Universidade de São Paulo

Instituto de Matemática e Estatística
Departamento de Ciência da Computação

MAC 5701
Tópicos em Ciência da Computação

Relatório de Estudo
Gerenciamento de Memória

Área de Concentração: Sistemas Operacionais

Aluno
Rodrigo Souza de Castro
Orientador
Prof. Dr. Alair Pereira do Lago

Sumário

Resumo

1 Plano de Estudos

O plano de estudos montado para a disciplina *MAC 5701 - Tópicos em Ciência da Computação* foi baseado em artigos científicos e uma tese de PhD diretamente relacionada ao projeto de pesquisa do mestrado do aluno. A escolha da maioria dos artigos foi feita baseado numa breve leitura preliminar feita pelo aluno durante a confecção do seu projeto de pesquisa para a FAPESP.

As leituras foram divididas em algumas seções para melhor organização. São elas: *Cache Comprimido*, *Linux - Sistema de Gerenciamento de Memória*, *NetBSD - Sistema de Memória Virtual*, *MachOS* e *Tópicos Variados de VM*.

A primeira seção (*Cache Comprimido*) visa aprimorar o conhecimento do aluno em relação ao assunto principal do seu projeto. Nessa seção estão listados todos os artigos relevantes ao assunto, incluindo o artigo do Scott F. Kaplan juntamente com a sua tese de PhD que aprofunda bastante o assunto. Tanto o artigo como a sua tese de Kaplan são a base principal do projeto de mestrado do aluno em questão. Um artigo foi adicionado ao plano de estudos montado originalmente. O seu título é *Operating System Support for Fast Hardware Compression of Main Memory Contents*, escrito por uma equipe de pesquisa da IBM.

Na segunda seção (*Linux - Sistema de Gerenciamento de Memória*) encontrar-se-á um artigo apresentado na Freenix 2001 por Rik van Riel, um dos principais responsáveis pelo gerenciamento de memória do kernel do Linux, sobre a política de substituição de páginas na versão 2.4 do kernel do Linux. Diante do fato de que o projeto de mestrado do aluno terá uma implementação no kernel do Linux, atualmente na sua versão 2.4, é importante estar a par do único artigo especificamente sobre ele.

Uma visão global sobre as políticas e implementações de gerenciamento de memória existentes em outros sistemas além do Linux é altamente desejada para um aprimoramento do conhecimento do aluno. Dessa forma, as terceira (*NetBSD - Sistema de Memória Virtual*) e quarta (*MachOS*) seções são focadas em papers disponíveis sobre o sistema de memória virtual utilizado nos sistemas operacionais NetBSD e MachOS. São bastante interesssantes visto que o primeiro é bastante conhecido por ter um gerenciamento de memória reconhecidamente estável e o segundo por ser um inovador em algumas técnicas no gerenciamento de memória para sistemas operacionais. Muitas das idéias presentes em sistemas operacionais como Linux vieram do MachOS, que foi desenvolvido pela Carnegie Mellon University.

Para completar, a última seção (*Tópicos Variados de VM*) trata de dois papers que visam enriquecer o conhecimento do aluno sobre aspectos diversos do gerenciamento de memória como *QoS* (Quality of System) explicado no primeiro paper, e o suporte a páginas de memória de tamanho múltiplos (detalhado no segundo).

Com essas cinco seções, cremos que o conhecimento do aluno tenha tido uma ótima oportunidade de ser bastante melhorado, provendo uma base mais sólida e um amadurecimento que aumentarão as chances de uma finalização do seu projeto de mestrado com um maior sucesso.

Esse relatório de estudos é composto por resumos desenvolvidos ao longo de todo o semestre letivo depois da leitura de cada artigo ou da tese de doutorado que forma o plano de estudos.

Também há uma pequena análise do paper depois de cada resumo, principalmente na questão da qualidade de escrita do paper e a sua relação com o trabalho sendo desenvolvido pelo aluno. Antes do resumo, é exibida uma pequena tabela com alguns dados importantes de cada artigo, como os autores, conferência em cujo anal foi publicado (se foi o caso), a universidade ou empresa à qual o autor é vinculado e assim por diante.

2 Projeto de Pesquisa - Mestrado

Todo o estudo para essa disciplina foi feito baseado no projeto de pesquisa atualmente sendo desenvolvido pelo aluno. Dessa forma, é bastante interessante prover uma visão geral do que o projeto se trata para que o leitor entenda a motivação para esses papers.

Em um sistema operacional, o tempo de acesso à memória depende do tempo de acesso à memória principal, aos diversos níveis de cache e à área de swap em disco. Latências de disco são duas ordens de magnitude maiores que as latências de acesso à memória, enquanto outros níveis adjacentes na hierarquia de memória diferem por menos que uma ordem de magnitude. Dessa forma, o tempo de acesso ao disco é uma das maiores causas na degradação do tempo de acesso à memória. Um jeito de melhorar o desempenho do sistema de memória virtual é diminuir o número de acessos ao disco. Isso pode ser feito inserindo um novo nível de cache entre a memória principal e o disco de swap na hierarquia de memória.

Cache Comprimido é um dos métodos propostos para diminuir o número de acessos ao swap, de modo a se tentar o tempo de acesso médio às páginas de memória. Esse método é composto de se dividir a memória principal em duas partes: cache não-comprimido e cache comprimido, seguindo a nomenclatura escolhida por Scott Kaplan nos seus estudos sobre o assunto. O primeiro cache armazena páginas no seu formato natural, e as páginas são acessadas pela CPU através do sistema de gerenciamento de memória. No outro cache, páginas são armazenadas na forma comprimida.

A idéia de cache comprimido já foi proposta no passado e até algumas implementações foram feitas, mas nenhum estudo demonstrou o benefício do uso do cache comprimido. Na sua tese de doutorado, Kaplan mostrou, usando simulações, que um cache comprimido de tamanho adaptativo pode prover redução de até 80% nos custos de paginação, enquanto raramente aumenta os custos de paginação, nunca por mais que 10%.

Nenhuma implementação desse cache comprimido de tamanho adaptativo foi feita num sistema operacional. Um dos maiores objetivos da pesquisa do aluno é exatamente implementar essa idéia no sistema operacional Linux. Desde 2000, na iniciação científica, ele vem trabalhando no Linux 2.2 visando implementar essa idéia de cache comprimido, mas infelizmente não foi possível terminar durante a graduação.

No começo desse ano, a implementação foi reiniciada do zero, mas dessa vez em cima do Linux 2.4. O projeto tem uma página[1] e uma seção de downloads[2] que demonstra que o projeto é atraente para as pessoas em geral (23 mil visitações e as nossas modificações para o kernel foram baixadas da página 1700 vezes desde abril de 2001).

[1] Acesse `http://linuxcompressed.sourceforge.net`.
[2] Visite `http://sourceforge.net/project/stats/?group_id=13472`.

Na próxima fase do projeto, a adaptabilidade, o aluno estudará com mais detalhes o algoritmo adaptivo do Kaplan. Com algumas idéias já discutidas na lista de discussão do projeto e disponíveis na nossa página de design, tentar-se-á desenvolver um modelo de implementação que possa ser adaptado ao sistema de memória virtual do Linux. Depois essa idéia será implementada, sempre tentando-se minimizar o *overhaead* introduzido no sistema original. E após este estágio, serão feitos diversos testes para provar a sua estabilidade e para efetuar o ajuste fino da implementação. Finalmente serão rodados testes e benchmarks adicionais para comparar o seu desempenho com outros sistemas (sem compressão).

3 Cache Comprimido

3.1 The Case for Compressed Caching in Virtual Memory Systems

Esse paper foi apresentado na conferência técnica da USENIX em 1999, tenho ganhado o prêmio de melhor paper na conferência [3]. Com grande qualidade técnica e muito bem escrito, esse paper foi a motivação para o início do nosso trabalho em cima dessa idéia na iniciação científica em 1999, trabalho que evoluiu para o mestrado atual. Aqui uma visão mais geral da idéia de Cache Comprimido nos é apresentada e serve como um primeiro contato com a idéia antes de ler a tese de doutorado comentada no próximo item.

Tipo de Documento	Artigo (Paper em Conferência)
Autores	Paul R. Wilson, Scott F. Kaplan e Yannis Smaragdakis
Conferência	1999 USENIX Annual Technical Conference
Data	Junho de 1999
`http://www.usenix.org/publications/library/proceedings/usenix99/full_papers/wilson/wilson.ps`	

Resumo do Paper Cache Comprimido é a introdução de um novo nível no sistema de memória virtual. Esse cache utiliza uma parte da memória RAM para armazenar páginas que são comprimidas por algoritmos especiais de compressão. Isso visa aumentar o tamanho efetivo da memória e assim diminuir a paginação em disco, visto que esse tem o tempo de acesso mais caro no sistema de memória virtual.

Estudos anteriores a respeito não conseguiram demonstrar que o seu uso traria vantagens consistentes. Inicialmente uma das razões para isso é que as máquinas eram muito lentas para os padrões atuais e assim a diferença de desempenho entre a memória RAM e o disco não era tão grande. No entanto, o sistema de Cache Comprimido pode nos dar uma melhora de desempenho para as máquinas atuais muito maior que antigamente, e a melhora só deverá aumentar à medida que os processadores ficam mais rápidos.

Outro ponto importante da melhora no desempenho vem do uso de melhores algoritmos de compressão. Por esse motivo foram desenvolvidos algoritmos da família WK (Wilson Kaplan) para explorar particularidades dos dados da memória. O dicionário utilizado na compressão é pequeno, visto que iremos comprimir páginas de poucos kilobytes, e mesmo assim atinge taxas de compressão

[3] Confira `http://www.usenix.org/events/usenix01/cfp/papers.html`.

comparáveis as de compressores do estilo Ziv-Lempel. Além disso, a idéia e a implementação são bastante simples.

Um aspecto bastante 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 alguma 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 suficiente explorado por estudos anteriores, e até mesmo alguns não levaram em conta essa questão de adaptabilidade.

O mecanismo de adaptabilidade apresentado no artigo cuida dessa questão fazendo 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 passado recente, o mecanismo mantém registro dos aspectos do comportamento do programa que afetam diretamente o desempenho do Cache Comprimido para diferentes tamanhos de cache e comprime mais ou menos páginas para aumentar o desempenho. Esse sistema utiliza dados mantidos atualmente pelos SOs, como a ordem das páginas na memória, e também um número comparável de páginas "recently-evicted" (recentemente removidas da memória para o *swap*). Isso é usado para modelar o que o programa está fazendo.

Como dito, são usados diferentes tamanhos-alvos de Cache Comprimido. O componente adaptativo do sistema calcula os custos e benefícios de cada um dos tamanhos, baseados nas contagens recentes de acessos às regiões das ordem LRU, e escolhe o tamanho com o menor custo. Desse modo o tamanho é ajustado de acordo com a demanda. Memória é comprimida ou descomprimida somente quando um acesso a uma página comprimida ocorre.

