ARTUR BARUCHI

Memory Dispatcher: Uma Contribuição para a
Gerência de Recursos em Ambientes Virtualizados

São Paulo
2010

ARTUR BARUCHI

Memory Dispatcher: Uma Contribuição para a
Gerência de Recursos em Ambientes Virtualizados

Dissertação apresentada à Escola
Politécnica da Universidade de São
Paulo para obtenção do título de
Mestre em Engenharia Elétrica

São Paulo
2010

ARTUR BARUCHI

Memory Dispatcher: Uma Contribuição para a
Gerência de Recursos em Ambientes Virtualizados

Dissertação apresentada à Escola
Politécnica da Universidade de São
Paulo para obtenção do título de
Mestre em Engenharia Elétrica

Área de Concentração:
Sistemas Digitais

Orientador: Prof. Doutor
Edson Toshimi Midorikawa

São Paulo
2010

DEDICATÓRIA

Dedico este trabalho à minha esposa Camila que soube tão bem compreender os meus momentos de ausência em função deste trabalho e aos meus pais.

AGRADECIMENTOS

Agradeço à minha família pela educação e confiança que depositaram em mim.

Agradeço ao meu orientador, professor Edson Midorikawa, por me ajudar nos momentos mais críticos do mestrado e também pela amizade construída no decorrer dos anos.

Agradeço aos meus amigos Ricardo, Denis e Juliano pelas inúmeras contribuições feitas no decorrer do trabalho.

Agradeço as empresas por onde passei durante esse trabalho que permitiram a conciliação entre os estudos e o trabalho.

Agradeço aos professores e amigos do LAHPC pelas contribuições diretas e indiretas e pelo excelente ambiente de trabalho.

RESUMO

As Máquinas Virtuais ganharam grande importância com o advento de processadores *multi-core* (na plataforma x86) e com o barateamento de componentes de *hardware*, como a memória. Por conta desse substancial aumento do poder computacional, surgiu o desafio de tirar proveito dos recursos ociosos encontrados nos ambientes corporativos, cada vez mais populados por equipamentos *multi-core* e com vários Gigabytes de memória. A virtualização, mesmo sendo um conceito já antigo, tornou-se novamente popular neste cenário, pois com ela foi possível utilizar melhor os recursos computacionais, agora abundantes.

Este trabalho tem como principal foco estudar algumas das principais técnicas de gerência de recursos computacionais em ambientes virtualizados. Apesar de muitos dos conceitos aplicados nos projetos de Monitores de Máquinas Virtuais terem sido portados de Sistemas Operacionais convencionais com pouca, ou nenhuma, alteração; alguns dos recursos ainda são difíceis de virtualizar com eficiência devido a paradigmas herdados desses mesmos Sistemas Operacionais.

Por fim, é apresentado o *Memory Dispatcher* (MD), um mecanismo de gerenciamento de memória, com o objetivo principal de distribuir a memória entre as Máquinas Virtuais de modo mais eficaz. Este mecanismo, implementado em C, foi testado no Monitor de Máquinas Virtuais Xen e apresentou ganhos de memória de até 70%.

Palavras-chave: Máquinas Virtuais, Virtualização, Monitor de Máquinas Virtuais, Gerência de Recursos.

ABSTRACT

Virtual Machines have gained great importance with advent of multi-core processors (on platform x86) and with low cost of hardware parts, like physical memory. Due to this computational power improvement a new challenge to take advantage of idle resources has been created. The virtualization technology, even being an old concept became popular in research centers and corporations. With this technology idle resources now can be exploited.

This work has the objective to show the main techniques to manage computational resources in virtual environments. Although many of current concepts used in Virtual Machine Monitor's project has been ported, with minimal changes, from conventional Operating Systems there are some resources that are difficult to virtualize with efficiency due to old paradigms still present in Operating System's projects.

Finally, the Memory Dispatcher (MD) is presented, a mechanism used to memory management. The main objective of MD is to improve the memory share among Virtual Machines. This mechanism was developed in C and it was tested in Xen Virtual Machine Monitor. The MD showed memory gains up to 70%.

Keywords: Virtual Machines, Virtualization, Virtual Machine Monitor, Operating System.

LISTA DE ILUSTRAÇÕES

Figura 1 – Teorema de Popek e Goldberg. (a) Não satisfaz a condição (b) Satisfaz a condição (Smith, 2005).	18
Figura 2– Isomorfismo entre o Sistema Convidado e Hospedeiro (Smith, 2005).	21
Figura 3 – MVs do (a) Tipo I e (b) Tipo II (Laureano, 2006).	22
Figura 4 – Abordagem Híbrida empregada no (a) Tipo I e (b) Tipo II (Laureano, 2006).	23
Figura 5 – Taxonomia de acordo com a camada ISA (Smith, 2005).	24
Figura 6 – Estrutura do I/O Ring (Barham, 2003).	28
Figura 7 – Interação entre o I/O Ring e Device Driver.	29
Figura 8 – <i>Kernel</i> Do Linux como MMV – Proposta do KVM (Kivity, 2007).	31
Figura 9 – Arquitetura do PowerVM com LPAR e Micro-Particionamento (Hales, 2008).	32
Figura 10 – Relação entre Unidade de Processamento e Processador Virtual (Hales, 2008).	38
Figura 11 – Funcionamento do Balloon Driver (Waldspurger, 2002).	40
Figura 12 – Técnica de Compartilhamento de Memória Baseada em Conteúdo (Waldspurger, 2002).	41
Figura 13 – Funcionamento do Virtual I/O Server (Hales, 2008).	43
Figura 14 – Funcionamento do Virtual I/O Server (Hales, 2008).	44
Figura 15– Processo de Realocação de Recursos no Xen (Rodrigues, 2009).	46

Figura 16– (a) Provisão para Picos (b) e (c) Provisão Média (Armbrust, 2010)	47
Figura 17– Desempenho do Netperf com o Dom0 configurado com prioridade 25% maior que as MV's (Baruchi, 2008).	51
Figura 18– Desempenho do Netperf com a MV configurada com prioridade 25% maior que o Dom0 (Baruchi, 2008).	51
Figura 19– Desempenho do ambiente ao alterar a configuração de memória (Baruchi, 2009).	53
Figura 20– Arquitetura do <i>Memory Dispatcher</i>	54
Figura 21– Algoritmo do MD. (a) Parte executada na MV. (b) Parte executada no Dom0.	55
Figura 22– Gráfico com o desempenho do DBench – Configuração I / MV01.	65
Figura 23– Gráfico com o desempenho do DBench – Configuração I / MV02.	66
Figura 24– Gráfico com o desempenho do DBench – Configuração II / MV01.	67
Figura 25– Gráfico com o desempenho do DBench – Configuração II / MV02.	68
Figura 26– Gráfico com o desempenho do DBench – Configuração II / MV03.	68
Figura 27– Ganho de Memória nas MVs – Configuração I.	70
Figura 28– Ganho de Memória nas MVs – Configuração II.	70
Figura 29– Desempenho da Compilação do <i>Kernel</i> – Configuração I.....	71
Figura 30– Desempenho da Compilação do <i>Kernel</i> – Configuração II.....	72
Figura 31– Ganho de memória – Compilação do <i>Kernel</i>	73
Figura 32– Desempenho da Compilação do <i>Kernel</i> – MV01 / Configuração I.	74

Figura 33– Desempenho do DBench – MV02 / Configuração I.....	75
Figura 34– Desempenho do Httpperf – MV01 / Configuração II.....	76
Figura 35– Desempenho da Compilação do <i>Kernel</i> – MV02 / Configuração II.	77
Figura 36– Desempenho do Dbench – MV03 / Configuração II.	77
Figura 37– Ganho de memória – Configuração II – MV01 / MV02.....	78
Figura 38– Ganho de memória – Configuração II – MV03.	79
Figura 39– Ilustração do cálculo da demanda de memória.	80
Figura 40– Alocação de Memória Fixa (2666MB).	81
Figura 41– Alocação de Memória utilizando o MD.	81
Figura 42– Demanda de Memória – Compilação do <i>Kernel</i>	82
Figura 43– Demanda de Memória – Compilação do <i>Kernel</i>	83
Figura 44– Demanda de Memória – DBench.	84
Figura 45– Demanda de Memória – DBench.	84
Figura 46– Demanda de Memória – DBench.	85
Figura 47– Demanda de Memória – DBench.	86
Figura 48– Demanda de Memória – DBench.	86

LISTA DE TABELAS

Tabela 1– Benchmarks executados e seus comportamentos.....	61
Tabela 2– Configuração das Máquinas Reais	62
Tabela 3– Configuração das Máquinas Virtuais	63
Tabela 4– Configuração das Máquinas Virtuais	63
Tabela 5 – Desempenho obtido com o DBench na Config.I / MV01	66
Tabela 6 – Desempenho obtido com o DBench na Config.I / MV01	67
Tabela 7– Desempenho obtido com o DBench na Config.II / MV01	69
Tabela 8– Desempenho obtido com o DBench na Config.II / MV01	69
Tabela 9– Desempenho obtido com o DBench na Config.II / MV01	69
Tabela 10– Desempenho obtido com o DBench na Config.II / MV01	72
Tabela 11– Desempenho obtido com a compilação do <i>Kernel</i> – Config II.....	73
Tabela 12– Desempenho obtido com o DBench na configuração I / MV02.....	75
Tabela 13– Desempenho com a compilação do <i>Kernel</i> na configuração I / MV01.....	75
Tabela 14– Desempenho do HTTPERF e Comp do <i>Kernel</i> – Configuração II.	78
Tabela 15– Desempenho Dbench – Configuração II / MV03.....	78

LISTA DE ABREVIATURAS E SIGLAS

<i>Benchmark</i>	Ponto de referência de uma medida
<i>Core</i>	Núcleo físico de um processador
CPU	<i>Central Processing Unit</i> (Unidade Central de Processamento)
<i>Hash</i>	Sequência de Bits gerada por um algoritmo de dispersão
<i>Hyper Call</i>	Chamada a rotinas privilegiadas do MMV
I/O	<i>Input/Output</i> (Entrada e Saída)
ISA	<i>Instruction Set Architecture</i>
<i>Linker</i>	Combina um ou mais objetos compilados em um único arquivo executável
<i>Load</i>	Unidade de medida para a quantidade de trabalho em um sistema computacional
<i>Loader</i>	Carrega um arquivo executável em memória
MD	<i>Memory Dispatcher</i>
MMU	<i>Memory Management Unit</i>
MMV	Monitor de Máquinas Virtuais
MV	Máquina Virtual
MTU	<i>Maximum Transmission Unit</i>
<i>Overhead</i>	Custo extra para execução de um processo
SO	Sistema Operacional
<i>System Call</i>	Chamada a rotinas privilegiadas do Sistema Operacional
<i>Threshold</i>	Limite superior ou inferior para que uma ação seja tomada
<i>Time Share</i>	Tempo que um determinado processo tem para usar o processador
TLB	<i>Translation Lookaside Buffer</i>
VCPU	<i>Virtual Central Processing Unit</i>
<i>Workload</i>	Carga de Trabalho

SUMÁRIO

1	INTRODUÇÃO.....	13
1.1	Objetivos	14
1.2	Metodologia	15
1.3	Contribuições	15
1.4	Estruturas dos Capítulos	16
2	CONCEITOS DE VIRTUALIZAÇÃO	17
2.1	Virtualização Na Plataforma x86	17
2.1.1.	<i>Virtualização e Emulação.....</i>	<i>19</i>
2.2	Taxonomia.....	21
2.2.1.	<i>Camada do MMV.....</i>	<i>22</i>
2.2.2.	<i>Camada ISA.....</i>	<i>24</i>
2.3	Monitores De Máquinas Virtuais	27
2.3.1.	<i>XEN .....</i>	<i>27</i>
2.3.2.	<i>VMWare.....</i>	<i>30</i>
2.3.3.	<i>Kernel Virtual Machine (KVM).....</i>	<i>31</i>
2.3.4.	<i>PowerVM (IBM SystemP Virtualization) .....</i>	<i>32</i>
3	GERÊNCIA DE RECURSOS EM MÁQUINAS VIRTUAIS.....	34
3.1	Gerência do Processador	34
3.2	Gerência da Memória.....	39
3.3	Gerência de Dispositivos de I/O.....	42

3.4	Trabalhos Relacionados.....	44
4	MEMORY DISPATCHER	47
4.1	Descrição da Proposta	47
4.2	Pesquisas Preliminares.....	48
4.2.1.	<i>Efeito da Prioridade no Desempenho .....</i>	<i>49</i>
4.2.2.	<i>Efeito da Alocação de Memória no Desempenho .....</i>	<i>52</i>
4.3	Arquitetura do <i>Memory Dispatcher</i>	53
4.4	Algoritmos	55
4.5	Implementação	57
5	RESULTADOS.....	60
5.1	Metodologia e Configuração	60
5.2	Resultados Obtidos.....	64
5.2.1.	<i>Caso de Teste A.....</i>	<i>65</i>
5.2.2.	<i>Caso de Teste B .....</i>	<i>70</i>
5.2.3.	<i>Caso de Teste C .....</i>	<i>74</i>
5.3	Demanda de Memória.....	79
5.4	Sobrecarga	87
6	CONCLUSÕES.....	88
6.1	Contribuições	89
6.2	Trabalhos Futuros	90
	REFERÊNCIAS.....	92

1 INTRODUÇÃO

Apesar do conceito das Máquinas Virtuais (MV) serem do final dos anos 60 (Goldberg, 1974), a virtualização ganhou muita força com o advento de processadores *multicores* e com o barateamento de componentes como a memória. O principal objetivo da virtualização é melhorar o gerenciamento de recursos que se tornaram subutilizados. Diversas propostas de virtualização para a plataforma x86 podem ser encontradas na Internet e na literatura, algumas já consolidadas como o VMWare e Xen e outras ainda em processo de consolidação, como o KVM.

Apesar da virtualização não ser um conceito novo, a virtualização na plataforma x86 é mais recente. A virtualização na plataforma x86 é mais complexa e de acordo com a teoria de Popek e Goldberg (Smith, 2005) não é uma plataforma que suporta, de forma nativa, a virtualização. Para que este problema fosse contornado, algumas propostas, como a paravirtualização (Whitaker, 2002), foram concebidas e mostraram-se muito eficientes.

A virtualização apresenta-se como uma tecnologia com grande aceitação no mundo corporativo. Os maiores fornecedores de TI contam com alguma solução de virtualização em seus portfólios. Vários casos de sucesso na consolidação de *data centers* podem ser encontrados na Internet, validando os benefícios que a virtualização é capaz de trazer.

Para que os objetivos da virtualização sejam atingidos de forma satisfatória, foram definidas algumas características que um ambiente virtual deve possuir para atingir estes objetivos. Estas características são as seguintes:

- **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, i386);
- **Encapsulamento:** Esta característica é o que torna a MV portátil, 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 essas características são necessárias para que a virtualização seja viável, também 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.

1.1 Objetivos

O objetivo deste trabalho é analisar técnicas de gerenciamento de recursos utilizadas em Monitores de Máquinas Virtuais (MMVs). A análise tem como objetivo verificar possíveis pontos de melhoria e viabilizar a criação do *Memory Dispatcher* (MD), que é um mecanismo para gerenciamento de memória colaborativo.

