Gerência de Memória Adaptativa no XEN

Artur Baruchi, Edson Toshimi Midorikawa

Laboratório de Arquitetura e Computação de Alto Desempenho

Departamento de Engenharia de Computação e Sistemas Digitais

Escola Politécnica da Universidade de São Paulo – São Paulo – Brasil

{artur.baruchi,edson.midorikawa}@poli.usp.br

Abstract*. Many researches related to resource management in Virtual Machines are being performed. However, the proposals until now are very expensive, such as Live Migration, or have a prohibitive complexity, like resource managements based on application's response time. This work has the main objective to propose a simple mechanism for memory management and a better response time. Preliminary tests have shown memory save up to 54%.*

Resumo. *Muitos trabalhos relacionados ao gerenciamento de recursos em Máquinas Virtuais vêm sendo realizado. Entretanto, os trabalhos existentes até o momento possuem um custo muito alto, como no caso da Live Migration ou complexidade proibitiva, como ocorre em gerenciamento de recursos baseado em tempo de resposta de uma aplicação especifica. Este trabalho tem como principal objetivo propor um mecanismo de gerenciamento de memória simples de ser implementado e com melhor tempo de resposta. Testes preliminares demonstraram ganho de memória de até 54%.*

1. Introdução

Apesar do conceito das Máquinas Virtuais (MV) ser do início dos anos 70 [Goldberg, 1974], a virtualização começou a popularizar-se há pouco tempo. Esta popularidade é, em grande parte, devido ao barateamento de componentes de *Hardware*, como memória e com o advento de processadores *Multi-Core*. Com estes eventos, tornou-se bastante comum encontrar equipamentos com recursos subutilizados. Entre os fatores que ajudaram a virtualização tornar-se uma realidade, está a promessa de melhor utilizar recursos computacionais que, de outra forma, estavam sendo subutilizados dentro dos *Data Centers*.

Para que a virtualização seja implementada de forma correta, foram definidas algumas características que um ambiente virtualizado deve contemplar. Estas regras garantem que os recursos sejam divididos entre as MVs sem que uma sofra influência da outra. Estas características são as seguintes [Smith, J. E. e R. Nair, 2005]:

- **Isolamento:** Uma MV não pode influenciar no desempenho de outra MV;
- **Compatibilidade:** A MV deve ser completamente compatível com os padrões da plataforma que ela virtualizará (no caso do Xen, x86);
- **Encapsulamento:** Esta característica é o que torna a MV portável, pois viabiliza a cópia ou a movimentação de uma MV que está em execução de uma Máquina Física para outra;

- **Independência de *Hardware*:** As MVs devem ser independentes do *Hardware* em que elas estão em execução, por exemplo, é possível configurar interfaces de rede e outros dispositivos na MV que são completamente diferentes dos dispositivos físicos disponíveis;

Assim como estas características são necessárias para que a virtualização seja viável, estas podem limitar propostas para um gerenciamento de recurso mais elaborado e colaborativo. Desta forma, é um grande desafio manter todas as características citadas acima, que prezam por um isolamento do ambiente virtual e criar um canal de comunicação entre as MVs e o Sistema Operacional hospedeiro.

Neste trabalho é demonstrado um mecanismos de gerência de memória adaptativa, implementado no Monitor de Máquinas Virtuais (MMV) Xen. As próximas seções estão divididas da seguinte forma: na seção 2 são abordados alguns trabalhos relacionados à gerência de recursos e a comunicação entre as MV's e o MMV. A seção 3 tem como objetivo descrever a proposta e os critérios de alocação de memória. Alguns resultados, obtidos através da implementação de um protótipo, são apresentados na seção 4. Por fim, na seção 5, são abordados trabalhos futuros e a conclusão.

2. Trabalhos Relacionados

O *Difference Engine* (DE) [Gupta et. Al, 2008] tem como prinicipal objetivo implementar o *overcommit*[1] de memória. O primeiro passo executado pelo DE é o compartilhamento das páginas que possuem o mesmo conteúdo entre as MVs. Nas páginas que não puderam ser compartilhadas executa-se outro algoritmo para identificar páginas com conteúdo semelhante, as páginas que satisfazem os critérios desta busca têm um *patch* aplicado para que estas páginas sejam compartilhadas entre as MVs. A última técnica aplicada é a de compactação de páginas, cujo alvo são as páginas que possuem pouca probabilidade de uso no futuro (com base no algoritmo do LRU [O'Neil e Weikum, 1993]).