Simulações foram executadas para essa idéia de cache adaptativo. Para tal, foram usados seis diferentes programas comuns do UNIX, três diferentes algoritmos de compressão e quatro tamanhos do Cache Comprimido (10%, 23%, 37%, 50% do tamanho de memória da simulação). Os resultados foram bastante animadores. Todos os algoritmos obtiveram significativas melhoras sobre a memória virtual não comprimida. Durantes grandes partes da execução do programa houve melhora de mais de 40% em média, chegando até 80%, e as perdas de desempenho foram raras e pequenas. Perdas são minimizadas com a utilização de algoritmos de compressão mais rápidos (a diferença de desempenho do Cache Comprimido para 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 redimensionar o cache, mas ele executa muito bem para a ampla maioria dos programas. Ainda mais é capaz de demonstrar que há benefício na utilização do Cache Comprimido para programas com footprints pequenos, médios e grandes.

Análise do Paper Esse paper é muito bem escrito e transmite com muita clareza a idéia proposta por Kaplan para a diminuição dos acessos a disco no sistema de memória virtual. Os seus argumentos parecem bastante razoáveis para nos motivar para a utilização da compressão em sistemas de memória atualmente. Ademais é bastante sensata a justificativa pela qual nós devamos estudar novamente esse assunto, já que estudos anteriores não foram consistentes em demonstrar uma melhora pelo uso do Cache Comprimido.

Infelizmente como seria aplicada a adaptabilidade numa implementação não é muito bem descrita no paper. Os seus parâmetros, ou seja, os dados coletados para decidir qual é o melhor tamanho do cache não são detalhados, nem como uma possível aproximação da idéia (para uma implementação) faria alguma diferença no desempenho do algoritmo.

As simulações feitas por Kaplan são "puras" no sentido em que só levam em questão o tempo exato gasto na compressão. Por isso, certos detalhes de implementação, que eventualmente tornariam inviável a idéia do Cache Comprimido, não são levadas em conta. Ademais, as simulações são feitas para aplicativos isolados, não para a execução de diversos aplicativos concorrentemente, como ocorre nos sistemas operacionais multiprogramados.

3.2 Compressed Caching and Modern Virtual Memory Simulation

Na sua tese de doutorado, Scott Kaplan detalha todo o seu trabalho na idéia de Cache Comprimido e também de simulações de memória virtual. Aqui toda a motivação e *background* técnico são providos para apresentar as suas idéias inéditas em torno do Cache Comprimido e por que ela é interessante nos dias de hoje. Uma das principais motivações para a leitura é o detalhamento que ele faz nesse documento acerca da sua política de substituição de páginas EELRU e como um algoritmo adaptivo pode ser aplicado a essa política como também ao tamanho de um Cache Comprimido de memória. Como visamos implementar e verificar na prática essas idéias do Kaplan, precisamos ter o maior conhecimento possível sobre o que já foi estudado e publicado a seu respeito.

Tipo de Documento	Tese de Doutorado
Autor	Scott Frederick Kaplan
Orientadores	Paul Wilson e Donald S. Fussell
Universidade	The University of Texas at Austin
Data	Dezembro de 1999
`http://www.cs.amherst.edu/~sfkaplan/papers/sfkaplan-dissertation.ps.gz`	

Resumo Breve da Tese Observa-se que à medida que as velocidades das CPU crescem numa taxa de 60% ao ano, e que o tempo de latência do disco descresce a menos do que 15% ao ano, o número de ciclos computacionais gastos pelo sistema de paginação de memória virtual crescerá. Atualmente os custos de desempenho devido às decisões não muito boas de "caching" são muito mais altos do que na época em que os estudos iniciais de memória virtual e paginação foram feitos. Também é possível notar que nos dias de hoje o gerenciamento de memória é um problema crítico no desempenho de muitos tipos de computação e paginação é a operação com maior custo no gerenciamento de memória.

Paginação de memória virtual é conhecida pela queda de desempenho drástica que causa. Ao considerar o conceito de relatividade na escala temporal, foi possível ter um melhor entendimento do desempenho de sistemas de memória virtual para diferentes padrões de referência e esse entendimento permitiu desenvolver métodos simples e eficientes de reconhecimento dos diversos padrões no momento de execução. Para um sistema de memória virtual, o conceito de tempo depende do comportamento de referência e do tamanho da memória. Ele pode avançar muito rapidamente ou vagarosamente dependendo das fases de execução do programa.

Entre outros, um dos importantes desenvolvimentos apresentados aqui nessa tese é a habilidade de decair a informação de referência numa maneira que seja relativa ao tempo. Métodos que dão maior peso a eventos de memória virtual do passado recente se adaptam muito mais facilmente às mudanças de fases. A nova política de substituição de páginas apresentada, EELRU ou "Early Eviction LRU", e o Cache Comprimido com Tamanho Adaptativo, utilizam esse tipo de decaimento de informação. Ao se coletar informação de uma fila LRU para páginas residentes e não-residentes, esse tipo de adaptabilidade pode detectar padrões de referências para os quais eles podem se adaptar. Essa adaptabilidade foi aplicada de um modo bastante simples: utilizamos informações de recentidade para calcular a análise de custo-benefício que permita a um esquema de memória virtual escolher a abordagem de gerenciamento que traga mais benefício.

Uma nova política de substituição de páginas é proposta aqui. Ela se chama EELRU, o que significa "Early Eviction LRU". É de conhecimento geral que a política amplamente utilizada nos sistemas operacionais atuais é o LRU ("Least Recently Used"), que por sinal executa muito bem na grande maioria dos casos. No entanto, em determinadas ocasiões, de acordo com o padrão de referência, o LRU tem um desempenho terrível, removendo da memória exatamente a página que será mais brevemente usada. Por esse motivo o EELRU foi criado, com base numa idéia muito simples: execute o LRU a não ser que, no passado recente, muitas páginas lidas tenham acabado de ser despejadas. Se isso ocorreu, aplique um algoritmo de "retirada", ou seja, ou despeje a página escolhida pelo LRU ou despeje a e-ésima página mais recentemente usada, onde e é uma posição de recentidade pré-determinada.

Muitos avanços na área de memória virtual foram perdidos porque diversas novas idéias foram verificadas através de uma implementação real, e os resultados foram influenciados por aspectos de um sistema em particular. Um bom exemplo é o próprio Cache Comprimido, que não demonstrou ter pouco ou nenhum benefício, apesar da idéia ser válida e os culpados foram a implementação e/ou o hardware. Por outro lado, idéias que deram certo em implementações reais são raramente examinadas com mais detalhamento, e a causa do seu benefício não é bem compreendida. Em resposta a esses problemas com uma implementação real, as técnicas apresentadas aqui, como o EELRU e o Cache Comprimido, foram desenvolvidas e compreendidas através de uma simulação dirigida a *reference traces*.

De modo a determinar o efeito de diferentes variáveis num sistema de memória virtual, gostaríamos de poder controlar os experimentos. Também gostaríamos que nossos experimentos fossem reproduzíveis e que pudessem ser utilizados com uma grande quantidade de configurações. A resposta que surge para esses problemas, como já dito no parágrafo anterior, são os *reference traces*. No entanto, esses *traces* são muito difíceis de se coletar e de se compartilhar devido ao seu tamanho. Além dos *traces* serem enormes pelo fato dos processadores efetuarem dezenas a centenas de milhares de referências por segundo, é importante, para o estudo de Cache Comprimido, que se aumente os *traces* com as imagens de cada página referenciada.

Para resolver os problemas citados acima, existem alguns métodos de redução de *traces*. Entretanto, os métodos existentes sofrem severas limitações, como: erro introduzido pela redução maior que o desejado, *traces* que não reduzem o tempo de simulação, uma determinada saída que não era um *trace* de referências, etc. Para acabar com essas limitações, é apresentado nessa tese o SAD (Safely Allowed Drop) e o OLR (Optimal LRU Reduction). Eles foram usados para coletar *traces* que conseguiriam os resultados exatos para memória LRU da escolha do usuário, de tamanho mínimo. Graças a essas reduções, os *traces* SAD e OLR permitiram a simulação de idéias como o Cache Comprimido possíveis.

De modo a tentar obter dados mais positivos sobre a viabilidade do uso de Cache Comprimido, era de vital importância a utilização de um bom algoritmo de compressão. O conceito de bom significa um algoritmo que comprima bem os dados do nosso domínio (memória) e também que seja eficiente na compressão em relação aos algoritmos de uso geral existentes. Os algoritmos WK desenvolvidos nesse tese demostraram duas idéias muito importantes: que há regularidades muito fortes e simples para dados de memória que podem ser capturados sem um grande custo computacional. Em segundo lugar, a especificidade de domínio do WK demonstra que compressores de uso geral podem ser usados em larga escala, mas não nos dão necessariamente a melhor compressão nem a execução mais rápida.

Cache Comprimido foi uma idéia prematuramente descartada, e a opinião generalizada que se tem hoje é que somente é útil para máquinas que sejam devagar ou sem swap. Considerando a grande diferença (que só tende a aumentar) entre processadores e velocidades de disk, como os melhoramentos na compressão de dados de memória, resolveu-se reexaminar a idéia de Cache Comprimido. A adaptabilidade relativa à escala de tempo utilizada pelo EELRU para detectar padrões na recentidade de referência também foi aplicada ao problema de escolher o tamanho do Cache Comprimido. A adaptabilidade introduziu um pequeno *overhead* ao selecionar tamanhos de Cache Comprimido que levavam a uma paginação reduzida para muitos programas diferentes numa grande variedade de tamanhos de memória. Apesar desse *overhead* introduzido, demonstrou-se nas simulações efetuadas que é possível ter uma grande melhora nos custos de paginação, entre 20 e 80%, com uma média de 40%.

As vantagens dos Cache Comprimido devem ainda ser maiores à medida que a diferença entre a velocidade de processador e velocidade de disco cresça. Compressão e descompressão serão menos caras. Não será apenas possível acomodar um número maior de hits no Cache Comprimido, mas também o método de adaptatividade poderá tentar ser menos agressivo, visto que erros serão menos custosos. Esses ciclos de processador poderão também ser usados para comprimir mais, resultando em tamanhos de memória efetivos ainda maior.

Ainda é necessário estudar os efeitos do Cache Comprimido em ambientes multiprogramados. Escalonamento e alocação de espaço para variáveis podem mudar drasticamente dadas as informações providas pela adaptabilidade baseada em recentidade, como também as opções utilizadas pelo Cache Comprimido. Cache Comprimido não é mais uma idéia relegada a máquinas pequenas ou mal conectadas, e sim pode ser aplicável a muitos diferentes tipos de "workloads" e configurações de hardware.

Análise da Tese Essa tese de doutorado, escrita por Scott F. Kaplan, é de fundamental importância para o nosso trabalho de mestrado. A motivação para o nosso estudo surgiu de um artigo que é fruto do trabalho que viria a se tornar essa tese de doutorado. Depois da leitura dessa grande tese (em tamanho também, pois são 320 páginas), pudemos ter noções bastante interessantes e mais completas sobre as idéias que tínhamos visto de um modo mais superficial nos seus artigos.

