela de hash que armazena todas as páginas comprimidas. O `mapping` e o `index` são utilizados como chaves da tabela de hash, que serve para a verificação e localização de uma determinada página comprimida, visto que esses dados conjuntamente identificam unicamente uma página do cache de páginas.

5.4.3 Páginas Temporárias de I/O

Quando o cache comprimido não pode ou não deve aumentar o seu tamanho, páginas comprimidas armazenadas nele devem ser liberadas. Para tal, as páginas comprimidas sujas devem ser escritas no seu dispositivo secundário de armazenamento para serem candidatas à liberação.

De modo a serem submetidas à operação de escrita, precisamos de uma página temporária que armazene os dados. As razões pelas quais não utilizamos diretamente a página da célula que contém esses dados são descritas a seguir:

1. Os dados das páginas comprimidas precisam ser descomprimidos para efetuar a escrita em dispositivos de armazenamento que não o swap.

2. No swap, em que não é necessária a descompressão, os dados precisam ser alinhados com a página se formos armazenar uma página por bloco. Dessa forma, precisamos de uma página temporária para armazenar esses dados.

3. Mesmo que várias páginas sejam armazenadas juntamente em um bloco de swap, precisamos montar essa página com os dados e os metadados que devem ser escritos no bloco. Logo não é possível fazer uma escrita com os dados diretamente de uma célula.

4. Como uma célula pode conter mais de uma página de memória (como será visto em "referência"), os dados podem cruzar a fronteira dessas duas páginas, o que é mais um impedimento para fazer I/O diretamente com a página que pertence à célula.

5. Por questões de desempenho, durante uma possível escrita de uma página pertencente a uma célula, essa página estaria travada e não seria acessível enquanto a operação não fosse completada.

Por essas razões, adicionamos o suporte a essas páginas temporárias na nossa implementação para permitir a escrita de páginas comprimidas sujas. Essas páginas temporárias são conhecidas por *swap buffers*, por razões históricas do tempo que o cache comprimido comprimia apenas páginas que fossem armazenáveis no swap.

O número de páginas temporárias de I/O é de 32 páginas de memória. Esse número mostrou-se ótimo para os testes efetuados. Números maiores de páginas temporárias não trouxeram benefício suficiente que compensasse o espaço de memória que eles consumiam, enquanto números menores aumentavam o tempo de espera por uma página temporária livre, por fim prejudicando o desempenho.

5.5 Arquivos da Implementação

A nossa implementação do cache comprimido altera e cria 42 arquivos na árvore do código-fonte do Linux. Ela contém cerca de 11 mil linhas de código (incluindo os *headers*), sendo que parte desse montante (cerca de 4900 linhas) deve-se ao código dos algoritmos de compressão aos quais provemos suporte e foram incorporados à implementação graças às suas licenças de distribuição. Algumas alterações foram necessárias para fazer esses algoritmos compilarem e rodarem no espaço do kernel, mas a implementação das funcionalidades originais está íntegra. Na Tabela 5.1, encontramos a listagem de todos os arquivos modificados ou criados pela implementação.

Dado que a implementação possui um volume considerável de linhas de código, foi criado um diretório dentro do `mm/` chamado `comp_cache/` para armazenar os arquivos da implementação. Esses são:

- `Makefile`: Arquivo utilizado pela ferramenta `make` [42] para a compilação e construção do objeto do cache comprimido.

- `WK4x4.c/WKdm.c/minilzo.c`: Código-fonte dos algoritmos de compressão aos quais se é dado suporte na nossa implementação do cache comprimido. Mais detalhes são dados na Seção 3.6.

Documentation/Configure.help	mm/comp_cache/WKdm.c
MAINTAINERS	mm/comp_cache/adaptivity.c
arch/i386/config.in	mm/comp_cache/aux.c
fs/buffer.c	mm/comp_cache/free.c
fs/inode.c	mm/comp_cache/main.c
fs/ncpfs/dir.c	mm/comp_cache/minilzo.c
fs/proc/proc_misc.c	mm/comp_cache/proc.c
fs/smbfs/dir.c	mm/comp_cache/swapin.c
include/linux/WK4x4.h	mm/comp_cache/swapout.c
include/linux/WKcommon.h	mm/comp_cache/vswap.c
include/linux/WKdm.h	mm/filemap.c
include/linux/comp_cache.h	mm/memory.c
include/linux/fs.h	mm/mmap.c
include/linux/lzoconf.h	mm/mremap.c
include/linux/minilzo.h	mm/oom_kill.c
include/linux/mm.h	mm/page_alloc.c
include/linux/swap.h	mm/page_io.c
include/linux/sysctl.h	mm/shmem.c
mm/Makefile	mm/swap_state.c
mm/comp_cache/Makefile	mm/swapfile.c
mm/comp_cache/WK4x4.c	mm/vmscan.c

Tabela 5.1: Arquivos criados (em cinza) ou modificados (em preto) no Linux pela nossa implementação do cache comprimido.

- `adaptivity.c`: A heurística da política de adaptabilidade do tamanho do cache comprimido está nesse arquivo. Nele também se encontra a política de suspensão da compressão de páginas limpas, além de funções auxiliares para o redimensionamento de tabelas de hash, das tabelas de endereços virtuais de swap, entre outras.
- `aux.c`: Contém funções auxiliares diversas utilizadas ao longo do código. Entre elas, funções das tabelas de hash, de procura (quando da inserção de uma nova página comprimida), de busca (para recuperar uma página comprimida) e para manipulação das listas LRU de ordenação das páginas.
- `free.c`: Funções para remoção das páginas comprimidas e da compactação dessas dentro de uma célula.
- `main.c`: Possui a função principal de inicialização do cache comprimido, assim como as funções de alto nível do armazenamento de páginas no cache comprimido.
- `proc.c`: Funções de estatística do cache comprimido, como número de páginas comprimidas e descomprimidas, número de páginas lidas de voltas, número de páginas escritas, entre outras. Também contém funções de tratamento das entradas no diretório especial `/proc`. São armazenadas funções de compressão e descompressão de alto nível (ou seja, as funções que chamam as funções que realmente comprimem).
- `swapin.c`: Aqui temos funções de leitura de páginas comprimidas do cache comprimido para uma nova página alocada. Funções auxiliares de uso de funções de sistemas de arquivos também possuem o seu código aqui.
- `swapout.c`: Importantes funções de procura de espaço para uma nova página comprimida estão armazenadas aqui. Entre elas, incluem-se as funções que liberam as páginas comprimidas e também as funções que efetuam o gerenciamento da escrita de páginas comprimidas sujas. Para essa escrita, o gerenciamento dos buffers de swap é necessário e as funções para tal estão armazenadas aqui.
- `vswap.c`: As funções para tratamento dos endereços virtuais de swap estão nesse arquivo.

Capítulo 6

Parte Experimental

Nesse capítulo, nós descrevemos o conjunto de testes, a metodologia e os resultados da nossa implementação do cache comprimido no Linux 2.4.18.

O código-fonte da implementação do cache comprimido com todos os programas de teste usados e os seus dados de entrada estão disponíveis no sítio do projeto [35]. Os resultados presentes nessa dissertação foram rodados com a versão 0.24pre6 do nosso código.

6.1 Descrição do Conjunto de Testes

Nessa seção apresentamos cada teste utilizado para avaliação da nossa implementação de cache comprimido. Há uma descrição breve do funcionamento de cada teste, juntamente com a configuração do sistema que foi utilizada para a sua execução.

Os testes descritos abaixo foram selecionados seguindo alguns critérios, que são apresentados a seguir:

1. Testes com uso intenso da memória ou do sistema de I/O.

   Selecionamos testes que fazem intenso uso da memória pois somente com esses testes a influência do cache comprimido no desempenho dos aplicativos pode ser detectada. Em geral, esse tipo de teste sofre uma grande degeneração na sua execução quando a memória necessária para o seu working set não está disponível.

2. Experimentos com aplicativos reais.

   Procuramos efetuar experimentos com aplicativos reais, inclusive aplicativos populares para medir o desempenho do kernel do Linux [28], como o próprio processo de compilação dele. Outros também presentes na nossa lista de aplicativos reais são aplicativos científicos como o MUMmer [48], o Matlab [44] ou a execução do GIMP [21].

   Tentamos efetuar a medição do impacto do cache comprimido na utilização de um desktop padrão, mas não encontramos metodologia adequada. Não há nenhum benchmark específico

para essa medição, e não encontramos programas que permitissem a reprodução de scripts que executassem um conjunto determinado de eventos em modo gráfico, de modo reproduzível.

3. BenchMarks sintéticos com dados realistas.

   Além de alguns experimentos com aplicativos reais, fizemos também testes com benchmarks sintéticos que utilizassem dados que eram considerados realistas, e dessa forma não beneficiassem o cache comprimido por uma alta compressibilidade.

   Encontrar benchmarks com dados realistas não foi uma tarefa fácil. Geralmente, os benchmarks sintéticos não são criados levando em conta a compressibilidade dos dados.

Com todos os experimentos, o sistema foi especialmente preparado para a sua execução de modo a melhor avaliar o impacto da execução desses sistemas sob a mais diferentes pressões de memória. Essas pressões variaram desde a inexistente pressão de memória, quando o uso do cache comprimido não faz diferença significativa alguma, até o momento em que há uma grande pressão de memória no sistema. Em particular no último caso, ao definirmos o tamanho da memória, selecionamos o tamanho que implicasse em uma grande pressão de memória, porém ainda realista.

O Linux permite que no instante do boot seja lhe informado qual parcela da memória deverá ser utilizada. Dependendo do aplicativo a ser testado, vários tamanhos de memória de forma a implicar alguma pressão de memória foram utilizados. Acreditamos que, mesmo em casos de experimentos em que a faixa de valores utilizada para definir a memória é inferior à quantidade média de memória que esteja disponível em sistemas atuais, os resultados desses experimentos são um indicativo do impacto do cache comprimido para sistemas com mais memória disponível.

Os três primeiros testes apresentados abaixo, especificamente a compilação do kernel do Linux 2.4.18, MUMmer 1.0 e o Open Source Database BenchMark (OSDB) 0.14, foram os testes que balizaram o desenvolvimento do cache comprimido. Por essa essa razão, são apresentados resultados de um grande variedade de configurações do cache comprimido para esses testes. Na Seção 6.4.6 apresentamos os resultados dos demais testes comparando um kernel sem o cache comprimido e outro com a configuração que consideramos ser a melhor para o cache comprimido.

6.1.1 Compilação do kernel do Linux 2.4.18

A compilação do kernel do Linux é um benchmark muito realista com alto uso da CPU e de memória, principalmente quando é feito através de diversos processos concorrentes. O processo de compilação consiste em, dada uma configuração do kernel do Linux, compilar os arquivos relacionados às funcionalidades configuradas. Esse processo é muito realista por fazer alto uso da CPU e ser bastante influenciado por mínimas alterações na memória, em particular quanto ao cache de dados de arquivos.

Os nossos testes consistiram em executar a compilação do kernel do Linux 2.4.18. O nível de concorrência durante a execução foi definido através da opção `-j` da ferramenta `make` [42] utilizada para a construção do kernel. Compilamos com apenas um processo (`-j1`) e com dois e quatro processos concorrentes (`-j2/-j4`). Utilizamos o código-fonte do kernel do Linux 2.4.18 conforme disponível em [28], sem nenhum patch adicional. A configuração do kernel que compilamos é a padrão do kernel 2.4.18.

A memória configurada para ser disponível ao sistema foi decidida de modo a verificar o comportamento do cache comprimido com esse teste em situações de maior pressão de memória até uma leve pressão de memória. Os valores de memória, dessa forma, variaram de 18 a 48 Mb. A partir de 48 Mb, a pressão de memória não se altera, ou se altera imperceptivelmente, o que torna experimentos com maiores quantidades de memória desinteressantes.

A compilação do kernel utiliza uma parcela de suas páginas para armazenar os dados gerados para a compilação dos arquivos, mas uma importante quantidade de memória, que corresponde a maioria das páginas da memória durante esse processo, é utilizada para o cache de dados de arquivos normais. A árvore do kernel do Linux 2.4.18 possui cerca de 150 Mb, e apesar de nem todos os arquivos serem compilados de acordo com a configuração e com a arquitetura, é um número grande de arquivos que devem ser lidos e armazenados na memória para efetuar a compilação. Também é necessário espaço na memória para os arquivos de saída da compilação (arquivos objetos).

Esse teste foi utilizado para verificar a estabilidade do código, assim como seu desempenho. Ele também possui um grande valor por ser um dos benchmarks mais referenciados na lista de discussão do kernel do Linux [29].

6.1.2 MUMmer 1.0

O MUMmer [48] é um aplicativo científico com um grande uso de memória. A sua função é executar o alinhamento rápido de grandes seqüências de DNA, em particular genomas completos de batérias.

Para os nossos experimentos, alinhamos genomas de duas espécies de *Xanthonomas* recentemente seqüenciados pelo projeto Genoma-FAPESP [20]. Cada arquivo possui aproximadamente 5 Mb de tamanho, o que faz o MUMmer atingir o pico de uso de memória de mais de 400 Mb de uso durante a sua execução. De modo a repetir o processo de averiguar o comportamento do cache comprimido sob grande pressão de memória até uma leve pressão de memória, determinamos que a memória disponível ao sistema para a execução dos testes variasse de 330 a 500 Mb. No caso de 500 Mb, não possuímos qualquer pressão de memória, ao passo que com 330 Mb estamos a beira de afetar o working set do aplicativo seriamente, o que pode ser notado quando iniciamos o sistema com 320 Mb, pois o tempo de execução passa de dois minutos para cerca de duas horas.

A memória usada pelo MUMmer se caracteriza pela presença maciça de páginas armazenáveis no swap. A quantidade de memória utilizada para o armazenamento do código executável é de menos de 100 Kb, o que é irrisório perto da quantidade de memória consumida por outras páginas que são necessárias para o alinhamento dos genomas utilizados como entrada.

Utilizamos esse teste para averiguar o desempenho da implementação sob essas condições, além de servir para testar a sua estabilidade.

6.1.3 Open Source Database BenchMark (OSDB) 0.14

O OSDB [53] é um benchmark que executa uma conjunto de operações de banco de dados em cima de um conjunto de dados pré-definidos ou gerados. Cada operação é medida separadamente e o tempo total da execução das diversas operações, contabilizado pelo próprio OSDB, é a prin-

cipal medida do benchmark. Ele possui suporte para trabalhar com os gerenciadores de banco de dados MySQL [49] e PostgreSQL [55]. Ademais, o OSDB pode ser executado como apenas uma thread executando as operações no banco de dados ou diversas threads executando as threads concorrentemente.

Nós utilizamos a versão 0.14 do OSDB para a execução dos nossos experimentos. Utilizamos o gerenciador de banco de dados PostgreSQL e um banco de dados de 40 Mb disponível no sítio desse projeto. Em nossos experimentos, foram executados testes com o OSDB com apenas uma thread. Nós rodamos experimentos com 24 e 48 Mb de memória no sistema. O caso de 48 Mb quase não tem pressão de memória, ao passo que em um sistema com 24 Mb de memória, o OSDB executa sob intensa pressão de memória.

As páginas de memória utilizadas durante a execução do OSDB são compostas quase completamente por páginas armazenando dados de arquivos. No caso, os dados do banco de dados que está sendo manipulado pelo OSDB.

Esse benchmark foi utilizado para verificação do desempenho do cache comprimido, assim como para testar a sua estabilidade.

6.1.4 Matlab 6.0

O Matlab [44], cujo nome vem de *Matrix Laboratory*, é uma ferramenta para efetuar computações numéricas e para elaborar gráficos. Ele é especialmente desenvolvido para computações envolvendo matrizes tais como a resolução de sistemas de equação linear, o cálculo de autovalores e autovetores e a fatoração de matrizes. Essa ferramenta funciona em um ambiente de programação interativo, podendo ter a sua entrada como um script. Quanto a licença de uso, é um software comercial, produzida pela The MathWorks, Inc. [43].

Em nossos experimentos, fizemos uso do Matlab para o cálculo da dimensão fractal de uma imagem que faz intenso uso de memória e de processamento. Esse cálculo é feito utilizando a imagem como sendo de 3 dimensões, sendo a terceira dimensão o valor do pixel. Utilizamos três imagens, cujo uso de memória para o cálculo da dimensão fractal é respectivamente de 80, 256 e 1000 Mb. Esse uso de memória é composto, na sua imensa maioria (mais que 95%), por páginas armazenáveis no swap.

A intenção da execução desse experimento é a medição do desempenho do cache comprimido com mais um aplicativo científico que faça amplo uso de memória.