Outra proposta interessante de gerenciamento de memória é a implementada no VMWare ESX Server (ESX) [Waldspurger, 2002]. O ESX utiliza o compartilhamento de memória entre as MVs instaladas, permitindo o *overcommit* de memória. No ESX, um algoritmo gera um *hash* do conteúdo das páginas de memória e as páginas que contém o mesmo *hash* são compartilhadas. O problema da solução do ESX é que em ambientes com o *workload* heterogêneo o ganho de memória diminui substancialmente.

A Realocação de Recursos no Xen (RRX) [Zorzo et. al, 2008] é uma proposta que tem como objetivo realocar recursos dinamicamente entre MVs gerenciadas pelo Xen. O objetivo inicial desta proposta [Rossi, 2005] era fornecer ao administrador informações precisas de quantidade de memória e processador para a criação de uma MV que em condições normais é feito ao acaso e/ou de forma empírica. Esta forma de gerenciamento de recursos, além de ser complexa para a implementação é, na maioria das vezes, focada em apenas uma aplicação e não na MV como um todo.

[1] O *Overcommit* permite que a soma da memória das MVs seja maior que quantidade de memória física disponível.

3. Proposta de Gerenciamento de Memória

Nesta seção é apresentado o algoritmo que realiza o gerenciamento da memória entre as MV's. O algoritmo pode ser estendido para outros MMV's, entretanto, deve-se criar (ou usar alguma estrutura definida) para a troca de informações entre as MV's e o MMV. O Xen foi escolhido para efetuar os testes e a implementação do protótipo, pois possui seu código fonte aberto e já disponibiliza um mecanismo de memória compartilhada para a troca de informações entre as MV's.

O objetivo inicial da proposta é a implementação de uma forma de gerenciamento de memória que seja capaz de identificar as necessidades de cada uma das MV's instaladas e distribuir, de forma mais justa, a memória entre estas. Desta forma, uma MV que esteja utilizando pouca memória poderá doar memória para uma MV que possua maior necessidade.

Um desafio a ser transposto é manter o desempenho da MV aceitável. Para que a proposta fosse implementada sem causar sérios danos ao desempenho. Foi identificada a necessidade de monitorar, além da quantidade de memória disponível, a utilização de outros recursos como o processador[2]. Nos testes preliminares apresentados no neste trabalho, não foram analisados outros recursos como a taxa de I/O em disco e vazão de rede. O protótipo desenvolvido para este trabalho, foi implementado em Perl.

Para que haja troca de informações entre as MV's, foi utilizada uma estrutura de memória compartilhada disponibilizada pelo Xen, chamada de Xenstore [Chisnall, David, 2007]. Através desta estrutura, as MV's informam a tendência momentânea da utilização de memória e qual recurso está sendo onerado (memória ou processador). De posse destas informações, um *daemon* que fica em execução no Dom0 pode alocar ou retirar memória da MV. A quantidade de memória a ser adicionada ou retirada é calculada em função da quantidade e de qual recurso está escasso.

O *daemon* que fica execução na MV escreve no Xenstore, de forma periódica, a carga do processador dos últimos quinze, cinco e um minuto. Além destas informações, o *daemon* também escreve a quantidade de memória livre relativa. O *daemon* que fica em execução no Dom0, utiliza as informações escritas no Xenstore para verificar (i) se existe a necessidade de adicionar ou remover memória e (ii) uma vez que foi identificada a necessidade da MV, qual a quantidade de memória que pode ser removida ou adicionada.

A verificação da necessidade de adicionar ou remover memória da MV é calculado utilizando o valor da carga do processador no ultimo minuto. Já a quantidade de memória a ser adicionada ou removida é verificada através da quantidade de memória livre e da carga do processador nos últimos cinco e quinze minutos. Caso os valores extrapolem de forma acentuada alguns *threshoulds* a quantidade a ser adicionada ou removida de memória é maior, caso contrario, os valores extrapolem mas com uma diferença mínima, a quantidade de memória é menor. Este mecanismo foi necessário para que seja possível responder de forma mais rápida a necessidades mais criticas das MV's.

[2] Neste trabalho são mostrados resultados utilizando somente a carga do processador. Entretanto, foram executados testes, ainda em análise, utilizando o processamento de código do Kernel (sys).

4. Resultados Obtidos

Foram escolhidos três *benchmark*s com comportamentos diferentes para avaliar o simulador em situações de diversidade. O primeiro *benchmark* utilizado foi o Dbench, que simula um servidor Samba. No teste seguinte foi utilizado o *benchmark* Sysbench, que é capaz de realizar diversos testes, porém, dentre os testes disponibilizados apenas o teste que realiza transações no MySQL foi utilizado. O ultimo *benchmark* usado para aferir a proposta foi a compilação do *Kernel* do Linux.