Boa parte do trabalho do Kaplan foi gasto em cima dos métodos que ele desenvolveu para redução dos *traces*, muito importantes para as simulações que seriam efetuadas no estudo da idéia de cache comprimido, entre outros. Apesar de muito interessantes as idéias, o nosso trabalho atual não faz uso delas.

Outra parte da trabalho do Kaplan está voltando para a família de algoritmos WK, que desenvolveu com o seu orientador (Paul Wilson). Na tese, as idéias do algoritmo estão descritas com muito mais detalhes que no artigo "The Case for Compressed Caching in Virtual Memory Systems". Esse conhecimento é de fundamental importância para a nossa análise de desempenho dos algoritmos na implementação que estamos desenvolvendo como também em um possível melhoramento do código dele, visto que o utilizaremos dentro do kernel e qualquer melhora pode fazer uma grande diferença.

O ponto de maior interesse nosso na leitura foi o algoritmo de adaptabilidade relativo à escala de tempo, que é utilizado para o algoritmo de adaptabilidade do Cache Comprimido como também do algoritmo de substituição de páginas EELRU. A impressão tida depois da leitura é que o Kaplan não nos indica como se relaciona os parâmetros do algoritmo de adaptabilidade com as implementações de sistemas operacionais reais. Além disso, o Kaplan faz simulações para aplicações isoladamente, não para ambientes multiprogramados, o que consequentemente não leva em conta diversos parâmetros de uma implementação real nem o *overhead* real que uma implementação da sua política de substituição de páginas (EELRU) ou o Cache Comprimido introduziriam.

A tese é muito bem escrita e a bibliografia é bastante completa, tendo servido como base para muitos dos nossos estudos.

3.3 The Compression Cache: Using On-Line Compression to Extend Physical Memory

Primeiro trabalho de implementação conhecido do cache comprimido, a leitura desse paper é imprescindível para quem deseja estudar esse assunto. Alvo de várias críticas por parte de estudiosos mais recentes, o trabalho de Douglis merece ser lido com uma visão crítica. A sua experiência de implementação pode eventualmente nos ajudar em alguns aspectos na experiência que estamos vivenciando atualmente.

Tipo de Documento	Artigo (Paper em Conferência)
Autor	Fred Douglis
Conferência	Winter 1993 USENIX Conference
Empresa	Matsushita Information Technology Laboratory
Data	Janeiro de 1993
`http://www.research.att.com/~douglis/papers/ccache.ps.gz`	

Resumo do Paper Compressão é amplamente usada para armazenamento secundário e redes. O que Douglis sugere é que hoje é viável usar compressão para reduzir a demanda por memória também. A idéia básica é tomar alguma memória que seria normalmente usada diretamente por um aplicativo, e usá-lo para armazenar um número maior de páginas no formato comprimido. E essa área é chamada de cache de compressão.

Intuitivamente a idéia de trocar processamento por IO é bastante atrativa: processadores estão tendo o seu desempenho aumentado mais rapidamente que os dispositivos de IO, especialmente discos. Se pudermos comprimir algumas páginas de modo que elas ocupem memória o suficiente de modo a permitir que todo o espaço de endereçamento do processo caiba na memória, será possível evitar IO para swap completamente. Consequentemente o processo executaria mais rápido.

As diferenças dessa idéia em relação a paginar num sistema de arquivos ou a um disco que faz a sua própria compressão é: IO reduzido, alocação de memória variável e *overhead* de espaço e de complexidade. Independentemente se a compressão é feita pelo sistema de VM ou implicitamente quando as páginas são transferidas para o local de armazenamento, a efetividade de comprimir páginas do VM depende de diversos fatores: velocidade de compressão, taxa de compressão, padrão de acesso às páginas, overhead de memória e implementações da compressão.

O desenho e implementação do cache de compressão foram feitos no sistema operacional Sprite. O Sprite é compatível com o 4.3BSD, mas a memória física é "negociada" entre o sistema de memória virtual (VM) para processos e para o buffer cache do sistema de arquivos, ou seja, eles competem entre si. Essa é a base para que o cache de compressão varie o seu tamanho dinamicamente também.

O ambiente alvo para essa pesquisa consiste de computadores móveis com memória e banda de rede limitadas, e com discos pequenos ou mesmo sem disco. O protótipo do cache de compressão foi feito num ambiente de workstation, rodando numa DECstation 5000/200, paginando para um disco RZ57.

O cache de compressão forma um novo nível na hierarquia de gerenciamento de memória. A descrição do cache é a seguinte:

– páginas LRU são comprimidas para dar espaço a novas páginas.

– se todas as páginas não cabem na memória, mesmo com algumas comprimidas, as páginas comprimidas LRU são escritas em disco.

– para tratar uma página que não está descomprimida e residente na memória, o sistema de VM verifica se a página está comprimida na memória ou então armazenada no disco.

Inicialmente o cache de compressão foi implementado como uma região de tamanho fixo da memória física. Isso foi feito por simplicidade e parcialmente porque a necessidade de variar o seu tamanho não era aparente. O cache de compressão consistia de um número de páginas, cada uma dividia em N fragmentos. Quando uma página era comprimida, o sistema alocava fragmentos suficientes para armazenar os dados comprimidos (não precisam ser contínuos). Infelizmente essa implementação era adequada apenas para aplicações que paginavam pesadamente mesmo sem o cache comprimido e que cabiam no cache comprimido sem o tráfego excessivo para o disco. E por esse motivo o cache de compressão foi redesenhado para variar com o tempo.

Para estender o desenho anterior, havia um problema relativo à reclamação de uma página inteira do cache de compressão e como ela seria copiada para disco. Para evitar esse problema, o cache de compressão foi tratado como um buffer circular de tamanho variável. Páginas físicas são mapeadas no espaço de endereçamento virtual do kernel, uma atrás da outra, dando a volta para o começo da faixa de endereços para um cache de compressão. E há uma noção de página física mais velha e novas páginas.

Uma thread do kernel escreve dados sujos no cache de compressão de modo a se tentar manter uma quantidade de páginas físicas livres e prontas para reclamação.

O método de escolher quando aumentar ou diminuir o cache de compressão é similar ao algoritmo do Sprite para negociar memória entre o sistema de arquivos e o VM. Ele compara a idade do bloco de arquivo menos recentemente usado com a idade da página do VM LRU e reclama a mais velha das duas, modulo um ajuste em favor de manter as páginas do VM um pouco mais. Na implementação, alocação de cada uma dos três tipos de memória (blocos de cache do sistema de arquivos, memória não-comprimida e memória comprimida) requer uma comparação das idades das páginas mais velhas para os três tipos. O sistema tem um viés para as idades em favor das páginas comprimidas sobre as páginas não-comprimidas e ambas em cima dos blocos de cache de arquivos. Quanto mais o sistema favorecer as páginas comprimidas, mais o sistema tenderá a crescer em períodos de paginação alta. Com um viés muito baixo, o cache de compressão vira um buffer para comprimir e descomprimir páginas entre a memória e o disco.

Há um pequeno *overhead* de espaço introduzido pelo cache de compressão. Primeiro temos o *overhead* para o algoritmo de compressão. Em segundo há um pequeno *overhead* nas estruturas de dados para cada página física utilizada para a compressão e também nas tabelas de páginas.

A melhora dada pelo cache de compressão foi estimada por cima para aplicações reais. Para tal, foi utilizado um programa chamado *thrasher* que circular linearmente um "working set", lendo (e opcionalmente escrevendo) uma palavra de memória para cada página por vez pelo working set. Dado essa estimativa por cima, foram feitos testes com algumas aplicações para verificar qual seria a melhora (ou piora) com o cache de compressão. Apesar de algumas poucas melhorarem, outras não demonstram nenhuma melhora e várias tiveram uma piora significativa.

Segundo Douglis, a compressão pode reduzir a quantidade de IO em tráfego para o disco, até eliminado-o completamente. Dependendo de uma série de fatores, essa técnica pode melhorar o desempenho por fatores de 3 a 4 vezes no melhor caso. No entanto, aplicações "reais" geralmente não obtém esse grau de melhora por causa de:

- localidade, que causa uma aplicação a tomar falhas em páginas comprimidas que seriam acessadas diretamente num sistema sem modificações;
- baixa compressibilidade;
- IO restrito, que causa transferências maiores serem executadas mais do que o necessário.

Análise do Paper Esse paper foi um dos primeiros a se tratar de cache comprimido (tratado aqui por cache de compressão) e o primeiro a implementar essa idéia para verificar na prática como ela se comportava. Ainda hoje é uma referência para todos os estudos de cache comprimido.

Infelizmente a pesquisa do paper resultou numa visão bastante negativa do uso de cache comprimido, mesmo para sistemas sem disco ou com pequeno espaço em disco. Kaplan, no seu artigo e tese de doutorado, argumenta que o estudo do Douglis tinha alguns problemas, entre eles o fato do cache adaptativo dele não se adaptar corretamente para o comportamento dos aplicativos e também pelo fato de ser um estudo velho, em que a diferença entre a velocidade de processamento e velocidade dos discos não era tão grande como hoje em dia.

É sem dúvida, apesar dessas críticas mais atuais a esse estudo, uma leitura imprescindível para quem quer estudar cache comprimido. Creio que podemos aprender bastante com alguns problemas de implementação que o Douglis teve que solucionar e que também estamos enfrentando ou ocasionalmente enfrentaremos.

3.4 Improving Application Performance through Swap Compression

Esse paper foi publicado na mesma época que o paper do Wilson, Kaplan e Smaragdakis nos anais de um evento que ocorreu paralelamente à conferência técnica anual da USENIX de 1999, a FREENIX. Com um trabalho de implementação próximo ao nosso, visto que eles implementaram a idéia de cache comprimido no Linux 2.0, é muito interessante conhecer a experiência do grupo da Universidade Politécnica da Catalunha, mesmo que o trabalho deles não venha a acrescentar nada em termos de idéias relacionadas ao cache comprimido em si (talvez somente em relação às técnicas de implementação).

Tipo de Documento	Artigo (Paper em Conferência)
Autores	R. Cervera, T. Cortes e Y. Becerra
Conferência	1999 FREENIX
Universidade	Universitat Politèctnica de Catalunya - Barcelona
Data	Junho de 1999
`http://www.usenix.org/events/usenix99/full_papers/cervera/cervera.ps`	

Resumo do Paper O mecanismo de swap adiciona um grande *overhead* ao desempenho de grandes aplicações. Além disso, também se encontra esse problema quando temos diversos aplicativos grandes rodando. Para que todos executem ao mesmo tempo, é necessário que se use o espaço em swap. Ainda mais é usual que esses aplicativos grandes não rodem em computadores mais modestos, como são normalmente os laptops ou computadores em casa.