6.1.5 piGIMP 1.0

O piGIMP [25] é um projeto que intenciona melhorar o desempenho do aplicativo gráfico GIMP [21]. Ele consiste de um script feito utilizando a linguagem Perl-Fu com os contadores Perl de alta resolução para obter estatísticas de como estão funcionando os plug-ins GIMP. Dessa forma, o script do benchmark executa cinco operações do GIMP em modo batch, alguns com parâmetros diferentes, totalizando em onze execuções. Ao término, é gravado um arquivo de log com o tempo de execução deles. Os cinco scripts são:

– IIR Gaussian Blur

- Rotate
- Unsharp Mask
- Scale
- Radial Blur

O uso de memória do script do piGIMP em si é insignificante. No entanto, ao ser executado, o piGIMP executa o GIMP para rodar determinada operação. O GIMP ocupa até 47 Mb durante a execução do benchmark, além de cerca de 2 Mb ocupado pelo plugin responsável pela operação, 3 Mb para o servidor de Perl e 3 Mb para o processo do script-fu, todos necessários para a execução do benchmark. Dessa maneira, o total de memória diretamente consumida pela execução desse teste é de mais de 50 Mb. Além disso, esse teste é executado no ambiente gráfico X Window [86, 87], que, na versão que utilizamos, o seu servidor consome ao menos 40 Mb de memória virtual (e 8 Mb de memória física), além de ambientes como KDE [27] que foram iniciados e que tem um consumo alto de memória. Com base nesses dados, nos nossos experimentos, variamos a quantidade de memória no sistema de 48 Mb a 96 Mb.

A memória utilizada ao longo da execução desse benchmark é composto por aproximadamente 1/3 de páginas armazenando dados de dispositivos secundários de armazenamento e 2/3 de páginas com dados armazenáveis no swap.

Esse benchmark foi executado com o fim de medir o desempenho do cache comprimido com esse tipo de aplicativo. Como sempre, a estabilidade do código foi verificada também.

6.1.6 httperf 0.8

O httperf [47] é uma ferramenta para medir o desempenho de servidores web, simulando uma base de usuários infinita através da geração e manutenção uma grande carga. Ele pode ser usado para executar diversos tipos de medições de servidores web, incluindo medições do tipo SPECweb/WebStone, s-client e baseado em sessões.

O seu uso de memória não é muito grande, menos de 2 Mb é utilizado pelo próprio httperf. Para servir as requisições dele, são disparadas instâncias do servidor web, que constituem o maior uso de memória durante a sua execução. Cada instância do servidor web, que no nosso caso é o Apache [2], ocupa cerca de 3 Mb de memória virtual, mas deve ser observado que uma parte dessa memória é compartilhada entre as diversas instâncias do mesmo executável. São disparadas cerca de 150 instâncias do Apache durante a execução do httperf.

O nosso experimento executa um número de requisições do arquivo `index.html` ao servidor web localizado na própria máquina e mede o número de requisições por segundo que foi observado. São executadas um milhão de conexões e em cada conexão são feitas 20 mil requisições desse arquivo, sendo que a taxa de criação de conexões é de 100 por segundo. A memória disponível ao sistema variou de 32 Mb, sob intensa pressão de memória, até 64 Mb, quando não há praticamente nenhuma pressão para a execução desse benchmark.

Esse benchmark foi utilizado para medir o desempenho do cache comprimido, possuindo como objetivo secundário verificar a sua estabilidade em rodar em cenários diversos.

6.1.7 Sort - GNU textutils 2.0

O programa sort [69] é parte do pacote de utilitários de texto da GNU. A sua função é efetuar a ordenação de um arquivo texto utilizando o mínimo de memória.

A utilização de memória diretamente pelo sort é de cerca de 2 Mb para ordenar um arquivo com mais de 100 Mb de tamanho. Dessa forma, a maior parte da utilização da memória, quando o sort é executado, é para o cache de páginas do arquivo que está sendo ordenado. Os resultados para os experimentos do nosso trabalho foram coletados com 24 Mb de memória no sistema.

Ele não foi utilizado como um teste para medir o desempenho do cache comprimido por possuir um comportamento não condizente com os demais programas, como fazer um uso muito pequeno da memória e baixa utilização dos dados utilizados uma vez. Por outro lado, por possuir essas características fortes de baixa utilização dos dados que foram comprimidos, esse teste foi utilizado para a verificação do comportamento do cache comprimido num cenário que é desfavorável para a concepção de cache comprimido que havia até esse trabalho. A suas características marcantes deixam mais nítido o comportamento do cache comprimido.

6.1.8 PostMark 1.4

O PostMark [56] é um benchmark para medir o desempenho do sistema de arquivos para aplicativos que utilizam muitos arquivos pequenos. Ele cria um grande número de arquivos que continuamente têm os seus conteúdos modificados e mede as taxas de transação para um workload que tenta se aproximar ao de um grande servidor de email da internet.

Esse benchmark foi rodado com 128 Mb de memória disponível no sistema utilizado de modo a apresentar um cenário específico que torna bastante interessante uma análise sobre a presença do cache comprimido.

O uso de memória pelo PostMark é muito pequeno, não atinge 3 Mb. Como ele lida apenas com arquivos, quase 100% das páginas utilizadas no sistema são pertencentes a páginas que estão vinculadas a algum arquivo em disco. Nesse caso, o uso de memória dele não se limita ao seu executável somado a memória alocada, mas também às páginas que estão na memória (mais especificamente no cache de páginas) devido ao seu uso. Utilizando essa contabilização, utiliza-se mais de 128 Mb para os processos do sistema juntamente com a execução do PostMark.

Assim como o sort, o PostMark não foi utilizado como um teste para medir o desempenho da nossa implementação. Ele é um benchmark sintético que possui dados incomprimíveis (taxa de compressão maior que 95%) e praticamente nenhuma de suas páginas lidas são reaproveitadas depois da primeira utilização. Essas contundentes características, que foram fundamentais para a não utilização dele como um teste de desempenho, foram o motivo para a sua escolha na verificação do cache comprimido em situações como essas por deixaram mais destacado o comportamento da nossa implementação.

6.1.9 contest 0.51

O contest [34] é um benchmark desenvolvido com a finalidade de medir o tempo de reatividade do kernel do Linux. Isso é feito ao medir o tempo de compilação do kernel do Linux em situações

diferentes de carga do sistema. Essas situações de carga são mantidas durante a compilação do kernel e o objetivo de cada uma delas é tentar simular workloads reais que ocorrem por curtos períodos de tempo em máquinas do dia-a-dia. As cargas que são utilizadas, concorrentemente à compilação do kernel, são:

- **Null Load**

  Sem carga alguma, ou seja, a compilação do kernel é executada sem a presença de alguma carga concorrente no sistema.

- **Process Load**

  Efetua um fork e executa $4 \times CPUs$ ($CPUs$ é o número de CPUs) processos, conectados através de pipes unidirecionais, que ficam passando uma determinada quantidade de dados.

- **Memory Load**

  Repetidamente 110% da memória física é referenciada seguindo um padrão que objetiva causar cache misses.

- **IO Load**

  Copia continuamente o `/dev/zero` para um arquivo do tamanho da memória física.

- **Read Load**

  Lê um arquivo do tamanho da memória.

- **List Load**

  Lê o sistema de arquivo inteiro do diretório raiz. O comando utilizado é `ls -lRa`.

- **CTar Load**

  Repetidamente cria um arquivo do tipo `tar` contendo toda a árvore do kernel do Linux.

- **XTar Load**

  Extrai repetidamente um arquivo do tipo `tar` contendo a árvore do kernel do Linux.

Dependendo da carga que é executada com a compilação do kernel, o contest possui uma configuração de uso de memória diferente. A compilação do kernel, conforme dito acima, tem uma maioria de páginas armazenando dados de arquivos. Como a maioria das cargas lida com arquivos, no final a maior parte da memória utilizada é para o armazenamento de dados de arquivos, exceção feita à carga do "memory load".

Independente do tamanho da memória, a execução do contest com a carga de "memory load" utiliza toda a memória disponível no sistema, causando a utilização do swap (principalmente pelos dados do "memory load"). O "memory load" em si aloca 110% da memória disponível no sistema com valores nulos.

Esse teste não foi utilizado diretamente como benchmark, por ser um teste sintético cujos dados são altamente comprímiveis (para cerca de 1%) e por provocar, ao se utilizar o cache comprimido,

alterações no reescalonamento. No entanto, essas alterações nítidas no escalonamento em situações em que há vários processos rodando ao mesmo tempo no sistema foram a razão para o utilizarmos na explicação e tratamento desse fenômeno.

6.2 Testes Previamente Considerados

Nesse seção apresentamos testes que, em uma primeira instância, consideramos incluir em nosso conjunto de testes. Em uma avaliação posterior, foram descartados por serem benchmarks sintéticos que possuíam dados que não eram condizentes com a realidade da maior parte dos aplicativos reais.

É importante observar que a maior parte deles foi descartada mesmo que tenha sido obtido um desempenho melhor pelo cache comprimido. Neste caso, achamos que essa melhora é devida a dados que são artificialmente criados e cujos benchmarks foram criados sem levar em conta que poderiam ser comprimidos.

6.2.1 dbench 1.3

O dbench [71] é um benchmark que executa cerca de 90 mil operações de I/O, as mesmas chamadas que um servidor Samba produziria rodando o netbench [50]. Foi, e de alguma maneira ainda é, o padrão para gerar carga no sistema do VFS (Virtual File System) do Linux. Ele é comumente utilizado para verificar a estabilidade de novos kernels de desenvolvimento, além de servir como um dos benchmarks que muitos desenvolvedores Linux utilizam para medir o desempenho do kernel.

Pelo fato do dbench ser comumente usado para a medição do desempenho do kernel do Linux (já foi o mais popular no passado), verificamos a possibilidade de executá-lo para verificar o impacto do cache comprimido em um benchmark como esse, cujo efeito é sobre o sistema de arquivos. Chegamos a resultados bastante animadores com esse benchmark mas, ao verificar a compressibilidade dos seus dados, notamos que esses dados são de altíssima compressibilidade. Pela nossa política de incluir apenas benchmarks sintéticos que apresentassem comportamento e dados realistas, o dbench foi considerado inadequado e, por essa razão, foi descartado como teste para medir o desempenho do cache comprimido.

6.2.2 Memtest 0.0.4

O memtest [45] é um conjunto de testes desenvolvido para verificar a estabilidade e consistência do sistema de gerenciamento de memória do Linux. Ele é composto pelos seguintes testes:

- `fillmem`

  Testa a alocação de memória no sistema. Útil para verificar o sistema de memória virtual quanto à alocação de memória, paginação e utilização consistente do swap. O parâmetro para esse teste é o tamanho da alocação de memória. O seu funcionamento consiste simplesmente da alocação e uso da memória do tamanho definido.

- `mmap001`

  Esse teste mapeia um arquivo em disco do tamanho determinado pelo usuário. Ao término do mapeamento, esse arquivo tem os seus dados atribuídos por valores seqüenciais. Por fim, é sincronizado com o disco e é apagado do sistema de arquivos.

- `mmap002`

  Análogo ao mmap001, esse teste mapeia dois arquivos, sendo um do tamanho da memória determinado pelo usuário e outro do dobro desse tamanho. Um dos arquivos tem dados atribuídos e é sincronizado no disco, seguido pela atribuição ao dois arquivos e a sincronização deles com o disco.

- `shm-stress`

  O shm-stress é utilizado para testar o kernel com relação à memória compartilhada privada. Um número de processos são disparados acessando (lendo e escrevendo) em uma memória compartilhada (SHM) e verificando a consistência dos dados.

- `mtest`

  Similar ao shm-stress, o mtest verifica a consistência de dados para um número de processos acessando e modificando o mesmo conjunto de dados. Nesse caso, não é criada uma área de memória compartilhada, mas há compartilhamento das mesmas páginas de dados privadas.

- `misc001`

  Esse teste roda diversos testes ao mesmo tempo por tempo indeterminado. O primeiro consiste em alocar um buffer e ficar "sujando" esse buffer (através de atribuição de valores diversos). O segundo fica alocando memória, a utilizando (atribuindo um valor aleatório a toda essa memória) e a liberando. E por fim, o terceiro mapeia um arquivo na memória, utiliza a memória atribuindo valores ao arquivo mapeado, e o desmapeia.

- `ipc001`

  Esse programa verifica o sistema de memória compartilhada. É alocado um determinado número de processos, que alocam segmentos de memória compartilhada de acordo com o número de iterações pré-estabelecido.

O memtest é muito utilizado durante o desenvolvimento de alterações no sistema de gerenciamento de memória para a verificação da estabilidade. A princípio consideramos verificar o desempenho com esse tipo de teste simples, mas ele não foi utilizado por se tratar de um benchmark sintético com dados não realistas. A compressibilidade dos seus dados é muito alta em todos os seus testes, pois utilizada números de valor baixo ou zeros do arquivo `/dev/zero`.

6.2.3 OSDL Database Test 1

O Database Test 1 [14], ou DBT1, do Open Source Development Lab [53], simula as atividades de usuários da Web pesquisando e comprando itens de uma livraria online. Ele é baseado nas

características do benchmark TCP-W [70] do Transaction Processing Performance Council. Ele utiliza o banco de dados SAP [63] para gerenciamento do dados para o benchmark.

Esse benchmark é bastante interessante, em particular por ser um benchmark que simula uma atividade muito comum de servidores hoje em dia, que é a funcionalidade de comércio eletrônico. Infelizmente os dados gerados para esse benchmark também não são realistas e, como os demais testes dessa seção, possuem alta compressibilidade.

6.3 Metodologia

Nossos experimentos foram rodados em um sistema com um processador Intel Pentium III de 1 GHz, 768 Mb de memória RAM e um disco rígido de 60 Gb, UltraDMA 100 e de 7200 rpm. O disco está configurado para ter uma partição de swap de aproximadamente 1 Gb, que é mantida constante ao longo dos nossos experimentos. Apenas o tamanho da memória disponível ao sistema é variado de acordo com o experimento, como explicado anteriormente. Isso é feito através de um parâmetro do kernel do Linux (`mem=`) que especifica a quantidade máxima de memória que ele usará.

A distribuição Linux instalada nesse sistema foi a Debian [15], versão Sarge [16]. Cada teste foi rodado depois de uma reinicialização do sistema de modo a evitar efeitos de "hot cache". Em particular, esse efeito é ainda mais sério por dados continuarem no cache comprimido, o que permite que a quantidade de dados que estão em cache no sistema seja maior. Apenas uma coleta de dados foi efetuada para os testes. Isso se deve pelo fato de que, na grande maioria dos casos, a variação é quase desprezível, contudo alguns testes provavelmente exigiriam repetidas coletas de dados para melhor precisão.

6.4 Resultados de Desempenho

Nessa seção apresentamos todos os resultados de experimentos que efetuamos, apresentando uma análise detalhada de cada um desses resultados. Inicialmente apresentamos os testes efetuados que justificaram algumas decisões de projetos apresentadas na Seção 3.5, com uma análise de cada resultado que culminou na decisão de projeto apresentada anteriormente. Em seguida, experimentos adicionais verificando o desempenho do cache comprimido são exibidos. E por fim, apresentamos efeitos e situações diversas que verificamos e não foram relatadas por nenhum trabalho anterior (veja Capítulo 7).