A figura 1 mostra o resultado obtido com a execução do DBench. A variação do desempenho em função da quantidade de memória deve-se ao fato do mecanismo de *buffer* de disco do Linux. O Linux utiliza a memória livre do sistema para a realização do *buffering*. Por conta deste mecanismo, aplicações que realizam muitas operações de I/O se beneficiam de forma substancial com o aumento de memória. Esta característica pode ser claramente verificada no gráfico da figura 1.

O desempenho obtido com a proposta na execução do DBench foi bastante satisfatório. Em quase todos os casos a proposta foi superior ao caso intermediário, quando a MV estava configurada para utilizar 2 *GB* de memória. Com o aumento de memória dinamicamente, o Linux pode aumentar o *buffer* das operações de I/O e conseqüentemente melhorando o desempenho. Ao utilizar a proposta, os ganhos[3] de memória para este *benchmark* chegaram a 48%.

O *Sysbench* possui uma grande variedade de testes a disposição. Para este trabalho, foi utilizada somente a parte que realiza testes em bancos de dados transacionais. Como Sistema Gerenciador de Banco de Dados (S*GB*D) foi usado o MySQL, por ser amplamente usado na comunidade acadêmica e em corporações. O *Sysbench* executa uma variedade considerável de testes em uma base de dados previamente criada por ele. Estes testes são desde adições e alterações de dados a indexação da base de dados. Para este teste, foi usada uma tabela com 1 milhão de registros, de forma a manter a pressão sobre a memória em um nível alto.

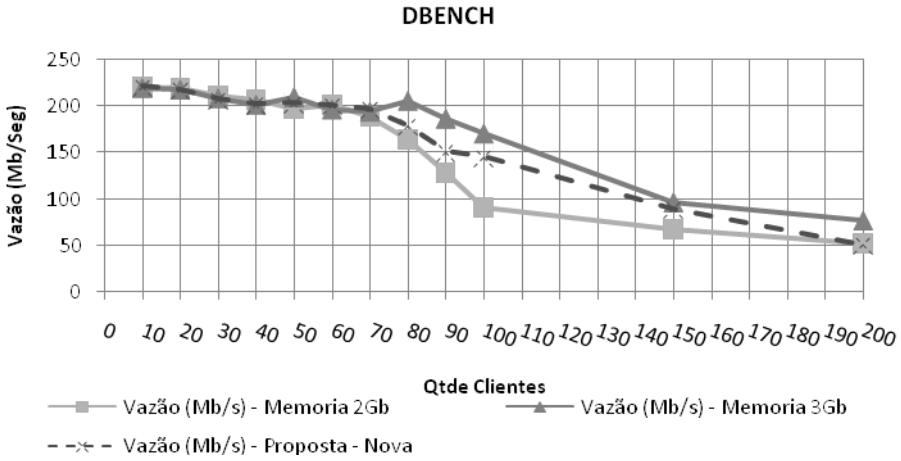

Figura 1. Execução do DBENCH

[3] Ganho de Memória é a quantidade de memória que foi economizada ao utilizar a proposta. Os ganhos de memória foram calculados em comparação com a memória fixa de 3GB.

Na figura 2 é apresentado o tempo total da execução do *benchmark*. Apesar do desempenho da proposta para este *benchmark* ter ficado ligeiramente abaixo da configuração intermediária (MV configurada com 2 *GB*), o ganho médio de memória foi de 54%.

Estes resultados se devem ao fato do MySQL ter seus próprios mecanismos de cache. Uma vez que o MySQL tenha alocado uma determinada quantidade de memória, esta será usada até o final da execução. Portanto o MySQL se beneficiará pouco com a alocação dinâmica de memória. Pode-se observar que mesmo com as configurações que usam memória fixa a diferença de desempenho não foi substancial para 2 *GB* e 3 *GB*.

O último *benchmark* executado foi a compilação do *Kernel*. Este teste é largamente usado na comparação de desempenho em ambientes computacionais. O *Kernel* do Linux é composto por mais de 10 milhões de linhas de código, desta forma a sua compilação é uma alternativa válida como *benchmark*.

Ao compilar o *Kernel*, o GCC alocará grandes porções de memória devido ao tamanho do *Kernel*, além de realizar muitos acessos a disco. A figura 3 ilustra o desempenho das configurações e da proposta para este *benchmark*. O desempenho da proposta foi satisfatório ao ficar abaixo da configuração intermediária.

Neste *benchmark* o ganho foi identificado nos momentos de compilação do *Kernel* em si. A parte em que o *Kernel* compila os módulos não há muito ganho pelo fato de serem códigos menores. Existem outros momentos durante a compilação que também se beneficiam como no instante da compressão do *Kernel* e em acessos a disco. O ganho é obtido pelo mecanismo de *buffer* descrito anteriormente. Com isto, ao utilizar a proposta o ganho de memória ficou em torno de 47% neste *benchmark*.