Para esses dois problemas aqui é apresentada uma solução que foi implementada no kernel do Linux. A idéia consiste em comprimir páginas que foram enviadas para swap e mantê-las no *swap cache* o quanto for possível, tentando evitar ao máximo o acesso ao disco. Além disso, essas páginas são armazenadas comprimidas no swap, aumentando o tamanho do swap e possibilitando dessa maneira que se rode programas maiores. Ainda outra vantagem é poder agrupar diversas páginas de modo que as escreva continuamente no disco.

Agora falemos de alguns detalhes técnicos do protótipo. Através de experimentos com os protótipos, foi decidido armazenar as páginas comprimidas no cache apenas no momento de *swap out*, não no *swap in*. O algoritmo utilizado para a compressão das páginas foi o LZO, baseado no compressor Ziv-Lempel, obtendo taxas de compressão de compressão de mais de 50% na maior parte dos experimentos.

Foram adicionadas algumas estruturas ao kernel do Linux, entre elas o *virtual_swap_info*, que armazena informações de todas as páginas comprimidas que estão no swap. O *swap_cache_info*, por sua vez, é uma estrutura de todas as páginas do cache comprimido na memória. Outras pequenas alterações foram feitas nas estruturas existentes no kernel.

No momento que se obtém uma página de swap, na função *get_swap_page()*, ao invés do offset do arquivo de swap no disco, com essa implementação obtém-se um indíce do *virtual_swap_info*. No caso de se liberar uma página, a idéia é bastante simples, apenas um contador é atualizado. A discussão aqui é sobre a recompactação das páginas quando uma página no começo ou no meio do bloco é liberada. Isso não é feito por questões de grande *overhead* quando falamos de blocos de disco. Na questão dos *buffers*, é possível essa recompactação, mas não aumenta a performance do sistema e torna o código mais complexo. Dessa maneira, o espaço só será reutilizado quando todas as páginas no bloco ou no *buffer* tiverem sido liberadas.

Uma vez o sistema deseja efetuar *swap out* de uma página, ela será comprimida e se tentará inserí-la no cache. No caso de não haver mais espaço, é executada uma operação de limpeza no cache. Essa operação consiste em enviar os chamados *buffers cheios* para o disco, que são *buffers* que têm menos espaço em média que as últimas 100 páginas comprimidas. Essa escrita em disco é feita sincronamente. Tratemos agora do *swap in*. Quando o sistema precisa efetuar o *swap in* de páginas, o sistema procura por elas no cache comprimido. Se for achada, o sistema a descomprime e retorna para o tratador de exceção. Senão, o disco é lido e os dados lidos são descomprimidos. É importante observar que para modificar alguns parâmetros do cache, existe um driver que permite que o superusuário os modifique desde que o swap esteja desligado.

Para verificar a eficácia da idéia, foram feitos diversos testes. Foi verificada a influência da concorrência dos processos de testes, a quantidade de IO feita pelo sistema e a taxa de compressão. Apenas um dos testes teve queda de desempenho, de 4%, enquanto outro teve aumento de desempenho de até 650% (6,5 vezes). O ganho de performance foi entre 20 e 100% excluindo essas duas exceções. 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. Para a maior parte dos casos, o tamanho do swap foi “aumentado” por mais de 50%.

Análise do Paper O paper implementa a idéia do cache comprimido no kernel do Linux. No entanto, o foco desse projeto não é o aumento de desempenho ao se diminuir o acesso ao disco e sim o aumento do espaço de swap. Dessa maneira, o próprio tópico de cache comprimido não foi suficiente explorado, tanto que o cache não leva em conta o comportamento dos processos em execução no sistema para determinar o tamanho do cache.

Características de implementação são introduzidas pelo grupo que implementou, como a definição de buffer cheios e o limiar de limpeza (*cleaning threshold*), que é uma espécie de marca d’água para os buffers no cache comprimido. Entretanto, a escrita em disco é feita sincronamente no protótipo utilizado no paper, o que não é a melhor opção, pois tem um desempenho medíocre perto da escrita asíncrona. Também é duvidosa se a escolha de se armazenar um vetor para todos os dados das páginas em swap é a melhor possível. E ainda deve-se refletir sobre o limite que é imposto, pela implementação, para o tamanho de páginas que serão armazenadas no swap.

De qualquer forma, é bastante interessante o paper pois é a implementação mais próxima do que está sendo feita no nosso projeto. Além de ser, em parte, do cache comprimido, ainda é feita no kernel do Linux e algumas das experiências obtidas por esse experimento poderão servir como ponto de partida.

3.5 Operating System Support for Fast Hardware Compression of Main Memory Contents

O trabalho da equipe de pesquisa da IBM introduziu compressão em hardware para os dados da memória principal. Apesar dessa idéia parecer, à primeira vista, um pouco distante da idéia de cache comprimido, na teoria é a mesma. E para o funcionamento do hardware deles, foi necessária uma alteração no SO, cuja escolha foi o Linux. A implementação no Linux para o hardware deles é a principal motivação para a nossa leitura acerca desse trabalho.

Tipo de Documento	Artigo (Paper)
Autores	Bulent Abali e Hubertus Franke
Empresa	IBM T. J. Watson Research Center
`http://oss.software.ibm.com/mxt/publications/cmp4papr.pdf`	

Resumo do Paper Um novo computador foi construído pela IBM cujos dados da memória principal são armazenados comprimidos. Esse paper apresenta o suporte desenvolvido para sistema operacional Linux para que seja possível a utilização desse novo hardware. A principal característica envolvida nesse novo computador é que a memória principal não terá um tamanho estático, assim como observado nos computadores usuais, mas sim um tamanho que será em 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 L3 de 32 Mb. As *cache lines* desse cache L3 são de tamanho de 1 Kb que são comprimidas antes de serem escritas para a memória comprimida. Na memória principal (comprimida), a unidade de armazenamento é um "setor" de 256 bytes. Assim, cada cache line de 1 Kb pode ocupar de 0 a 4 setores na memória principal. O controlador de memória executa a tradução do endereço real (endereços de memória visto na bus externa do processador) para endereço físico (usados para endereçar a memória comprimida) utilizando uma tabela de tradução de compressão. O algoritmo de compressão utilizado é uma variante paralelizada do Ziv-Lempel conhecida como LZ1.

O suporte à compressão no kernel do Linux é bastante simples. O sistema operacional continua utilizando o seu endereçamento usual e o sistema de hardware é o responsável por toda a conversão do sistema de endereçamento real para o físico. Nesse sistema de memória comprimido, existem dois tipos de taxa de compressão: a estática e a dinâmica. A primeira é definida na bios e será a quantidade que a bios reportará para o sistema operacional como sendo a memória física disponível. A dinâmica está em função do tempo e depende dos aplicativos que se está rodando.

O SO utilizará a memória enquanto para ele houver disponível, o que é dependente da taxa de compressão estática definida na bios. Dessa forma, é preciso cuidar do caso em que há exaustão de memória e é necessário diminuir a utilização da memória física. Isso pode ser feito ao se reduzir a utilização da memória real pelo SO. O kernel do Linux utiliza certas marcas d'água para controlar a utilização da memória física. Essas marcas d'água, que são valores constantes para o kernel do Linux, foram mudadas para dinâmicas na implementação do suporte à compressão. Assim, quando a utilização da memória física excede os limiares pré-definidos, a marca d'água de baixa memória é aumentada e o *paging daemon* é sinalizado para que comece a retirar páginas dos processos e dos caches. Se o uso for muito intenso e não der tempo de liberar as páginas necessárias para se evitar a exaustão da memória física, algumas kernel threads são usadas 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 dos 12 *benchmarks* rodados foi de 2,30, sendo a menor de 1,78 e a maior de 2,68. Na questão do impacto na performance, a taxa média de penalidade foi de 1,5% para os mesmos 12 benchmarks, e o melhor caso foi de 0,7% e o pior de 3,1%.

Análise do Paper A solução implementada pela equipe da IBM foi bastante razoável para o problema que eles precisavam enfrentar. Como toda a parte de gerenciamento da parte do comprimida é feita por hardware, o kernel não precisa estar ciente de que existe uma compressão dos dados numa camada mais inferior do hardware. É importante notar que eles chamam o kernel modificado de "compression-aware", o que pode ser um tanto enganoso. O kernel precisa apenas estar ciente de que tem um limite de utilização da memória (definido na taxa estática do artigo) e que em um determinado momento pode ser necessário que se reduza a utilização mesmo sem chegar ao limite máximo (esse é o caso da exaustão da memória física).

Principalmente pelo fato de eles terem utilizados kernel threads para evitar o esgotamento memória assim que se passou de um determinado limite, isso mostra que o desenho do novo sistema de computador não leva em conta o último nível na hierarquia de memória padrão, o swap. Talvez isso seja justificado em algum outro artigo sobre essa nova tecnologia.

Infelizmente não foi possível encontrar relação direta entre esse suporte ao sistema de memória comprimida com o suporte ao cache comprimido que é o tema do projeto do aluno. Uma das diferenças está no suporte do sistema operacional à compressão. No caso do cache comprimido, o cache irá utilizar todo o suporte do SO para gerenciar as páginas que conterão dados comprimidos. Além disso, a compressão será feita dentro do kernel, o que envolve sérias questões a respeito, como o momento mais apropriado para se comprimir, o momento apropriado para se fazer *swap out*, o momento apropriado para se descomprimir e a questão mais importante de todas: o tamanho do cache e a sua adaptabilidade. É importante notar que o enfoque do cache comprimido é aumentar o desempenho dos aplicativos ao se diminuir a paginação em disco pelo fato de se aumentar o tamanho efetivo da memória, enquanto o enfoque desse sistema de hardware da IBM é o aumento da memória, aproveitando a compressibilidade dos dados armazenados.

4 Linux - Sistema de Gerenciamento de Memória

4.1 Page replacement in Linux 2.4 memory management

A nossa implementação no kernel do Linux para o projeto de mestrado começou a ser feita na versão 2.4. Visto que é um grande desafio técnico a compreensão do sistema de memória virtual e o desenho de uma implementação eficiente e simples, é bastante importante ter contato com o único documento a respeito do gerenciamento de memória do Linux 2.4. Além de ser o único documento, o seu autor é altamente conceituado para escrever a respeito, pois faz parte do seleto grupo que desenvolve o gerenciamento de memória do Linux e grande parte do atual código-fonte é de sua autoria.

Tipo de Documento	Artigo (Paper em Conferência)
Autor	Rik van Riel
Conferência	2001 FREENIX
Empresa	Conectiva Inc.
`http://www.usenix.org/events/usenix01/freenix01/full_papers/riel/riel.pdf`	

Resumo do Paper O objetivo do paper é descrever 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.

O gerenciamento do kernel do Linux 2.2 é focado em simplicidade e pouco *overhead*. Apesar de ele funcionar bem na prática para a maior parte dos sistemas, possui alguns pontos fracos e simplesmente não funciona para determinados cenários. Memória no Linux é unificada, então toda a memória livre está na mesma lista livre e pode ser alocada para qualquer um das próximas estruturas em demanda: *slab cache*, *page cache*, *buffer cache*, *inode cache*, *dentry cache*, *SYSV shared memory* e memória virtual mapeada para processos.