6.4.1 Resultados Gerais

Nessa seção, são apresentados e comentados de uma maneira geral os resultados para a compilação do kernel do Linux para diversos níveis de concorrência (*kernelj1*, *kernelj2* e *kernelj4*), execuções do aplicativo científico MUMmer (*Mummer*), do Open Source Database BenchMark (*OSDB*), execuções de um script no Matlab (*Matlab*), diversos plugins do GIMP no benchmark piGIMP (*piGIMP*) e do benchmark de servidores web httperf (*httperf*). Os resultados desses testes

teste	**mem**	**semCC**	**referência**	**lento**	**agress.**	**soswap**	**wkdm**	**wk4x4**	**lzowkdm**	**celula1**	**celula4**
	Mb	seg	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)
	18	467.8	21.73	18.07	21.46	-4.65	2.11	2.58	11.07	19.06	-0.55
	21	326.68	8.89	8.20	7.97	-1.01	3.20	3.83	5.71	7.33	1.64
	24	289.05	0.20	-0.35	0.66	-0.69	-1.44	-0.94	-0.44	0.49	-1.56
kernelj1	27	280.45	0.11	-0.64	0.17	-0.31	-0.44	-0.63	-0.09	-0.19	-0.60
	30	278.33	-0.23	0.11	0.03	0.46	0.48	-0.06	0.39	0.35	0.65
	48	274	0.18	0.09	0.28	0.47	0.31	0.19	0.33	0.26	0.34
	768	271.17	-0.27	-	-	-	-	-	-	-	-
	18	1002.62	33.13	11.68	31.76	1.60	0.36	-1.37	0.30	28.86	-4.97
	21	608.98	33.84	21.99	30.40	-4.12	4.76	1.96	9.50	25.19	-4.41
	24	395.05	18.78	12.55	19.38	1.22	-0.18	3.44	11.26	15.59	0.43
kernelj2	27	313.8	5.37	6.80	6.92	0.00	2.72	1.47	4.14	6.32	-0.53
	30	283.7	1.12	0.92	0.24	-0.00	-1.36	-0.50	0.60	1.38	-0.23
	48	272.3	0.19	0.37	0.28	0.51	0.40	0.53	0.36	0.31	0.52
	768	269.76	-0.01	-	-	-	-	-	-	-	-
	18	1826.14	14.98	11.84	9.77	-0.52	6.73	8.81	5.01	13.31	-9.91
	21	1067.47	15.62	8.38	-1.50	-5.41	-5.67	-2.88	-5.05	12.31	-11.72
	24	826.44	31.85	-2.16	20.60	-0.52	-1.73	1.61	-1.98	28.78	-7.65
kernelj4	27	654.83	34.72	7.90	29.08	3.86	1.47	9.67	-6.64	33.09	2.01
	30	489.67	26.45	13.45	25.47	0.84	0.34	1.98	3.14	26.48	5.89
	48	274.95	-0.39	-0.84	-0.64	-0.16	-0.56	-0.43	-0.70	-0.48	-0.66
	768	271.23	0.28	-	-	-	-	-	-	-	-
	330	143.5	16.09	16.47	-120.36	-51.03	23.80	29.14	-5.93	37.08	-29.84
	340	115.21	20.74	21.95	14.47	22.18	23.06	28.48	25.23	38.64	13.49
	360	82.86	26.25	26.64	24.04	24.33	26.90	25.38	26.60	24.13	25.67
	380	81.21	16.71	15.66	13.98	12.26	15.66	22.61	16.75	38.75	19.10
MUMmer	400	80.55	23.02	23.36	20.21	21.44	23.48	24.92	25.95	39.85	20.14
	420	58.51	15.11	14.19	14.10	16.34	20.18	19.55	20.15	14.80	10.48
	500	45.35	-0.22	-0.20	0.02	0.02	-0.26	0.02	-0.04	0.02	-0.07
	768	44.7	-0.09	-	-	-	-	-	-	-	-
	24	1242.4	30.70	31.59	31.91	-1.27	5.51	0.84	29.35	1.05	-7.69
OSDB	48	758.97	-0.07	-0.08	-0.63	0.75	0.30	0.28	0.08	-0.77	0.26
	768	735.5	0.00	-	-	-	-	-	-	-	-
Matlab 1Gb	768	5880.36	6.12								
Matlab 256Mb	768	1977.83	-0.01								
Matlab 80Mb	768	579.30	-0.03								
	48	-	12.37	-	-	-	-	-	-	-	-
	56	-	31.01	-	-	-	-	-	-	-	-
	64	-	10.08	-	-	-	-	-	-	-	-
piGIMP	72	-	3.51	-	-	-	-	-	-	-	-
	80	-	2.40	-	-	-	-	-	-	-	-
	96	-	0.64	-	-	-	-	-	-	-	-
	768	-	-0.38	-	-	-	-	-	-	-	-

teste	**mem**	**semCC**	**referência**	**lento**	**agress.**	**soswap**	**wkdm**	**wk4x4**	**lzowkdm**	**celula1**	**celula4**
	Mb	reqs/s	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)	ganho (%)
	24	38.5	171.38	-	-	-	-	-	-	-	-
	32	117.7	153.40	-	-	-	-	-	-	-	-
	36	1529.1	14.10	-	-	-	-	-	-	-	-
httperf	40	1849	1.40	-	-	-	-	-	-	-	-
	48	1646.1	14.95	-	-	-	-	-	-	-	-
	64	1819	3.20	-	-	-	-	-	-	-	-
	768	1894.1	-0.25	-	-	-	-	-	-	-	-

Tabela 6.1: Nossos resultados para um kernel sem o cache comprimido (*sem CC*), a principal implementação do cache comprimido (*referência*) e kernels que diferem desse por possuírem configurações diferentes.

são apresentados na Tabela 6.1. Nessa tabela é apresentada a comparação do tempo de execução de um kernel (*sem CC*) sem o cache comprimido; um kernel (*referência*) que foi tomado como referência para a comparação (esse kernel é explicado abaixo), e que consideramos ser a melhor configuração para o cache comprimido; e kernels com alguma configuração diferente (diferente política de adaptabilidade, tipo de páginas que armazena, algoritmo de compressão ou número de páginas por célula). Exibimos nessa tabela o tempo que levou para a execução do teste na coluna *sem CC* e nas demais colunas o ganho em relação a esse tempo. Por exemplo, 20% na coluna *referência* significa que levamos 80% do tempo que levou o kernel sem o cache comprimido. No caso do *httperf*, a unidade apresentada na coluna *sem CC* é requisições por segundo. Um ganho negativo significa que houve uma piora de desempenho. Nas figuras subseqüentes desse capítulo (Figuras 6.1, 6.2 e 6.3), apresentamos de modo gráfico os resultados para cada teste comparando o kernel sem o cache comprimido e o kernel tomado como referência.

Nas tabelas e nos gráficos, o kernel *referência* corresponde ao kernel cuja configuração do seu cache comprimido atingiu os melhores resultados em nossos experimentos. Isso significa que ele introduz melhoras na maioria dos workloads, mesmo se outras configurações do cache comprimido possam melhorar mais em alguns casos particulares. Essa configuração do cache comprimido é composta basicamente pelo uso do algoritmo de compressão LZO, de células compostas de duas páginas de memória e da política de adaptabilidade descrita na Seção 4.3.

Em comparação com os resultados do *sem CC* (kernel sem o cache comprimido), nós observamos ganhos significativos com o kernel *referência* nas situações com alta pressão de memória, atingindo até 171,38% de ganho de desempenho. Quando está sob baixa pressão de memória, um pequeno overhead (não mais que 0,39%) ocorre como verificado nos casos *kernelj4* 48 Mb, *Mummer* 500 Mb e *OSDB* 48Mb.

Conforme dito anteriormente, para cada teste nós escolhemos diferentes quantidades de memória disponíveis no sistema de modo a verificar o cache comprimido em situações de quase nenhuma pressão até alta pressão de memória. Por exemplo, 500 Mb é mais que suficiente para armazenar todos os dados que são utilizados pelo aplicativo científico MUMmer durante a sua execução, enquanto 330 Mb é muito perto de degenerar o comportamento do aplicativo intensivamente. De fato, verificamos que, com 320 Mb em um kernel sem cache comprimido, a execução do MUMmer leva quase 2 horas, i.e., cerca de 4600% mais tempo do que com 330 Mb.

Em relação a situações de pressão de memória, é interessante notar que, se a pressão de memória fica maior, o cache comprimido geralmente introduz ganhos positivos ao comportamento do teste que estamos avaliando. No caso da compilação do kernel do Linux 2.4.18 em um sistema com 30 Mb de memória, se o nível de concorrência aumenta (`j1, j2 ou j4`), a pressão de memória fica maior e o cache comprimido começa a fazer uma grande diferença (notadamente no caso `j4`). Isso também pode ser notado, por exemplo, na execução do benchmark httperf.

6.4.2 Outras Políticas de Adaptabilidade

Na Tabela 6.1 são apresentados resultados de kernels com cache comprimido em que as políticas de adaptabilidade são diferentes da adotada no kernel *referência*. Podemos ver esses resultados nas colunas *lento* e *agress*.

A política do kernel *lento* é menos sensível a mudanças do que a política adotada no *referência* (explicada na Seção 4.3). Nessa política, acessos a qualquer uma das listas, para realmente fazer efeito, precisa atingir um determinado patamar e, além disso, precisa compensar possíveis acessos a outra lista. Por exemplo, se os dois primeiros acessos às listas forem à lista custo, para que acessos à lista lucro determinem o comportamento do tamanho do cache comprimido, primeiro é necessário que se tenha dois acessos à essa lista para zerar e a partir daí que se começa a contar a sua influência.

A política do kernel *agress* é mais agressiva que a adotada no *referência*, tentando tomar uma ação mesmo quando há a menor evidência. Nessa política, sempre que há um acesso a uma página comprimida que está localizada na lista custo (veja mais detalhes sobre as listas na Seção 4.3), há uma tentativa de compactar o cache comprimido. Após essa tentativa, caso seja infrutífera, libera-se um fragmento do cache comprimido.

Ao analisar os resultados, verificamos que o kernel com o cache comprimido que utiliza a política lenta provê ganhos menores do que o kernel *referência* no teste de compilação do kernel, enquanto provê resultados semelhantes para a execução do MUMmer e levemente menores para o caso do Open Source Database BenchMark. A política agressiva também provê ganhos menores que o *referência* na média, apesar de prover ganhos levemente maiores nos casos como do Open Source Database BenchMark em um sistema com 24 Mb e a compilação do kernel do Linux, com nível de concorrência `j2`, para sistemas com 24 e 27 Mb de memória.

6.4.3 Compressão somente de páginas armazenáveis no swap

Ao comparar as colunas *soswap* e *referência* da Tabela 6.1, podemos ver a importância de também comprimir páginas não armazenáveis no swap, como discutido na Seção 3.5.1. Vamos avaliar cada um dos casos isoladamente.

Para os casos do MUMmer, exceto para o casos em que o sistema possui 330 Mb, pode-se ver que os kernels *soswap* e *referência* obtêm resultados similares. De fato, isso é conseqüência do fato de que poucas páginas não são armazenáveis no swap nesses casos.

Contudo, no caso da execução do Open Source Database BenchMark, o ganho obtido pelo kernel *referência* no caso em que o sistema possui 24 Mb é anulado quando se comprime apenas páginas armazenáveis no swap, como no kernel *soswap*. Na verdade, há inclusive um pequeno overhead.

O kernel *soswap* produz piora para a maioria dos casos de compilação do kernel do Linux em contraste com o kernel *referência*, que atinge ganhos na grande maioria dos casos. Durante o desenvolvimento, observamos que, ao rodarmos testes de compilação do kernel para caches comprimidos estáticos de diversos tamanhos fixos que armazenem apenas páginas armazenáveis, a mesma piora foi verificada. Por fim, conforme esperado, podemos notar que a compilação do kernel e a execução do OSDB têm uso maior de páginas armazenáveis em outros dispositivos de armazenamento que não o swap.

6.4.4 LZO vs WK4x4 vs WKdm vs LZO+WKdm

Nessa seção comparamos o desempenho de configurações do cache comprimido em que diferentes algoritmos de compressão são utilizados para as páginas armazenadas no cache comprimido. Na Tabela 6.1 possuímos os resultados do kernel com cache comprimido que utiliza o algoritmo de compressão WKdm (coluna *wkdm*), com um kernel com cache comprimido com o algoritmo de compressão WK4x4 (coluna *wk4x4*), e com um kernel que utiliza os algoritmos de compressão LZO e WKdm em conjunto (coluna *lzowkdm*).

	LZO	**WKdm**	**WK4x4**
tempo por compressão	0,09 ms	0,05 ms	0,08 ms
tempo por descompressão	0,04 ms	0,03 ms	0,06 ms
taxa de compressão (kernel)	39,4%	60,6%	61,8%
taxa de compressão (mummer)	35,5%	41,6%	37,3%
taxa de compressão (osdb)	64,5%	86,5%	85,9%

Tabela 6.2: Tempo médio, por algoritmo de compressão, para comprimir e descomprimir uma página e a taxa de compressão média para alguns testes.

Na Tabela 6.2, podemos verificar o tempo médio para comprimir e descomprimir uma página e a taxa de compressão média para alguns testes. Através desses dados, notamos que o algoritmo de compressão WKdm comprime e descomprime uma página mais rapidamente que o LZO, mas não comprime tanto quanto o LZO. No caso do WK4x4, o mesmo vale para a compressão, ou seja, ele comprime as páginas mais rapidamente que o LZO. Contudo, a descompressão é mais lenta que os demais algoritmos.

Através dos resultados de desempenho, podemos verificar que o kernel com cache comprimido que utiliza o WKdm executa bem nos testes onde a taxa de compressão obtida com ele não é substancialmente pior que a taxa obtida pelo LZO, como nos casos da execução do aplicativo MUMmer. Essa verificação também é válida para o cache comprimido que utiliza o WK4x4. Por possuir taxa de compressão melhor que a obtida pelo WKdm, observamos uma execução ainda melhor do MUMmer quando o WK4x4 é utilizado. Entretanto, nos outros testes, a taxa de compressão mostrou-se mais importante que a velocidade de compressão ganha com a utilização do WKdm ou WK4x4. Isso é conseqüência do fato de que uma taxa de compressão melhor implica que menos páginas serão liberadas e logo mais leituras dos dispositivos de armazenamento poderão ser economizadas.

Deve ser dito que o MUMmer é um aplicativo de uso intensivo da memória, utilizando na sua maioria páginas armazenáveis no swap, enquanto todos os outros aplicativos dependem muito mais da compressão de outras páginas do page cache (veja a análise da Seção 6.4.3 sobre a compressão apenas de páginas armazenáveis no swap). O WKdm e o WK4x4 foram desenvolvidos para comprimir páginas do segmento de dados e, portanto, não foram projetados para comprimir outros tipos de páginas do page cache, por isso eles atingem taxas de compressão não muito diferentes do LZO quando comprimem dados utilizados pelo MUMmer.

Pelo fato do WKdm e do WK4x4 não possuirem um bom desempenho em termos de taxa de compressão com outras páginas além daquelas armazenáveis no swap, mas executam a compressão

em um tempo menor que o exigido pelo LZO, nós efetuamos um experimento no qual o algoritmo de compressão LZO é aplicado a todas as páginas exceto àquelas armazenáveis no swap, que acabam sendo comprimidas pelo WKdm. Esses resultados são apresentados na Tabela 6.1 na coluna *lzowkdm*.

Esse kernel com o cache comprimido que utiliza os dois algoritmos de compressão executa melhor que o *referência* para o MUMmer (exceto 330 Mb), mas os seus resultados são muito insatisfatórios para compilações do kernel, em particular nos casos de alta concorrência durante a sua construção, o `j4`. Isso provavelmente se deve ao fato de que a taxa de compressão média das páginas comprimidas armazenáveis no swap é 36% com o LZO contra 50% com WKdm, o que nos leva a crer que essa diferença de taxa de compressão é mais importante que o tempo economizado comprimindo as páginas. Apesar desse resultado insatisfatório, ainda acreditamos que se poderia obter melhoras nessa direção.

6.4.5 Células de uma, duas e quatro páginas de memória

Aqui apresentamos uma análise sobre configurações do cache comprimido em que o número de páginas de memória na composição das células são diferentes. A configuração com células com apenas uma página de memória e com células de quatro páginas estão na Tabela 6.1 (colunas *celula1* e *celula4*, respectivamente). A configuração *referência* possui células de duas páginas.

Na configuração com células compostas de quatro páginas de memória, o kernel com cache comprimido provê piora – em relação ao kernel sem o cache comprimido e em relação ao kernel *referência* – em quase todos os casos, exceto para o MUMmer. Acreditamos que isso seja devido à fragmentação que ocorre dentro das células, ao custo de compactação das páginas comprimidas dentro das células e a maior dificuldade em alocar células com maior número de páginas contíguas.

A configuração com células compostas por apenas uma página executa quase tão bem quanto células com duas páginas para os testes de compilação do kernel. Nos testes do MUMmer, caso em que há uma boa compressibilidade dos dados, o kernel com a configuração do cache comprimido com células de uma página de memória executa ainda melhor em quase todos os casos, alcançando ganhos de até 39,85%. Entre esses resultados apresentados, o único caso em que o *referência* é claramente melhor que *celula1* é o caso do OSDB em um sistema com 24 Mb de memória. A razão é que nós temos baixa compressibilidade (64,5%), como discutido na Seção 3.5.3. Dessa forma, nesse caso o *celula1* melhora o desempenho do sistema em apenas 1,05%, ao passo que o *referencia* o melhora em 30,70%.

6.4.6 Resultados por Teste

Nessa seção apresentamos uma análise dos resultados de todos os testes que utilizamos para verificar o desempenho do cache comprimido. Aqui, apresentamos uma comparação dos resultados de desempenho do kernel com cache comprimido que nomeamos de *referência* ao longo desse capítulo, que contém a configuração do cache comprimido que consideramos a melhor, e o kernel sem o cache comprimido.

Exibimos aqui os resultados desses kernels rodando os experimentos listados abaixo, cujas descrições mais detalhadas são apresentadas na Seção 6.1.

Compilação do Kernel

A compilação do kernel foi executada em um sistema com uma grande variedade de tamanhos de memória, de 18 a 48Mb, ao simular as diferentes pressões de memória durante a sua compilação, e sem restrições de memória, ao ter disponível para o seu uso os 768Mb disponíveis no nosso sistema. Além disso, esse teste possui uma opção de execução a mais, que é o nível de concorrência entre as instâncias do compilador ao executar a compilação da árvore do código-fonte, conforme explanado na Seção 6.1.