O MD tem como principal objetivo melhorar a distribuição da memória entre as MVs. Diferente do que ocorre com o processador, a memória alocada a uma MV é geralmente constante desde a sua inicialização até o término da execução. Desse modo, se a quantidade de memória for superestimada, haverá desperdício de recurso; ou se subestimada, poderá ocasionar danos maiores ao desempenho do sistema. O mecanismo de MD, implementado no Xen, visa a melhoria da distribuição da memória, procurando alocar a quantidade de memória que cada MV necessita em um dado momento.

1.2 Metodologia

Para este trabalho, um mecanismo de gerência de memória chamado *Memory Dispatcher* foi implementado . Este mecanismo foi desenvolvido em C e tem como principal objetivo melhorar a distribuição da memória física entre as MVs. Para a avaliação do MD, três *benchmarks* foram executados com comportamentos diferentes em duas máquinas com configurações distintas.

Durante os testes, informações foram coletadas da memória alocada, memória usada e a taxa de falta de páginas por segundo. Com esses dados, foi possível verificar se o MD estava realizando as alocações de memória de forma correta e também quantificar o ganho de memória que o MD proporcionou para os ambientes virtualizados.

1.3 Contribuições

A principal contribuição deste trabalho é propor uma estratégia de gerenciamento de memória concebida após a análise das técnicas de gerência de recursos utilizadas por MMVs chamada de *Memory Dispatcher* (MD). O MD tem como objetivo melhorar a forma como a memória física é distribuída entre as MVs.

A melhor distribuição de recursos, como a memória, torna a virtualização mais eficiente e possibilita que um mesmo ambiente físico consiga comportar mais MVs. Para que este objetivo pudesse ser alcançado, o MD aloca apenas a quantidade necessária de memória para cada MV e tem como principal critério de alocação a quantidade de falta de páginas das MVs.

1.4 Estruturas dos Capítulos

O capítulo 2 trata de conceitos básicos da virtualização. Este capítulo também descreve algumas estruturas internas do Xen, que serão utilizadas para a implementação do *Memory Dispatcher*. No capítulo três, conceitos de gerência de recursos em MVs são abordados. Este capítulo tem como objetivo demonstrar as diferentes técnicas existentes para o gerenciamento de CPU, Memória e outros dispositivos de I/O.

O capítulo quatro demonstrará os critérios de alocação de memória e também estruturas internas do *Memory Dispatcher*. Neste capítulo também é abordado como ocorre a comunicação entre as MVs e o Sistema Operacional convidado para que a proposta funcione corretamente.

O capítulo cinco contém os resultados e as análises do *Memory Dispatcher*. Esse capítulo contém comparações que foram realizadas com um ambiente virtual convencional e gráficos obtidos a partir dos dados coletados durante os testes com os *benchmarks*.

2 CONCEITOS DE VIRTUALIZAÇÃO

Este capítulo abordará alguns conceitos de virtualização necessários para o entendimento do *Memory Dispatcher* (MD) e das análises realizadas no próximo capítulo. A taxonomia da virtualização será apresentada, que tem como principal referência a camada na qual a referência acontece.

2.1 Virtualização Na Plataforma x86

A virtualização é um conceito que surgiu no final da década de 60 (Goldberg, 1974), foi bastante estudada e utilizada por um curto período após seu surgimento. Nas décadas de 80 e 90, com o advento de Sistemas Operacionais multi-tarefas e um considerável barateamento do hardware, a tecnologia da virtualização foi depreciada. Com os *mainframes* dando espaço a máquinas menores, a virtualização, tanto na academia como na indústria, era lembrada apenas como um fato histórico (Rosenblum, 2005).

A plataforma x86 popularizou-se no período em que a virtualização não tinha mais todo o prestígio de outrora. Devido a este fato, essa plataforma não foi concebida para suportar a virtualização de forma nativa. Por duas décadas, a Intel e outros fornecedores acertaram ao não integrar suporte à virtualização em seus projetos de processadores x86.

Entretanto, no início do século XXI, a virtualização começou a ganhar força novamente, ironicamente, pelos mesmos motivos que a fizeram ser esquecida por duas décadas antes – novas funcionalidades dos Sistemas Operacionais modernos e *hardware* mais barato (Rosenblum, 2005). Além destes eventos, surgiram novas tecnologias que antes eram comuns apenas no parque computacional de grandes empresas e universidades (como o advento de processadores *multi-core* e discos de *storage* chegando aos *terabytes* de capacidade de armazenamento).

Esses eventos ocorridos simultaneamente tornaram os equipamentos que utilizam processadores x86 (computadores pessoais e servidores de menor porte) consideravelmente potentes e com recursos ociosos. Neste cenário, fez-se necessário o desenvolvimento de alguma tecnologia que fosse capaz de melhorar a utilização desses equipamentos. A virtualização começaria a ressurgir neste ambiente de grande disponibilidade de recursos computacionais e necessidade de utilizá-lo com maior eficiência. Entretanto, a virtualização da plataforma x86 carecia de alguns artifícios para ser virtualizada de forma eficiente.

Segundo o teorema de Popek e Goldberg (Smith, 2005), uma plataforma somente é virtualizada de forma eficiente se todas as suas instruções sensíveis forem privilegiadas. Isto é, uma instrução sensível se for executada pelo processador em modo não-privilegiado deve, invariavelmente, gerar uma interrupção (ou *trap*). Desta forma, a implementação de um Monitor de Máquinas Virtuais (MMV) poderia ser menos complexa. Entretanto, a plataforma x86 não satisfaz a condição do teorema, o que forçou os pesquisadores a encontrar novas soluções para contornar este problema.

Figura 1 – Teorema de Popek e Goldberg. (a) Não satisfaz a condição (b) Satisfaz a condição (Smith, 2005).

Para suprir essa deficiência da plataforma x86, algumas técnicas foram desenvolvidas. As técnicas mais difundidas para virtualizar a plataforma x86 é a paravirtualização e a Tradução Binária. A paravirtualização consiste na alteração do código fonte do Sistema Operacional que será instalado na MV. Essa alteração faz com que o Sistema Operacional instalado na MV tenha ciência de que está sendo virtualizado. Já a Tradução Binária altera algumas instruções privilegiadas antes de serem executadas (encontradas por meio de uma pré-análise do código). A alteração pode ser a troca da instrução por outra equivalente, mas não privilegiada ou a inserção de *flags* para que o MMV saiba que aquela instrução é privilegiada. Ambas as técnicas serão abordadas com mais detalhes a frente.

2.1.1. Virtualização e Emulação

Os sistemas computacionais, em geral, são extremamente complexos e para lidar com essa complexidade, os projetistas desenvolveram uma forma de lidar com a complexidade através de camadas de abstração e interfaces. A abstração permite que detalhes da implementação de um determinado componente ou *software* sejam encobertos de camadas superiores. A abstração é interessante, pois por meio dela os projetistas das outras camadas não precisam ter o conhecimento técnico do funcionamento interno de componentes de hardware ou de outros softwares. Por exemplo, um desenvolvedor não precisa ter o conhecimento do funcionamento de um *Hard Disk* para poder abrir um simples arquivo. Para isto, basta saber qual chamada de sistema (*System Call*) do Sistema Operacional para realizar a tarefa.

A comunicação entre as camadas de abstração é feita através das interfaces, desta forma, os projetistas de camadas mais elevadas devem ter o conhecimento somente da interface e como utilizá-la. Isto facilita tanto para os projetistas que utilizaram um determinado recurso como para os projetistas que desenvolvem e mantêm o recurso. As alterações realizadas em uma determinada camada são imperceptíveis para os que a utilizam desde que a interface não tenha sido alterada. O conjunto de instrução (*instruction set*) é um tipo de interface, pois os projetistas da Intel

e da AMD constroem processadores que implementam o conjunto de instruções Intel IA-32 (ou x86) e por sua vez, os engenheiros de software da Sun ou da Microsoft constroem compiladores que mapeiam linguagens de alto nível para o mesmo conjunto de instruções IA-32. Desde que os projetistas e os engenheiros satisfaçam todas as especificações os programas compilados executarão sem problemas em qualquer máquina que tenha como processador a arquitetura IA-32.

Apesar da abstração e das interfaces possuírem diversos benefícios, um dos problemas observados com essa solução foi a criação de uma grande variedade de tipos de interfaces, por exemplo, existem diversos processadores com diversos conjuntos de instruções que não são compatíveis entre si. Outros tipos de interfaces incompatíveis são as chamadas de sistemas dos Sistemas Operacionais, um programa implementado para Windows não poderá ser executado no Linux, pois as chamadas de sistemas são totalmente diferentes.

Foi neste cenário de heterogeneidade que a virtualização foi concebida. Em primeira instancia a virtualização foi implementada para amenizar esta situação de grande variedade e aumentar a flexibilidade. Quando um determinado componente ou sistema (processador, memória, disco, etc) é virtualizado as interfaces deste componente e todos os recursos visíveis através deste recurso são mapeados para a interface real do componente. Como consequência deste mapeamento, o recurso real é apresentado aos seus consumidores (ou usuários) como um recurso diferente do que ele realmente é. Portanto, a virtualização é a construção de um isomorfismo que mapeia um sistema convidado virtual para um sistema real. A ilustração do isomorfismo pode ser observada na figura 2.

A emulação difere da virtualização neste ponto. Enquanto a emulação é a implementação em software de um determinado processador ou recurso a virtualização cria uma outra camada de abstração entre o recurso real e os seus consumidores de tal forma que este pareça ser diferente. A virtualização possui melhor desempenho quando comparada com a emulação, entretanto, alguns MMVs como o Virtual Box e o VMWare utilizam alguns dispositivos emulados para aumentar a oferta de hardware aos seus ambientes virtualizados.

Figura 2– Isomorfismo entre o Sistema Convidado e Hospedeiro (Smith, 2005).

2.2 Taxonomia

Apesar de ser uma tecnologia bastante difundida atualmente, a virtualização é um conceito do final dos anos 60 (Goldberg, 1974) e comercializado pela primeira vez no IBM/370. Nos dias atuais, existem diversas técnicas de virtualização que podem ser classificadas de varias formas, duas formas de classificação serão apresentadas neste trabalho:

- De acordo com a camada que o Monitor de MVs é implementado;
- De acordo com a camada ISA;

A técnica de virtualização empregada é um fator determinante para diversos fatores como, desempenho, escalabilidade, portabilidade e isolamento. No decorrer deste capítulo, as taxionomias das MVs e alguns representantes de cada uma das classificações serão apresentados com maiores detalhes.

2.2.1. Camada do MMV

Esta forma de classificação de MVs é bastante comum na literatura por ser uma taxionomia bastante genérica e que engloba praticamente todas as técnicas de virtualização disponíveis. De acordo com esta classificação, existem dois tipos de virtualização, o tipo I e o tipo II. Pode existir ainda uma terceira forma denominada Híbrida que utiliza algumas técnicas de otimização ao implementar o tipo I ou II.

- Tipo I: Neste tipo de virtualização, o MMV se situa entre o Hardware e a MV (ou sistema convidado);
- Tipo II: Neste tipo de virtualização, o MMV é implementado como um processo comum no Sistema Operacional (SO) anfitrião;
- Tipo Híbrida: Em linhas gerais, o tipo Híbrida pode ser uma MV do tipo I ou do tipo II, porém possui técnicas de otimizações.

As figuras 3a e 3b ilustram como essas duas classificações de virtualização ocorrem.

Figura 3 – MVs do (a) Tipo I e (b) Tipo II (Laureano, 2006).

Nas MVs do Tipo I, o MMV possui o controle sobre o Hardware, acima dele são criadas as MVs que tem o comportamento de uma Máquina Real e podem executar seu próprio SO. Como exemplo para este tipo de virtualização, pode ser citado o VMWare Server e o Xen.

As MVs do Tipo II são implementadas como um processo comum em um SO anfitrião. Neste tipo de virtualização, existe uma camada a mais entre o MMV e o Hardware, que é justamente o SO anfitrião. Como exemplos deste tipo de MV, o Virtual Box e o Bochs podem ser citados.

A abordagem Híbrida é, na verdade, a forma de virtualização mais usada. Apesar do nome Híbrido, este tipo de virtualização não é uma combinação das duas técnicas anteriores e sim devido às oportunidades de otimização encontradas nos tipos I e II. Por exemplo, em algumas virtualizações do tipo II, determinadas operações de I/O podem ser implementadas acessando diretamente o Hardware por meio do Sistema Anfitrião e no caso do tipo I, o Sistema Convidado acessar diretamente o Hardware sem passar pelo MMV. Na figura 4a e 4b, as principais otimizações nos dois tipos de virtualização são apresentadas.

Figura 4 – Abordagem Híbrida empregada no (a) Tipo I e (b) Tipo II (Laureano, 2006).

Na figura 4, as setas maiores indicam as otimizações feitas nos tipos de virtualização. Estas otimizações, em geral, são realizadas por acesso direto a alguma das camadas que se situam abaixo sem ter que passar por camadas intermediárias. Existem outras formas de melhorar o desempenho das MVs, porém operações de I/O são os principais gargalos da virtualização (Fraser, 2005).

2.2.2. Camada ISA

A segunda forma de classificação das MVs é a que está de acordo com a camada ISA. Esta taxonomia é utilizada tanto para MVs de sistemas como para MVs de processos (como Java Virtual Machine, JVM (Esteire, 1998)). Nesta classificação, as MVs são divididas primeiramente em MV de sistema ou processo em seguida são classificadas de acordo com a ISA, isto é, se a MV possui emulação da camada ISA ou não.

Figura 5 – Taxonomia de acordo com a camada ISA (Smith, 2005).

Nessa taxonomia, a MV do tipo I é chamada de MV Clássica (*Classic System*) e a MV do tipo II é chamada de MV Convidada (*Hosted VM*). Como se pode observar na figura 5, a MV do tipo I e II está na mesma classe, que possuem a mesma camada ISA. Na classe em que a camada ISA do Hospedeiro e da MV Convidada são diferentes, estão as MVs completamente virtualizadas e as MVs Coprojetada.

- *MVs Completamente Virtualizadas*: Todo o sistema e as aplicações dependem de uma determinada camada ISA para poderem funcionar (por exemplo, Windows PCs e Apple PowerPC). Nessas situações, a MV deve emular tanto o código da aplicação como o do SO. Exemplo desse tipo de virtualização é o Virtual PC¹ da Microsoft.
- *MVs Coprojetada*: Este tipo de virtualização costuma implementar um tipo de camada ISA proprietária para melhorar o desempenho, eficiência na utilização de energia ou ambos. Uma MV coprojetada não possui aplicações ISA nativas, ao contrário, o MMV se apresenta como parte da implementação do Hardware. Sua principal função é emular a camada ISA da MV Convidada. Um exemplo deste tipo de virtualização é o Transmeta Crusoe (Dehnert, 2003).

Um exemplo interessante de virtualização em que o Hospedeiro e a MV possuem a camada ISA diferentes é o Simics (Magnusson, 1995). O Simics é um software que virtualiza por completo uma determinada plataforma. É possível executar binários compilados para o PowerPC em um ambiente x86. Além do processador, o Simics pode virtualizar device drivers da plataforma alvo possibilitando a execução de Sistemas Operacionais como AIX ou HP-UX em x86.