A substituição de páginas no Linux 2.2 funciona da seguinta maneira. Quando a memória livre fica abaixo de um certo limiar, o *pageout daemon* (kswapd) é acordado. Esse *deamon* tenta manter memória livre suficiente, mas se não conseguir a tempo, os próprios programas de usuário acabam chamando o código de *pageout* durante a alocação. As seguintes funções são chamadas no laço principal do *daemon*: *shrink_mmap* (limpa bits de referência, enfilera páginas sujas velhas e libera páginas livres), *shm_swap* (*swap out* páginas de segmentos compartilhados que não foram acessadas recentemente), *swap_out* (desmapeia páginas que não foram referenciadas recentemente, começando o IO de *swapout* e colocando as páginas no *page cache*) e *shrink_dcache_memory* (que reclama páginas do cache de nomes do VFS).

Entre os problemas que se encontra no Linux 2.2 temos:

- equilíbrio entre despejo de páginas do *file cache*, páginas não usadas de processos e páginas de segmentos compartilhados;
- simples substituição NR (não referenciado) pode não identificar com precisão o *working set* versus páginas incidentemente acessadas, e pode levar a falhas de páginas extras;
- devido ao simples algoritmo de relógio no shrink_mmap, algumas páginas limpas, acessadas, podem ser despejadas antes de páginas sujas e velhas;
- o sistema reage muito mal a cargas de VM variáveis ou picos de carga depois de um período sem atividade no VM pois o kswapd só é executado quando o sistema tem pouca memória;
- com altíssimas cargas, a substituição NRU de páginas simplesmente não funciona.

A nova versão do Linux, a versão 2.4, 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 (no 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.

As mudanças no código de substituição de páginas aconteceram durante o período de congelamento (freeze) do código do kernel do Linux 2.4. Aqui será exibida uma visão geral das mudanças na substituição de páginas do 2.4.

- O envelhecimento de páginas, que estava nos kernels do Linux 1.2 e 2.0, foi reintroduzido. Algumas mudanças foram feitas para evitar os artifatos de envelhecimento baseado na página virtual.
- Envelhecimento e *flushing* de páginas foram separados para evitar o despejo de páginas "erradas" devido a interações entre eles. Eles sao as listas de páginas ativas e inativas.
- O *flushing* de páginas foi otimizado para evitar muitas interferência do IO de escrita no IO de leitura em momentos críticos.
- Envelhecimento controlado das páginas em *background* durante os períodos de pouca ou nenhuma atividade do 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 *readahead*.

O subsistema de memória virtual do kernel do Linux 2.4 está sendo bastante ajustado, então é muito cedo para se ter conclusões sobre o desempenho. Dados iniciais mostram 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 VM do 2.4 é muito menos suscetível a mudanças sutís do que o Linux 2.2. O bom desempenho e a insensibilidade relativa a mudanças sutis no ambiente podem ser tomadas como um bom indicativo de que o VM do Linux 2.4 é, além de consertos para os problemas vividos no Linux 2.2, também uma boa base para um desenvolvimento futuro.

Análise do Paper O artigo, como já mencionado, é escrito por uma das grandes autoridades do sistema de gerenciamento de memória do Linux 2.3 e 2.4: Rik van Riel. O artigo indica com bastante precisão os problemas encontrados no gerenciamento de memória do Linux 2.2 e os consertos introduzidos durante o desenvolvimento e primeiras versões do Linux 2.4.

Infelizmente o sistema de gerenciamento de memória foi a parte mais crítica durante as primeiras versões do 2.4 e sofreu enormes modificações, principalmente recentemente, onde algumas abordagens citadas no paper foram mudadas (por exemplo, o envelhecimento, que tinha mudado do 2.2 para o 2.4, voltou aos moldes do 2.2). De qualquer forma, não deixa de ser uma leitura imprescindível para sabermos o que levou o Linux 2.4 (ou pelo menos boa parte dele, depois das mudanças recentes) a ser como ele é.

5 NetBSD - Sistema de Memória Virtual

5.1 The UVM Virtual Memory System

O UVM é um novo sistema de memória virtual que foi implementado no NetBSD. Indicado para leitura por desenvolvedores do sistema de gerenciamento de memória do Linux, esse paper apresenta diversas inovações no tocante a melhoramentos no sistema de memória virtual dos *BSD, que tem um desempenho reconhecidamente excelente (mesmo em comparação com o Linux). Visto que novas idéias e experiências aumentam a visão global do pesquisador e podem vir a ser útil, a leitura desse paper nos parece bastante importante.

Tipo de Documento	Artigo (Paper em Conferência)
Autores	Charles D. Cranor e Gurudatta M. Parulkar
Conferência	1999 USENIX Annual Technical Conference
Universidade	Washington University
Data	Junho de 1999
`http://www.usenix.org/events/usenix99/full_papers/cranor/cranor.ps`	

Resumo do Paper Esse paper introduz o UVM, um novo sistema de memória virtual para o kernel do BSD. Esse sistema possui um melhor desenho e também um desempenho melhor em relação ao VM do 4.4BSD (que por sua vez é baseado no VM do antigo Mach).

A primeira pergunta que surge é: por que substituir o VM do BSD? As estruturas de dados e funções utilizadas pelo BSD para gerenciar memória são complexos e por isso difícil de manter. A abordagem do UVM foi tentar desenvolver as estruturas de dados e funções que o BSD herdou do Mach para um sistema de VM mais eficiente. No UVM foi mantida a camada dependente/independente de arquitetura e as estruturas de mapeamento do VM do BSD. O código de objeto de memória virtual, tratamento de falhas (*fault handling*) e de *pager* foram substituídos e novos mecanismos de movimentação de dados baseados na nova memória virtual foram introduzidos.

As principais estruturas de dados tanto do BSD VM como do UVM na camada independente de arquitetura são: espaço de memória virtual, mapa de memória, objeto de memória, *pager* e página.

O UVM introduz dois importantes melhoramentos aos mapas de memória. O primeiro é que as funções de mapeamento de memória foram redesenhadas de modo que elas fossem mais eficientes e seguras. Em segundo, a fragmentação de memória devido ao *memory wiring* foi bastante reduzido. O UVM evita a fragmentação da entrada do mapa e as penalidades associadas em 4 dos 5 jeitos em que a memória pode ser *wired*.

No caso de objetos de memória, em contraste com o BSD VM, no UVM é considerada uma estrutura secundária desenhada para ser incorporada dentro de uma estrutura maior de modo a prover o tratamento de mapeamento de memória. O gerenciamento de objetos de memória no UVM é mais eficiente, mais flexível e cria menos conflito entre o sistema de VM e os subsistemas externos do kernel do que no BSD VM. Olhando a questão de eficiência, no UVM, por exemplo, objetos de memória são alocados e gerenciados em cooperação com as suas fontes de dados. No caso do BSD VM, os objetos de memória e suas fontes de dados são alocados e gerenciados separadamente.

Ao se tratar de memória anônima, o UVM consertou o vazamento de memória do swap, criou um sistema de *copy-on-write* mais eficiente e ainda tem um código menos complexo. No BSD VM, é preciso estar preparado para percorrer uma cadeia de objetos multi-níveis para achar os dados necessários. Cada objeto na cadeia tem o seu próprio conjunto de operações de I/O, o seu próprio lock, o seu próprio objeto sombra e o seu próprio conjunto de memória física e espaço em swap. No UVM, pode-se fazer a mesma função usando um mecanismo de busca de apenas dois níveis. É necessário apenas verificar o amap e então a camada de objeto para achar dados. Em vez de usar listas de objetos, o UVM usa contadores de refências no amap e nos anons para restrear os acessos à memória anônima.

O UVM introduz três importantes melhoramentos para os *pagers*. A alocação de estruturas de dados é mais eficiente, a API dos *pagers* agora é mais flexível dando ao pager mais controle sobre as páginas que ele posssui, e *clustering* agressivo foi introduzido no sistema de memória anônima.

O UVM também inclue três novos mecanismos de movimentação de dados baseados na nova memória virtual que são mais eficientes que cópias de grandes dados quando transferindo grandes blocos de dados. Empréstimo de páginas permite que páginas do espaço de endereçamento de um processo sejam emprestadas a outros processos. Transferência de páginas permite que páginas do kernel ou de outros processos sejam inseridas no espaço de endereçamento do processo facilmente. Passagem de entradas do map permite aos processos copiarem, compartilharem, ou moverem partes do seu espaço de endereçamento virtual entre eles mesmos.

Substituindo o sistema de VM do BSD com o UVM melhorou tanto a eficiência geral como o desempenho geral do kernel do BSD. O UVM se comporta muito melhor e mais eficientemente em todos os testes analisados em comparação com o BSD VM.

Análise do Paper Esse paper é de bastante alto nível, tendo a sua leitura sido indicado por desenvolvedores do kernel do Linux para o aluno. O autor descreve com bastante conhecimento as diversas limitações e problemas existentes no sistema de memória virtual do BSD 4.4.

O conhecimento dos diversos problemas do BSD 4.4 como também as bastante interessantes soluções estudadas e implementadas pelos autores serão de bastante utilidade para uma análise mais crítica do sistema de memória virtual com o qual estamos trabalhando e para uma visão mais ampla do que existe nos sistemas de memória virtual nos sistemas operacionais conhecidos.

6 MachOS

O MachOS foi um sistema desenhado e implementado por pesquisadores na Carnegie Mellon University. Em diversos aspectos dos sistemas operacionais, o MachOS foi revolucionário, com novas e admiráveis idéias. Entre elas, o sistema de gerenciamento de memória, na qual muitas das idéias fazer parte de sistemas bastante atuais e são muito admiradas.

Foram estudados três papers sobre o sistema de gerenciamento de memória do Mach para entender a motivação e detalhes das idéias que encontramos implementadas hoje em dia. O primeiro aborda a remoção do gerenciador de memória padrão do Mach de dentro do kernel, movendo-o para o espaço de usuário. O segundo é um paper específico sobre o desenho e a implementação do sistema de gerenciamento de memória no kernel do Mach, e o terceiro é um trabalho para uma nova abordagem que melhora o gerenciamento de memória distribuída no Mach.

6.1 Moving the Memory Manager out of the Mach Kernel

Tipo de Documento	Artigo (Paper)
Autores	David B. Golub e Richard P. Draves
Universidade	Carnegie Mellon University
`ftp://ftp.cs.cmu.edu/project/mach/doc/published/defaultmm.ps`	

Resumo do Paper Esse paper trata da implementação no kernel do Mach 3.0 do gerenciador de memória que reside completamente no espaço do usuário.

O gerenciador de memória padrão é um componente essencial do sistema operacional Mach 3.0. Ele provém armazenamento (*backing storage*) para objetos de memória temporários. Um gerenciador para memória temporária precisa operar sob as mais adversas condições. Ele é, no Mach, o gerenciador de memória dos últimos recursos para todos os outros gerenciadores de memória.