Esse teste atingiu melhoras de até 34,72% em relação ao kernel sem o cache comprimido (veja a Tabela 6.1 e a Figura 6.1). Podemos observar que, quanto maior o nível de concorrência, maior a pressão de memória e a melhora provida pelo cache comprimido. Se verificarmos o *kernelj1*, vemos que apenas o caso de 18Mb possui uma melhora acima de 10%. No caso do *kernelj2*, isso já ocorre com os casos 18, 21 e 24Mb, ao passo que o *kernelj4* tem melhoras acima de 10% para os casos até 30Mb. Exceto no caso em que há pouca ou nenhuma pressão, a nossa implementação oferece benefícios para esse teste. Esse é um resultado bastante interessante tendo em vista que é um popular teste para a medição do desempenho do kernel do Linux e cuja melhora é difícil de se alcançar, conforme a nossa experiência no contato com os desenvolvedores.

Em relação às taxas de compressão, esse teste possui taxas que consideramos realistas. Ela varia de cerca de 40% até cerca de 60% dependendo do algoritmo de compressão (veja Seção 6.4.4).

MUMmer

O MUMmer foi executado em um sistema com quantidades de memória disponíveis diferentes para verificar o desempenho sob diversas condições de pressão de memória. Em todos os casos em que há pressão de memória, o MUMmer possui melhoras quando utilizamos o cache comprimido variando de cerca de 15% até 26,25% (veja a Tabela 6.1 e a Figura 6.2). Uma característica muito interessante dos resultados obtidos com o MUMmer é que, apesar das melhoras obtidas, não há um aumento da melhora obtida com o cache comprimido à medida em que se diminui a quantidade disponível de memória (e portanto aumenta-se a pressão de memória).

Open Source Database BenchMark

Executamos o Open Source Database BenchMark em duas configurações de memória, além da configuração em que não limitamos o uso da memória. Apesar dos dados não serem altamente comprimíveis, conseguimos uma melhora significativa (30,70%) com o uso do cache comprimido quando há pressão de memória. No caso de 48Mb, em que não há pressão, obtemos uma leve piora de desempenho. Veja a Tabela 6.1 e a Figura 6.2 para os resultados.

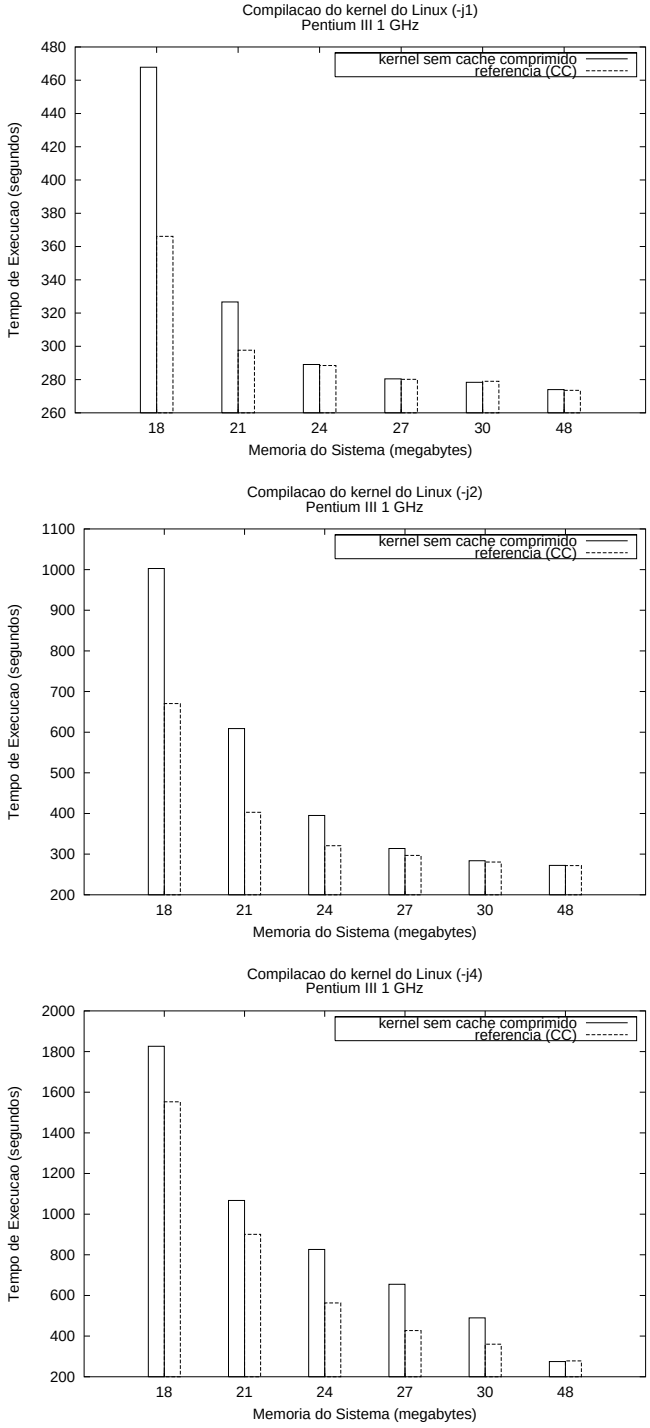

Figura 6.1: Resultados comparando o kernel sem o cache comprimido com o kernel *referência*.

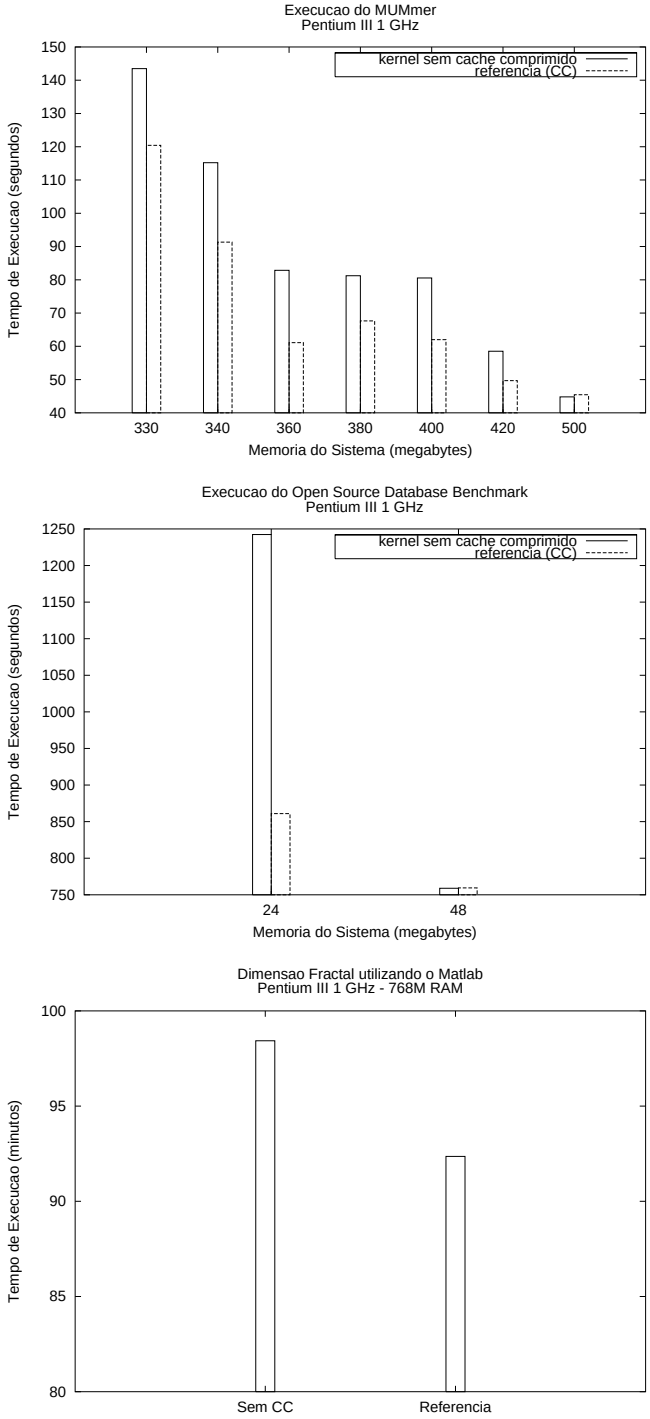

Figura 6.2: Resultados comparando o kernel sem o cache comprimido com o kernel *referência*.

Matlab

O Matlab foi executado em um sistema com 768Mb – o máximo disponível no nosso sistema de testes, e o seu uso de memória atingiu 1Gb. Com esse teste, obtemos cerca de 7% (veja a Tabela 6.1 e a Figura 6.3) de melhora utilizando o cache comprimido em relação a um kernel sem o cache comprimido. Aqui é interessante notar que esse é o primeiro aplicativo real que obtemos alta compressibilidade, sendo que o tamanho comprimido atinge 1% do tamanho do original em média. Suspeitamos que possa haver algum gargalo nas estruturas de dados devido ao grande número de páginas comprimidas por célula, o que deve ser investigado no futuro.

piGIMP

O piGIMP foi executado em um sistema com diferentes tamanhos de memória, conforme pode ser verificado na tabela de resultados de desempenho apresentada anteriormente (Tabela 6.1) e na Figura 6.3. Nesse gráfico apresentamos a melhora de desempenho relativa obtida utilizando um kernel com o cache comprimido em relação a um kernel sem esse cache.

Podemos observar, através desse gráfico, que para valores variando desde uma grande pressão de memória (48Mb) até uma leve pressão de memória (96Mb), o cache comprimido provê benefícios na execução desse popular aplicativo gráfico GIMP, atingindo até uma melhora de mais de 30% para o caso de 56Mb de memória do sistema.

httperf

O httperf foi executado no nosso sistema de testes com diferentes tamanhos de memória disponíveis ao sistema. Na Tabela 6.1 e na Figura 6.3 exibimos os resultados para 24, 32, 36, 40, 48 e 64Mb. Como o uso de memória desse benchmark não é muito grande (na verdade, o uso real de memória vem das instâncias do servidor web que são iniciadas para tratar as requisições), não há diferença notável entre o desempenho nos casos de 40, 48 e 64Mb. No entanto, quando há 24, 32 ou 36Mb, o sistema é colocado sob maior pressão de memória, e é nessa situação que o cache comprimido começa a fazer uma grande diferença. Apesar de bastante degradado em relação aos casos de 40, 48 e 64Mb, o desempenho de um sistema com o cache comprimido com 24, 32 e 36Mb chega a ser mais de 170% melhor do que o kernel equivalente sem a sua presença.

6.4.7 Resultados em Sistema sem Limite de Memória

Nessa seção, apresentamos uma análise dos resultados da comparação entre um kernel sem o cache comprimido, *sem CC*, e o kernel *referência* (que possui a configuração do cache comprimido que consideramos atingir os melhores resultados até o momento), em situações em que não há uma limitação do tamanho da memória para simular situações de nenhuma pressão de memória. Esses experimentos foram efetuados para verificar o impacto do cache comprimido em um sistema com uma grande quantidade de memória, sem nenhuma pressão de memória. É esperado que o cache comprimido não possa prover ganhos de desempenho e queremos verificar se chega a introduzir algum overhead.

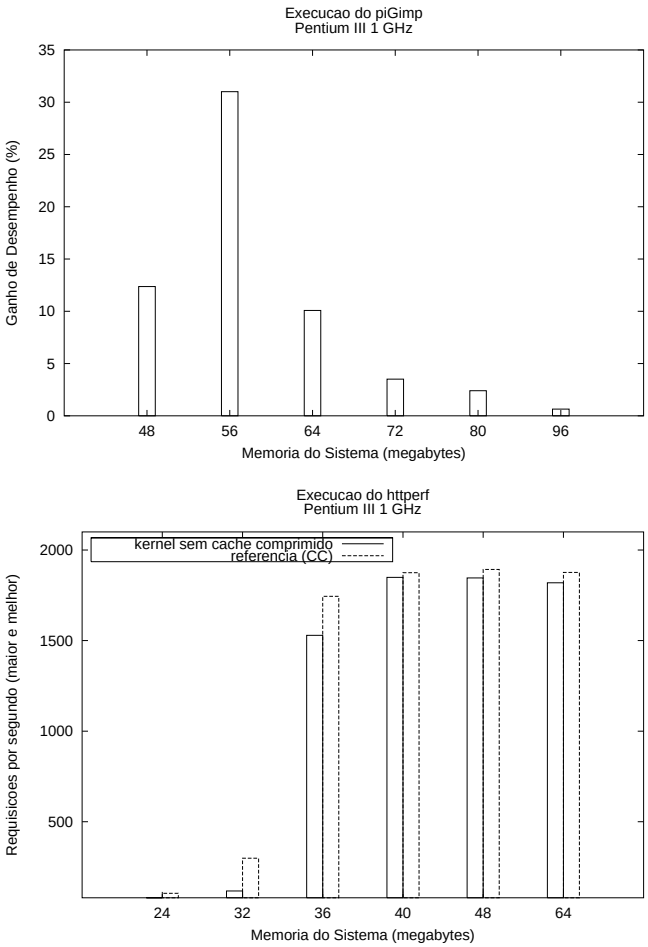

Figura 6.3: Resultados comparando o kernel sem o cache comprimido com o kernel *referência*.

Na Tabela 6.1, apresentamos os resultados de todos os nossos testes num sistema com todos os 768Mb de memória disponíveis no sistema. No caso do Matlab, apresentamos o cálculo de dimensão fractal com dados de entrada que fazem com que o programa utilize menos memória que os 768Mb disponíveis, de modo a verificar o impacto do cache comprimido em situações sem pressão de memória.

Pelos resultados vistos na Tabela 6.1, para a maioria dos casos (*kernelj2*, *MUMmer*, *OSDB*, *Matlab (256Mb)* e *Matlab (80Mb)*), nenhuma diferença foi observada. Para três casos (*kernelj2*, *httperf* e *pigimp*), verificamos que o cache comprimido introduziu overheads de 0,25, 0,27 e 0,38%. Para um caso (*kernelj4*), observamos que o cache comprimido atingiu uma melhora de 0,28%. Dessa forma, acreditamos ser justo afirmar que praticamente nenhum overhead é esperado da nossa implementação se uma aplicação for executada sem nenhuma pressão de memória.

6.4.8 Comportamento sob diferentes pressões de memória

Conforme visto nos resultados da Tabela 6.1 e nas análises acima, o cache comprimido possui diferentes comportamentos dependendo da pressão de memória ao qual o sistema está submetido.

Quando não há qualquer pressão de memória, o cache comprimido não introduz praticamente nenhum overhead, o que é esperado visto que o cache comprimido não é utilizado pelo fato de não possuir nenhuma página sendo armazenada. Consequentemente, ele não precisa tentar se adaptar de modo a evitar introduzir overheads (por exemplo, ao comprimir páginas que não serão reutilizadas pelo sistema).

No outro extremo, quando existe uma razoável pressão de memória, o sistema acaba armazenando em geral um grande número de páginas no cache comprimido. A compressão desse número de páginas e os acessos a elas fornecem dados suficientes para que a nossa heurística (veja Seção 4.3) se adapte bem, conforme verificado experimentalmente, e assim o cache comprimido provenha melhora do desempenho no sistema. Isso ocorre pois a nossa heurística de adaptabilidade funciona como um sistema de controle cujo "feedback" baseia-se na coleta do padrão de acesso às páginas do cache comprimido. Ela ajusta o tamanho do mesmo quando este padrão de acesso aponta para um desajuste da quantidade de memória reservada para o cache comprimido. Este reajuste tenta reverter um comportamento desfavorável e até prover ganhos. É necessária, portanto, uma quantidade razoável de dados para que, de um modo geral, a nossa heurística possa adaptar o cache comprimido ao longo da execução e assim ter melhorias de desempenho no sistema.

Em situações em que há uma baixa pressão de memória, existe uma menor quantidade de dados para uma adaptabilidade ótima da nossa heurística. Acreditamos que esse pouco feedback essa seja uma razão para haver um pequeno overhead quando há baixa pressão de memória, como verificado em análises desse capítulo.

6.5 Desabilitando a compressão de páginas limpas

Nós executamos experimentos com o programa de ordenação do GNU textutils 2.0 [69], o sort, e com o benchmark de sistemas de arquivos PostMark [56], que justificaram modificações que inibissem ou reativassem a compressão de páginas limpas.

Esses programas têm uma queda substancial de desempenho quando o cache comprimido comprime indiscriminadamente todas as páginas que são despejadas pelo sistema de memória virtual., conforme podemos notar na Tabela 6.3 ao comparar o desempenho de um kernel que não desabilita a compressão de páginas limpas com um kernel sem o cache comprimido. O programa sort roda 47,7% mais devagar num kernel com cache comprimido (que não desabilita a compressão de páginas limpas) do que num kernel sem o cache comprimido, ao passo que o PostMark leva 60,3% mais tempo para completar nessas mesmas condições.