Em princípio, o Simics foi desenvolvido para prover uma plataforma de desenvolvimento completa para diferentes arquiteturas. É possível gerar *traces* de um programa em execução que não seriam possíveis em um ambiente real. Entre outras

---

¹ Virtual PC: <http://www.microsoft.com/windows/virtualpc/default.mspx>

características, pode-se alterar o comportamento da plataforma alvo, como por exemplo, adicionar *traps* ou exceções no processador para estudar seu comportamento em determinadas situações. A virtualização de plataformas *multicore*, multiprocessada e clusters também são outro atrativo do Simics.

As Máquinas Virtuais de Processo proveem um ambiente de Interface de Aplicação Binária (ABI – *Application Binary Interface*). Um tipo de MV de processo bastante comum são as MVs de Linguagem de Alto Nível. Estas MVs têm como objetivo minimizar as dependências de características específicas de Hardware e de Sistemas Operacionais com a finalidade de criar interoperabilidade entre plataformas.

Estas MVs se tornaram populares com o ambiente de programação Pascal (Bowles, 1980). Em um ambiente convencional, o compilador consiste em um *front end* que realiza análise lexical, sintática e semântica para gerar código intermediário, bastante parecido com o código de máquina. O código de máquina é gerado a partir desse código intermediário que contém instruções específicas de camada ISA e do Sistema Operacional. Assim sendo, o programa gerado somente poderá ser executado em um ambiente que possua compatibilidade com a camada ISA e Sistema Operacional da plataforma que o código binário foi gerado.

Em MVs de Linguagens de Alto Nível, o modelo descrito acima sofre algumas alterações. O processo de compilação é bastante parecido, porém o ponto em que o programa é distribuído ocorre em um nível mais alto. Nas Linguagens de Alto Nível, o código é gerado para a ISA de uma MV. Este código é então distribuído para execução em diferentes plataformas.

Para cada uma das plataformas deve ser implementada uma MV que é capaz de interpretar este código. Por fim, a MV transforma o código gerado para a camada ISA virtual em um código compatível com a plataforma alvo. A maior vantagem destes tipos de Linguagens é a portabilidade entre plataformas, a desvantagem é a criação de uma MV para cada plataforma existente e viabilizar a compilação e a execução do código.

2.3 Monitores De Máquinas Virtuais

Nesta seção, alguns dos MMVs mais difundidos serão apresentados, tanto nas corporações como nas Universidades. O principal objetivo desta seção é apresentar os MMVs que são usados como exemplos nas técnicas de virtualização que empregam. Atualmente, a virtualização atingiu grandes proporções e o desenvolvimento de MMVs tem sido feito nas mais diversas áreas da computação e cada vez mais os MMVs estão se tornando *softwares* especializados. Os MMVs abordados nesta seção são utilizados de forma mais genérica, entretanto é possível encontrar variações de suas implementações para objetivos mais específicos.

2.3.1. XEN

O Xen (Barham, 2003) é um MMV do tipo I e se situa na classificação em que o Hospedeiro e a MV Convidada possuem a mesma camada ISA. Este MMV é muito usado em pesquisas que tratam de virtualização, principalmente pelo fato de ter o código fonte aberto e pela ampla documentação encontrada na Internet e literatura.

A virtualização implementada pelo Xen é chamada de paravirtualização, pois a MV (na denominação do Xen, Domínio Convidado ou DomU) deve ter o *Kernel* alterado para que utilize algumas estruturas disponibilizadas pelo Xen. Estas estruturas são responsáveis pelo excelente desempenho do Xen em comparação a outras técnicas de virtualização. Durante o *boot* da máquina real, o MMV cria um Domínio chamado Domínio 0 (também conhecido como *Driver Domain* ou Dom0), este Dom0 além de possuir o privilégio de criar ou editar os DomU's, ele tem acesso ao Hardware por meio dos *Device Drivers* reais.

Além da paravirtualização, o Xen implementa a virtualização completa, porém para isso é necessário que o processador tenha suporte à virtualização, no caso da Intel o processador deve possuir a *flag* VMX e no caso da AMD, a *flag* SVM. A virtualização completa no Xen não possui o mesmo desempenho se comparado à

paravirtualização, pois alguns dispositivos, como interface de rede, são emulados, o que torna a MV bastante lenta. Em contra partida, com a virtualização completa é possível instalar outros Sistemas Operacionais, como o Windows, o que não ocorre quando se opta por usar a paravirtualização.

De forma simplificada, a paravirtualização é assim chamada, pois a MV tem consciência de que está sendo executada em ambiente virtualizado. Isto só é possível devido a alterações realizadas no *Kernel* do Sistema Convidado (DomU). Estas alterações forçam o DomU a utilizar estruturas de comunicação entre o DomU e o MMV.

As estruturas de comunicação estão divididas em duas partes:

- *I/O Rings* (ou *I/O Channel*);
- *Device Drivers* (*Back End* e *Front End*);

O *I/O Ring* é uma estrutura de dados circular, situada em uma região de memória compartilhada. Nesta estrutura existem dois ponteiros, um ponteiro que representa o consumo dos dados e outro que representa o produtor dos dados. Para que esta estrutura seja mais eficiente, o conteúdo da memória é uma referência para uma posição de memória que contém o dado. Com esta técnica, não há necessidade de realizar excessivas cópias de dados.

Figura 6 – Estrutura do I/O Ring (Barham, 2003).

Os *Device Drivers* fazem uso desta estrutura de *I/O Ring* para transmitir ou receber dados dos Domínios Convidados. Em cada um dos Domínios Convidados é instalado o *Front End* do *Device Driver*, que irá consumir ou produzir dados nos *I/O Rings*. A parte do *Device Driver* que fica no Dom0 é chamada de *Back End*, pois essa parte do *Device Driver* é responsável por passar os dados ao *Device Driver* real. Na figura 7 é mostrado como as estruturas do *I/O Ring* e os *Device Drivers* paravirtualizados interagem entre si.

Figura 7 – Interação entre o I/O Ring e Device Driver.

O custo de realizar estas alterações em um SO é relativamente baixo. Em média, a quantidade de linhas de código alterado para portar o Linux para o Xen é de 2995 linhas (algo em torno de 1,36% do total do *Kernel*) (Barham, 2003). Além das estruturas mostradas anteriormente, o Xen ainda implementa *Hypercalls* que têm a mesma função de uma *System Call* implementada em um SO comum, mas realiza solicitações ao MMV (ou *Hypervisor*).

2.3.2. VMWare

O VMWare (VMWare, 2000) é um software de virtualização bastante popular nas corporações. Basicamente são disponibilizadas duas versões do VMWare, o ESX Server e a versão para *desktop*. O ESX Server é bastante comum em corporações que optam por virtualizar o ambiente de TI. A outra versão do VMWare é para usuários finais, que por alguma razão necessitam usar outros Sistemas Operacionais, a versão para uso em desktop pode ser adquirida do site do VMWare, porém com algumas limitações de uso.

O VMWare é um MMV bastante maduro e conceituado e é por esta razão que se tornou comum nas corporações. Como visto anteriormente, a virtualização não é um conceito novo, porém o VMWare foi o primeiro software a virtualizar computadores baseados na arquitetura IA-32 com relativo sucesso. De forma simplificada, virtualizar computadores baseados na arquitetura IA-32 não é uma tarefa simples, pois não foi projetada com este objetivo. Os principais problemas enfrentados pela VMWare ao virtualizar essa arquitetura foram:

- A arquitetura IA-32 cresceu como um processador barato que a princípio jamais seria usado em grandes e complexos sistemas computacionais, além disso, esta arquitetura possui características difíceis de se virtualizar;
- É uma arquitetura aberta, isto é, há uma proliferação de diversos modelos e configurações da IA-32. A situação fica ainda pior com a proliferação de diferentes dispositivos de I/O disponíveis para os usuários no mercado;
- A VMWare é uma empresa separada de outras grandes corporações que fabricam *hardware* (a pouco tempo foi comprada pela EMC), como a IBM e empresas que possuem Sistemas Operacionais largamente utilizados, como a Microsoft. Por esta razão, o entendimento do funcionamento, tanto do hardware como do Sistema Operacional tornou-se um desafio para a VMWare.

A técnica que o VMWare implementa é conhecida como Tradução Binária (Altman, 2001). Esta técnica tem se mostrado bastante eficiente e desde a sua implementação, nas primeiras versões do VMWare, vem sendo otimizada não somente

na camada de software; mas algumas empresas de processadores, como a Intel, cogitam ter que suportar esta técnica em futuras versões do Xeon.

2.3.3. *Kernel Virtual Machine (KVM)*

O KVM (Kivity, 2007) é um MMV nativo do Linux, a proposta do KVM é tornar o Linux um MMV, isto é, sem a necessidade de instalar ou recompilar um MMV. Adicionando esta característica de virtualização no próprio *Kernel* do Linux, cada MV é tratada como um processo comum, isto é, as MVs beneficiam-se dos algoritmos de gerenciamento do próprio Linux, como por exemplo, o escalonador e gerenciamento de memória.

Em um ambiente normal do Linux, um processo pode ser executado em dois modos, no modo *Kernel* ou no modo Usuário. O KVM adiciona um terceiro modo, conhecido como modo Convidado (modo *Guest*). O modo Convidado possui seu próprio modo *Kernel* e modo Usuário para executar os processos.

A estrutura do KVM divide-se em duas partes:

- *Device Driver*: Usado para gerenciar a virtualização do *Hardware*. Este *driver* cria entradas no `/dev/kvm` que podem ser usados pelas MVs;
- Componente para o espaço de usuário para emular o *Hardware* do Computador. Este componente é o Qemu com algumas alterações.

Na figura 8 é apresentada a proposta do KVM em tornar o próprio Linux em um MMV:

Figura 8 – *Kernel* Do Linux como MMV – Proposta do KVM (Kivity, 2007).

2.3.4. PowerVM (IBM SystemP Virtualization)

PowerVM (Hales, 2008) é uma tecnologia de virtualização desenvolvida pela IBM e integra a virtualização tanto na camada de *Hardware* como no *Software*. Com esta integração de Hardware e Software, a IBM conseguiu atingir um grau elevado de flexibilidade. Com esta tecnologia, é possível dividir o processador em 0.1 (10%) para uma MV (ou na nomenclatura da IBM, *Logical Partition* – LPAR). Desde os processadores Power5 (até a presente data, Power6), a virtualização é suportada nativamente. O AIX, que é instalado nas LPARs, também possui suporte à virtualização, implementando de forma nativa as *Workload Partitions* (ou WPAR) (Blanchard, 2007).

Figura 9 – Arquitetura do PowerVM com LPAR e Micro-Particionamento (Hales, 2008).

A base do PowerVM é o *Power Hypervisor*, um MMV implementado entre o Sistema Operacional hospedeiro e o Hardware. O Power Hypervisor é, na verdade, um firmware e por meio dele são implementadas as técnicas de micro-particionamento, processador virtual, etc. As funções deste MMV são (Hales, 2008):

- Garantir a integridade, provendo uma camada de segurança entre as LPARs;
- Prover uma camada de abstração entre os recursos de Hardware e as LPARs que utilizam esses recursos. Controla como os processadores reais serão mapeados para processadores virtuais e durante as trocas de contexto ele é responsável por salvar e restaurar o estado dos processadores;
- Controle das interrupções de hardware e o gerenciamento das partições;

A comunicação entre o Sistema Operacional hospedeiro e o *Hypervisor* é realizada por meio de *hcalls* (ou *Hypervisor Calls*). Com essas *hcalls*, é possível obter estatísticas do *Hypervisor* e do hardware subjacente. Na figura 9 há uma ilustração da arquitetura da virtualização oferecida pela IBM nas pSeries.

Apesar do Linux ser suportado por essa técnica de virtualização da IBM, nem todas as características suportadas pelo AIX são implementadas no Linux. Outro problema enfrentado quando o Linux é usado, é o suporte a uma quantidade superior a 8 processadores. Após essa quantidade de processadores, o Linux não os utiliza de forma satisfatória.

3 GERÊNCIA DE RECURSOS EM MÁQUINAS VIRTUAIS

A melhor utilização dos recursos computacionais é, sem dúvida, um dos grandes atrativos da virtualização e também um dos principais fatores que levaram ao ressurgimento dessa tecnologia. Por conta da importância dessa característica, a gerência de recursos é um assunto de grande relevância nos laboratórios de pesquisa das empresas e universidades. Existem muitos trabalhos com diversas propostas de otimizações de gerência de recursos, alguns deles apenas portados dos Sistemas Operacionais e outros elaborados de forma específica para ambientes virtualizados.

Neste capítulo serão abordados como os principais recursos são virtualizados pelos MMVs mais tradicionais (estado da arte). Na última seção serão apresentados alguns trabalhos interessantes no assunto de gerenciamento de recursos. Alguns deles, (como no caso do KSM (Arcangeli, 2009)) estão prestes a ser suportados oficialmente.

3.1 Gerência do Processador

O processador é um dos recursos mais críticos de um ambiente computacional e na virtualização isso não é diferente. Entretanto, a virtualização do processador é consideravelmente mais elaborada quando comparado com outros recursos. Isso se deve principalmente ao fato dos algoritmos de escalonamento de processadores presentes em Sistemas Operacionais comuns serem facilmente portáveis para um MMV.

Em um MMV, o algoritmo de escalonamento pode ter a mesma classificação que os algoritmos de um Sistema Operacional. Os algoritmos podem ser:

- *Work-conserving* (WC-mode) ou *Non Work-conserving* (NWC-mode): Utilizando o algoritmo WC-mode, o processador fica ocioso se, e somente se, não houver nenhum processo para ser executado. Na prática, isso significa que se dois processos têm a mesma prioridade e um deles está bloqueado, o

outro processo poderá utilizar todo o processador. Entretanto, em algoritmos *NWC-mode*, cada processo utilizará somente a parte de processamento que lhe foi concedida.

- **Preemptivo ou Não-Preemptivo:** Um algoritmo preemptivo será re-executado sempre que um novo processo ficar pronto para utilizar o processador. Assim, se o novo processo possuir uma prioridade maior que o processo em execução, este será escolhido para usar o processador antes do término do *time slice* atual. Em contrapartida, um algoritmo não preemptivo permitirá que os processos em execução terminem de utilizar seu *time slice*.

Nas definições acima, um processo pode ser entendido como uma MV. Em geral, os algoritmos implementados são *WC-mode* e Não-Preemptivos. O Xen possui uma grande flexibilidade para a implementação desses algoritmos. Na atual versão é possível escolher um (dentre dois algoritmos disponíveis) e a adição de um novo algoritmo pode ser realizado de forma relativamente simples.

Os escalonadores no Xen podem ser selecionados por meio de um parâmetro no momento de *boot*. Por padrão, o escalonador atual no Xen é o *Credit Scheduler*². O outro algoritmo disponível é o *Simple Earliest Deadline First* (SEDF) (Leslie, 1996). Nas versões anteriores do Xen, existiram outros escalonadores, como o *Borrow Virtual Time* (BVT), *Atropos* e *Round Robin*.

O SEDF utiliza uma tupla (s_i, p_i, x_i) , na qual um determinado Domínio (Dom_i) identifica as suas necessidades. Juntos o *slice* s_i e o *período* p_i , representam a quantidade de tempo que o Dom_i receberá para utilizar o processador. De forma simplificada, o Dom_i irá receber pelo menos s_i unidades de tempo em cada período de tamanho p_i . O valor de x_i indica se o Dom_i poderá ou não utilizar por mais tempo o processador do que lhe foi concedido.