O ganho de memória foi bastante significativo na execução dos três benchmarks. A tabela 1, apresenta a alocação média de memória nas MVs e o ganho relativo de memória nos três *benchmarks*. Como pode ser observado, a MV ficou a maior parte do tempo com uma quantidade de memória bem abaixo do limite máximo e recebeu porções de memória maiores somente quando foi identificada esta necessidade.

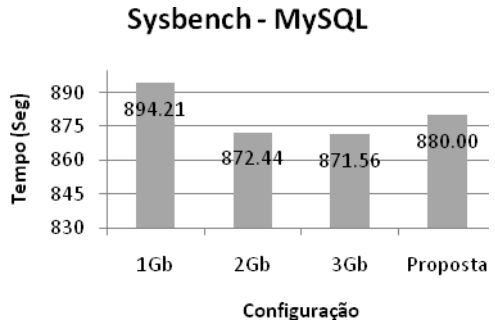

Figura 2. Execução do Sysbench

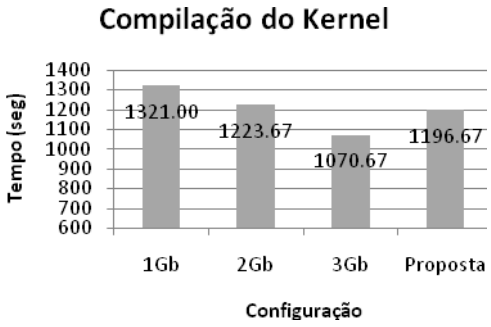

Figura 3. Compilação do *Kernel* do Linux

Tabela 1. Média de Quantidade de Memória Alocada e Ganho de Memória

Benchmark		*Memória Alocada (MB)*	*Ganho (%)*
Dbench	10 Clientes	1580 (±271)	48%
	150 Clientes	1892 (±691)	38%
	200 Clientes	2378 (±746)	22%
Sysbench		1404 (±122)	54%
Kernel		1607 (±500)	47%

5.Conclusões e Trabalhos Futuros

Apesar de a proposta apresentar melhores resultados em ambientes com um comportamento bem definido, os testes mostraram que em ambientes estáveis a alocação de memória é feita de forma satisfatória. Outro ponto importante verificado é o comportamento da proposta em momentos de picos de utilização. Devido ao mecanismo de carência entre uma alocação e outra, que varia de acordo com a falta de memória e quantidade de memória inserida na ultima alocação, cria-se uma forma de proteger a MV contra situações de falso positivo.

A contribuição deste trabalho foi demonstrar uma técnica de gerenciamento de memória que se adapta de acordo com as necessidades das MVs. O ganho de memória foi bastante significativo (de 22~54%) e o desempenho das MVs sofreu pouca degradação ao utilizar a proposta, o que tornam estes resultados encorajadores.

Como trabalho futuro, pretende-se analisar a proposta utilizando outros mecanismos para a identificação da tendência do uso de memória momentâneo no ambiente. Alguns testes realizados mostraram que o uso quantidade de processamento em espaço de kernel (sys) é um parâmetro que apresentou resultados satisfatórios até o momento. Já a implementação do protótipo em linguagem C está bastante avançada e alguns resultados observados apresentaram melhora no tempo de resposta da proposta.

7. Referências

Chisnall, David (2007). “The Definitive Guide to the Xen Hypervisor”, Prentice Hall, 1st Edition.

Goldberg, Robert P. (1974). "Survey of Virtual Machine Research". IEEE Computer. pp. 34-45.

Gupta, Diwaker et. al (2008). "Difference Engine: Harnessing Memory Redundancy in Virtual Machines". Proceedings of the 8th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI). San Diego.

O'NEIL, E. J.; O'Neil, P. E.; Weikum, G (1993). "The LRU-K Page Replacement Algorithm for Database Disk Buffering". Proc. ACM SIGMOD Conf. pp. 297-306.

Rossi, F. D. (2005). "Alocação dinâmica de Recursos no Xen". Master's thesis, Dissertação (Mestrado em Ciência da Computação) - Faculdade de Informática - PUCRS, Porto Alegre.

Smith, J. E. e R. Nair (2005). “Virtual Machines: Versatile Platforms for Systems and Processes". Morgan Kaufmann. 1st Edition.

Waldspurger, C. A. (2002). "Memory Resource Management in VMWare ESX Server". Proc. of the 5th USENIX OSDI.

Zorzo, Avelino et. al (2008). "Uso de Modelos Preditivos e SLAs para Reconfiguracão de Ambientes Virtualizados". V Workshop de Sistemas Operacionais (WSO'2008). pp. 147-158.