O PostMark é um benchmark que foi desenvolvido para a verificação do desempenho do sistema de arquivos e por isso tem uma proposta diferente de aplicativos reais, exibindo comportamentos que não encontramos nas situações realistas que testamos. Como exemplo desse tipo de acontecimento, ao verificar o número de páginas que foram lidas do cache comprimido em relação ao número total de páginas comprimidas, chegamos à relação de 0,0012%, ou seja, na prática nenhuma página comprimida ou liberada pelo sistema é reaproveitada. O cenário ainda é pior pois a taxa de compressão dos dados dessas páginas é de 99%, ou seja, quase todas as páginas são incomprimíveis.

sistema	**sort**	**postmark**
CC - suspendendo a compressão de páginas limpas	55,28s	329s
CC - não suspendendo a compressão de páginas limpas	80,35s	529s
sem CC	54,38s	330s

Tabela 6.3: Tempos de execução dos programas sort e postmark em kernels sem o cache comprimido e com o cache comprimido. Nesse últimos caso, temos com a política que suspende a compressão de páginas limpas, e sem ela.

Ao adotar a nossa política de suspensão da compressão de páginas limpas (veja Seção 3.5.4), houve uma modificação substancial no quadro de piora observado anteriormente. A piora verificada com o sort é reduzida quase totalmente, chegando a apenas 1,65%. No caso do PostMark, a degradação do tempo de execução não foi mais observada. É importante observar que uma pequena diminuição de desempenho é esperada dado que a nossa política leva algum tempo para detectar que comprimir as páginas limpas e as armazenar no cache comprimido pode não estar sendo vantajoso.

6.6 Efeito no Escalonamento

Uma conseqüência sutil do cache comprimido é o seu efeito no escalonamento de processos. Na nossa opinião, esse efeito colateral é bom uma vez que a divisão do tempo da CPU entre os processos é mais justa se o cache comprimido está presente, como nós mostraremos nessa seção. Na verdade, em um kernel sem cache comprimido, sempre que uma falha acontece em uma página armazenada em um dispositivo de armazenamento, uma operação de leitura é submetida ao disco. Dado que ler uma página leva um tempo significativo para completar, a operação de leitura é submetida e o processo corrente libera a CPU. Então outro processo, se disponível, é escalonado para ter controle

da CPU. Por outro lado, em um kernel com cache comprimido, se uma página está armazenada no cache comprimido, ela será descomprimida para servir qualquer falha ocorrida nela. Ao contrário de uma falha em uma página armazenada em um dispositivo de armazenamento, esse serviço não libera a CPU uma vez que essa operação não depende de dispositivos lentos, como discos rígidos.

Graças ao cache comprimido, aplicativos provavelmente terão menos falhas em páginas armazenadas nos dispositivos de armazenamento. Por esse motivo, elas liberarão menos vezes a CPU para servir uma falha de página. Se dois ou mais aplicativos estão rodando no sistema e o cache comprimido evita falhas de páginas de alguns deles que no fim gerariam leituras dos dispositivos de armazenamento, esses aplicativos rodariam muito mais rapidamente que em um sistema sem cache comprimido. Por outro lado, os aplicativos com menos falhas de páginas evitadas poderiam rodar mais devagar porque eles provavelmente não fariam uso do tempo de CPU previamente liberado pelos aplicativos que tiveram mais falhas evitadas.

sistema	**compilação do kernel tempo de execução**	**número de iterações do *mem_load***
CC	94,90 s	174
sem CC	90,64 s	41

Tabela 6.4: Resultados do benchmark Contest 0.51 para a carga *mem_load*. O sistema utilizado para os testes foi configurado para utilizar 256 Mb de memória (acima, **CC** é o sistema com o kernel *referência* que possui o cache comprimido).

Como um exemplo desse efeito, nós apresentamos uma análise do comportamento do contest, um benchmark de reatividade do kernel do Linux [34]. Especificamente, nós discutiremos acerca do teste de carga de memória. Ele consiste em rodar uma compilação do kernel do Linux conjuntamente a um programa (*mem_load*) que aloca 110% do tamanho da memória física e executa acessos diversos a essa memória alocada. A principal medida do benchmark é o tempo que o kernel do Linux leva para compilar, mas o contest também exibe como o *mem_load* executou durante a compilação do kernel, através do número de iterações feitas. Depois de alguns experimentos, verificamos que o programa *mem_load* falha em mais páginas dos dispositivos de armazenamento que a compilação do kernel. Por esse motivo, sem o cache comprimido, o processo de compilação tem a vantagem do tempo ocioso da CPU como uma conseqüência do escalonamento executado pelo *mem_load* para aguardar pelas páginas a serem lidas. Com o cache comprimido, o *mem_load* não tem falhas em páginas armazenadas nos dispositivos de armazenamento uma vez que os seus dados são altamente comprimíveis. Por isso, a compilação do kernel leva mais tempo para rodar em um sistema com cache comprimido, porque o *mem_load* usa uma fatia mais justa da CPU, não a liberando para esperar por páginas serem lidas dos dispositivos de armazenamento. Na verdade, o *mem_load* roda muito mais com o cache comprimido.

Capítulo 7

Trabalhos Relacionados

No Capítulo 1, nós apresentamos um breve histórico e descrição a respeito dos trabalhos anteriores sobre cache comprimido, sejam eles com ou sem implementações. Nesse capítulo, comparamos ao máximo possível nossa implementação com esses trabalhos. Ademais, a partir do entendimento obtido com a nossa própria implementação, também contribuímos com algumas análises iniciais a respeito deles. Estamos particularmente interessados nas implementações de Fred Douglis [18], Russinovich e Cogswell [62], Raúl Cervera et al. [8] e também na proposta de esquema adaptativo de Scott Kaplan [80, 26].

Os trabalhos foram divididos em seções, de acordo com as suas características. Trabalhos envolvendo implementações do cache comprimido estão na Seção 7.1, enquanto trabalhos envolvendo simulações ou apenas estudos teóricos estão na Seção 7.2. Os trabalhos sobre compressão em hardware são apresentados na Seção 7.3 e o trabalho a respeito do gerenciamento de memória do Linux 2.4 está na Seção 7.4.

7.1 Implementações em Software

Aqui apresentamos os dois trabalhos anteriores que envolveram implementações do cache comprimido em software que temos conhecimento. O primeiro trabalho, devido a Fred Douglis [18] em 1993, trata-se de uma implementação de um cache comprimido adaptativo no sistema operacional Sprite [66]. O segundo, por sua vez, que se deve a Cervera et al. [8] em 1999, é uma implementação mais recente de um cache comprimido de tamanho estático no Linux 2.0.

Nós tentamos comparar a nossa implementação com essas anteriores, mas o conjunto de testes das implementações de Douglis e a de Cervera et al. não estão disponíveis. Quanto aos códigos-fonte das implementações em si, apenas a de Cervera et al. está disponível. Contudo, fomos incapazes de rodar alguns dos nossos testes em um kernel do Linux 2.0 com essa implementação, devido a violações de segmento na execução do kernel (*oopses*, na terminologia do Linux)[1]. Não investigamos o suficiente a ponto de poder afirmar termos descoberto novos erros de programação

[1]Estas violações de segmento foram geradas nas alterações de código introduzidas por Cervera et al.

na implementação de Cervera et al. ou se as ferramentas da distribuição Debian [15], versão Potato [17], introduzem incompatibilidades com a versão do kernel do Linux que eles utilizaram.

7.1.1 Cache de Compressão Adaptativo no Sprite

A primeira implementação de cache comprimido foi feita por Fred Douglis [18] em 1993 no sistema operacional Sprite. O seu cache comprimido, chamado de cache de compressão, possuía o tamanho adaptativo, e esse redimensionamento adaptativo foi baseado no algoritmo do Sprite para negociar memória entre o sistema de arquivos e o sistema de memória virtual (VM). Ele compara a idade do bloco de arquivo menos recentemente usado (cujos dados estão presentes em uma página no cache de arquivos), com a idade da página do sistema de memória virtual (i.e., uma página que não contenha dados de arquivos) que seja a menos recentemente usada e libera a mais velha das duas, *modulo* um ajuste a favor de manter as páginas do VM uma quantidade maior de tempo na memória. Na implementação do cache comprimido, como temos também o cache comprimido para competir por memória, a alocação de cada um dos três tipos de memória (páginas do cache do sistema de arquivos, memória não-comprimida e memória comprimida) requer uma comparação das idades das páginas mais velhas de cada tipo de página para efetuar o redimensionamento de cada memória. Quando as idades são comparadas, o sistema tem um viés em favor das páginas comprimidas sobre as páginas não-comprimidas e ambas sobre os blocos de cache de arquivos. Aqui, ao mencionarmos páginas mais velhas, usando a nossa terminologia, referimo-nos às páginas alocadas para as células e não às páginas comprimidas.

O desempenho do cache comprimido implementado por Douglis alcançou resultados inconclusivos: melhora[2] para alguns aplicativos (até 62,3%) e pioras para outros (até 36,4%). Algumas razões foram apresentadas para os resultados que foram negativos, especificamente:

1. baixa compressibilidade dos dados;
2. localidade, que é responsável por fazer um aplicativo falhar em páginas armazenadas em um cache comprimido que seriam acessíveis sem o overhead da compressão e descompressão se ele não existisse;
3. restrições nas operações de I/O (entrada e saída, ou E/S) que não permitem que as leituras transfiram quantidades de dados menores que uma página de memória, beneficiando-se, dessa maneira, da compressão.

Experimentalmente, assim como Douglis, verificamos que a baixa compressibilidade obtida em alguns aplicativos era um problema para o cache comprimido. Em nosso trabalho, abordamos esse problema usando células com um maior número de páginas contíguas de memória (como visto na Seção 3.5.3).

[2]Aqui utilizamos o nosso conceito de ganho apresentado na Seção 6.4.1 que é a porcentagem de melhora em relação ao tempo gasto pelo sistema sem o cache comprimido. Um exemplo: 30% de melhora significa que o sistema com o cache comprimido consumiu 30% menos tempo que o sistema sem o cache comprimido. Douglis utiliza, no seu artigo, a razão entre o tempo do sistema sem o cache comprimido sobre o tempo com o cache comprimido.

Com relação à localidade, em nossa implementação, aplicativos que tenham muitas falhas em páginas que, sem o cache comprimido, estariam armazenadas na memória não-comprimida e, portanto, seriam acessíveis não sofrendo o custo da compressão e descompressão (i.e., acessos à lista custo) e que não tenham benefícios suficientes (i.e., acessos à lista lucro) forçarão o cache comprimido a diminuir de tamanho de modo a se adaptar a um tamanho que não sofra do problema de localidade.

Acreditamos que as restrições de I/O que Douglis mencionou não impedem o cache comprimido de prover melhoras, principalmente porque atualmente temos discos com alta taxa de transferência, mas ainda com altos tempos de acesso.

Por fim, Douglis observou que o cache comprimido se tornará mais interessante se a distância entre a capacidade de processamento da CPU e o tempo de acesso aos discos continuar a crescer. Kaplan [80, 26], em 1999, apontou que a máquina utilizada por Douglis para verificar a sua implementação era muitas vezes mais lenta que as de hoje. Ele também apontou que o crescimento da distância é uma tendência nos próximos anos. Para concluir, Kaplan afirma que o esquema adaptativo proposto e implementado por Douglis pode se adaptar mal para muitos workloads, e propõe um novo esquema, como veremos a seguir.

Também observarmos outros problemas na implementação de Douglis. Na sua implementação, a ordem em que as páginas são armazenadas no cache comprimido não é seguida quando o cache comprimido está cheio e páginas precisam ser liberadas. Isso ocorre pois, quando o seu esquema adaptativo decide que o cache comprimido deve diminuir, a célula mais antiga é liberada, e, com ela, todas as páginas comprimidas que elas armazenavam. Isto não assegura que a ordem LRU das páginas no cache comprimido seja mantida. Isto deve ser fonte de degradação do desempenho do sistema.

A respeito das páginas que o cache comprimido implementado por Douglis armazena, temos que somente páginas armazenáveis no swap são comprimíveis, o que verificamos ter desvantagens para certos tipos de workloads. Apesar desse fato, o esquema adaptativo proposto também leva em conta páginas que não são candidatas a serem comprimidas por ser baseado no algoritmo de competição por memória do Sprite. Para concluir, como Cervera et al., Douglis também implementou suporte para armazenar mais de uma página comprimida em um bloco do swap, efetivamente aumentando o espaço de swap.

7.1.2 Cache Comprimido Estático no Linux 2.0

Em 1999, Raúl Cervera et al. [8] implementaram um cache comprimido no Linux 2.0 como parte de uma implementação de um swap comprimido. Esse cache comprimido possuía tamanho estático, que poderia ter o seu tamanho modificado através de um *driver*, juntamente com outros parâmetros da implementação, quando o swap estivesse desligado.

Em comparação com a nossa implementação, o cache comprimido de Cervera et al. tem algumas limitações. Primeiramente, eles implementaram um cache comprimido de tamanho estático, que é adequado apenas para uma pequena quantidade de aplicativos, como observado por Douglis e Kaplan. Em seus experimentos, por exemplo, foi utilizado um cache comprimido estático de 1 Mb (em um sistema de 64 Mb de memória física total).

O cache comprimido em questão é alocado com células compostas de apenas uma página de memória, por isso pode-se ter problemas devido a baixa compressibilidade observada em alguns aplicativos, e por conseqüência uma célula pode não ser capaz de armazenar mais que uma página comprimida.

Em sua implementação, no caso de não haver mais espaço no cache comprimido, é executada uma operação de limpeza nele. Essa operação consiste em escrever, de modo síncrono, todos os chamados *buffers cheios* para o disco, i.e., todas as células que têm menos espaço livre que o tamanho médio das últimas 100 páginas comprimidas. Dessa maneira, assim como na implementação de Douglis, a ordem em que as páginas são liberadas quando o cache comprimido está cheio não segue a ordem em que elas foram armazenadas nele.

Outra limitação em relação à nossa implementação é o fato do cache comprimido de Cervera et al. armazenar apenas páginas armazenáveis no swap que são marcadas como sujas, logo não comprime páginas armazenáveis em outros dispositivos secundários de armazenamento nem qualquer tipo de página limpa.

Apesar dessas limitações, Cervera et al. relatam ganhos de desempenho para a maioria dos experimentos efetuados. Apenas um dos testes teve queda de desempenho[3], de 4%, ao passo que outro teve um aumento de desempenho muito maior que a média, com ganho de cerca de 85%. Excluindo essas duas exceções, o ganho de desempenho foi entre 16,7% e 50%. A taxa de compressão obtida foi alta, em média 50%. Outro aspecto observado foi o aumento do tamanho do espaço de swap, que é o principal objetivo do projeto, atingindo um "aumento" do tamanho do swap de mais de 50% para a maior parte dos casos.

No entanto, não é claro como separadamente o swap comprimido e o cache comprimido influenciaram esses resultados de desempenho. Citemos um exemplo desse fenômeno. Em alguns experimentos relatados por Cervera et al., a taxa média de compressão era superior a 50%. Dado que as células utilizadas eram compostas por apenas uma página de memória, em média armazenava-se uma página comprimida, o que não aumenta o tamanho efetivo da memória, e conseqüentemente é improvável que ajude a aumentar o desempenho do experimento. Nesse casos, suspeitamos que a compressão do swap seja o provável responsável pela melhora relatada para esses experimentos e não o cache comprimido em si.

7.2 Simulações e Estudos Teóricos

Nessa seção apresentamos os trabalhos sobre cache comprimido que executaram simulações ou que fizeram somente um estudo teórico para verificar o uso da compressão de memória. Em todos os casos, não envolveram implementações, seja em software ou hardware, do cache comprimido. O primeiro trabalho, apresentado na Seção 7.2.1, é a respeito de um modelo matemático de um cache comprimido estático no Windows 95. A seguir, na Seção 7.2.2, os dados de memória são analisados através de um estudo empírico, seguido na Seção 7.2.3 por uma análise de desempenho da compressão de memória. E por fim, na Seção 7.2.4 temos um estudo, parcialmente baseado em

[3]Também utilizamos aqui o nosso conceito de ganho apresentado na Seção 6.4.1 que é a porcentagem de melhora em relação ao tempo gasto pelo sistema sem o cache comprimido.

simulações, da questão da adaptabilidade no cache comprimido, assim como de um algoritmo de compressão especial para dados de processos.

7.2.1 Modelo Matemático de Cache Comprimido no Windows 95

Em 1996, Russinovich e Cogswell [62] analisaram a eficácia do cache comprimido através de um modelo matemático para o cache comprimido. O objetivo foi medir o efeito do cache comprimido do swap cache em aplicativos reais do Windows 95 [81] e, para tal, foram medidas as taxas de leitura e gravação do swap cache do benchmark Ziff-Davis Winstone [83]. Ademais, foi implementado um compressor do swap cache de modo a medir a compressão e descompressão versus os overheads de disco, assim como a taxa de compressão dos dados de paginação.