O algoritmo padrão do Xen, *Credit Scheduler*, atribui um peso (*weight*) e um valor de Cap para cada Domínio. O Cap identifica se o domínio poderá ou não exceder o tempo de processamento que lhe foi concedido. Se o valor de Cap for igual a zero, então ele poderá utilizar por mais tempo o processador. O peso de um Domínio está

² Credit Scheduler: <http://wiki.xensource.com/xenwiki/CreditScheduler>

ligado à quantidade de vezes que esse irá utilizar o processador. Em um ambiente concorrente, um Domínio com peso de valor 512 irá utilizar duas vezes mais o processador que um Domínio cujo peso é 256.

Cada um dos processadores reais é responsável pelo gerenciamento de uma fila de VCPUs. A prioridade de uma VCPU pode ser alta (*over*) ou baixa (*under*), que identificará se a VCPU excedeu ou não o tempo disponibilizado para utilização do processador. Quando uma VCPU é inserida na *runqueue*, essa é colocada atrás de todas as outras VCPUs que possuem a mesma prioridade.

Conforme a VCPU é executada, ela consome créditos. Ao término de cada execução, o algoritmo refaz o cálculo de quantos créditos cada Domínio ganhou ou gastou. Se a quantidade de créditos for negativa, implica em uma prioridade baixa (*under*), até que a VCPU consuma seus créditos, essa é considerada de prioridade alta (*over*).

Por padrão, o *Credit Scheduler* disponibiliza 30ms de *time slice*. Como este algoritmo não é preemptivo, cada Domínio utilizará até o final o tempo disponibilizado de 30ms. Uma característica bastante interessante desse escalonador é o balanceamento de carga entre as *runqueues* de um ambiente SMP. Se em uma *runqueue* não existir nenhuma VCPU pronta para execução, o algoritmo irá verificar a situação em outras *runqueues* em busca de VCPUs prontas.

O algoritmo de escalonamento implementado no VMWare ESX é parecido com o algoritmo implementado no Unix. Nele, os processos são analisados de acordo com a sua prioridade, que é obtida por diversos fatores. O processo que será escolhido para utilizar o processador é o que possui a maior prioridade e caso o processo em execução possua prioridade menor, este será retirado para que o novo processo possa utilizar o processador.

No VMWare ESX é implementado um algoritmo de escalonamento baseado em proporcionalidade de compartilhamento (VMWare, 2009) (em inglês, *Proportional-Share Based Algorithm*). Este algoritmo possui uma estrutura semelhante ao implementado no Unix. Entretanto, a métrica utilizada como determinante para organizar a fila de execução é chamada de *entitlement* (em português, direito). O *entitlement* é calculado

de acordo com algumas informações que são provenientes do usuário, como compartilhamento, reservas e limites. Quando uma MV está sendo executada, ela consome o *entitlement*. Se na fila de execução estiver alguma MV que consumiu menos do que teria direito (consumiu menos *entitlement*) então esta MV é considerada de maior prioridade.

A principal diferença entre os escalonadores do Unix e do VMWare é, basicamente, como as prioridades são determinadas. No VMWare a prioridade é calculada de forma dinâmica e apesar do usuário poder alterar alguns parâmetros (como no caso do Unix, alterar o valor *nice* de um determinado processo); o que determinará a verdadeira prioridade é como a MV consumirá o processador, que depende de diversos fatores, como o comportamento da carga de trabalho e da carga sobre o sistema.

No KVM, o uso do processador pelas MVs ficará a critério do escalonador utilizado. Neste ambiente, a MV é tratada como um processo tradicional e deve concorrer com outros processos pela utilização do processador. Este comportamento é muito observado em MMV do tipo II, em que a MV é apenas mais um processo dentro do Sistema Operacional, como é o caso do VMWare player, Virtual Box entre outros.

A gerência de processador no PowerVM da IBM é bastante avançada quando comparado com os outros MMVs. Neste MMV, é possível criar um ambiente muito flexível, com MVs possuindo processadores dedicados e outras MVs dividindo grupos de processadores (*processor pools*) em um mesmo ambiente físico. Uma característica interessante é que uma MV pode possuir o mínimo de 0,10 (ou 10%) de um processador até o máximo de todos os processadores disponíveis. Esta capacidade de poder dividir um processador é chamada de micro-particionamento (*micro-partitioning*) pela IBM.

Ao instalar uma MV (ou LPAR) o usuário deve definir alguns parâmetros que serão usados pelo MMV no momento de mapear e escalonar um processador para uma dada MV. Estes parâmetros são:

- Unidades de Processamento Mínima, Desejável e Máxima: Define a quantidade de unidades de processamento que uma MV poderá variar

enquanto ativa;

- Modo de compartilhamento, que pode ser *capped* ou *uncapped*, isto é, se a MV poderá utilizar mais o processador caso haja disponibilidade;
- Quantidade mínima, desejável e máxima de processadores virtuais, isto é, a quantidade de processadores que serão apresentados para o Sistema Operacional;

Nota-se que a unidade de processamento é informada em notação decimal e a quantidade de processadores virtuais deve ser um número inteiro. Esses números possuem uma relação de proporcionalidade. Por exemplo, ao assinalar 0,5 (50% de um processador) unidades de processamento a uma MV, deve-se alocar pelo menos um processador virtual. Entretanto, caso seja alocado 2,4 unidades de processamento para uma MV, deve-se alocar, no mínimo, 3 processadores virtuais, pois essa MV utilizará 100% de dois processadores (somando 200%) e 40% de um terceiro processador (ao todo, 240%). A figura 10 apresenta uma ilustração de como essa relação funciona.

Figura 10 – Relação entre Unidade de Processamento e Processador Virtual (Hales, 2008).

3.2 Gerência da Memória

Diferente do que ocorre com a gerência do processador, os algoritmos de gerenciamento de memória de Sistemas Operacionais convencionais não são facilmente portáveis para um MMV. Isto se deve ao fato de que um Sistema Operacional sempre tem como premissa de que possui toda a memória física disponível para ser gerenciada. Os objetivos dos algoritmos de gerenciamento de memória de um Sistema Operacional são diferentes dos algoritmos de um MMV.

Em geral, algoritmos como o LRU (Piantola, 2008) tem como principal objetivo minimizar a quantidade de faltas de páginas, um MMV deve disponibilizar uma parte da memória física para uma MV e garantir a integridade dessa região de memória disponibilizada. Outra função do MMV, não implementada de forma eficiente ainda, é a distribuição eficaz de memória entre as MVs, isto é, fazer a distribuição da memória da mesma forma como ocorre com o processador, garantindo que uma MV receba apenas o que é necessário.

Essa deficiência está ligada a incapacidade dos Sistemas Operacionais, de modo geral, não serem capazes de identificar novas quantidades de memória. Algumas técnicas foram criadas para contornar o problema e garantir que a memória não utilizada pela MV seja devolvida para o MMV. Outra técnica implementada torna possível a alocação de quantidades de memória para as MVs de tal modo que a soma dessa alocação seja superior à quantidade física disponível.

A primeira técnica, desenvolvida pela VMWare e implementada no ESX Server é conhecida como *Balloon Driver* (Waldspurger, 2002). Basicamente, o *Balloon Driver* é um mecanismo usado para forçar uma MV a devolver memória sem utilização do MMV. Por conta da difícil comunicação entre o MMV e a MV, o problema reside principalmente em saber quais páginas de memória podem ser liberadas e usadas por outras MVs.

A proposta do *Balloon Driver* é conseguir identificar quais páginas de memória podem ser usadas pelo MMV sem realizar alterações no SO convidado. De forma simplificada, o *Balloon* é acionado quando há necessidade de devolver uma determinada quantidade de memória para o MMV. Existem várias razões para que o

MMV solicite memória das MVs, a mais comum é a necessidade de alocar memória para outra MV.

Quando o *Balloon* for acionado, ele pressionará a memória da MV, esta “pressão” faz com que os algoritmos do SO selecionem quais páginas de memória podem ser enviadas para disco (*swapping*). Essas páginas, na verdade, são devolvidas para o MMV. Quando a necessidade do MMV for satisfeita, o *Balloon* irá parar a pressão na memória da MV fazendo que seus algoritmos parem de paginar memória. A figura 11 mostra de forma simplificada o funcionamento do *Balloon Driver*.

Figura 11 – Funcionamento do Balloon Driver (Waldspurger, 2002).

Apesar de ser implementado no Xen, o *Balloon* não tem o mesmo comportamento que a implementação do ESX Server. No Xen, as MVs não tem capacidade de realizar o “*self-balloon*”, isto é, o *Balloon* somente pode ser ativado do Dom0. Outra diferença é que o *Balloon* implementado nos DomU's não são usados pelo MMV, isto é, não há nenhuma situação em que o MMV irá inflar ou esvaziar o *Balloon*. Em geral, o *Balloon* é acionado nos DomU's por alguma intervenção do Administrador, a única situação em que o MMV aciona o *Balloon* é na criação de uma nova MV, neste

caso, o *Balloon* é inflado no Dom0 para coletar e alocar uma determinada quantidade de memória para a nova MV.

No VMWare, é implementada também uma técnica conhecida como *over commit* (Waldspurger, 2002) de memória. Essa técnica consiste em alocar memória para as MVs de tal forma que a soma das memórias alocadas para todas as MVs seja superior à quantidade de memória física disponível. Esta característica do VMWare foi alcançada por meio de uma técnica de compartilhamento de páginas de memória.

As páginas de memória das MVs que possuem o mesmo conteúdo são compartilhadas. Desta forma, o VMWare consegue diminuir consideravelmente a utilização de memória – Figura 12. Para verificar o conteúdo de uma página, o VMWare gera um *hash* e compara com o *hash* das páginas de outras MVs e as que possuem o mesmo *hash* são compartilhadas entre as MVs.

Figura 12 – Técnica de Compartilhamento de Memória Baseada em Conteúdo (Waldspurger, 2002).

A forma como a memória é compartilhada no PowerVM não é disponibilizada pela IBM. Entretanto, a princípio não há implementações do *Balloon Driver* nem do *Over Commit* de memória. Aparentemente, a memória é alocada de forma estática e persistente até que a LPAR seja terminada. No KVM, a memória usada pela MV é

alocada da mesma forma que um processo comum. Entretanto, ao disponibilizar a memória para a MV, o KVM utiliza uma tabela de páginas virtual na qual cria uma abstração para MV e dá a impressão que a memória alocada é contígua.

3.3 Gerência de Dispositivos de I/O

Em geral, os dispositivos de I/O são considerados o maior gargalo quando se trata de virtualização. Isso se deve ao fato de que esses dispositivos são lentos por natureza e muitas vezes emulados nos ambientes virtuais, o que acentua a característica de lentidão.

A paravirtualização, implementada pelo Xen, é uma das poucas técnicas de virtualização que não utilizam a emulação de dispositivos de I/O. A forma como o Xen virtualiza esse tipo de recurso foi apresentado na seção 2.3.1. Por conta das estruturas internas no Xen, a paravirtualização consegue obter ótimo desempenho, chegando muito próximo a uma máquina real.

O VMWare solucionou em parte o problema da emulação dos dispositivos. Em um ambiente que utiliza essa técnica para cada dispositivo existente no ambiente real, deve haver um dispositivo equivalente emulado. Essa solução foi brevemente substituída, pois além da enorme quantidade de dispositivos emulados a serem implementados, a emulação tornou o desempenho destes dispositivos ainda mais lento.

A solução encontrada pela VMWare para este problema foi utilizar o próprio Sistema Operacional anfitrião. Cada uma das MVs possui o *driver* do dispositivo sem modificação alguma, porém, quando a MV for usá-lo para, por exemplo, realizar uma operação de escrita, o VMWare gera uma *trap* e esta solicitação de escrita é enviada ao Sistema Operacional que fará uso do *driver* de dispositivo real. Com isso, as operações de I/O são executadas diretamente pelo Sistema Anfitrião, diminuindo de forma considerável o gargalo causado pelos dispositivos de I/O.

No KVM, quando uma MV executa uma instrução de I/O, o *Kernel* da Máquina Hospedeira realiza uma *trap* nessa instrução e alterna para o modo usuário. A instrução

é então executada e ao término o *Kernel* é alternado para a MV voltar a execução de onde havia parado anteriormente.

De forma geral, o modo como as operações de I/O são executadas no KVM são menos complexas em comparação com outros MMVs, como visto anteriormente. Entretanto, ao utilizar o Qemu para emular alguns *Drivers*, as operações de I/O acabam ficando lentas, mesmo com uma complexidade menor.

A virtualização realizada pelo *Hypervisor* da IBM para dispositivos de I/O é feito por uma MV especializada. A IBM denominou essa MV de *Virtual I/O Server* (Hales, 2008) e essa MV é responsável por disponibilizar para as outras MVs discos e interfaces de rede. Os principais objetivos da IBM ao criar esta MV especializada é melhorar o desempenho de dispositivos de I/O e centralizar o gerenciamento. O *Virtual I/O Server* é um AIX modificado e, exceto algumas configurações de storage e rede, são permitidas poucas customizações nessa MV. Um problema dessa implementação é o ponto de falha único (SPOF – *Single Point Of Failure*) que o *Virtual I/O Server* representa para o ambiente. Entretanto, é possível configurar dois ou três VMs como *Virtual I/O Server* e evitar que esse tipo de situação ocorra. Na figura 13, o funcionamento do *Virtual I/O Server* é apresentado.

Figura 13 – Funcionamento do Virtual I/O Server (Hales, 2008).

3.4 Trabalhos Relacionados

As MVs vêm sendo alvo de diversas pesquisas devido ao seu grande potencial em melhorar a utilização de recursos e gerenciamento de ambientes computacionais. Estes fatores tornam o uso de MVs bastante atrativo para utilizar em clusters dos mais variados tipos e em outros ambientes que com o ressurgimento da virtualização se tornou mais simples.

O primeiro trabalho abordado trata-se do gerenciamento de memória no Xen, cuja proposta é utilizar diferentes técnicas para implementar o *overcommit* de memória. O *Difference Engine* (DE) (Gupta, 2008) primeiramente tenta compartilhar as 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 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 (Piantola, 2008)).

Figura 14 – Funcionamento do Virtual I/O Server (Hales, 2008).

Com este trabalho, os autores conseguem um ganho de memória em torno de 65% em *workloads* heterogêneos. O ganho de memória da solução proposta pelos autores varia principalmente em função do *workload* das MVs e também da diversidade dos Sistemas Operacionais instalados. Os maiores problemas encontrados pelos autores neste sistema residem na complexidade de implementação e no *overhead* de processamento gerado pelo sistema.

A Realocação de Recursos no Xen (RRX) (Rodrigues, 2009) é uma proposta que tem como objetivo realocar recursos dinamicamente em MVs gerenciada 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. No artigo publicado no WSO de 2008, a proposta inicial é complementada com um mecanismo de alocação dinâmica dos recursos. A forma como a alocação dos recursos é feita não é detalhada no artigo (por exemplo, se existe aumento da quantidade de processadores virtuais ou se a prioridade da MV com problema de desempenho é incrementada).

O aspecto interessante dessa proposta é a utilização de técnicas de *Data Mining* e a decomposição de SLA's (*Service Level Agreement*) para decidir qual é a melhor configuração para as MVs. O processo de *Data Mining* é executado em um conjunto de dados gerado por uma bateria de testes e tem como objetivo verificar se é possível melhorar o desempenho de uma MV que já está em execução. Os autores verificam também o custo da reconfiguração do ambiente; caso o custo seja elevado, as alterações não são efetivadas.