A interface externa de gerenciamento de memória presente na versões 2.5 e 3.0 do kernel do Mach permitiram que esses serviços sejam providos por servidores fora do kernel. Mas o gerenciador de memória temporária ainda continuou dentro do kernel. O motivo para isso é que ele precisa fazer uso de outros serviços do sistema em momentos em que eles seriam normalmente indisponíveis ou arriscados de se usar. Por exemplo, memória temporária não tem armazenamento em disco previamente alocado.

O gerenciador de memória padrão encontra diversas restriçoes de implementação pelo papel crucial no sistema:

- o gerenciador de memória padrão precisa ser residente;
- todas as páginas que estiverem movimentando entre o kernel e o gerenciador de memória padrão, e entre o gerenciador de memória padrão e o serviço de armazenamento em disco, precisam estar residentes;
- o gerenciador de memória padrão não pode bloquear para esperar por mais memória física;
- se o gerenciador de memória padrão é agrupado com outro serviço, as threads que tratam as funções dele precisam ser distinguidas das threads que não o fazem. Se as threads do gerenciador de memória padrão bloquearem esperando por threads não privilegiadas, um deadlock ocorrerá;
- o gerenciador de memória padrão não pode bloquear para esperar pelo sistema de arquivo alocar espaço em disco temporário, uma vez que o sistema de arquivo pode ser paginável.

Para tirar o gerenciador de memória padrão do espaço de endereçamento do kernel, as funções privilegiadas do kernel tinham que ser substituídas por funções equivalentes acessíveis a partir de programas de usuário. Apenas algumas funções eram acessadas diretamente: alocação de espaço em disco, sincronização de threads, alocação de memória travada, e prevenção de que threads fossem mandadas para swap. Essas operações foram substituídas por rotinas do modo do usuário ou foram adicionadas à interface do kernel onde estivessem faltando.

A interface do kernel foi estendida com duas chamadas originalmente propostas para suportar o real-time no Mach. E uma chamada ao kernel existente identifica o gerenciador de memória padrão. O kernel usa o port de serviços do gerenciador de memória padrão para tratar a paginação para objetos temporários.

A maior função que é usada diretamente pelo gerenciador de memória padrão no Mach é o código do sistema de arquivos. No entanto, isso exige que todo o código do sistema de arquivos fique residente e que qualquer thread que acesse o sistema de arquivo esteja residente também enquanto executar aquele código. A solução para a remoção do gerenciador de memória padrão é que ele use uma lista pré-especificada de blocos de disco através de um arquivo de paginação existente no disco com o nome pré-definido.

Para sincronização e *locking*, o gerenciador de memória padrão utiliza o pacote CThreads. Os alocadores de memória da biblioteca C padrão (malloc e free) foram substituídas pelas nossas próprias rotinas.

O gerenciador de memória padrão recebe memória em mensagens de poucas fontes no kernel e no serviço de dispositivos. Eles foram modificados para saber quando estivessem enviando memória para o gerenciador de memória padrão e travá-la na memória.

O gerenciador de memória padrão foi construído e opera como parte da tarefa de serviço do *bootstrap* do Mach 3.0. Ela é a primeira a ser rodada depois que o kernel é inicializado. Além do gerenciador de memória padrão, ele contém o carregador *bootstrap* para servidores do modo do usuário. E visto que ambos compartilham um sistema de arquivos em comum, ele foram removidos do kernel como um só.

Entre os resultados, construir o kernel do Mach 3.0 e o gerenciador de memória padrão é muito fácil. Eles são auto-contidos e um novo serviço de *bootstrap* pode ser adicionado ao arquivo de boot do Mach 3.0 sem ter que refazer o kernel.

Comparando a versão MK63 do Mach 3.0 com o gerenciado de memória padrão fora do kernel desmonstra que há um pequeno efeito no desempenho geral do sistema, com degradação entre 1 e 6%, dependendo da quantidade de memória disponível. Espera-se que um ajustes finos do código irão diminiur essa perda de desempenho.

Análise do Paper Paper muito interessante de Golub e Davis que propõe a retirada do gerenciador de memória padrão do Mach (ou gerenciador temporário, ou pode ser chamado ainda da parte mais central do gerenciador) e um estudo sobre ele.

O texto não nos motivou para a necessidade da remoção do gerenciador de memória padrão. Apesar disso, abordou com competência essa questão e demonstrou que a sua implementação não oferece grandes perdas de desempenho. Um dos motivos pelo qual isso não acontece é que a comunição do gerenciador de memória padrão, mesmo dentro do kernel, utilizava IPC, logo o *overhead* introduzido ao se remover o gerenciador do kernel, nesse aspecto, não foi grande.

6.2 Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures

Tipo de Documento	Artigo (Paper em Conferência)
Autores	Richar Rashid, Avadis Tevanian, Michael Young et al.
Conferência	2nd Symposium on Architectural Support for Programming Languages and Operating Systems - ACM
Universidade	Carnegie Mellon University
Data	Outubro de 1987
`ftp://mach.cs.cmu.edu/doc/published/MIvmm.ps`	

Resumo do Paper Esse paper descreve o desenho e a implementação do gerenciamento de memória virtual do sistema operacional Mach da CMU e as experiências adquiridas pelo grupo do kernel do Mach em portar esse sistema para uma variedade de arquiteturas.

Sistemas UNIX tradicionalmente abordam o problema da portabilidade do VM restringindo facilidades que são oferecidas e baseando implementações para novas arquiteturas de gerenciamento de memória em versões já prontas para sistemas anteriores. Dessa forma, eles oferecem pouco na questão de memória virtual além de suporte à simples paginação.

A CMU trabalhou durante alguns anos no desenvolvimento de um sistema operacional portável para multiprocessadores chamado Mach. Um dos objetivos do Mach é explorar a relação entre arquiteturas de memória de hardware e de software e para desenhar um sistema de gerenciamento de memória que fosse portável para sistemas de multiprocessadores como sistemas de um processador tradicionais.

O Mach provém compatibilidade com o UNIX 4.3BSD e estende as noções UNIX de gerenciamento de memória virtual e comunicação entre processos. Ele suporta: espaço de endereçamento virtual esparso e grande; operações de cópia virtual *copy-on-write*; *copy-on-write* e compartilhamento de memória *read-write* entre as tarefas; arquivos mapeados para memória e *pagers* e objetos de armazenamento fornecidos pelo usuário.

Basicamente existem cinco abstrações do Mach:

Tarefa: um ambiente de execução no qual as threads rodam. Tem um espaço de endereçamento paginado e acesso protegido aos recursos do sistema. Consiste de uma coleção de mapeamentos para objetos de memória.

Thread: unidade básica de utilização da CPU. Todas as threads de uma tarefa compartilham acesso a todos os recursos.

Port: canal de comunicação. São objetos de referência do desenho do Mach.

Mensagem: coleção (tipada) de objetos de dados usados em comunicação entre threads.

Objeto de Memória: coleção de dados providos e gerenciados por um servidor que pode ser mapeado no espaço de endereçamento de uma tarefa.

Operações em objetos são efetuadas enviando-se mensagens para as ports. O Mach permite, assim, que serviços de sistema e recursos sejam gerenciados por tarefas do usuário. Tradicionalmente sistemas baseados em mensagens desse tipo funcionam com uma desvantagem de desempenho distinta aos sistemas operacionais implementados convencionalmente. A chave para a eficiência no Mach é a noção de que o sistema de gerenciamento de memória virtual foi integrado com a facilidade de comunicação orientada a mensagens.

Cada processo no Mach possui um espaço de endereçamento que consiste de mapeamentos entre faixas de memória endereçáveis para a memória e objetos de memória. Uma tarefa pode modificar o espaço de endereçamento de diversos modos, como:

- alocar uma região de memória virtual alinhada com o tamanho da página;
- desalocar uma região de memória;
- definir proteções de uma região de memória virtual;
- especificar a herança de uma região de memória virtual;
- criar e gerenciar um objeto de memória que pode ser mapeado no espaço de endereçamento de outra tarefa.

Compartilhamento de escrita/leitura pode ser criado alocando uma região de memória e definindo o seu atributo de herança. Herança pode ser especificada como *shared*, *copy* ou *none* e pode ser especificada por página. Proteção também é especificada por página.

Quatro estruturas de dados básicas do gerenciamento de memória são usadas no Mach:

- tabela de página residente;
- mapa de endereços;
- objeto de memória;
- pmap.

A implementação é dividida entre seções dependente e independente de máquina. Código dependente de arquitetura somente implementa operações necessárias para criar, atualizar e gerenciar as estruturas de dados de mapeamento necessárias para o hardware. Toda a informação importante para a memória virtual é mantida no código independente de arquitetura.

Informação acerca das páginas físicas é mantida em entradas de página numa tabela indexada pelo número da página física. As entradas de páginas podem ser mantidas simultaneamente em diversas listas: lista de objetos de memória, fila de alocação de memória e tabela de hash objeto/offset.

Espaços de endereçamento de uma tarefa são mapeados para offsets de bytes nos objetos de memória por uma estrutura de dados chamada mapa de endereçamento. Essa mapa é uma lista ligada de entradas do mapa de endereços e cada um desses mapeia uma área contínua dos endereços virtuais numa área contínua de um objeto de memória. Cada entrada do mapa de endereçamento tem informação sobre os atributos de herança e proteção da região de memória que define. O mapa de endereçamento eficientemente implementa operações de um espaço de endereçamento de tarefa: procura no tratamento de falhas de páginas, operações de cópia/proteção em regiões de endereços e faixas de endereços de alocação/desalocação.

Um objeto de memória é um repositório para dados, indexado por byte, sobre o qual várias operações podem ser feitas. Uma importante característica da memória virtual do Mach é a habilidade de tratar falhas de páginas e pedidos de *pageout* fora do kernel. Isso é feito com objetos de memória ao se associar a ele uma tarefa de gerenciamento.

Quando uma cópia *copy-on-write* é feita de uma tarefa para outra, se uma das tarefas escreve nos dados, uma nova página é alocada na qual a tarefa que está escrevendo coloque as modificações. O Mach gerencia a informação sobre quais objetos foram modificados ou não com a criação de objetos de memória que terão as páginas modificadas que originalmente pertenciam a outro objeto. Esses objetos criados para esse propósito são chamados de *shadow objects*. Visto que podemos ter cadeias grandes de *shadow objects* que podem surgir de remapemento copy-on-write repetido de um espaço de endereçamento para outro, o Mach automaticamente faz coleta de lixo dos *shadow objects* quando reconhece que um *shadow* intermediário não é mais necessário.

O código de memória virtual do Mach foi originalmente implementado em máquinas de arquitetura VAX, cuja primeira versão estável foi disponibilizada na CMU em Fevereiro de 1986. Uma versão foi portada logo em seguida para o IBM RT PC. Logo em seguida foram feitas implementações no SUN 3, Sequent Balance e Encore MultiMAX.

Alguns testes foram efetuados para comparar os desempenho do Mach com uma sistema tradicional UNIX, o 4.3BSD. Em todos os testes efetuados, o Mach tem um desempenho bastante melhor que o 4.3BSD.