No modelo desenvolvido, todos os acessos a páginas de memória que estariam na memória sem o cache comprimido são contabilizadas como overhead, devido à compressão e descompressão necessárias para acessá-las, ao passo que acessos a páginas que estão na memória graças à compressão resultam em economia de acessos a disco e são contabilizadas como benefício. Assim, para acessos de leitura e escrita, dado o overhead de compressão, subtraem-se os acessos economizados para obter o efeito no desempenho. Resultados positivos indicam degradação de desempenho e resultados negativos melhora de desempenho.

Não foi possível obter melhoras com esse modelo utilizando o compressor implementado como base (nem com outro produtos comerciais como MagnaRAM 2 e RAM Doubler) para o Winstone em um computador com processador Intel 80486 DX2/66 [62]. De fato, uma piora de 10% foi relatada. Em uma tradução de um trecho do artigo em que esse trabalho é apresentado, temos que "apesar de na teoria parecer possível obter um benefício do cache comprimido do swap cache, para parâmetros realistas é muito difícil."

Na nossa opinião, existem quatro razões para esses resultados insatisfatórios, enumeradas a seguir:

1. A máquina utilizada era mais lenta e tinha um distância entre o poder de processamento da CPU e o tempo de acesso aos discos menor que as máquinas atuais, como já relatado por Kaplan.

2. A taxa de compressão obtida foi de 62,5%. Essa é uma baixa taxa de compressão se comparada com a obtida a partir dos dados dos aplicativos e benchmarks que nós testamos. Se não há nenhum suporte específico para baixa compressibilidade, o desempenho pode cair substancialmente, como nós verificamos para o Open Source Database BenchMark [53] quando o rodamos com um sistema de 24 Mb de memória.

3. Eles relataram uma enorme diferença entre o tempo gasto com compressões e descompressões de páginas (0,05 ms) e o tempo para servir uma falha de página (2 ms, sem o *seek time*[4]). Em contraste a esse relato, essa diferença quase não existe nos nossos experimentos no Linux. Eles atribuem essa diferença a um comportamento ruim do sistema operacional usado nas simulações (Windows 95).

[4]Tempo de procura no dispositivo secundário de armazenamento.

4. Outra possível razão é que o tempo gasto na própria compressão ou descompressão (0,05 ms) parece muito pequeno, como já notado por Kaplan. Apesar da nossa máquina ser muito mais rápida, nós obtivemos um valor perto a esse apenas com o algoritmo de compressão WKdm [85, 26, 80]. Uma vez que não é dito qual algoritmo de compressão usado por eles, esse pode ser o sinal de um algoritmo de compressão demasiadamente rápido mas não muito eficaz.

7.2.2 Estudo Empírico dos Dados de Memória

Nessa seção é apresentado um estudo empírico das características e compressibilidade dos dados de memória feito por Kjelso et al [32] em 1998. Nesse trabalho, eles apresentam resultados de uma investigação representativa e detalhada das características e compressibilidade de 10 aplicativos Unix no sistema operacional SunOS em um ambiente de engenharia. É fundamental notar que, nesse caso, dados de memória são qualquer informação armazenada na memória principal durante a execução de um aplicativo.

Os resultados mais significativos dos dados de memória são citados a seguir: *(i)* grande quantidade de zeros, que ocorrem normalmente em regiões contíguas; *(ii)* inteiros potência de 2 e 255 têm probabilidades significativamente maior que a média; e *(iii)* valores baixos e letras ASCII minúsculas também têm probabilidades maiores.

Para comparação de algoritmos de compressão diferentes, foi implementado um software específico para essa finalidade. O algoritmos comparados foram o algoritmo LZW [76], representante da família de compressores LZ, e o X-RL, que é o compressor de Kjelso et al projetado para pequenos blocos de dados e implementação em hardware de alto desempenho. Pelos seus experimentos, todos os aplicativos atingem taxas de compressão de 55% usando qualquer um dos dois algoritmos, o que é equivalente a um aumento de 80% na capacidade de memória, enquanto a maioria comprime para 50-40% (100-150% de aumento do tamanho da memória). Para dados de memória com todas as seqüências de zero acima de 16 bytes excluídas, temos os seguintes resultados: todos os aplicativos comprimem para pelo menos 65%, enquanto a maioria comprime para 60-50%, equivalente a aumento de 66-100% na capacidade de memória.

Esse trabalho demonstra que os dados de memória apresentam uma compressibilidade que pode ser explorada pelo cache comprimido. Não foi em todos os casos que Kjelso et al. conseguiram uma taxa de compressão de 50% ou menos (ou seja, o aumento de memória não chega aos 100%), logo é importante que o cache comprimido seja projetado levando em conta esse fator. Em nossos experimentos também pudemos verificar essa situação e a nossa implementação aborda essa questão da compressibilidade com as células compostas de duas páginas de memória. Esse estudo de Kjelso et al. também certifica o que Douglis verificou na sua implementação do cache comprimido em relação a baixa compressibilidade de alguns dados. Para concluir, em nossa implementação, pudemos verificar taxas de compressão próximos às relatadas por Kjelso et al.

7.2.3 Avaliação de Desempenho da Compressão da Memória

Em 1999, Kjelso et al. [33] apresentaram um trabalho em que é feita uma avaliação do desempenho de arquiteturas de computadores que possuem compressão de memória. Mais especificamente, nesse trabalho eles descreveram uma arquitetura de memória principal comprimida, que utiliza compressão por software (utilizando o algoritmo de compressão LZRW1 [77]) e hardware (compressor X-Match), e desenvolveram o modelo de desempenho para essa arquitetura, efetuando uma análise para aplicativos com uso intensivo de memória.

O modelo desenvolvido para avaliar o desempenho da compressão da memória calcula o tempo médio de acesso à memória. Esse cálculo utiliza quatro componentes: modelo de hierarquia de memória, características de hierarquia de memória (latências e larguras de bandas entre os níveis da hierarquia de memória), características da compressão de dados (taxa e velocidade de compressão) e comportamento do workload (informação da taxa de *miss*[5] para cada nível da hierarquia da memória).

A investigação de desempenho utiliza os workloads DEC-WRL [58], que são o MULT1, TV, SOR e TREE. As características de memória são as típicas de um sistema daquela época. Da avaliação de desempenho desses workloads, verificou-se que o impacto no desempenho é dependente do aplicativo. Também a paginação pode aumentar o tempo de execução em até uma ordem de magnitude, o que permite que a compressão (em hardware ou software) melhore bastante o desempenho. Taxas de compressão melhores significam que necessidades de memória maiores podem ser suportadas sem necessidade de paginação, aumentando assim o desempenho do sistema. Os resultados de desempenho são encorajadores para a compressão da memória principal, especialmente quando feita em hardware.

Dessa forma, o trabalho de avaliação de desempenho de Kjelso et al. verificou que a compressão dos dados de memória principal pode diminuir os custos de paginação, aumentando o desempenho dos aplicativos. Isso acontece apesar de contar com uma política de adaptabilidade bastante simples, em que o cache comprimido aumenta sempre de tamanho enquanto há necessidade de mais memória que a disponível no sistema. Eles apontam que, se continuar a tendência tecnológica de aumento da diferença entre a velocidade da CPU e o tempo de acesso a disco, a compressão dos dados de memória tornar-se-á cada vez mais interessante.

7.2.4 Cache Comprimido Adaptativo

Scott Kaplan [80, 26], em seus trabalhos sobre cache comprimido de 1999, observa que um importante aspecto do desempenho do cache comprimido é a rapidez dos algoritmos de compressão. Por esse motivo, Kaplan juntamente com Wilson, orientador e co-autor em um dos seus trabalhos, desenvolveram os algoritmos de compressão que exploram particularidades dos dados da memória. Esses algoritmos integram a família WK (Wilson-Kaplan). O dicionário utilizado nesses algoritmos de compressão é pequeno, em relação aos utilizados em algoritmos genéricos de compressão, tendo em vista que páginas de poucos Kb serão comprimidas. Apesar disso, atinge taxas de compressão

[5] A taxa de *miss* corresponde à proporção de acessos que não foram encontrados em determinado nível da hierarquia de memória.

comparáveis e desempenho melhores que os de compressores do estilo Ziv-Lempel para os dados testados. Ademais, a idéia e a implementação são bastante simples.

Outro aspecto importante para que o cache comprimido funcione bem é que ele deve adaptar-se aos diversos conjuntos de aplicativos que estejam rodando. Em alguns momentos, pode ser desnecessário que qualquer página seja comprimida. Em outros momentos, se uma parte das páginas for comprimida, isso permitiria que o conjunto de aplicativos caiba inteiramente na memória. Esse aspecto não foi suficientemente explorado por estudos anteriores a esse, e até mesmo alguns não levaram em conta essa questão de adaptabilidade.

O mecanismo de adaptabilidade apresentado por Kaplan aborda essa questão. Ele faz uma análise de custo/benefício online, baseado nas estatísticas do comportamento do programa. Assumindo que o comportamento no futuro próximo deverá ser parecido com o do passado recente, o mecanismo mantém registro dos aspectos do comportamento do programa que afetam diretamente o desempenho do cache comprimido, efetuando o cálculo de custo/benefício para diferentes tamanhos de cache e assim comprimindo mais ou menos páginas para aumentar o desempenho. Esse sistema utiliza dados como a ordem das páginas na memória, e também um número comparável de páginas recentemente removidas da memória para o swap. Todos esses dados são usados para modelar o que o programa está fazendo.

Para esse mecanismo, são utilizados diferentes tamanhos-alvos de cache comprimido no sistema. A componente adaptativa do sistema calcula os custos e benefícios associados a cada um dos tamanhos, baseados nas contagens recentes de acessos às diversas regiões da ordenação, mesmo que aproximada, da política de substituição de páginas LRU, e por fim escolhe o tamanho com o menor custo. Desse modo, o tamanho é ajustado de acordo com a demanda. A memória é descomprimida somente quando um acesso a uma página comprimida ocorre. Também é somente nesse cenário em que a compressão ocorre se for necessário liberar uma página da memória não-comprimida para armazenar os dados a serem descomprimidos do cache comprimido.

Simulações foram executadas para avaliação dessa idéia de cache adaptativo. Para tal, foram usados seis diferentes programas comuns de sistemas operacionais Unices, três diferentes algoritmos de compressão e quatro tamanhos-alvos do cache comprimido (10%, 23%, 37%, 50% do tamanho de memória da simulação). Os resultados relatados nas simulações foram animadores: todos os algoritmos obtiveram significativas melhoras sobre a memória virtual não comprimida com relação ao tempo gasto para paginação. Durante grande parte da execução dos programas houve melhora de mais de 40% em média, chegando até 80%, e as perdas de desempenho foram raras e pequenas (não mais que 10%). Perdas são minimizadas com a utilização de algoritmos de compressão mais rápidos (a diferença de desempenho do cache comprimido entre os diferentes algoritmos pode ser maior que 15%, segundo as simulações). Sobre a adaptabilidade, notou-se que ela não é perfeita, já que alguns custos são acarretados por tentativas erradas de redimensionamento do cache, mas ela executa bem para a ampla maioria dos programas. Pode-se também observar que há benefício na utilização do cache comprimido para programas com "footprints" de qualquer tamanho: pequenos, médios ou grandes.

Não obstante esses valores animadores de suas simulação, é importante observar que Kaplan apresenta um cache comprimido apenas para páginas armazenáveis no swap. Apesar disso, acreditamos que o esquema adaptativo de Kaplan baseado numa análise de custo/benefício pode ser

estendida de modo a detectar a quantidade de memória que o cache comprimido deve utilizar mesmo se são armazenados todos os tipos de páginas armazenáveis em algum dispositivo de armazenamento. Contudo, é muito difícil coletar os dados necessários para a análise de custo/benefício nos sistemas atuais. Por exemplo, o Linux não mantém uma lista LRU em memória, nem aproximada, de todas as páginas de dados do processo. Não há tal informação e somente quando o sistema está sob pressão de memória é que uma parte aproximada dessa informação é obtida. Ademais, a memória extra necessária para armazenar esses dados no kernel do Linux e conseqüentemente overhead de metadados não foi contabilizado na análise de Kaplan. Isso pode ser desencorajador uma vez que o overhead de metadados mostra ter uma grande influência, principalmente sob pressão de memória. A despeito desses problemas para a análise de custo/benefício de Kaplan, o seu trabalho apresenta uma grande contribuição mostrando que implementações inconclusivas anteriores podem ser corrigidas por uma boa política de adaptabilidade para usufruir da atual distância entre a velocidade de processamento da CPU e o tempo de latência do disco.

7.3 Compressão em Hardware

O nosso cache comprimido está direcionado aos sistemas de hardware padrão, não necessitando de modificações na máquina para ser empregado. Entretanto, nós devemos mencionar que alguns trabalhos desenvolveram e implementaram o cache comprimido em hardware. Kjelso et al [31] desenvolveram e implementaram em hardware um compressor de memória. Usando simulações, eles demonstraram altos ganhos de desempenho. Abali e Franke [1] avaliaram um sistema [4] construído para comprimir todos os dados de memória, focando no aumento do tamanho da memória. No sistema deles, toda a memória é comprimida e um cache em hardware é utilizado para armazenar dados descomprimidos. Cada um dos trabalhos é apresentado abaixo.

7.3.1 Compressor X-Match

Nesse trabalho de 1996, Kjelso et al. [31] apresentam o projeto e a implementação em hardware de um algoritmo de compressão, o X-Match, além de mostrar que a compressão de memória pode dobrar a capacidade de memória para aplicativos comumente usados.

O algoritmo de compressão X-Match comprime os dados de memória (qualquer dado presente na memória), tipicamente para metade do seu tamanho original. Esse algoritmo foi modelado para dados de memória, de modo a trabalhar com um sistema síncrono como a via de dados do computador, além de ter sido desenvolvido para ter uma alta vazão e baixa latência em hardware.

O seu funcionamento consiste na manutenção de um dicionário de dados previamente vistos. Os dados atuais são verificados sobre a possibilidade de casamento com alguma entrada do dicionário, e nesse caso são substituídos por um código relativo à entrada. Quando não há casamento, os dados são transmitidos literalmente, prefixados por um bit. Um casamento parcial ocorre quando pelo menos dois dos caracteres da tupla de entrada casam com uma entrada do dicionário.

Os dados de memória do sistema operacional SunOS e de 8 aplicativos reais e programas utilitários foram utilizados para comparar os algoritmos de compressão Arithmetic [84], Compress [76], X-Match e X-RL, que é basicamente o mesmo que o X-Match, mas com técnicas especiais para

cadeias de zeros. Pelos resultados experimentais, o Arithmetic teve o pior desempenho de todos. Os demais desempenharam similarmente, sendo que o X-RL atingiu a melhor taxa de compressão. As taxas de compressão ficaram na faixa de 40 a 50%, logo a compressão de memória foi capaz de duplicar o tamanho da memória para a maior parte dos workloads.

De modo a verificar o impacto da compressão no desempenho dessas aplicativos, também uma arquitetura de paginação foi simulada. Três programas do DEC-WRL [58] (SOR, TREE e TV) foram testados através do *traces* das suas execuções, e o tempo médio de acesso à memória usando paginação é substancialmente maior que aquele usando compressão. Isso mostra que paginação para disco é cada vez mais cara, com aplicativos rodando uma ordem de magnitude mais devagar do que se tivesse toda a memória necessária disponível. Usando compressão de memória, o tempo de execução seria piorado apenas fracionalmente.

7.3.2 IBM Memory eXpansion Technology (MXT)

No trabalho apresentado a seguir, Abali e Franke [1] avaliaram um novo computador [4] construído pela IBM Research [23] cujos dados da memória principal são armazenados comprimidos. A principal característica envolvida nesse novo computador é que a memória principal não terá um tamanho estático, assim como observado nos computadores usuais, mas um tamanho que será função da taxa de compressão dos dados. Visto que a taxa de compressão não é fixa, pois depende dos dados presentes na memória, o tamanho da memória também não é fixo.

Esse computador com o sistema de memória comprimido é formado por uma memória comprimida, que pode ter até 16 Gb, e um cache de nível 3 (ou L3) de 32 Mb, que armazena os dados descomprimidos. O algoritmo de compressão utilizado é uma variante paralelizada do Ziv-Lempel conhecida como LZ1 [10].

O suporte a esse hardware no Linux também foi apresentado por Abali e Franke nesse trabalho. Com esse hardware, o sistema operacional utilizará a memória enquanto houver memória disponível para ele, o que é dependente da taxa de compressão estática que é definida na BIOS. Como a taxa de compressão pode ser aquém daquela definida na BIOS, é preciso cuidar do caso em que há exaustão de memória e, nesse caso, é necessário diminuir a utilização da memória física. Isso pode ser feito ao se reduzir a utilização da memória real pelo sistema operacional. No kernel do Linux, utilizam-se certas marcas d'água para controlar a utilização da memória física. Essas marcas d'água, que são constantes usadas no kernel do Linux, de modo a se adequar a esse hardware, devem ser mudadas dinamicamente na implementação do suporte à compressão. Assim, quando a utilização da memória física excede os limiares pré-definidos, a marca d'água que indica que há uma quantidade reduzida de memória disponível é aumentada e a kernel thread responsável pela paginação é sinalizada para começar a retirar páginas dos processos e dos caches. Se o uso for muito intenso e não houver tempo para liberar as páginas necessárias para se evitar a exaustão da memória física, algumas kernel threads for criadas especificamente para evitar isso. Essas kernel threads, num momento emergencial, entram em *spinning* e isso causa o travamento de outros processos no sistema uma vez que eles não podem executar até que a utilização da memória caia para níveis considerados normais.