Os problemas dessa proposta residem na complexidade de todo o processo, apesar de conseguir unir diversos conceitos da computação, essa complexidade pode ser um impeditivo. Outra questão em aberto nesta solução é a forma como é realizada as alocações nos ambientes virtuais e se existe a necessidade de alterações no código do Xen para suportar a proposta.

Figura 15– Processo de Realocação de Recursos no Xen (Rodrigues, 2009).

Um trabalho que vem sendo realizado no KVM e deve ser disponibilizado na árvore do *Kernel* do Linux é o KSM (*Kernel Shared Memory* ou *Kernel Samepage Merging*) (Arcangeli, 2009). O principal objetivo desse trabalho é aumentar a densidade da memória no Linux. O KSM irá fundir (*merge*) páginas de memória com o mesmo conteúdo e com isso gera uma quantidade maior de memória livre. A memória livre disponibilizada pelo KSM pode ser usada tanto para criar novas MVs como para ser alocada por outros processos.

O KSM está disponível a partir do *Kernel*³ 2.6.31 e para versões mais antigas do *Kernel* 2.6 é possível aplicar um *patch* para utilizá-lo. Para implementar o KSM, os autores utilizaram duas árvores binárias *red-black* (*red-black tree*) (Cormen, 2009), uma das árvores é chamada de árvore estável e deve estar sempre balanceada. Nessa árvore são mantidas as páginas que foram fundidas com outras páginas de forma satisfatória; e a outra árvore, chamada de árvore instável, contém páginas de memória que tem grandes chances de serem fundidas com outras páginas e caso isso ocorra, essa página entra na árvore estável.

³ Kernel: <http://kernel.org/>

4 MEMORY DISPATCHER

Neste capítulo é apresentado o *Memory Dispatcher* (MD) que foi desenvolvido com o objetivo de melhorar a distribuição de memória entre as MVs instaladas em um hospedeiro. Serão apresentadas algumas pesquisas preliminares ao desenvolvimento do MD, as ferramentas utilizadas para a implementação do MD, o algoritmo desenvolvido para tomar as decisões de alocação de memória e sua arquitetura.

4.1 Descrição da Proposta

O principal objetivo do MD é conciliar o desempenho de um ambiente virtualizado sem o desperdício de recursos computacionais. A maioria dos ambientes virtualizados não é capaz de prover um ambiente com elasticidade, isto é, um ambiente que seja possível alocar os recursos de forma automática de acordo com as necessidades do sistema.

Figura 16– (a) Provisão para Picos (b) e (c) Provisão Média (Armbrust, 2010)

Existem hoje duas situações em que um ambiente virtual pode ser provisionado. A primeira situação é quando os recursos computacionais são alocados para uma MV com o objetivo de suprir a demanda em momentos de pico de utilização, nessa situação quando o ambiente não está sob alta demanda, observa-se desperdício de recursos (Figura 16a). A outra situação é quando os recursos são alocados para suprir uma necessidade média de demanda (Figura 16a e 16b). Nessa situação, o ambiente poderá sofrer sérios problemas na ocorrência de um pico de utilização do ambiente, pois os recursos alocados não poderão suprir as necessidades da MV.

Nesse contexto, o MD tem como principal foco tornar o ambiente virtualizado mais flexível. Em linhas gerais, o MD realiza um monitoramento do ambiente virtual e de acordo com determinados critérios aumenta a alocação de memória do ambiente. A alocação do recurso é feita de acordo com a necessidade do ambiente e levando em conta as prioridades alocadas de cada MV instalada.

Utilizando o MD, uma MV com pouca demanda de recurso disponibilizará uma determinada quantidade de memória ao MMV (a quantidade de memória a ser disponibilizada é calculada de acordo com a prioridade da MV no Credit Scheduler, MVs com maior prioridade doam menos memória ou mais memória, caso contrário). Essa memória disponibilizada pelas MVs com menos demanda são, então, alocadas para MVs com maior necessidade. É importante ressaltar que esse mecanismo utiliza as estruturas já disponibilizadas pelo Xen, não havendo necessidade de re-compilação ou de re-instalação do produto.

4.2 Pesquisas Preliminares

Para que fosse possível implementar o MD, foram realizadas algumas pesquisas com o objetivo de identificar algumas formas de configuração e como alterá-las de tal forma que os recursos possam ser alocados dinamicamente na MV de acordo com as necessidades. Para as pesquisas foi utilizado o Xen como MMV, entretanto a maioria dos MMVs possui alguma forma de configurar as prioridades (ou alguma outra

característica análoga) das MVs em execução.

O primeiro parâmetro analisado foi o valor das prioridades nos escalonadores de processador do Xen. A análise foi realizada no *Credit Scheduler* e mostrou como alterações na prioridade da MV influenciam o desempenho nas operações de I/O. O segundo parâmetro analisado foi a memória. No entanto, diferente do que foi feito com a prioridade, essa pesquisa tinha como principal objetivo verificar alguns critérios de alocação de memória com um protótipo do MD desenvolvido em Perl. Apesar da possibilidade de ajuste dos dois parâmetros em uma MV serem viáveis sem a necessidade de reconfigurar ou reiniciar a MV, ambos ficam sujeitos a intervenção do administrador. Portanto, em uma situação normal os valores serão constantes desde a criação da MV até o seu término.

4.2.1. Efeito da Prioridade no Desempenho

Quando uma determinada MV, utilizando o escalonador *Credit Scheduler* é configurada com maior prioridade que outras MVs instaladas no mesmo hospedeiro, ela terá mais acesso a ciclos de processador que as outras. Portanto, maior prioridade implica em maior poder computacional. Outros escalonadores, como o SEDF, permitem que o *time slice* da MV com maior prioridade também seja maior.

Ao utilizar a tecnologia da virtualização, um dos principais pontos de contenção de desempenho é o subsistema de I/O. Por esta razão, foi realizada uma pesquisa (Baruchi, 2008) com o objetivo de quantificar o efeito da prioridade de uma MV em operações de I/O (comunicação de rede).

Com o objetivo de analisar o uso do processador pelas MVs durante os testes, foi utilizado o Xenmon (Cherkasova, 2005). Essa ferramenta possibilitou acompanhar o uso do processador ante às alterações de prioridades efetuadas no decorrer dos testes. Como *benchmark* para a realização das operações de I/O foi escolhido o Netperf⁴. Uma

---

⁴ Netperf: www.netperf.org

das características mais interessantes do Netperf é a possibilidade de alterar algumas configurações da comunicação de rede (como tamanho do *buffer* de sistema e aplicação).

Além da variação das configurações de prioridade nas MVs, foi feita uma análise do efeito da prioridade no Domínio 0 (Dom0) do Xen. O Dom0 desempenha um papel de extrema importância na arquitetura do Xen. Além de possuir privilégios de criar, destruir e efetuar comandos administrativos nas MVs, o Dom0 tem acesso direto ao *Hardware* subjacente.

Após as execuções dos testes, foi verificado que na maioria das configurações os resultados se mantiveram próximos da configuração padrão (todas as MVs com prioridade igual e sem alteração nos *buffers* de aplicação e sistema). Entretanto, em casos pontuais, principalmente quando a configuração do MTU (*Maximum Transmission Unit*) é alterada, observou-se que a vazão foi melhor quando o Dom0 estava com maior prioridade, isto é, utilizou com mais frequência o processador. Nas situações em que o MTU foi configurado com valores menores a vazão foi melhor quando a MV possuía maior prioridade. Este experimento demonstra que quando há maior quantidade de operações de I/O o ideal é aumentar a prioridade do Dom0. Já em aplicações que demandam mais processamento, é interessante aumentar a prioridade da MV. As figuras 17 e 18 mostram a eficiência do ambiente com a alteração da prioridade da MV ou do Dom0. A eficiência é calculada tendo como base a configuração padrão (um ambiente onde as prioridades são iguais tanto para o Dom0 como para as MVs). Uma eficiência de +15% significa que naquela configuração o ambiente foi 15% superior quando comparado com a configuração padrão.

Figura 17– Desempenho do Netperf com o Dom0 configurado com prioridade 25% maior que as MV's (Baruchi, 2008).

Figura 18– Desempenho do Netperf com a MV configurada com prioridade 25% maior que o Dom0 (Baruchi, 2008).

4.2.2. Efeito da Alocação de Memória no Desempenho

A quantidade de memória física disponível em um ambiente computacional tem grande influência no desempenho de todo o sistema. Desta forma, a pesquisa feita não tinha como objetivo analisar como a quantidade de memória influencia no desempenho e sim como decidir se há necessidade de alocação de memória e a quantidade de memória a ser alocada no ambiente para suprir às necessidades (Baruchi, 2009).

Para tal análise, foi implementado um protótipo do MD em Perl. Esse protótipo utilizava a carga do processador como referência para identificar tendências de uso de memória. De acordo com o uso do processador, o protótipo classificava as MVs em cinco níveis de tendência de uso de memória:

Nível 1: Tendência fraca de decremento de uso de memória;

Nível 2: Tendência forte de decremento de uso de memória;

Nível 3: Estável;

Nível 4: Tendência fraca de incremento de uso de memória;

Nível 5: Tendência forte de incremento de uso de memória;

De acordo com o nível da tendência, o protótipo calcula uma quantidade de memória a ser alocada ou removida da MV. A quantidade era obtida em múltiplos de 128MB (chamado de *memory slice*). No caso da MV estar em alguma tendência forte (Nível 1 ou 5) a quantidade de memória poderia ser cinco vezes o *memory slice*.

Por meio de testes realizados com o protótipo, foi possível verificar que a alocação de memória deve ser realizada com cautela, pois a adição ou remoção constante de memória na MV pode causar grande perda de desempenho no ambiente. Foi verificado também que os critérios de alocação devem ser mais variados, pois existe a possibilidade de se beneficiar uma determinada aplicação e onerar outras. O uso de memória livre como parâmetro também se mostrou fraco, pois o Linux utiliza a memória física como *cache* de operações de I/O para melhorar o desempenho. Por fim, durante essa pesquisa, surgiu a idéia do cálculo de memória ser feito na MV e não no Dom0. Com isso, o *daemon* que fica em execução no Dom0 tem como principal objetivo adicionar ou remover a memória solicitada pela MV e gerenciar as MVs que estão sob o

controle do MD.

Observando a figura 19 é possível verificar como a alocação de memória pode influenciar no desempenho do sistema. Com o aumento gradual de 1GB de memória no ambiente, o desempenho melhora de forma proporcional. Ao utilizar o protótipo do MD (indicado por Proposta no gráfico) o desempenho fica entre a configuração de 2GB e 3GB.

Figura 19– Desempenho do ambiente ao alterar a configuração de memória (Baruchi, 2009).

4.3 Arquitetura do *Memory Dispatcher*

O MD utiliza uma arquitetura análoga à arquitetura cliente-servidor (Figura 20). Na MV foi implementado um processo (*daemon*) que coleta as informações da MV e realiza o cálculo da quantidade de memória estimada para suprir as necessidades da MV. Esta informação é então passada ao Dom0 que verifica a possibilidade de alocar a quantidade de memória para a MV.

Durante as implementações iniciais, o cálculo da memória era realizado pelo Dom0, por meio de informações passadas pela MV. Entretanto, essa abordagem apresentou dois problemas:

- Excesso de dados a serem escritos/lidos na região de memória compartilhada;
- Sobrecarga de processamento para o Dom0; com duas MVs, o processamento a sobrecarga é aceitável, mas para um número maior de MVs pode tornar-se inviável;

A parte do MD que fica em execução na MV, ao iniciar, indica por uma *flag* no Xenstore de que aquela MV está sob gerenciamento do MD. Quando o processo termina, esta *flag* é removida do Xenstore e o MD retira esta MV das suas estruturas internas de gerenciamento. Com esta característica, é possível que em um mesmo ambiente físico haja MVs gerenciadas e não gerenciadas pelo MD (por exemplo, pode ser necessário que uma MV possua uma quantidade fixa de memória).

Figura 20– Arquitetura do *Memory Dispatcher*.

4.4 Algoritmos

O MD na parte cliente coleta diversas informações a respeito do uso de memória e verificar a quantidade de memória que, teoricamente, aumentaria ou poderia manter seu desempenho satisfatório. Da mesma forma que um cálculo é feito para a solicitação de memória, é feito um cálculo inverso para a retirada de memória. Desse modo, a parte do MD em execução na MV deve informar dois parâmetros para o Domínio 0, o primeiro, um valor absoluto, representando a quantidade de memória (em *Kilo Bytes* - KB) e outro parâmetro informando se a quantidade de memória deve ser retirada ou alocada.

Figura 21– Algoritmo do MD. (a) Parte executada na MV. (b) Parte executada no Dom0.

Com essas informações, a parte do MD que fica em execução no Dom0 deve então verificar se a quantidade de memória disponível pode ser alocada e efetiva a alocação (no caso de remoção, não são realizadas verificações). A parte do Dom0 apenas monitora a região de memória compartilhada e quando há alguma solicitação nova, uma interrupção virtual é gerada (Chisnall, 2007) e o MD toma as devidas providências. Já a parte da MV possui uma complexidade maior, pois deve realizar as verificações de necessidade de memória.

Na figura 21, é apresentado um diagrama simplificado dos passos executados pelo MD, tanto na parte da MV (21a) como na parte do Dom0 (21b). Nesse diagrama são mostrados os principais passos efetuados pelo MD e pelo diagrama é possível identificar como as duas partes interagem entre si.

Existem dois parâmetros que são ajustáveis (ambos *hard coded*, em futuras implementações pode-se criar algum parâmetro ao iniciar o *daemon* ou mesmo algum arquivo de configuração), os parâmetros para definir quando há necessidade de adicionar memória e a quantidade mínima que uma MV pode ter alocado (para evitar que toda a memória seja removida e causar danos ao sistema). De acordo com a variação desses parâmetros, o MD pode ficar mais ou menos sensível o tentar identificar problemas de escassez de memória.

A memória mínima é alocada proporcionalmente à quantidade de memória física disponível e a prioridade da MV. No Xen, o usuário pode definir a prioridade de execução de uma MV alterando os créditos do *Credit Scheduler* (vide seção 3.1). Quando um usuário altera esse valor, é desejável que esta MV tenha prioridade maior que as outras, pois pode estar processando algo importante para o usuário e por essa razão, a memória mínima possível será maior. É importante observar que quanto maior o valor mínimo que uma MV pode ter, menos memória ela poderá doar caso esteja ociosa. A variação desse parâmetro reside na proporcionalidade que ele possui em relação à prioridade da MV.

Outra característica é a verificação da necessidade de memória não ser baseada na quantidade de memória livre e sim na quantidade de faltas de páginas e na quantidade de tempo gasto pelo processador em espera por I/O (*I/O wait*). Esta decisão

foi tomada, pois no Linux a memória livre é usada como *buffer* de operações de I/O. Por isso, nem sempre pouca memória livre é indício de baixo desempenho. Outro fator para essa decisão foi a portabilidade do MD para outros Sistemas Operacionais, assim como no Linux, a falta de memória não implica em perda de desempenho; essa situação pode ocorrer com outros Sistemas Operacionais, já a ocorrência de falta de páginas é um parâmetro mais genérico e presente na maioria dos Sistemas Operacionais.