Análise do Paper Esse paper é extremamente importante pois apresentou idéias que foram marcantes nos sistemas de gerenciamento de memória que vieram depois. De um modo geral, o sistema operacional Mach da Carnegie Mellon University inovou em diversos aspectos no sistema operacional. Analisando as idéias por todo o paper encontramos muitas similaridades com sistemas operacionais que vieram depois, como é o caso do Linux. Um exemplo bem claro dessa similaridade é a separação entre código dependente e independente de arquitetura.

Sistemas UNIX como os *BSD até recentemente (ou até hoje em dia em alguns casos) foram bastante baseados nessas implementações do Mach. Certamente alguns problemas na abordagem foram notados e a idéia do UVM, de Charles Cranor, veio corrigir alguns desses, como por exemplo a questão dos *shadow objects*.

Essa descrição do desenho e implementação do sistema de memória virtual talvez constitua um dos papers mais importantes para se ter uma idéia histórica e também da motivação por trás de diversas estruturas que hoje fazem parte de sistemas operacionais comuns.

6.3 A New Approach to Distributed Memory Management in the Mach Microkernel

Tipo de Documento	Artigo (Paper em Conferência)
Autores	Stephan Zeisset, Stefan Tritscher e Martin Mairandres
Conferência	1996 USENIX Annual Technical Conference
Empresa	Intel GmbH
Data	Janeiro de 1996
`http://www.usenix.org/publications/library/proceedings/sd96/full_papers/zeisset.ps`	

Resumo do Paper Esse paper descreve uma nova abordagem para estender as semânticas de memória virtual do Mach por nós de um sistema de multicomputadores. Essa abordagem é chamada de Advanced Shared Virtual Memory (ASVM).

Sistemas de multicomputadores geralmente utilizam diferentes cópias do seu sistema operacional em cada um dos nós processantes. Sistemas operacionais, entre eles o OSF1/AD TNC, suportam o conceito de uma imagem de sistema único. Para conseguir isso, o OSF1/AD TNC se baseia em dois adendos ao micro-kernel do Mach que são conhecidos como extensões NORMA (No Remote Memory Access): NORMA-IPC para estender comunicação entre processos e XMM para extender semânticas de memória virtual por nós de multi-computadores.

O desenho original do NORMA-IPC e do XMM não executavam muito bem em máquinas paralelas de grande escala. As principais deficiências do NORMA-IPC é o seu controle de fluxo quebrado nos cenários de comunicação muitos-para-um e a sua utilização pobre de banda de comunicação disponível. No caso do XMM, os problemas são a abordagem não escalável para tratar de memória compartilhada e um protocolo de comunicação ineficiente.

O Paragon é um sistema multiprocessado com memória distribuída. Três propriedades dele tiveram uma grande influência no desenho do ASVM: alta largura banda e interconexão de baixa latência, o grande número de nós que um sistema pode conter e o fato que há tipicamente um nó com um disco para 32 nós que são usados para computação.

Baseado nas experiências com o XMM, esse artigo apresenta o novo sistema com as seguintes características. Ao contrário do XMM, o ASVM usa uma abordagem de gerenciador distribuído para gerenciar relações de memória entre os nós. Cada página tem então o seu próprio gerenciador, que também é chamado do proprietário da página. A posse pode migrar entre todos os nós que usam uma página.

O ASVM não somente distribui a informação de estado da página pelo sistema, mas também a liga com a página física naquele nó que apenas tem informação de estado sobre páginas que estão na sua memória física.

Para evitar problemas de bloqueio da thread do pager durante as *page-faults*, o ASVM usa transições de estado asíncronas, que significa que nem a thread nem outros recursos são bloqueados enquanto esperam por um pedido ser respondido.

Na questão do protocolo de comunicação especializado, o ASVM usa o XMMI somente como uma interface para um sistema de VM e um *pager* local enquanto define o seu próprio protocolo para toda comunicação entre as instâncias ASVM em nós diferentes.

O protocolo ASVM é mapeado para uma interface de transporte dedicada, o serviço de transporte SVM (STS). Ela permite que o uso de pilhas do nível do protocolo menores.

O ASVM adota a abordagem do gerenciador distribuído dinâmico para os gerenciadores de página (entidade que mantém informação sobre o estado atual da página). Mas faz uma clara distrinção entre o método para distribuir gerenciadores de páginas e o método para achar o gerenciador atual de uma página. Um gerenciador é o proprietário da página, que mantém informação sobre o estado da página e é responsável por atender pedidos de acesso de outros nós. O outro é o gerenciador de posse, que é responsável por encaminhar pedidos para o proprietário da página.

Quando o sistema de memória virtual de um nó precisa acessar alguma página, um pedido de acesso é gerado e enviado para o redirecionador de pedidos, que é responsável para entregá-lo para o proprietário atual da página. Quando o pedido chega num nó proprietário da página, ele é enviado para uma máquina de estados que mantém a coerência da página com o uso compartilhado, cópia atrasada e paginação entre os nós.

São implementadas no ASVM as três estratégias de mecanismos de encaminhamento: encaminhamento dinâmico, estático e global. O ASVM tem a informação de posse para encaminhamento dinâmico e estático nos caches para as páginas mais recentemente acessadas.

O único modelo de coerência suportado pelo ASVM é a coerência forte, que significa que qualquer operação de leitura numa região de memória compartilhada retornará os dados da operação de escrita mais recente para esse endereço.

O ASVM estende a estratégia de cópia asimétrica do sistema de memória virtual pelos nós. A base para relacionamento de cópia entre nós é provida pela funcionalidade de memória virtual compartilhada do ASVM. Quando uma cópia de um objeto de memória deve ser mapeada em algum nó, primeiro um mapeamento compartilhado do objeto fonte é feita nesse nó. Então uma cópia local é criada pelos mecanismos padrões do sistema de VM. Depois todas as páginas residentes do objeto fonte em todos os nós que compartilham o objeto são marcadas como somente leitura. Isso é feito enviando-se uma mensagem para nós que estão compartilhando, o que faz com que eles executem um *memory_object_lock_request* na região copiada.

A funcionalidade de memória virtual compartilhada do ASVM foi implementada sem mudar o EMMI. Mas foram necessárias algumas extensões para dar ao gerenciador de memória controle sobre os mecanismos de cópia.

Um sistema Paragon com 72 nós GP foi utilizado para executar medidas de desempenho do ASVM com o XMM. Tanto nas falhas de página em memória compartilhada como em memória herdade, o desempenho do ASVM é incrivelmente maior que o XMM, tendo latência até 90% menores e uma escalabilidade excepcional.

Análise do Paper Esse paper é muito bem escrito e aborda uma série de limitações do kernel do Mach para o gerenciamento de memória distribuída. Muito interessante a sua leitura para o conhecimento das soluções implementadas pelos autores e importante para uma visão de um sistema distribuído, mas com o foco em gerenciamento de memória nos seus pequenos detalhes. Infelizmente não vemos aplicação imediata de alguma idéia com o nosso projeto tampouco alguma técnica de implementação pois as dificuldades envolvidas na implementação não são citadas no paper.

7 Tópicos Variados de VM

Nessa seção estudamos dois papers variados de SO relacionados à área de gerenciamento de memória. São papers sem aplicação clara ao assunto do projeto de mestrado, mas que permitem um conhecimento maior das últimas pesquisas relevantes na área, o que é bastante importante para um pesquisador. Os papers são recentes, de 1998 e 1999, e ambos de conferências da USENIX (conferência técnica anual e OSDI).

7.1 General Purpose Operating System Support for Multiple Page Sizes

Tipo de Documento	Artigo (Paper em Conferência)
Autores	Narayanan Ganapathy e Curt Schimmel
Conferência	1998 USENIX Annual Technical Conference
Empresa	Silicon Graphics Computer Systems, Inc.
Data	Junho de 1998
`http://www.usenix.org/publications/library/proceedings/usenix98/full_papers/ganapathy/ganapathy.pdf`	

Paper Muitas arquiteturas de microprocessadores suportam entradas no TLB com múltiplos tamanhos de páginas. Esse suporte pode ser usado para reduzir enormemente o *overhead* introduzido pelas faltas (*misses*) de TLB ocorridas quando o processador roda um aplicativo com um grande *working set*. Mas hoje em dia os aplicativos não fazem uso desse suporte pois a maior parte dos sistemas operacionais só suportam um tamanho de página.

Aqui nesse paper foi apresentado o desenho de um suporte geral de sistemas operacionais para múltiplos tamanhos de páginas. Ele foi implementado no SGI IRIX 6.4 numa plataforma SGI Origin 2000 que usa o microprocessador MIPS R10000.

O desenho do suporte a multiplos tamanhos de páginas incluem alocação dinâmica de páginas grandes, efetuar o *pagein* para o endereçamento do processo, efetuar o *pageout*, e aumentar e diminuir o tamanho dos tamanhos de página. Aplicativos devem ser capaz de usar páginas grandes sem qualquer modificação significativo ao seu código. Chamadas de sistema que dependem do tamanho de página devem se comportar da mesma maneira quando estiverem usando páginas grandes como se estivesse usando as páginas de tamanho padrão.

Um dos primeiros desafios no desenho é como gerenciar as estruturas de dados das páginas físicas, ou as *pfdats*. Depois de vários estudos e considerações, notou-se que há diversas desvantagens ao se ter os *pfdats* representando páginas de tamanhos diferentes. Dessa maneira, cada página grande será tratada como um conjunto de páginas padrão. Se o tamanho da página padrão é 4K e usamos páginas de 64K para mapear um segmento de um arquivo, haverá 16 *pfdats* no page cache para mapear um segmento.

A próxima decisão é a respeito do formato da tabela de páginas. Decidiu-se manter o layout existente da tabela de páginas que mapeia para as páginas padrão. Os PTEs (*page table entries*) que mapeiam páginas grandes parecem similar àqueles que mapeiam páginas padrões. Além disso, todos os PTEs têm um campo que contém o tamanho da página. O TLB *handler* precisa olhar para apenas um PTE de modo a conseguir toda a informação a ser colocada na entrada do TLB. Uma importante vantagem desse formato é que os subsistemas que lidam com PTEs não precisam ser modificados uma vez que o formato do PTE continua o mesmo e outra vantagem é que a informação do tamanho da página é mantida somente nos PTEs e portanto é privado ao processo. No entanto, a desvantagem dessa abordagem é que não se reduz o tamanho da tabela de páginas que se precisa ao se usar páginas maiores.

No tocante ao TLB *miss handler*, para evitar penalizar processos que não usam múltiplos tamanhos de páginas, apenas os processos que precisam de páginas grandes usarão o TLB *miss handler* para múltiplos tamanhos de páginas.

Mais um dos problemas relacionados às páginas grandes é a questão de fragmentação que ela introduz na memória. Para tal, é utilizado uma thread do kernel que gerencia a desfragmentação de memória. Um dos mecanismos utilizados para isso é a migração de páginas, que migra páginas físicas para outras e pode ser usada para criação de páginas adjacentes livres que serão usadas para formar uma nova grande página.