Foram executados testes para medir a taxa de compressão e o impacto na desempenho dos

aplicativos. A taxa média de compressão (tamanho original sobre o comprimido) dos 12 *benchmarks* rodados foi de 43%, sendo a menor de 56% e a maior de 37%. Na questão do impacto no desempenho, a taxa média de penalidade foi de 1,5% para os mesmos 12 benchmarks, sendo o melhor caso de 0,7% e o pior de 3,1%.

7.4 Gerenciamento de Memória no Linux 2.4

Aqui apresentamos uma visão geral do trabalho de Rik van Riel [73] sobre a política de substituição de páginas na versão 2.4 do Linux e sobre os problemas que essa e a versão estável anterior, 2.2, possuíam em relação ao sistema de memória virtual.

Esse trabalho descreve o sistema de gerenciamento de memória do kernel do Linux 2.4 e como essa versão corrigiu diversos problemas presentes na versão estável anterior desse kernel, a versão 2.2.

A versão 2.4 do Linux é fruto de um grande esforço de desenvolvimento. Entre as novidades, destacamos: granularidade maior nos locks de SMP, unificação do buffer cache e do page cache, suporte para sistemas de até 64 Gb de RAM (com processadores compatíveis com Intel x86) e a substituição do código de memória compartilhada SYSV por um sistema de arquivos para memória simples que suporta as semânticas do POSIX SHM e o SYSV SHM.

Quanto às mudanças nas políticas de substituição de páginas do Linux 2.4, temos as seguintes:

- O envelhecimento de páginas reintroduzido no kernel.
- Procedimentos de envelhecimento e flushing de páginas foram separados para evitar a liberação de páginas "erradas" devido a interações entre eles. Eles são, respectivamente, as listas de páginas ativas e inativas.
- O flushing de páginas foi otimizado para evitar muitas interferências do sistema de I/O para escrita no sistema de I/O para leitura em momentos críticos.
- Envelhecimento controlado das páginas em *background* durante os períodos de pouca ou nenhuma atividade do sistema de memória virtual (VM) de modo a manter o VM num estado onde ele poderá facilmente lidar com picos de carga.
- *Stream IO* é detectado e assim nós podemos fazer despejo antecipado (early eviction) das páginas que já foram usadas e recompensar o *stream IO* com mais agressivo read-ahead.

Dados iniciais mostraram que o Linux 2.4 em geral tem melhor desempenho que o Linux 2.2 numa mesma configuração de hardware. Uma grande diferença é que o novo sistema de memória virtual do Linux 2.4 é muito menos suscetível a mudanças sutis do que o Linux 2.2.

Apesar desse trabalho ter sido feito por um dos autores das modificações iniciais acontecidas no Linux 2.4, infelizmente aconteceu uma mudança do sistema de memória virtual na versão 2.4.10-pre11 por ter sido a parte mais crítica nas primeiras versões quando sob pressão de memória.

Capítulo 8

Trabalhos Futuros

Nesse capítulo apresentamos alguns tópicos do projeto que podem ser investigados para estender a pesquisa sobre cache comprimido. Entre eles, apresentamos problemas não abordados devido a limitações da versão do Linux com a qual trabalhamos ou pelo fato do tempo disponível não ter sido suficiente para uma pesquisa mais profunda.

Na Seção 8.1 apresentamos como o cache comprimido pode usufruir de sistemas com multi-processadores e na Seção 8.2 é discutido investigação acerca do tamanho adaptativo de célula. Política de adaptabilidade é discutida na Seção 8.3, enquanto na Seção 8.4 apresentamos a possibilidade de haver uma política de adaptabilidade de algoritmos de compressão. Na Seção 8.5 é apresentada a possível investigação sobre compartilhamento de páginas comprimidas no cache comprimido, enquanto o melhoramento do desempenho do sistema em caso de alta compressibilidade dos dados é mostrado na Seção 8.6. A Seção 8.7 trata da possível redução do overhead quando há baixa pressão de memória e a Seção 8.8 do efeito do cache comprimido em sistemas sem swap, como PDAs. Por fim, a Seção 8.9 sobre compressão antecipada de páginas e a Seção 8.10 aborda a possibilidade de um estudo mais científico sobre a reatividade do sistema.

8.1 Sistemas com multi-processadores

O benefício do cache comprimido baseia-se na distância entre a velocidade de processamento da CPU e o tempo de acesso ao disco. Quanto maior essa distância, mais interessante é possuir compressão de memória para se evitar acessos ao disco. Além disso, a compressão de páginas de memória é ainda mais interessante se o sistema possuir tempo de processamento disponível para ser gasto nas compressões, pois dessa forma não se afeta negativamente os processos ativos pelo fato de utilizar-se tempo de processamento que poderia estar sendo usado por eles para a sua execução.

Dessa forma, o cache comprimido em sistemas com mais de um processador torna-se ainda mais atraente atualmente, pois possuímos mais poder de processamento, além de uma possibilidade muito maior de termos tempo de processamento disponível para a compressão, não impactando diretamente sobre os processos ativos. Ademais, como sistemas de multi-processadores estão cada vez mais comuns hoje em dia, e a tendência é a sua maior popularização, esse é um fator a mais

para que o cache comprimido seja estendido nessa direção.

O Linux oferece suporte a sistemas multi-processados a partir da versão 2.2, logo a implementação atual pode ser incrementada de modo a funcionar estavelmente nesses sistemas. Ela já possui suporte à preempção [57] no kernel do Linux, que na verdade é a mesma infra-estrutura para sistemas multi-processados, porém com uma concorrência em menor intensidade. Dessa forma, a base para a extensão para um suporte estável existe na implementação do cache comprimido para o Linux 2.4.18.

Além do suporte estável, essa implementação deve verificar possíveis gargalos de desempenho que se apresentam em sistemas de multi-processadores, notadamente contenção de *locks*, pois a implementação não foi voltada a esses tipos de sistemas e provavelmente algumas partes deverão ser melhoradas.

8.2 Tamanho Adaptativo de Célula

O tamanho das células na nossa atual implementação do cache comprimido é fixo durante a execução do sistema, e é escolhido através de uma opção de configuração. O tamanho de célula que atingiu, na média, os melhores resultados de desempenho possui duas páginas de memória e faz parte da configuração padrão do cache comprimido. No entanto, esse tamanho não atinge os melhores resultados em todos os casos segundo os nossos experimentos.

Consideramos que a implementação pode ser estendida para que o tamanho da célula seja adaptável durante a execução do sistema para atingir resultados melhores que a implementação atual. A implementação da infra-estrutura de modo que isso seja possível não nos parece ter muitas dificuldades. O maior desafio será a definição da política para decidir qual o tamanho da célula deve ser usado em qual ocasião, e quais parâmetros serão utilizados para essa decisão. De um modo bastante grosseiro, a idéia principal é que, se a compressão das páginas estiver atingindo taxas de compressão ruins, o sistema utiliza páginas de duas páginas de memória, e caso as taxas de compressão sejam boas, começa-se a utilizar células de uma página de memória.

8.3 Adaptabilidade por Processo

A nossa política de adaptabilidade atual é global, ou seja, leva em conta todo o sistema e todas as páginas comprimidas do cache comprimido. A análise é feita baseada nos custos e benefícios para o conjunto de todos os processos, não é feita uma análise do custo/benefício para cada processo. Consideramos grandes as chances de que, caso uma política de adaptabilidade seja feita levando em conta dados por processo, possamos ter uma política de adaptabilidade que ofereça uma relação de custo/benefício ainda melhor ao aumentar os benefícios para aqueles processos que estejam se beneficiando da compressão e ao diminuir os custos daqueles que não obtenham grande benefício do cache comprimido. Isso pode ser útil para workloads compostos de processos diferentes com comportamentos diferentes.

O Linux 2.4.18 não dá suporte para acesso de informação de processos sobre qualquer página de memória despejada. A partir de um processo, é possível verificar a página que ele possui mapeada

devido às tabelas de páginas, mas, a partir de uma página, não dá para se verificar qual processo a está mapeando. A partir da implementação do sistema de memória virtual rmap [60] (reverse mapping), que não é incluído por padrão no Linux 2.4, é possível obter dados do(s) processo(s) mapeando uma determinada página. Esse sistema de memória virtual está incluído na versão de desenvolvimento 2.5.

A partir do benefício que consideramos que seja possível obter com uma política de adaptabilidade a partir de dados de processo, e levando em conta que existe já uma infra-estrutura, embora ainda não-oficial no Linux 2.4, acreditamos que o desenvolvimento do cache comprimido nessa direção pode trazer resultados interessantes.

Para concluir, é importante observar que os dados obtidos através de uma página de memória aplicam-se somente às páginas armazenáveis em swap, e não às demais páginas do page cache. Esse fato torna mais complexa uma política de adaptabilidade por processo.

8.4 Adaptabilidade de algoritmos de compressão

Da mesma forma que uma célula possui o seu tamanho constante ao longo da execução do sistema, o algoritmo de compressão utilizado para comprimir as páginas armazenadas no cache comprimido é definido no momento de inicialização do sistema através de um parâmetro do kernel e não é alterado durante a execução do sistema.

Segundo os resultados dos nossos experimentos, pudemos observar que dependendo da ocasião diferentes algoritmos desempenham de maneiras diferentes. Apesar do LZO ser o algoritmo que consideramos que funcione relativamente melhor para a compressão das páginas para o cache comprimido na maioria dos casos, em certas situações o cache comprimido com o algoritmo WKdm ou o WK4x4 pode desempenhar melhor. Por isso, acreditamos possa ser desenvolvida e implementada uma política que, de acordo com a situação do sistema, seleciona o algoritmo que será utilizado para comprimir as páginas.

Em relação à parte técnica da implementação, o suporte a mais de um algoritmo de compressão já foi implementado, mas removido devido à compressão do swap (pois esse dado deveria ser armazenado nos metadados das páginas gravados no dispositivo de swap). Não há dificuldade técnica em reimplementar esse suporte e possivelmente nem na política de adaptabilidade dos algoritmos de compressão. A parte complexa consiste do desenvolvimento da política de adaptabilidade a ser implementada e testada.

8.5 Compartilhamento de páginas

No seu artigo [75] sobre o gerenciamento dos recursos de memória nos servidores ESX do VMware [74], Carl Waldspurger desenvolve uma técnica para o compartilhamento de páginas de memória baseado no seu conteúdo. No seu caso, a motivação para o compartilhamento das páginas surge do fato de diversas máquinas virtuais rodarem sistemas similares. Além das suas técnicas, que são interessantes, devemos observar os resultados obtidos, que acabam compartilhando uma

grande quantidade de memória. Em particular, nota-se que mesmo com apenas uma máquina virtual, somos capazes de compartilhar um número de páginas.

Acreditamos que, a partir dos dados comprimidos, seja possível utilizar uma técnica parecida para verificar e utilizar o compartilhamento de páginas no cache comprimido. Eventualmente, até mesmo o compartilhamento de páginas no swap. Não sabemos se o custo de tal implementação justificará os seus ganhos, pois não sabemos qual é a quantidade de páginas que podem ser compartilhadas na memória, mas acreditamos que um estudo nessa direção seja interessante.

8.6 Melhoramento do caso de alta compressibilidade

Em nossa implementação, tendo em vista que as taxas de compressão obtidas com os nossos experimentos, focamos em desenvolver o cache comprimido com um bom desempenho para dados com as taxas de compressão próximas às obtidas. No entanto, um dos nossos experimentos atingiu taxa de compressão de 1% do tamanho original, caso em que não focamos durante o desenvolvimento. Esse tipo de caso, por armazenar um grande número de páginas comprimidas, pode ter alguns gargalos nas estruturas de dados que não prevemos e, portanto, ter o seu desempenho abaixo do possível com a utilização do cache comprimido.

Consideramos que um melhoramento da implementação para casos que há alta compressibilidade devem ser tentados, possuindo grandes probabilidades de um aumento de desempenho nessas situações. Deve-se certificar-se que não há prejuízo para o desempenho do cache comprimido com páginas cujas taxas de compressão são maiores.

8.7 Redução de overhead sob baixa pressão

O cache comprimido é capaz de prover benefícios para sistemas quando estão sob pressão de memória. No entanto, dependendo do sistema, uma parte considerável do tempo de execução é despendida executando programas em situações de muito pouca pressão de memória, e nesse caso o cache comprimido introduz um pequeno overhead, conforme verificamos com nos resultados da parte experimental.

Melhorias na direção de uma redução de overhead sob baixa pressão de memória não foram tentadas nessa implementação. Acreditamos que elas são possíveis e devem ser tentadas no futuro.

8.8 Efeito em sistemas sem swap

Sistemas sem swap são comumente lembrados quando fala-se sobre cache comprimido. Na verdade, isso ocorre não pela melhora de desempenho que pode ser provida pelo cache comprimido, mas pelo aumento efetivo do tamanho da memória para esses sistemas. Personal Digital Assistents, ou PDAs, são exemplos de sistemas sem swap, possuem em geral pouca memória, e o aumento da sua capacidade é muito bem-vindo para permitir a execução e armazenamento de mais programas.

O sistema de endereçamento virtual de swap (Seção 5.1) aborda o problema que teríamos para executar em um sistema sem swap. Para o seu efetivo funcionamento num PDA, é necessário portar o código-fonte para a sua plataforma, além de efetuar alguns ajustes eventuais no sistema de endereçamento virtual.

Acreditamos que o estudo do efeito do cache comprimido nesse tipo de sistema seja possível e possa trazer resultados interessantes.

8.9 Compressão antecipada

De modo a utilizar de melhor maneira o tempo de processamento da CPU, poderíamos efetuar a compressão de páginas antecipadamente enquanto a CPU estivesse sem processamento (*idle time*). Dessa maneira, quando houvesse pressão de memória, as páginas já estariam comprimidas e o seu impacto seria minimizado.

Acreditamos que esse enfoque precise ser amadurecido e deve requerer um bom estudo para o seu desenvolvimento. Ele também pode envolver grandes alterações no sistema de memória virtual do Linux. No entanto, achamos que a idéia pode ser promissora e o seu desenvolvimento deve ser tentado.

8.10 Estudo de Reatividade

Durante o desenvolvimento do cache comprimido, devido ao fato do código estar público e fazer parte de conjuntos de patches para o Linux, ele foi utilizado por diversos usuários. Alguns desses usuários nos relataram as suas sensações em relação ao desempenho do sistema quando o cache comprimido era utilizado. Além do relato geral de que era notada uma melhora no desempenho do sistema, foi nos relatado que o tempo de reatividade (*responsiveness*) do sistema na interatividade de um sistema desktop possuía uma grande melhora quando o cache comprimido era utilizado.

Fizemos algumas tentativas infrutíferas de um estudo metódico e científico do tempo de reatividade do sistema na interação de um usuário. Tentamos utilizar programas para gravar as atividades de um usuário na interação com um sistema desktop, mas não fomos bem-sucedidos pois esses programas não funcionavam corretamente nos nossos testes.

Acreditamos que um método científico de estudo de reatividade deva ser tentado para verificar a influência que o cache comprimido em sistemas em que a resposta do sistema tem uma grande influência. Acima de tudo, desenvolver um método desses poderá ser utilizado em outros aplicativos para essa mensuração.

Capítulo 9

Principais Contribuições

Nesse capítulo apresentamos, de modo suscinto, as principais contribuições desse trabalho de um modo geral. Aqui desejamos mostrar no que esse trabalho destaca-se de trabalhos anteriores, e quais as principais inovações que foram apresentadas ao longo do texto ou que fizeram parte durante o seu desenvolvimento.

Armazenamento de outros tipos de páginas

Esse é o primeiro trabalho com projeto e implementação armazenando outros tipos de páginas, como páginas do cache de arquivos. Conforme dito anteriormente, o armazenamento dessas outras páginas auxiliou a diminuir o impacto sobre as páginas do cache de arquivos que se tinha quando comprimia-se apenas páginas armazenáveis no swap. A grande maioria dos workloads que encontramos é composto por uma substancial quantidade de páginas do cache de arquivos, quando não a quase totalidade. Para esses casos, o cache comprimido, antes de mais nada, deixou de se tornar um problema e, além disso, possibilitou inclusive ganhos de desempenho.