O cálculo utilizado para saber a quantidade de memória desejável também é baseado na taxa de faltas de página por segundo. Dependendo da taxa, é aplicado um fator de multiplicação (altas taxas de falta de página podem indicar situação *trashing* (Santos, 2009)). Entretanto, para evitar alocações desnecessárias, o MD implementa uma contagem de carência; isto é, somente fará a solicitação de memória caso ocorram três medidas consecutivas de falta de páginas, isso é importante para evitar alocação de memória, principalmente em situações de *cold start* (Easton, 1978) (quando um processo novo começa a executar, a taxa de falta de páginas aumenta pois, obviamente, não há nenhuma página desse novo processo em memória).

Para definir o momento de retirar memória, o MD verifica a ociosidade do Sistema Operacional. Se a quantidade de faltas de páginas estiver baixa por algum tempo e for identificado ociosidade, o MD retirará memória de forma proporcional (se a memória alocada para a MV for superior à 60% da memória total do sistema, então a memória é retirada a taxas de 10% da memória alocada e quando atingir 50% da memória total, a remoção ocorre a taxas de 5%).

4.5 Implementação

O MD foi desenvolvido em linguagem C e para a coleta de informações do Sistema Operacional instalado nas MVs e realizar a comunicação entre MV e MMV foi necessário o uso de algumas ferramentas e estruturas disponibilizadas pelo Xen. O Xen disponibiliza uma API para programadores desenvolverem aplicações e ferramentas. Além da API, o Xen é um software de código aberto, o que permite a realização de

estudos e alteração em seu código fonte.

Para a coleta de algumas informações e execução de comandos no MMV, foram utilizadas duas bibliotecas, a Libgtop⁵ e Libvirt⁶. A libgtop é um conjunto de bibliotecas que disponibiliza informações do Sistema Operacional coletando-as no `/proc` (Cesati, 2005) do Linux. Esta biblioteca fornece informações como quantidade de falta de páginas, carga do processador (*load average*) e outras informações detalhadas do uso de memória (como memória em *cache*, *buffer*, quantidade de páginas livres, etc).

A biblioteca Libvirt é uma iniciativa (*software open source*) que tem como principal objetivo padronizar as interfaces (API) dos MMVs, de tal forma que seja possível portar um determinado programa para outros MMVs sem drásticas alterações no código fonte. Essa biblioteca provê uma quantidade razoável de instruções para gerenciamento das MVs (como criação e destruição da MV, ajuste de memória e processador, etc). Entretanto, como o objetivo é padronizar, algumas características que o Xen disponibiliza a Libvirt não suporta (como a criação de uma região de memória compartilhada, criação de *I/O rings*, etc).

Para a comunicação entre a MV e o Dom0 foi utilizado o Xenstore (Chisnall, 2007). O Xenstore é um sistema de armazenamento compartilhado entre os Domínios Convidados e o Dom0. As informações são armazenadas no Xenstore de forma hierárquica e mantidas pelo Dom0. O Dom0 consegue acessar as informações pertinentes a qualquer Domínio, porém os Domínios Convidados conseguem acessar somente as informações delas mesmas.

A composição do Xenstore é feita de diretórios, que podem conter outros diretórios ou chaves. Cada uma das chaves contém um valor associado. Em geral, o Xenstore é usado para transmitir pequenas quantidades de dados entre o Dom0 e um Domínio Convidado. Uma característica do Xenstore é a monitoração de algumas chaves, isto é, se uma determinada chave tiver seu valor alterado, ele irá gerar uma interrupção. Esta interrupção pode ter alguma rotina atrelada e ser executada sempre que necessário.

---

⁵ Libgtop: <http://library.gnome.org/devel/libgtop/stable/>

⁶ Libvirt: <http://libvirt.org/>

O Xen permite que novas chaves ou diretórios sejam criados dentro da estrutura do Xenstore. Para isso, são disponibilizados comandos bastante intuitivos no espaço de usuário e também rotinas que podem ser usadas em espaço do *Kernel*. Os mecanismos de monitoração são implementados somente em espaço do *Kernel*. Por meio dessa estrutura, o MD transmite informações da MV para o Dom0 que deverá alocar a quantidade de memória solicitada pela MV.

5 RESULTADOS

Para a avaliação da solução de gerenciamento de memória proposta foram executados alguns *benchmarks* para verificar sua viabilidade. Os testes foram executados em dois ambientes distintos (*Hardware* e *Software*) controlado e diversas vezes para se obter uma quantidade de massa de dados suficiente para análise e conclusões.

Além da avaliação da proposta, os testes possuem o objetivo de verificar se os critérios discutidos anteriormente estão atuando de forma coerente. Para esta verificação, o MD foi implementado para registrar o estado do ambiente no momento da alocação e outras informações que serão úteis para a análise. Com estas informações foram gerados gráficos que ajudarão a observar o comportamento das simulações.

Foram escolhidos três *benchmarks* com comportamentos diferentes para avaliar o MD em situações de diversidade. O primeiro *benchmark* utilizado foi o Dbench⁷, que simula um servidor Samba. No teste seguinte foi utilizada a compilação do *Kernel* do Linux (*Kernel*, 2009). Por fim, no ultimo caso de teste (os testes de carga de trabalho heterogênea), foi utilizado o *httpperf*⁸, um *benchmark* que simula acessos a um servidor Web.

5.1 Metodologia e Configuração

Para a execução dos *benchmarks* foram criados *scripts* que iniciaram os testes. Esses *scripts* repetiram as execuções por um total de 15 vezes. Tanto no ambiente sob gerência do MD como em um ambiente virtual convencional os testes foram executados de forma idêntica. A tabela 1 apresenta os *benchmarks* e as suas características.

Os *benchmarks* foram executados em dois ambientes diferentes. Ambos os ambientes diferem tanto na configuração de *Hardware* como na configuração de *Software*. Essa variação foi importante para analisar a diferença de desempenho do MD em

---

⁷ DBench: <http://freshmeat.net/projects/dbench/>

⁸ HTTPERF: <http://www.hpl.hp.com/research/linux/httpperf/docs.php>

configurações distintas. Para haver uma distinção entre as configurações os ambientes serão nomeados como configuração I e configuração II, conforme apresentado na Tabela 2.

Tabela 1– Benchmarks executados e seus comportamentos

<i>Benchmark</i>	<i>Medida</i>	<i>Unidade de Medida</i>	<i>Comportamento</i>
Dbench	Vazão	MB/Seg	I/O Bound
Comp. <i>Kernel</i>	Tempo Compilação	Segundos	CPU Bound
HTTP Perf	Tempo de Resposta	Resposta/Seg	I/O e CPU Bound

O benchmark DBench tem uma característica de I/O intensivo pois simula um servidor de arquivos Samba. Esse benchmark permite que a quantidade de clientes (ou usuários) acessando o servidor seja configurado. Para que houvesse um aumento gradativo na carga de trabalho o DBench foi executado simulando o acesso de 50, 75, 100 e 150 clientes simultâneos. Os dois primeiros valores (50 e 75) podem ser considerados uma carga de trabalho pequena e os dois últimos (100 e 150) uma carga de trabalho mais alta. A compilação do *Kernel* não possui parâmetros a serem configurados, foi utilizado um arquivo de configuração (.config) padrão da distribuição Fedora. Por fim, no HTTPPerf é possível ajustar a quantidade de acessos simultâneos ao servidor Web e para os testes executados neste trabalho, foram configurados 200 clientes acessando simultaneamente o servidor Web.

Para que houvesse um parâmetro de comparação, foram executados os mesmos testes em um ambiente virtualizado sem o gerenciamento do MD. Nesse cenário, cada um dos testes foi executado com as MVs possuindo uma determinada quantidade de memória configurada fixa. Na configuração I, foram instaladas duas MVs (idênticas) e na configuração II, foram instaladas três MVs (idênticas). Na tabela 3 é possível verificar como as MVs em cada uma das configurações foram instaladas.

Tabela 2– Configuração das Máquinas Reais

Configuração	Hardware		Software	
Config. I	Processador (<i>Vendor</i> / # Core)	<i>AMD Athlon</i> / 2	Sistema Operacional	<i>Fedora release 8</i>
	Memória (GB)	5	Versão <i>Kernel</i>	2.6.21.7-5.fc8xen
	Disco (GB)	250	Versão Xen	xen-3.1.2-5.fc8
	Rede (# Interfaces)	2	Versão libgtop	libgtop2-devel-2.20.2-1
			Versão Libvirt	libvirt-devel-0.4.4-2
Config. II	Processador (<i>Vendor</i> / # Core)	Intel / 8 (com HT)	Sistema Operacional	<i>openSUSE 11.1</i>
	Memória (GB)	3	Versão <i>Kernel</i>	2.6.27.37-0.1-xen
	Disco (GB)	500	Versão Xen	xen-3.3.1
	Rede (# Interfaces)	2	Versão libgtop	libgtop-devel-2.24.0-1
			Versão Libvirt	libvirt-devel-0.4.6-11

Após a configuração das MVs no ambiente, foram executados os *benchmarks* conforme apresentado na tabela 4. Cada execução foi realizada de forma simultânea, por exemplo, no Caso de Teste A, as máquinas MV01 e MV02 da configuração I executaram o DBench (mesma carga de trabalho) ao mesmo tempo. O mesmo foi aplicado para a configuração II e suas MVs. A última execução foi realizada com cargas de trabalho distintas, conforme pode-se observar na coluna do Caso de Teste C da Tabela 4.

Tabela 3– Configuração das Máquinas Virtuais

Configuração	Hardware Alocado		Software Instalado	
Config. I	Processador (#VCPUs)	1	Sistema Operacional	Fedora release 8
	Memória (MB)	512, 768, 1024, 1536, 2048, 2560 e 2666	Versao do Kernel	2.6.21-2952.fc8xen
	Disco (GB)	7, 3		
	Rede (# Interfaces)	1		
Config. II	Processador (#VCPUs)	2	Sistema Operacional	openSUSE 11.1
	Memória (MB)	256, 512 e 741	Versao do Kernel	2.6.27.7-9-xen
	Disco (GB)	10		
	Rede (# Interfaces)	1		

Após a execução dos testes com as configurações da Tabela 3, com a quantidade de memória fixa, os testes foram implementados em MVs (com as mesmas configurações da Tabela 3, mas com a quantidade de memória variando por conta do MD). Dessa forma o total de execuções para a configuração I foi de 360 execuções e para a configuração II, total de 180 execuções.

Tabela 4– Configuração das Máquinas Virtuais

Configuração	Máquina Virtual	Caso de Teste A	Caso de Teste B	Caso de Teste C	Qtde Execuções
Config. I	MV 01	Dbench	Comp. Kernel	Comp. Kernel	360
	MV02	Dbench	Comp. Kernel	Dbench	
Config. II	MV01	Dbench	Comp. Kernel	Http Perf	180
	MV02	Dbench	Comp. Kernel	Comp. Kernel	
	MV03	Dbench	Comp. Kernel	Dbench	

Ao término das execuções foi calculado a média e a distribuição *student-t* (Pearson, 1939) dos resultados de desempenho. A distribuição *student-t* foi utilizada por conta da quantidade de dados coletados e foi calculado com precisão de 95%. Para a análise dos ganhos de memória, observados mais adiante, foi escolhido o teste que mais se aproximou da média e desse teste foram extraídos outras informações como a quantidade de memória alocada e usada e a taxa de falta de páginas.

5.2 Resultados Obtidos

Um grande desafio a ser transposto ao se tratar de gerência de recursos é lidar com o desempenho. O desafio reside em manter o desempenho ao máximo com o mínimo de recursos, entretanto essa situação costuma ser antagônica. O MD foi projetado para se aproximar dessa situação, manter um desempenho aceitável com uma economia de recursos (no caso, de memória). O termo “ganho de memória” será usado nas próximas seções para demonstrar a quantidade de memória que o ambiente deixou de utilizar ao estar sob gerenciamento do MD. O ganho de memória será calculado da seguinte forma (resultado em %):

$$\text{Ganho} = ((M_{\text{max}} - M_{\text{alocada}}) / M_{\text{max}}) * 100$$

Onde,

M_{max} = Quantidade de memória máxima configurável.

M_{alocada} = Média da quantidade de memória alocada pelo MD

5.2.1. Caso de Teste A

Nas figuras 22 e 23 são apresentados o desempenho médio das execuções nas MVs da configuração I. Em geral o desempenho do MD foi satisfatório, ficando com um desempenho acima da configuração de 1,5GB. Em alguns casos de teste o desempenho foi superior, como observado na Figura 22 com o DBench ajustado para 50 clientes. Nas tabelas 5 e 6 são apresentados os valores medidos com o intervalo de confiança *student-t*.

Figura 22– Gráfico com o desempenho do DBench – Configuração I / MV01.

O desempenho da MV02 foi parecido com o desempenho observado na MV01, ficando acima da configuração de 1,5GB. Em alguns testes o desempenho do MD foi até superior (pode-se observar pela variação nas tabelas 5 e 6) e essa oscilação se deve, principalmente, a momentos em que uma das duas MVs estava ociosa e os recursos não eram disputados. Apesar dos testes terem sido iniciados simultaneamente e com as mesmas configurações não havia nenhum mecanismo que evitasse o término de um teste primeiro em uma MV e depois na outra. Por essa razão durante um pequeno intervalo, uma das MVs podia ser beneficiada com a ociosidade da outra MV.

Figura 23– Gráfico com o desempenho do DBench – Configuração I / MV02.

Tabela 5 – Desempenho obtido com o DBench na Config.I / MV01

	50 Clientes		75 Clientes		100 Clientes		150 Clientes	
Memória (MB)	Média (MB/S)	Student-t (95%)	Média (MB/S)	Student-t (95%)	Média (MB/S)	Student-t (95%)	Média (MB/S)	Student-t (95%)
512	9,49	5,90	3,10	0,90	3,06	0,61	3,37	0,94
768	21,83	12,49	5,02	1,47	3,51	1,19	3,60	0,96
1024	69,78	24,78	5,74	4,67	3,97	1,78	3,59	1,09
1536	98,78	19,88	74,64	13,80	65,29	9,97	66,56	5,24
2048	93,44	11,13	56,99	11,53	61,01	13,62	62,49	7,71
2560	110,52	16,42	63,23	15,13	52,75	8,74	56,27	4,80
2666	87,21	6,33	49,34	17,50	58,14	8,61	56,87	5,94
MD	102,67	21,61	44,44	29,36	22,82	12,25	19,59	7,52

Tabela 6 – Desempenho obtido com o DBench na Config.I / MV02

Memória (MB)	50 Clientes		75 Clientes		100 Clientes		150 Clientes	
	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
512	3,78	1,15	2,91	0,46	5,06	1,36	2,32	0,50
768	36,00	21,87	4,95	1,77	3,95	0,85	3,11	0,76
1024	100,66	29,84	22,65	10,57	4,02	1,27	6,07	4,02
1536	126,22	20,23	73,48	20,93	67,85	13,97	66,21	6,94
2048	124,10	19,55	112,33	27,88	78,49	5,54	67,67	7,61
2560	107,91	13,51	65,77	20,03	60,94	8,26	73,94	5,66
2666	151,98	26,47	142,87	32,37	108,03	24,45	80,80	6,60
MD	121,39	27,78	58,16	29,69	38,76	16,81	17,24	9,04

O desempenho na Configuração II (Figuras 24, 25 e 26) sofreu variação maior quando comparado com a Configuração I. Nessa configuração, o desempenho do MD foi próximo da memória fixa de 256MB em quase todos os testes, apresentando pouca oscilação. A variação de desempenho para essa configuração ficou abaixo da configuração I, isso porque havia três MVs concorrendo pela memória. Nesse caso, a probabilidade de ocorrer um intervalo de tempo com pelo menos duas MVs ociosas é menor. Os resultados e a variação podem ser observados nas Tabelas 7, 8 e 9.