Para a plena utilização de páginas grandes, é importante que o *page fault handler* saiba lidar com falhas de páginas grandes. Para tal, o *handler* precisa ser capaz de alocar e inicializar páginas grandes, certificar-se de que as restrições de processador não são violadas e definir os PTEs. O *page fault handler* para páginas grandes foi criado em cima do *page fault handler* padrão para páginas de tamanho padrão e as extensões são mínimas.

Alguns benchmarks foram efetuados para verificar a implementação. O tamanho de página padrão era de 16K e os tamanhos testados foram 16K, 64K, 256K e 1M com uma variedade de programas. Os resultados foram bastante animadores. Entre os aplicativos testados, foram obtidas melhoras de desempenho de 10 a 20%. A queda nas faltas de TLB com páginas grandes foram surpreendentes, chegando a 98% em dois dos aplicativos ao se aumentar de 16K para 64K o tamanho da página e em mais de 80% para todos ao se aumentar o tamanho das páginas ainda mais.

Análise do Paper Esse paper foi o resultado de um trabalho de bastante alto nível da equipe da Silicon Graphics. Eles abordaram com bastante consciência o problema de modo a ter uma solução que fosse eficiente e trouxesse benefícios ao sistema com o mínimo de *overhead*. Baseado nos nossos conhecimentos de sistema operacional e da experiência que estamos tendo com o kernel do Linux, vemos com muitos bons olhos as abordagens que eles fizeram.

O paper não tem aplicação direta ao problema do Cache Comprimido, mas pode servir de inspiração na resolução de algum problema de implementação que possamos vir a enfrentar. Foi muito bem escrito e achamos que é um paper bastante importante para quem está trabalhando com pesquisa em sistemas operacionais, sobretudo quando envolve implementações.

7.2 Self-Paging in the Nemesis Operating System

Tipo de Documento	Artigo (Paper em Conferência)
Autor	Steven M. Hand
Conferência	3rd Symposium on Operating Systems Design and Implementation
Universidade	University of Cambridge Computer Laboratory
Data	Fevereiro de 1999
`http://www.usenix.org/publications/library/proceedings/osdi99/full_papers/hand/hand.pdf`	

Resumo do Paper Esse paper apresenta um esquema onde cada aplicação é responsável por toda a sua paginação (e outras atividades da memória virtual). Ao se fornecer às aplicações garantias para memória física e banda de disco, é possível isolar aplicações sensíveis a tempo (por exemplo CM ou *continuous media*, que lidam com áudio e/ou vídeo) do comportamento de outras.

Hoje escalonamento de CPU é feita com base num esquema de prioridade, que especifica quem mas não quando ou quanto. Outros recursos no caminho dos dados (disco ou rede, por exemplo) não são nem mesmo escalonados. Além disso, o SO executa um quantidade de tarefas potencialmente críticas de tempo para os aplicativos. Dessa maneira, os sistemas tendem a exibir um mau desempenho para aplicativos CM quando contenção é introduzida.

Gerenciamento de memória é reconhecidamente bastante complexo. Uma das maneiras de simplificar a abordagem tem sido a abordagem de micro-kernel. No entanto, nenhum dos sistema de micro-kernel conhecidos provém *Quality of Service* (QoS) no seu gerenciamento de memória: não há isolamento das aplicações, exposição insuficiente do hardware e não há responsabilidade dos aplicativos pelo desempenho dos outros.

O sistema operacional Nemesis foi desenhado e implementado no Laboratório de Computação da Universidade de Cambridge. Ele é um SO multi-serviço – ou seja, suporta um conjunto de aplicativos convencionais e sensitíveis a tempo. Um dos problemas importantes que ele aborda é a prevenção de *crosstalk*.

Auto-paginação (*self-paging*) provém uma solução simples para *crosstalk*: exige que cada aplicativo lide com todas as suas falhas de memória usando os seus próprios recursos concretos. Todas as operações de paginação são removidas do kernel e o kernel, por sua vez, é responsável por enviar notificações de falha.

Auto-paginação envolve três princípios:

controle: acesso a recursos é multiplexado em espaço e tempo. Recursos são garantidos em escalas de tempo de médio prazo.

poder: interfaces são suficientemente expressivas para dar às aplicações a flexibilidade que precisam. Altos níveis de abstração não são impostos.

responsabilidade: cada aplicação é diretamente responsável por cuidar das suas próprias operações de memória virtual. O sistema não provém uma rede de segurança.

As abstrações de memória virtual básicas do Nemesis são o *stretch* e o *stretch driver*. Um *stretch* simplesmente representa uma faixa de endereços virtuais com uma certa acessibilidade. Ele não possui, nem é garantido, nenhum recurso físico. Um *stretch driver* é responsável por prover a cobertura para o *stretch*. De um modo geral, ele é responsável por lidar com qualquer falha num *stretch*. Só através do *stretch driver* é possível saber o conteúdo de um *stretch*. *Stretch drivers* são desprivilegiados, aplicações do nível do objeto, instanciadas pelos módulos de criação do nível do usuário e fazendo uso apenas dos recursos possuídos pela aplicação.

Assim como no caso de memória virtual, a alocação de memória física é efetuada centralmente, no caso pelo alocador de *frames*. O alocador de *frames* permite um controle fino sobre a alocação de memória física, incluindo espaço de IO se for o caso. Um domínio pode exigir frames físicos específicos, ou frames dentro de uma região especial. Isso permite a uma aplicação com conhecimento de plataforma a possibilidade do uso de coloração de páginas, ou de mapeamentos de superpáginas no TLB. Uma política de alocação padrão também é fornecida para as que não tem exigências especiais.

O sistema de tradução lida com inserção, leitura ou remoção de mapeamento entre endereços físicos e virtuais. Ele pode ser considerado como uma tabela de informação sobre esses mapeamentos. O mapeamento real será tipicamente feito por qualquer hardware ou software de gerenciamento de memória que esteja presente. Ele é dividida em duas partes: um módulo de gerenciamento de alto nível e chamadas de sistema e tratadores de *traps* de baixo nível.

As faltas de TLB são tratadas pelo sistema de tradução de baixo nível. Todas as outras falhas são enviadas diretamente para o aplicativo faltante de modo a previnir *QoS crosstalk*. Para evitar a necessidade de bloquear o kernel para uma entidade do nível de usuário, a parte do kernel do tratamento de falhas é completada assim que o envio ocorreu. O aplicativo precisa executar qualquer operação adicional, incluindo continuação (ou terminação) da thread faltante.

Em algum momento depois que uma aplicação causou uma falha de memória, ela será ativada pelo escalonador. A aplicação então precisa lidar com todos os eventos desde a última vez que foi reativado. Isso é conseguido chamando um tratador de notificações para cada ponto final contendo um novo valor. Se não há um tratador de notificações registrado para um determinado ponto final, nenhuma ação é tomada. Seguindo isso, o escalonador de threads do nível do usuário é executado que escolherá qual thread rodará.

Como foi descrito anteriormente, a resolução atual de uma falha é da alçada do *stretch driver*. Um *stretch driver* é algo que provém recursos físicos para cobrir os endereços virtuais dos *stretches* pelos quais é responsável. *Stretch drivers* adquirem e gerenciam os seus próprios frames físicos e são responsáveis por definir o mapeamento de virtual para físico ao chamar o sistema de tradução.

O *stretch driver* pode potencialmente ter uma conexão com um dispositivo de armazenamento. Os mais importantes desses são o *User-Safe Backing Store* (USBS). Ele é dividido em duas partes: o sistema de arquivos de swap (SFS) e o disco *user-safe* (USD). O SFS é responsável por operações de controle como alocação de uma extensão (uma faixa contínua de blocos) para uso como um arquivo de swap e a negociação de parâmetros de *Quality of Service* para o USD, que é responsável por escalonar operações de dados.

Para verificar a combinação de funções de sistema de memória no nível da aplicação e de baixo nível, um conjunto de micro-benchmarks foram feitas no Nemesis e comparado com o Digital OSF1 V4.0 no mesmo hardware. Os resultados são impressionantes: de seis testes, cinco executaram em 50% ou menos tempo no Nemesis em comparação com o OSF1. O teste restante executou no dobro do tempo no Nemesis. Outros experimentos foram feitos para mostrar o comportamento do sistema sob alta carga, e estes demonstraram satisfazer todas as garantias de real-time que os testes necessitavam.

Análise do Paper Esse paper apresenta uma técnica bastante interessante para prover QoS para aplicações CM, ou continuous media. A idéia supera algumas limitações existentes em sistemas micro-kernel. Infelizmente a idéia envolve mexer nas aplicações também de modo que elas gerenciem as suas próprias falhas de páginas e não constitui uma solução que altere tão somente o sistema operacional. Demonstrou entretanto ser uma boa idéia com bons resultados.

O paper é bem escrito, mas, como é de esperar, supõe que o leitor já tenha um conhecimento prévio de sistemas operacionais e de conceitos avançados.

Não visualizamos aplicação direta, pelo menos no momento, dessa idéia no nosso trabalho de modificação do sistema de gerenciamento de memória do Linux. Mas é bastante interessante de modo a aumentar o nosso conhecimento geral do que se é pesquisado em termos gerais de SO, e nesse caso, também do gerenciamento de memória.

8 Conclusão

Durante todo esse segundo semestre do ano de 2001 foi desenvolvido um estudo intensivo sobre diferentes áreas relacionadas ao projeto de pesquisa de mestrado do aluno. O método escolhido de estudo foi a leitura e análise de diversos artigos bastante importantes, direta ou indiretamente, para o amadurecimento do aluno como também do projeto de pesquisa em si.

Os artigos foram divididos em diversas seções, especificamente *Cache Comprimido*, *Linux - Sistema de Gerenciamento de Memória*, *NetBSD - Sistema de Memória Virtual*, *MachOS* e *Tópicos Variados de VM*. A primeira seção contém, além de artigos, uma tese de doutorado que tem importância vital para o projeto.

Para relatar o estudo de cada um dos artigos, o método escolhido pelo aluno e o seu orientador foi a escrita de um resumo com uma pequena análise de cada artigo. Nesse relatório de estudos, todos os artigos como também a tese de doutorado da seção de *Cache Comprimido*, possuem um pequeno quadro informativo (autores, universidade, conferência, etc) juntamente do seu resumo e análise. Feito dessa forma, esse relatório será bastante importante para servir como base de referência a esses artigos para o aluno.

Ademais, todos artigos foram publicados em anais de conferências muito famosas no mundo de sistemas operacionais (ACM SOSP, USENIX Conferência Técnica e USENIX OSDI). Parte deles apresentou simulações ou detalhamento de implementações, o que certamente ajudará no conhecimento global do aluno sobre problemas enfrentados e soluções encontradas para eles.

Para concluir, achamos que foi uma experiência bastante enriquecedora para o aluno e com certeza ajudou a formar uma base mais concreta para a continuação do projeto de mestrado e o seu término com sucesso.