Comparação de algoritmos de compressão

Nesse trabalho, efetuamos uma comparação do desempenho de diferentes algoritmos de compressão na mesma implementação do cache comprimido, comparando os algoritmos WKdm, WK4x4 e LZO. Em trabalhos diferentes, eles foram utilizados para verificar a validade do cache comprimido, seja através de implementação ou de simulaçãos. Apesar de Scott Kaplan ter comparado mais de um algoritmo de compressão em suas simulações, ainda não se tinha verificado qual era a real diferença entre esses algoritmos e o seu impacto na prática partindo-se de uma implementação. Em seu trabalho, Kaplan nos levou a crer que a rapidez do algoritmo é mais importante que a sua taxa de compressão. Por outro lado, as nossas experiências, levou-nos a conclusões diferentes, por isso a escolha por um algoritmo que demora mais para comprimir, mas que possui melhor taxa de compressão. Além disso, também fizemos experimentos com mais de algoritmo de compressão ao

mesmo tempo (WKdm para páginas armazenáveis no swap e LZO para as demais páginas), sem resultados conclusivos no momento, mas acreditamos que pode ser alvo de uma maior pesquisa no futuro.

Política de adaptabilidade simples e eficaz

Implementamos um política de adaptabilidade que, dentro do nossa infra-estrutura de cache comprimido implementada, teve um desempenho bastante eficaz, além de ser uma política bastante simples. Essa política não necessita do armazenamento de grandes quantidades de dados de modo a conseguir decidir qual atitude tomar, e os resultados provenientes das suas decisões demonstram que um cache comprimido adaptativo pode ser bastante útil ao sistema.

Dados de pouca compressibilidade

Esse trabalho aborda em seu projeto dados de pouca compressibilidade, diferentemente de estudos e implementações anteriores, onde essa questão não foi abordada. Além disso, foi colocada como sendo uma das responsáveis pelo mau desempenho do cache comprimido, porém sem ter sido, de alguma forma, tratada. Esse problema foi identificado nos nossos experimentos e tratado através da implementação e experimentações de células com um número de páginas contíguas.

Política de suspensão de compressão

Tendo em vista o nosso estudo aprofundado dos trabalhos anteriores sobre o assunto do cache comprimido, esse é o primeiro projeto que estuda a ineficiência do cache comprimido para certos casos, exibe experimentos claros em que isso acontece e sugere, verificando através de implementação, uma solução: suspensão da compressão de páginas. No nosso caso, as páginas limpas podem ter a compressão suspensa, sendo liberadas sem sofrerem o processo de compressão e armazenamento no cache comprimido.

Análise de trabalhos anteriores

Fazemos uma análise detalhada e comparativa de trabalhos anteriores relacionados ao nosso assunto. Essa análise inclui todos os trabalhos de implementação em software do cache comprimido e de simulações e estudos téoricos sobre cache comprimido, compressão de memória e compressibilidade dos dados. Por fim, fazemos uma breve análise de trabalhos de compressão em hardware e sobre o gerenciamento de memória do Linux 2.4.

Estudo de efeitos diretos e indiretos

Apresentamos um estudo dos efeitos direitos e indiretos do cache comprimido no sistema. Dessa maneira, verificamos o efeito do cache comprimido em outras partes do sistema, como outros caches de memória, partes dos sistemas de arquivos (buffers, por exemplo) e o escalonamento de processos, entre outros. Aspectos negativos dessas influências foram relatados e abordados, ao passo que aspectos que não são considerados negativos têm uma descrição do seu comportamento e a razão dele.

Situações de pouca ou nenhuma pressão de memória

Nesse trabalho, fazemos uma análise de desempenho em situações de pouca ou nenhuma pressão de memória, inédito em trabalhos desse assunto. Essa análise e experimentação acerca desses cenários são importantes para a demonstração de que o cache comprimido pode ser adotado em sistemas operacionais vigentes, devido ao fato de não possuir overhead nas situações consideradas mais comuns, que são as de pouca ou nenhuma pressão de memória.

Experiência com a comunidade de software livre

Pudemos ter experiência com a comunidade de software livre e com uma interessante base de usuários, o que permitiu verificar o real interesse pelo cache comprimido, especialmente se estiver disponível uma implementação bem-feita e com bons resultados. Durante o desenvolvimento do projeto, o código-fonte esteve sempre disponível para ser utilizado por usuários e contamos com listas de discussões de email abertas ao públicos.

Variedade de trabalhos futuros

O desenvolvimento desse trabalho permitiu-nos verificar a vastidão de pesquisa que esse assunto nos possibilita. Diversas possibilidades de extensão da idéia do cache comprimido e da sua implementação são listadas nos Capítulo 9.

Capítulo 10

Conclusões

Nós propomos uma política de adaptabilidade nova e simples para um sistema de cache comprimido, implementamos esse sistema, e obtivemos aumentos de desempenho significantes para todos os workloads testados sob pressão de memória (até 171,4%) com overhead desprezível para baixa pressão de memória (até 0,39%). Ademais, quase nenhum overhead foi detectado quando nenhuma pressão de memória existia. Esse sistema de cache comprimido é o primeiro a abordar workloads com baixa compressibilidade e também a comprimir páginas não armazenáveis no swap. Essas características mostram ser fundamentais para a melhora de desempenho em alguns workloads que não poderia se beneficiar do uso do cache comprimido caso contrário de outra forma.

Também verificamos que o overhead introduzido pela manutenção de metadados pode ter um forte efeito negativo no desempenho do sistema, especialmente sobre pressão de memória. Considerando que um sistema de cache comprimido tente obter melhoras justamente nesse cenário, observamos que os nossos algoritmos mais simples que requerem menos metadados tendem a obter resultados melhores.

Melhoramentos na direção de uma redução do overhead sob muito baixa pressão de memória não foram tentados. Eles são possíveis e devem ser tentados no futuro.

Essa implementação deve também ser estendida para fazer uso de arquiteturas com multiprocessadores. O Linux 2.4.18 não permite que se descubra de forma imediata a que processo pertence uma página despejada da memória. A partir desse tipo de informação, poderia-se tentar fazer uma análise adaptativa que poderia também levar em conta a informação por processo em conta. Isso poderia ajudar para workloads compostos por processos diferentes com comportamentos diferentes.

Muitos usuários do nosso sistema de cache comprimido relataram melhor reatividade para workloads típicos de um sistema desktop. Eles relataram interação com o sistema mais suave, que não pode ser avaliada objetivamente ainda, mas que é um forte indicativo dos benefícios do cache comprimido adaptativo para workloads comuns de um desktop. Nós acreditamos que bons métodos para testar reatividade para workloads de desktops devem ser desenvolvidos para comprovar (ou não) cientificamente esses relatos muito positivos (mas ainda subjetivos). Considerando que os workloads de desktop são os mais importantes workloads para a maioria dos usuários, podemos ver a significância desses relatos e quão importante seria uma confirmação científica dos mesmos.

Acreditamos também que o sistema de cache comprimido pode ser extremamente útil para

estender efetivamente a memória em dispositivos sem swap, como muito PDA's que rodam Linux. Um método de comparação científico para essa aplicação deve ser desenvolvido e utilizado.

Pode-se argumentar que memória está ficando barata e que, por isso, o cache comprimido não é necessário. Baseado nos nossos experimentos, afirmamos que os dois juntos - mais memória e o cache comprimido adaptativo - são ainda melhores. Nossos experimentos com aplicativos para os quais a memória instalada não era suficiente mostrou que o desempenho obtido pode economizar o gasto extra ao se adiar expansões de memória.

Resumindo, o objetivo principal da implementação atual foi atingido com o aumento de desempenho verificado em todos os workloads sob pressão de memória. Por essas razões, acreditamos que um cache comprimido adaptativo poderia ser adotado nos sistema operacionais de propósito geral atuais como um mecanismo para uma melhoria considerável no desempenho do sistema.

Referências Bibliográficas

[1] B. Abali and H. Franke. Operating system support for fast hardware compression of main memory contents. In *Memory Wall Workshop of the 27th Annual International Symposium on Computer Architecture (ISCA-2000)*, Vancouver, BC, Canada, 2000.

[2] Welcome! - The Apache Software Foundation. <http://www.apache.org>.

[3] A. W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, USA, 1991.

[4] S. Arramreddy, D. Har, K.-K. Mak, T. B. Smith, R. B. Tremaine, and M. Wazlowski. Ibm memory expansion technology (mxt) debuts in a serverworks northbridge. In *Hot Chips 12 Symposium*, 2000.

[5] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer Technical Conference*, 1994. <http://www.usenix.org/publications/library/-proceedings/bos94/full_papers/bonwick.ps>.

[6] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, October 2000.

[7] BSD. URL: <http://www.bsd.org>.

[8] R. Cervera, T. Cortes, and Y. Becerra. Improving Application Performance through Swap Compression. In *Usenix '99 – Freenix Refereed Track*, 1999.

[9] Computer Design. <http://www.ecr.mu.oz.au/~malam/comp_sci/comp_des/comp_des.-html#mem>.

[10] D. J. Craft. A fast hardware data compression algorithm and some algorithmic extensions. *IBM Journal of Research and Development*, 42(6), 1998.

[11] M. D. Dahlin. Technology Trends. <http://www.cs.utexas.edu/users/dahlin/tech-Trends/data/memPrices/plot.ps>.

[12] M. D. Dahlin. Technology Trends. <http://www.cs.utexas.edu/users/dahlin/tech-Trends/data/diskPrices/disk.ps>.

[13] M. D. Dahlin. The impact of trends in technology on file system design. Technical report, University of California, Berkeley, 1996. <http://www.cs.utexas.edu/users/dahlin/-techTrends/trends.ps>.

[14] OSDL Database Test 1. URL: <http://www.osdl.org/projects/performance/dbt1--page.html>.

[15] Debian GNU/Linux – The Universal Operating System. <http://www.debian.org>.

[16] Debian GNU/Linux – debian sarge Release Information. <http://www.debian.org/-releases/sarge>.

[17] Debian GNU/Linux 2.2 (potato) Release Information. <http://www.debian.org/releases/-potato>.

[18] F. Douglis. The Compression Cache: Using On-line Compression to Extend Physical Memory. In *Winter 1993 USENIX Conference*, pages 519–529, 1993.

[19] FreeBSD. URL: <http://www.freebsd.org>.

[20] Genoma-FAPESP. <http://watson.fapesp.br/onsa/Genoma3.htm>.

[21] The GIMP Homepage. <http://www.gimp.org/>.

[22] GNU General Public License. <http://www.gnu.org/copyleft/gpl.html>.

[23] IBM Research. <http://www.research.ibm.com/>.

[24] The Internet Operating System Counter. <http://leb.net/hzo/ioscount/>.

[25] D. Johnson and D. Dellanave. piGIMP. <http://pigimp.sourceforge.net/>.

[26] S. F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation.* PhD thesis, University of Texas at Austin, 1999.

[27] KDE - the K Desktop Environment. <http://www.kde.org>.

[28] The Linux Kernel Archives. <http://www.kernel.org>.

[29] The linux-kernel mailing list FAQ. <http://www.kernel.org/pub/linux/docs/lkml/>.

[30] Linux Kernel Historic. URL: <http://www.kernel.org/pub/linux/kernel/Historic/>.

[31] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compression. In I. C. S. Press, editor, *22nd Euromicro Conference*, pages 423 – 430, September 1996.

[32] M. Kjelso, M. Gooch, and S. Jones. Empirical study of memory data. In IEE, editor, *IEE Proceedings Comput. Digit. Tech.*, volume 145, pages 63 – 67, January 1998.

[33] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression. *Journal of Systems Architecture*, 45:571 – 590, 1999.

[34] C. Kolivas. The homepage of contest, The linux kernel responsiveness benchmark. URL: <http://contest.kolivas.net>.

[35] Compressed Caching. <http://linuxcompressed.sourceforge.net/>.

[36] linux-history: Once upon a time... URL: <http://www2.educ.umu.se/~bjorn/linux/misc/-linux-history.html>.

[37] Ports of Linux OS. URL: <http://perso.wanadoo.es/xose/linux/linux_ports.html>.

[38] Linux Kernel Documentation Project. URL: <http://www.lkdp.tk/>.

[39] Linux Support for NUMA Hardware. URL: <http://lse.sourceforge.net/numa/>.

[40] Markus F.X.J. Oberhumer: LZO data compression library. <http://www.oberhumer.com/-opensource/lzo/>.

[41] The Mach Project Home Page. URL: <http://www-2.cs.cmu.edu/afs/cs/project/mach/-public/www/mach.html>.

[42] GNU Make. URL: <http://www.gnu.org/software/make/make.html>.

[43] The MathWorks. <http://www.mathworks.com/>.

[44] The MathWorks - MATLAB. <http://www.mathworks.com/products/matlab/>.

[45] Memtest Suite. <http://carpanta.dc.fi.udc.es/~quintela/memtest/>.

[46] Microsoft Corporation. URL: <http://www.microsoft.com/>.

[47] D. Mosberger and T. Jin. httperf A Tool for Measuring Web Server Performance. <http://www.hpl.hp.com/personal/David_Mosberger/httperf.html>.

[48] The MUMmer Home Page. <http://www.tigr.org/software/mummer/>.

[49] MySQL. URL: <http://www.mysql.com/>.

[50] NetBench 7.0.2. <http://www.etestinglabs.com/benchmarks/netbench/netbench.asp>.

[51] M. F. X. J. Oberhumer. *LZO — real-time data compression library*, 2000. <http://wildsau.-idv.uni-linz.ac.at/mfx/lzo.html>.

[52] Linux online - Ongoing Projects. URL: <http://www.linux.org/projects/ports.html>.

[53] The Open Source Database Benchmark. <http://osdb.sourceforge.net/>.

[54] D. A. Patterson, J. L. Hennessy, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2 edition, 1996.

[55] PostgreSQL. URL: <http://www.postgresql.org/>.

[56] PostMark: A New File System Benchmark. <http://www.netapp.com/tech_library/3022.-html>.

[57] Linux Kernel Patches — rml. URL: <http://www.tech9.net/rml/linux/>.

[58] K. J. Richardson and M. J. Flynn. TIME: Tools for Input/Output and Memory Evaluation. In *Proceedings of the Twenty-Fifth International Hawaii Systems Science Conference*, pages 58–66, January 1992.

[59] L. Rizzo. A very fast algorithm for ram compression. In *Operating Systems Review*, pages 36–45, 1997.

[60] Rik van riel's linux kernel patches. <http://www.surriel.com/patches/>.

[61] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., 2 edition, 2001.

[62] M. Russinovich and B. Cogswell. RAM Compression Analysis. Technical report, O'Reilly, 1996.

[63] SAP DB. URL: <http://www.sapdb.org>.

[64] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*. John Wiley, 5 edition, 1997.

[65] Solaris Operating System (SPARC & x86). URL: <http://wwws.sun.com/software/-solaris/>.

[66] Sprite. URL: <http://www.cs.berkeley.edu/projects/sprite/sprite.html>.

[67] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[68] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems, Design and Implementation*. Prentice Hall, 2 edition, 1997.

[69] GNU Textutils 2.0 source code. <ftp://ftp.gnu.org/gnu/textutils/textutils-2.0.-tar.gz>.

[70] TPCW. URL: <http://www.tpc.org/tpcw>.

[71] A. Tridgell. dbench filesystem benchmark. <ftp://samba.org/pub/tridge/dbench/>.

[72] U. Vahalia. *UNIX Internals: The new Frontiers*. Prentice Hall, 1995.

[73] R. van Riel. Page Replacement in the Linux 2.4 VM. In *Usenix 2001's Freenix Track*, 2001. <http://www.surriel.com/lectures/linux24-vm-freenix01.pdf>.

[74] Enterprise Class Virtual Machine Software. <http://www.vmware.com/>.

[75] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI' 02)*, December 2002.

[76] T. A. Welch. A Technique for High Performance Data Compression. *IEEE Computer*, 17(6):8–19, 1984.

[77] R. N. Willians. An extremely fast ziv-lempel compression algorithm. In *Data Compression Conference*, pages 362–371, 1991.

[78] P. R. Wilson. Some Issues and Strategies in Heap Management and Memory Hierarchies. In *OOPSLA/ECOOP Workshop on Garbage Collection in Object-Oriented Systems*, 1990.

[79] P. R. Wilson. Operating System for Small Objects. In *Internation Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, CA, USA, 1991. IEEE Press.

[80] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Summer 1999 USENIX Conference*, pages 101–116, Monterey, CA, USA, 1999.

[81] Windows 95. URL: <http://www.microsoft.com/windows95/>.

[82] Windows NT Home. URL: <http://www.microsoft.com/ntserver/ProductInfo/default.asp>.

[83] Business Winstone. <http://www.etestinglabs.com/benchmarks/bwinstone/bwinstone.-asp>.

[84] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30:520–540, 1987.

[85] WKdm source code. <http://www.cs.utexas.edu/users/oops/compressed-caching/-WKdm.tgz>.

[86] XFree86. <http://www.xfree86.org>.

[87] X.Org. <http://www.x.org>.

[88] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[89] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.