Figura 24– Gráfico com o desempenho do DBench – Configuração II / MV01.

Figura 25– Gráfico com o desempenho do DBench – Configuração II / MV02.

Figura 26– Gráfico com o desempenho do DBench – Configuração II / MV03.

Tabela 7– Desempenho obtido com o DBench na Config.II / MV01

	50 Clientes		75 Clientes		100 Clientes		150 Clientes	
Memoria (MB)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
256	4,27	0,75	4,02	0,98	3,71	1,08	3,22	0,68
512	19,34	8,53	6,72	1,62	7,19	2,38	5,58	2,99
741	35,92	16,74	12,09	2,98	7,78	1,67	5,96	1,22
MD	6,90	4,46	4,33	1,38	3,24	0,93	3,71	1,01

Tabela 8– Desempenho obtido com o DBench na Config.II / MV02

	50 Clientes		75 Clientes		100 Clientes		150 Clientes	
Memoria (MB)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
256	4,35	0,80	3,95	0,97	3,33	0,87	3,08	0,62
512	21,51	9,27	5,46	1,17	4,30	1,17	3,64	0,88
741	30,29	11,56	11,84	2,74	8,40	2,11	8,25	2,35
MD	3,54	1,43	3,66	1,26	2,49	0,41	2,90	0,65

Tabela 9– Desempenho obtido com o DBench na Config.II / MV03

	50 Clientes		75 Clientes		100 Clientes		150 Clientes	
Memoria (MB)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
256	4,46	0,98	4,21	0,99	3,44	0,97	3,87	1,04
512	7,31	1,25	5,39	1,02	4,16	0,99	3,39	0,96
741	110,82	38,94	33,35	12,14	10,43	2,69	9,06	2,16
MD	10,71	6,17	5,13	1,72	3,98	1,28	3,39	0,55

O ganho de memória alcançado pelo MD foi bastante significativo, chegando a 70% na configuração I, conforme observado na figura 27. Quando o DBench foi configurado com maior quantidade de clientes (100 e 150), o uso do *buffer* foi mais agressivo e o ganho de memória diminuiu. Na configuração II o ambiente foi mais estável, com o ganho de memória ficando por volta de 70% também para todos os cenários (Figura 28).

Figura 27– Ganho de Memória nas MVs – Configuração I.

Figura 28– Ganho de Memória nas MVs – Configuração II.

5.2.2. Caso de Teste B

O desempenho médio da compilação do *Kernel* do Linux utilizando as configurações descritas anteriormente pode ser observado nas figuras 29 e 30. Na configuração I, o

desempenho do MD foi parecido com o da configuração fixa de 1,5GB. Utilizando o MD, o desempenho foi 10% pior quando comparado com a memória fixa máxima (2,6GB) na MV01 e 5% pior para a MV02.

Na tabela 10 são apresentados os resultados e a distribuição *student-t* para este *benchmark*. A variação dos resultados na configuração I foi baixa, demonstrado estabilidade tanto para as configurações fixas de memória como para as configurações utilizando o MD como gerenciador de memória.

Figura 29– Desempenho da Compilação do *Kernel* – Configuração I.

O desempenho na configuração II, sofreu grandes variações no desempenho durante as execuções desse *benchmark* (Tabela 11). Na figura 30, o desempenho na configuração fixa com mais memória (741MB) foi pior quando comparada com o desempenho médio das configurações com menor quantidade de memória (256 e 512MB, respectivamente). Atribui-se esse resultado devido à escassez de memória causada no Dom0 quando alocado 741MB as três MVs. O baixo desempenho no Dom0 pode causar danos a todo o ambiente (Baruchi, 2008), pois ele é o responsável pelo acesso direto ao *Hardware*.

Utilizando o MD, esse problema é minimizado devido ao controle de alocação de

memória implementado que não permite que o Dom0 fique abaixo de uma quantidade de memória mínima. Entretanto, mesmo com esse controle, o desempenho foi pior em relação as configurações de 256 e 512MB, mas melhor quando comparado com 741MB.

Figura 30– Desempenho da Compilação do *Kernel* – Configuração II.

Tabela 10– Desempenho obtido com a compilação do *Kernel* na Config.II / MV01

Memória (MB)	MV01		MV02	
	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
512	1924,73	18,30	1847,73	14,13
768	1835,40	29,25	1770,47	8,50
1024	1809,27	17,80	1738,07	15,58
1536	1719,60	17,20	1656,20	18,40
2048	1654,07	5,87	1661,47	26,45
2560	1608,73	20,84	1573,47	22,81
2666	1538,73	43,31	1608,60	34,16
MD	1726,27	31,51	1692,67	22,98

Tabela 11– Desempenho obtido com a compilação do *Kernel* na Config.II / MV02.

	MV01		MV02		MV03	
Memória (MB)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
256	3345,93	36,16	3354,07	41,85	3353,07	40,24
512	3129,07	20,90	3118,73	21,29	3112,93	16,89
741	3702,53	263,84	3701,20	256,49	3698,67	255,59
<i>MD</i>	<i>3538,93</i>	<i>332,59</i>	<i>3486,33</i>	<i>388,42</i>	<i>3591,67</i>	<i>361,43</i>

O ganho de memória para esse *benchmark* (Figura 31) foi parecido com o ganho obtido no *benchmark* anterior. Na configuração I, o ganho ficou próximo de 70% e na configuração II observa-se uma oscilação maior no ganho de memória. Essa oscilação é resultado do controle de memória implementado que em determinados momentos não fez a alocação de memória devido ao problema de deixar o Dom0 com pouca memória.

Figura 31– Ganho de memória – Compilação do *Kernel*.

5.2.3. Caso de Teste C

Os testes realizados neste cenário diferem das outras execuções, pois neste cenário a carga de trabalho é diferente entre as MVs, isto é, em cada MV foi executado um *benchmark*. O objetivo dessa execução é verificar o comportamento do ambiente sob gerenciamento do MD com carga de trabalho heterogênea.

O desempenho executando a compilação do *Kernel* (Figura 32 e Tabela 13) foi próximo da execução com a MV configurada com 1,5GB de memória fixa. Para o segundo *benchmark*, com o Dbench (Figura 33 e tabela 12) o desempenho do MD situou-se entre as execuções fixas de 1,5 e 2GB. Com exceção para a primeira execução com 50 clientes que o desempenho foi consideravelmente melhor que as outras configurações fixas de memória. Neste caso, o melhor desempenho é atribuído ao comportamento do uso de memória do *benchmark* de compilação do *Kernel*. Nesse *benchmark*, a memória vai sendo usada aos poucos até chegar ao limite máximo. Portanto, no começo da execução desse *benchmark*, a MV não necessita de toda a memória e a disponibiliza para uso de outras MVs.

Figura 32– Desempenho da Compilação do *Kernel* – MV01 / Configuração I.

Figura 33– Desempenho do DBench – MV02 / Configuração I.

Tabela 12– Desempenho obtido com o DBench na configuração I / MV02.

	50 Clientes		75 Clientes		100 Clientes		150 Clientes	
Memoria (MB)	Média	Student-t (95%)	Média	Student-t (95%)	Média	Student-t (95%)	Média	Student-t (95%)
512	16,00	4,00	5,63	0,99	5,10	1,26	3,99	1,17
768	75,92	19,16	7,45	1,90	6,55	1,22	6,53	1,26
1024	136,04	22,74	40,04	18,53	13,36	10,02	7,76	1,64
1536	176,39	17,07	93,75	15,42	34,99	13,42	5,22	1,72
2048	195,22	8,84	111,27	16,55	117,29	26,69	52,85	9,26
MD	145,57	20,87	31,93	14,61	26,94	13,52	10,48	7,33

Tabela 13– Desempenho com a compilação do *Kernel* na configuração I / MV01.

Memoria (MB)	Média	Student-t (95%)
512	2717,73	600,49
768	2376,47	450,93
1024	2437,40	492,14
1536	2186,67	329,37
2048	1876,40	180,98
MD	2298,00	405,31

Na configuração II, com três MVs instaladas, o desempenho do MD pode ser observado nas figuras 34, 35, 36 e nas tabelas 14 e 15. Na execução do Httpperf o desempenho ficou abaixo das outras duas configurações com memória fixa (algo em torno de 33% abaixo). O desempenho na compilação do *Kernel* foi estável nas três execuções, com poucas variações (estatisticamente desprezível). Por fim, na figura 36, é apresentado o desempenho com o DBench. Nas execuções com poucos clientes (50 e 75) o desempenho foi abaixo ao compararmos com as configurações de memória fixa, neste cenário com poucos clientes, a memória RAM disponível para realização do *cache* de I/O consegue manter grande parte dos dados a serem escritos e como não há variação de memória, o desempenho é maior. Na comparação com uma quantidade maior de clientes (100 e 150) o desempenho ficou mais próximo das configurações com memória fixa, pois a quantidade de memória livre para a realização de *cache* diminui consideravelmente devido ao aumento de clientes concorrendo pelo *cache*.

Figura 34– Desempenho do Httpperf – MV01 / Configuração II.

Figura 35– Desempenho da Compilação do *Kernel* – MV02 / Configuração II.

Figura 36– Desempenho do Dbench – MV03 / Configuração II.

Tabela 14– Desempenho do HTTPERF e Comp do *Kernel* – Configuração II.

	HTTPERF (MV01)		Comp. <i>Kernel</i> (MV02)	
Memória (MB)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
512	9,33	1,31	2750,93	154,50
741	10,71	1,65	2720,73	149,50
MD	6,20	1,53	2768,13	221,32

Tabela 15– Desempenho Dbench – Configuração II / MV03.

	50 Clientes		75 Clientes		100 Clientes		150 Clientes	
Memória (MB)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)	Média	<i>Student-t</i> (95%)
512	89,77	25,09	15,49	1,82	14,75	0,54	14,81	2,05
741	170,44	31,12	61,96	11,08	21,12	3,40	17,83	2,01
MD	32,58	10,89	10,43	2,22	7,68	2,34	11,14	2,97

O ganho de memória (observados nas figuras 37 e 38) para o Httpperf foi pouco acima de 70% e da compilação do *Kernel*, pouco abaixo de 50%. Os ganhos de memória para o DBench foram ainda maiores, chegando a 80% (pouco acima do ganho observado no caso de teste A para essa configuração – 10% superior).

Figura 37– Ganho de memória – Configuração II – MV01 / MV02.

Figura 38– Ganho de memória – Configuração II – MV03.

5.3 Demanda de Memória

O ganho de memória, apresentado anteriormente, mostra a quantidade de memória que o ambiente deixou de usar em relação à alocação total possível. Essa metodologia é interessante para analisar a quantidade de memória que foi economizada no ambiente. Entretanto, para verificar se a alocação está sendo feita de forma coerente com as demandas da MV, foi necessário utilizar outra forma de mensurar a qualidade da alocação de memória realizada pelo MD.

A comparação a ser utilizada é conhecida como produto espaço-tempo (PET) (Denning, 1980). O objetivo dessa medida é comparar a quantidade de memória que a MV efetivamente utilizou e a quantidade de memória que a MV tem a sua disposição no decorrer da execução dos *benchmarks*. Através desse método, é possível verificar a quantidade de memória desperdiçada, pois uma diferença grande entre a memória que foi demandada (memória utilizada) e a memória disponível é um indicio de grandes

quantidades de memória sem uso. Caso contrário, se a diferença for pequena, há indícios de falta de memória.

Para o cálculo da demanda de memória ao utilizar o MD, as métricas de PET serão obtidas através da diferença entre a área do gráfico da memória alocada e a área do gráfico da memória utilizada. Portanto, a demanda de memória será a área A-B da figura 39. O comportamento ideal é uma diferença próximo de zero, isto é, nessa situação ideal a alocação de memória foi muito próxima da demanda do ambiente. A unidade de medida dessa métrica será MBxSegundo e a partir do resultado dessa métrica poderá ser aferido a qualidade da alocação de memória realizada pelo MD.

$$\text{Área}_{[\text{MB} \times \text{Segundo}]} = \int_{t_1}^{t_2} y(t) dt, \text{ onde:}$$

t_1 e t_2 : Intervalo de execução do Benchmark;

$y(t)$: Curva que representa a alocação ou demanda de memória;

Figura 39– Ilustração do cálculo da demanda de memória.

As figuras 40 e 41 representam o comportamento de uma alocação fixa de memória e a alocação utilizando o MD, respectivamente. Pode-se observar que na alocação fixa, o desperdício de memória é maior, pois a área do gráfico que representa

a diferença entre a quantidade de memória alocada e a quantidade de memória utilizada é maior que a mesma área representada na figura 38, ao utilizar o MD.

Figura 40– Alocação de Memória Fixa (2666MB).

Figura 41– Alocação de Memória utilizando o MD.

Nas figuras 42 e 43 são apresentados os gráficos da demanda de memória na compilação do *Kernel* (caso de teste B) para a configuração I e II. Na configuração I, observa-se que a relação entre memória demandada e memória alocada ao utilizar-se o MD ficou próximo da configuração fixa de 1024MB, entretanto, o desempenho apresentado (figura 26) para essa configuração foi próximo das configurações de memória fixa de 1,5 e 2GB.

Figura 42– Demanda de Memória – Compilação do *Kernel*.

No gráfico que representa a demanda de memória para a configuração II, observa-se que a relação entre memória alocada e memória usada teve uma variação considerável para a MV02. Essa alocação demasiada na MV02 foi causada pela não liberação da memória em tempo hábil entre um teste e outro. Quando essa situação ocorre o teste seguinte já começa com uma maior alocação. A liberação de memória no MD também leva em conta a ociosidade da MV, foi observado que eventualmente a MV pode demorar mais que o intervalo de tempo programado no *script* para voltar a ficar ociosa.

Figura 43– Demanda de Memória – Compilação do *Kernel*.

Apesar do resultado discrepante na MV02, as outras duas tiveram um comportamento semelhante. O desempenho para esse caso de teste (Figura 30) foi muito próximo do observado com a configuração fixa de memória máxima (741MB) e com uma alocação de memória mais próxima da real necessidade das MVs.

Para o caso de teste A, os gráficos que representam a demanda de memória podem ser verificados nas figuras 44 e 45. Com 50 clientes configurados nota-se que há quantidade excessiva de memória. Com as outras configurações, o uso da memória fica condizente com a memória alocada. Para quantidades maiores de clientes, como 100 e 150, a relação entre alocação e uso de memória com o MD é próximo da configuração máxima de memória fixa. A principal razão dessa alocação próxima é devido ao comportamento do DBench e o principal critério de alocação de memória do MD, que é a falta de páginas. Pelo fato do DBench ser um *benchmark* de I/O intensivo, a memória livre acaba sendo alocada para *buffer*. Com a diminuição da memória, há também, aumento de falta de páginas, que induz o MD a alocar memória, mesmo não melhorando de forma substancial o desempenho do benchmark.

Figura 44– Demanda de Memória – DBench.

Figura 45– Demanda de Memória – DBench.

Nas figuras 46, 47 e 49 observa-se a demanda de memória na configuração II do caso de teste A. Nessa configuração, o MD apresenta boa relação entre alocação e memória usada nas MV01 e MV02. Entretanto, observa-se uma anomalia na MV03, quando o DBench estava ajustado com 50 e 75 clientes. Atribui-se essa alteração há uma demora para retirar a memória da MV03. Quando o DBench está configurado para utilizar poucos usuários e está chegando ao final do teste, há um decréscimo no uso de memória, porém, lentamente. Nesse caso, apesar do MD usar como referencia a quantidade de falta de páginas para adicionar ou remover memória, a queda gradual no uso de memória não se refletiu na queda de falta de páginas, fazendo com que o MD não retirasse a memória.

Figura 46– Demanda de Memória – DBench.

Figura 47– Demanda de Memória – DBench.

Figura 48– Demanda de Memória – DBench.

5.4 Sobrecarga

Foi realizada uma análise de sobrecarga ao utilizar o gerenciamento de memória com o MD. Essa análise tem como principal objetivo quantificar a sobrecarga imposta tanto na MV como no Dom0 ao utilizar o MD. A análise de sobrecarga foi realizada executando-se a compilação do *Kernel* por 15x e extraindo a média dos resultados. Os testes foram realizados na MV e no Domínio 0. Para verificar a sobrecarga nas MVs, primeiramente foi executada a compilação do *Kernel* sem nenhum *daemon* do MD e com memória fixa, em seguida, o mesmo teste foi realizado, entretanto com o *daemon* do MD em execução (com o *daemon* do Dom0 parado) somente na MV, isto é, o *daemon* estava realizando os cálculos e a monitoração, entretanto nenhuma adição ou remoção de memória era efetivamente realizada. O mesmo procedimento foi realizado para o Domínio 0, mas com o *daemon* das MVs parados.

De acordo com o que foi mensurado a parte do MD referente a MV causou uma sobrecarga de 1%. Na parte que executa no Dom0, o desempenho foi afetado em 4%. A maior sobrecarga no Dom0 ocorre devido às interrupções virtuais geradas pelo Xenstore. As interrupções são geradas pela MV sempre que há alguma necessidade de adição ou remoção de memória e são mais constantes quando há um aumento na quantidade de MVs.

A sobrecarga na MV torna-se praticamente desprezível, pois na MV o MD realiza apenas cálculos para verificar a necessidade ou não de mais memória e eventualmente realizar a solicitação para o Dom0 de acordo com suas necessidades. Um fator determinante para a baixa sobrecarga é a implementação do MD em C. Em protótipos anteriores, a sobrecarga chegava a algo em torno de 15% no Dom0 quando implementado em Perl.

6 CONCLUSÕES

Este trabalho analisou as principais técnicas de gerenciamento de recursos utilizadas em ambientes virtualizados. Através desse estudo, foi possível identificar alguns pontos que a virtualização ainda não foi capaz de resolver por completo. Um dos pontos identificados foi a elasticidade do ambiente, isto é, a capacidade de adaptação da MV para uma determinada carga de trabalho. Após a realização desse estudo foi concebido o *Memory Dispatcher*, um mecanismo de monitoração que realoca os recursos de memória entre as MVs instaladas em um mesmo ambiente físico.

O desempenho do MD sofreu grandes variações de desempenho no decorrer dos testes. Alguns resultados foram satisfatórios enquanto que outros tiveram um resultado abaixo da configuração mínima de memória fixa. Os resultados menos satisfatórios podem ser explicados pelos seguintes fatores:

- **Balloon Driver:** Conforme descrito anteriormente, o balloon driver tem como objetivo forçar os algoritmos do próprio Sistema Operacional a buscar páginas que podem ser retiradas da memória. Portanto o constante acionamento desse mecanismo pode causar danos ao desempenho;
- **Desempenho no Dom0:** Apesar do MD evitar que toda a memória do Dom0 seja retirada, pode haver necessidade de melhorar esse mecanismo fazendo com que ele mantenha mais memória para o Dom0;
- **Concorrência:** Apesar do MD alocar uma quantidade de memória suficiente para satisfazer as necessidades imediatas da MV, no caso do MD alocar muita memória para uma MV e uma segunda MV necessitar de memória, esta terá que esperar até haver liberação de memória para conseguir o recurso. Este mecanismo, principalmente em um ambiente com uma grande carga de trabalho, pode onerar o desempenho de alguma MV;

Apesar das grandes variações no desempenho, ao utilizar o MD, o ganho de memória conquistado através desse mecanismo de gerenciamento de memória foi bastante significativo. Este ganho de memória abre a possibilidade de criação de outras

MVs melhorando a forma como os recursos são distribuídos dentro do ambiente virtualizado.

Alguns ajustes ainda precisam ser feitos no MD para que a alocação de memória se torne mais precisa. Conforme foi visto na análise de demanda de memória, o MD ainda pode sofrer com situações de falso-positivo, adicionando ou removendo memória em momentos ruins e com isso causando mais desperdício de memória ou retirando em demasiado. A sobrecarga causada no ambiente também é baixa, chegando a apenas 4% no Dom0. Ainda há espaço para otimizações, como a diminuição da quantidade de interrupções geradas na MV (que é a responsável pela sobrecarga observada no Dom0).

6.1 Contribuições

Este trabalho realizou um estudo sobre as principais técnicas de gerenciamento de recursos em ambientes virtuais. Foi mostrado como os principais MMVs realizam a gerência dos recursos entre as MVs instaladas e também alguns trabalhos em andamento (e alguns deles já disponíveis como o KSM) que mostraram como o assunto vem sendo abordado em alguns laboratórios de pesquisa.

O texto, escrito em português, também é uma contribuição. A maioria dos trabalhos abordados e livros sobre virtualização estão, em sua maioria, em inglês. O texto apresenta um conteúdo resumido, mas ao mesmo tempo com certo grau de detalhamento suficiente para ser usado como referência em futuros estudos sobre virtualização ou sobre gerenciamento de recursos computacionais.

Por fim, o trabalho apresentou o *Memory Dispatcher*, um mecanismo cujo objetivo é contribuir para um melhor balanceamento de recursos em um ambiente virtualizado. Além da contribuição no gerenciamento de memória o MD foi o primeiro passo para a implementação de um ambiente totalmente colaborativo, com o uso de mecanismos de comunicação entre Hóspede e Hospedeiro (MV e MMV) já implementados, o MD foi capaz de elevar o grau de interação de todo o ambiente.

6.2 Trabalhos Futuros

O presente trabalho mostrou como o gerenciamento de recursos em um ambiente virtualizado ocorre, além de apresentar algumas pesquisas relativas a esse estudo. Através desse trabalho foi possível identificar alguns pontos de exploração para trabalhos futuros, como:

- Comunicação entre MV e MMV: Existem poucos mecanismos implementados para que a MV se comunique com o MMV. Esta Em geral o MMV sabe apenas o consumo de CPU e a memória alocada para uma determinada MV. Para que haja um compartilhamento melhor dos recursos, é interessante que o MMV saiba mais detalhes da utilização dos recursos alocados a uma MV;
- Paradigmas de Sistemas Operacionais: A virtualização já é uma realidade, entretanto alguns Sistemas Operacionais de uso geral ainda não são flexíveis para serem usados dentro de um ambiente virtualizado. Por exemplo, os Sistemas Operacionais ainda são projetados tendo como premissa de que a memória identificada em sua inicialização será permanente até o término de sua execução, tornando inviável a alocação de memória dinâmica;
- Computação em Nuvem (*Cloud Computing*): Implementação do MD em um ambiente de computação em nuvem;

Sobre o MD, algumas melhoras podem ser feitas em seu algoritmo, como por exemplo:

- Predição: Apesar do MD identificar faltas de páginas, ele somente fará a alocação quando as faltas de páginas ocorrerem efetivamente e evita que futuras faltas de páginas aconteçam. O uso de um mecanismo de predição pode evitar que as faltas de páginas iniciais aconteçam o que melhoraria o desempenho do sistema;
- Implementação de Algoritmos de Controle: Alguns algoritmos utilizados em outras áreas (como controladores P.I.D – Proporcional Integral e Derivativo) podem ser implementados no MD de tal forma que melhore não só a identificação de falta de memória como também na alocação proporcional a

essa falta de memória identificada;

- Inteligência Artificial: Implementação de algoritmos de IA, como de lógicas paraconsistentes, para melhorar na identificação de necessidades da MV e no gerenciamento de recursos, não se limitando somente à alocação de memória, mas também a alocação de outros recursos em conjunto;
- Outros MMV's: A idéia apresentada, apesar de ter sido implementada apenas no Xen, também pode ser desenvolvida em outros MMVs. Alguns protótipos estão sendo desenvolvidos para o KVM, mas por conta da não existência de um mecanismo como o Balloon Driver, algumas modificações precisam ser feitas. O que mais facilita nestes dois MMV's é o fato de possuírem o código fonte aberto, que torna o trabalho de alteração mais simplificado;

REFERÊNCIAS

- ALTMAN, E. R.; et. al. **Advances and Future Challenges in Binary Translation and Optimization**. IEEE Proceedings, pp. 1710-1722. Novembro 2001.
- ARCANGELI, Andrea; EIDUS, Izik; WRIGHT, Chris. **Increasing Memory Density by Using KSM**. Proceedings of the 2009 Linux Symposium. 2009.
- ARMBRUST, Michael; et. al. **Above the Clouds: A Berkeley View of Cloud Computing**. Technical Report UCB/EECS-2009-28. EECS Department, University of California, Berkeley. Fevereiro 2010.
- BARHAM, P; et. al. **Xen and The Art of Virtualization**. Proceedings of the 19th ACM Symposium on Operating Systems Principles. pp. 164-177. Outubro 2003.
- BARUCHI, Artur; MIDORIKAWA, Edson M. **Influência do Algoritmo de Escalonamento Credit Scheduler no Desempenho de Rede**. V Workshop de Sistemas Operacionais. Belém - PA. Julho 2008.
- BARUCHI, Artur; MIDORIKAWA, Edson T. **Gerencia de Memória Adaptativa no Xen**. VI Workshop de Sistemas Operacionais. Bento Gonçalves, RS. Julho 2009.
- BELLARD, Fabrice. **QEMU, a Fast and Portable Dynamic Translator**. Proceedings of the USENIX Annual Technical Conference. 2005.
- BLANCHARD, Bruno; et. al. **Introduction to Workload Partition Management in IBM AIX Version 6.1**. 1a Edição. Redbook, IBM Corporation. Novembro 2007.
- BOWLES, Kenneth L. **Beginner's Guide for the UCSD Pascal System**. New York. Ed. McGraw-Hill. 1980.
- CESATI, Cesar; BOVET, Daniel P. **Understanding the Linux Kernel**. 3a Edição. Ed. O'Reilly Media. Novembro 2005.
- CHERKASOVA, L.; et. al. **Xenmon: QoS monitoring and performance profiling tool**. Technical Report HPL-2005-187, HP-Labs. 2005.
- CHISNALL, David. **The Definitive Guide to the Xen Hypervisor**. 1a Edição. Ed. Prentice Hall PTR. Novembro 2007.
- CORMEN, Thomas H.; et. al. **Introduction to Algorithms**. 3a Edição. Ed. The MIT Press. Setembro 2009.

DEHNERT, James C.; et. al. **The Transmeta Code Morphing Software: using speculation, recovery, and adaptative retranslation to address real-life challenges.** Proceedings of the International Symposium on Code Generation and Optimization. pp. 15-24. 2003.

DENNING, P. J. **Working Sets Past and Present.** IEEE Transactions on Software Engineering. pp. 64-84. Janeiro 1980.

EASTON, Malcolm C.; FAGIN, Ronald. **Cold-Start vs. Warm-start miss ratios.** Communications of the ACM. pp. 866-872. 1978.

ESTEIRE, Oscar Azanón; LOVELLE, Juan Manuel Cueva. **Set of tools for native code generation for Java Virtual Machine.** ACM SIGPLAN Notices, Volume 33, pp 73-79. Março 1998.

FRASER, K.; et. al. **Reconstructing I/O,** Technical Report UCAM-CL-TR-596, University of Cambridge. 2005.

GOLDBERG, R. P. **Survey of Virtual Machine Research.** Computer pp. 34–45, Junho 1974

GUPTA, Diwaker; et. al. **Difference Engine: Harnessing Memory Redundancy in Virtual Machines.** In 8th USENIX Symposium on Operating System Design and Implementation. Dezembro 2008.

HALES, Christopher; et. al. **PowerVM Virtualization on IBM System p: Introduction and Configuration.** 4a Edição. Redbook, IBM Corporation. Maio 2008.

KIVITY, Avi; et. al. **KVM: The Linux Virtual Machine Monitor,** Proceedings of The Linux Symposium, Ottawa, Ontario. 2007.

LAUREANO, Marcos. **Máquinas Virtuais e Emuladores: Conceitos, Técnicas e Aplicações.** 1a Edição, São Paulo, Ed. Novatec. Junho, 2006.

LESLIE, I. M.; et. al. **The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.** IEEE Journal of Selected Areas in Communications. 1996

MAGNUSSON, Peter; WERNER, B. **Efficient Memory Simulation in SimICS.** Proceedings of the 1995 Annual Simulation Symposium. pp. 62-73. Abril 1995.

PEARSON, E. S. **Student as a Statistian.** Biometrika, v. 30, pp. 210-250. 1939.

PIANTOLA, Ricardo; MIDORIKAWA, Edson T.; CASSETARI, Hugo H. **On Adaptative Replacement based on LRU with working area restriction algorithm.** ACM SIGOPS Operating System Review. pp. 81-92. 2008.

RODRIGUES, Elder M.; et. al. **Reconfiguração de Ambientes Virtualizados através do uso de teste baseado em modelos de SLAs**. VI Workshop de Sistemas Operacionais. Bento Gonçalves, RS. Julho 2009.

ROSENBLUM, M.; GARFINKEL, T. **Virtual Machine Monitors: Current Technology and Future Trends**. Computer, p. 39-47, Maio 2005.

ROSSI, F. D. **Alocação Dinamica de Recursos no Xen**. Dissertação de Mestrado. PUC Porto Alegre. 2005.

SANTOS, Douglas; MAZIERO, Carlos. **Avaliação do comportamento de sistemas operacionais de mercado em situação de trashing**. I Workshop de Sistemas Operacionais. Bento Gonçalves, RS. Julho 2009.

SMITH, James E.; NAIN, Ravi. **Virtual Machines: Versatile Platforms For Systems And Processes**. 1a Edição, Ed. Morgan Kaufmann, 2005.

VMWARE Virtual Platform Technical White Paper.
<http://ftp.hit.edu.cn/linux/Datas/Docs/webmirror/www.vmware.com/virtualplatform.html>

VMWARE. **VMWare vSphere4: The CPU Scheduler in VMWare ESX 4**. White Paper, Technical Marketing, SD. Outubro 2009.

WALDSPURGER, C. A. **Memory Resource Management in VMWare ESX server**. Proceeding of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002). Boston, MA. pp. 181-194. Dezembro 2002.

WHITAKER, A.; SHAW M.; GRIBBLE, S. D. **Denali: Lightweight Virtual Machines for Distributed and Networked Applications**. Technical Report 02-02-01, University of Washington. 2002.